PINE SCRIPT™ AUTHORS · RUNTIME INTEGRATORS

PYNE DOCUMENTATION

Complete PYNE manuals — language, runtime, LSP, API, DevOps

CONTENTS

01 PYNE Documentation

02 End User Hub

03 Installation

04 Quick Start

05 Configuration

06 CLI Guide

07 Library API

08 Evaluate Scripts

09 Editors & LSP

10 Pro API Usage

11 Troubleshooting

12 CLI Commands Reference

13 Runtime modes

14 Glossary

15 FAQ

16 Language Core

17 Grammar (ANTLR4)

18 ASDL Schema

19 AST Builder

20 Helper API

21 Unparser

22 Visitor & Transformer

23 Type System

24 Linter

25 Error Model

26 Runtime

27 Series and history

28 Expressions and statements

29 Builtins

30 Technical analysis (ta.*)

31 Strategy builtins

32 Collections

33 Drawing and plotting

34 request.* and input.*

35 Libraries

36 Strategy events

37 Alerts

38 Compiler overview

39 Numba path

40 Compile strategy broker

41 Interpret ↔ compile parity

42 Language Server

43 LSP Architecture

44 Diagnostics

45 Completion

46 Hover

47 Navigation

48 Formatting

49 Semantic Tokens

50 Inlay Hints

51 Builtin Metadata

52 VS Code Extension

53 Editor Clients

54 Pro API

55 App Lifecycle

56 Auth and Keys

57 POST /run

58 Hosted runner

59 POST /optimize

60 Preview Endpoints

61 Backtest Endpoint

62 Chart Renderer

63 Runtime Bridge

64 Evaluate Contract

65 DevOps

66 Local development

67 Continuous integration
68 Release
69 PyPI publish
70 Publish checklist
71 Docker
72 Nuitka build (LSP + CLI)
73 Metadata encryption
74 GCP & Cloud Run
75 Observability
76 Security
77 Ecosystem
78 Compatibility guarantee
79 Implementation status
80 Missing features
81 Pine v6 surface inventory
82 Numerical validation
83 Roadmap
84 Contributing
85 Changelog
86 PyneTS
87 Install PyneTS
88 PyneTS CLI
89 PyneTS runtime
90 PyneTS compile
91 PyneTS parity
92 pyne-worker
93 pyne-worker evaluate
94 pyne-worker deploy
95 pyne-agent-worker
96 pyne-agent-worker chat
97 pyne-agent-worker deploy

PYNE DOCUMENTATION

Open-source Pine Script™ evaluation — grammar, AST, interpret/compile runtime, alerts + webhooks, LSP, Pro API. pip install hoox-pyne · github.com/hoox-sh/pyne

PYNE DOCUMENTATION

PYNE is the open evaluation stack for TradingView® Pine Script™: a formal ANTLR4 grammar, an ASDL algebraic AST, a deterministic bar-loop evaluator (interpret / compile / `mode=auto`), a Language Server Protocol surface, a Flask Pro API, and edge workers that share one evaluate contract — including **alerts** and optional **L2 webhooks**.

For **natural-language authoring** (PYNE Agent), see the sister project **pyne-agent-worker** — standalone Cloudflare® Workers AI™ agent with optional AXIS plugin; does not require pyne-worker.

Pine Script™ and TradingView® are trademarks of TradingView, Inc. This project is independent and not affiliated with or endorsed by TradingView, Inc. Cloudflare® is a trademark of Cloudflare, Inc.

Abstract

Where closed hosts couple the language to a proprietary charting host, PYNE treats the language as an inspectable pipeline:

```
Source (.pyne / .pine)
  → ANTLR4 lexer/parser (resource grammar)
  → ASDL AST (builder)
  → bar-loop (interpret | compile | auto)
  → plots / fill / events / drawings / alerts
  → optional L2 webhooks (Pro API + edge)
```

Live on PyPI as **`hoox-pyne`** (`import pynescript`). Source: github.com/hoox-sh/pyne.

```
pip install "hoox-pyne[lsp]"
pyne --version
# aliases: pynescript / pynescript-lsp
```

The same pipeline powers the desk CLI, the LSP binary, the Pro API, browser Pyodide (via AXIS), and Cloudflare® workers. Round-trip fidelity (`parse → unparse`) and interpret↔compile plot parity are first-class invariants — see compatibility and implementation status. Local open-corpus set01–04 (2026-08-09): **parse 99.96%**, **Runtime interpret 100% excl. intentional demos** (set01 **249/249**). The repository does not ship third-party script corpora.

TypeScript consumers import **PyneTS** (`@hoox-sh/pynets`) — same public names; Python remains the oracle.

What's new

`hoox-pyne` 0.6.4 (2026-09-14) — convert any older Pine version to v6 from the editor, LSP code action, `pyne convert --to 6`, and `POST /lsp/convert`.

0.6.3 — optional Flask/VPS/container hosted runner (`PYNE_RUNNER`) plus CLI `pyne runner` (`deploy` / `tick` / `list`). Scripts keep evaluating after upload without AXIS holding the connection.

0.6.2 — interpret wraps subscripted UDF parameters (`s[1]`) per call site so `src = close; updown(src)` matches compile. Official TradingView® builtin scripts are **0 MISMATCH** vs compile.

0.6.1 — persistent `InterpretSession` (`Runtime.create_session` , delta `append_bars` , forming-bar `update_last_bar`). Dual-host `strategy.cash` qty-type, fill-time risk gates, session confirm-on-append, finalized-bucket HTF `lookahead_on` .

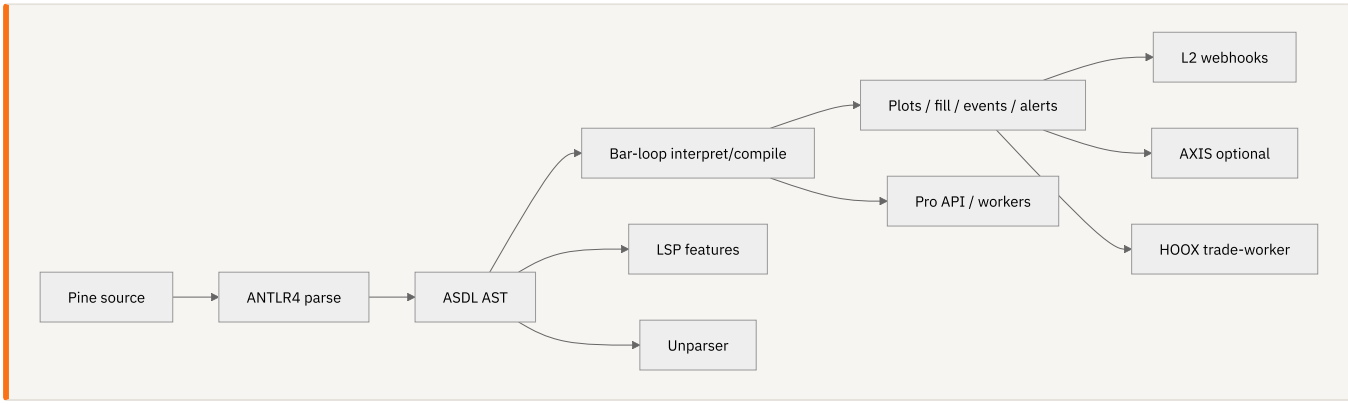
0.6.0 — type-aware intraday risk on both brokers, intrabar `var` rollback on realtime ticks, exchange-calendar `timeframe.change` D/W/M, `request.security` `gaps_on` / `lookahead_on` on HTF resample paths.

Full notes: GitHub releases · compatibility · evaluate scripts.

How to use

Surface	Role	Start here
Library + CLI	Parse, lint, unparse, evaluate. Library `Runtime.run` defaults to **interpret**; `pyne run` is **compile-only** smoke	End User hub · modes
Language core	Grammar, AST, visitors, type system	Core
Runtime	Series caps, builtins, strategy, alerts, compiler parity	Runtime
LSP	Diagnostics, completion, hover, format (`.pyne` / `.pine`)	LSP
Pro API	HTTP `/run` (alerts + L2 webhooks), `/optimize` , preview, backtest	API
Ops	CI, Docker, Nuitka, secrets, GCP	DevOps
PyneTS	TypeScript / Bun library (`parse` / `unparse` / `Runtime.run`)	PyneTS
AXIS (separate product)	Charting PWA AXIS	AXIS docs
HOOX	Edge trade mesh	HOOX docs

Conceptual model



Invariant: evaluation never requires a proprietary chart host. AXIS is an optional chart; HOOX is an optional *execution mesh*.

Tracks

End User

Install the package (`pip install hoox-pyne`), run the CLI, embed the library API, wire an editor (`.pyne` / `.pine`), or call the Pro API as a consumer — including alerts export and webhooks.

→ End User hub

Core / Runtime / LSP / API

Contributor and systems manuals: how the front-end, semantics, editor protocol, and HTTP bridge are built. Runtime covers series caps, incremental TA, drawing GC, `fill()` export, `request.security` HTF resample + barmerge (foreign/complex → `na`), strategy optimize, and parity.

DevOps

CI matrices, Fernet metadata encryption, Nuitka LSP binaries, containers, Cloud Build.

→ DevOps hub

Reference

Compatibility guarantees, missing surface, numerical validation, PyneTS, ecosystem map, roadmap.

Sister surfaces

- **PyneTS** — TypeScript / Bun library. Standalone hoox-sh/pynets / `@hoox-sh/pynets` . See PyneTS.
- **pyne-worker** (sister repo) — Python Cloudflare® Worker for production `POST /run` (alerts + cron + R2).
- **pyne-agent-worker** (sister repo) — NL authoring host (`POST /v1/chat`); optional validate via pyne-worker.
- **AXIS** — installable PWA; documented at hoox.sh/axis/docs.
- **Family map** — Ecosystem · modes.

Offline & agent exports

Auto-generated like HOOX manuals (`bun run docs:exports`):

Kind	Path
Full-corpus LLM pack	`llm.txt`
LLM site map (llmstxt.org)	`llms.txt`
Track PDFs (A4)	`/exports/pyne-*-manual.pdf`

See also

- Installation
- Quick start
- Runtime modes
- Alerts & webhooks
- Grammar pipeline

- Evaluate contract
- POST /optimize
- pyne-worker
- pyne-agent-worker
- Roadmap

02 · ENDUSER

END USER HUB

Install PYNE (pip install hoox-pyne), run CLI/library eval with mode=auto, wire editors (.pyne/.pine), and call Pro API alerts + webhooks.

END USER HUB

This track is for people who **consume** PYNE: parse and reformat Pine Script™, lint before upload, evaluate expressions or full scripts against OHLCV (`mode=auto` / interpret / compile), export **alerts** (optional L2 webhooks), wire an editor via LSP (`.pyne` / `.pine`), or call the HTTP evaluate contract. Contributor internals (grammar regeneration, AST builder, bar-loop semantics) live under Core, Runtime, and DevOps.

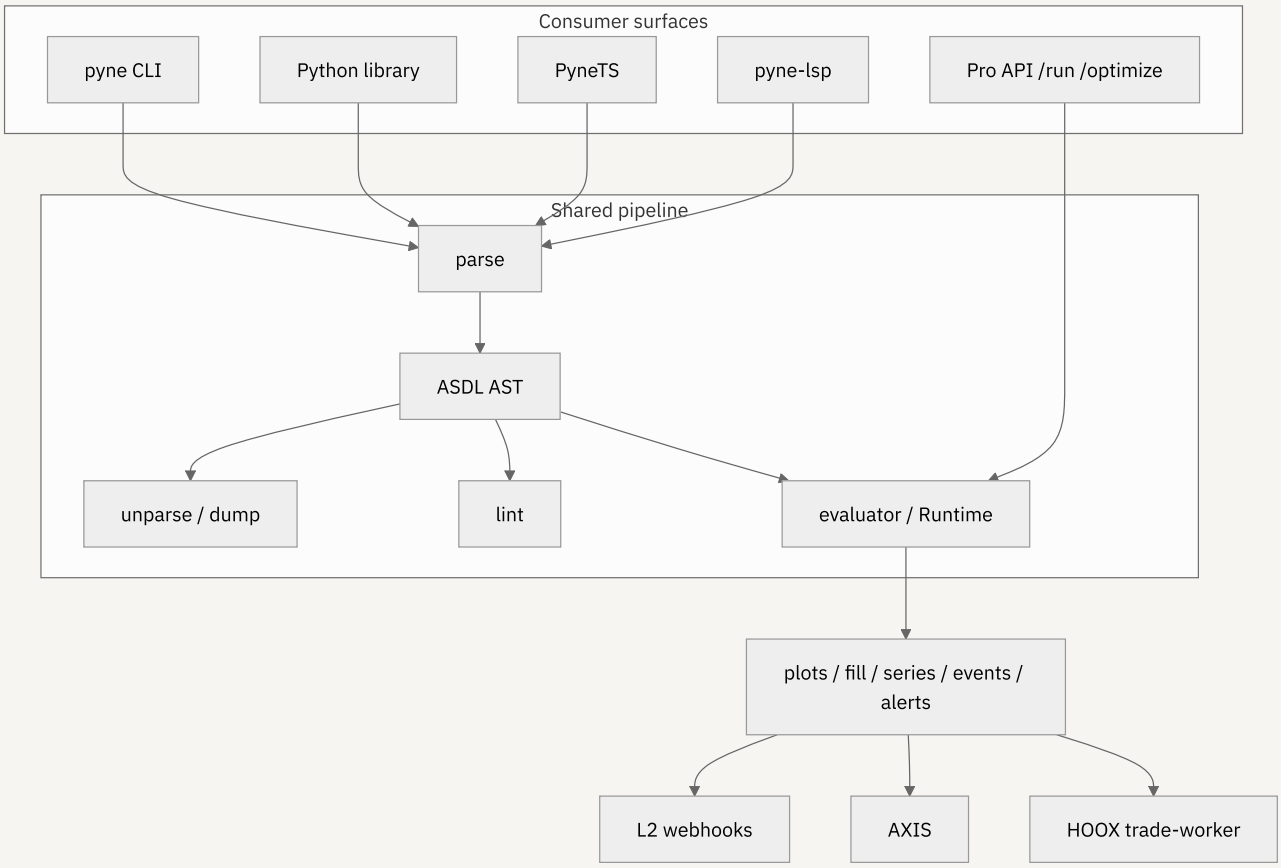
Abstract

PYNE exposes one language pipeline through several thin surfaces:

```
.pyne / .pine source
  → parse (ANTLR4 + ASDL AST)
  → optional: dump | unparse | lint | walk/transform
  → optional: evaluate (literal_eval | Runtime mode=auto|interpret|compile)
  → plots / fill / events / drawings / alerts / metrics
  → optional L2 webhooks (Pro API / edge)
```

The **desk path** is the `pyne` Click CLI (alias `pynescript`) and `pynescript.ast` / `pynescript.runtime` library API (PyPI distribution name `hoox-pyne`). The **TypeScript desk path** is **PyneTS** (`bun add @hoox-sh/pynets`). The **editor path** is `pyne-lsp` (alias `pynescript-lsp`; stdio; first-class `.pyne` association). The **HTTP path** is the Flask Pro API (`POST /run` with alerts export + webhooks, `POST /optimize`, preview, backtest). The optional chart is AXIS; the optional trade mesh is HOOX. Evaluation does not require either. Defaults for `mode` differ by surface — modes.

Conceptual model



Surface	Entry	Typical job
CLI	`pyne …` (alias `pynescript`)	One-shot parse, format, lint, compile, `run`, `optimize`, prewarm, data
Library	`from pynescript.ast import parse, unparse`	Embed parse/transform/eval in tools
PyneTS	`import { parse, Runtime } from "@hoox-sh/pynets"`	Same names in TypeScript / Bun
LSP	`pyne-lsp` (alias `pynescript-lsp`)	Diagnostics, completion, hover (`.pyne` / `.pine`)
Pro API	`POST /run` · `POST /optimize`	Evaluate contract + strategy `input.*` search
AXIS	separate product	Chart PWA AXIS over evaluate results
HOOX	separate product	Edge execution mesh after strategy events

Choose your path

1. Getting started

New install, first parse, and environment knobs:

- **Installation** — PyPI **`pip install hoox-pyne`** (import `pynescript`), extras (`lsp`, `compile`, `data`, `pro`), Hatch checkout, smoke checks.
- **Quick start** — Parse → dump → unparse → lint → `literal_eval` in under ten minutes.

- **Configuration** — Extras, console scripts, Pro API env vars (`ALERT_WEBHOOK_URL`, compile cache), editor settings.

2. Operational guides

Daily workflows against the same pipeline:

- **CLI** — `check`, `format`, `lint`, `compile`, `prewarm`, `run`, `optimize`, `data`, …
- **Library API** — `parse`, `unparse`, `dump`, `walk`, visitors, transformers, linter, `optimize.run_study`.
- **Evaluate scripts** — Expression eval, bar-loop Runtime (`mode=auto`), strategy events, **alerts export**, mock vs live data.
- **Editors** — VS Code (`.pyne` first-class), Neovim, Zed, Emacs, Helix; `clients/` snippets.
- **Pro API usage** — `/run` (+ `webhook_url` / L2 webhooks), `/run/batch`, `/optimize`, preview, backtest as a **consumer**.
- **Troubleshooting** — Parse failures, recursion limits, missing extras, CORS, auth.

3. Reference

- **CLI commands** — Full option trees for every Click command.
- **Modes** — `interpret` / `compile` / `auto` defaults: library **interpret**, `POST /run` **auto**, `pyne run` **compile-only**.
- **Glossary** — AST, series, bar-loop, Runtime, ASDL, and related terms.
- **FAQ** — Version support, TradingView parity, licensing, AXIS/HOOX boundaries.

Interface surface (consumer map)

Want	Command / import	Docs
Install core	`pip install hoox-pyne` (import `pynescript`)	Installation
Install LSP	`pip install "hoox-pyne[lsp]"`	Editors
Install market data deps	`pip install "hoox-pyne[data]"`	CLI data
Parse file	`pyne parse-and-dump path.pyne`	CLI
Normalize source	`pyne parse-and-unparse path.pine`	CLI
Lint	`pyne lint path.pine`	CLI
Library parse	`from pynescript.ast import parse, unparse`	Library API
Expression eval	`from pynescript.ast.helper import literal_eval`	Evaluate
Bar-loop eval	`from pynescript.runtime import Runtime` (`mode` default **interpret**)	Evaluate · modes
Incremental interpret	`Runtime.create_session` → `append_bars` / `update_last_bar` (0.6.1)	Evaluate
Compile-only smoke	`pyne run script.pine --bars 50`	CLI
Warm Numba / IR	`pyne prewarm` / `POST /compile/prewarm`	CLI
Strategy HPO	`pyne optimize` / `POST /optimize` / `pynescript.optimize.run_study`	Optimize
Alerts + webhooks	`/run` → `alerts[]` / `webhook_url`	Alerts
Start LSP	`pyne-lsp` (stdio)	Editors
Local Pro API	`make run` / `python -m backend.app`	Pro API usage

Preferred console scripts (aliases in parentheses) are registered in `pyproject.toml`:

Script	Module	Role
`pyne` (`pynescript`)	`pynescript.__main__:cli`	Desk CLI (Click group)
`pyne-lsp` (`pynescript-lsp`)	`pynescript.langserver.__main__:main`	Language server (pygls)

Do not conflate them: lint and parse live on `pyne`; editor features live on `pyne-lsp`.

Internals (repo paths)

Consumer-relevant code only — deeper design is linked from each guide:

Path	Role
`src/pynescript/__main__.py`	Click CLI: check, format, lint, compile, prewarm, run, optimize, data, info
`src/pynescript/ast/helper.py`	Public `parse`, `unparse`, `dump`, `walk`, `literal_eval`
`src/pynescript/ast/linter.py`	`lint_script` / `PineLinter`
`src/pynescript/ast/evaluator/`	Bar-aware and literal evaluators
`src/pynescript/runtime/`	Package Runtime SoT (bar-loop host, series, CustomEvaluator)
`src/pynescript/langserver/`	LSP features and server
`src/pynescript/util/data.py`	Historical data providers (mock, yahoo, alphavantage, ccxt)
`backend/app.py`	Flask Pro API entry (`/run`, auth, blueprints)
`backend/runtime.py`	Compat shim → `pynescript.runtime`
`examples/`	Worked scripts (parse, RSI strategy, datafeed)
`clients/`	Editor client configs (Neovim, Zed, Emacs)
`vscode-extension/`	VS Code extension package

Invariants & edge cases

- **Round-trip is first-class.** `parse` → `unparse` should preserve semantics; use it to normalize formatting, not as a semantic rewrite.
- **Mode split.** `parse(..., mode="exec")` yields a script tree; `mode="eval"` parses a single expression (used by `literal_eval`).
- **No proprietary host required.** Evaluation works offline with mock or supplied OHLCV; AXIS/HOOX are optional.
- **Extras are opt-in.** Core parse/lint needs only base deps (`antlr4-python3-runtime`, `click`, …). LSP needs `[lsp]`; CCXT paths need `[data]` / `[datafeed]`.
- **Optional example dir.** `pinescript_filepath` only expands when `--example-scripts-dir` points at local `*.pine` files; no third-party corpus is shipped.

Worked examples

Minimal library round-trip:

```
from pynescript.ast import parse, unparse

source = """
//@version=6
indicator("My RSI")
plot(ta.rsi(close, 14))
"""
tree = parse(source)
print(unparse(tree))
```

Minimal CLI:

```
pip install hoox-pyne
pyne parse-and-dump examples/rsi_strategy.pine
pyne lint examples/rsi_strategy.pine
```

Minimal HTTP evaluate (local server running):

```
curl -s http://127.0.0.1:5002/ \
  | python -m json.tool
```

Failure modes

Symptom	Likely cause	Where to go
`pyne: command not found`	Package not on PATH / venv inactive	Installation
`pyne-lsp` missing	Installed without `[lsp]` extra	Editors
`SyntaxError` on parse	Grammar mismatch or truncated source	Troubleshooting
`/run` 400 `NO_DATA`	Missing OHLCV list	Pro API usage
Numerical drift vs TradingView	Builtin / series edge case	Compatibility

See also

- PYNE product map — full stack overview
- Installation
- Quick start
- Runtime modes
- Evaluate contract (API)
- AXIS docs — optional chart
- HOOX docs — optional edge trade mesh

INSTALLATION

Install hoox-pyne from PyPI (live) or a source checkout — CLI, LSP, compile, data, and Pro API extras.

INSTALLATION

Abstract

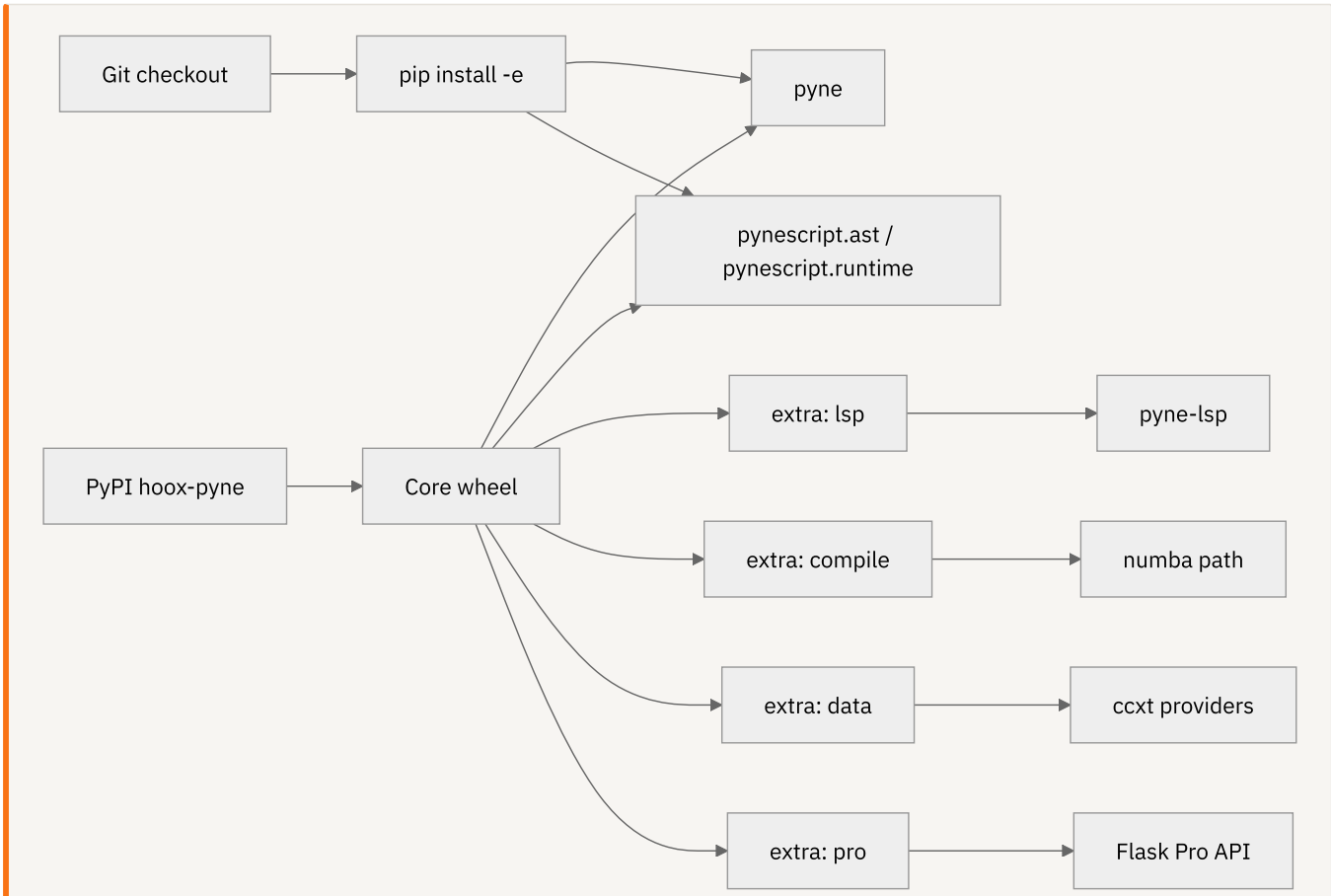
PYNE is live on PyPI as `hoox-pyne` (currently **0.3.14+**). The **import package** remains `pynescript` (src-layout under `src/pynescript/`). Preferred console scripts are `pyne` / `pyne-lsp`; legacy `pynescript` / `pynescript-lsp` remain installed as aliases.

Do **not** install:

- plain `pyne` / `PyNE` from PyPI without the `hoox-` prefix — unrelated 0.1.0 process-networking library (our wheel is `hoox-pyne`)
- upstream `pynescript` (elbakramer) — different project

```
pip install hoox-pyne
pip install "hoox-pyne[lsp]"       # language server
pip install "hoox-pyne[compile]"   # Numba compile path
pip install "hoox-pyne[data]"      # ccxt market data
pip install "hoox-pyne[pro]"       # Flask Pro API stack
```

Conceptual model



Install flavor	What you get	When
`pip install hoox-pyne`	CLI + library (`pynescript.runtime` SoT)	Desk use, scripts, CI lint
`pip install "hoox-pyne[lsp]"`	+ pygls LSP server	Editors
`pip install "hoox-pyne[compile]"`	+ numpy / numba	`compile` / `prewarm` / `run` Numba modes
`pip install "hoox-pyne[data]"`	+ ccxt	Exchange data providers
`pip install "hoox-pyne[pro]"`	+ Flask Pro API stack	Self-hosted `/run`
`pip install -e ".[dev-lsp]"`	Editable + LSP test harness	Contributing to LSP
Docker CLI image	Full CLI image	CI gates / isolated runs
Hatch env	Matrix tests, lint, docs	Full monorepo workflow

Interface surface

Prerequisites

- **Python ≥ 3.10** (classifiers declare 3.10–3.13)
- `pip` or an equivalent installer; virtualenv strongly recommended
- Optional: Docker for the CLI image; Node 22+ only if building the VS Code extension

Core install (PyPI)

```
python3 -m venv .venv
source .venv/bin/activate   # Windows: .venv\Scripts\activate
pip install -U pip
pip install "hoox-pyne[lsp,compile]"
pyne --version
pyne info
```

Project page: pypi.org/project/hoox-pyne · owner account jango-blockchained.

Extras

Defined in `pyproject.toml` under `[project.optional-dependencies]`:

```
pip install "hoox-pyne[lsp]"
pip install "hoox-pyne[compile]"
pip install "hoox-pyne[data]"      # alias: datafeed
pip install "hoox-pyne[pro]"
pip install "hoox-pyne[lsp,compile,data]"
```

Source checkout

```
git clone https://github.com/hoox-sh/pyne.git
cd pyne
pip install -e ".[lsp,pro]"
# or:
make install
```

Docker CLI

```
docker pull   # when published to a registry; or build from source:
git clone https://github.com/hoox-sh/pyne.git && cd pyne
make docker-build-cli
docker run --rm -v "$PWD:/work" -w /work pynescript-cli:latest check script.pine
# ≡ make docker-cli ARGS="check script.pine"
```

Console scripts

Command	Purpose
`pyne`	Preferred Click CLI (check, format, lint, compile, prewarm, run, data, info)
`pyne-lsp`	Preferred language server (requires `[lsp]`)
`pynescript`	Alias of `pyne` (same callable)
`pynescript-lsp`	Alias of `pyne-lsp` (same callable)

```
pyne --version
pyne --help
pyne check script.pine
which pyne-lsp   # only after [lsp]
```

Standalone binaries and a CLI image tarball also ship on GitHub Releases.

Internals (repo paths)

Path	Role
`pyproject.toml`	Package metadata (`name = "hoox-pyne"`), extras, scripts
`src/pynescript/`	Library + CLI + LSP package tree
`src/pynescript/__about__.py`	`__version__`
`src/pynescript/__main__.py`	CLI implementation
`src/pynescript/runtime/`	Package Runtime SoT (bar-loop host, series, evaluator)
`src/pynescript/langserver/__main__.py`	`pyne-lsp` entry
`Dockerfile` target `cli`	Ephemeral CLI image
`Makefile`	`package` , `docker-build-cli` , `install` , ...
`CONTRIBUTING.md`	Publish / Hatch workflow

Invariants & edge cases

- **PyPI name ≠ import name.** Install **`hoox-pyne`** ; `import pynescript` . CLIs: **`pyne`** / **`pyne-lsp`** .
- **License.** AGPL-3.0-or-later. Network use triggers AGPL source-offer obligations (see GNU AGPL §13).
- **LSP is not a CLI subcommand.** Use **`pyne-lsp`** (or `python -m pynescript.langserver`).
- **Pro API.** `pip install "hoox-pyne[pro]"` or monorepo `pip install -r backend/requirements.txt` + `make run` (default `:5002`).
- **Source extensions.** Prefer **`.pyne`** for stack sources; `.pine` / `.pinev5` / `.pinev6` still work.

Worked examples

Smoke-test CLI and library

```
python - <<'PY'
from pynescript.ast import parse, unparse
from pynescript import __about__
print("version", __about__.__version__)
src = '//@version=6\nindicator("t")\nplot(close)\n'
print(unparse(parse(src)))
PY

pyne check examples/rsi_strategy.pine
pyne lint examples/rsi_strategy.pine
# optional (needs [compile] for Numba):
# pyne prewarm
# pyne run examples/rsi_strategy.pine --bars 50
```

Monorepo desk + Pro API

```
git clone https://github.com/hoox-sh/pyne.git
cd pyne
pip install -e ".[lsp,pro]"
make run   # Flask on :5002
```

Failure modes

Failure	Diagnosis	Fix
`ModuleNotFoundError: pynescript`	Wrong env / not installed	Activate venv; `pip install hoox-pyne`
`No module named 'pygls'`	LSP without extra	`pip install "hoox-pyne[lsp]"`
`No module named 'numba'`	compile path without extra	`pip install "hoox-pyne[compile]"`
`No module named 'ccxt'`	data path without extra	`pip install "hoox-pyne[data]"`
Wrong package / missing APIs	Installed upstream `pynescript` or plain `pyne`	`pip uninstall pynescript pyne` then `pip install hoox-pyne`
`pyne` / `pynescript` not found	Scripts not on PATH	`python -m pynescript` or fix venv `bin/`

See also

- Quick start
- CLI guide
- Editors / LSP
- PyPI publish (maintainers)
- End User hub

QUICK START

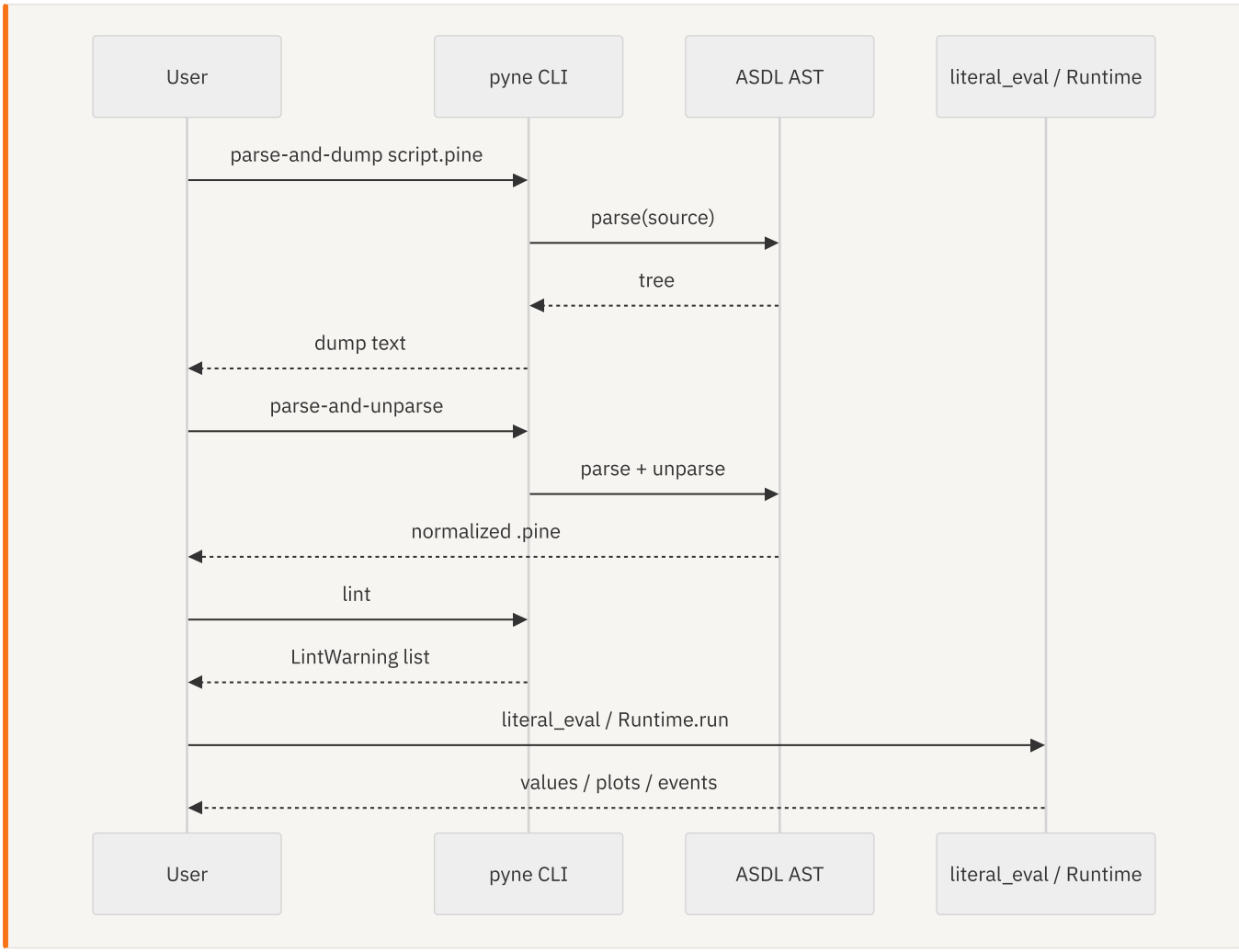
Parse, dump, unparse, lint, and evaluate Pine Script™ with the pyne CLI and library in under ten minutes.

QUICK START

Abstract

This page is a linear path from a working install to the five desk operations most users need: **parse**, **dump**, **unparse**, **lint**, and **evaluate**. Full option trees live in CLI commands; deeper evaluation semantics live in Evaluate scripts.

Conceptual model



Interface surface

Assumes:

```
pip install hoox-pyne                # https://pypi.org/project/hoox-pyne/ · 0.3.14+
# optional:
# pip install "hoox-pyne[lsp]"       # editors
# pip install "hoox-pyne[compile]"   # compile / run / prewarm
# pip install "hoox-pyne[data]"      # ccxt providers
```

```
pyne info
pyne check script.pine
pyne lint script.pine
# aliases still work: pynescript info …
```

Sample script in-tree: `examples/rsi_strategy.pine`. Prefer **`//@version=6`** and **`.pyne`** / **`.pine`** for new work (`.pinev5` remains a supported legacy association).

Internals (repo paths)

Step	Implementation
CLI parse/dump	`src/pynescript/__main__.py` → `parse_and_dump`
CLI unparse	`parse_and_unparse`
CLI lint	`lint` → `pynescript.ast.linter.lint_script`
Library helpers	`src/pynescript/ast/helper.py`
Bar-loop Runtime SoT	`src/pynescript/runtime/` (`from pynescript.runtime import Runtime`)
Example scripts	`examples/parse_dump_unparse.py`, `examples/evaluate_expressions.py`

Invariants & edge cases

- Always declare **`//@version=6`** at the top (v5 still parses); the linter warns (`W001`) if the pragma is missing.
- `parse-and-unparse` is a **formatter via AST**, not a semantic optimizer.
- `literal_eval` is for expressions / built-in calls with optional series context — not a full multi-bar strategy backtester. For bar loops, use `pynescript.runtime.Runtime` (or the Pro API `/run`).
- Filename arguments to the CLI must exist and be readable; lint may also read stdin (`-` or no file).

Worked examples

1. Parse and dump the AST

```
pyne parse-and-dump examples/rsi_strategy.pine
```

Pretty-print with indent and optional file output:

```
pyne parse-and-dump examples/rsi_strategy.pine --indent 2 --output-file /tmp/rsi.ast.txt
```

Library equivalent:

```
from pynescript.ast import parse, dump

with open("examples/rsi_strategy.pine", encoding="utf-8") as f:
    tree = parse(f.read(), "examples/rsi_strategy.pine")
print(dump(tree, indent=2))
```

2. Round-trip unparse (normalize)

```
pyne parse-and-unparse examples/rsi_strategy.pine
pyne parse-and-unparse messy.pine --output-file clean.pine
```

```
from pynescript.ast import parse, unparse

source = """
//@version=6
indicator("My RSI")
rsi(close, 14)
"""
print(unparse(parse(source)))
```

Canonical demo: `examples/parse_dump_unparse.py`.

3. Lint before upload

```
pyne lint examples/rsi_strategy.pine
pyne lint --fail-on warnings examples/rsi_strategy.pine
echo '//@version=6\nindicator("x")\nplot(close)' | pyne lint -
```

Library:

```
from pynescript.ast.linter import lint_script

issues = lint_script(open("examples/rsi_strategy.pine").read(), "rsi_strategy.pine")
for w in issues:
    print(w)  # severity: [CODE] message at line N
```

4. Evaluate expressions

```
from pynescript.ast.helper import literal_eval

print(literal_eval("1 + 2 * 3"))  # 7
print(literal_eval("math.sqrt(16)"))
prices = [100, 102, 101, 103, 105, 104, 106, 108, 107, 110]
print(literal_eval(f"ta.rsi({prices}, 9)"))

# Series history with context
ctx = {
    "close": [100, 102, 101, 103, 105],
    "open": [99, 101, 100, 102, 104],
}
print(literal_eval("close[0] - close[1]", ctx))
```

Broader demo: `examples/evaluate_expressions.py`.

5. Fetch sample market data (CLI)

```
# mock provider (no network)
pyne data AAPL --provider mock

# Yahoo (network)
pyne data AAPL --provider yahoo --period 6mo --interval 1d

# CCXT (needs hoox-pyne[data])
pyne data BTC/USDT --provider ccxt --exchange binance
```

6. Optional: local Pro API `/run`

```
# from monorepo with backend deps
make run
```

```
curl -s -X POST http://127.0.0.1:5002/run \
  -H 'Content-Type: application/json' \
  -d '{
    "script": "//@version=6\nindicator(\"t\")\nplot(close)",
    "mode": "auto",
    "data": [
      {"time": 1, "open": 1, "high": 2, "low": 0.5, "close": 1.5, "volume": 100},
      {"time": 2, "open": 1.5, "high": 2.5, "low": 1.0, "close": 2.0, "volume": 120}
    ]
  }' | python -m json.tool
```

Omit `mode` to use the schema default **`auto`** (warm compile when eligible, else interpret). Response may include `series`, `plot_meta`, `events`, `drawings`, and `alerts`. See Pro API usage for batch mode, webhooks, free-tier guards, and prewarm.

Prefer saving stack sources as **`.pyne`** (`.pine` still works).

7. Library bar-loop vs CLI smoke

Library `Runtime.run` defaults to **interpret**. `pyne run` is **compile-only** on synthetic bars and does not accept `--mode`.

```
from pynescript.runtime import Runtime

ohlcv = [
    {"time": 1, "open": 1, "high": 2, "low": 0.5, "close": 1.5, "volume": 100},
    {"time": 2, "open": 1.5, "high": 2.5, "low": 1.0, "close": 2.0, "volume": 120},
]
out = Runtime(symbol="DEMO").run(
    '//@version=6\nindicator("t")\nplot(close)\n',
    ohlcv,
    mode="interpret",
    timeout_seconds=5.0,
)
print(out.get("mode"), list(out.get("series", {})))
```

```
pip install "hoox-pyne[compile]"
pyne run examples/rsi_strategy.pine --bars 50
```

See modes.

8. Optional: start LSP for editors

```
pip install "hoox-pyne[lsp]"
pyne-lsp   # stdio; normally launched by the editor
# alias: pynescript-lsp
```

Wire Neovim / Zed / Emacs using Editors and `clients/`.

Failure modes

Step fails	Check
Dump raises `SyntaxError`	Script version / unsupported construct — try a minimal `indicator` + `plot(close)`
Unparse output differs cosmetically	Expected; compare semantic structure via dump
Lint exits non-zero	`--fail-on` threshold; inspect codes `E001`, `W001`, …
`literal_eval` `NotImplementedError`	Expression not in literal/builtin subset; use full Runtime
`data` provider error	Network, API key, missing `ccxt` extra

See also

- Installation
- Configuration
- CLI guide
- Library API
- Evaluate scripts
- Runtime modes
- CLI command reference

CONFIGURATION

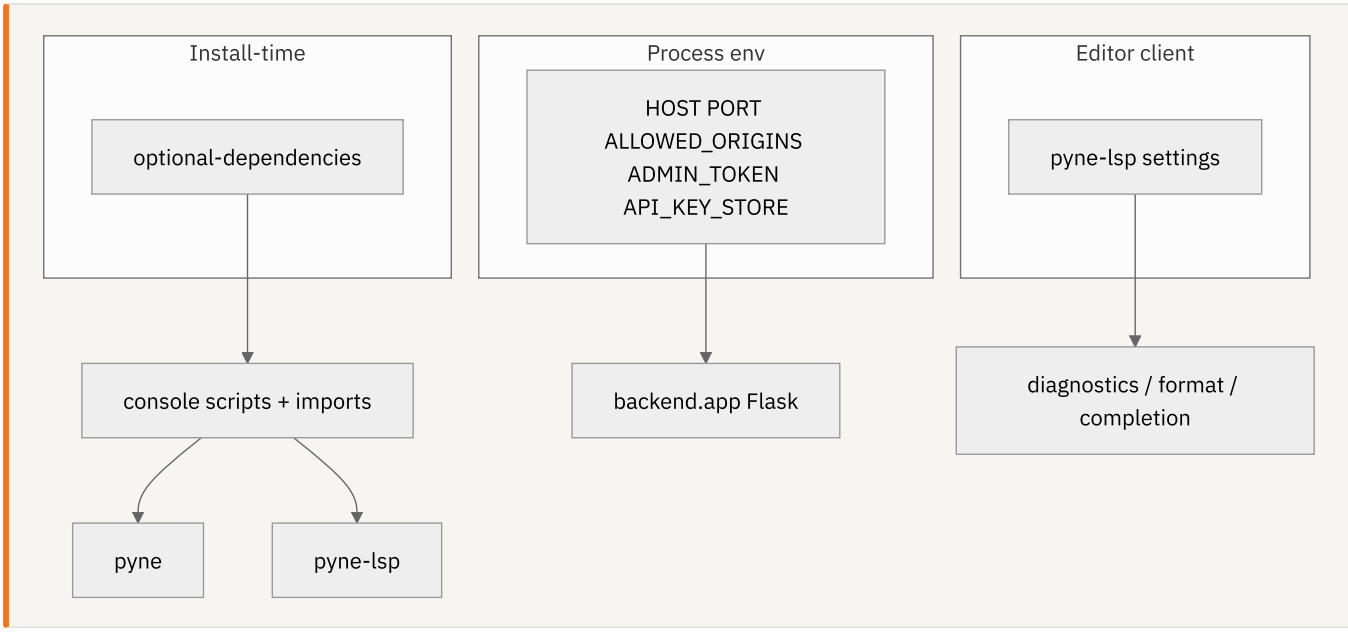
Package extras, console scripts, Pro API environment variables, data providers, and editor LSP settings for PYNE.

CONFIGURATION

Abstract

PYNE is mostly **convention-over-config** at the library layer: parse and evaluate take explicit arguments rather than a global config file. Configuration appears at three boundaries: **install extras** (what code is available), **process environment** (Pro API host, CORS, API keys store), and **editor / LSP client settings** (diagnostics, formatting, binary path). This page inventories those knobs for end users.

Conceptual model



There is no `pynescript.toml` (or similar) for core use. Parse, evaluate, and CLI flags are per-invocation.

Interface surface

Package extras (`pyproject.toml`)

Extra	Dependencies	Enables
(none)	antlr4-runtime, click, requests, tqdm	CLI + AST + linter + literal_eval + `pynescript.runtime`
`lsp`	pygls, lsprotocol	`pyne-lsp` (alias `pynescript-lsp`)
`dev-lsp`	pygls, lsprotocol, pytest-lsp	LSP + protocol tests
`data`	ccxt	Exchange historical data
`datafeed`	ccxt	Same as `data` (alias for realtime feed paths)
`compile`	numpy, numba	Compile / run / prewarm Numba path
`pro`	Flask stack (+ redis optional)	Self-hosted Pro API

Console scripts

Name	Entry point	Notes
`pyne`	`pynescript.__main__:cli`	Preferred desk CLI
`pyne-lsp`	`pynescript.langserver.__main__:main`	Preferred language server
`pynescript`	same as `pyne`	Legacy alias
`pynescript-lsp`	same as `pyne-lsp`	Legacy alias

Invoke without scripts on PATH:

```
python -m pynescript --help
python -m pynescript.langserver
```

CLI flags that act as “config”

Per-invocation only (no persistence):

Command	Notable options
`check`	`--encoding` , `-q/--quiet` , `--ext` (directory walk)
`format` / `fmt`	`-w/--write` , `--check` , `-o/--output-file`
`parse-and-dump`	`--encoding` , `--indent` , `--output-file`
`parse-and-unparse`	`--encoding` , `--output-file`
`lint`	`--encoding` , `--fail-on {errors,warnings,all,never}` , `--json` , `-q`
`compile`	`--emit` , `-o` , `--time/--no-time`
`prewarm`	`--force` , `--json`
`run`	`--bars` (default 50), `--json` , `-q` — **compile-only** synthetic OHLCV
`data`	`--provider` , `--period` , `--interval` , `--api-key` , `--secret` , `--exchange` , `--format`

`lint` has **no** `--fix`. Structural rewrite is `pyne format -w` (parse → unparse).

Pro API environment variables

Consumed by `backend/app.py` and middleware:

Variable	Default	Purpose
`HOST`	`127.0.0.1`	Bind address when running `__main__`
`PORT`	`5002`	Bind port
`ALLOWED_ORIGINS`	`https://pynescript.ai`, `https://app.pynescript.ai`, plus localhost regex	CORS allowlist (comma-separated; regex allowed). Code also appends localhost, private-LAN, and product-origin (`hoox.sh` / `pynescript.ai` / `pynescript.online`) regexes unless `*`
`ADMIN_TOKEN`	unset	Required for `POST /auth/create_key`; if unset, create returns 403
`API_KEY_STORE`	`/root/pynescript/data/api_keys.json`	Path for key store (override for local)
`ALERT_WEBHOOK_URL`	unset	Default L2 alert webhook for `/run` when body omits `webhook_url`
`ALERT_WEBHOOK_TIMEOUT`	`10`	Webhook HTTP timeout (seconds)

Also relevant:

Variable	Context
`MAX_CONTENT_LENGTH`	Hardcoded 5 MiB on Flask app config (not env) — large OHLCV POSTs fail above this

Runtime / evaluate environment variables

Used by `pynescript.runtime.Runtime` (`src/pynescript/runtime/host.py`) and the compiler host. `backend.runtime` is a compat re-export only.

Variable	Default	Purpose
`PYNE_RUNTIME_MODE`	`interpret`	Default when `Runtime.run(..., mode=None)` omits mode (**Pro API schema default is still `auto`**; `pyne run` ignores this — compile-only)
`PYNE_SERIES_CAP`	**on**	Trim append-only host series lists; set `0` / `false` / `off` to disable (oracle / debug)
`PYNE_SERIES_MAX`	unset	Absolute series history depth (overrides default 256 + `max_bars_back`)
`PYNE_TA_INCREMENTAL`	**on**	Bar-mode incremental `ta.*` hot path; set `0` to force full recompute
`PYNE_SERIES_RING`	**off**	Chronological tail view; skip dual list write. Keep off unless you need the ring path
`PYNE_PARSE_CACHE`	**on**	Process-local LRU of successful `parse()` trees; set `0` to disable
`PYNE_PARSE_CACHE_MAX`	`128`	Max parse-cache entries
`PYNE_COMPILE_DISK_CACHE`	often `1` in deploy	Persist compiled IR across processes
`PYNE_COMPILE_CACHE_DIR`	XDG cache / Docker `/data/compile-cache`	Disk IR location
`PYNE_COMPILE_PREWARM`	deploy-dependent	Host cold-start prewarm of Numba builtins

See Series & history, Technical builtins, and Compiler overview.

Local run:

```
python -m backend.app
# or: make run
```

Data providers

CLI `pynescript data` / library providers:

Provider	Auth	Notes
`mock`	none	Deterministic offline bars (default)
`yahoo`	none	Yahoo Finance path
`alphavantage`	`--api-key` (falls back to `demo`)	Limited with demo key
`ccxt`	optional key/secret; `--exchange`	Requires `[data]` extra

Pro API `/run` optional fields:

Field	Values	Role
`data_source`	`""`, `mock`, `ccxt`, `ccxtpro`, `yahoo`, `alphavantage`	Wires `request.*` resolution
`data_options`	object	exchange, api_key, seed, …
`mode`	`auto` (default), `interpret`, `compile`	Prefer warm compile + fallback; strict compile; full interpret
`inputs`	object	`input.*` overrides by title (forces interpret under `auto`)
`libraries`	list	`[{namespace, name, version, source}]` — max 32; also on `/run/batch`; interpret / `auto` fallback
`timeout_seconds`	number	Optional interpret wall-clock budget; omit / `null` / `≤ 0` → no timeout
`profiler`	bool	Per-line timings; forces interpret
`webhook_url`	string	Per-request L2 alert webhook (overrides `ALERT_WEBHOOK_URL`)
`forward_alerts` / `alert_last_bar` / `alert_batch`	bool	Webhook delivery controls (see Alerts)

`timeout_seconds` is accepted on `Runtime.run`, `POST /run`, and `POST /run/batch`. Extra keys that are not in the schema still → `UNKNOWN_FIELDS`.

Operators can also `POST /compile/prewarm` or run `pyne prewarm` (and `pyne prewarm script.pine …`) to warm Numba builtins / script IR caches before interactive traffic. Requires `pip install "hoox-pyne[compile]"` for full Numba warm-up.

Free-tier Pro API guards (unauthenticated `/run`) are **off by default**. Set `FREE_TIER_LIMITS=1` to enable `FREE_MAX_BARS`, `FREE_MAX_SCRIPT_CHARS`, `FREE_MAX_CONCURRENT`, `FREE_RATE_LIMIT` / `FREE_RATE_WINDOW_SEC` — see Pro API usage.

Editor / LSP client settings

VS Code extension keys (from `clients/README.md`):

Setting	Default	Meaning
`pynescript.lsp.enabled`	`true`	Toggle server
`pynescript.lsp.command`	`auto`	`auto` tries `pyne-lsp` → `pynescript-lsp` → `python3 -m pynescript.langserver`; or an absolute path
`pynescript.lsp.python`	`python3`	Interpreter when command is `auto` and no binary is on PATH
`pynescript.lsp.args`	`[]`	Extra args to the language server
`pynescript.formatting.enabled`	`true`	Format document
`pynescript.diagnostics.enabled`	`true`	Lint squiggles
`pynescript.completion.snippets`	`true`	Snippet completions

Extension id: **`hoox-sh.pyne`**.

Neovim (`clients/neovim.lua`):

```
settings = {
  pinescript = {
    formatting = { enabled = true },
    diagnostics = { enabled = true },
    completion = { snippets = true },
  },
}
```

Zed: `language_servers.pynescript.command` / `arguments: ["--stdio"]` — see Editors.

AXIS / frontend coupling (optional)

When running SuperChart Lite PWA against local Flask:

Make target	Port (typical)
`make run`	API `:5002`
`make run-frontend`	PWA `:8081`
`make worker-dev`	CF Worker `:8787`

CORS must allow the PWA origin via `ALLOWED_ORIGINS` .

Internals (repo paths)

Path	Config surface
`pyproject.toml`	extras, scripts, Python requires
`src/pynescript/__main__.py`	CLI options
`src/pynescript/runtime/host.py`	`Runtime.run` (`mode` , `libraries` , `timeout_seconds` , `inputs`)
`backend/app.py`	Flask, CORS, HOST/PORT, body size
`backend/middleware/auth.py`	`API_KEY_STORE` , key tiers
`backend/middleware/schemas.py`	Request field defaults (`mode` , `symbol` , ...)
`clients/*.json	lua
`vscode-extension/`	Extension contribution points

Invariants & edge cases

- **Unknown JSON fields on Pro API are rejected** (`UNKNOWN_FIELDS`) — schemas are strict.
- **CORS defaults exclude arbitrary production domains** — set `ALLOWED_ORIGINS` deliberately.
- **Key store default path is production-oriented** (`/root/...`) — always override locally.
- **Lint** `--fail-on` only affects process exit code; it does not change which rules run.
- **Recursion limit** during parse is raised to at least 5000 temporarily inside `helper._parse` for deep nests; not user-configurable.

Worked examples

Desk-only machine

```
python -m venv .venv && source .venv/bin/activate
pip install hoox-pyne
# no env vars required
pyne lint strategy.pine --fail-on errors
```

Editor workstation

```
pip install "hoox-pyne[lsp]"
# ensure which pyne-lsp  (alias: pynescript-lsp)
# paste clients/neovim.lua or clients/zed.json as documented
```

Local API + AXIS

```
pip install -e ".[lsp]"
pip install -r backend/requirements.txt

make run
# other terminal:
make run-frontend
```

Scripted Alpha Vantage fetch

```

pynescript data EUR/USD --provider alphavantage --api-key "$AV_KEY" --period 3mo
```

Failure modes

Symptom	Config fix
Browser CORS error on `/run`	Add origin to `ALLOWED_ORIGINS`
`create_key` always 403	Set `ADMIN_TOKEN` and send `X-Admin-Token`
API keys vanish / permission error	Point `API_KEY_STORE` to writable path
Payload too large	Split bars or stay under 5 MiB body limit
Editor “server failed to start”	Fix `pynescript.lsp.command` PATH; install `[lsp]`
`ccxt` import errors	Install `[data]` extra

See also

- Installation
- Pro API usage
- Editors

- CLI commands
- Runtime modes
- API auth (systems)
- AXIS docs

CLI GUIDE

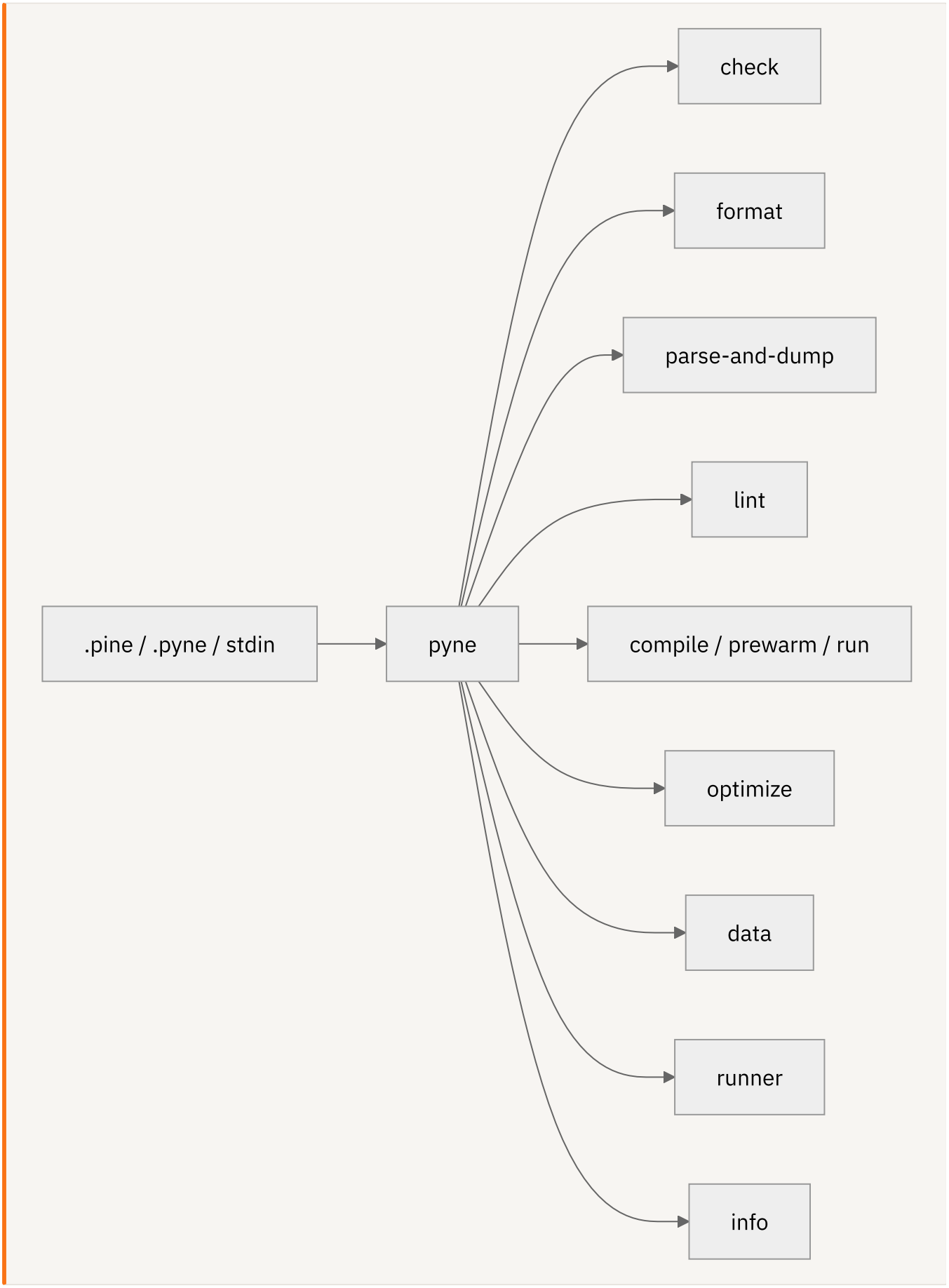
Use the pyne Click CLI to check, format, lint, compile, run, optimize, prewarm, fetch data, and more.

CLI GUIDE

Abstract

The preferred `pyne` console script (alias `pynescript`) is a Click group over the same helpers used by the library API. Install with `pip install hoox-pyne` (live on PyPI). It is optimized for **shell pipelines, CI gates, one-shot inspection, and light compile/run smokes**. Full multi-bar evaluation with real OHLCV belongs on the package `pynescript.runtime.Runtime` or Pro API. Exhaustive flags: CLI commands.

Conceptual model



CLI	Library / subsystem
`check`	`parse` only (exit codes)
`format` / `fmt`	`parse` + `unparse`
`convert`	`pynescript.util.pine_convert` (any older version → v6)
`parse-and-dump` / `dump`	`parse` + `dump`
`lint`	`lint_script`
`compile` / `prewarm` / `run`	`pynescript.compiler` (+ `[compile]` extra)
`optimize`	`pynescript.optimize.run_study`
`data`	`pynescript.util.data.get_provider`
`runner`	HTTP client for Flask `/scripts` + `/cron/run` (`PYNE_RUNNER`)
`info`	version + optional extras

Interface surface

```
pyne --help
pyne --version
pyne <command> --help
# alias: pynescript …
```

Command	Input	Output
`check PATHS…`	files / dirs / stdin	ok/fail lines + exit code
`format PATH`	file or `-`	formatted source (`-w` / `--check`)
`convert PATH --to 6`	file or `-`	rewritten v6 source (`-w`)
`parse-and-dump PATH`	file	AST dump
`parse-and-unparse PATH`	file	Pine source
`lint [PATH|-]`	file or stdin	diagnostics (`--json`)
`compile PATH`	file	emit Python or compile-check
`prewarm [PATH…]`	optional files	warm Numba / IR cache
`run PATH`	file	synthetic-bar execute
`optimize PATH`	file + `--space` JSON	strategy hyperparameter search
`data SYMBOL`	symbol + provider opts	table / json / csv
`runner …`	file / id	hosted deploy + tick (runner)
`info`	—	version + extras

Not part of this Click group: language server — use **`pyne-lsp`** (alias `pynescript-lsp`).

Install options

Method	When
`pip install hoox-pyne`	Default desk / CI use
`pip install "hoox-pyne[compile]"`	Need Numba `compile` / `run` modes
GitHub Release binary `pynescript-cli-*`	No Python install
Docker `pynescript-cli`	Isolated CI / one-shot checks

```
# Docker
docker buildx bake cli
docker run --rm -v "$PWD:/work" -w /work pynescript-cli:latest check script.pine

# Compose (repo root)
make docker-cli ARGS="lint script.pine --json"

# Standalone binary (from GitHub Release)
chmod +x pynescript-cli-linux-x86_64
./pynescript-cli-linux-x86_64 info
```

Internals (repo paths)

Path	Role
`src/pynescript/__main__.py`	All command implementations
`src/pynescript/ast/helper.py`	`parse` , `dump` , `unparse`
`src/pynescript/ast/linter.py`	`lint_script`
`src/pynescript/compiler/`	`compile` / `prewarm` / `run`
`src/pynescript/util/data.py`	providers + `DataProviderError`
`src/pynescript/util/runner_cli.py`	`pyne runner` HTTP client
`examples/`	sample `.pine` inputs
`tests/test_cli.py`	CLI unit suite (CI)

Invariants & edge cases

- **Encoding default is UTF-8** for file reads/writes.
- **`--output-file -`** means stdout (Click `allow_dash=True`).
- **Lint without a path** reads stdin — useful in git hooks and `cat script.pine | pyne lint`.
- **`--fail-on`** defaults to `errors` (syntax/ `E001`); `warnings` fails on any warning severity too; `all` fails if any issue exists.
- **`data` default provider is `mock`** — offline-safe.
- **Alpha Vantage without key** prints a warning and uses `demo`.
- **`lint` is one file or stdin** — no directory walk. Use `check` for recursive parse-only (`--ext`).
- **`run` is compile-only** on synthetic bars (`--bars` default 50). It does not call `Runtime.run` and has no `--mode`.

Worked examples

Inspect structure of a strategy

```
pyne parse-and-dump examples/rsi_strategy.pine --indent 2 | less
```

Compare two scripts' shapes in CI:

```
pyne parse-and-dump a.pine --output-file /tmp/a.ast
pyne parse-and-dump b.pine --output-file /tmp/b.ast
diff -u /tmp/a.ast /tmp/b.ast
```

Format / normalize

```
pyne format messy.pine -w
pyne format strategy.pine --check    # CI drift gate (exit 1 if would change)
pyne parse-and-unparse messy.pine --output-file clean.pine
```

Lint gate in CI

```
set -e
# lint is one file or stdin — not a directory walk
for f in strategies/*.pine strategies/*.pyne; do
  pyne lint --fail-on warnings "$f"
done
# parse-only gate over a tree:
pyne check strategies/ --ext .pine
```

Stdin pipeline:

```
git show HEAD:strategies/rsi.pine | pyne lint --fail-on errors -
```

Market data smoke checks

```
pyne data AAPL
pyne data AAPL --provider yahoo --period 1y --interval 1d
pyne data BTC/USDT --provider ccxt --exchange binance
pyne data EUR/USD --provider alphavantage --api-key "$AV_KEY"
```

Typical table keys (from `__main__.py`): `symbol`, `provider`, `bars`, `period / interval`, `first close`, `last close`, `change`, `high / low`, `avg volume`. `--format json` emits full OHLCV lists.

Compile, prewarm, synthetic run

```
pip install "hoox-pyne[compile]"
pyne compile script.pine              # compile-check (Numba or object-mode)
pyne compile script.pine --emit       # print generated Python
pyne prewarm                          # warm shared Numba kernels
pyne prewarm script.pine              # also compile that source into IR caches
pyne run script.pine --bars 100       # compile-only execute on synthetic OHLCV
```

Compose with library for evaluation

`pyne run` is compile-only smoke. Real bars + `mode` / `libraries` / `timeout_seconds` belong on the package Runtime:

```
pyne format strat.pine -o /tmp/s.pine
python - <<'PY'
from pathlib import Path
from pynescript.runtime import Runtime
# hand off to Runtime or your own visitor
print(Path("/tmp/s.pine").read_text()[:40])
PY
```

Failure modes

Exit / message	Cause	Mitigation
Click path errors	missing file	check `PATH` exists and is a file
Parse exception during dump	invalid syntax	fix source; use smaller repro
`Lint failed with errors.`	`--fail-on` threshold	read `E00x` / `W00x` lines
`DataProviderError` → ClickException	network / auth / missing dep	install `[data]`, fix keys
Empty dump / unexpected tree	`mode` always exec for CLI	expression-only needs library `mode="eval"`

See also

- CLI commands reference
- Runtime modes
- Library API
- Quick start
- Troubleshooting

LIBRARY API

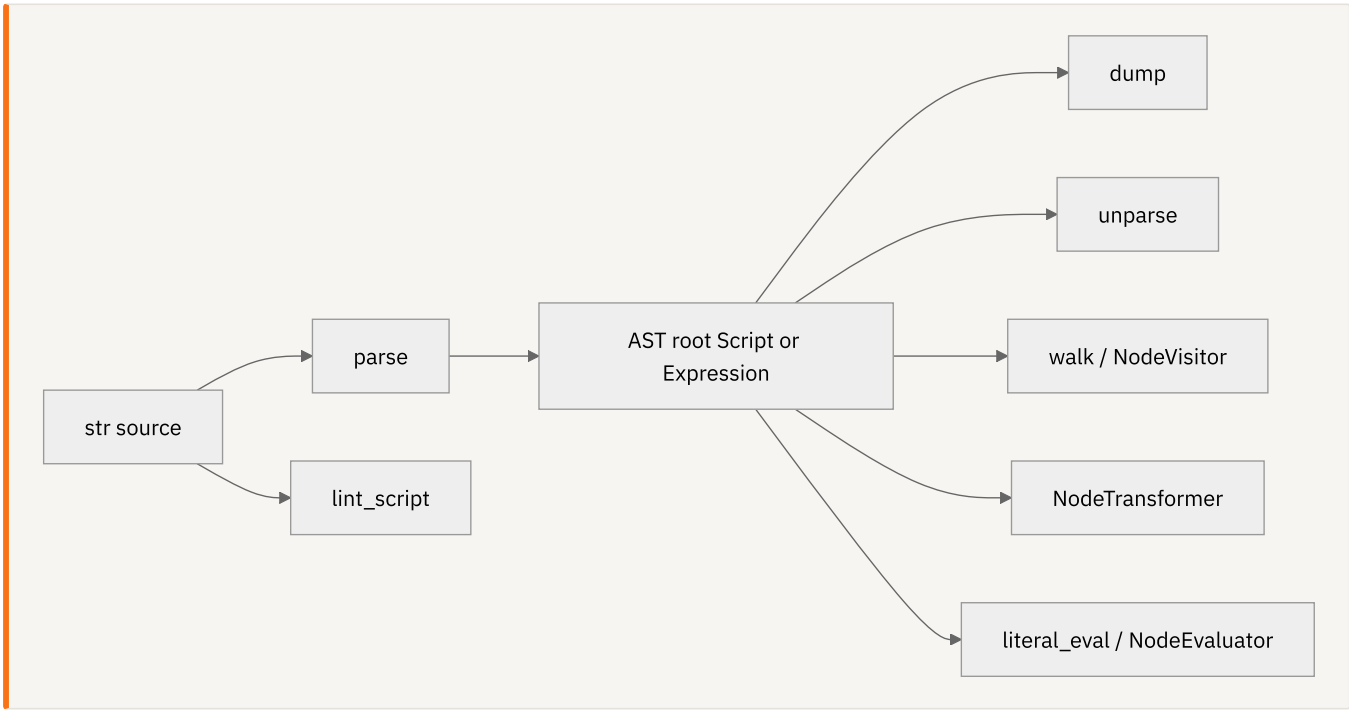
Embed pynescript: parse, dump, unparse, walk, transform, and lint Pine Script™ from Python.

LIBRARY API

Abstract

The public Python surface for language work lives primarily under **`pynescript.ast`**: parse source into an ASDL-backed tree, serialize it (`dump`), regenerate source (`unparse`), traverse (`walk`, visitors), rewrite (`NodeTransformer`), and lint (`lint_script`). Expression evaluation (`literal_eval`) and full script execution (`pynescript.runtime.Runtime`) sit adjacent; this page focuses on **structural** APIs. Evaluation workflows are covered in Evaluate scripts. Strategy hyperparameter search is **`pynescript.optimize.run_study`** (not a Pine builtin) — POST /optimize. Install via **`pip install hoox-pyne`** (`import pynescript`).

Conceptual model



Design intent mirrors Python's `ast` module: **stable tree operations**, location attributes, and visitor protocols — specialized for Pine's script model (annotations, series-aware semantics live at evaluation time).

Interface surface

Imports

```
# Re-exported convenience (node classes, helpers, visitors, transformers)
from pynescript.ast import parse, unparse, dump, walk
from pynescript.ast import NodeVisitor, NodeTransformer
from pynescript.ast.helper import literal_eval, iter_child_nodes, get_source_segment
from pynescript.ast.linter import lint_script, lint_file, PineLinter, LintWarning
```

`pynescript.ast.__init__` star-imports `helper`, `node`, `visitor`, `transformer`, and `error`.

`parse(source, filename="<unknown>", mode="exec") -> AST`

Arg	Meaning
`source`	Full script or expression text
`filename`	Error reporting; absolutized if file exists
`mode`	`"exec"` → script statements; `"eval"` → single expression

Raises `ValueError` on bad mode; `SyntaxError` (via error listener) on grammar failure.

Successful trees are cached in a process-local LRU (`sha256(source)` + `mode`). Disable with `PYNE_PARSE_CACHE=0`. Cached trees are **read-only** (shared by identity).

```
tree = parse('//@version=6\nindicator("x")\nplot(close)\n')
expr = parse("close > open", mode="eval")
```

`unparse(node) -> str`

Round-trips an AST through `NodeUnparser`.

```
normalized = unparse(parse(messy_source))
```

`dump(node, *, annotate_fields=True, include_attributes=False, indent=None) -> str`

`indent` may be an `int` (spaces per level) or a string. `None` is a single line.

Human-readable tree serialization (debug, tests, CLI).

```
print(dump(tree, indent=2, include_attributes=True))
```

`walk(node) -> Iterator[AST]`

Breadth-first (deque) traversal yielding every node.

```
from pynescript.ast import walk
names = [n for n in walk(tree) if n.__class__.__name__ == "Name"]
```

Location helpers

Function	Role
`copy_location(new, old)`	Copy lineno/col offsets
`fix_missing_locations(node)`	Fill missing positions
`increment_lineno(node, n=1)`	Shift lines
`get_source_segment(source, node, *, padded=False)`	Slice original text
`iter_fields` / `iter_child_nodes`	Schema-aware iteration

Visitors and transformers

```
from pynescript.ast import NodeVisitor, NodeTransformer, parse

class CallCounter(NodeVisitor):
    def __init__(self):
        self.calls = 0
    def visit_Call(self, node):
        self.calls += 1
        self.generic_visit(node)

class RenameClose(NodeTransformer):
    def visit_Name(self, node):
        # Name fields: id (identifier), ctx (Load | Store | …)
        if node.id == "close":
            node.id = "price"
        return node

tree = parse(source)
CallCounter().visit(tree)
tree2 = RenameClose().visit(tree)
```

Linter

```
from pynescript.ast.linter import lint_script, LintWarning

warnings: list[LintWarning] = lint_script(source, "strat.pine")
for w in warnings:
    print(w.code, w.severity, w.line, w.message)
```

Code family	Examples
`E001`	Syntax error (error severity)
`W001` – `W002`	Missing / old `@version`
`W101`	Deprecated `security('EXCHANGE:SYM')` (W102/W103 retired)
`C001` / `C002` / `C004`	Naming / style (line length 120, trailing newline). `C003` retired.

`PineLinter.lint` runs: syntax → version → deprecated → naming → style.

`literal_eval` (bridge to evaluation)

```
from pynescript.ast.helper import literal_eval

literal_eval("ta.sma([1,2,3,4,5], 3)")
literal_eval("close[1]", {"close": [10.0, 11.0, 12.0]})
```

Optional `data_feed` / `data_provider` for `request.*` in literal contexts.

Internals (repo paths)

Path	Role
`src/pynescript/ast/helper.py`	Public parse/dump/walk/unparse/literal_eval
`src/pynescript/ast/builder.py`	Parse tree → ASDL nodes
`src/pynescript/ast/unparser.py`	`NodeUnparser`
`src/pynescript/ast/visitor.py`	`NodeVisitor`
`src/pynescript/ast/transformer.py`	`NodeTransformer`
`src/pynescript/ast/node.py`	Node re-exports / helpers
`src/pynescript/ast/grammar/asdl/resource/Pinescript.asdl`	Algebraic schema
`src/pynescript/ast/linter.py`	Static rules
`src/pynescript/ast/error.py`	Error types
`examples/parse_dump_unparse.py`	Round-trip demo

Pipeline inside `parse`:

1. `InputStream` / `FileStream`
2. `PinescriptLexer` + `PinescriptParser` (error listener)
3. `PinescriptASTBuilder.visit`
4. In `exec` mode: `StatementCollector`, comment collection, `@` annotation attach

Invariants & edge cases

- **Annotation comments** (`//@version`, etc.) attach to script / function / type / assign nodes by kind suffix — see `_add_annotations` in `helper.py`.
- **Deep nesting:** temporary `sys.setrecursionlimit(max(old, 5000))` during parse.
- **`dump` requires a true AST node** — `TypeError` otherwise.
- **Transformers mutate or replace nodes** depending on your visit methods; prefer returning new nodes if you need purity.
- **Round-trip fidelity** is tested against a large corpus but is not a proof of bit-identical source; whitespace and some sugar may normalize.
- **Do not edit generated grammar modules**; extend via resource grammars if contributing.

Worked examples

Strategy hyperparameter search (0.3.12)

```
from pynescript.optimize import run_study, space_from_payload
from pynescript.optimize.types import ValidationSpec

space = space_from_payload({
    "params": [{"name": "Fast", "kind": "int", "min": 3, "max": 12}],
})
study = run_study(
    open("strategy.pine").read(),
    ohlcv_bars,
    space,
    n_trials=20,
    sampler="tpe",
    validation=ValidationSpec(mode="holdout"),
)
print(study.best_params, study.best_oos_score)
```

HTTP: POST /optimize. CLI: `pyne optimize`.

End-to-end inspect

```
from pynescript.ast import parse, dump, unparse, walk

source = open("examples/rsi_strategy.pine", encoding="utf-8").read()
tree = parse(source, "examples/rsi_strategy.pine")
print(dump(tree, indent=2)[:500], "...")
print("---")
print(unparse(tree))
print("node count", sum(1 for _ in walk(tree)))
```

Collect all call names

```
from pynescript.ast import parse, NodeVisitor

class Calls(NodeVisitor):
    def __init__(self):
        self.names = []
    def visit_Call(self, node):
        func = node.func
        # Attribute vs Name depending on ta.rsi vs f()
        self.names.append(func)
        self.generic_visit(node)

c = Calls()
c.visit(parse(source))
```

Lint programmatically with fail semantics

```
from pynescript.ast.linter import lint_script

def assert_clean(path: str) -> None:
    text = open(path, encoding="utf-8").read()
    issues = lint_script(text, path)
    errors = [i for i in issues if i.severity == "error"]
    if errors:
        raise SystemExit("\n".join(map(str, errors)))
```

Expression mode for tooling

```
from pynescript.ast import parse, dump

print(dump(parse("ta.rsi(close, 14)", mode="eval"), indent=2))
```

Failure modes

Exception / result	Meaning
`SyntaxError`	Lexer/parser rejection
`ValueError: invalid argument mode`	mode not in `{exec,eval}`
Empty / odd tree	Empty body scripts; annotations-only edge cases
Unparse mismatch	Node kinds without unparser support — file issue against unparser
Linter false positives	Style rules are heuristic (regex naming) — treat `C00x` as advisory

See also

- Evaluate scripts
- CLI guide
- Helper API (core)
- Visitor / transformer (core)
- Linter (core)

EVALUATE SCRIPTS

Run Pine expressions and multi-bar scripts with literal_eval, pynescript.runtime.Runtime (libraries=, timeout_seconds, mode), data providers, and strategy events.

EVALUATE SCRIPTS

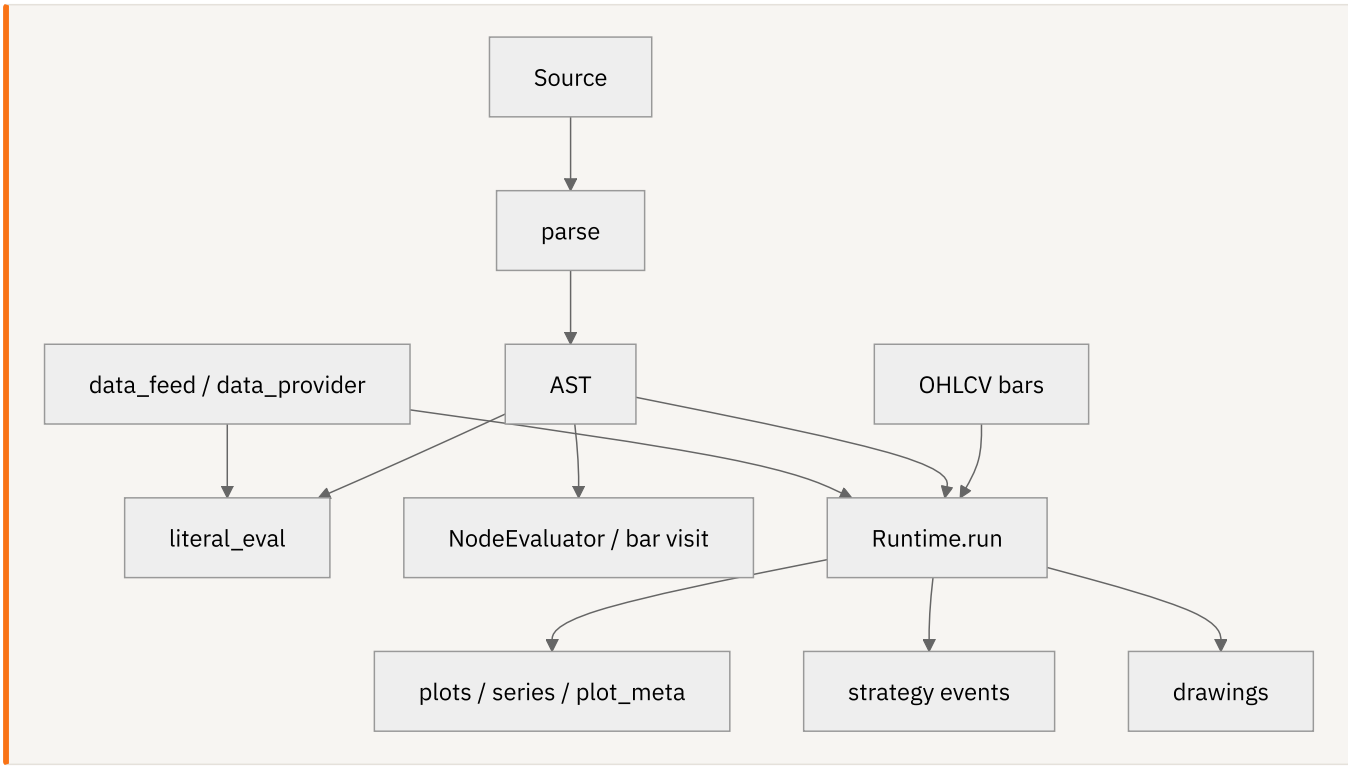
Abstract

Evaluation is the second half of the PYNE pipeline: given an AST (or source) and **market context**, produce plots, series, strategy events, and drawings. End users typically choose among three graded tools:

1. `literal_eval` — pure/builtin expressions, optional series context
2. **AST evaluator** (`NodeLiteralEvaluator` / full evaluator mixins) — script-shaped evaluation in-process
3. `pynescript.runtime.Runtime` (package SoT; `backend.runtime` re-exports) or **Pro API** `POST /run` — bar-loop over OHLCV with shared evaluate contract. TypeScript: `@hoox-sh/pynets` `Runtime.run` — PyneTS. Defaults differ by surface — modes.

This guide stays at the consumer level; series semantics and builtin inventories are under Runtime.

Conceptual model



Mode	Bar loop	Strategy	Typical use
`literal_eval`	no (single shot)	limited	TA on arrays, math, strings
Evaluator in tests / tools	optional	yes via state	Unit tests, notebooks
`Runtime.run` / `/run`	yes	yes	Charts, AXIS, backtests
`mode="compile"`	yes (numeric + object)	yes (object mode)	Faster bar loop when eligible
`mode="auto"`	yes	yes	Prefer compile; fall back to interpret

Runtime modes (`interpret` | `compile` | `auto`)

`pynescript.runtime.Runtime.run` accepts a `mode` argument that selects how the bar loop executes:

Mode	Behavior
`interpret`	AST walker (full host semantics: `input.*` overrides, `request.*`, libraries, profiler).
`compile`	Transpile to a Numba numeric kernel when possible, otherwise a pure-Python **object-mode** bar loop. Requires the compiler package.
`auto`	Try compile first; on eligibility failure, compile error, or compiled runtime error, **fall back to interpret**.

Defaults differ by surface:

- Direct `Runtime.run(..., mode=None)` → `PYNE_RUNTIME_MODE` env, else **`interpret`**.
- Pro API `POST /run` body omit `mode` → schema default **`auto`** (warm compile + fallback).

```
from pynescript.runtime import Runtime

runtime = Runtime(symbol="AAPL")
result = runtime.run(
    source,
    ohlcv_data=ohlcv,
    mode="auto",                # omit → PYNE_RUNTIME_MODE else "interpret"
    timeout_seconds=12.0,       # interpret wall-clock budget; None = no limit
    libraries=[                 # import ns/Name/ver (max 32 on POST /run)
        {"namespace": "ns", "name": "Lib", "version": 1, "source": '//@version=6\nlibrary("Lib")\r
    ],
    inputs={"len": 14},         # input.* by title — forces interpret under auto
)
# result["mode"] / result.get("auto_backend") → "compile" | "interpret"
# on fallback: result.get("compile_fallback_reason")
# on timeout: timed_out + error_kind="runtime" with partial plots
```

`timeout_seconds` is accepted on the library host and on `POST /run` / `POST /run/batch` (omit / `null` / `≤ 0` → no budget). `pyne run` does not call `Runtime.run` at all.

Warm compile / prewarm

Product hosts cache compiled IR (in-process LRU + optional disk under `PYNE_COMPILE_CACHE_DIR`). Pro API defaults `mode=auto` and can warm Numba builtins once per worker (`PYNE_COMPILE_PREWARM` , `POST /compile/prewarm` , or `pyne prewarm [PATH…]`). First-hit latency still pays JIT when cold; subsequent runs of the same source reuse IR. See Compiler overview.

Related host knobs (interpret path): `PYNE_SERIES_CAP` / `PYNE_SERIES_MAX` (history trim) and `PYNE_TA_INCREMENTAL` (bar-mode TA hot path)—see Series & history and Configuration.

From **0.3.10**, interpret also skips unused derived OHLCV (`hl2` / `hlc3` / `ohlc4` / `tr`) and inlines Assign / plot / Call after bar 0. Same-machine bench @ 2000 bars vs 0.3.9: minimal **2.7×**, `ta.sma` **2.0×**, multi-plot TA combo **1.45×**. Prefer `mode="auto"` on Pro `/run` when the script is compile-eligible; interpret remains the full-surface oracle.

When compile falls back (or is skipped)

`mode="auto"` (and explicit prefilters) skip or abandon compile for stable reasons. Common `compile_fallback_reason` values:

Reason (examples)	Why
`input.* overrides require interpret path`	Non-empty `inputs=` are interpret-only; compile does not apply overrides.
`import statements not supported in compile path`	Library `import` needs the interpreter registry.
`request.* not supported in compile path`	Multi-symbol / external data plumbing is interpret-only.
`compiler package unavailable`	`pynescript.compiler` not importable in this install.
`Compile Error: …` / Numba required messages	Deterministic transpile/env failures (cached per source for auto).
Compiled runtime errors	Data-dependent failures still fall back once; not permanently cached as “never compile.”

Other force-interpret paths:

- **`profiler=True`** — line timings need the AST walker; compile/auto are coerced to interpret.
- **`mode="compile"`** (strict) — does **not** fall back; you get an error payload instead of a silent interpret run.

Numba is **not** required for compile eligibility: strategy / UDT / map-heavy scripts can still run in object-mode compile. Missing Numba only blocks pure-numeric emit.

Plot series parity

Both backends export the same Runtime contract shape: `series` (title → per-bar values), `plots` , `plot_meta` , `events` , `drawings` , plus `mode` / diagnostics. For charting, **shared plot series keys** should agree within floating-point tolerance (`na` /None/NaN treated as NA).

First-party `hline` / `fill` / `bgcolor` / `plotshape` **keys match** interpret ↔ compile (0.3.11). **0.3.12** also matches `barcolor` / `plotarrow` / `plotbar` / `plotcandle` kinds and geometry-only `drawings` (line/box/label/polyline/table/linefill) — coordinates are snapshotted at create/set. Harness `--ignore-hline-keys` / `--ignore-fill-keys` stay optional for leftover **corpus** noise, not first-party fixtures. Deeper compiler notes: Compiler overview.

Compare interpret vs compile (corpus harness)

From the repo root (monorepo with `backend/` + editable `pynescript`):

```
# Smoke: first 50 corpus scripts × 1000 synthetic bars
python scripts/compare_interp_compile.py --bars 1000 --limit 50

# Full list from a file, no script limit, 4 workers, 30s per script;
# ignore one-sided hline/fill keys when judging value parity
python scripts/compare_interp_compile.py --file-list path.txt --limit 0 --workers 4 --timeout-sec
```

The harness runs each script with `mode="interpret"` and `mode="compile"`, then nan-aware `allclose` on common `result["series"]` keys (`rtol=1e-5`, `atol=1e-6`). Report: `.cache/interp_compile_parity.json`. Exit 0 when there are no value/NaN mismatches on shared keys (`both_error_same` counts as success unless `--strict-errors`).

Interface surface

Expression evaluation

```
from pynescript.ast.helper import literal_eval

literal_eval("1 + 2 * 3")
literal_eval("math.max(1, 5, 3)")
literal_eval('str.upper("hi")')
literal_eval("array.size([1, 2, 3])")

prices = [100, 102, 101, 103, 105, 104, 106, 108, 107, 110]
literal_eval(f"ta.sma({prices}, 5)")
literal_eval(f"ta.rsi({prices}, 9)")
bb = literal_eval(f"ta.bb({prices}, 5, 2)")  # middle, upper, lower
```

Series history context (Pine-style `[0]` current / `[1]` previous as implemented):

```
context = {
    "close": [100, 102, 101, 103, 105],
    "open": [99, 101, 100, 102, 104],
    "high": [101, 103, 102, 104, 106],
    "low": [98, 100, 99, 101, 103],
}
literal_eval("close[0]", context)
literal_eval("close[0] - close[1]", context)
```

Optional live/historical wiring:

```
literal_eval(expr, context, data_feed=feed, data_provider=provider)
```

Script evaluation helper

```
from pynescript.ast.evaluator import NodeLiteralEvaluator

ev = NodeLiteralEvaluator()
result = ev.evaluate_script(
    """
//@version=6
indicator("demo")
// body depends on what the evaluator implements for statements
"""
)
```

Libraries:

```
ev.register_library_source(namespace="MyNs", name="Lib", version=1, source=lib_source)
mod = ev.lookup_library(namespace="MyNs", name="Lib", version=1)
```

Bar-loop Runtime (package SoT · HTTP contract shape)

From **0.3.4** the bar-loop host lives in the installable package:

```
from pynescript.runtime import Runtime  # SoT (backend.runtime re-exports for Pro API)

runtime = Runtime(symbol="AAPL")
ohlcv = [
    {"time": 1_700_000_000, "open": 100, "high": 101, "low": 99, "close": 100.5, "volume": 1_000},
    {"time": 1_700_086_400, "open": 100.5, "high": 102, "low": 100, "close": 101.5, "volume": 1_20
]
result = runtime.run(
    source_code='//@version=6\nindicator("t")\nplot(close)\n',
    ohlcv_data=ohlcv,
    mode="auto",  # interpret | compile | auto (see Runtime modes)
    timeout_seconds=12.0,
    # libraries=[{"namespace": "ns", "name": "Lib", "version": 1, "source": lib_src}],
)
# result: plots, series, plot_meta, events, drawings, script_id, run_id, count, ...
# auto may set auto_backend + compile_fallback_reason
# or {"error": "...", "error_kind": "parse|compile|runtime|data|order|mode", ...}
```

`backend.runtime.Runtime` remains a thin compat import for monorepo Pro API code. Dual-host TA goldens (ATR / Supertrend / Keltner) and `tests/test_ta_incremental` gate interpret↔compile numerical parity. Pro API wraps the same runtime — see Pro API usage.

Persistent interpret sessions (0.6.1)

`Runtime.run` always replays the full history. For live or incremental feeds, **`Runtime.create_session`** keeps the evaluator, series, TA state, and plot columns across calls so new closed bars only pay `visit(tree)` for the delta:

```
from pynescript.runtime import Runtime

rt = Runtime(symbol="AAPL")
session = rt.create_session(source, ohlcv_data=bars)  # interpret only
result = session.append_bars(new_closed_bars)         # O(delta)
tick = session.update_last_bar(forming_bar)           # overwrite last bar in place
```

Method	Role
`create_session(source, ohlcv, …)`	Parse once, evaluate the initial bars, return an `InterpretSession`. `mode` must be omitted or `"interpret"` (compile has no resumable evaluator state).
`append_bars(new_bars, realtime_ticks=1, …)`	Confirm the previous last bar, then visit only the appended batch. Realtime window params apply to **this batch**, not previously committed bars.
`update_last_bar(bar)`	Re-tick the forming last bar with revised OHLCV. Bar count does not advance. `var` scope and incremental TA windows roll back to bar-open (`varip` persists).

Streaming contract: already-committed bars keep their values (like TradingView® confirmed history). Scripts that branch on `barstate.islast` will diverge from a full `Runtime.run` replay for the previously last bar — that bar was committed with `islast=True`. Prefer `islastconfirmedhistory` / historical logic for append-parity, or re-run full `Runtime.run` when exact `islast` replay is required. `PYNE_SERIES_RING` must stay off (the default). Tests: `tests/test_runtime_session.py`.

Data providers and feeds

Historical CLI/library:

```
from pynescript.util.data import get_provider

prov = get_provider("mock")
# or yahoo / alphavantage / ccxt with kwargs
bars = prov.fetch("AAPL", "6mo", "1d")
# bars: dict with close/open/... lists
```

Realtime (requires ccxt / pro):

```
# examples/realtime_datafeed.py
from pynescript.util.datafeed import get_datafeed

feed = get_datafeed("ccxtpro", exchange="binance")
# async with feed: async for candle in feed.watch_ohlcv("BTC/USDT", "1m"): ...
```

Educational bar executor

`examples/execute_script.py` implements a **teaching** RSI strategy executor with a custom visitor and pandas history (`examples/historical_data.py` / yfinance). It is not the production `Runtime`, but demonstrates bar iteration patterns.

Internals (repo paths)

Path	Role
`src/pynescript/ast/helper.py`	`literal_eval`
`src/pynescript/ast/evaluator/base.py`	Context, series, visitor base
`src/pynescript/ast/evaluator/__init__.py`	`NodeLiteralEvaluator` composition
`src/pynescript/ast/evaluator/builtins/`	`ta.*` , `strategy.*` , arrays, …
`src/pynescript/ast/evaluator/events.py`	Event emission hooks
`src/pynescript/runtime/host.py`	Multi-bar `Runtime.run` SoT (`interpret` / `compile` / `auto` , `timeout_seconds`) plus `InterpretSession` (`create_session` / `append_bars` / `update_last_bar`)
`src/pynescript/runtime/series.py`	`PineSeries` history model
`backend/runtime.py` / `backend/series.py`	Compat re-exports of package Runtime
`src/pynescript/compiler/`	Transpile + Numba/object bar loops
`src/pynescript/util/data.py`	Providers + `resolve_request_sources`
`src/pynescript/util/datafeed.py`	Realtime feeds
`scripts/compare_interp_compile.py`	Interpret vs compile series parity harness
`tests/test_interp_compile_parity.py`	Always-on smoke + optional full mark
`examples/evaluate_expressions.py`	Expression gallery
`examples/execute_script.py`	Didactic strategy loop
`examples/rsi_strategy.pine`	Sample strategy source

Invariants & edge cases

- **Determinism.** Same source + same OHLCV + same mode should yield reproducible series (mock providers seedable via options where supported).
- **`na` / warmup.** TA functions need enough bars; early bars may be `na` /None-like — guard like Pine (`not na(x)`).
- **`mode="compile"`** uses numeric Numba when the script is pure-numeric; otherwise **object-mode** compile. Strict `compile` errors instead of falling back; use **`auto`** for production “prefer fast path” behavior (see Runtime modes).
- **`mode="auto"`** may set `compile_fallback_reason` and still return a successful interpret result — check `auto_backend` if you need to know which path ran.
- **Series parity.** Shared `series` keys should match within harness tolerances. First-party `hline` / `fill` / `bgcolor` / `plotshape` / `barcolor` / `plotarrow` / `plotbar` / `plotcandle` keys match; `drawings` is geometry-only. Ignore flags are for leftover corpus noise.
- **History indexing.** Confirm `[0]` / `[1]` semantics against your evaluator version before porting TV scripts that assume closed-bar rules.
- **Strategy events** include run/script ids when stamped by Runtime — useful for HOOX downstream.

- **Body size / bar count.** HTTP path caps payload size (5 MiB); very long histories should be downsampled or chunked.
- **InterpretSession.** `create_session` is interpret-only. `append_bars` confirms the previous last bar; scripts that branch on `barstate.islast` are not replay-identical to a full `Runtime.run` for that bar. `PYNE_SERIES_RING` must stay off.

Worked examples

RSI on a synthetic series

```
from pynescript.ast.helper import literal_eval

closes = [float(x) for x in range(100, 130)]
print(literal_eval(f"ta.rsi({closes}, 14)"))
```

Multi-indicator expressions

```
highs = [c + 1 for c in closes]
lows = [c - 1 for c in closes]
macd, signal, hist = literal_eval(f"ta.macd({closes}, 12, 26, 9)")
atr = literal_eval(f"ta.atr({highs}, {lows}, {closes}, 14)")
```

Full script via Runtime

```
from pathlib import Path
from pynescript.runtime import Runtime

script = Path("examples/rsi_strategy.pine").read_text(encoding="utf-8")
# Build ohlcv list from your provider...
rt = Runtime(symbol="EXAMPLE")
out = rt.run(script, ohlcv_data=ohlcv, mode="interpret")
if "error" in out:
    raise RuntimeError(out["error"])
print(out.get("count"), len(out.get("events", [])), out.get("series", {}).keys())
```

Strategy event inspection

```
for ev in out.get("events", []):
    # shape depends on StrategyState / event schema
    print(ev)
```

request.* with resolved sources

When calling `/run` or Runtime, pass `data_source` / providers so `request.security` and friends resolve; without configuration they may use chart bars or mocks.

Failure modes

Symptom	Interpretation
`NotImplementedError` / incomplete builtin	Expression outside supported surface — check missing features
`{"error": "Parse Error: ..."}`	Runtime caught parse failure
`{"error": …}` with `mode="compile"`	Strict compile path failed — retry with `auto` or `interpret`, or fix unsupported construct
`compile_fallback_reason` set under `auto`	Compile skipped/failed; result is from interpret if no top-level `error`
`EXECUTION_ERROR` via HTTP	Exception during bar loop
Empty plots	No `plot` / `plotshape` executed; wrong declaration type
Numerical mismatch vs TV	Warmup, float path, or builtin parity gap — numerical validation
Interpret vs compile series MISMATCH	Run `scripts/compare_interp_compile.py` on the script; check report buckets
`timed_out` + partial plots	`timeout_seconds` budget exceeded on interpret
`create_session supports interpret only`	`mode="compile"` / `"auto"` passed to `create_session` — sessions are interpret-only
`UNKNOWN_FIELDS`	Extra key not in `RUN_SCHEMA` / `RUN_BATCH_SCHEMA`
`DataProviderError`	Provider misconfig
Async feed errors	Missing ccxt.pro / network

See also

- Pro API usage
- Runtime modes
- Library API
- Runtime series model
- Compiler overview
- Strategy builtins
- Evaluate contract
- AXIS — visualizing series
- HOOX — consuming strategy events on the edge

EDITORS & LSP

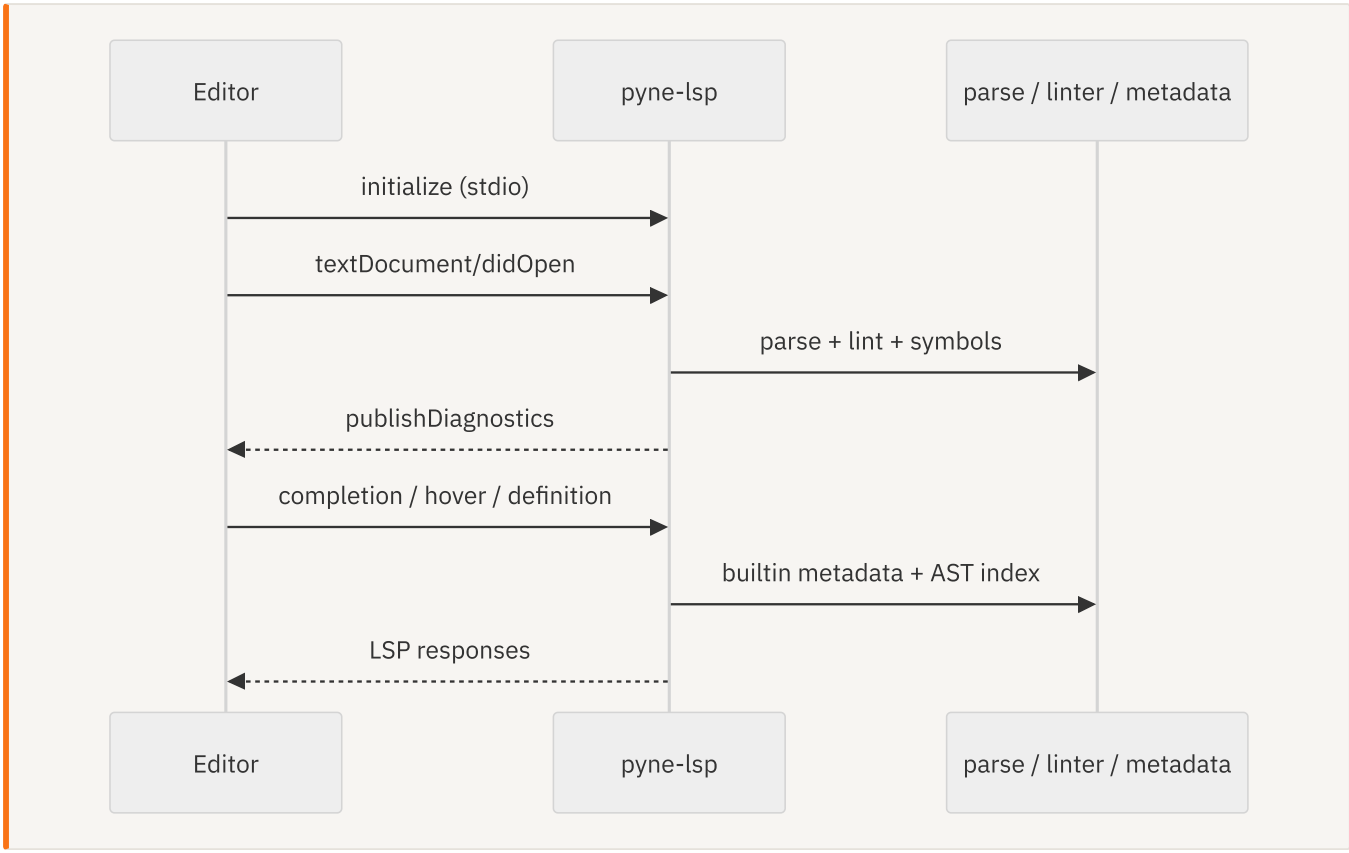
Wire pyne-lsp into VS Code, Neovim, Zed, Emacs, Helix, and other LSP clients for diagnostics, completion, hover, and formatting.

EDITORS & LSP

Abstract

`pyne-lsp` (alias `pynescript-lsp`) is a **stdio Language Server** (pygls) over the same parse/lint/metadata stack as the desk tools. Editors do not reimplement Pine Script™ intelligence; they speak LSP. This guide covers install, client configs in `clients/`, and feature expectations. Server architecture is documented under LSP.

Conceptual model



Feature	Role
Diagnostics	Lint rules + parse errors as squiggles
Completion	Builtins (`ta.*` , `strategy.*` , …) + snippets
Hover	Signature / docs from metadata
Definition / references	Symbol index in document
Document symbols	Outline / breadcrumb
Formatting	Full document + range via unparser path

Builtin metadata is generated (`scripts/generate_builtin_metadata.py`); encrypted blobs may appear in release builds.

Interface surface

Install server

```
pip install "hoox-pyne[lsp]"
# monorepo:
pip install -e ".[lsp]"
# or: make install

command -v pyne-lsp          # preferred
command -v pynescript-lsp    # alias
python -m pynescript.langserver  # equivalent entry
```

The process speaks **LSP over stdio** by default (`PynescriptLanguageServer.start_io()`). Do not expect a useful interactive TTY UI.

VS Code / Cursor / Antigravity

1. Build or install the extension from `vscode-extension/` :

```
cd vscode-extension
npm ci
npm run compile
# package: npx vsce package  (or make build-vscode)
```

2. Open a **`.pyne`** (preferred for HOOX / PYNE stack sources), `.pine` , `.pinev5` , `.pinev6` , or `.pinescript` file — LSP activates when enabled. The VS Code extension also activates on `workspaceContains:**/*.{pyne,pine,…}` .

Marketplace / VSIX: extension id **`hoox-sh.pyne`** (publisher `hoox-sh` , package name `pyne`). VS Marketplace. Site download: hoox.sh/pyne-vscode.vsix (unversioned alias). Local package: `cd vscode-extension && npm ci && npm run package` then `code --install-extension pyne-vscode-0.4.4.vsix` . The VSIX is self-contained; you still need `pip install "hoox-pyne[lsp]"` (or a `pyne-lsp` binary) on the machine.

Settings (from `vscode-extension/package.json`):

Key	Default	Notes
`pynescript.lsp.enabled`	`true`	Master switch
`pynescript.lsp.command`	`auto`	`auto` → `pyne-lsp` → `pynescript-lsp` → `python3 -m pynescript.langserver` ; or an absolute path / `docker`
`pynescript.lsp.python`	`python3`	Interpreter when command is `auto` and no binary is on PATH
`pynescript.lsp.args`	`[]`	Extra server args
`pynescript.formatting.enabled`	`true`	
`pynescript.diagnostics.enabled`	`true`	
`pynescript.completion.snippets`	`true`	

Default formatter id: `hoox-sh.pyne`.

Neovim (nvim-lspconfig)

```
require("lspconfig").pynescript.setup({})
```

Manual config from `clients/neovim.lua`:

```
return {
  cmd = { "pyne-lsp" },  -- alias: pynescript-lsp
  filetypes = { "pinescript" },
  root_dir = function(fname)
    return vim.fs.root(fname, { ".git", "*.pyne", "*.pine", "*.pinev5", "*.pinev6", "pyproject.tom
      or vim.fn.getcwd()
  end,
  settings = {
    pinescript = {
      formatting = { enabled = true },
      diagnostics = { enabled = true },
      completion = { snippets = true },
    },
  },
}
```

Suggested maps: `gd` definition, `gr` references, `K` hover, format via `vim.lsp.buf.format`.

Ensure filetype detection maps `*.pyne` / `*.pine` → `pinescript` if your distro does not ship it.

Zed

Merge `clients/zed.json` or:

```
{
  "languages": {
    "Pine Script": {
      "language_servers": ["pynescript"]
    }
  },
  "language_servers": {
    "pynescript": {
      "command": "pyne-lsp",
      "arguments": ["--stdio"]
    }
  }
}
```

Emacs (lsp-mode)

```
(use-package lsp-mode
  :hook ((pinescript-mode . lsp))
  :config
  (lsp-register-client
   (make-lsp-client
    :new-connection (lsp-stdio-connection '("pyne-lsp" "--stdio"))
    :major-modes '(pinescript-mode)
    :server-id 'pynescript)))
```

Or `(load-file "clients/emacs.el")` from a checkout.

Helix

`~/.config/helix/languages.toml`:

```
[[language]]
name = "pinescript"
scope = "source.pinescript"
file-types = ["pyne", "pine", "pinev5", "pinev6"]
roots = ["pyproject.toml"]
command = "pyne-lsp"
args = ["--stdio"]
```

Sublime Text (LSP package)

```
{
  "clients": {
    "pynescript": {
      "command": ["pyne-lsp", "--stdio"],
      "selector": "source.pinescript",
      "initializationOptions": {}
    }
  }
}
```

Generic

Any LSP client:

```
Command: pyne-lsp          # alias: pynescript-lsp
Args:    (none or --stdio depending on client)
Transport: stdio
```

Internals (repo paths)

Path	Role
`src/pynescript/langserver/__main__.py`	CLI entry `main()`
`src/pynescript/langserver/server.py`	`PynescriptLanguageServer`
`src/pynescript/langserver/features/`	diagnostics, completion, hover, ...
`clients/README.md`	Canonical client matrix
`clients/neovim.lua`, `zed.json`, `emacs.el`	Drop-in snippets
`vscode-extension/`	VS Code extension
`scripts/generate_builtin_metadata.py`	Completion/hover corpus

Invariants & edge cases

- **Python 3.10+** required on the machine running the server.
- **Server must be on PATH** or configured with absolute command.
- **One server per workspace** is typical; large monorepos with many `.pine` files still parse on demand.
- **Formatting** uses the unparser pipeline — expect normalization, not clang-format style knobs.
- **Diagnostics** reuse linter codes (`E001`, `W001`, ...); severity maps to LSP DiagnosticSeverity.
- **Nuitka binary** (`dist/lsp/pynescript-lsp`) may replace the Python entry in packaged distributions — point the editor at that binary if you ship it.

Worked examples

Verify server starts (stdio smoke)

```
# Editors start this; manual smoke is limited.
python -c "from pynescript.langserver.server import PynescriptLanguageServer; print(PynescriptLang
```

Neovim minimal `init.lua` fragment

```
vim.api.nvim_create_autocmd("FileType", {
  pattern = "pinescript",
  callback = function()
    vim.lsp.start({
      name = "pynescript",
      cmd = { "pyne-lsp" },
      root_dir = vim.fn.getcwd(),
    })
  end,
})
```

Force a specific venv binary in VS Code

```
{
  "pynescript.lsp.command": "/home/you/project/.venv/bin/pyne-lsp"
}
```

Failure modes

Symptom	Fix
Client: executable not found	Install `[lsp]`; fix PATH / command setting
`No module named pygls`	Reinstall extra in the same env as the command
No diagnostics	Enable diagnostics; confirm filetype; check server logs
Stale completions	Regenerate metadata in dev builds; restart server
Format does nothing	Enable formatting; ensure document is valid enough to parse
High CPU on huge files	Split libraries; wait for parse; check for pathological nesting

See also

- Installation
- LSP architecture
- VS Code extension
- LSP clients (systems)
- Troubleshooting

PRO API USAGE

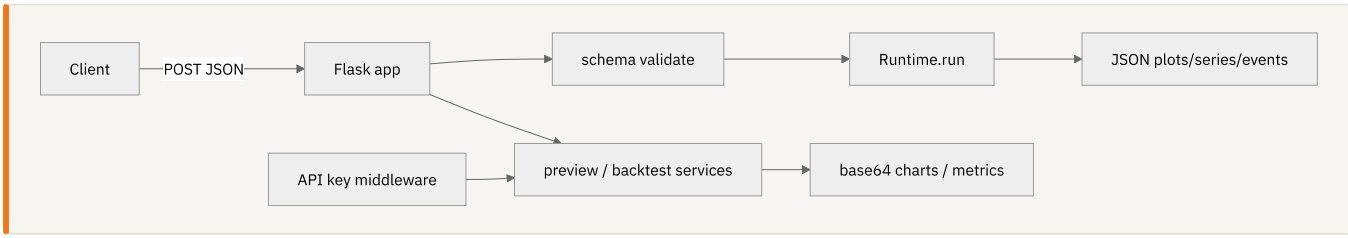
Call the PYNE Flask Pro API as a consumer: health, /run, /run/batch, /optimize, chart preview, indicator preview, quick backtest, and auth.

PRO API USAGE

Abstract

The Pro API is the **HTTP evaluate contract** for PYNE: POST Pine source plus OHLCV, receive plots, series, events, drawings, and **alerts**. Free-tier evaluate endpoints (`/run` , `/run/batch`) are usable without a key. Bar/script/rate/concurrency and chart/mock-only data guards apply **only when** `FREE_TIER_LIMITS` is truthy (`1` / `true` / `yes` / `on`); default **off**. Production compose sets `${FREE_TIER_LIMITS:-1}` . Webhook URLs stay SSRF-safe regardless. Optional **L2 alert webhooks** POST last-bar firings to `webhook_url` or `ALERT_WEBHOOK_URL` . Pro preview/backtest routes track usage via API keys. This page is for **callers** (AXIS, scripts, workers). Server lifecycle and auth design are under API.

Conceptual model



Endpoint	Auth	Purpose
`GET /` / `GET /health`	none	Health + endpoint map (`default_run_mode` : `auto`)
`POST /run`	none (free)	Single script evaluate
`POST /run/batch`	none (free)	≤8 scripts, shared OHLCV
`POST /optimize`	none (free)	Strategy hyperparameter search (docs)
`POST /compile/prewarm`	none (free)	Warm Numba builtins / script IR
`WS /ws/run`	none (free)	AXIS WebSocket evaluate channel
`POST /preview/chart`	Pro usage	Chart thumbnail
`POST /preview/indicator`	Pro usage	Indicator-style preview
`POST /backtest/quick`	Pro usage	Equity / summary
`POST /auth/create_key`	admin token	Mint key
`GET /auth/usage`	Pro	Usage stats
`POST /auth/validate`	body key	Validate key
`POST /lsp/{completion,hover,diagnostics}`	none (free)	AXIS HTTP LSP bridge — see LSP
`/scripts` · `/cron/*`	`PYNE_RUNNER=1`	Hosted scripts + bar-close (runner)

Default local bind: `127.0.0.1:5002` .

Interface surface

Run the server (local)

```
pip install -r backend/requirements.txt

python -m backend.app
# or: make run
```

Health

```
curl -s http://127.0.0.1:5002/ | python -m json.tool
```

`POST /run`

Schema (`RUN_SCHEMA`):

Field	Type	Required	Default
`script`	string	yes	
`data`	list (OHLCV bars)	yes	
`symbol`	string	no	`"CHART"`
`data_source`	string	no	`""`
`data_options`	object	no	`{}`
`mode`	string	no	`"auto"`
`inputs`	object	no	`{}` — `input.*` by title (forces interpret under `auto`)
`libraries`	list	no	`[]` — `[{namespace, name, version, source}]`, max 32
`timeout_seconds`	number	no	omit — interpret wall-clock budget; passed only when set and `> 0`
`profiler`	bool	no	`false` — forces interpret
`webhook_url`	string	no	`""` (else env `ALERT_WEBHOOK_URL`)
`forward_alerts`	bool	no	`true`
`alert_last_bar`	bool	no	`true`
`alert_batch`	bool	no	`true`

Unknown extra keys → `UNKNOWN_FIELDS`. Query `?mode=` overrides only when the body omits `mode`.

Bar objects are expected to carry at least open/high/low/close/time (volume optional depending on script).

```
curl -s -X POST http://127.0.0.1:5002/run \
  -H 'Content-Type: application/json' \
  -d '{
    "script": "//@version=6\nindicator(\"Demo\")\nplot(ta.sma(close, 3))",
    "symbol": "AAPL",
    "mode": "interpret",
    "data": [
      {"time": 1, "open": 10, "high": 11, "low": 9, "close": 10.5, "volume": 1000},
      {"time": 2, "open": 10.5, "high": 12, "low": 10, "close": 11.5, "volume": 1100},
      {"time": 3, "open": 11.5, "high": 12.5, "low": 11, "close": 12, "volume": 1200}
    ]
  }'
```

Success shape (conceptual):

```
{
  "status": "success",
  "plots": [],
  "series": {},
  "plot_meta": {},
  "events": [],
  "drawings": [],
  "alerts": [],
  "script_id": "...",
  "run_id": "...",
  "count": 3,
  "mode": "interpret",
  "data_source": "chart"
}
```

When a webhook URL is configured, the response may also include `alert_forward` (delivery counts). See Alerts.

Error codes include `NO_SCRIPT`, `NO_DATA`, `DATA_SOURCE_ERROR`, `EXECUTION_ERROR`, plus schema codes `INVALID_BODY`, `MISSING_FIELD`, `INVALID_FIELD`, `UNKNOWN_FIELDS`.

`POST /run/batch`

Shared OHLCV, multiple scripts (max **8**). Envelope:

```
{
  "scripts": [
    {"id": "sma", "script": "//@version=6\nindicator(\"s\")\nplot(ta.sma(close, 5))"},
    {"id": "rsi", "script": "//@version=6\nindicator(\"r\")\nplot(ta.rsi(close, 14))"}
  ],
  "data": [/* bars */],
  "symbol": "AAPL",
  "mode": "interpret",
  "libraries": []
}
```

String entries in `scripts` are accepted and auto-id'd as `script_0`, ... Per-script errors do not necessarily fail the whole HTTP status (route returns 200 with per-item status — verify against deployment).

`POST /preview/chart` (Pro)

```
{
  "script": "",
  "data": {"close": [100, 101, 102], "volume": [1, 2, 3]},
  "options": {
    "type": "line",
    "color": "#2196F3",
    "width": 600,
    "height": 300,
    "show_volume": false
  }
}
```

Response includes base64 PNG `chart` + `meta`. Width/height clamped (e.g. max 1200×600).

`POST /preview/indicator` (Pro)

```
{
  "expression": "ta.sma(close, 20)",
  "data": {"close": [/* ... */]},
  "options": {}
}
```

`POST /backtest/quick` (Pro)

```
{
  "script": "//@version=6\nstrategy(\"x\")\n...",
  "data": {},
  "initial_capital": 10000.0,
  "mock_data": true,
  "mock_bars": 252
}
```

`POST /compile/prewarm` (free)

Warm Numba builtins / optional script list so the first interactive `/run` with `mode=auto|compile` skips cold JIT. Soft-fails without Numba (object-mode compile still works).

```
curl -s -X POST http://127.0.0.1:5002/compile/prewarm \
  -H 'Content-Type: application/json' \
  -d '{"force": true}' | python -m json.tool
```

Optional body: `scripts` (list of Pine sources to compile-warm), `force` (re-run builtins). Response includes timing / prewarm stats. Deploy hosts may also set `PYNE_COMPILE_PREWARM` for once-per-worker start. See Compiler overview.

Auth helpers

```
# Mint key (admin)
curl -s -X POST http://127.0.0.1:5002/auth/create_key \
  -H "Content-Type: application/json" \
  -H "X-Admin-Token: $ADMIN_TOKEN" \
  -d '{"tier":"hobby"}'

# Validate
curl -s -X POST http://127.0.0.1:5002/auth/validate \
  -H "Content-Type: application/json" \
  -d '{"api_key":"pyn_..."}'
```

Send Pro requests with the key as required by middleware (`Authorization: Bearer …` / documented API-key header — see Auth and keys).

There is **no** `pynescript.api.PynescriptAPI` client in this package. Call the HTTP contract with `requests` / `curl` as above.

Internals (repo paths)

Path	Role
`backend/app.py`	Routes `/`, `/run`, `/run/batch`, `/compile/prewarm`, auth
`backend/api/preview.py`	`/preview/*`, `/backtest/quick`
`backend/middleware/schemas.py`	Strict request schemas (`mode` default `auto`, alert flags)
`backend/middleware/auth.py`	Keys, usage, admin token
`src/pynescript/runtime/host.py`	Package Runtime SoT (`interpret` / `compile` / `auto`)
`backend/runtime.py`	Compat shim → `pynescript.runtime`
`backend/alert_forwarder.py`	L2 webhook POST batching
`backend/services/chart_renderer.py`	PNG rendering
`backend/services/backtest.py`	Quick backtest + mock OHLCV

Invariants & edge cases

- **Max body 5 MiB** — oversized JSON → Flask 413.
- **Strict schemas** — unknown fields rejected.
- **CORS** — browser clients need listed origins; server-to-server without `Origin` is fine.
- **`mode` default `auto`** — prefer warm compile; fall back to interpret (`auto_backend`, optional `compile_fallback_reason`). Strict `compile` errors instead of falling back; use `interpret` for alerts, `input.*` overrides, libraries, and full `request.*`.
- **`libraries` (0.3.7+)** — `[{namespace, name, version, source}]` binds `import ns/Name/ver` (max 32). `register_library_source` finalizes exports in 0.3.8; auto-mode keeps the list on interpret fallback. Same field on Flask `/run`, `/run/batch`, pyne-worker `/run`, and deployed scripts.
- **`timeout_seconds`** — optional interpret wall-clock budget on `/run` and `/run/batch`. Omit / `null` / `≤ 0` → no timeout. Exceeded runs surface `timed_out` on the HTTP body.
- **Alerts** — `alerts[]` on success (interpret). L2 webhooks: body `webhook_url` or env `ALERT_WEBHOOK_URL`; defaults last-bar + batch POST. See Alerts.
- **Batch cap 8** — `TOO_MANY_SCRIPTS` if exceeded.
- **Free-tier guards (0.3.4+; opt-in)** — process-local limits on unauthenticated `/run`, `/run/batch`, `/compile/prewarm` (and WS run). **Off unless** `FREE_TIER_LIMITS` is `1` / `true` / `yes` / `on`. Production compose sets that to `1`.

Env	Default	Role
`FREE_TIER_LIMITS`	off	Master switch for the rows below
`FREE_MAX_BARS`	5000	Max OHLCV bars (when on)
`FREE_MAX_SCRIPT_CHARS`	256 KiB	Max Pine source length (when on)
`FREE_MAX_CONCURRENT`	4	Simultaneous free runs per worker (when on)
`FREE_RATE_LIMIT` / `FREE_RATE_WINDOW_SEC`	60 / 60s	Sliding-window IP rate limit (when on)
Free `data_source`	`chart` / `mock` / `none` only	No outbound ccxt/yahoo on free paths (when on)
Numeric knobs treat `0` as unlimited. Health `features.free_tier_limits` reports the master switch. Webhook URLs stay SSRF-filtered regardless. Put a reverse proxy in front for multi-worker production.		

- AXIS multi-indicator UIs prefer `/run/batch` to share bar payloads; read `series` + `plot_meta` (including fill band refs) for charts.

Worked examples

Python `requests` client for `/run`

```
SCRIPT = """
//@version=6
indicator("SMA", overlay=true)
plot(ta.sma(close, 5))
"""

bars = [
    {"time": i, "open": 100 + i, "high": 101 + i, "low": 99 + i, "close": 100.5 + i, "volume": 100
    for i in range(30)
]

r = requests.post(
    "http://127.0.0.1:5002/run",
    json={"script": SCRIPT, "data": bars, "symbol": "DEMO", "mode": "auto"},
    timeout=60,
)
r.raise_for_status()
payload = r.json()
assert payload["status"] == "success", payload
print(payload["count"], payload.get("mode"), list(payload.get("series", {})))
# optional: payload.get("alerts"), payload.get("alert_forward")
```

Batch two indicators

```
r = requests.post(
    "http://127.0.0.1:5002/run/batch",
    json={
        "scripts": [
            {"id": "sma", "script": "//@version=6\nindicator(\"s\")\nplot(ta.sma(close, 5))"},
            {"id": "ema", "script": "//@version=6\nindicator(\"e\")\nplot(ta.ema(close, 5))"},
        ],
        "data": bars,
    },
    timeout=120,
)
print(r.json())
```

Bind published libraries

```
r = requests.post(
    "http://127.0.0.1:5002/run",
    json={
        "script": '//@version=6\nimport Demo/Lib/1 as L\nindicator("t")\nplot(L.x)\n',
        "data": bars,
        "mode": "interpret",
        "libraries": [
            {
                "namespace": "Demo",
                "name": "Lib",
                "version": 1,
                "source": '//@version=6\nlibrary("Lib")\nexport x = 1.0\n',
            }
        ],
    },
    timeout=60,
)
```

Wire optional CCXT data source

```
{
  "script": "...",
  "data": [/* chart bars */],
  "symbol": "BTC/USDT",
  "data_source": "ccxt",
  "data_options": {"exchange": "binance"}
}
```

Requires server environment with `ccxt` installed.

Failure modes

Code / HTTP	Meaning	Action
400 `MISSING_FIELD`	No `script` / `data`	Fix body
400 `UNKNOWN_FIELDS`	Extra keys	Strip undocumented fields
400 `DATA_SOURCE_ERROR`	Provider config	Fix `data_options` / deps
500 `EXECUTION_ERROR`	Runtime exception	Simplify script; check message
403 on create_key	`ADMIN_TOKEN` unset/wrong	Export token
413	Body too large	Fewer bars / compress strategy
CORS browser error	Origin not allowed	Update `ALLOWED_ORIGINS`
Empty series	Script never plotted	Add `plot` / check declaration

See also

- Evaluate scripts
- Runtime modes
- Configuration
- API contract
- Auth and keys
- AXIS — AXIS over `/run` results
- HOOX — edge mesh after strategy events

TROUBLESHOOTING

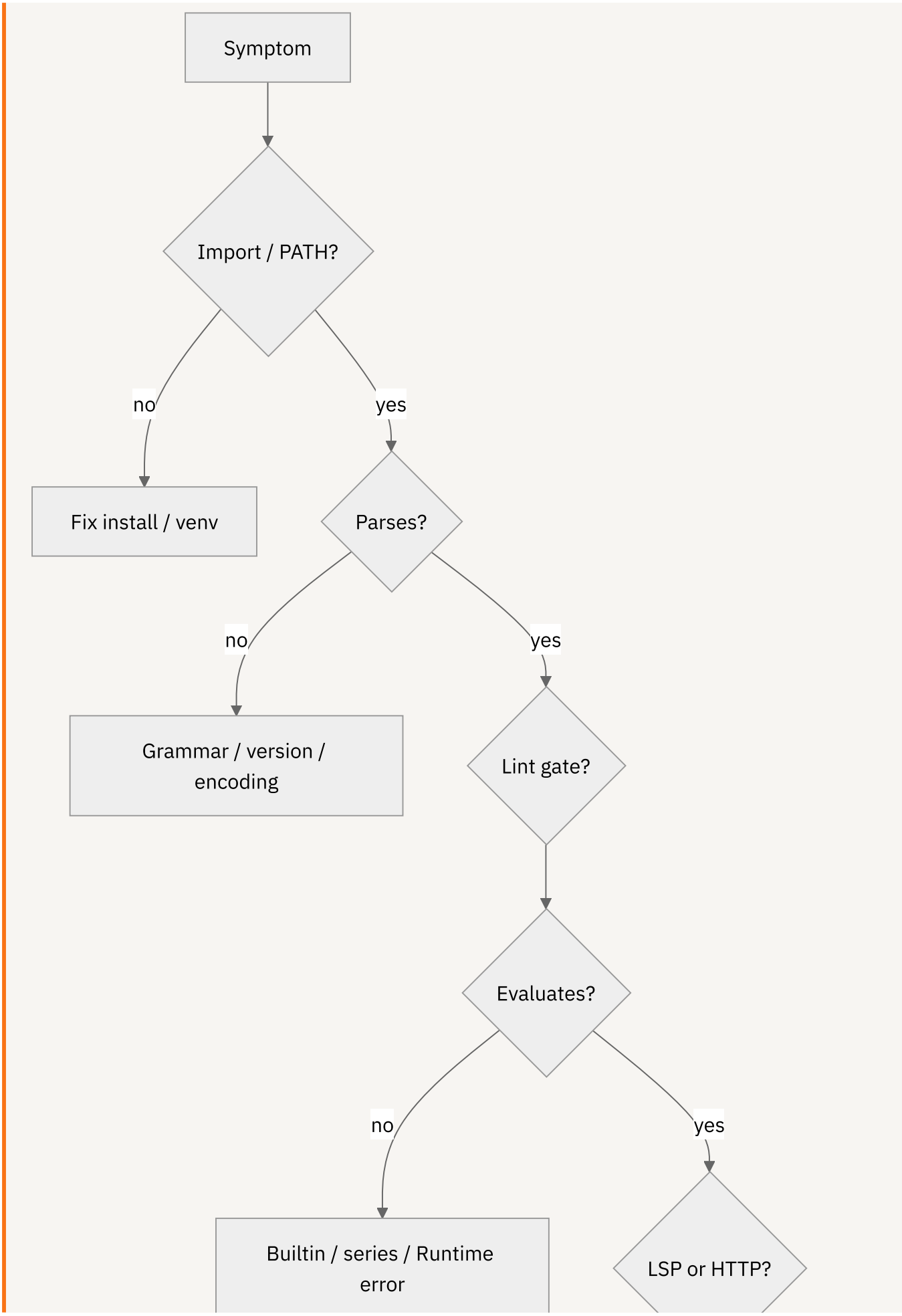
Diagnose pynescript install, parse, lint, evaluation, LSP, data providers, and Pro API failures with concrete checks.

TROUBLESHOOTING

Abstract

Failures cluster by layer: **environment**, **parse**, **lint**, **evaluation**, **data I/O**, **LSP**, **HTTP**. Work top-down: confirm the binary/module, then a minimal script, then full strategy complexity. This page is a runbook; deep protocol and runtime docs are linked per section.

Conceptual model


Symptom
Import / PATH?
no
yes
Fix install / venv
Parses?
no
yes
Grammar / version / encoding
Lint gate?
Evaluates?
no
yes
Builtin / series / Runtime error
LSP or HTTP?

Interface surface (diagnostic commands)

```
python -c "import pynescript; from pynescript import __about__; print(__about__.__version__)"
python -c "from pynescript.ast import parse, unparse; print(unparse(parse('//@version=6\nindicator
pyne --version
pyne lint - <<< $'//@version=6\nindicator("t")\nplot(close)\n'
python -c "import pygls"  # LSP extra
python -c "import ccxt"   # data extra
curl -s http://127.0.0.1:5002/ | head
```

Internals (where errors originate)

Layer	Path
CLI	`src/pynescript/__main__.py`
Parse	`src/pynescript/ast/helper.py`, ANTLR error listener
Lint	`src/pynescript/ast/linter.py`
Eval	`src/pynescript/ast/evaluator/`, `src/pynescript/runtime/`
Data	`src/pynescript/util/data.py`
LSP	`src/pynescript/langserver/`
HTTP	`backend/app.py`, `backend/middleware/schemas.py`

Invariants & edge cases

- Minimal green script for isolation:

```
//@version=6
indicator("t")
plot(close)
```

- Encoding must be UTF-8 unless you pass `--encoding`.
- Pine v4 and earlier trigger linter deprecation (`W002`); parser support may still differ by construct.
- Recursion limit is raised only during parse; extreme generated scripts can still fail.

Worked examples (by symptom)

1. `pyne` / `pynescript: command not found`

```
which python
python -m pynescript --help
# If module works but script does not:
python -m pip show hoox-pyne
ls "$(python -c 'import sysconfig; print(sysconfig.get_path("scripts"))')"
```

Fix: activate the venv used for install; reinstall `pip install hoox-pyne` (import package is still `pynescript`). Do **not** `pip show pynescript` — that is a different PyPI project.

2. `ModuleNotFoundError: pynescript`

Wrong interpreter (IDE using system Python). Point the IDE at the venv; reinstall editable from repo root: `pip install -e .`.

3. Parse / SyntaxError

```
pyne parse-and-dump broken.pine
```

Checks:

- `//@version=6` present (v5 still accepted)
- Balanced parentheses / indentation (Pine uses line structure)
- Multiline strings / v6 features need a package version that includes them
- Binary/null bytes in file

Reduce to a minimal repro; if TV accepts it and PYNE does not, consult missing features and file a grammar issue.

4. Round-trip looks “wrong”

`unparse` normalizes formatting. Compare dumps:

```
pyne parse-and-dump a.pine > /tmp/a.ast
pyne parse-and-unparse a.pine | pyne parse-and-dump /dev/stdin
# stdin may not work for parse-and-dump (file path required) — use a temp file:
pyne parse-and-unparse a.pine --output-file /tmp/a2.pine
pyne parse-and-dump /tmp/a2.pine > /tmp/a2.ast
diff -u /tmp/a.ast /tmp/a2.ast
```

Structural diff empty ⇒ cosmetic only.

5. Lint noise / CI red

Code	Action
`E001`	Fix syntax first
`W001`	Add `//@version=6`
`W002`	Upgrade version pragma
`C001` – `C004`	Style; use `--fail-on errors` if style should not gate CI

```
pyne lint script.pine --fail-on errors
```

6. `literal_eval` fails

- Use `mode="eval"`-compatible **expressions**, not full `strategy()` scripts
- Pass series as Python lists inside the expression or via `context`
- Missing builtin → incomplete implementation

Escalate to `Runtime.run` for multi-bar scripts.

7. Runtime / `/run` execution error

```
curl -s -X POST http://127.0.0.1:5002/run -H 'Content-Type: application/json' -d '{"script":"//@ve
```

If minimal works but strategy fails: remove `request.*`, drawings, then strategy entries until isolated. Check `mode`; try `"interpret"`. Library `Runtime.run` defaults to **interpret**; `POST /run` defaults to **auto**; `pyne run` is compile-only synthetic smoke — they will not match each other unless you set `mode` explicitly. See modes.

8. Schema validation errors

`UNKNOWN_FIELDS` means you sent a key not in the schema — remove it. Do not rely on servers ignoring extras.

9. CORS from AXIS / browser

```

```

Restart Flask after changing env.

10. LSP will not start

```
pip install "hoox-pyne[lsp]"
python -c "import pygls, lsprotocol"
command -v pyne-lsp   # alias: pynescript-lsp
# Run with editor logging enabled; check stderr for import traces
```

Ensure the editor's `command` matches the venv. For Neovim, confirm `filetypes` include your buffer's filetype.

11. Data provider failures

Provider	Checklist
`mock`	Should always work offline
`yahoo`	Network; symbol format
`alphavantage`	API key; demo limits
`ccxt`	`pip install "hoox-pyne[data]"`; exchange id; symbol `BTC/USDT`

```
pyne data AAPL --provider mock
```

12. Version / parity confusion

PYNE is independent of TradingView®. Behavioral gaps are expected for edge builtins, broker emulators, and UI-only calls. Use compatibility and implementation status.

Failure modes (quick matrix)

Symptom	Layer	First check
Command not found	env	`python -m pynescript`
Import error pygls	extra	`[lsp]`
Import error ccxt	extra	`[data]`
SyntaxError	parse	minimal script + version
Lint CI fail	lint	`--fail-on` level
Empty plots	eval	`plot()` present; enough bars
`pyne run` ≠ `Runtime.run`	modes	CLI is compile-only synthetic; library default is interpret
`UNKNOWN_FIELDS` + `timeout_seconds`	HTTP	field is valid on `/run` and `/run/batch` (0.3.11+); other extra keys still 400
400 UNKNOWN_FIELDS	HTTP	schema
403 create_key	HTTP	`ADMIN_TOKEN`
413	HTTP	body size
Squiggles missing	LSP	diagnostics enabled + filetype
Numerical drift	eval	warmup / parity docs

See also

- Installation
- Configuration
- CLI guide
- Evaluate scripts
- Pro API usage

- Editors
- FAQ

CLI COMMANDS REFERENCE

Complete option reference for the pyne Click CLI: check, format, lint, compile, run, optimize, prewarm, data, info, and more.

CLI COMMANDS REFERENCE

Abstract

Machine-oriented reference for every command on the preferred **`pyne`** Click group (alias **`pynescript`**; `src/pynescript/__main__.py`). Install via **`hoox-pyne`** (`pip install hoox-pyne`). For workflows see the CLI guide. The language server is a **separate** console script (`pyne-lsp`; alias `pynescript-lsp`).

Version option: `pyne --version` / `-V` reports `pynescript.__about__.__version__` (e.g. `0.3.14`). Global `--no-color` honors `NO_COLOR`.

Command index

Command	Aliases	Purpose
`check`		Parse-only validation (CI-friendly exit codes)
`format`	`fmt`	Parse → unparse (structural format)
`convert`		Rewrite older Pine toward v5 or v6
`parse-and-dump`	`dump`, `ast`	AST dump
`parse-and-unparse`	`unparse`	Round-trip source
`lint`		Static diagnostics (JSON / fail-on)
`compile`		Transpile or compile-check (Numba path)
`prewarm`		Warm Numba builtins / script IR caches (H2)
`run`		Compile + execute on synthetic OHLCV
`optimize`		Search `input.*` values over N interpret Runtime runs
`data`		Fetch market bars
`runner`		Hosted scripts on the Pro API (`deploy` / `tick` / `list`)
`info`	`ls`	Version + optional extras

```
pyne --help
pyne --version
pyne <command> -h
# alias: pynescript …
```

`pyne check`

Validate that path(s) parse as Pine Script.

Kind	Name	Default	Description
argument	`PATHS…`	(optional)	Files and/or directories; omit or `-` → stdin
option	`--encoding`	`utf-8`	Text encoding
option	`-q, --quiet`	off	Exit code only
option	`--ext`	`.pine`	When PATH is a directory, only this suffix

```
pyne check script.pine
pyne check scripts/ --ext .pyne
pyne check -q strategy.pine   # exit 0/1 only
cat x.pine | pyne check -
```

Exit: `0` all ok; `1` any parse failure.

`pyne format` (`fmt`)

Canonicalize Pine by parse → unparse (structural, not a full style formatter).

Kind	Name	Default	Description
argument	`PATH`	required	File or `-` (stdin)
option	`--encoding`	`utf-8`	
option	`-w, --write`	off	Write back to PATH (not for stdin)
option	`--check`	off	Exit `1` if format would change (no write)
option	`-o, --output-file`	`-`	Output when not using `--write`

```
pyne format script.pine            # stdout
pyne fmt script.pine -w
pyne format script.pine --check    # CI drift gate
```

`pyne convert`

Rewrite older Pine toward v5 or v6. `--to 6` converts **any** older version (missing `//@version` counts as v1): color / `n` / timeframe idents, `ta.*` / `math.*` / `request.*` namespaces, `study(` → `indicator(`, typed `input.*()`, `iff` / `offset`. Not a semantic v6 migrator (bool-as-int, `na`).

Kind	Name	Default	Description
argument	`PATH`	required	File or `-` (stdin)
option	`--to`	required	`5` / `6` / `v5` / `v6`
option	`--encoding`	`utf-8`	
option	`-w, --write`	off	Write back to PATH (not for stdin)
option	`-o, --output-file`	`-`	Output when not using `--write`

```
pyne convert old.pine --to 6
pyne convert strategy.pine --to 6 -w
```

`pyne parse-and-dump` (`dump` / `ast`)

Kind	Name	Default	Description
argument	`PATH`	required	Input file
option	`--encoding`	`utf-8`	
option	`--indent`	`2`	Dump indent
option	`-o, --output-file`	`-`	Output path

```
pyne dump strategy.pine
pyne parse-and-dump strategy.pine --indent 4 -o tree.txt
```

`pyne parse-and-unparse` (`unparse`)

Kind	Name	Default	Description
argument	`PATH`	required	Input file
option	`--encoding`	`utf-8`	
option	`-o, --output-file`	`-`	Output path

```
pyne unparse messy.pine -o clean.pine
```

`pyne lint`

Kind	Name	Default	Description
argument	`PATH`	optional	File; omit or `-` → stdin
option	`--encoding`	`utf-8`	
option	`--fail-on`	`errors`	`errors` | `warnings` | `all` | `never`
option	`--json`	off	JSON array of findings
option	`-q, --quiet`	off	Summary only

```
pyne lint strategy.pine
pyne lint --json --fail-on never strategy.pine
pyne lint --fail-on warnings strategy.pine
cat strategy.pine | pyne lint -
```

`pyne compile`

Compile via `pynescript.compiler.compile_script`. Requires `pip install "hoox-pyne[compile]"` for full Numba path; `--emit` works without JIT.

Kind	Name	Default	Description
argument	`PATH`	required	Pine file
option	`--encoding`	`utf-8`	
option	`--emit`	off	Print generated Python only (no exec/JIT)
option	`-o, --output-file`	`-`	Where to write `--emit` output
option	`--time / --no-time`	time on	Print compile timing

```
pyne compile script.pine --emit
pyne compile script.pine --emit -o out.py
pyne compile script.pine            # warm load / compile-check
```

`pyne prewarm`

H2 warm-compile product path: pay cold JIT before first interactive run.

Kind	Name	Default	Description
argument	`PATH…`	optional	Pine files to compile into IR caches
option	`--encoding`	`utf-8`	
option	`--force`	off	Re-run builtin warm-up
option	`--json`	off	Machine-readable summary

```
pyne prewarm                    # shared Numba kernels only
pyne prewarm strategy.pine lib/*.pine
pyne prewarm --json
```

`pyne run`

Compile and execute on deterministic synthetic OHLCV (same compile pipeline as Pro API `mode=compile`). **No `--mode`.** Does **not** call `Runtime.run`. Prefer package `pynescript.runtime.Runtime` / Pro API for real bars, `libraries=`, and `timeout_seconds`.

Kind	Name	Default	Description
argument	`PATH`	required	Pine file
option	`--encoding`	`utf-8`	
option	`--bars`	`50`	Synthetic bar count
option	`--json`	off	Plot series summary as JSON
option	`-q, --quiet`	off	Errors / exit only

```
pyne run strategy.pine --bars 100
pyne run strategy.pine --json
```

`pyne optimize`

Search `input.*` values for a **`strategy()`** script. Loops interpret `Runtime` (not `/backtest/quick`). Requires `--space` JSON.

Kind	Name	Default	Description
argument	`PATH`	required	Pine file (`-` = stdin)
option	`--encoding`	`utf-8`	
option	`--bars`	`120`	Synthetic bar count (min 8)
option	`--trials`	`20`	Trial budget (max 200)
option	`--space`	required	JSON `{"params":[{"name","kind","min","max"}]}`
option	`--sampler`	`auto`	`auto` | `random` | `tpe` | `grid`
option	`--objective`	`composite`	`composite` | `net_pnl` | `profit_factor` | `calmar`
option	`--validation`	`holdout`	`holdout` | `walk-forward` | `in-sample`
option	`--train-bars`	derive	Walk-forward train window
option	`--test-bars`	derive	Walk-forward test window
option	`--step-bars`	derive	Walk-forward step (default: the test window)
option	`--holdout-frac`	`0.3`	Holdout test fraction
option	`--seed`	unset	RNG seed
option	`--json`	off	Full study JSON

`--validation walk-forward` derives windows from `--bars` when the bar flags are omitted: `train=max(8, bars*2//3)`, `test=max(4, bars//6)`, `step=max(1, test)` (so default `--bars 120` → 80/20/20). Explicit `--train-bars` / `--test-bars` / `--step-bars` override that. The CLI refuses walk-forward when `--bars < train+test`. All-errored studies exit 1 even if the study status is `success`.

```
pyne optimize strat.pine --trials 20 --sampler tpe \
  --space '{"params":[{"name":"Fast","kind":"int","min":3,"max":12}]}'
pyne optimize strat.pine --validation walk-forward --bars 120 --trials 20 \
  --space '{"params":[{"name":"Fast","kind":"int","min":3,"max":12}]}'
pyne optimize strat.pine --validation walk-forward --train-bars 80 --test-bars 20 \
  --space '{"params":[{"name":"Fast","kind":"int","min":3,"max":12}]}'
```

Library: `pynescript.optimize.run_study`. HTTP: POST /optimize.

`pyne data`

Kind	Name	Default	Description
argument	`SYMBOL`	required	Ticker / market symbol
option	`--provider`	`mock`	`mock` | `yahoo` | `alphavantage` | `ccxt`
option	`--period`	`1y`	e.g. `6mo`, `1y`
option	`--interval`	`1d`	e.g. `1h`, `1d`
option	`--api-key` / `--secret`	empty	Provider credentials
option	`--exchange`	`binance`	CCXT exchange id
option	`--format`	`table`	`table` | `json` | `csv`
option	`-o, --output-file`	`-`	Output path

```
pyne data AAPL
pyne data AAPL --provider yahoo --period 6mo
pyne data BTC/USDT --provider ccxt --exchange binance --format csv
```

CCXT needs `pip install "hoox-pyne[data]"`.

`pyne runner`

HTTP client for the optional Flask hosted runner (`PYNE_RUNNER=1`). Group options apply to every subcommand.

Kind	Name	Default	Description
option	`--url`	`http://127.0.0.1:5002`	Pro API origin (`PYNE_API_URL`)
option	`--token`	empty	`X-Admin-Token` (`ADMIN_TOKEN`)

Subcommand	Role
`deploy PATH`	Read a `.pine` / `.pyne` file and `POST /scripts`
`list`	`GET /scripts`
`show ID`	`GET /scripts/:id`
`delete ID`	`DELETE /scripts/:id`
`tick`	`POST /cron/run` (`--id`, `--force`)
`jobs`	`GET /cron/jobs`
`enable ID` / `disable ID`	`PUT /cron/jobs`

```
pyne runner deploy script.pine --id demo --symbol BTCUSDT --source mock
pyne runner tick --id demo --force
pyne runner list
pyne runner --url http://127.0.0.1:5002 jobs
```

Full HTTP contract: hosted runner.

`pyne info` (`ls`)

Kind	Name	Description
option	`--json`	Machine-readable payload

Prints package version, Python, platform, and whether `numba` / `rich` are available.

Related: `pyne-lsp` (not a subcommand)

```
pip install "hoox-pyne[lsp]"
pyne-lsp
# alias: pynescript-lsp
python -m pynescript.langserver
```

Internals

Item	Path
CLI group	`src/pynescript/__main__.py`
Entry	`pyproject.toml` → `pyne` / `pynescript` = `pynescript.__main__:cli`
parse/dump/unparse	`src/pynescript/ast/helper.py`
lint	`src/pynescript/ast/linter.py`
compile / prewarm	`src/pynescript/compiler/`
optimize	`src/pynescript/optimize/`
data	`src/pynescript/util/data.py`

Invariants

- Aliases: `dump` / `ast` → `parse-and-dump`; `unparse` → `parse-and-unparse`; `fmt` → `format`; `ls` → `info`.
- Prefix match works when unique (`pynescript ch` → `check`).
- `check` walks directories; dump/format take a single file.
- **Docker:** `pynescript-cli` image `ENTRYPOINT` is `pynescript` (non-root).
- **Standalone binaries:** GitHub Release `pynescript-cli-*` (Nuitka).

Failure modes

Situation	Result
Parse failure (`check` / dump)	Non-zero exit
Format `--check` drift	Exit `1`
Lint threshold	Non-zero per `--fail-on`
Compile without numba (full path)	Clear error; `--emit` still works
Provider failure	Click error with message
Missing `ccxt` for data	Import/provider error → install `[data]`

See also

- CLI guide
- Runtime modes
- Installation
- Compiler overview
- Docker

RUNTIME MODES

interpret / compile / auto defaults differ by surface. Library interpret, POST /run auto, pyne run compile-only. PyneTS 0.2.0 has interpret + JS compile.

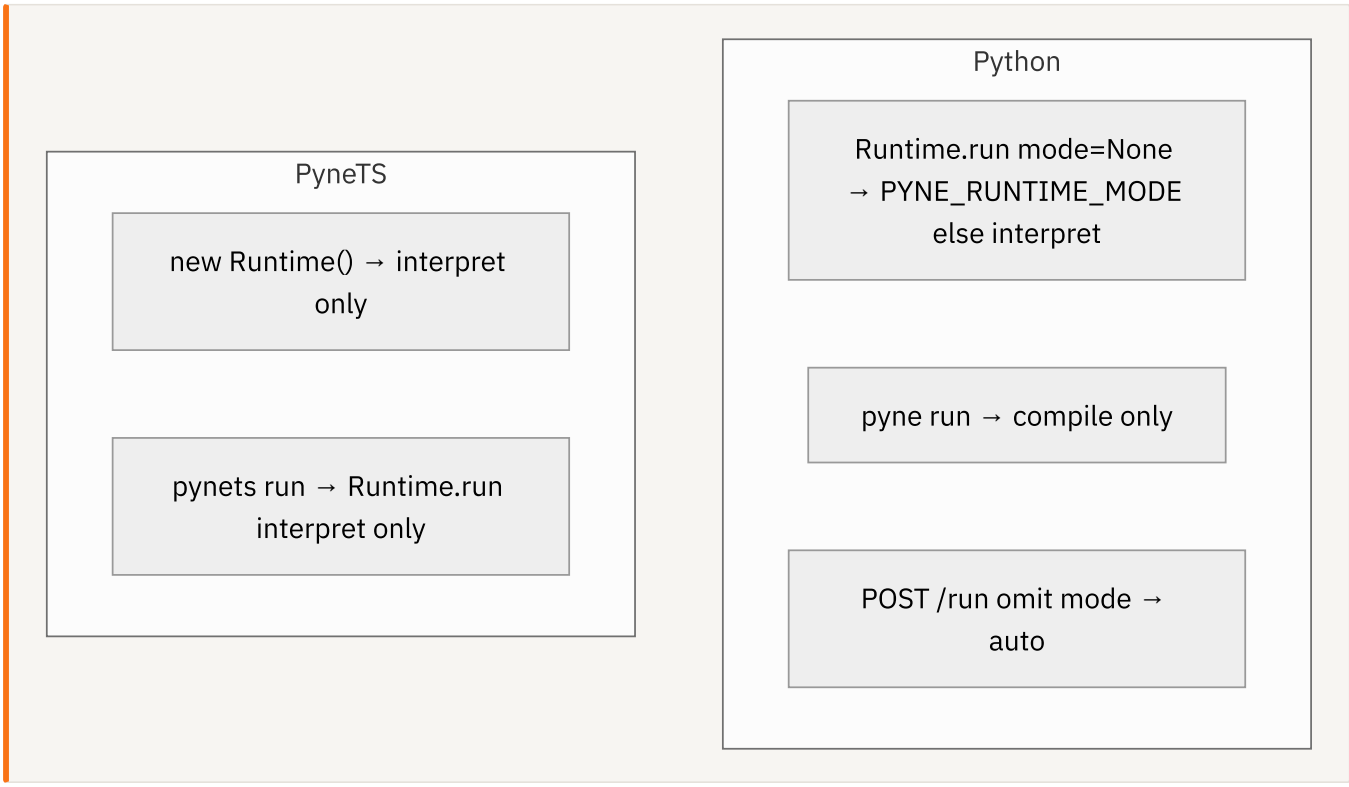
RUNTIME MODES

Abstract

`mode` selects how the bar loop executes: **`interpret`** (AST walk), **`compile`** (transpiled loop), or **`auto`** (try compile, fall back to interpret). The **default is not the same on every surface**. This is the most common footgun when moving a script from the library to the Pro API or between Python and PyneTS.

There is **no** `Runtime.evaluate`.

Conceptual model



Interface surface

What each mode means

Mode	Python	PyneTS (this checkout)
`interpret`	AST walker. Full host: `input.*`, `request.*`, libraries, profiler, alerts	AST walker only (`pynets/src/runtime/interpret.ts`). Result `mode` is always `"interpret"`
`compile`	Numba nopython **or** object-mode Python loop	**Not implemented** — no `src/runtime/compile/` in this tree
`auto`	Try compile; on ineligibility / compile error / compiled runtime error → interpret	**Not implemented**

Defaults by surface

Surface	Default `mode`	Calls `Runtime.run`?
`pynescript.runtime.Runtime.run(..., mode=None)`	`PYNE_RUNTIME_MODE` else **`interpret`**	yes
Pro API `POST /run` body omits `mode`	schema **`auto`**	yes (host wrap)
`?mode=` query (legacy)	overrides body	yes
CLI `pyne run`	**compile only** — not `Runtime.run`	**no**
`new Runtime()` / `pynets run`	**interpret only** (no `--mode`; `--bars` default **20**)	yes
AXIS engine `server`	whatever you pass; UI default often `auto`	HTTP `/run`
AXIS engine `pyodide`	object-mode in Wasm (no Numba)	in-browser Python

`Runtime.run` (Python, exact)

```
Runtime(symbol="AAPL").run(
    source_code,
    ohlcv_data,
    data_feed=None,
    data_provider=None,
    mode=None,            # env PYNE_RUNTIME_MODE else "interpret"
    inputs=None,
    profiler=False,
    timeout_seconds=None,
    libraries=None,
    realtime_last_bar=False,
    realtime_ticks=1,
    realtime_bars=0,
    realtime_from_bar=None,
)
```

Argument	Surfaces	Effect on `mode`
`mode`	library / `POST /run`	See defaults table. `pyne run` has no `--mode`.
`inputs`	library / `POST /run`	Non-empty overrides **force interpret** (compile does not apply them).
`libraries`	library / `POST /run` / `POST /run/batch` (max 32)	`[{namespace, name, version, source}]` for `import ns/Name/ver`. Compile-ineligible; `auto` keeps the list on interpret fallback. **Not** on `pyne run`.
`timeout_seconds`	library / `POST /run` / `POST /run/batch`	Interpret wall-clock budget. Exceeded → partial plots + `timed_out`. Omit / `null` / `≤ 0` → no timeout. **Not** on `pyne run`.
`profiler`	library / `POST /run`	Forces interpret (line timings).
`realtime_*`	library only	Interpret forming-bar window (`realtime_last_bar`, `realtime_ticks`, `realtime_bars`, `realtime_from_bar`).

`inputs` / `profiler` / `libraries` / `timeout_seconds` / `realtime_*` force or keep the **interpret** host (or force interpret after compile miss).

`Runtime.run` (PyneTS)

```
new Runtime(symbol, { inputs, broker, timeframe })
  .run(source, ohlcv, extra?)
```

There is **no** constructor/`extra.mode` and **no** `libraries` option on this host. `RuntimeResult.mode` is always `"interpret"`.

Internals

Surface	Implementation
Python library	`src/pynescript/runtime/host.py`
Python CLI `run`	`src/pynescript/__main__.py` → `compile_script` + synthetic bars
Pro API	`backend/app.py` `RUN_SCHEMA` default `auto`
PyneTS	`pynets/` **v0.2.0** — interpret + JS compile (`mode` `interpret` / `compile` / `auto`). Python remains the oracle
PyneTS CLI	`pynets/src/cli.ts` → `new Runtime("AAPL", { broker }).run`

Invariants & edge cases

1. **Do not document `pyne run` as `mode=auto`.** It is compile-only smoke (`--bars` default 50).
2. **Do not document `pynets run` as Python `pyne run`.** PyneTS `Runtime.run` defaults to interpret; **0.2.0** also has JS `compile` / `auto`. Default `--bars` is **20** (Python `pyne run` defaults to **50**).
3. `result.mode` is what **executed**. After `auto`, also read `auto_backend` and `compile_fallback_reason`.
4. Foreign `request.security` is `na` on every host. Compile does not invent bars to become "more complete".
5. AXIS Pyodide `compile` / `auto` cannot be Numba (Wasm).

Worked examples

Force interpret everywhere you care about `input.*` + libraries + alerts:

```
Runtime(symbol="AAPL").run(src, bars, mode="interpret", inputs={"len": 14})
```

```
new Runtime("AAPL", { inputs: { len: 14 } }).run(src, bars);
```

```
# Python desk smoke (compile only)
pyne run script.pine --bars 50

# PyneTS desk smoke (Runtime.run; default interpret, --bars 20)
pynets run script.pine --bars 50
```

Failure modes

Symptom	Cause	Fix
Pro API "misses" an input	`auto` compiled a path that drops overrides	Pass `mode: "interpret"`
`pyne run` ≠ notebook `Runtime.run`	Different host	Use the library or Pro API
Expect PyneTS ≡ Python compile	JS emit is object-mode analog, not Numba	Treat Python Runtime as the oracle
`mode` omitted, two hosts disagree	Different defaults	Set `mode` explicitly

See also

- Evaluate scripts
- Python compiler
- PyneTS compile
- Configuration
- Evaluate contract

GLOSSARY

Definitions of PYNE end-user terms: AST, ASDL, bar-loop, series, Runtime, LSP, Pro API, extras, and related concepts.

GLOSSARY

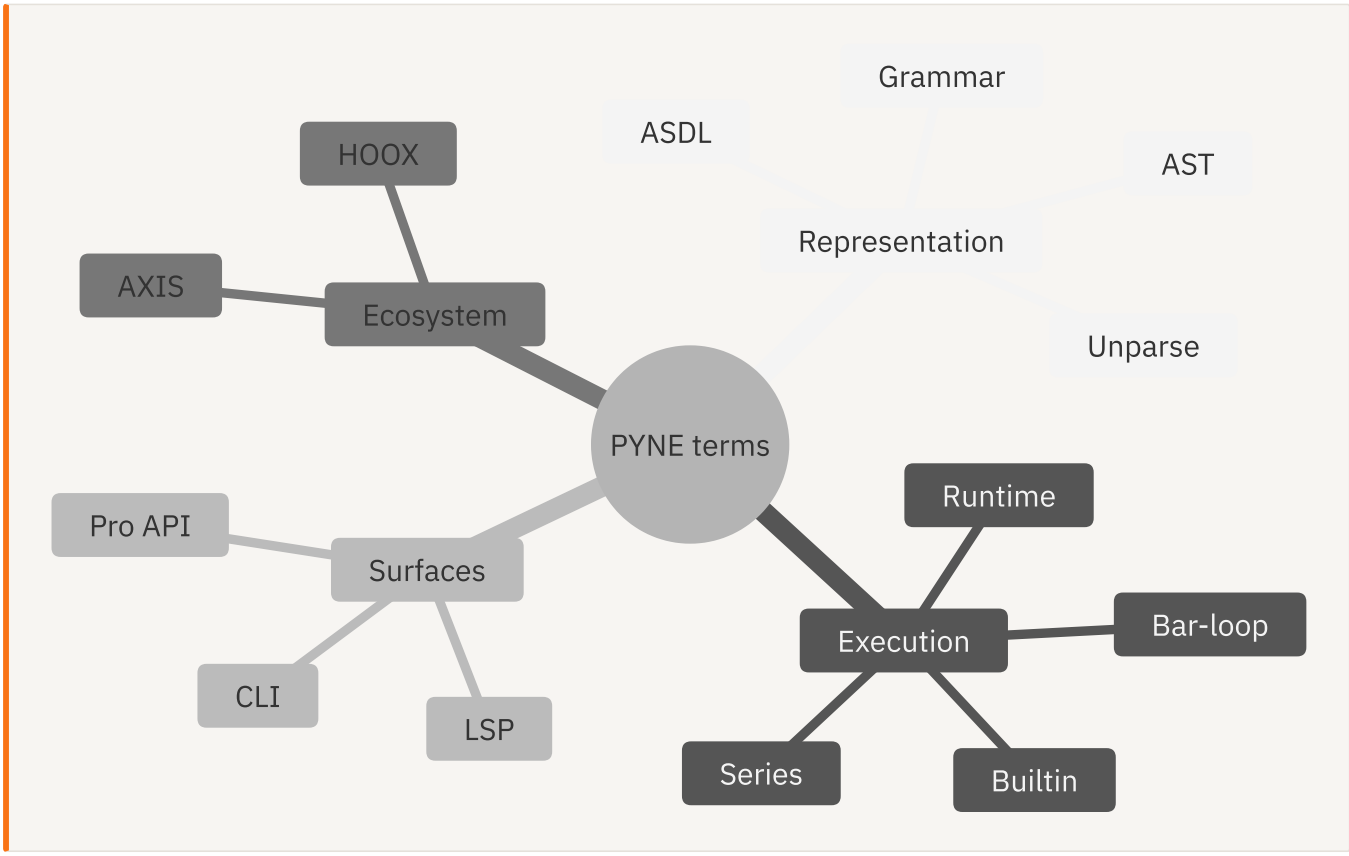
Abstract

Shared vocabulary for the End User track. Terms are defined as **used in this repo**, not as MarketingView marketing copy. Cross-links point at deeper systems manuals where useful.

Conceptual model

Terms fall into four buckets:

1. **Language representation** — grammar, AST, unparser
2. **Execution** — series, bar-loop, builtins, strategy events
3. **Surfaces** — CLI, library, LSP, Pro API
4. **Ecosystem** — AXIS, HOOX, PyneTS, pyne-worker, pyne-agent-worker



Interface surface

This page is pure reference — no commands required.

Internals (repo paths)

Term cluster	Primary paths
Parse / AST	`src/pynescript/ast/helper.py`, `builder.py`, `grammar/`
Evaluate	`src/pynescript/ast/evaluator/`, `src/pynescript/runtime/` (`backend/runtime.py` is a compat shim)
Lint	`src/pynescript/ast/linter.py`
LSP	`src/pynescript/langserver/`
HTTP	`backend/app.py`

Invariants & edge cases

- **Pine Script™ / TradingView®** are trademarks of TradingView, Inc. PYNE is independent (see product index).
- "Parity" is empirical (tests/corpus), not a legal claim of equivalence.

Terms

ASDL

Abstract Syntax Description Language. Schema language used to generate Python AST node classes (`Pinescript.asdl` → generated node module). Gives a single algebraic description of script structure.

Annotation

Special comments such as `//@version=6` (preferred) attached to script or declaration nodes during parse (`_add_annotations` in `helper.py`).

AST

Abstract Syntax Tree. In-memory tree of ASDL-generated nodes produced by `parse`. Root is typically a `Script` (`mode="exec"`) or `Expression` (`mode="eval"`).

AXIS

Optional **charting PWA "AXIS"** product (`frontend/`, docs at `/axis/docs`). Consumes evaluate results (series/plots); not required for evaluation.

Bar / bar-loop

A discrete OHLCV time step. A **bar-loop** evaluator re-executes script logic once per bar with updated series history (`pynescript.runtime.Runtime.run`).

Builtin

Host-provided namespace function or value (`ta.rsi`, `strategy.entry`, `math.sqrt`, ...). Implemented under `ast/evaluator/builtins/` and catalogued for LSP metadata.

CLI

The preferred **`pyne`** Click command group (alias `pynescript`: `check`, `format`, `lint`, `compile`, `prewarm`, `run`, `data`, `info`). `pyne run` is compile-only synthetic smoke — not `Runtime.run`.

Compile mode

`mode="compile"` on Runtime / `/run`: attempts a **Numba (or similar) compiled subset** of bar logic for speed. Not full language coverage.

Data provider

Historical OHLCV source (`mock`, `yahoo`, `alphavantage`, `ccxt`) via `pynescript.util.data`.

Data feed

Realtime stream abstraction (`datafeed`, e.g. CCXT Pro) for live candles/trades.

Dump

`dump(node)` — pretty textual representation of an AST for debugging and tests (not executable Pine).

Evaluate / evaluator

Execution of AST nodes to values. Ranges from `literal_eval` (expression-safe) to full statement/builtin evaluators and package **`pynescript.runtime.Runtime`** (Pro API / workers share the same host).

Extra (package extra)

Optional dependency set in `pyproject.toml`: `lsp`, `dev-lsp`, `compile`, `data`, `datafeed`, `pro`.

HOOX

Edge **trade execution mesh** (separate monorepo / docs under `/docs`). Can consume strategy events from evaluate pipelines; optional.

Interpret mode

Default Runtime mode: AST walking per bar without the compiled subset path.

`libraries=`

Optional `Runtime.run` / `POST /run` list of `{namespace, name, version, source}` dicts registered before `import ns/Name/ver` resolves. Compile-ineligible. HTTP cap is 32. Not available on `pyne run`.

`literal_eval`

Helper that parses an expression (or accepts an AST) and evaluates literals plus supported builtins/series context — analogous in spirit to Python's `ast.literal_eval` but Pine-aware and broader.

Lint / `LintWarning`

Static diagnostics with `code`, `message`, `line`, `severity` (`error` | `warning`). Entry: `lint_script`.

LSP

Language Server Protocol. `pyne-lsp` process (alias `pynescript-lsp`) implementing diagnostics, completion, hover, navigation, formatting.

`once`

August 2026 Pine **conditional structure** (`hoox-pyne` **0.4.4**). `once [cond]` runs its indented block the first time the condition is true on a **closed** bar, then never again. Optional condition defaults to `true`. Unlike `if` / `switch`, it **does not return a value** (`x = once …` does not parse). Interpret may re-run the body on unconfirmed realtime ticks until `barstate.isconfirmed`. See Expressions & statements.

NodeVisitor / NodeTransformer

Patterns for traversing or rewriting ASTs (`visitor.py`, `transformer.py`).

OHLCV

Open, high, low, close, volume — bar fields passed to Runtime / `/run` (plus `time`).

PyneTS

TypeScript / Bun library (`@hoox-sh/pynets`). Public names match Python: `parse`, `unparse`, `Runtime.run`. **Not** a Worker and **not** pyne-worker. See PyneTS.

pyne-agent-worker

Sister Cloudflare Worker that turns natural language into Pine source (`POST /v1/chat`). Standalone by default; optional validate via pyne-worker. See pyne-agent-worker.

pyne-worker

Python Cloudflare Worker (sister repo) embedding pynescript for edge `POST /run`. See pyne-worker. HOOX mesh notes: isolate profile.

Plot / series / plot_meta

Evaluate outputs: plotted values, named multi-series maps, and metadata for AXIS UIs (AXIS).

Pro API

Flask application in `backend/` exposing `/run`, preview, backtest, and auth endpoints.

PYNE

Product name for this evaluation stack (docs under `/pyne/docs`). PyPI distribution is **`hoox-pyne`**; the import package is **`pynescript`**.

Round-trip

`parse` → `unparse` (optionally → `parse` again). Used to normalize formatting and test fidelity.

Runtime

`pynescript.runtime.Runtime` — multi-bar execution engine (`src/pynescript/runtime/host.py`). `backend.runtime` re-exports the same class for the Pro API. Library default `mode` is **interpret**; `POST /run` defaults to **auto**. See modes.

Series

Time-indexed value stream with history access (`close[1]`). Implementation includes `PineSeries` and evaluator series objects.

Strategy event

Structured record of entries/exits/orders emitted during strategy evaluation for downstream systems (HOOX, analytics).

`timeout_seconds`

Optional `Runtime.run` wall-clock budget (interpret path). Also a `POST /run` / `/run/batch` body field (0.3.11+). Exceeded runs return partial plots with `timed_out`. Omit / `null` / `≤ 0` → no timeout.

Optimize (hyperparameter search)

`pynescript.optimize.run_study` / `pyne optimize` / `POST /optimize` (0.3.12+). Loops interpret `Runtime` with `input.*` overrides for a `strategy()` script. Not a Pine builtin and not `/backtest/quick`.

`timeframe.change`

`timeframe.change(tf)` is true on the first bar of a new UTC fixed-width period (0.3.12+). Not exchange-calendar / DST aware. See known divergences.

Unparse

`unparse(node)` — regenerate Pine source text from an AST.

`//@version`

Version pragma annotation; linter expects v5+ for modern scripts.

Warmup

Bars required before a TA function has enough history; early outputs may be `na`.

Worked examples

Using terms together:

```
Source --parse--> AST --unparse--> normalized source
AST + OHLCV --Runtime bar-loop--> series + strategy events
AST --lint_script--> LintWarnings
AST --LSP--> editor diagnostics/completion
series --AXIS--> chart
events --HOOX--> exchange orders
```

Failure modes

Misusing terms often causes support confusion:

Confused pair	Distinction
dump vs unparse	dump is debug AST text; unparse is Pine source
literal_eval vs Runtime	single-shot expressions vs multi-bar scripts
pyne vs pyne-lsp	desk CLI vs language server (aliases: pynescript*)
AXIS vs PYNE	AXIS vs evaluate engine
data provider vs data feed	historical fetch vs realtime stream

See also

- End User hub
- FAQ
- PYNE product map
- AXIS docs
- HOOX docs

FAQ

Frequently asked questions about pynescript install, Pine version support, TradingView parity, CLI vs library, LSP, Pro API, licensing, and ecosystem boundaries.

FAQ

Abstract

Short answers to questions that recur when adopting PYNE as a **consumer**. Where behavior is evolving, answers point at status docs rather than overclaiming parity.

Conceptual model

Questions group into: install & package shape, language coverage, evaluation fidelity, tooling surfaces, and product boundaries (AXIS / HOOX / workers).

Interface surface

No dedicated FAQ API — pointers only.

Internals (repo paths)

Answers cite:

- `pyproject.toml` — version, extras, license
- `src/pynescript/__main__.py` — CLI reality
- `src/pynescript/runtime/host.py` — `Runtime.run`
- `docs/missing_features.md` / reference status pages — coverage
- `backend/app.py` — HTTP contract

Invariants & edge cases

- Prefer reading code + status docs over README marketing tables when they disagree; mark drift with issues.
- Trademark disclaimer lives on the product index once — not repeated here in full.

Questions

What is PYNE vs pynescript vs pyne?

- **PYNE** — product documentation brand for the open evaluate stack (grammar → AST → runtime → LSP → Pro API → workers).
- **`hoox-pyne`** — **PyPI distribution** name (live, **0.3.14+**). Install with `pip install hoox-pyne`. Do **not** install plain `pyne` / `PyNE` from PyPI without the `hoox-` prefix (unrelated) or the upstream `pynescript` package (elbakramer).
- **`pynescript`** — **import package** (`import pynescript` , including `pynescript.runtime`).

- `pyne` / `pyne-lsp` — preferred console scripts; `pynescript` / `pynescript-lsp` remain aliases.

What Python versions are supported?

`requires-python = ">=3.10"`. Classifiers list **3.10–3.13**. CI exercises that matrix; pin to 3.10+ for production.

How do I install just the parser?

```
pip install hoox-pyne
```

No extra required for `parse` / `unparse` / `lint` / basic `literal_eval`.

How do I install the LSP?

```
pip install "hoox-pyne[lsp]"
```

Run `pyne-lsp` (alias `pynescript-lsp`) — not a Click subcommand of `pyne`.

Why is `pynescript lsp` missing?

The registered entry point is the separate script `pynescript-lsp`. Some docs historically mentioned a `lsp` subcommand; trust `pyproject.toml` `[project.scripts]` and `__main__.py` for ground truth.

Which Pine versions are supported?

v6 is the recommended surface (declare `//@version=6`); v5 remains fully supported. See implementation status, missing features, and pine v6 surface.

Does PYNE support the `once` keyword?

Yes, as of **hoox-pyne 0.4.4** (TradingView® August 2026). `once [cond]` runs its block the first time the condition is true on a closed bar. It is **not** an expression — you cannot assign its result. See Expressions & statements.

Is this affiliated with TradingView?

No. Independent open-source toolchain. Pine Script™ and TradingView® are trademarks of TradingView, Inc.

Will my TradingView strategy produce identical PnL?

Not guaranteed. Broker emulator rules, fill models, request.* data, and some builtins differ. Use PYNE for offline analysis, CI, and self-hosted evaluate; validate critical strategies numerically (numerical validation).

Can I format Pine like Black?

`parse-and-unparse` / LSP format normalize via the AST unparser. There is no rich style config comparable to Black's profile system.

Does the CLI evaluate strategies?

`pyne run script.pine --bars 50` compiles and executes on **synthetic** OHLCV. It is **not** `Runtime.run` (no real bars, no `mode` / `libraries` / `timeout_seconds` / `inputs`). For real OHLCV use:

- `from pynescript.runtime import Runtime` — library default `mode` is **interpret**

- `POST /run` — schema default `mode` is **auto**
- `examples/execute_script.py` as a didactic sample (not the production host)

See modes.

How do I lint in CI?

```
pynescript lint --fail-on errors path/to/script.pine
```

Loop files yourself; the CLI is single-file / stdin.

What lint rules exist?

Syntax (`E001`), version (`W001` – `W002`), deprecated patterns (`W101` –), style/naming (`C001` – `C004`). See `src/pynescript/ast/linter.py` and the library API guide.

How do I get market data?

```
pynescript data AAPL --provider yahoo
pip install "hoox-pyne[data]"  # for ccxt
```

Or providers in Python via `pynescript.util.data.get_provider` .

What is the Pro API free tier?

`POST /run` and `POST /run/batch` are exposed as free evaluate endpoints (still your CPU if self-hosted). Default body `mode` is **`auto`** (prefer warm compile; fall back to interpret). Bar/script/rate/concurrency caps are **off unless** you set `FREE_TIER_LIMITS=1` . Responses include `alerts[]` when scripts call `alert()` / true `alertcondition()` (interpret path). Optional L2 webhooks: body `webhook_url` or env `ALERT_WEBHOOK_URL` . Preview/backtest routes are Pro-tier with API keys.

How do interpret and compile compare?

Use the dual-host harness (repo root):

```
python scripts/compare_interp_compile.py --bars 1000 --limit 50
```

See Interpret ↔ compile parity. Prefer `mode=auto` in production; force `interpret` when you need alerts, `input.*` overrides, or full `request.*` .

What file extension should I use?

Prefer **`.pyne`** for HOOX / PYNE stack sources. `.pine` / `.pinev5` / `.pinev6` remain supported by the VS Code extension and parsers.

How complete is real-world script runtime?

PYNE aims for practical fidelity on first-party fixtures and unit tests. It does **not** claim 100% TradingView® platform parity (chart host, proprietary data, every edge-case builtin). See Compatibility and implementation status. Third-party script corpora are not shipped in this repository.

How do I self-host the API?

Install backend requirements, set `API_KEY_STORE` / `ALLOWED_ORIGINS` / optional `ADMIN_TOKEN`, run `python -m backend.app` or `make run`. See configuration and Pro API usage.

What is AXIS?

Optional charting PWA that can call the evaluate contract and render series. Docs: AXIS. PYNE evaluates without AXIS.

What is HOOX?

Optional edge trading mesh (trade-worker, etc.). Docs: HOOX. Strategy events from PYNE can feed HOOX; not required for parsing/eval.

Alerts: `alert()` / `alertcondition()` firings export on Pro API and pyne-worker `/run`. Optional HTTP webhooks via `webhook_url` or `ALERT_WEBHOOK_URL` (last-bar by default). See Alerts.

What is PyneTS vs pyne-worker vs pyne-agent-worker?

- **PyneTS** (`@hoox-sh/pynets`) — TypeScript / Bun **library + CLI**. `parse` / `unparse` / `Runtime.run`. Not a Worker. Python is the oracle.
- **pyne-worker** — Python Cloudflare Worker. Production `POST /run`. Vendors `pynescript.runtime`.
- **pyne-agent-worker** — NL authoring (`POST /v1/chat`). Standalone chat; optional generate → `/run` validate.

The name **pine-worker** is a leftover TypeScript experiment (hoox-sh/pine-worker). It is not colocated here and is not a product docs target.

See the ecosystem map and modes.

What is the default `mode`?

It depends on the surface — this is the usual footgun:

Surface	Default
`pynescript.runtime.Runtime.run(..., mode=None)`	`PYNE_RUNTIME_MODE` else **interpret**
`POST /run` (body omits `mode`)	**auto**
`pyne run`	**compile-only** (not `Runtime.run`, no `--mode`)

See modes.

Is there `Runtime.evaluate`?

No. The host API is **`Runtime.run`**. Python CLI `pyne run` is compile-only smoke and is not `Runtime.run`. PyneTS `pynets run` **does** call `Runtime.run`.

What license is pynescript?

AGPL-3.0-or-later per `pyproject.toml`. Network use of modified versions requires offering corresponding source (AGPL §13). Proprietary embedding usually needs a commercial license; consult your counsel.

Can I use this commercially?

AGPL permits commercial use under its terms, but distribution or network service of modified versions requires source availability. Hosted Pro API terms (if any) are separate from the library license.

Why is parse slow on huge files?

ANTLR parse + AST build scale with file size and nesting. Deep expression nests also stress recursion (limit temporarily raised to ≥5000). Split libraries when possible.

Round-trip changed my spacing — is that a bug?

Usually no. Unparser emits a normalized style. File a bug if **semantics** change (identifiers, call structure, annotations lost).

Where is the builtin list for autocomplete?

Generated metadata used by the LSP (`scripts/generate_builtin_metadata.py`). Do not hand-edit encrypted release blobs.

How do I report a grammar bug?

Minimal `.pine` repro + expected TV behavior + PYNE version. Prefer failing pytest if contributing.

Is Jupyter supported?

Yes — `from pynescript.ext.jupyter import load_ipython_extension` . The `%%pinescript` cell magic lints, parses, and unparses; it is **not** a bar-loop Runtime. You can also call `parse` / `literal_eval` / `Runtime.run` directly in notebooks.

Can I transform scripts programmatically?

Yes — `NodeVisitor` / `NodeTransformer` on the AST, then `unparse` . See library API.

Worked examples

“Is my install healthy?”

```
python -c "from pynescript.ast import parse, unparse; print(unparse(parse('//@version=6\nindicator
pynescript lint - <<< $'//@version=6\nindicator("t")\nplot(close)\n'
```

“Does HTTP evaluate work?”

```
curl -s http://127.0.0.1:5002/ | python -m json.tool
```

“Is LSP importable?”

```
python -c "import pygls; from pynescript.langserver.server import PynescriptLanguageServer"
```

Failure modes

If an FAQ answer appears wrong against your checkout:

1. Check package version (`pynescript --version`).
2. Read the cited source file.
3. Prefer `__main__.py` / `pyproject.toml` over secondary docs.
4. Open an issue or PR against the MDX page.

See also

- End User hub
- Installation
- Troubleshooting
- Glossary
- Compatibility
- CLI commands

16 · CORE

LANGUAGE CORE

ANTLR4 grammar, ASDL AST, builder, visitors, unparser, type system, linter, and error model for PYNE.

LANGUAGE CORE

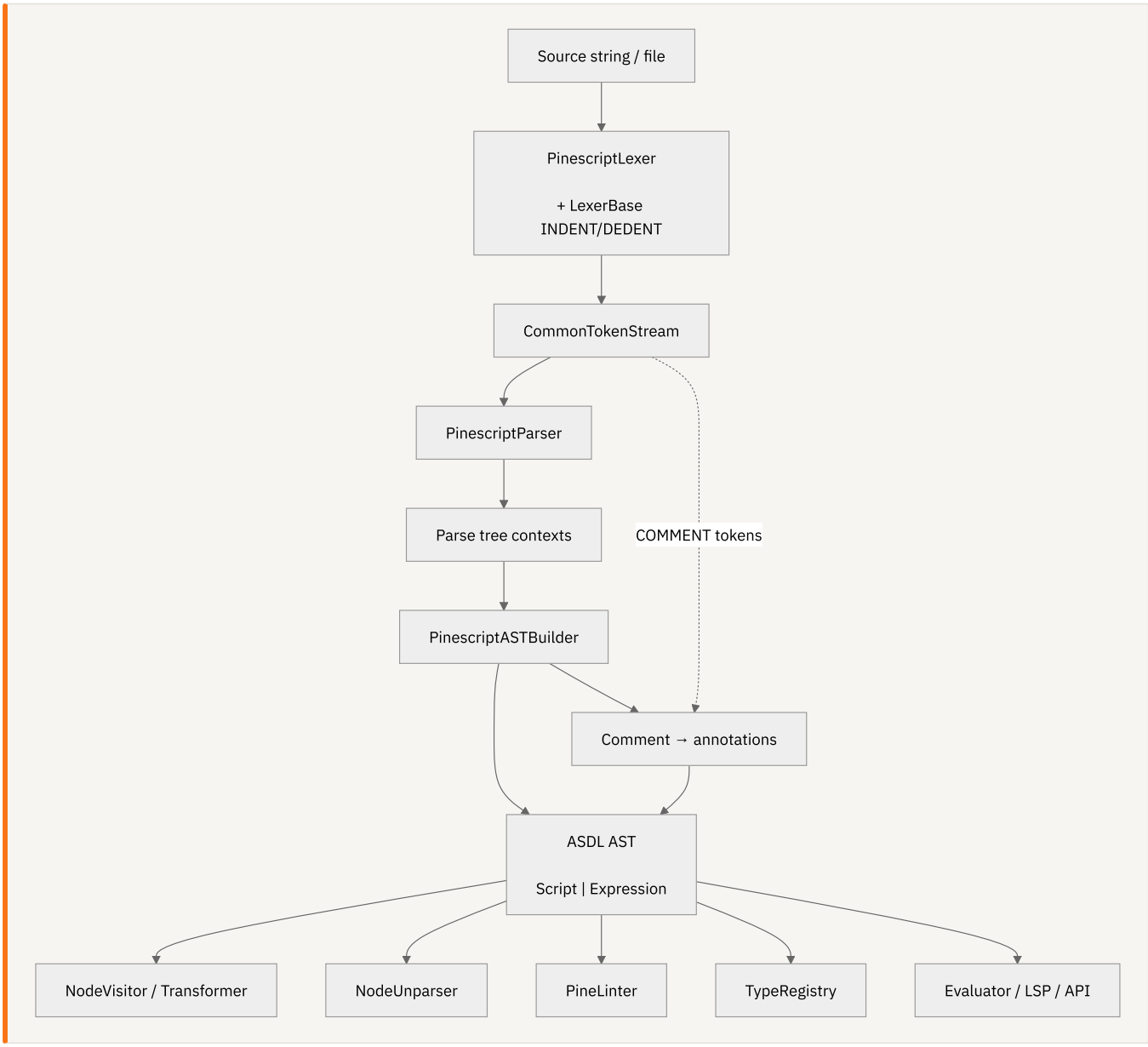
The language core is the front half of the evaluate contract: source text becomes a typed tree; the tree can be walked, rewritten, dumped, and unparsed without ever entering the bar loop.

Abstract

PYNE treats Pine Script as a **formal pipeline**, not a host plugin. Lexing and parsing are ANTLR4-driven. The concrete syntax tree is lowered by a hand-written visitor (`PinescriptASTBuilder`) into ASDL-generated dataclasses. Public helpers in `helper.py` mirror Python's `ast` module: `parse`, `unparse`, `dump`, `walk`, `literal_eval`, plus a process-local sha256 LRU parse cache (`clear_parse_cache`, `parse_cache_info`). Static tools (linter, type registry, LSP) consume the same tree. Evaluation is out of scope here — see Runtime.

Hard rule: edit grammar only under `resource/`. Paths under `generated/` are regenerated artifacts and must not be hand-edited.

Conceptual model



Stage	Role	Primary path
Grammar	Concrete syntax	`src/pynescript/ast/grammar/antlr4/resource/*.g4`
ASDL	Algebraic node shape	`src/pynescript/ast/grammar/asdl/resource/Pinescript.asdl`
Builder	CST → AST	`src/pynescript/ast/builder.py`
Helper	Public parse/unparse API	`src/pynescript/ast/helper.py`
Nodes	Re-export generated types	`src/pynescript/ast/node.py`

Interface surface

Consumers almost always enter through `pynescript.ast.helper`:

```
from pynescript.ast.helper import parse, unparse, dump, walk, literal_eval

tree = parse('//@version=6\nindicator("x")\nplot(close)')
print(dump(tree, indent=2))
print(unparse(tree))
```

Successful `parse` results are cached by `sha256(source)` + `mode` (default ON; `PYNE_PARSE_CACHE=0` disables). Treat returned trees as **read-only**.

Modes:

`mode`	Entry rule	Root node
`"exec"` (default)	`start_script`	`Script`
`"eval"`	`start_expression`	`Expression`

Full page inventory:

Page	Topic
Grammar (ANTLR4)	Resource vs generated; typed names; `=` vs `:=`
ASDL schema	Algebraic AST; codegen
Builder	CST visitor → ASDL nodes
Helper API	`parse`, cache, `dump`, `walk`, locations
Unparser	Precedence-aware source regen
Visitor / transformer	Read vs rewrite
Type system	Qualifiers, UDT registry
Linter	Static rules, codes
Error model	`SyntaxError`, indent, listener

Internals

```
src/pynescript/ast/
  helper.py              # public pipeline + sha256 LRU parse cache
  builder.py             # PinescriptASTBuilder (_setLocations = direct attrs)
  unparser.py            # NodeUnparser + Precedence
  visitor.py             # NodeVisitor
  transformer.py         # NodeTransformer
  collector.py           # StatementCollector (annotation pairing)
  linter.py              # PineLinter
  type_system.py         # Type*, TypeRegistry, MethodResolver
  error.py               # SyntaxError / IndentationError
  node.py                # from grammar.asdl.generated import *
  grammar/
    antlr4/
      resource/          # EDIT HERE: .g4 + *Base.py
      generated/         # NEVER HAND-EDIT
      error_listener.py
      tool/generate.py
    asdl/
      resource/          # EDIT HERE: Pinescript.asdl
      generated/         # NEVER HAND-EDIT
      tool/generate.py
```

Thin wrappers `grammar/antlr4/lexer.py`, `parser.py`, `visitor.py` re-export generated classes so application code imports stable paths.

`parse` uses a thread-local ANTLR engine and a two-stage **SLL then LL** strategy; successful trees go into a sha256 LRU (`PYNE_PARSE_CACHE`, default 128).

Invariants

1. **Resource is source of truth.** Lexer/parser `.g4` and `Pinescript.asdl` define syntax and node shape; generated Python is a build product (committed so install does not require Java).
2. **Round-trip is a test oracle.** For the supported surface, `unparse(parse(src))` should re-parse and preserve semantics (not byte-identical formatting).
3. **Locations are first-class.** Statements and expressions carry `lineno`, `col_offset`, optional `end_*` for diagnostics and LSP.
4. **Annotations are not free-floating.** Comment tokens on `COMMENT_CHANNEL` are reified as `Comment` nodes and attached to `Script` / `FunctionDef` / `TypeDef` / `Assign` via kind suffixes (`S`, `F`, `T`, `V`).
5. **Recursion budget.** Deep ternary chains raise the Python recursion limit temporarily during parse (cap ≥ 5000).
6. **Cached trees are shared by identity.** Mutating a `parse()` result (transformer, `increment_lineno`) mutates the LRU entry. Call `clear_parse_cache()` after intentional mutation, or disable the cache.
7. **`FunctionDef.returns` is a type spec**, not an executable expression. Optional leading `type_specification` on UDFs/methods is stored there (0.3.9+).

Worked examples

Minimal script:

```
from pynescript.ast.helper import parse, dump

src = """
//@version=6
indicator("demo")
x = ta.sma(close, 14)
plot(x)
"""
tree = parse(src)
assert tree.__class__.__name__ == "Script"
assert any("version" in a for a in (tree.annotations or []))
print(dump(tree, include_attributes=True, indent=2))
```

Expression-only eval mode:

```
from pynescript.ast.helper import parse, literal_eval

expr = parse("1 + 2 * 3", mode="eval")
assert literal_eval(expr) == 7
```

Failure modes

Symptom	Likely cause	Where to look
`SyntaxError` with caret	Lexer/parser reject	Error model
`IndentationError`	Bad INDENT/DEDENT from LexerBase	`PinescriptLexerBase.py`
Missing `visit_*` after grammar change	Builder not updated for new rule	Builder
Hand-edit of `generated/` lost on regen	Violated resource-only rule	Grammar
Full regen breaks `builder.py`	Parser context accessors changed	Prefer targeted lexer refresh (v6 case study)

See also

- Grammar (ANTLR4)
- Helper API
- Runtime
- LSP
- Compatibility

GRAMMAR (ANTLR4)

Resource .g4 grammars, generated artifacts, INDENT/DEDENT, regeneration, and the never-edit-generated rule.

GRAMMAR (ANTLR4)

Concrete syntax for Pine Script in PYNE is defined by a split lexer/parser ANTLR4 grammar plus hand-written base classes that inject Python-like indentation tokens.

Abstract

Two grammars live under `src/pynescript/ast/grammar/antlr4/resource/`:

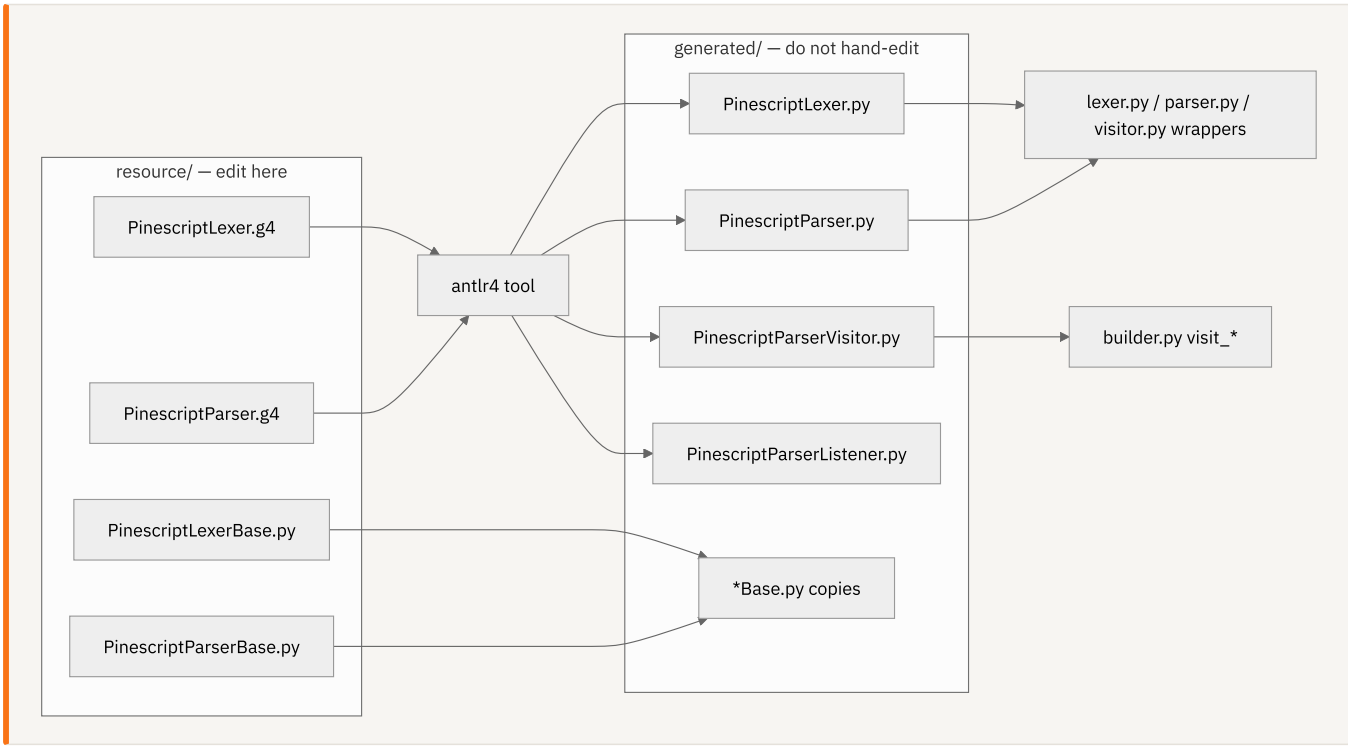
- `PinescriptLexer.g4` — keywords, operators, literals (including v6 triple-quoted strings), comments, hidden whitespace.
- `PinescriptParser.g4` — statements, expressions, type/enum/function/method declarations, control structures.

Python runtime classes under `generated/` are **outputs** of `antlr4 -Dlanguage=Python3`. They are committed so end users never need a JDK. **Never edit files under `generated/`** — changes evaporate on the next regeneration and fight the build pipeline.

Hand-written bases (copied into `generated/` by the generate tool):

- `resource/PinescriptLexerBase.py` — INDENT/DEDENT, newline elision, multiline string handling.
- `resource/PinescriptParserBase.py` — parser superClass hooks.

Conceptual model



Interface surface

Application code should not import `generated` symbols directly when a wrapper exists:

```
from pynescript.ast.grammar.antlr4.lexer import PinescriptLexer
from pynescript.ast.grammar.antlr4.parser import PinescriptParser
from pynescript.ast.grammar.antlr4.visitor import PinescriptParserVisitor
from pynescript.ast.grammar.antlr4.error_listener import PinescriptErrorListener
```

The public parse path is higher level — Helper API — which constructs lexer, token stream, parser, and strips default console error listeners in favor of `PinescriptErrorListener`.

Entry rules (`PinescriptParser.g4`)

Rule	Purpose
`start` → `start_script`	Full script (`statements? EOF`)
`start_expression`	Single expression + optional NEWLINE + EOF
`start_comments`	Comment stream only

Lexer structural facts

Token / channel	Behavior
`INDENT` / `DEDENT`	Imaginary tokens emitted by `PinescriptLexerBase` (declared in `tokens { }`)
`COMMENT`	`//` line comments → `COMMENT_CHANNEL` (not the default channel)
`HASH_COMMENT`	`# …` line comments remapped `-> type(COMMENT)` (must not swallow `#RRGGBB`)
`BACKTICKS`	Markdown fence `` ` `` remapped to `COMMENT` (paste noise)
`WS`	Spaces/tabs/form-feed → `HIDDEN`
`UNICODE_NOISE`	Curly quotes, bullets, arrows → `HIDDEN`
`ERROR_TOKEN`	Catch-all `.` for recovery/reporting
`TRIPLE_DQ_STRING` / `TRIPLE_SQ_STRING`	v6 multiline strings, typed as `STRING`
`LSHIFT` / `RSHIFT` / `TILDE` / `AMP` / `PIPE` / `CARET`	Bitwise `<<` `>>` `~` `&` `` ` ``

Keywords include `var` , `varip` , `method` , `export` , `enum` , `type` , qualifiers `const` / `input` / `simple` / `series` , and control forms `if` / `for` / `while` / `switch` / `once` (August 2026). Many of those words are **soft** in identifier position (`name` alternatives) so `as = input(...)` and `once = 3` parse.

Parser structural facts

- **Compound vs simple statements.** Functions, methods, types, enums, and structure-valued assignments are compound; imports, break/continue, expression statements, and simple assignments terminate with `NEWLINE` .
- **Trailing structures.** `a = 0, for i = 0 to n` is `trailing_structure_statements` (comma-separated simples ending in a structure).

- **Structures are dual-use.** `if` / `for` / `while` / `switch` appear both as statements and as **expressions** (`structure_expression`) — the classic Pine "if returns a value" form. **`once` is statement-only** (`once_structure` is not in `value_structure`); `x = once …` is a parse error.
- **Local blocks.** Either indented (`NEWLINE INDENT statements DEDENT`) or inline single statement.
- **Library export.** `EXPORT?` prefixes function/method/type/enum and June-2025 `export const T name = …` compound initializers.
- **UDF return types.** `function_declaration` / `method_declaration` accept an optional leading `type_specification` (`int ilog2(int n) => …`). Builder stores it on `FunctionDef.returns`.
- **Bitwise layer.** Between `and` and comparisons: `|` `^` `&` then `<<` / `>>`, then additive. Unary `~` is `Invert`.
- **Assign vs reassignment (0.3.9).** `=` on a bare name is **initialization** (`Assign` via `variable_declaration`). Reassignment of a name is `:=` (`ReAssign`). `=` is reassignment only for attribute / subscript targets (`obj.f =`, `a[i] =`).
- **Left-factored typed names.** `variable_declaration` is `declaration_mode? type_specification? name_store` with the typed alternative first so `int x = 1` does not consume `int` as the identifier. Same left-factor on `for_iterator` and parameter lists.

Internals

Paths

Path	Edit?	Role
`src/pynescript/ast/grammar/antlr4/resource/PinescriptLexer.g4`	**Yes**	Lexer grammar
`src/pynescript/ast/grammar/antlr4/resource/PinescriptParser.g4`	**Yes**	Parser grammar
`src/pynescript/ast/grammar/antlr4/resource/PinescriptLexerBase.py`	**Yes**	Indentation machine
`src/pynescript/ast/grammar/antlr4/resource/PinescriptParserBase.py`	**Yes**	Parser base
`src/pynescript/ast/grammar/antlr4/generated/**`	**No**	ANTLR output + copied bases
`src/pynescript/ast/grammar/antlr4/tool/generate.py`	Tool	Runs `antlr4`, copies `*.py` bases
`src/pynescript/ast/grammar/antlr4/error_listener.py`	Hand	Maps ANTLR errors → `ast.error.SyntaxError`

LexerBase responsibilities

`PinescriptLexerBase` is not optional sugar; it is part of the language definition:

- Ignore leading / collapse consecutive newlines; ensure trailing newline.
- Suppress newlines inside open `()` / `[]`, after operators, and for soft wrap lines whose indent width is not a multiple of four.
- Push `INDENT` / `DEDENT` with tab length and indent length = 4.
- Special-case multiline string tokens so wrap indentation inside strings is not mis-tokenized as structure.

Indent mistakes surface as `pynescript.ast.error.IndentationError` (subclass of the project `SyntaxError`).

Regeneration

```
# Project-aware entry (sets -o generated, -lib resource, copies *Base.py)
python -m pynescript.ast.grammar.antlr4.tool.generate

# hatch lint:gen-parser is only the antlr4 CLI (`antlr4 {args}`) — pass flags yourself.
# hatch run lint:gen-parser -- -o …/generated -lib …/resource -listener -visitor -Dlanguage=Pythor
```

`tool/generate.py` invokes `$(sys.executable)/../antlr4` with `-o generated -lib resource -listener -visitor -Dlanguage=Python3` on `resource/*.g4`, then copies `resource/*.py` into `generated/`. Requires a working `antlr4` CLI (hatch lint env pulls `antlr4-cli`). Generated modules are excluded from mypy/ruff strictness in `pyproject.toml` — do not "lint-fix" them by hand.

Downstream after grammar change

1. Regenerate (or selectively refresh lexer — see case study).
2. Add/adjust `visit_*` methods in `src/pynescript/ast/builder.py` for new or renamed rules.
3. If the **shape** of the language changed (new statement/expression kind), update `Pinescript.asdl` and regenerate ASDL nodes — ASDL schema.
4. Smoke-test with `parse` / `unparse` on a minimal first-party snippet before broader regression.

Invariants

1. **Never hand-edit `generated/`.** Patches belong in `resource/*.g4` or `resource/*Base.py`.
2. **Visitor contract.** Generated `PinescriptParserVisitor` method names follow ANTLR rule names; the builder subclass must track renames.
3. **Comments are off the default channel.** Annotation logic in `helper._collect_comment_nodes` iterates `token_stream.tokens` for `COMMENT` type after `fill()`, and keeps only texts containing `@`. `HASH_COMMENT` / `BACKTICKS` are remapped to that same token type.
4. **Indent unit is four spaces.** Soft-wrap lines that are not multiples of four are newline-elided, not nested blocks.

Worked examples

Minimal grammar-level mental model

```
//@version=6
indicator("x")
if close > open
    plot(1)
else
    plot(0)
```

Lexer emits NEWLINE, INDENT, DEDENT around the `if` bodies. Parser builds `if_structure` → builder yields `ast.If` with `body` / `orelse` statement lists.

v6 triple-quoted strings

Resource lexer rules (illustrative — see live `PinescriptLexer.g4`):

```
TRIPLE_DQ_STRING : '"""' ( ~'"' | '"' ~'"' | '""' ~'"' )* '"""' -> type(STRING) ;
TRIPLE_SQ_STRING : '\'\'\'' ( ~'\'' | '\'' ~'\'' | '\'\'' ~'\'' )* '\'\'\'' -> type(STRING) ;
```

ANTLR quoting pitfall: putting `"'''"` or `'"""'` as fragment literals can produce tool errors (`quote came as a complete surprise`). Factor starters when needed:

```
fragment TRIPLE_SQ_START: '\'' '\'' '\'';
```

Content and source indentation inside the triple quotes are preserved literally (no automatic dedent).

Typed names and `=` vs `:=` (0.3.9)

Pine lets type names occupy identifier positions (`int` is both a type and a legal `NAME` alternative via soft keywords). Alternatives must try the **typed** form first:

```
variable_declaration
    : declaration_mode? type_specification name_store
    | declaration_mode? name_store
    ;

simple_name_initialization: EXPORT? variable_declaration EQUAL expression;

simple_reassignment
    : assignment_target_attribute EQUAL expression
    | assignment_target_subscript EQUAL expression
    | primary_expression COLONEQUAL expression
    ;
```

Source	Node
`x = 1` / `var float x = 1.0` / `export const int N = 3`	`Assign` (`mode` / `type` / `export` as present)
`x := 2`	`ReAssign`
`obj.field = 3` / `arr[i] = 4`	`ReAssign` (attribute/subscript `=`)
`x += 1`	`AugAssign`

Putting `name_store` before `type_specification name_store` made `int x = 1` parse `int` as the variable and then fail on `x`. Do not invert those alternatives.

Case study: targeted lexer refresh (2026-07)

Full regeneration once produced a `PinescriptParser.py` whose context accessors diverged from what the hand-written builder expected (e.g. missing `template_spec_suffix()`), breaking even trivial parses. The practical recovery:

1. Edit **only** `resource/` grammar.
2. Generate the lexer in a clean temp directory (avoids nested `src/` mirror paths).
3. Copy **only** `PinescriptLexer.py` (+ refreshed `LexerBase.py`) into `generated/`.
4. Leave committed `PinescriptParser.py` and visitors untouched unless prepared to patch the builder in the same change.
5. Verify immediately:

```
from pynescript.ast.helper import parse, unparse
ast = parse('indicator("x")\ns = """\nfoo\n  bar\n"""\n')
assert "foo" in unparse(ast)
```

Failure modes

Failure	Cause	Mitigation
`mismatched input` after `"""`	Old lexer ATN without triple rules	Refresh generated lexer from resource
Full regen breaks builder	Parser context API drift	Selective copy; or update all `visit_*` together
`antlr4` not found	CLI not on PATH	Use hatch env or full path to `antlr4-cli`
Nested `generated/src/pynescript/...` junk	Running antlr with wrong `-o` cwd	Generate from clean temp + explicit `-o`
IndentationError mid-file	Mixed tabs/spaces or wrong nest	Align to 4-space blocks; check soft-wrap rules
Silent loss of hand patches	Edited `generated/`	Revert; re-apply in `resource/`

See also

- ASDL schema
- Builder
- Error model
- Helper API
- Missing features

ASDL SCHEMA

Algebraic AST definition in Pinescript.asdl, generated dataclasses, and how schema changes flow into the pipeline.

ASDL SCHEMA

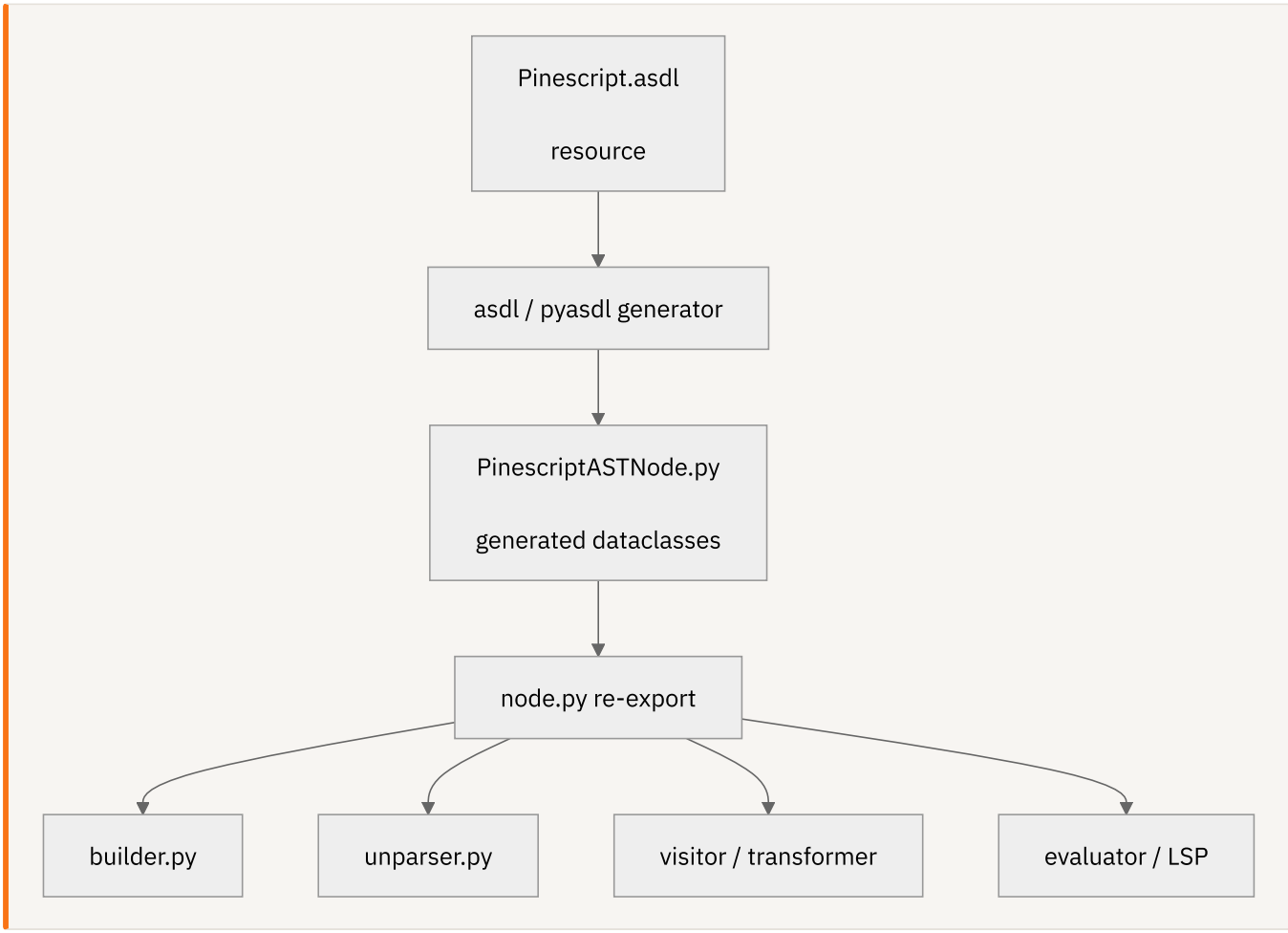
The abstract syntax tree is not free-form dicts. It is an **algebraic data type** declared in ASDL and materialized as Python dataclasses.

Abstract

`src/pynescript/ast/grammar/asdl/resource/Pinescript.asdl` defines the `Pinescript` module: roots (`Script`, `Expression`), statements, expressions (including control structures that are also expressions), operators, params, args, cases, and comments. Codegen writes `grammar/asdl/generated/PinescriptASTNode.py`. `src/pynescript/ast/node.py` re-exports that module so the rest of the codebase imports `from pynescript.ast import node as ast` (or `from pynescript.ast.node import Script`).

ASDL is the contract between **builder** (produces nodes), **evaluator/LSP** (consume nodes), and **unparser** (serializes nodes). Changing a field without regenerating and updating all three is a schema break.

Conceptual model



Each product sum type becomes a Python class hierarchy under `AST` with:

- `_fields` — logical children (walked by `iter_fields` / visitors)
- `_attributes` — location metadata (`lineno` , `col_offset` , `end_lineno` , `end_col_offset`) where declared

Interface surface

Module roots (`mod`)

Constructor	Fields	Use
`Script`	`stmt* body` , `string* annotations`	Full file (`mode="exec"`)
`Expression`	`expr body`	Single expression (`mode="eval"`)

Statements (`stmt`)

Constructor	Notable fields
`FunctionDef`	`name` , `param* args` , `body` , **`expr? returns`** , `method?` , `export?` , `annotations`
`TypeDef`	UDT `type Name` body (fields + methods as stmts)
`EnumDef`	`enum Name` members
`Assign`	`target` , optional `value` / `type` / `mode` (`Var` / `VarIp`), `export?` , `annotations`
`ReAssign`	`target := value` , or `obj.f =` / `a[i] =` (not bare `name =`)
`AugAssign`	`+=` ... ops
`Import`	`namespace/name/version` , optional `alias`
`Expr`	Expression statement
`Break` / `Continue`	Loop control

All `stmt` and most other attributed products carry location attributes.

Expressions (`expr`)

Operators and primaries:

- `BoolOp` , `BinOp` , `UnaryOp` , `Conditional` (`? :`), `Compare`
- `Call` , `Constant` , `Attribute` , `Subscript` , `Name` , `Tuple`
- **Structures as expressions:** `ForTo` , `ForIn` , `While` , `If` , `Switch` — same shapes can appear as values (Pine's expression-oriented control flow)
- **`Once(expr? test, stmt* body)`** — August 2026 `once [cond]` block. Lives under `expr` like other structures so `structure_statement` can wrap it in `Expr` , but it is **not** a usable value (parser forbids it on `=` / `:=` RHS)
- `Qualify` — `const` / `input` / `simple` / `series` applied to a type/expression
- `Specialize` — generic specialization form (`value` + `args`)

Auxiliaries

Product	Role
`decl_mode`	`Var` | `VarIp`
`type_qual`	`Const` | `Input` | `Simple` | `Series`
`expr_context`	`Load` | `Store`
`param`	Function/method parameter
`arg`	Call argument (optional keyword `name`)
`case`	Switch arm (`pattern?`, `body`)
`cmnt` / `Comment`	Comment with `value` and `kind` (annotation classification)

Operators are empty product constructors used as tags — the unparser maps class names back to tokens:

- Arithmetic / bool / compare: `Add`, `Sub`, `Mult`, `Div`, `Mod`, `And`, `Or`, `Eq`, `NotEq`, `Lt`, `LtE`, `Gt`, `GtE`
- Unary: `Not`, `UAdd`, `USub`, **`Invert`** (`~`)
- Bitwise (Pine v5+): **`BitAnd`**, **`BitOr`**, **`BitXor`**, **`LShift`**, **`RShift`**

Importing nodes

```
from pynescript.ast import node as ast

script = ast.Script(body=[
    ast.Expr(value=ast.Call(
        func=ast.Name(id="plot", ctx=ast.Load()),
        args=[ast.Arg(value=ast.Name(id="close", ctx=ast.Load()))],
    ))
])
```

Prefer constructing via `parse()` unless synthesizing trees for tests.

Internals

Paths

Path	Edit?
`src/pynescript/ast/grammar/asdl/resource/Pinescript.asdl`	**Yes** — schema source of truth
`src/pynescript/ast/grammar/asdl/generated/PinescriptASTNode.py`	**No** — regenerated
`src/pynescript/ast/grammar/asdl/tool/generate.py`	Generator entry
`src/pynescript/ast/node.py`	Thin re-export (`from .grammar.asdl.generated import *`)

Generated class shape

Excerpt of generated form (do not edit this file; shown for orientation):

```
@_dataclasses.dataclass
class FunctionDef(stmt):
    name: identifier = field(default=None)
    args: list[param] = field(default_factory=list)
    body: list[stmt] = field(default_factory=list)
    returns: expr | None = field(default=None)
    method: int | None = field(default=None)
    export: int | None = field(default=None)
    annotations: list[string] = field(default_factory=list)
    _fields: ClassVar[list[str]] = [
        "name", "args", "body", "returns", "method", "export", "annotations",
    ]
```

`stmt` base injects location `_attributes`. Hashability is forced to `object.__hash__` so nodes can live in sets/maps by identity when needed. `returns` is the optional leading type spec (`int f(x) => …`), not a runtime return *value*.

Regeneration

When the schema changes:

```
# Project entry (wraps asdl/tool/asdlgen.py, then ruff format)
python -m pynescript.ast.grammar.asdl.tool.generate
# equivalent: asdlgen.py resource/Pinescript.asdl -o generated/PinescriptASTNode.py
```

Do not invoke bare `pyasdl … -o generated/` — the project generator writes a **file** (`PinescriptASTNode.py`), not a directory of modules.

Then update, in the same change set:

1. `builder.py` construction sites
2. `unparser.py` `visit_*` methods
3. Evaluator expression/statement handlers
4. Any isinstance checks in LSP features

Dual nature of structures

ASDL places `If`, `ForTo`, `ForIn`, `While`, `Switch` under `expr`, while the builder often wraps structure-as-statement under `Expr(value=…)`. Annotation collection (`StatementCollector`) treats assign/reassign/augassign/expr that hold structure values specially so nested statements remain visible for `//@…` pairing.

Invariants

1. **Schema first.** New language constructs need an ASDL constructor (or a clear encoding into an existing one) before evaluator semantics.
2. **Fields vs attributes.** Visitors walk `_fields` only. Location is metadata; missing locations are fillable via `fix_missing_locations`.
3. **Optional flags as `int?`.** `method` and `export` are integer flags (truthy when set), not booleans — match builder assignments (`export = 1`).
4. **`FunctionDef.returns` is a type expression.** Engines must not `visit()` it as an executable value.

5. **Generated file is not a style playground.** Formatting/lint of `PinescriptASTNode.py` is overwritten on regen.

Worked examples

Map ASDL to a short script

```
//@version=6
f(x) => x + 1
a = f(close)
```

Approximate tree:

```
Script(
  annotations=["//@version=6", ...],
  body=[
    FunctionDef(name="f", args=[Param(name="x")], returns=None, body=[Expr(BinOp(...))]),
    Assign(target=Name("a", Store), value=Call(...)),
  ],
)
```

Dump fields programmatically

```
from pynescript.ast.helper import parse, iter_fields

tree = parse("x = 1")
for name, value in iter_fields(tree):
    print(name, type(value).__name__)
```

Failure modes

Failure	Cause
`AttributeError` on new field	Schema regenerated but consumer not updated
Builder constructs wrong product	ASDL and grammar out of sync
Unparser omits new node	Missing `visit_NewNode` → `generic_visit` may emit nothing useful
Hand-edit of generated nodes lost	Edited `generated/` instead of `.asdl`

See also

- Builder
- Visitor / transformer
- Unparser
- Grammar (ANTLR4)

19 · CORE

AST BUILDER

PinescriptASTBuilder: ANTLR parse-tree visitor that constructs ASDL nodes, locations, and comment kinds.

AST BUILDER

The builder is the semantic boundary between ANTLR's concrete parse tree and PYNE's ASDL abstract tree.

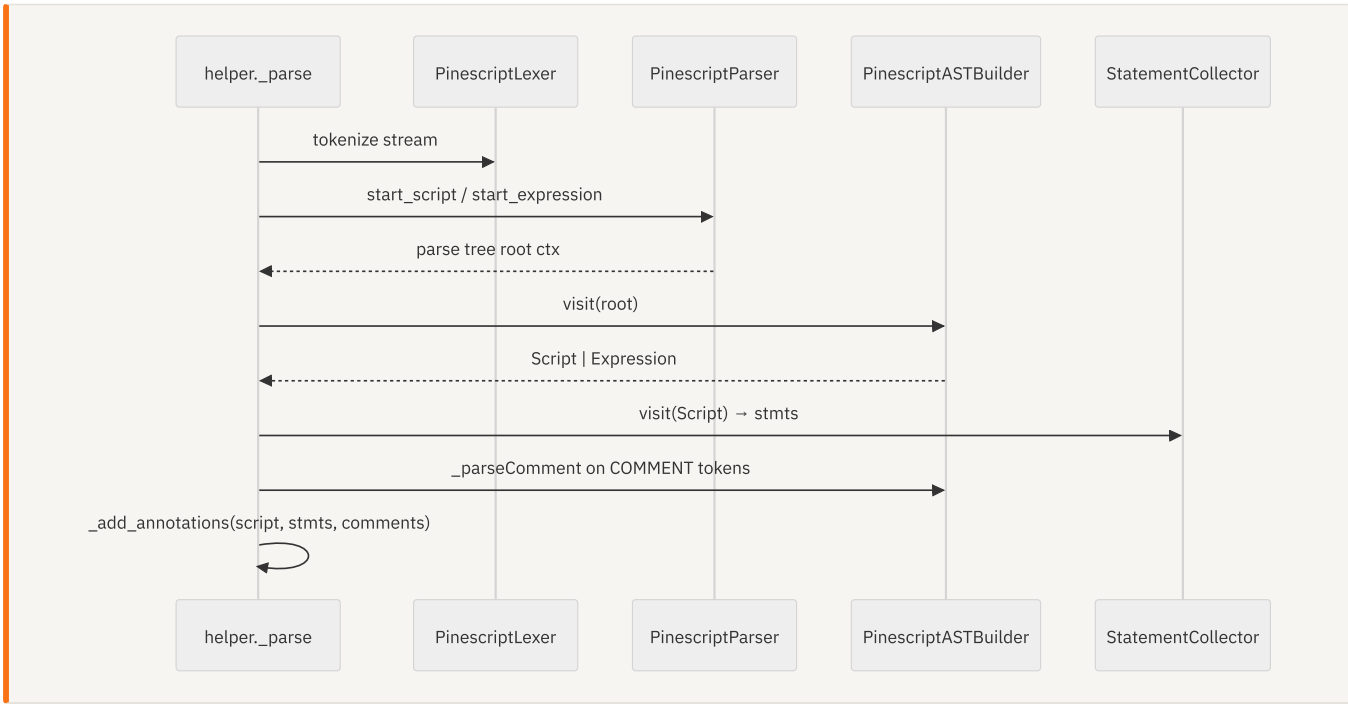
Abstract

`src/pynescript/ast/builder.py` defines three cooperating mixins:

1. **`PinescriptASTLocator`** — maps `ParserRuleContext.start/stop` tokens to `lineno` / `col_offset` / `end_*`.
2. **`PinescriptCommentParser`** — classifies `//…` text into annotation kinds (`@=S` , `@0F` , `#` , plain `//` , …).
3. **`PinescriptASTBuilder`** — subclasses generated `PinescriptParserVisitor` , implements `visit_*` for every meaningful rule, returns ASDL nodes.

`helper._parse` constructs a builder, runs `builder.visit(rule_context)` , then (in exec mode) uses `StatementCollector` + comment tokens to attach annotations. The builder itself does not walk the default token channel for comments; that is a post-pass.

Conceptual model



Interface surface

You rarely instantiate the builder directly; `parse()` does. For tools that already hold a parse tree:

```
from pynescript.ast.builder import PinescriptASTBuilder
from pynescript.ast.grammar.antlr4.parser import PinescriptParser
# ... construct parser, parse to ctx ...
node = PinescriptASTBuilder().visit(ctx)
```

Locator API

The live path writes attributes **directly**. Do not describe `_setLocations` as a dict apply — that was an older micro-opt leftover.

Method	Role
`_setLocations(node, ctx)`	**Hot path.** Sets `node.lineno` / `col_offset` / `end_*` from `ctx.start` / `ctx.stop` (no intermediate dict).
`_getLocations(ctx)`	Unused helper that still **returns** a four-key dict. Not called by visit methods.

End column calculation accounts for embedded newlines in the stop token text. The builder instance is **stateless**; `helper._SHARED_BUILDER` reuses one object for every parse.

Comment kinds

`_parseComment` returns `(kind, parts)`:

Pattern	Kind prefix	Suffix	Example
`//@version = 5`	`@=`	`S` (script)	version assignment
`//@description …` / `//@strategy_alert_message …`	`@0`	`S`	script description / alert
`//@function` / `//@returns`	`@0`	`F`	function docs
`//@type`	`@0`	`T`	type docs
`//@variable`	`@0`	`V`	variable docs
`//@param name …`	`@1`	`F`	param docs
`//@field name …`	`@1`	`T`	field docs
`//# region` / `endregion`	`#`	—	editor regions
other `//`	`//`	—	ordinary comment

`helper._add_annotations` filters by suffix when attaching to nodes.

Store context

`_set_store_ctx` walks assignment targets (`Name`, nested `Tuple`) and sets `ctx = Store()` so load/store distinction is available to later passes (Python-ast style).

Representative visit methods

Rule visitor	Emits
`visitStart_script`	`Script(body)`
`visitStart_expression`	`Expression(body)`
`visitFunction_declaration` / `visitMethod_declaration`	`FunctionDef` (`method=1` for methods; optional `returns=` type spec)
`visitType_declaration`	`TypeDef`
`visitEnum_declaration`	`EnumDef`
`visitVariable_declaration`	`Assign` shell (`target` / `type` / `mode`); parent fills `value`
`visitSimple_name_initialization` / `visitCompound_name_initialization`	`Assign` (`export=1` if `EXPORT`)
`visitSimple_reassignment` / `visitCompound_reassignment`	`ReAssign` (`:=` , or `=` on attr/subscript)
`visit*_*_augassignment`	`AugAssign`
`visitIf_structure` / `visitFor_*` / `visitWhile_*` / `visitSwitch_*` / `visitOnce_structure`	Structure expressions (`once` is statement-only)
`visit*_*_expression` (disjunction → … → primary)	Operator trees with correct associativity
`visitLiteral_*`	`Constant` (numbers via `ast.literal_eval` , strings, bools, colors)
`visitPrimary_expression_call`	`Call` + `Arg` list (positional/keyword)

Expression climbing follows the grammar's layered rules (`conditional` → `disjunction` → `conjunction` → **bitwise or/xor/and** → equality → inequality → **shift** → additive → multiplicative → unary (`~` / `not` / `+/-`) → primary).

`returns` is visited from `ctx.type_specification()` and must not be evaluated as a value by later engines.

Internals

Path

- Implementation: `src/pynescript/ast/builder.py`
- Visitor base: `src/pynescript/ast/grammar/antlr4/visitor.py` → generated `PinescriptParserVisitor`
- Node types: `src/pynescript/ast/node.py`
- Annotation pairing: `src/pynescript/ast/collector.py` + `helper._add_annotations`

Statement list flattening

`visitStatements` flattens each statement visitor's result: some rules return a **list** (simple multi-statement lines joined by commas), so the body is always a flat `list[stmt]` .

Export flag

When `EXPORT` is present on library declarations or name initialization, the builder sets `export = 1` on the corresponding node (integer flag per ASDL `int?`).

Assign vs `ReAssign`

`visitVariable_declaration` always builds `Assign` (optional `type` / `var` / `varip` mode). `=` after a bare or typed name is initialization. `visitSimple_reassignment` emits `ReAssign` for `obj.f =` / `a[i] =` / `name :=` — never for `name =`. See Grammar.

Field / enum members

Field definitions become `Assign`-like statements inside `TypeDef.body` (with optional `VarIp` mode). Enum members become assignment-shaped stmts under `EnumDef.body`. The unparser special-cases method members of types when pretty-printing.

Invariants

1. **Every constructed stmt/expr that has a source span should receive `_setLocations`** (direct attribute writes) so LSP diagnostics and `get_source_segment` work.
2. **Visitor method names must track generated rule names.** Renaming a parser rule without updating the builder is a silent `generic_visit` fallback (wrong or empty trees).
3. **Do not edit `builder.py.bak`.** Live builder is `builder.py` only (stale backups are forbidden under project policy).
4. **Grammar and builder co-evolve.** A resource-only grammar change that adds rules is incomplete until `visit_*` exists.

Worked examples

What `x = close + 1` becomes

Rough structure after build:

```
Assign(
  target=Name(id="x", ctx=Store()),
  value=BinOp(
    left=Name(id="close", ctx=Load()),
    op=Add(),
    right=Constant(value=1),
  ),
)
```

Export const initialization

Grammar: `EXPORT? variable_declaration EQUAL structure_expression`

Builder sets `assign.export = 1` when `ctx.EXPORT()` is present — required for library export of typed constants.

Typed UDF (`FunctionDef.returns`)

```
int ilog2(int n) =>
    // ...
```

`visitFunction_declaration` stores the leading `type_specification` on `FunctionDef.returns` (here a `Name(id="int")`). Methods do the same with `method=1`. The unparser prints it back as a leading type.

Debugging a missing visitor

```
from pynescript.ast.helper import parse, dump
print(dump(parse(your_snippet), indent=2))
```

If a subtree is `None` or a raw unexpected type, find the rule in `PinescriptParser.g4` and the matching `visitRule_name` in `builder.py`.

Failure modes

Symptom	Cause
`AttributeError: '…Context' object has no attribute '…'`	Full parser regen changed accessors; builder outdated
Locations all zero / missing	Forgot `_setLocations` on new visit method
Annotations not attached	Comment kind suffix wrong, or statement not yielded by `StatementCollector`
Tuples not assignable	Store context not propagated through `Tuple.elts`
Structure expression lost	Wrong rule branch between statement vs expression path

See also

- Grammar (ANTLR4)
- ASDL schema
- Helper API
- Error model

HELPER API

parse, sha256 LRU cache, unparse, dump, walk, literal_eval, and location utilities — the public AST surface.

HELPER API

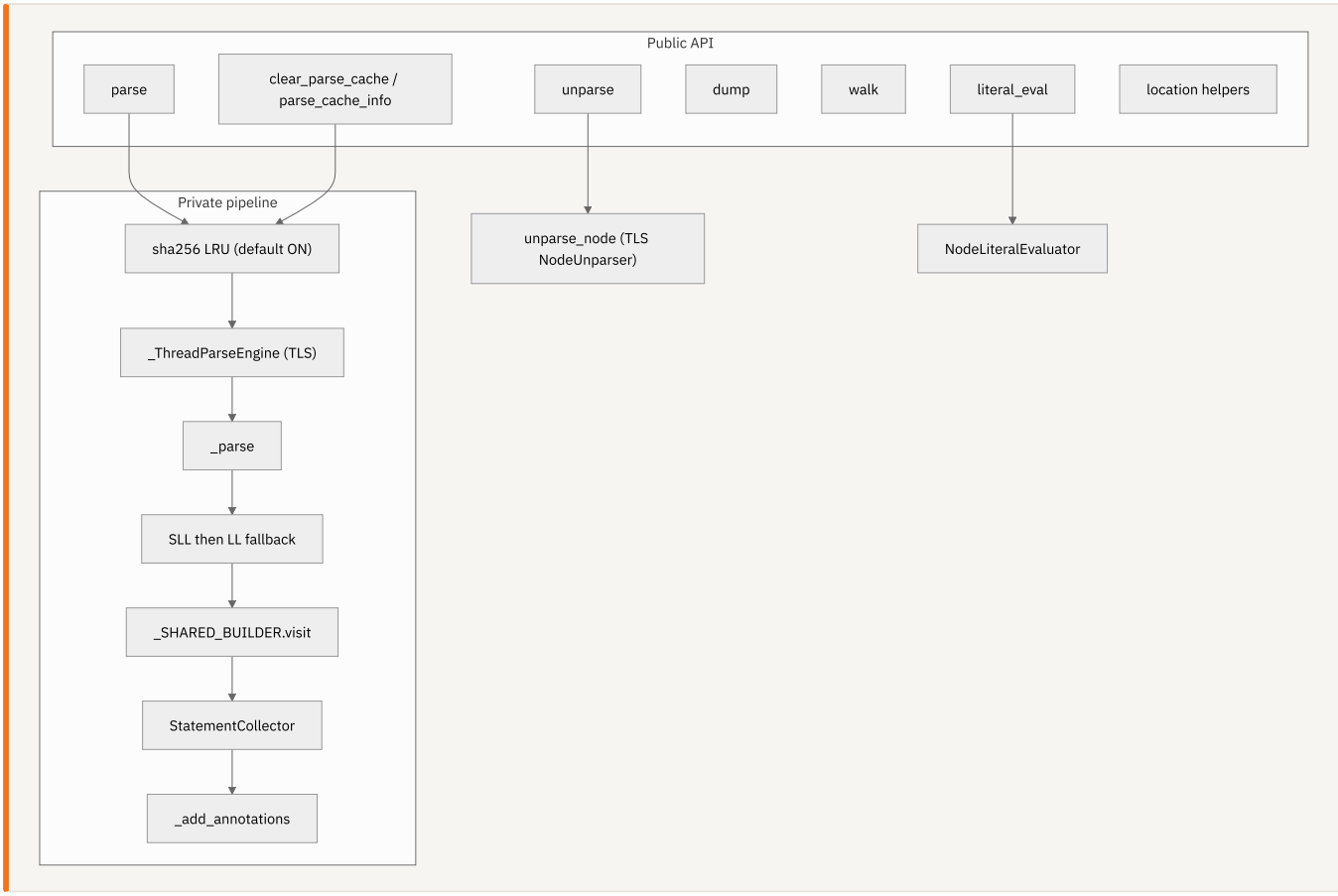
`src/pynescript/ast/helper.py` is the stable library face of the language core. It orchestrates ANTLR, the builder, annotations, and the unparser behind functions deliberately similar to CPython's `ast` module.

Abstract

Call `parse` to obtain an ASDL tree; `dump` / `walk` / `iter_*` to inspect it; `unparse` to regenerate source; `literal_eval` for constant-folding-safe evaluation of literal expressions. Location helpers (`copy_location`, `fix_missing_locations`, `get_source_segment`, `increment_lineno`) support tooling that rewrites or reports on trees. `clear_parse_cache` / `parse_cache_info` control the process-local sha256 LRU.

Everything higher in the stack (CLI, LSP, Pro API, workers) ultimately routes through this module for front-end work.

Conceptual model



Interface surface

`parse(source, filename="<unknown>", mode="exec") -> AST`

Parameter	Meaning
`source`	Full script or expression text
`filename`	Error reporting name; resolved to absolute path if the file exists
`mode`	`"exec"` → `start_script` → `Script`; `"eval"` → `start_expression` → `Expression`

Raises:

- `ValueError` — invalid mode
- `pynescript.ast.error.SyntaxError` — lexer/parser failure (via `PinescriptErrorListener`)

Side effects during exec mode:

1. Temporarily raises `sys.getrecursionlimit` to at least 5000.
2. Collects COMMENT tokens containing `@` and attaches annotations to eligible statements. If the source has no `@`, the annotation pass is skipped.

`filename` is **not** part of the cache key (only diagnostics).

Parse cache

Successful trees are stored in a process-local, thread-safe `OrderedDict` LRU keyed by `(sha256(source.encode("utf-8")), mode)`.

Control	Default	Meaning
`PYNE_PARSE_CACHE`	**on**	`0` / `false` / `off` / `no` disables
`PYNE_PARSE_CACHE_MAX`	`128`	Max entries (must be ≥ 1)
`clear_parse_cache()`	—	Drop all entries and reset hit/miss counters
`parse_cache_info()`	—	`{enabled, size, maxsize, hits, misses}`

On a cache **hit**, the same AST object is returned (after scrubbing stale `_pine_call_site` attrs). Treat it as read-only: a `NodeTransformer` or `increment_lineno` mutates the cached entry for every later caller. After intentional mutation, call `clear_parse_cache()` or parse with the cache disabled.

`unparse(node) -> str`

Delegates to `unparse_node` — a **per-thread** reused `NodeUnparser` (warm visitor cache). Public return value is still a new `str`. See Unparser.

`dump(node, *, annotate_fields=True, include_attributes=False, indent=None) -> str`

Recursive pretty-print of node trees. With `indent` (int spaces or string), multi-line layout is used for non-trivial nodes.

`literal_eval(node_or_string, context=None, data_feed=None, data_provider=None) -> Any`

- Strings are `parse(..., mode="eval")` first.
- Unwraps `Expression.body`.

- Uses `NodeLiteralEvaluator` — **not** full script evaluation. Non-literal graphs raise.

Tree navigation

Function	Behavior
`iter_fields(node)`	Yields `(name, value)` for set `_fields`
`iter_child_nodes(node)`	Direct AST children (incl. list items)
`walk(node)`	BFS over the full subtree

Location utilities

Function	Behavior
`copy_location(new, old)`	Copy lineno/col/end_* when both define the attribute
`fix_missing_locations(node)`	Fill gaps from parent defaults starting at `(1, 0)`
`increment_lineno(node, n=1)`	Shift all line numbers
`get_source_segment(source, node, *, padded=False)`	Slice original source by node span; `None` if incomplete ends

`__all__` export list

`clear_parse_cache` , `copy_location` , `dump` , `fix_missing_locations` , `get_source_segment` , `increment_lineno` , `iter_child_nodes` , `iter_fields` , `literal_eval` , `parse` , `parse_cache_info` , `unparse` , `walk` .

Internals

Path

`src/pynescript/ast/helper.py` — all public helpers above.

Related:

- `src/pynescript/ast/collector.py` — `StatementCollector` for annotation pairing
- `src/pynescript/ast/unparser.py` — `unparse_node` (thread-local `NodeUnparser`)
- `src/pynescript/ast/evaluator` — `NodeLiteralEvaluator` (literal_eval only)

Parse pipeline (`_parse`)

1. Validate `mode` ∈ `{exec, eval}` .
2. Raise recursion limit to ≥ 5000 if needed.
3. Bind a **thread-local** lexer + token stream + parser (`_ThreadParseEngine`). Process-wide reuse is unsafe (indent / token-stream bleed).
4. **SLL first** (`BailErrorStrategy`). On `ParseCancellationException` , reset and re-parse **LL** (`DefaultErrorStrategy`). Trees match pure-LL on success.
5. `PinescriptASTBuilder.visit` via the shared stateless builder.
6. Exec mode: if source contains `@` , collect statements + `@` comments and `_add_annotations` .

Default ANTLR console listeners are removed; only `PinescriptErrorListener.INSTANCE` is installed.

Annotation algorithm (`_add_annotations`)

1. Merge comments and statements; sort by `(lineno, col_offset)` .
2. Group consecutive comments vs statements.
3. Keep only kinds starting with `@` .
4. Script-level: first group members with kind ending in `S` → `script.annotations` .
5. Pair remaining comment groups with following statements; attach `F` / `T` / `V` filters to `FunctionDef` / `TypeDef` / `Assign` .

Comment **nodes** are not left in `Script.body` ; only string annotation lists are stored on targets. Collection itself skips tokens whose text has no `@` (plain `//` and `//# region` never become `Comment` nodes).

Stream helpers

Internal	Role
`_parse_inputstream`	`InputStream(source)` + `stream.name = filename`
`_parse_filestream`	`FileStream` for on-disk scripts
`_get_absolute_path`	Resolve existing paths; leave `<unknown>` alone

Invariants

1. **`mode` is a closed set.** Only `exec` and `eval` .
2. **Default listeners are removed.** Console ANTLR spam is replaced by raising project `SyntaxError` .
3. **`dump` requires an `AST` instance.** Non-nodes raise `TypeError` .
4. **Round-trip tests should use `parse` + `unparse`** , not raw builder access, so annotation behavior matches production.
5. **Cache identity is shared.** Do not mutate a cached tree unless you also `clear_parse_cache()` .
6. **`filename` is diagnostic-only** and is not part of the sha256 key.

Worked examples

Inspect a tree

```
from pynescript.ast.helper import parse, dump, walk

tree = parse("""
//@version=6
indicator("demo")
plot(close)
""")
print(dump(tree, indent=2))
print(sum(1 for _ in walk(tree)), "nodes")
```

Source segment

```
from pynescript.ast.helper import parse, get_source_segment

src = "a = 1\nb = a + 2\n"
tree = parse(src)
assign_b = tree.body[1]
print(get_source_segment(src, assign_b))  # "b = a + 2\n" or similar span
```

Literal evaluation

```
from pynescript.ast.helper import literal_eval

assert literal_eval("2 * 3 + 4") == 10
assert literal_eval("'hi'") == "hi"
```

Parse cache

```
from pynescript.ast.helper import parse, parse_cache_info, clear_parse_cache

src = 'indicator("x")\nplot(close)\n'
a = parse(src)
b = parse(src)
assert a is b  # same object on hit
print(parse_cache_info())  # enabled, size, maxsize, hits, misses
clear_parse_cache()
```

Failure modes

Failure	Notes
Deeply nested ternaries hit recursion	Mitigated by temporary limit ≥ 5000; pathological depth can still fail
`get_source_segment` returns `None`	Missing `end_lineno` / `end_col_offset`
`literal_eval` raises	Non-literal AST (names, calls beyond allowed set)
Filename `<unknown>` in errors	Expected when parsing pure strings without a path
Annotations missing	Comment not `@…` form, or not immediately preceding eligible stmt
Transformer “leaks” across `parse()` calls	Cache hit returned the same object you mutated — clear or disable cache
`PYNE_PARSE_CACHE_MAX` ignored	Non-integer or `< 1` falls back to 128

See also

- Builder
- Unparser
- Visitor / transformer
- Error model

UNPARSER

NodeUnparser: precedence-aware AST → Pine Script source regeneration.

UNPARSER

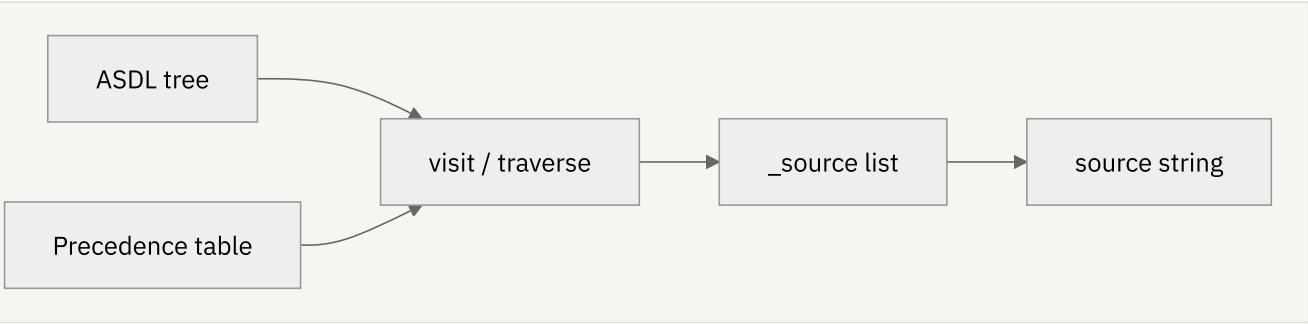
The unparser is the inverse of the builder: ASDL nodes become source text with correct operator parenthesization and Pine idioms (`=>` , `:=` , `var` / `varip` , `export` , method vs function).

Abstract

`src/pynescript/ast/unparser.py` implements `NodeUnparser(NodeVisitor)` and a `Precedence` enum. `helper.unparse(node)` calls `unparse_node` , which reuses a **thread-local** `NodeUnparser` (warm type-visitor cache; `visit` still resets the buffer each call). Output is **semantically faithful**, not a pretty-printer of original trivia: whitespace is normalized; comments that never became annotations are not reconstructed; annotation strings that *were* attached are re-emitted.

Round-trip tests (`parse → unparse → parse`) are the primary oracle for grammar/builder/unparser coherence.

Conceptual model



Parenthesization: higher `Precedence` values bind **tighter** (`TEST` loosest, `ATOM` tightest). A node is wrapped in `()` when the precedence **stored on it** (the parent context) is **strictly greater** than the node's own operator level. Equal levels do not wrap (left-associative chains).

Interface surface

```
from pynescript.ast.helper import parse, unparse

src = """
//@version=6
f(x) => x * 2
plot(f(close))
"""
print(unparse(parse(src)))
```

`Precedence` (low → high binding)

Matches `IntEnum` order in `unparser.py` :

Level	Operators / forms
`TEST`	ternary `? :` (weakest)
`OR`	`or`
`AND`	`and`
`BITOR`	`|`
`BITXOR`	`^`
`BITAND`	`&`
`EQ`	`==` `!=`
`INEQ` / `CMP`	`<` `>` `<=` `>=`
`SHIFT`	`<<` `>>`
`EXPR`	reserved slot (unused by current op tables)
`ARITH`	`+` `-`
`TERM`	`*` `/` `%`
`FACTOR` / `NOT`	unary `+` `-` `not` `~`
`ATOM`	names, literals, calls, subscripts

`Precedence.next()` steps one level tighter for right-hand associativity control. Bitwise tokens are emitted from `visit_BinOp` / `visit_UnaryOp` maps (`BitAnd` / `BitOr` / `BitXor` / `LShift` / `RShift` / `Invert`), not standalone `visit_BitAnd` methods.

Buffer helpers (internal but useful when subclassing)

Method	Role
`write(*text)`	Append fragments
`fill(text="")`	Newline + indent + text
`block()`	Indent context manager
`delimit(start, end)`	Wrap sub-output
`require_parens(prec, node)`	Conditional parentheses
`items_view` / `interleave`	Comma-separated lists
`buffered()`	Capture substring generation

Node coverage (high level)

Node	Emission sketch
`Script`	annotations then body
`FunctionDef`	optional `export` / `method`, optional `returns` type spec, `name(args) => body` (inline single `Expr` or indented block)
`TypeDef`	`type Name` with fields then methods
`EnumDef`	`enum Name` body
`Assign`	annotations, `export?`, `var` / `varip?`, type?, target `= value`
`ReAssign`	`target := value`
`AugAssign`	`target op= value`
`ForTo` / `ForIn` / `While` / `If` / `Switch`	Pine control syntax; `else if` chain collapse
`Import`	`import ns/name/version as alias?`
`BinOp` / `BoolOp` / `UnaryOp` / `Compare` / `Conditional`	precedence-aware, including bitwise `
`Call` / `Attribute` / `Subscript` / `Name` / `Constant` / `Tuple`	primaries

Internals

Path

- `src/pynescript/ast/unparser.py` — `Precedence`, `NodeUnparser`, `unparse_node`
- Entry: `helper.unparse` → `unparse_node` → thread-local `NodeUnparser.visit`

Visit vs traverse

`visit` resets `_source`, `_precedences`, and `_indent`, then calls `traverse`. `traverse` accepts lists (statement bodies) or single nodes, then dispatches via `_type_visitor_cache` (type-object keys, not class-name strings). This split lets statement lists and expression trees share code without double-buffering.

`visit_FunctionDef` emits `returns` (when set) as a leading type spec: `export method int name(args) => …`.

If / else if chains

`visit_If` detects `orelse == [Expr(If(...))]` and emits `else if` rather than nested `else` + `if` blocks — matching idiomatic Pine.

`visit_Once` emits `once` plus an optional test and an indented body.

Type body ordering

`visit_TypeDef` partitions body into field statements vs `FunctionDef` with `method` set, emitting fields first then methods.

Constants

String constants use JSON-style quoting helpers where needed; colors and numbers re-serialize from their Python values. Exact original quote style (single vs double) is not guaranteed.

Invariants

1. **Precedence tables must stay aligned with the parser's expression layers** (including bitwise `|` `^` `&` `<<` `>>` between `and` and comparisons). A mismatch causes either redundant parens (benign) or wrong binding after re-parse (severe).
2. **Annotations are part of the round-trip surface** when attached to Script/FunctionDef/TypeDef/Assign.
3. **Unparser does not type-check.** Ill-typed but well-shaped trees still print.
4. **Indent unit is four spaces** (`" " * self._indent`).

Worked examples

Precedence

```
from pynescript.ast.helper import parse, unparse

# Builder produces BinOp(Add, left=1, right=BinOp(Mult, 2, 3)) or inverse depending on parse
print(unparse(parse("1 + 2 * 3", mode="eval")))
# Expect something that re-parses to the same value: e.g. "1 + 2 * 3"
```

Function forms

```
# Single-expression body → f(x) => expr
# Multi-statement body → f(x) =>\n    stmt\n    stmt
# With return type → int f(x) => expr
# Method → method int f(self) => expr
```

Reassignment vs declaration

```
// Assign with mode → var float x = 1.0
// ReAssign → x := 2.0
// AugAssign → x += 1
```

Failure modes

Symptom	Cause
Missing `visit_*` output	Falls through `generic_visit` which only walks children — may emit empty string for leaf-like unhandled nodes
Extra parentheses everywhere	Over-conservative precedence
Re-parse differs in binding	Under-parenthesized output
Lost comments	Only annotation strings are stored; plain `//` comments are dropped after parse
Method printed as function	`FunctionDef.method` flag not set by builder

See also

- Helper API
- Builder
- ASDL schema
- Visitor / transformer

VISITOR & TRANSFORMER

NodeVisitor and NodeTransformer: read-only traversal vs in-place AST rewrite with caching dispatch.

VISITOR & TRANSFORMER

Tree walks in PYNE follow the same dual pattern as CPython's `ast` module: a visitor for analysis, a transformer for structural rewrite.

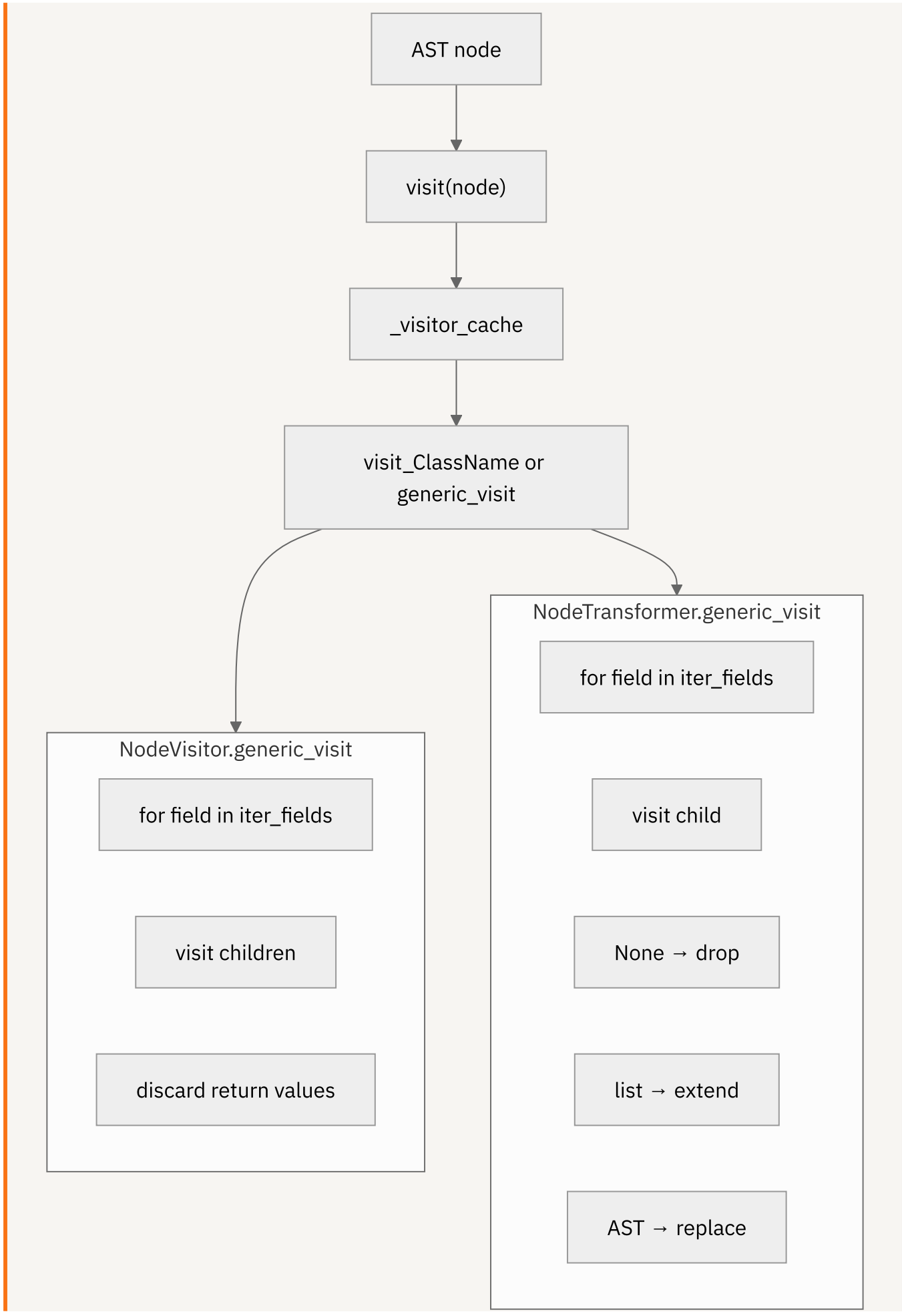
Abstract

- **`NodeVisitor`** (`src/pynescript/ast/visitor.py`) — dispatch `visit_<ClassName>`, default `generic_visit` walks children via `iter_fields`. Method lookups are cached **per visitor instance, keyed by `type(node)`** (the type object, not the class-name string).
- **`NodeTransformer`** (`src/pynescript/ast/transformer.py`) — same dispatch, but `generic_visit` **replaces** list items and fields according to return values (`None` removes, list splices, AST replaces).

Specialized visitors elsewhere:

Class	Path	Role
`PinescriptASTBuilder`	`builder.py`	CST visitor (ANTLR), not `NodeVisitor`
`NodeUnparser`	`unparser.py`	Source emission
`StatementCollector`	`collector.py`	Yield statements for annotations
Evaluator classes	`evaluator/*.py`	Runtime interpretation

Conceptual model


AST node
visit(node)
_visitor_cache
visit_ClassName or
generic_visit
NodeVisitor.generic_visit
for field in iter_fields
visit children
discard return values
NodeTransformer.generic_visit
for field in iter_fields
visit child
None → drop
list → extend
AST → replace

Interface surface

`NodeVisitor`

```
from pynescript.ast.visitor import NodeVisitor
from pynescript.ast.helper import parse

class CallCounter(NodeVisitor):
    def __init__(self):
        super().__init__()
        self.count = 0

    def visit_Call(self, node):
        self.count += 1
        self.generic_visit(node)

tree = parse("plot(ta.sma(close, 14))")
c = CallCounter()
c.visit(tree)
assert c.count >= 2
```

Method	Contract
`visit(node)`	Resolve `visit_<type(node).__name__>` with cache; fallback `generic_visit`
`generic_visit(node)`	Recurse into AST fields and lists of AST

Returns are whatever the override returns (analysis often returns `None`).

`NodeTransformer`

```
from pynescript.ast import node as ast
from pynescript.ast.transformer import NodeTransformer
from pynescript.ast.helper import parse, unparse

class RenameClose(NodeTransformer):
    def visit_Name(self, node: ast.Name):
        if node.id == "close":
            return ast.Name(id="open", ctx=node.ctx)
        return node

tree = parse("plot(close)")
tree = RenameClose().visit(tree)
assert "open" in unparse(tree)
```

Return value semantics when transforming a **list parent** (e.g. `body`):

Return	Effect
`None`	Remove this element
`AST`	Replace element
non-AST iterable	Splice multiple nodes in place of one
same node	Keep

For a **single AST field**, `None` deletes the attribute (`delattr`); otherwise `setattr` replaces.

`StatementCollector`

Generator-style visitor used only for annotation attachment:

- Yields `FunctionDef` , `TypeDef` , `EnumDef` , `Assign` , `ReAssign` , `AugAssign` , `Import` , `Expr` , `Break` , `Continue` .
- Descends into nested structures and into structure-valued assignments.
- Does **not** yield the structure nodes themselves as statements when they appear only as values — it yields their inner bodies.

```
from pynescript.ast.collector import StatementCollector
from pynescript.ast.helper import parse

stmts = list(StatementCollector().visit(parse("f() => 1\nx = 2")))
```

Internals

Paths

File	Symbols
`src/pynescript/ast/visitor.py`	`NodeVisitor`
`src/pynescript/ast/transformer.py`	`NodeTransformer`
`src/pynescript/ast/collector.py`	`StatementCollector` , `Structure` tuple
`src/pynescript/ast/helper.py`	`iter_fields` , `iter_child_nodes` , `walk`

Dispatch cache

```
cls = type(node)
visitor = cache.get(cls) or getattr(self, "visit_" + cls.__name__, generic_visit)
cache[cls] = visitor
```

`NodeUnparser` keeps a **second** type-keyed cache (`_type_visitor_cache`) used by `traverse` , bypassing `NodeVisitor.visit` so it does not reset the source buffer mid-render.

Subclass instances that dynamically add methods after first visit of a type will not see them unless the cache is cleared — prefer defining methods on the class body.

In-place lists

Transformers mutate list fields with `old_value[:] = new_values` rather than rebinding, so aliased references to the same list see updates.

Invariants

1. **Always call** `super().__init__()` on subclasses so `_visitor_cache` exists.

2. **Transformers should return the node** from overrides that use `generic_visit` (the default returns `node` after rewriting children).
3. **Do not assume identity stability** after a transformer pass if you replaced roots.
4. **`walk` is not a visitor** — it is a pure BFS helper without dispatch.

Worked examples

Strip all `Expr` statements that call `plot`

```
from pynescript.ast import node as ast
from pynescript.ast.transformer import NodeTransformer
from pynescript.ast.helper import parse, unparse

class DropPlots(NodeTransformer):
    def visit_Expr(self, node):
        if isinstance(node.value, ast.Call):
            func = node.value.func
            if isinstance(func, ast.Name) and func.id == "plot":
                return None
        return self.generic_visit(node)

tree = DropPlots().visit(parse("x = 1\nplot(x)\n"))
print(unparse(tree))
```

Collect all string constants

```
from pynescript.ast.visitor import NodeVisitor
from pynescript.ast.helper import parse

class Strings(NodeVisitor):
    def __init__(self):
        super().__init__()
        self.found = []

    def visit_Constant(self, node):
        if isinstance(node.value, str):
            self.found.append(node.value)

s = Strings()
s.visit(parse('indicator("hello")'))
```

Failure modes

Failure	Cause
`AttributeError: _visitor_cache`	Forgot `super().__init__()`
Transformer silently no-ops	Override returns nothing (`None`) unintentionally → node deleted
Infinite recursion	`visit_X` calls `self.visit(node)` on same node without change
Partial rewrite	Overrode `visit_*` without `generic_visit` — children untouched
Collector misses nested assign	Structure not in `Structure` tuple and not assigned as value

See also

- Helper API
- Unparser
- Builder
- ASDL schema

23 · CORE

TYPE SYSTEM

Pine v6 type model: qualifiers, builtins, collections, UDTs, TypeRegistry, and MethodResolver.

TYPE SYSTEM

PYNE's type layer models Pine's *qualified* types and user-defined types (UDTs) as Python objects usable by analysis and evaluation — not as a full independent typechecker IR (yet), but as the shared vocabulary for "what kind of value is this?"

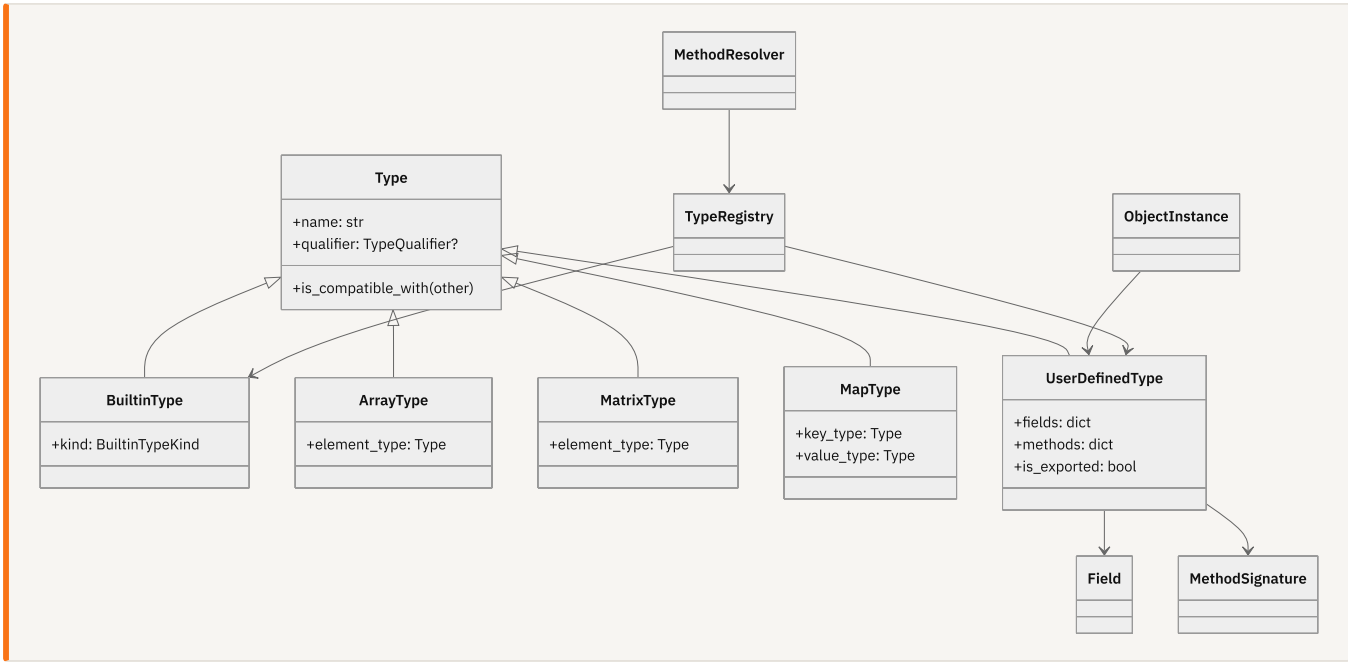
Abstract

`src/pynescript/ast/type_system.py` defines:

- **Qualifiers** — `const`, `simple`, `series`, `input` (`TypeQualifier`)
- **Builtin kinds** — `int`, `float`, `bool`, `string`, `color`, `na`, `enum`
- **Composite types** — `array<T>`, `matrix<T>`, `map<K,V>`
- **UDTs** — `UserDefinedType` with `Field`s and `MethodSignature`s
- **Runtime instances** — `ObjectInstance` for UDT values
- **Registry / resolution** — `TypeRegistry`, `MethodResolver` (`.new`, `.copy`, user methods)

ASDL already has `Qualify` / `Specialize` expression forms and `type_qual` constructors; the type system module is the **semantic** counterpart used when evaluating UDT construction and when tools reason about types.

Conceptual model



Interface surface

Qualifiers

```
from pynescript.ast.type_system import TypeQualifier

TypeQualifier.CONST   # compile-time constant
TypeQualifier.SIMPLE  # bar-invariant
TypeQualifier.SERIES  # per-bar series
TypeQualifier.INPUT   # input.* style parameter
```

`Type.__str__` renders `"series float"` when a qualifier is set.

Builtins and factories

```
from pynescript.ast.type_system import (
    BuiltinTypeKind,
    int_type,
    float_type,
    bool_type,
    string_type,
    color_type,
    TypeQualifier,
)

t = float_type(TypeQualifier.SERIES)  # "series float"
```

Factory	Kind
`int_type`	`INT`
`float_type`	`FLOAT`
`bool_type`	`BOOL`
`string_type`	`STRING`
`color_type`	`COLOR`

Collections

```
from pynescript.ast.type_system import ArrayType, MatrixType, MapType, int_type, string_type

ArrayType(int_type())                # array<int>
MatrixType(float_type())             # matrix<float>
MapType(string_type(), int_type())   # map<string, int>
```

UDTs

```
from pynescript.ast.type_system import UserDefinedType, Field, MethodSignature, float_type, int_ty

point = UserDefinedType("point")
point.add_field(Field("x", float_type(), default_value=0.0))
point.add_field(Field("y", float_type(), default_value=0.0, varip=False))
point.add_method(MethodSignature("length", [], return_type=float_type()))
```

`Field` supports optional `default_value` and `varip` (mirrors `varip` fields in type declarations).

`TypeRegistry`

```
from pynescript.ast.type_system import TypeRegistry

reg = TypeRegistry()
reg.register_type(point)
assert reg.is_builtin_type("float")
assert reg.is_user_defined_type("point")
assert reg.get_type("float") is not None
```

Builtins are seeded in `_init_builtin_types` (`int`, `float`, `bool`, `string`, `color`, `na`, `enum`).

`ObjectInstance` + `MethodResolver`

```
from pynescript.ast.type_system import ObjectInstance, MethodResolver

inst = ObjectInstance(point)
inst.set_field("x", 1.5)
resolver = MethodResolver(reg)
# .new / .copy handled specially
copy = resolver.resolve_method(inst, "copy", [])
```

Method name	Behavior
`new`	Construct instance; positional args fill fields in declaration order
`copy`	Shallow copy of field dict
other	Lookup `UserDefinedType.methods`; missing → `AttributeError`

Unknown field get/set on `ObjectInstance` raises `AttributeError` with type name context.

Relation to ASDL `type_qual`

ASDL	`TypeQualifier`
`Const`	`CONST`
`Input`	`INPUT`
`Simple`	`SIMPLE`
`Series`	`SERIES`

Builder produces `Qualify(qualifier=…, value=…)` for annotated types in the AST; the type system module is used when those need runtime/type-registry meaning (especially UDTs in builtins/evaluator).

Internals

Path

`src/pynescript/ast/type_system.py` — sole definition site for the classes above. **Not** star-exported from `pynescript.ast`; import the module directly.

Consumers (outside this doc's page set, for orientation):

- Evaluator UDT / collection code (`evaluator/builtins/`, `matrix_evaluator`, etc.)
- Future/static analysis hooks (LSP hover may grow toward this model)

Compatibility

`Type.is_compatible_with` is currently shallow: `BuiltinType` compares by equality; the base method returns `False` for other pairs. Treat richer subtyping (series/simple lattice, numeric promotion) as **not fully encoded** here — bar-loop semantics still own much of the coercion behavior.

`na` and `enum`

Registered as builtin kinds for name lookup. Enum *definitions* in scripts appear as AST `EnumDef`; linking enum members into `BuiltinTypeKind.ENUM` instances is evaluator-side work.

Invariants

1. **Registry lookup prefers builtins** over UDTs of the same name.
2. **Field presence is schema-true.** Setting an undeclared field is an error (no open bags).
3. **`.new` does not type-check arguments** against field types in the resolver — it assigns positionally.
4. **`varip` is metadata on `Field`**, not a separate type qualifier enum member.

Worked examples

Model a simple UDT from Pine

```
type candle
    float o
    float h
    float l
    float c
```

```
from pynescript.ast.type_system import UserDefinedType, Field, float_type, TypeRegistry, MethodRes

candle = UserDefinedType("candle")
for name in "o h l c".split():
    candle.add_field(Field(name, float_type()))
reg = TypeRegistry()
reg.register_type(candle)
c = MethodResolver(reg).resolve_method(ObjectInstance(candle), "new", [1, 2, 0.5, 1.5])
assert c.get_field("h") == 2
```

Qualified series float

```
from pynescript.ast.type_system import float_type, TypeQualifier
assert str(float_type(TypeQualifier.SERIES)) == "series float"
```

Failure modes

Failure	Cause
`AttributeError: Field '…' not found`	Typo or outdated UDT schema
`AttributeError: Method '…' not found`	Method never `add_method`'d; not `new` / `copy`
Silent type confusion	`is_compatible_with` too weak — do not rely on it alone for safety
Name shadowing	UDT registered with a builtin name is unreachable via `get_type`

See also

- ASDL schema — `Qualify`, `TypeDef`, `EnumDef`
- Runtime — bar-loop use of values
- Builder — parsing type declarations
- Linter — style-level checks (not full typing)

24 · CORE

LINTER

PineLinter static rules: syntax via parse, version, deprecations, naming, style codes.

LINTER

The core linter is a lightweight static checker that sits on the parse pipeline plus regex rules. It is not a full dataflow analyzer; it is the first automated gate for “does this even look like modern Pine?”

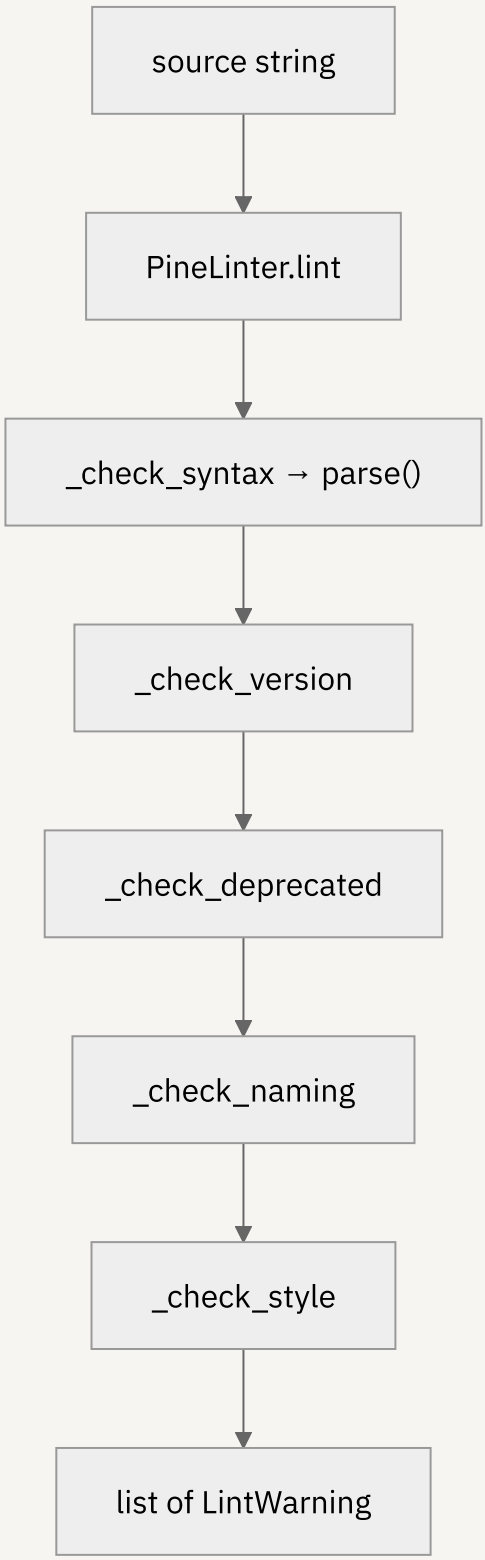
Abstract

`src/pynescript/ast/linter.py` exports:

- `LintWarning` — dataclass (`code`, `message`, `line`, `column`, `severity`)
- `PineLinter` — stateful runner accumulating warnings
- `lint_script(source, filename)` / `lint_file(filepath)` — convenience entry points

Rules run in fixed order: **syntax → version → deprecated patterns → naming → style**. Syntax failures become severity `"error"` with code `E001`; other rules are mostly `"warning"`.

Conceptual model



Interface surface

```
from pynescript.ast.linter import lint_script, PineLinter, LintWarning

warnings = lint_script("""
indicator("x")
plot(close)
""")
for w in warnings:
    print(w)  # warning: [W001] Missing @version ... at line 1
```

`LintWarning`

Field	Type	Meaning
`code`	`str`	Stable id (`E001` , `W001` , `C002` , …)
`message`	`str`	Human-readable explanation
`line`	`int | None`	1-based when known
`column`	`int | None`	Reserved / optional
`severity`	`str`	`"warning"` (default) or `"error"`

`__str__` format: `{severity}: [{code}] {message} at {location}` .

`PineLinter.lint(source, filename="<input>") -> list[LintWarning]`

Resets `self.warnings` , runs all checks, returns the list (also stored on the instance).

File helper

```
from pynescript.ast.linter import lint_file

issues = lint_file("strategies/mean_reversion.pine")
```

Reads UTF-8 text then delegates to `lint_script` .

Rule catalog

Syntax

Code	Severity	Condition
`E001`	error	`parse(source, filename)` raises any exception

Does not attempt recovery; one syntax error check per run. Location is taken from `pynescript.ast.error.SyntaxError.details` (`lineno` , `offset`) when present; otherwise a `line N` regex on the exception text. The warning message prefers the short `.message` attribute so caret dumps do not land in chips.

Version

Code	Condition
`W001`	No `//@version = N` (flexible whitespace) match
`W002`	Version integer `< 5` (deprecated; suggest v6)

Pattern: `//\s*@version\s*=\s*(\d+)`.

Deprecated patterns (`_check_deprecated`)

Code	Pattern (simplified)	Advice
`W101`	`security('EXCHANGE:SYM'…)` style	Prefer `request.security()` with explicit params
`W102`	*(retired)* `plot(… style=plot.style_histogram`	Was: consider `plotcandle`. Code kept greppable; the rule no longer fires.
`W103`	*(retired)* `var int name = na`	Was: prefer `0` for type safety. Code kept greppable; the rule no longer fires.

Matches set `line` from prefix newline count. Case-insensitive search. Live deprecated rule is `W101` only.

Naming (`_check_naming`)

Code	Condition
`C001`	LHS of `name = ta.…` starts with lowercase — message suggests camelCase via `_to_camel`

Heuristic only: line-local regex `(\w+)\s*=\s*ta\.`.

Style (`_check_style`)

Code	Condition
`C002`	Line length (rstrip) > 120
`C003`	*(retired)* Line matches `^\s+if\s+` — was “single-line if without braces”; code kept greppable
`C004`	File does not end with newline

Internals

Path

`src/pynescript/ast/linter.py`

Dependencies:

- `pynescript.ast.parse` (re-exported path via `from pynescript.ast import parse` — subject to the helper sha256 LRU)
- Standard library `re`, `dataclasses`

Design stance

The linter intentionally uses **regex over AST** for several rules so it still partially works when parse fails (version/deprecations/style still run after a failed syntax check — note: `_check_syntax` records `E001` but does not abort the pipeline). That means:

- False positives/negatives on fancy formatting are possible.
- Deep semantic issues (wrong series type, undefined names) belong to evaluator/LSP diagnostics, not these codes.

Integration points

CLI, editor save hooks, and CI can call `lint_script` without spinning up the full evaluator. LSP diagnostics may layer additional semantic checks beyond this module.

Invariants

1. `lint()` **always clears prior warnings** on that instance before running.
2. **Codes are stable public strings** — treat renames as breaking for tooling.
3. **Syntax errors are not fatal to the function** — you get `E001` plus any later regex hits.
4. **No mutation of source** — pure analysis.

Worked examples

Clean modern script

```
from pynescript.ast.linter import lint_script

src = """//@version=6
indicator("ok")
length = 14
basis = ta.sma(close, length)
plot(basis)
"""
assert lint_script(src) == []  # may still flag C001 if naming heuristic fires
```

Note: `basis = ta.sma(...)` triggers `C001` under the current rule (lowercase LHS). Prefer documenting that heuristic when teaching style.

Missing version

```
ws = lint_script('indicator("x")\nplot(close)\n')
assert any(w.code == "W001" for w in ws)
```

Programmatic filter

```
errors = [w for w in lint_script(src) if w.severity == "error"]
if errors:
    raise SystemExit(1)
```

Failure modes

Issue	Explanation
`C003` on legitimate indented `if` blocks	Rule retired in 0.6.0; the code no longer fires (was a coarse `^\s+if\s+` heuristic)
`C001` noise	Many valid snake_case or lower identifiers
`E001` still not a caret dump	Linter stores the short `.message` plus line/column; catch `pynescript.ast.error.SyntaxError` for the full caret `__str__`
Encoding errors in `lint_file`	Non-UTF-8 files raise at read time
False security deprecation	Pattern looks for quoted `EXCHANGE:SYM` form only

See also

- Helper API — underlying `parse`
- Error model
- LSP diagnostics
- End-user troubleshooting

ERROR MODEL

SyntaxError, IndentationError, SyntaxErrorDetails, and PinescriptErrorListener bridge from ANTLR to diagnostics.

ERROR MODEL

Parse failures in PYNE are not bare ANTLR console messages. They are structured exceptions with file, line, column, source excerpt, and a caret — suitable for CLI, tests, and LSP adapters.

Abstract

Two layers cooperate:

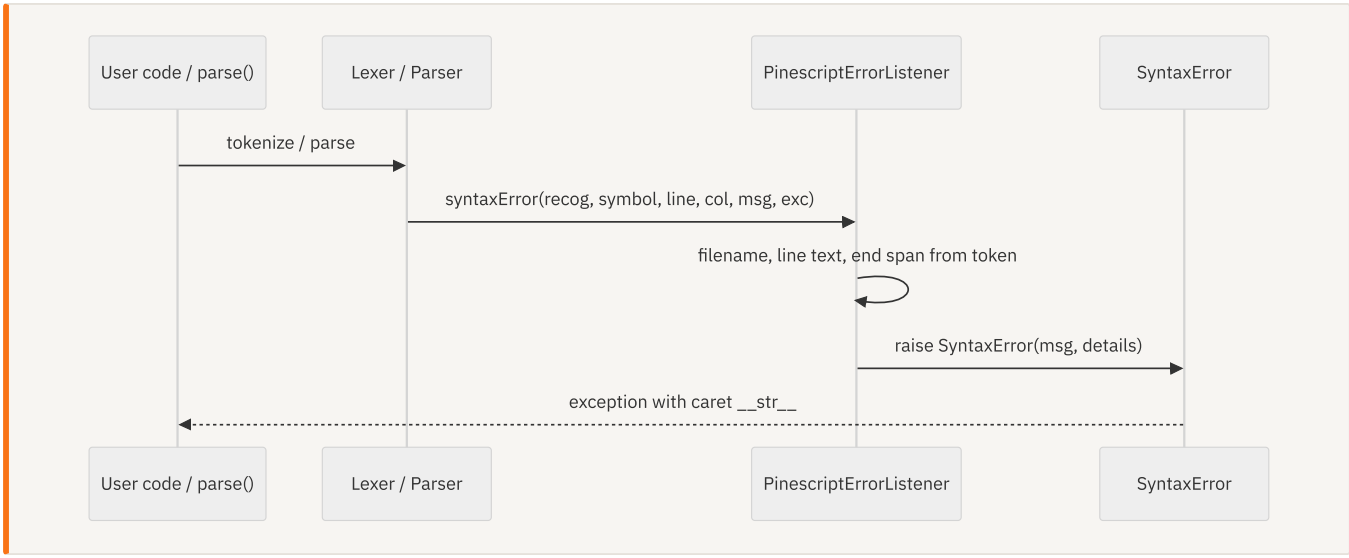
1. **`src/pynescript/ast/error.py`** — `SyntaxErrorDetails`, `SyntaxError`, `IndentationError`.
2. **`src/pynescript/ast/grammar/antlr4/error_listener.py`** — `PinescriptErrorListener` implements ANTLR's `ErrorListener.syntaxError`, builds details from the recognizer + offending token, and **raises** the project `SyntaxError`.

`helper._parse` installs this singleton listener on both lexer and parser after removing default listeners. Indentation problems raised from `PinescriptLexerBase` use the same exception hierarchy (`IndentationError` subclass).

Note: these names intentionally shadow Python builtins (`SyntaxError`, `IndentationError`) inside the package — import carefully:

```
from pynescript.ast.error import SyntaxError as PineSyntaxError
```

Conceptual model



Interface surface

`SyntaxErrorDetails` (NamedTuple)

Field	Type	Meaning
`filename`	`str`	Path or `<unknown>`
`lineno`	`int`	1-based line
`offset`	`int`	0-based column
`text`	`str`	Source line (or excerpt)
`end_lineno`	`int | None`	Multi-line span end
`end_offset`	`int | None`	End column

`SyntaxError(Exception)`

```
class SyntaxError(Exception):
    def __init__(self, message: str, *details): ...
```

`details` may be:

- a single `SyntaxErrorDetails` instance, or
- positional components matching the NamedTuple fields.

Attributes: `message`, `details`.

`__str__` renders:

```
{message}
  File "{filename}", line {lineno}
    {stripped_line}
    {spaces}^
```

Offset is adjusted when the displayed line is `lstrip()`'d so the caret still points at the logical column.

`IndentationError(SyntaxError)`

Empty subclass for indent-stack failures from the lexer base (wrong nest, inconsistent levels). Catch either specifically or via the base `SyntaxError`.

`PinescriptErrorListener`

```
from pynescript.ast.grammar.antlr4.error_listener import PinescriptErrorListener

listener = PinescriptErrorListener.INSTANCE  # singleton
lexer.removeErrorListeners()
lexer.addErrorListener(listener)
parser.removeErrorListeners()
parser.addErrorListener(listener)
```

Method	Role
`syntaxError(...)`	Build details; raise `SyntaxError`
`_getFilenameFrom(recognizer)`	Walk Parser → TokenStream → Lexer → InputStream/FileStream
`_getInputTextFrom(recognizer, lineno)`	Full buffer or single line
`_splitLines`	Portable newline split (`\r\n` / `\n` / `\r`)

If the ANTLR exception `e` is already a project `SyntaxError`, its `details` are updated; otherwise the new error's `__cause__` is set to `e`.

Internals

Paths

Path	Role
`src/pynescript/ast/error.py`	Exception types
`src/pynescript/ast/grammar/antlr4/error_listener.py`	ANTLR bridge
`src/pynescript/ast/grammar/antlr4/resource/PinescriptLexerBase.py`	May raise `IndentationError` / `SyntaxError` during tokenization
`src/pynescript/ast/helper.py`	Wires listener; maps filename onto streams

Filename resolution order (InputStream)

1. `getSourceName()` if present
2. `sourceName` attribute
3. `input_stream.name` (set by helper to the user filename)
4. `"<unknown>"`

`FileStream` uses `fileName`.

End span from offending token

```
symbol_len = stop - start + 1
end_lineno = token.line + newlines_in_text
end_offset = adjusted if multiline else column + symbol_len
```

Mirrors the builder's location math so diagnostics and AST spans speak the same coordinate system.

Relationship to linter `E001`

`PineLinter._check_syntax` catches **any** `Exception` from `parse` and wraps it as `LintWarning(code="E001", severity="error")`. It **does** copy `details.lineno` / `details.offset` onto the warning when present, and prefers the short `.message` over the caret `__str__`. For the full caret rendering, catch `pynescript.ast.error.SyntaxError` at the call site instead of only reading linter output.

LSP / tooling

Adapters typically map:

Exception field	LSP `Diagnostic`
`lineno` / `offset`	`Range.start`
`end_lineno` / `end_offset`	`Range.end`
`message`	`message`
IndentationError	same, possibly different `code`

Invariants

1. **Default ANTLR listeners must stay removed** on the parse path — otherwise errors print twice and may not raise.
2. **Singleton listener is stateless** enough to share (`INSTANCE`); thread-local needs would require new instances.
3. **Raising aborts parse immediately** — no error recovery producing a partial tree through the public helper.
4. **Package `SyntaxError` ≠ builtin** — `except SyntaxError` without import may catch the wrong class.

Worked examples

Catching a parse error

```
from pynescript.ast.helper import parse
from pynescript.ast.error import SyntaxError as PineSyntaxError

try:
    parse("f( =>\n", filename="bad.pine")
except PineSyntaxError as e:
    print(e)  # caret form
    print(e.details.lineno, e.details.offset)
    print(e.details.filename)
```

Distinguishing indentation

```
from pynescript.ast.error import IndentationError as PineIndentError
from pynescript.ast.error import SyntaxError as PineSyntaxError

try:
    parse("if true\nplot(1)\n")  # missing indent under if — may indent-error depending on tokens
except PineIndentError as e:
    print("indent", e.message)
except PineSyntaxError as e:
    print("syntax", e.message)
```

Manual construction (tests)

```
from pynescript.ast.error import SyntaxError, SyntaxErrorDetails

details = SyntaxErrorDetails("t.pine", 3, 4, "    plot(\n", 3, 9)
err = SyntaxError("missing RPAR", details)
assert "t.pine" in str(err)
assert "^" in str(err)
```

Failure modes

Failure	Cause
`TypeError: unexpected type of input`	Listener received a recognizer/stream combo it does not support
Caret misaligned	Tabs vs spaces; display strips leading WS while offset counts raw
Filename always `<unknown>`	Stream name not set (bypassed helper)
Exception swallowed as generic `E001`	Only used linter path (line/column may still be filled from `details`)
`except SyntaxError` misses Pine errors	Caught builtin only
Partial trees	Not available via `parse()` after error — redesign would need recovery listeners

See also

- Helper API
- Grammar (ANTLR4)
- Linter
- LSP diagnostics
- Language core hub

RUNTIME

Bar-loop evaluator — series caps, incremental TA, alerts, fill export, foreign request.security → na, and interpret↔compile parity.

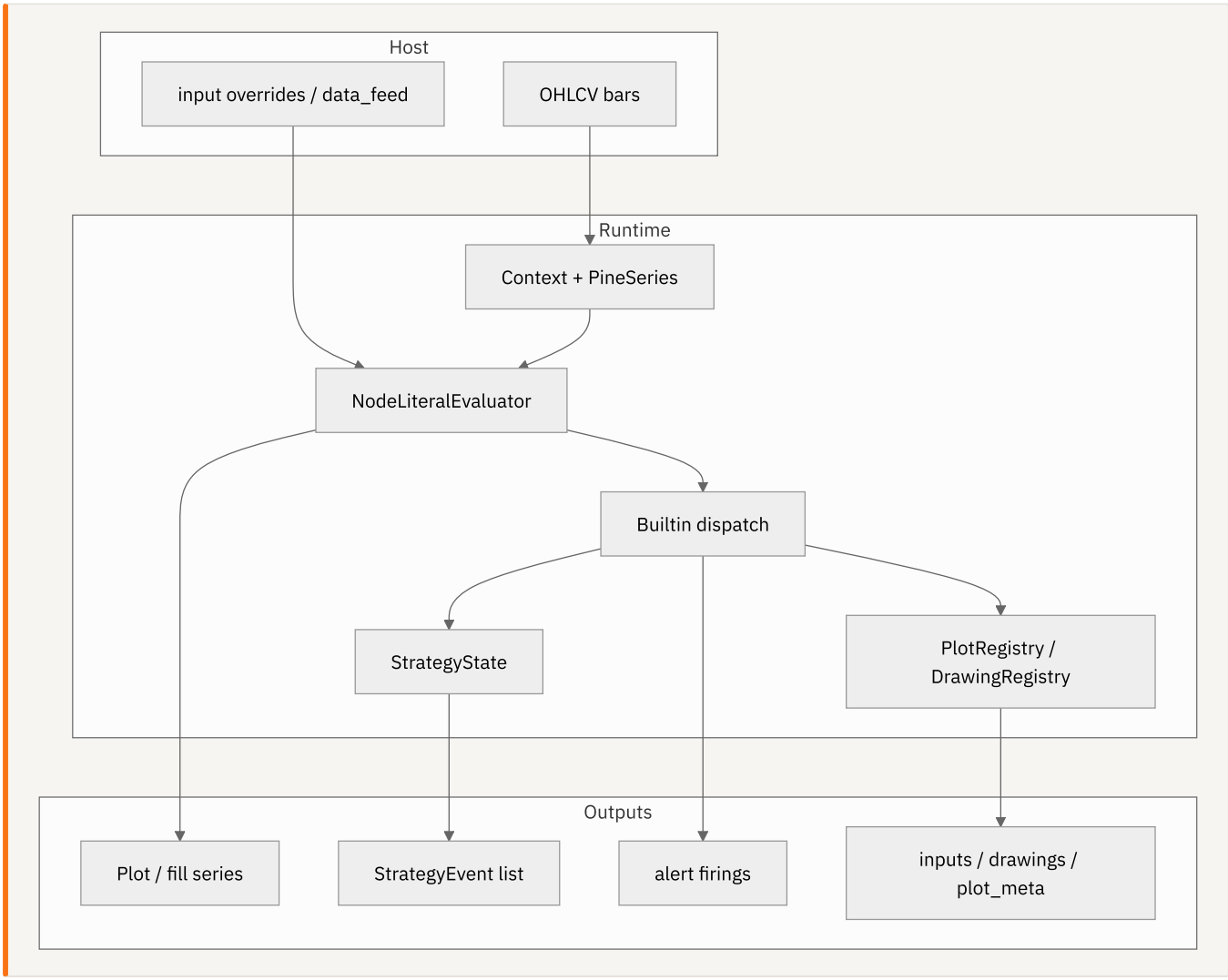
RUNTIME

Abstract

PYNE's runtime is the deterministic **bar-loop** that turns a parsed ASDL AST into series values, drawing side-effects, strategy events, and **alerts**. It is deliberately host-agnostic: the same visitor mixins power the library API, the Flask Pro API (`Runtime.run`), browser Pyodide, and edge workers. A second path—source-to-source compilation into a Numba or object-mode bar loop—trades interpretive flexibility for throughput on long histories; Pro API defaults to **`mode=auto`**.

This section documents **semantics**, not the chart. AXIS consumes plots / `fill` / drawings; hosts export alerts (optional L2 webhooks); HOOX consumes strategy events. None of those hosts is required to evaluate a script.

Conceptual model



Execution modes share the same parse tree (Pro API / `Runtime` also expose `auto`):

Mode	Entry	Bar loop	Typical use
Interpret	`NodeLiteralEvaluator.visit(tree)` per bar	Host updates `PineSeries` / context, re-visits AST	Full language surface, strategy, drawings, alerts, `request.*`
Compile	`pynescript.compiler.engine.compile_script`	Generated `for __bar_idx in range(n)` over OHLCV + **`time_arr`**	Numeric indicators (Numba) or object-mode subset
Auto	`Runtime.run(..., mode="auto")`	Compile when safe; interpret on fallback	Pro API default — prefer throughput without hard-failing exotic scripts

Compile inputs are contiguous arrays: `open_arr` … `vol_arr` plus **`time_arr`** (bar-open Unix ms; synthetic `bar_index * 60_000` when the host omits times) so `time` / calendar builtins match the interpret `time` series. Disk IR cache + product warm-compile (prewarm API/CLI) cut cold JIT latency. See Compiler overview and Interpret ↔ compile parity.

`request.security` (landed): interpret resamples same-symbol simple OHLCV and allowlisted `ta.sma` / `ema` / `rsi` / `atr` on a **coarser** TF (last **completed** HTF bucket only). Compile passthrough is same-symbol simple OHLCV only. Foreign symbols and complex / nested expressions resolve to **`na`** on both hosts (no inventing chart close as foreign fundamentals). Policy tags land on `result["meta"]["request_security"]`. Real multi-symbol feeds remain adapter work (roadmap **B1**).

Series caps / incremental TA: host list trim defaults **on** (`PYNE_SERIES_CAP`); bar-mode TA hot path defaults **on** (`PYNE_TA_INCREMENTAL`). Drawing GC honors `max_*_count` on the interpret registry; `fill` exports series keys + `plot_meta` for AXIS bands.

Interpret hot path (0.3.10 / Round 9): the AST walker inlines Assign / `Expr(Call)` after the first bar (function/type/import decls stay locked), skips unused derived OHLCV series (`hl2` / `hlc3` / `ohlc4` / `tr` unless named, `input.source`, or `ta.vwap`), and reuses plot cells by call-site. Same-machine bench @ 2000 bars vs 0.3.9: minimal **2.7×**, `ta.sma` **2.0×**, `ta_combo` **1.45×**. `PYNE_SERIES_RING` stays **off**.

Host flags and knobs

Knob	Default	Effect
`PYNE_SERIES_CAP`	**on**	Trim chronological `current_series` lists (`max_bars_back` / 256)
`PYNE_TA_INCREMENTAL`	**on**	Call-site incremental `ta.*` in bar mode
`PYNE_SERIES_RING`	**off**	Chronological ring lookback; skip dual list write when on
`PYNE_LIGHT_PLOTS`	**off**	Skip plot columns + input meta (corpus OK/fail only)
`PYNE_RUNTIME_MODE`	`interpret`	Default when `Runtime.run(mode=)` is omitted (`auto` is the Pro API body default)
`timeout_seconds=`	`None`	Interpret wall-clock budget; checked every 32 bars → `timed_out` + `error_kind=runtime`
`libraries=`	`[]`	`{namespace, name, version, source}` registered before `import` (auto forwards into interpret fallback)
`realtime_*`	historical	Interpret-only `varip` / forming-bar simulation (`realtime_last_bar` / `ticks` / `bars` / `from_bar`)
`Runtime.create_session`	interpret	Persistent `InterpretSession` (0.6.1): delta `append_bars` , forming-bar `update_last_bar` ; compile has no resumable state

Interface surface

Concern	Start here
Series history, `[]` , `var` / `varip` , `PYNE_SERIES_CAP` , `na`	Series & history
Expressions, statements, control flow	Expressions & statements
Builtin namespaces (`ta.*` , `strategy.*` , ...)	Builtins hub
Drawing / `plot` / `fill` / `max_*_count` GC	Drawing & plotting
`request.*` / `input.*` (foreign → `na`)	request / input
Library `export` / `import`	Libraries
`StrategyEvent` parity contract	Events
`alert()` / webhooks	Alerts
Numba / object-mode compiler	Compiler overview
Interpret vs compile plot parity	Parity testing
Incremental interpret sessions (`create_session`)	Evaluate scripts

Library callers typically use `NodeLiteralEvaluator` or the package **`pynescript.runtime.Runtime`** (Pro API re-export) rather than wiring the bar loop by hand:

```
from pynescript.ast.evaluator import NodeLiteralEvaluator
from pynescript.runtime import Runtime  # package SoT (0.3.4+)

ev = NodeLiteralEvaluator(context={"close": [1.0, 2.0, 3.0], "bar_index": 2})
result = ev.evaluate_script('//@version=6\nindicator("x")\nplot(close)')

# Full bar loop (preferred host API):
# Runtime(symbol="CHART").run(
#     source, ohlcv_data=bars, mode="auto",
#     timeout_seconds=None, libraries=None,
# )
# Incremental interpret (0.6.1):
# sess = Runtime(symbol="CHART").create_session(source, bars)
# sess.append_bars(new_closed_bars)
# sess.update_last_bar(forming_bar)
```

Hosts that own OHLCV (Pro API, workers) use the package host: push bar → fill pending orders → `visit(tree)` → drain events → collect plots. **SoT:** `src/pynescript/runtime/` (`host.py`, `series.py`, `evaluator.py`). `backend.runtime` / `backend.series` / `backend.evaluator` are identity re-export shims for the Pro API.

Internals

Path	Role
`src/pynescript/runtime/`	Package Runtime SoT — host, series, CustomEvaluator
`src/pynescript/ast/evaluator/`	Mixin evaluators (base, names, expressions, statements, libraries, events)
`src/pynescript/ast/evaluator/builtins/`	Builtin dispatch tables
`src/pynescript/compiler/`	`CompilerVisitor`, Numba builtins, strategy broker, engine façade
`backend/runtime.py`	Compat re-export (`sys.modules` alias) of `pynescript.runtime.host`
`tests/test_evaluator.py`, `tests/test_parity.py`, `tests/test_strategy_*.py`, `tests/test_first_party_ta_goldens.py`	Semantic oracles + dual-host TA goldens

Composition of the full interpreter:

```
BaseEvaluator
  + LiteralEvaluator
  + NameEvaluator
  + ExpressionEvaluator
  + StatementEvaluator
  + BuiltinEvaluator
  → NodeLiteralEvaluator
```

`BuiltinEvaluator` itself aggregates mixins (`TechnicalAnalysisMixin`, `StrategyBuiltinsMixin`, `PlottingFunctionsMixin`, ...) into one dispatch map built at construction time.

Invariants & edge cases

1. **Bar re-entrancy.** The AST is visited once per bar. Function/type definitions lock after bar 0 (`_pine_defs_locked`) so multi-dispatch tables do not grow (O(\text{'{'}bars{'}'}^2)).
2. **Order of broker steps.** Pending limit/stop fills run **before** script body evaluation on each bar (interpreter and compile object-mode agree). Pending-fill averaging when pyramiding ≤ 0 matches both brokers (**F2**).
3. **`na` is `None`.** Unresolved missing data, OOB history, and arithmetic with missing operands produce Python `None`, not IEEE NaN—except on the Numba path, which uses `np.nan` as the array sentinel. Cross-mode plot compares must normalize (`scripts/compare_interp_compile.py`).
4. **Series caps.** Unbounded `current_series` growth is capped (`PYNE_SERIES_CAP` default ON, `max_bars_back` / `_SERIES_MAX`) so long histories stay memory-bounded (**T1**).
5. **Incremental TA.** Hot-path kernels (MAs, oscillators, `bb` / `kama` / `cmo` / `stochrsi`, volume `obv` / `wad` / `cmf` / `klinger` / `nvi` / `pvi`, …) use call-site incremental updates in bar mode (`PYNE_TA_INCREMENTAL=0` to disable).
6. **Strategy state is per-evaluator.** `StrategyState` lives on the instance (`evaluator._strategy_state`), never as a process-global singleton—required for concurrent runs and parity tests.
7. **Drawing/plot registries are side channels.** Visual objects do not participate in pure expression values except where `plot()` returns a plot id for `fill()`. Hosts export `fill` bands for AXIS via series + `plot_meta` (and `__drawings` on compile). Drawing objects honor `max_*_count` GC (package + Pro API + AXIS Pyodide).
8. **Alerts are a host side-channel.** `alert()` / `alertcondition()` firings are not strategy events; dual-host export + L2 webhooks are documented under Alerts.
9. **Host series are not aliased on assign.** `last = time` (or `open` / `high` /…) copies the current scalar into a fresh series for `last`—see Series & history.
10. **Omitted bid/ask stay `na`.** Host `bid` / `ask` update only when those keys appear on a bar dict; they are not synthesized from close.

Worked example — minimal bar loop mental model

```
for bar_index, bar in ohlcv:
    open/high/low/close.update(bar)
    context[bar_index, time, barstate, …] = …
    process_pending_orders(OHLC)   # broker sim
    evaluator.visit(ast)           # script body
    events += drain_events()
    series.append(plot values)
```

Compile mode collapses this into one generated function over full arrays (`open_arr` … `vol_arr`, `time_arr`), with `__bar_idx` standing in for `bar_index`.

Failure modes

Symptom	Likely cause
`unexpected type of node: …`	AST node not implemented in any mixin's `visit_*`
`Unknown built-in function: …`	Missing dispatch key or wrong qualified-name resolution
Divergent strategy events vs TV	Partial-fill / OCA / risk settings, or fill timing vs bar
Numba path raises / falls back	Script uses UDT/map/drawing → object mode; or Numba not installed
Plots empty but no error	Host forgot to collect `plot_outputs` / `PlotRegistry` after visit; or `PYNE_LIGHT_PLOTS=1`
`timed_out` + partial series	`timeout_seconds` elapsed (checked every 32 interpret bars)

See also

- Language core — grammar, AST, types before evaluation
- Alerts — `alert()` engine + L2 webhooks
- Pro API runtime bridge
- Interpret ↔ compile parity — `scripts/compare_interp_compile.py`
- Numerical validation
- Roadmap — H1/H2/T1/T2/F1/F2/L2/P1p official builtins landed; optional set0x unmeasured
- pyne-worker
- AXIS docs — optional AXIS for series/drawings

SERIES AND HISTORY

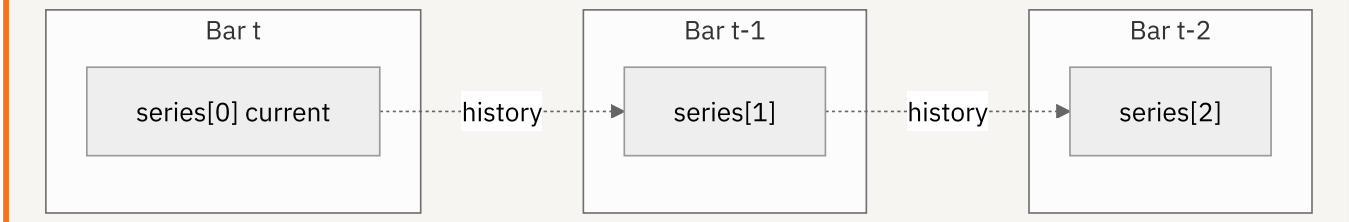
Pine series model: history operator, na propagation, var/varip persistence, and bar-mode scalars.

SERIES AND HISTORY

Abstract

Pine's core data structure is the **series**: a value that evolves bar by bar, with random access to past values via the history operator `[]`. PYNE implements that model with host-managed series wrappers (`PineSeries` in the Pro API runtime), plain Python lists in unit tests, and flat `numpy` arrays on the compile path. Understanding history indexing and `na` is prerequisite to every builtin and strategy claim.

Conceptual model



Pine indices are **most-recent-first**:

Pine	Meaning	List representation (chronological)
`close[0]`	Current bar	`list[-1]`
`close[1]`	Previous bar	`list[-2]`
`close[n]` with (n \ge \text{'{'}len{'}'})	Out of range	`na` (`None`)

Negative offsets are **not** Python wraparound; they **soft-fail** to `na` (`None`).

Interface surface

History operator (`visit_Subscript`)

Implemented in `NameEvaluator.visit_Subscript`:

- Float indices coerce to `int` (e.g. `depth / 2`); NaN index → `na`.
- Lists map Pine index (i) to Python index `-(i + 1)`.
- Scalars: `x[0]` is `x`; `x[i]` for (i > 0) is `na` (no history buffer).
- Matrices use `[row, col]` (tuple/list of length 2), not series semantics.

Series wrappers

Hosts expose OHLCV as objects with:

- `.current` — scalar for the active bar

- `.history` — most-recent-first buffer (deque or list)

Arithmetic in `ExpressionEvaluator` coerces such wrappers via `_as_scalar_operand` so `close + 1` does not attempt object addition.

Assigning host series (copy, not alias)

Binding a **host** multi-bar series into a **new** name copies the current scalar into a fresh series for that name—it does **not** alias the underlying `open` / `high` / `low` / `close` / `volume` / `time` buffers:

```
last = time          // copy of current time into series `last`
last := last + 1     // mutates only `last`, never the host `time` buffer
```

Implemented in `StatementEvaluator._bind_series_name`: if the RHS has `.current` / `.history` / `.update` (a host or tracked series), the RHS is reduced to its **current scalar** before allocating or updating the LHS series. Same-bar reassignment (`x = 0` then `x := expr`) overwrites the current sample so `x[1]` remains the previous bar's final value. This prevents scripts such as dividend TTM time tracking from corrupting `time[j]` history when they rebind through a local.

UDF series parameters (0.6.2)

A Pine function parameter that is subscripted (`s[1]` inside `f(s) => …`) is a **series of that argument**, not a scalar that happens to be named `s`. Interpret now allocates a per-call-site history wrapper when the caller passed a scalar or a host-series current (`src = close; updown(src)`). Compile already tracked UDF series state; the two hosts agree on official TradingView® builtins (P1p **0 MISMATCH**). Nested calls get distinct streams. See Expressions & statements.

`var` / `varip` / `const`

Qualifier	Semantics in PYNE
`var` / `varip`	Initializer runs on first **execution** of that declaration site (tracked in `_var_declarations`), not only `bar_index == 0`. Later bars skip re-init so the value carries. History-tracked `var` series get a **start-of-bar carry** (`_commit_unwritten_history`) so `x[1]` is last bar's persist even when no `:=` ran this bar. On realtime ticks, `varip` re-evaluates the RHS.
`const` (v6)	Always initializes when the statement runs; not a cross-bar carry lock like `var`.
bare assign	Re-evaluates every bar; series "history" is the host series or prior list values.

Nested `var` inside `if barstate.islast` or a function therefore initializes on first path-taken bar—matching Pine's execution-based persistence.

`na` propagation

Operation	Rule
Binary arithmetic / comparison	Any `None` operand → `None` (or element-wise for lists)
Division by zero	→ `None` (not exception)
Soft type errors (`"a" + 1`)	→ `None`
Unary	`None` stays `None`
History OOB	→ `None`
Bare name `na`	Builtin / sentinel resolving to `None`
`nz(x, r)` (Numba path)	`nan` → replacement

List-valued series apply operators element-wise, aligning on the **trailing edge** when lengths differ (pad leading `None`).

`ta.highestbars` / `ta.lowestbars` offsets

These return a **bars-back offset** (not the extreme price):

Result	Meaning
`0`	Extreme is on the current bar
`-1` , `-2` , ...	Extreme is 1, 2, ... bars ago (down to `-(length-1)`)
`-1` sentinel (warm-up / invalid / all-`na`)	Short history (`bar_index+1 < length`), bad length, or no finite samples in the window

On ties, the **oldest** extreme in the window wins (leftmost bar). Interpret and Numba kernels share this contract (`_highestbars` / `numba_highestbars`).

History subscript interaction: Pine history indices are non-negative (`series[n]` with (n \ge 0)). Negative indices soft-fail to `na` on interpret series wrappers so warm-up / auto-step- `-1` / `highestbars` misuse do not abort the bar loop. On the compile path, dynamic offsets are coerced float→int (NaN→0) and the computed array index is clamped to `[0, n)` so expressions like `high[-ta.highestbars(...)]` (future-looking form) soft-fail to `nan` near series end rather than raising.

Bar mode vs full-series mode

Technical helpers (`technical_submodules/core.py`) distinguish:

- **Full-series mode** (unit tests with explicit lists): `ta.sma` may return a full list of values.
- **Bar mode** (`_pine_bar_mode`): returns the **current scalar** so expressions like `ta.ema(close,12) - ta.ema(close,26)` stay numeric per bar.

History buffers for indicators are truncated to a rolling window (`_SERIES_MAX = 256` for wrapper histories) to avoid (O(n^2)) full-history recomputation every bar.

Host series caps (`PYNE_SERIES_CAP`)

Pro API / `pynescript.runtime.Runtime` optionally **trims** chronological `current_series` lists and related host history so long charts do not grow (O(\text{'{'}bars{'}'})) memory per series.

Knob	Default	Meaning
`PYNE_SERIES_CAP`	**on**	Enable trimming. Disable with `0` / `false` / `no` / `off` (oracle / debug only).
Cap size	`256` (`DEFAULT_SERIES_MAX`)	Raised when the script declares a larger `max_bars_back=…`.
`PYNE_SERIES_MAX`	unset	Absolute override of the cap (positive int).
PineSeries history floor	`1000`	Separate from list caps; raised by `max_bars_back` / `PYNE_SERIES_MAX`.

Correctness notes (see `src/pynescript/runtime/series.py`; `backend/series.py` is a re-export shim):

- Window kernels (`ta.sma`, `highest`, …) need `length ≤ cap`.
- Recursive smoothers under **incremental** TA (default `PYNE_TA_INCREMENTAL` on) carry state and stay safe independent of list length once warm.
- With **full recompute** (`PYNE_TA_INCREMENTAL=0`) and `bars ≫ cap`, EMA/RMA-style paths can diverge from a full-history oracle—prefer incremental (default) or raise the cap / disable series cap for goldens.

Out-of-range history offsets still return `na` (`None`); never `0`.

Unused derived series (0.3.10)

The interpret host updates `open` / `high` / `low` / `close` / `volume` / `time` every bar. Derived series (`hl2`, `hlc3`, `ohlc4`, `tr`, `time_close`) are written only when the source names them, uses `input.source`, or calls **`ta.vwap`** / **`vwap`** (default source is `hlc3` even when that identifier is absent). `ta.ao` rebuilds `hl2` from high/low if the derived list is empty.

`PYNE_SERIES_RING` (chronological tail view) remains **default off**. When on, the host skips the second chronological list write.

Internals

Path	Role
`src/pynescript/ast/evaluator/names.py`	`visit_Name`, `visit_Attribute`, `visit_Subscript`
`src/pynescript/ast/evaluator/expressions.py`	NA-safe binary/unary ops, series coerce
`src/pynescript/ast/evaluator/statements.py`	`var` / `varip` / `const` assign
`src/pynescript/ast/evaluator/builtins/technical_submodules/core.py`	`_as_series`, bar mode finalize
`src/pynescript/ast/evaluator/builtins/utility.py`	`max_bars_back`, `last_bar_index`, `na` helpers
`src/pynescript/runtime/host.py`	Bar loop; derived-series skip; series-cap trim (`backend.runtime` re-exports)
`src/pynescript/runtime/series.py`	`PineSeries`, `resolve_series_cap`, `PYNE_SERIES_CAP` / `PYNE_SERIES_MAX` / `PYNE_SERIES_RING`

Compile path: series are `np.full(n_bars, np.nan)` written at `__bar_idx`; history is `arr[__bar_idx - n]` with OOB → `np.nan` (see Compiler overview). Host **`time_arr`** is a chronological `float64` vector of bar-open Unix ms (same length as OHLCV); bare `time` lowers to `time_arr[__bar_idx]`, and `time[n]` uses the same history indexing rules as `close[n]`. When the host omits bar times, `Runtime` synthesizes `i * 60_000` so length always matches—calendar / `time` scripts stay defined on both modes.

Invariants & edge cases

1. **Chronology of `.history`.** Most-recent-first; reverse when converting to chronological lists for `ta.*` / `array.*`. Compile arrays are chronological (index 0 = first bar).
2. **No negative Pine indices.** Soft-fail to `na` / `nan` (not Python wraparound, and **not** a raise)—different from Python lists. See highestbars note above.
3. **`None` vs `float('nan')`.** Interpreter prefers `None`; Numba kernels use `np.nan`. Cross-mode comparisons must normalize.
4. **`max_bars_back`.** Raises the host series cap when larger than the default 256; still not a TV-style hard chart depth, but it affects trim policy.
5. **`PYNE_SERIES_CAP` default on.** Disable only for debugging / full-history oracles; pair with incremental TA for production long histories.
6. **Derived skip.** `ta.vwap()` still requires a live `hlc3` series; the host treats `vwap` as a derived-series consumer.
7. **Tuple returns from multi-value `ta.*`.** Unpacking `[a, b, c] = ta.macd(...)` uses `current` when it is a sequence of matching arity; otherwise history heuristics apply.
8. **Host series assign is by value.** New names never share the host OHLCV/`time` buffer reference.

Worked examples

History lag

```
//@version=6
indicator("lag")
delta = close - close[1]
plot(delta)
```

On bar 0, `close[1]` is `na` → `delta` is `na`. On bar 1+, arithmetic is ordinary floats (or series lists in batch tests).

`var` counter

```
//@version=6
indicator("count")
var int n = 0
n := n + 1
plot(n)
```

`n` initializes once; subsequent bars reassignment (`:=`) increments. Declaration sites are recorded in `_var_declarations`.

Element-wise NA

Given two list series of different lengths, `a + b` aligns tails and pads the longer prefix with `None`—preserving "most recent bars line up" intuition.

Failure modes

Symptom	Cause
All `na` after first bar	Host not updating series / `bar_index` between visits
`var` re-inits every bar	Fresh evaluator or `reset_var_declarations` each bar incorrectly
TypeError on `close + 1`	Series wrapper missing `.current` / not coerced
Compile vs interpret plot mismatch	`nan` vs `None`, or seed differences in EMA/RSI kernels — see parity
`time[j]` wrong after rebinding `last = time`	Expected only if host series were aliased; verify copy-on-assign path
`highestbars` always `-1`	Warm-up (`bar_index+1 < length`) or all-`na` window
Long-chart OOM / growing RSS	Series cap disabled or `PYNE_SERIES_MAX` huge; leave `PYNE_SERIES_CAP` on
EMA drift after many bars with cap	Full recompute + short cap — enable incremental TA or raise cap

See also

- Expressions & statements
- Technical builtins
- Numba path

- Interpret ↔ compile parity
- Runtime hub

EXPRESSIONS AND STATEMENTS

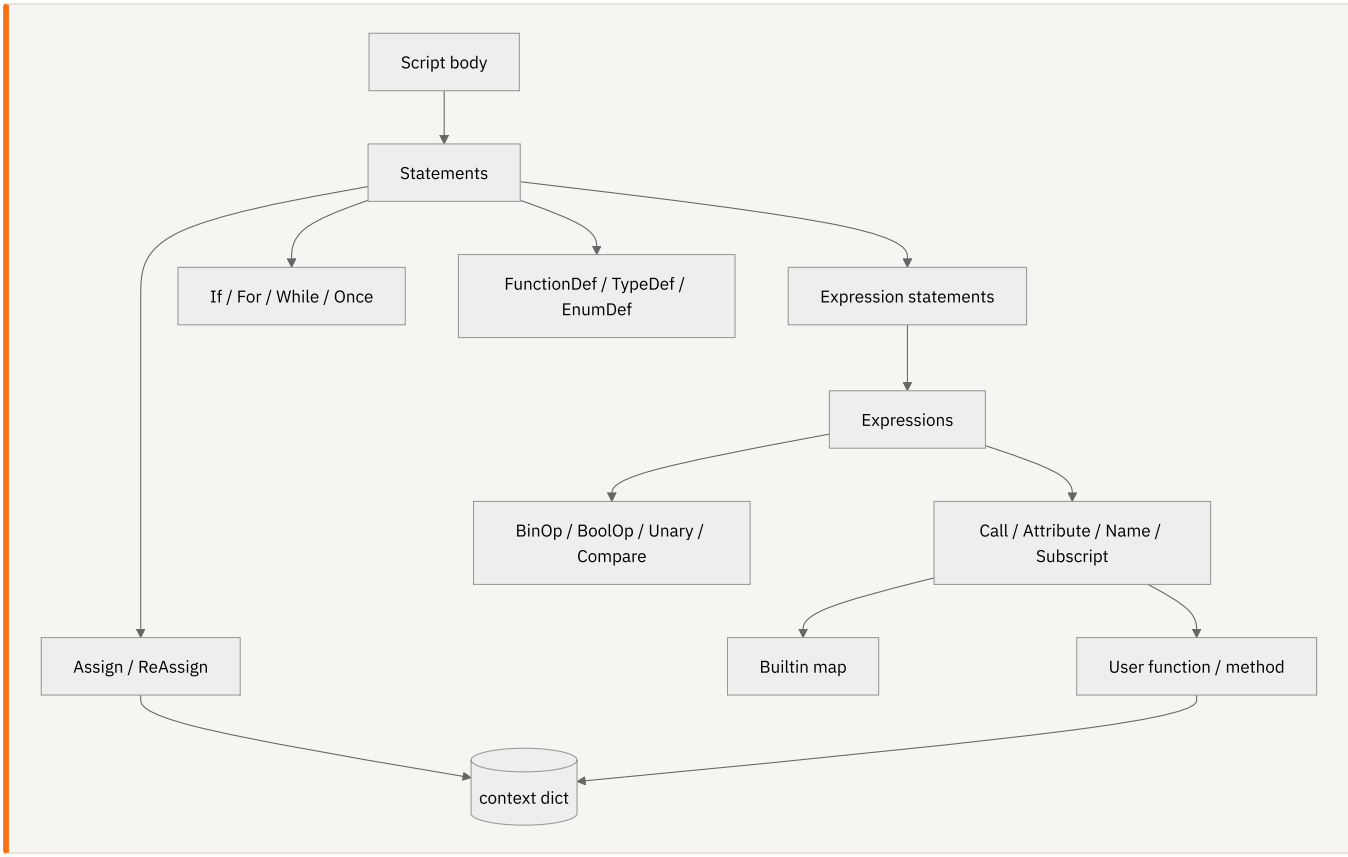
How the AST walker evaluates operations, calls, control flow, assignments, and user definitions.

EXPRESSIONS AND STATEMENTS

Abstract

Evaluation is pure visitor dispatch: each ASDL node type maps to a `visit_*` method on a mixin. Expressions produce values (with Pine `na` and series rules); statements mutate context, register types/functions, or emit side effects via builtins. This page is the operational semantics of the interpreter path—what runs inside `evaluator.visit(tree)` each bar.

Conceptual model



Interface surface

Expressions (`ExpressionEvaluator`)

Node	Behavior
`BoolOp` (`and` / `or`)	Manual short-circuit: `and` stops at first falsy (`na` / `0` / `False`); `or` stops at first truthy. Returns a Python `bool` .
`BinOp`	NA-safe `+ - * / %` ; division by zero → `na`
`UnaryOp`	`not` , unary `+` / `-` with NA and list map
`Compare`	Chained comparisons with NA-safe operators
`IfExp`	Ternary: condition, then, else
`Call`	Builtin dispatch, user functions, methods, multi-dispatch overloads
`Tuple` / lists	Literal sequences for unpacking and multi-arg returns

Binary ops coerce `PineSeries` -like objects to `.current` before operating. List operands broadcast or zip with trailing alignment (see Series & history).

Names (`NameEvaluator`)

Node	Behavior
`Name`	Context lookup; bare series builtins (`na` , `year` , ...); else string id (lazy)
`Attribute`	Qualified context keys, builtins (`strategy.position_size`), library modules, UDT fields/methods, array/drawing/matrix method markers, Python getattr fallback
`Subscript`	Series history / array / matrix indexing

Critical: qualified names like `strategy.opentrades.entry_price` are resolved via AST path construction (`ast_qualified_name`) **without** evaluating intermediate zero-arg series that would collapse the path to an int.

Statements (`StatementEvaluator`)

Construct	Behavior
`Script`	Walk body; finalize library exports if `library(...)` active
`Assign`	`var` / `varip` once-per-declaration; `const` ; ordinary assign; tuple unpack; `export` registration. History-tracked `var` series get a start-of-bar carry so `x[1]` is the prior persist.
`ReAssign` (`:=`)	Update existing binding (and UDT fields)
`If` / `For` / `While`	Standard control flow; loop variables in context
`Once`	Fire body the first time `test` is true (omit `test` → `true`) on a **confirmed** bar, then skip. Unconfirmed realtime ticks may re-run until `barstate.isconfirmed` commits the fired flag. Does not return a value.
`FunctionDef`	Store callable in context; multi-dispatch overload lists; `export` ; methods tagged `__pine_method__` . After bar 0 the host walks a **hot body** that skips Function/Type/Enum/Import (already bound).
`TypeDef` / `EnumDef`	Type registry / enum member dicts
`Import`	Resolve `LibraryRegistry` → bind alias to `LibraryModule`
Declarations	`indicator` / `strategy` / `library` via declaration builtins

Function invocation binds parameters into context (with restore of prior bindings on exit) so nested calls and bar-local state interact cleanly. After bar 0, hosts set `_pine_defs_locked` so re-visiting the script does not re-register overload tables. The locked walk inlines `Assign` and `Expr(Call)` (no generic `visit` frame) and UDF/call-expression history keys use `_pine_site_id` stamped on the AST.

Calls and multi-dispatch

Method markers returned from attribute evaluation:

Marker	Meaning
`("_method_call", instance, name)`	UDT method
`("_array_method", list, name)`	`array.<name>(self, …)`
`("_ns_method", obj, "label.get_text")`	Drawing/matrix namespace method
`("_ext_method", receiver, name)`	Free `method` with receiver as first arg

Overloads with typed first parameters (e.g. `matrix.float` vs `array.label`) select via coarse type tags; `na` receivers deliberately exclude matrix/array/drawing tags so optional UDT fields do not mis-dispatch to `tostring` on collections.

Internals

Path	Role
`src/pynescript/ast/evaluator/expressions.py`	Ops, calls, conditionals
`src/pynescript/ast/evaluator/names.py`	Names, attributes, subscripts
`src/pynescript/ast/evaluator/statements.py`	Script structure, assign, defs, import
`src/pynescript/ast/evaluator/literals.py`	Constants, colors, strings
`src/pynescript/ast/evaluator/base.py`	Context, math constants, library registry hook
`src/pynescript/ast/evaluator/types.py`	`EvaluatorProtocol` typing for mixins

Invariants & edge cases

1. **Context is the single store.** Builtins read OHLCV and strategy series from `self.context`; hosts must keep it coherent each bar.
2. **String fallback names.** An unbound `Name` returns its id string so later attribute/call resolution can still hit the builtin map (`"ta.sma"` patterns via attribute chains).
3. **Parameter unbind.** Missing-before-bind params use a sentinel so unbind `pop`s rather than leaving `None` ghosts.
4. **`na` normalization.** String `"na"` / `"nan"` / `"none"` may coerce to `None` at dispatch boundaries for UDT optional args.
5. **User functions re-enter the visitor.** Recursive and higher-order patterns are limited by Python stack, not a Pine-specific trampoline.

Worked examples

Ternary and comparison chain

```
//@version=6
indicator("cmp")
v = close > open ? close - open : open - close
plot(v)
```

Each operator path is NA-safe: if `close` or `open` were missing, comparisons and arithmetic yield `na`.

User function with series

```
//@version=6
indicator("fn")
f(x, n) =>
    ta.sma(x, n)
plot(f(close, 14))
```

`f` is stored as a Python callable that rebinds `x` / `n` and visits the function body AST each call.

UDF series parameters (0.6.2): when a caller passes a scalar (or a host-series current) into a function that then reads `s[1]`, interpret wraps that parameter as a per-call-site history stream. `src = close; updown(src)` therefore matches compile — the previous-bar sample is the previous argument at that call site, not an empty scalar. Official TradingView® builtin scripts are 0 MISMATCH vs compile on this contract.

`once` (August 2026)

```
//@version=6
indicator("once")
var int n = 0
once close > ta.sma(close, 20)
    n := 1
plot(n)
```

Equivalent to a `var bool` latch plus `if`, but cheaper: the structure is not re-evaluated after it fires on a closed bar. On a forming realtime bar (`Runtime.run(..., realtime_last_bar=True, realtime_ticks=N)`), the latch is not committed until `barstate.isconfirmed`. Compile is historical (`isconfirmed` always true).

Intrabar rollback on realtime ticks (0.6.0)

On a forming realtime bar, intermediate ticks snapshot the `var` scope (context bindings, series currents, declaration set) and restore it after the visit — every tick starts from confirmed state and only the final tick commits. `varip` names persist across ticks by design. Shallow by design: history pushes, the series-assign map, in-place container mutations, and broker side effects are not rolled back.

Method multi-dispatch sketch

```
method log(this, string s) => ...
method log(this, float x) => ...
```

Both register under the same name with `__pine_overloads__`; call sites pick by receiver/arg type tags.

Failure modes

Error	Meaning
`unexpected type of node`	Missing `visit_*` for an AST construct
`Unsupported binary operator`	Op class not in the dispatch table
History `series[-1]` is `na`	Negative Pine offsets **soft-fail** to `None` (not Python wraparound, not a raise)
Wrong overload / destroyed receiver	Historical `na` string path; now normalized—report residual mismatches as bugs
Stack overflow	Deep recursion in user functions

See also

- Series & history
- Builtins hub
- Libraries
- Type system (core)

BUILTINS

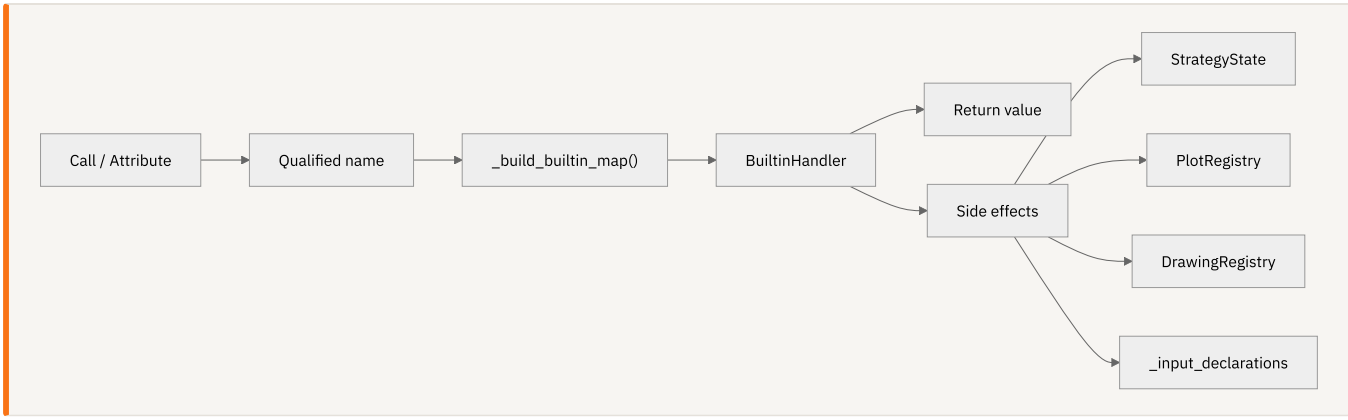
Dispatch architecture for ta.*, strategy.*, collections, plotting, request.*, and other standard namespaces.

BUILTINS

Abstract

Pine's standard library is not a single module: it is a **flat qualified-name dispatch table** (`"ta.sma"` , `"strategy.entry"` , `"array.push"` , ...) assembled from category mixins. Each entry is a handler `(args, kwargs?) → value` with side effects when the language requires them (orders, plots, drawings, inputs). This hub orients the surface; child pages go deep on technical, strategy, collections, drawing/plotting, and request/input.

Conceptual model



`BuiltinEvaluator` merges mixin maps in a fixed order and then registers ticker, logging, color, timeframe, and script-declaration functions. The `strategy()` declaration handler is wrapped so broker settings apply to `StrategyState` .

Interface surface

Namespace / area	Mixin / module	Docs
`ta.*`	`TechnicalAnalysisMixin` + `technical_submodules/*`	Technical
`strategy.*`	`StrategyBuiltinsMixin`, `StrategyConstantsMixin`	Strategy
`array.*` / `matrix.*` / `map.*`	`ArrayBuiltinsMixin`, matrix/map evaluators	Collections
`plot*` / `hline` / `line` / `label` / …	`PlottingFunctionsMixin`, `DrawingBuiltinsMixin`	Drawing & plotting
`request.*` / `input.*`	`RequestBuiltinsMixin`, `InputBuiltinsMixin`	Request & input
`math.*`, min/max, `nz`, …	`NumericBuiltinsMixin`	(this page)
`str.*`	`StringBuiltinsMixin`	(this page)
`color.*`	`register_color_functions`	(this page)
`syminfo` / `ticker.*`	`register_ticker_functions`	(this page)
`timeframe.*`	`register_timeframe_functions`	(this page)
`alert` / logging	`AlertsMixin`, `register_logging_functions`	Alerts
`indicator` / `strategy` / `library`	`register_script_declaration_functions`	Libraries
Footprint / volume_row	`FootprintBuiltinsMixin`	Request & input

Numeric and string essentials

- **Numeric:** `math.*` constants from `BaseEvaluator` (`math.pi`, `math.e`, golden-ratio helpers); functions cover abs, min/max, rounding, logs, trig, and NA helpers (`nz`, `na` predicates via utility).
- **String:** `str.*` formatting, conversion (`str.tostring` respects `format.*` constants), and manipulation aligned with common Pine scripts in the builtin corpus.

Color, ticker, timeframe

- **Colors:** hex constants (`color.red` → `#F23645`, etc.) plus composition helpers.
- **syminfo / ticker:** host-provided `Syminfo` object or flat context keys; ticker construction for `request.security` symbols.
- **timeframe:** period flags (`isintraday`, `isdaily`, …) default daily in base constants; hosts override from bar spacing.

Alerts

`alert(message, freq)` and `alertcondition(condition, title, message)` record structured firings (`message`, `freq`, `bar_index`, `time`, `source`) with TV-style frequency rules (`once_per_bar`, `once_per_bar_close`, `all`). Hosts export them on `/run` and can POST last-bar batches to webhooks (**L2**). Side channel only — **not** strategy events.

Full detail: Alerts.

Internals

Path	Role
`src/pynescript/ast/evaluator/builtins/__init__.py`	`BuiltinEvaluator` composition
`src/pynescript/ast/evaluator/builtins/base.py`	Dispatch mixin, `_call_builtin`, registration
`src/pynescript/ast/evaluator/builtins/*.py`	Category implementations
`scripts/generate_builtin_metadata.py`	LSP/metadata surface (not hand-edited JSON)

Handlers are looked up by the fully qualified string built during name/attribute evaluation. Zero-arg series (e.g. `strategy.equity`) are registered as builtins that ignore empty args.

Invariants & edge cases

1. **One map, many mixins.** New builtins require a map entry *and* (usually) metadata regeneration for LSP.
2. **Kwargs and positionals.** Handlers accept both; Pine named arguments arrive as kwargs when the call AST carries them.
3. **Bar mode.** Stateful `ta.crossover` / similar use per-bar call indices reset by the host (`_cross_call_i`). Incremental `ta.*` (`PYNE_TA_INCREMENTAL`, default on) uses a sibling `_ta_call_i` slot.
4. **Mock fallbacks.** Bare equity-style `request.security` strings may still mock offline; foreign / fundamental prefixes and complex expressions return **`na`** (see request / input).
5. **Compile path subset.** Numba builtins reimplement only a fraction of this table; object mode covers more via Python helpers—see Compiler.

Worked example — resolution path

```
plot(ta.sma(close, 14))
```

1. Attribute/call chain builds qualified name `ta.sma`.
2. `_call_builtin("ta.sma", [close_series, 14])`.
3. Handler coerces series → list, computes SMA, finalizes scalar or list.
4. `plot` registers a `Plot` with that series value and returns the plot id.

Failure modes

Symptom	Fix
`Unknown built-in function`	Typo, version surface gap, or failed attribute chain
Always mock prices	Wire `data_feed` / `data_provider` into evaluator context
LSP knows a function runtime lacks	Metadata ahead of implementation—check missing features
Handler arity errors	Positional vs keyword mismatch in script

See also

- Technical

- Strategy
- Collections
- Drawing & plotting
- Request & input
- Alerts
- Builtin metadata (LSP)

TECHNICAL ANALYSIS (TA.*)

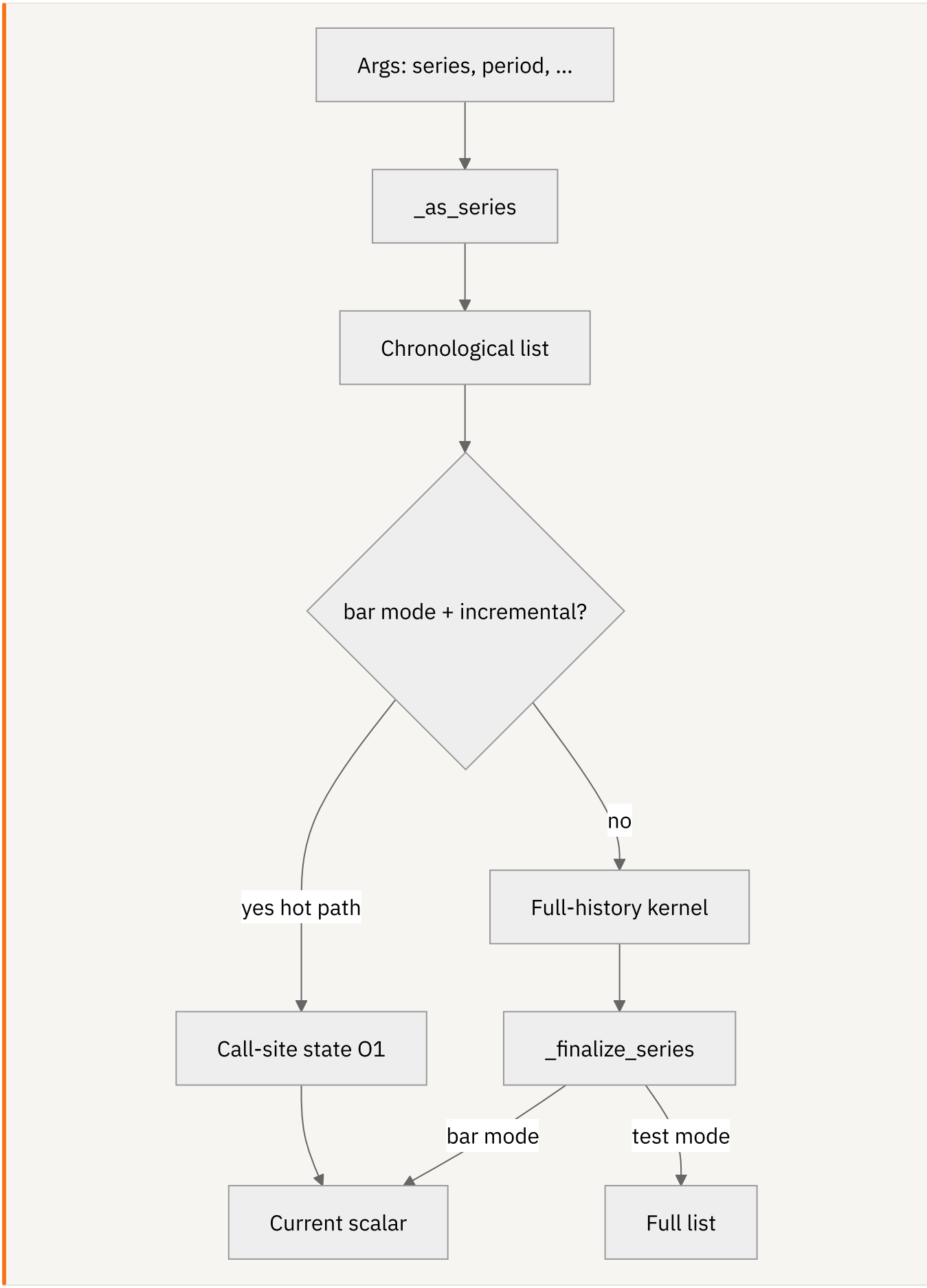
ta.* indicators: series helpers, bar mode, interpret↔compile parity, and submodule layout.

TECHNICAL ANALYSIS (`ta.*`)

Abstract

The `ta.*` namespace is the largest pure-compute surface in PYNE: moving averages, oscillators, volatility bands, volume studies, pattern helpers, and cross/rise/fall predicates. Implementations live under `technical_submodules/` and are composed into `TechnicalAnalysisMixin`. Numerical behavior is validated against TradingView reference series (see numerical validation); the design goal is IEEE-limit parity, not "approximate TA." Interpret and compile (Numba) hosts share the same formulas for the kernels listed under Interpret ↔ compile parity.

Conceptual model



Hosts in production set bar mode so indicator calls return **scalars for the active bar**, matching how Pine expressions compose (`ta.ema(a) - ta.ema(b)`).

Incremental hot path

When `_pine_bar_mode` and `_pine_ta_incremental` are enabled (Runtime default; disable with **`PYNE_TA_INCREMENTAL=0`**), hot `ta.*` builtins update **call-site state** once per bar instead of recomputing full history. Call sites are indexed like crossovers (`_ta_call_i` reset each bar). Nested forms such as `ta.ema(ta.sma(close, 14), 10)` stay correct because each call keeps its own slot. Golden suite: `tests/test_ta_incremental.py` (incremental last values ≡ full recompute).

Family	Incremental builtins (non-exhaustive)
Moving averages	`sma` , `ema` , `rma` , `wma` , `hma` , `vwma` , `swma` , `kama` , `dema` , `tema` , `alma`
Oscillators	`rsi` , `macd` , `stoch` / `stochrsi` , `cci` , `cmo` , `tsi` , `roc` , `wpr` , `aroon` , `dpo` , `kst`
Volatility / structure	`atr` , `tr` , `stdev` , `bb` , `highest` / `lowest` , `linreg` , `adx` / `dmi` , `supertrend` , `kc` , `sar` , `donchian`
Volume (**0.3.10–0.3.11**)	`obv` , `wad` / `wvad` , `cmf` , `klinger` , `mfi` , `vwap` , `accdist` , `nvi` , `pvi`
Other	`change` / `mom` / `cum` , `median` , `percentrank` , `rising` / `falling` , `barssince` , pivots

`ta.nvi` / `ta.pvi` are incremental (O(1)/bar; `PYNE_TA_INCREMENTAL=0` full-list). **`ta.atr` is Wilder RMA of true range** on interpret (full + incremental) and Numba (`ta.rma(ta.tr, length)` — 0.3.4+). Supertrend is locked mid±factor·ATR (not the TV band ratchet).

Interface surface

Dispatch keys (non-exhaustive; map is authoritative in `technical.py`):

Family	Examples
Moving averages	`ta.sma` , `ta.ema` , `ta.wma` , `ta.rma` , `ta.hma` , `ta.vwma` , `ta.swma` , `ta.alma`
Oscillators	`ta.rsi` , `ta.stoch` , `ta.cci` , `ta.cmo` , `ta.mfi` , `ta.roc` , `ta.wpr` , `ta.tsi` , `ta.rci`
Trend / channels	`ta.macd` , `ta.adx` , `ta.dmi` , `ta.supertrend` , `ta.sar` , `ta.linreg`
Volatility	`ta.atr` , `ta.tr` , `ta.bb` , `ta.bbw` , `ta.kc` , `ta.kcw` , `ta.stdev` , `ta.variance`
Volume	`ta.obv` , `ta.vwap` , `ta.mfi` (shared), volume submodule helpers
Structure	`ta.highest` / `lowest` / `highestbars` / `lowestbars` , `ta.pivothigh` / `pivotlow`
Predicates	`ta.crossover` , `ta.crossunder` , `ta.cross` , `ta.rising` , `ta.falling`
Stats / other	`ta.change` , `ta.mom` , `ta.cum` , `ta.median` , `ta.mode` , percentiles, `ta.barssince` , `ta.valuewhen` , `ta.correlation`

Argument conventions

- Typical form: `ta.fn(series, length)` .

- Some functions allow **period-only** calls (e.g. `ta.highest(20)`) defaulting source to `high` / context series via `_expect_series(..., allow_period_only=True)`.
- Multi-value returns (e.g. MACD, BB, Supertrend) unpack as tuples/lists; assignment uses StatementEvaluator unpack rules.
- Fractional lengths floor to int (TradingView-compatible).

Stateful crosses

`ta.crossover` / `ta.crossunder` need previous-bar pairs. In bar-mode runs the host resets a call-site index (`_cross_call_i`) each bar so multiple cross calls in one script keep independent state.

Internals

Path	Role
`src/pynescript/ast/evaluator/builtins/technical.py`	Dispatch map aggregation
`.../technical_submodules/core.py`	Series coerce, expect helpers, bar finalize, **incremental kernels**
`.../moving_averages.py`, `oscillators.py`, `volatility.py`, `volume.py`, …	Kernels + inc wiring
`.../common.py`, `basic.py`, `advanced.py`, `patterns.py`, `strategies.py`, `synthesizer.py`, `economics.py`	Additional families
`src/pynescript/compiler/numba_builtins.py`	Compile-path Numba kernels (must track interpret formulas)
`tests/test_ta_indicators_*.py`, `tests/test_indicators.py`	Regression
`tests/test_ta_incremental.py`	Inc ≡ full recompute golden (full CI gate)
`tests/test_first_party_ta_goldens.py`	Dual-host first-party goldens (ATR / Supertrend / Keltner)
`tests/test_interp_compile_parity.py`	Always-on dual-host smoke + full-corpus mark
`scripts/compare_interp_compile.py`	Corpus harness (report under `.cache/interp_compile_parity.json`)
`tests/test_compiler_numba.py` (`TestInterpCompilePlotParityFixes`, highestbars offsets, …)	Targeted formula locks
`docs/numerical_validation_report.md`	Published precision summary (TV / IEEE bounds)

History for wrapper series is reversed to chronological order and may be truncated (`_SERIES_MAX`) before full kernels run. Incremental path only needs `series[-1]` per call, so truncation does not freeze state.

Interpret ↔ compile parity

Numba kernels are **separate code** from the interpreter, but the dual-host contract is **same formula, same warm-up mask, same `na` rules** for the surface below. Drift is a bug, not an allowed “approx compile.” Prove regressions with the interp↔compile harness and the numerical validation page (TV / IEEE bounds).

Topic	Shared behavior
`ta.rsi`	**Wilder** on both hosts: SMA seed of the first `period` deltas, then RMA of gains/losses. First finite bar is index `period` (`period` deltas need `period+1` prices). Not a simple window average.
`ta.roc`	Standard TV formula `100 * (src - src[length]) / src[length]`. Warm-up is **`na`**, never an early `0.0`. Zero or missing baseline → `na`. Interpret no longer used a wrong lookback denominator or early zero.
`ta.wma`	Requires a **full non-`na` window** of length `N`. Nested forms such as `ta.wma(ta.roc(...), …)` stay `na` until the inner series has produced `N` finite samples (no partial-window reweight).
`ta.cum`	Running sum; Pine **`na` / IEEE NaN treated as 0** (skipped contribution), matching TradingView cumulative-sum semantics.
`ta.highestbars` / **`ta.lowestbars`**	Return **negative bars-back offsets**: `0` if the extreme is the current bar, `-1` one bar ago, …, `-(length-1)` at the far edge. Short history (`i + 1 < length`) returns `-1`. Matches Aroon-style scripts that index with `high[ta.highestbars(...)]` / `math.abs(ta.lowestbars(...))`.
`math.avg` **(multi-arg)**	Arithmetic mean of the arguments. **Any `na` argument → `na`** (do not skip). Not a rolling `ta.sma`.
`ta.linreg`	Least-squares fit over the window; **endpoint at `x = n−1` when `offset=0`** (TV): `mean_y + slope * ((n−1) − mean_x)` / compile `intercept + slope * (n − 1 − offset)`. Length < 2 → `na`.
`ta.mfi`	Warm-up aligned on both hosts: needs **`length + 1` typical-price samples** (direction vs previous bar). Equal typical prices contribute to neither side; only-pos / only-neg MF → 100 / 0 (or 50 when both empty).
`ta.rci`	Rank Correlation Index (Spearman of time vs value ranks) is implemented on the **compile path** (`numba_rci`) as well as interpret; stays in numeric mode when called.

0.3.4+: `ta.atr` is Wilder RMA of TR on interpret + Numba (not EMA-of-TR). EMA dual-host seed is full-list SMA seed matching incremental/Numba. First-party dual-host goldens (ATR / Supertrend / Keltner) and `test_ta_incremental` gate residual drift. Any remaining gaps are tracked under missing features—not as silent “close enough” for the rows above.

Parity harness

```
# Always-on smoke (stable scripts under tests/test_interp_compile_parity.py)
pytest tests/test_interp_compile_parity.py -q

# Corpus compare (default ~50 scripts × 1000 bars; writes report JSON)
python scripts/compare_interp_compile.py --bars 1000 --limit 50
python scripts/compare_interp_compile.py --glob 'average_*.pine' --bars 200

# Optional longer pytest path
pytest tests/test_interp_compile_parity.py -m interp_compile_full

# Formula locks (RSI/ROC/WMA/cum/math.avg/highestbars, …)
pytest tests/test_compiler_numba.py -k 'InterpCompilePlotParity or highestbars_lowestbars_negative
```

Report path: `.cache/interp_compile_parity.json`. Methodology and acceptable relative-error bands vs TradingView: Numerical validation.

Invariants & edge cases

1. **Warm-up → `na`.** Insufficient bars (e.g. SMA length (N) needs (N) samples; RSI needs `period+1` prices) yield `None` / leading `na` in full-series mode.
2. **NA in window.** Many kernels propagate `na` if any window element is missing (SMA-like sums, **WMA**, MFI money-flow samples). **`ta.cum` is the TV exception** (`na` → 0 contribution).
3. **Default sources.** `ta.atr` pulls high/low/close from `current_series` / context when not passed explicitly.
4. **Compile kernels must track interpret.** Numba `numba_*` implementations are separate source; for the parity table above they are formula-locked by dual-host tests—do not "approximate" for speed.
5. **No chart look-ahead.** Kernels only see history available at the current bar index; `request.security` lookahead is a separate concern.
6. **Incremental ≡ full recompute (oracle).** ATR seed is already Wilder RMA on both hosts; changing seed rules is a correctness project, not a silent perf flag.

Worked examples

Classic overlay

```
//@version=6
indicator("SMA 14")
v = ta.sma(close, 14)
plot(v)
```

In bar mode each visit returns one float (or `na`); the host stacks them into a series for the response envelope.

Cross entry signal

```
//@version=6
strategy("X")
fast = ta.ema(close, 12)
slow = ta.ema(close, 26)
if ta.crossover(fast, slow)
    strategy.entry("L", strategy.long)
```

Cross state is bar-local; ensure the runtime resets cross call indices when reusing an evaluator.

Multi-value unpack

```
[macdLine, signal, hist] = ta.macd(close, 12, 26, 9)
plot(hist)
```

Failure modes

Symptom	Cause
Always `na`	Length longer than available history; or series not updated
Wrong vs TV by large margin	Wrong source series; mock OHLCV; or non-bar-mode list composition bug
Cross never fires	Call-index state not reset / shared incorrectly across bars
Compile diverges from interpret	Kernel drift in `numba_builtins.py` (RSI must stay Wilder; ROC warm-up must stay `na`; WMA full window; highestbars sign; `math.avg` na rule)—re-run the parity harness
Early ROC / nested WMA finite too soon	Legacy quirks: ROC returned `0.0` before lookback, or WMA reweighted partial/`na` windows—fixed on both hosts
`high[ta.highestbars(...)]` mostly `na`	Offsets are **negative**; only offset `0` indexes the current bar without `math.abs`
`math.avg` finite when an arg is `na`	Bug: multi-arg avg must propagate `na`, not skip

See also

- Series & history
- Strategy builtins
- Numba path
- Numerical validation
- Missing features (residual dual-host gaps)
- Compiler overview

STRATEGY BUILTINS

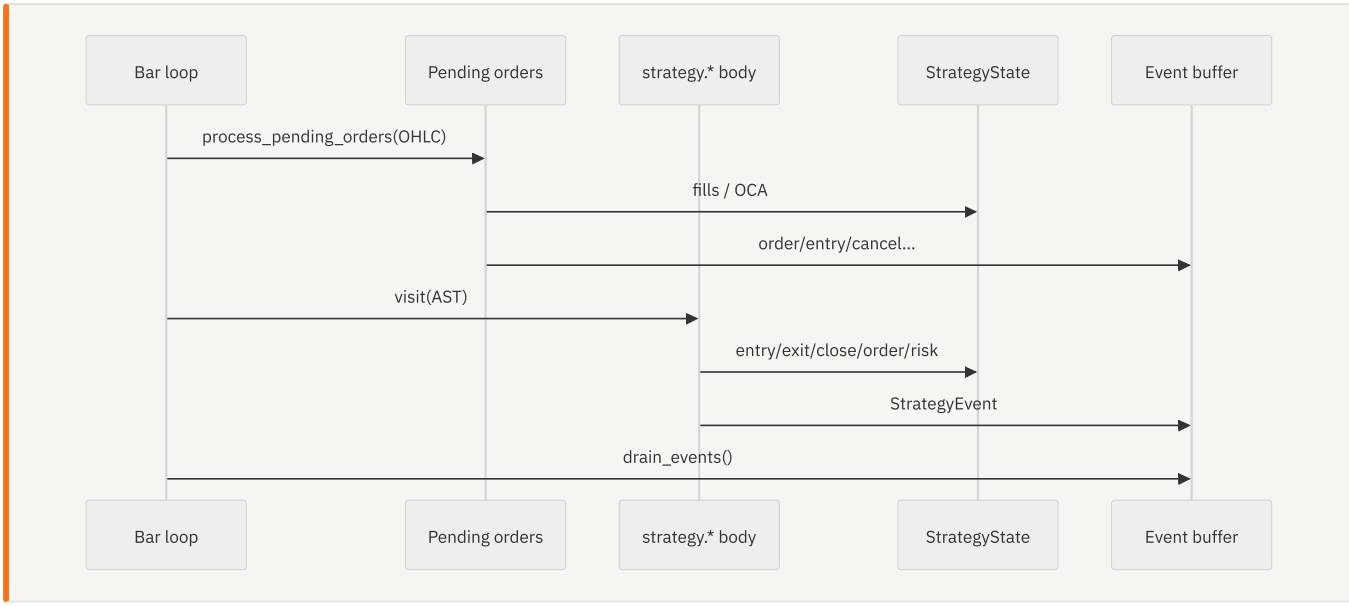
Orders, fills, OCA, commission, risk gates, position metrics, and StrategyState.

STRATEGY BUILTINS

Abstract

PYNE's strategy layer is a **per-run broker simulation** driven by `strategy.*` calls during the bar loop. It is not a live exchange adapter: orders become fills against bar OHLC (market immediately; limit/stop via pending book), commissions and slippage adjust PnL, and risk helpers can block entries. Structured StrategyEvent records form the parity contract with pyne-worker and the HOOX trade mesh.

Conceptual model



Fill-before-script matches the interpreter `Runtime` and the compile-path strategy broker.

Interface surface

Order placement

Builtin	Role
`strategy.entry`	Open/add (or reverse) by direction; market or pending limit/stop
`strategy.exit`	Bracket-style exit (pending limit/stop + OCA); market exit when no levels
`strategy.close` / `strategy.close_all`	Flatten by id or all (`qty_percent` supported)
`strategy.order`	Lower-level order with OCA name/type, partial fill cap
`strategy.cancel` / `strategy.cancel_all`	Remove pending orders

Directions accept Pine constants (`strategy.long` / `strategy.short`) and common string aliases.

`strategy.exit` surface (0.3.4+)

Feature	Status
Pending stop/limit brackets + OCA	Interpret + compile — fills via `process_pending_orders` OHLC path
`from_entry` multi-leg filter	Interpret + compile — unknown id is soft no-op
`qty_percent`	Interpret + compile `close` — `%` of target (whole pos or `from_entry`); wins over `qty`; `≤0`/na → no-op; `>100` capped
Trail (`trail_offset` / `trail_points` / optional `trail_price`)	Interpret + compile — distances are **ticks** (`× mintick`). Pending stop ratchets from bar high/low (OHLC approx). `trail_points=0` / `na` is ignored so a valid `trail_offset` still applies (0.3.8).
`profit` / `loss`	**Ticks** from the exit's entry average (`ticks * mintick`). Long: `entry ± offset`; short flips the sign. `na` / `≤0` ignore that leg. Absolute `limit` / `stop` win when both are set.

```
//@version=6
strategy("Exit demo", overlay=true, initial_capital=100000)
if bar_index == 5
    strategy.entry("L", strategy.long, qty=10)
if bar_index == 6
    strategy.exit("X", from_entry="L", stop=low * 0.98, limit=high * 1.02, qty_percent=50)
// Tick form (same surface): profit/loss are offsets from entry avg, not prices.
// strategy.exit("X", from_entry="L", profit=200, loss=100)
```

Position and performance series

Zero-arg builtins include `strategy.position_size` (signed: +long / −short), `position_avg_price`, `position_entry_name`, `opentrades` / `closedtrades` counts, `netprofit`, `openprofit`, `equity`, `cash`, gross win/loss stats, averages, max drawdown/runup, and max contracts held.

`strategy.position_avg_price` is the VWAP of **open** entry fills. A filled position is **not** overwritten when `strategy.entry` is called again with the same id (`if cond: strategy.entry("L", strategy.long)` stays at the first fill when `pyramiding=0`). Extra same-direction fills only occur when `opentrades < pyramiding + 1` (PYNE cap = `pyramiding + 1`; TV `pyramiding` default 1 ≡ PYNE 0). Pending limit/stop orders with that id are still replaced. When flat the series is `na`.

Trade queries

Indexed accessors:

- `strategy.closedtrades.entry_*` / `exit_*` / `profit` / `size` / `commission`
- `strategy.opentrades.entry_*` / `size` / `profit` / `commission`
- Open/closed trade fields include bar/time/id/comment plus approximate MAE/MFE (`max_drawdown` / `max_runup` from bar high/low)

Risk (interpret + compile halt cascade)

Builtin	Effect
`strategy.risk.max_position_size`	Cap size as % of equity
`strategy.risk.max_intraday_loss`	Intraday loss gate — percent-of-initial-capital default, `strategy.cash` absolute (0.6.0, both brokers)
`strategy.risk.max_intraday_filled_orders`	Order count cap (day-scoped, enforced on both brokers)
`strategy.risk.max_drawdown`	Absolute / percent drawdown halt
`strategy.risk.max_cons_loss_days`	Consecutive losing calendar days → `entries_blocked`
`strategy.risk.allow_entry_in`	`"all"` | `"long"` | `"short"`

Blocked entries still emit a diagnostic-style event with comment `risk_blocked` where implemented. Compile broker shares the common halt cascade above (not a full TradingView® risk engine).

Declaration

`strategy(title, …)` applies broker settings onto `StrategyState`: `initial_capital`, commission type/value, slippage ticks, pyramiding, `avg_price_model`, etc.

Average price model (pynescript extension)

Not part of official TradingView Pine. Controls how `strategy.position_avg_price` evolves when size changes:

`avg_price_model`	On same-direction adds	On partial reduce
`"stock"` (default)	Arithmetic VWAP of fills	Re-average remaining open legs (FIFO multi-trade style)
`"futures"`	Same arithmetic VWAP	**Sticky** net AEP until flat (linear USDT-M / BTCUSDT-like)
`"inverse"`	Arithmetic today; harmonic add planned	Sticky until flat (same reduce rule as futures)

```
strategy("Perp style", avg_price_model="futures")
// or tokens:
// strategy(..., avg_price_model=strategy.avg_price_futures)
```

Linear add formula (stock and futures):

[avg' = \frac{'{'}avg \cdot |size| + p \cdot q{'}'}{'{'}|size| + q{'}'}]

Under `"futures"`, partial closes realize PnL against the sticky average (not per-leg FIFO entry prices), and closed-trade `entry_price` records that same average. Commission is never folded into the average numerator.

There is no `strategy.order.avg_future_price` series—use `strategy.position_avg_price` with the mode switch.

Leverage (pynescript extension — simpler futures UI)

Exchanges expose **leverage** (e.g. 5× / 10× / 20×). Prefer that over TV-style `margin_long` / `margin_short` percentages:

```
//@version=6
// TV-safe: strategy() kwargs need *const* (literals / const vars).
// Host UI can still inject runtime input overrides into PYNE.
strategy(
     "BTCUSDT style",
     avg_price_model="futures",
     leverage=10,
     default_qty_type=strategy.percent_of_equity,
     default_qty_value=100,  // use full equity as *margin*
     initial_capital=10_000)

// Optional Inputs-tab control for *trade logic* (not TV strategy Properties):
// lev = input.float(10, "Leverage", minval=1, maxval=125)
```

Setting	Effect
`leverage=N` (default `1`)	Buying power multiplier
`percent_of_equity` / `cash` default qty	`qty = margin × leverage / price`
`fixed` default qty	Unchanged (contracts as written)
`strategy.cash` / capital held	Margin locked = notional / leverage
`strategy.margin_liquidation_price`	Simple isolated estimate when `leverage > 1`
`strategy.leverage`	Read-back of configured multiplier

`leverage=10` also sets internal margin % to `100/10 = 10` (TV `margin_long` / `margin_short` equivalent). If only `margin_long` is set, leverage is derived as `100 / margin_long`.

`input` before `strategy()` — TV vs PYNE

	TradingView	PYNE
Statements before `strategy()`	Allowed for **const** vars (and types/enums/imports)	Allowed
`input.*()` return qualifier	**`input`** (not `const`)	Plain Python value (no qualifier system)
`strategy(leverage=input.float(...))`	**Compile error** — most `strategy()` params require **`const`**	**Interpret:** works. **Compile:** folds **const-like defval** into the broker ctor (`input.float(10)` → `leverage=10`)
User-adjustable leverage on TV	**Settings → Properties** ("Long/Short leverage", mapped from `margin_long` / `margin_short`)	Declaration `leverage=` or host-injected inputs

Qualifier hierarchy on TV: `const < input < simple < series`. A parameter that needs `const` **cannot** accept an `input` value. So even if `lev = input.float(10)` appears *before* `strategy()`, you still cannot pass `lev` into `strategy(...)` on TV.

PYNE does **not** enforce qualifiers; for compile-path parity with TV's const rule, non-const strategy kwargs are skipped unless their series folds to a constant defval.

Internals

`StrategyState` (`strategy.py`)

Per-evaluator instance fields include:

- Position: direction (`flat` / `long` / `short`), size (non-negative internal), entry metadata
- Books: `pending_orders`, `open_trades`, `closed_trades`
- Economics: capital, commission model, slippage, mintick
- Risk: max position %, drawdown, consecutive loss days, `entries_blocked`
- Equity curve peak/trough for max DD / runup
- `_events: list[StrategyEvent]` drained each bar

`signed_position_size()` implements Pine's signed `strategy.position_size`.

`Order` and fills

```
Order: market | limit | stop | stop-limit
  + oca_name / oca_type ∈ {none, cancel, reduce}
  + max_fill_per_bar (0 = fill remaining)
```

`process_pending_orders(open, high, low, close)` walks the book, computes trigger prices from OHLC (gap-aware open logic for limits/stops), applies commission/slippage, updates trades, runs OCA side effects, and emits events.

OCA

`oca_type`	On fill of a group member
`cancel`	Cancel other pending in group
`reduce`	Reduce sibling quantities by fill size
`none`	Independent

Covered by `tests/test_oca_commission.py` and order-fill suites.

Invariants & edge cases

1. **Isolation.** Never use class-level strategy state; concurrent runs must not share books.
2. **Pyramiding.** Additional entries respect `pyramiding` from the declaration.
3. **Partial fills.** `max_fill_per_bar` / defaults allow multi-bar completion of large orders.
4. **Calendar buckets for cons-loss.** Exit timestamps in ms vs seconds vs bar index are normalized in `note_closed_trade_day`.
5. **Compile path.** Object-mode uses `CompileStrategyBroker`—aligned fill rules, smaller surface than full interpreter metrics. Prefer interpret mode for full `strategy.closedtrades.*` analytics unless compile coverage is verified.

Worked examples

Market long / close

```
//@version=6
strategy("Demo", overlay=true, initial_capital=100000)
if bar_index == 5
    strategy.entry("L", strategy.long, qty=10)
if bar_index == 10
    strategy.close("L", qty_percent=100)
```

Parity fixtures under `tests/fixtures/parity/pine/` encode expected event sequences for such scripts.

Pending stop entry

```
//@version=6
strategy("Breakout", overlay=true)
strategy.entry("Break", strategy.long, stop=high[1])
```

Creates a pending stop; subsequent bars' `process_pending_orders` fill when `high` trades through the stop, using open-gap-aware fill prices.

OCA bracket sketch

```
//@version=6
strategy("Bracket", overlay=true)
// after entry "L" is open:
strategy.order("TP", strategy.short, qty=1, limit=tp, oca_name="br", oca_type=strategy.oca.cancel)
strategy.order("SL", strategy.short, qty=1, stop=sl, oca_name="br", oca_type=strategy.oca.cancel)
```

First fill cancels the sibling.

Failure modes

Symptom	Cause
Entry never fills	Pending stop/limit; OHLC never trades through; or risk block
Double entries	Pyramiding > 0 or missing close
Events missing `script_id`	Host forgot to stamp after `drain_events` (Runtime does this)
PnL off vs TV	Commission type, slippage ticks, mintick, or fill price model
Interpret vs compile event drift	Broker subset / timing—diff with `tests/test_compiler_strategy.py`

See also

- Events
- Strategy broker (compile)
- pyne-worker

- Pro API backtest

COLLECTIONS

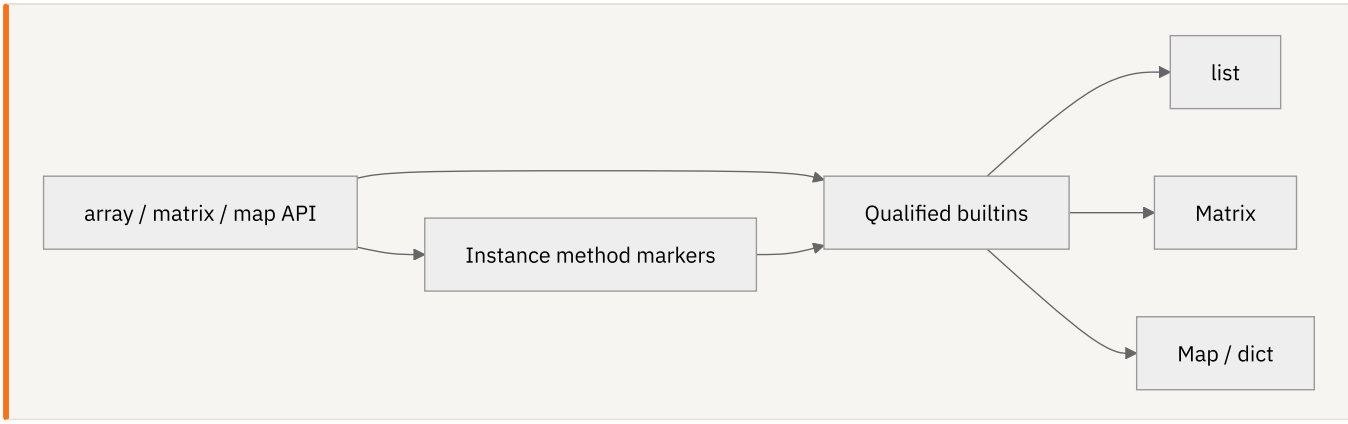
array.*, matrix.*, and map.* runtime semantics, method dispatch, and type tags.

COLLECTIONS

Abstract

Pine collections—**arrays**, **matrices**, and **maps**—are mutable values that live in the evaluator context (often under `var` so identity persists across bars). PYNE maps them to Python `list`, a dedicated `Matrix` type, and a `Map` / `dict`-like type respectively, with builtin functions and instance-method sugar (`a.push(x)` → `array.push(a, x)`).

Conceptual model



Interface surface

Arrays (`array.*`)

Represented as Python lists. Highlights from the dispatch map:

Category	Functions
Construct	`array.new` , `array.new_<type>` , `array.from`
Mutate	`push` , `pop` , `shift` , `unshift` , `insert` , `remove` , `set` , `fill` , `clear` , `reverse` , `sort`
Access	`get` , `first` , `last` , `size` , `includes` , `indexof` , `lastindexof` , `slice`
Aggregate	`sum` , `avg` , `min` , `max` , `median` , `mode` , `range` , `stdev` , `variance` , covariance, percentiles, percentrank
Search	`binary_search` , `binary_search_leftmost` , `binary_search_rightmost` — UDT arrays take `sort_field` (const int index, default 0, or const string name; array must be sorted by that field ascending)
Higher-order	`every` , `some` , `copy` , `concat` , `join` , `abs`

Indexing: Pine array indices are **0-based from the start of the list** (unlike series history). Negative indices are rejected for series; array paths use list semantics consistent with the builtin handlers.

UDT `sort_field` (0.3.6 / Pine August 2026): `array.sort`, `array.sort_indices`, and `array.binary_search*` compare one field of user-defined type elements. Pass a const int field index (`0` is the first declared field, and the default for search) or a const string field name. Binary search requires the array to already be sorted by that field in ascending order. Method form (`a.binary_search(value, "id")`) and `sort_field=` kwargs work on interpret and compile.

Series-like values passed into array ops unwrap via `.history` (reversed to chronological) when duck-typed.

Matrices (`matrix.*`)

`Matrix` supports:

- Construction: `matrix.new`
- Element access: `matrix.get` / `set`, and `m[row, col]` via subscript
- Shape: `rows`, `columns`, `elements_count`
- Row/column ops: add/remove/copy, sum/avg/min/max/mode, fill
- Aggregates: sum/avg/min/max/mode all
- Transforms: `transpose`, fill diagonal, and further linear helpers in the evaluator map

Method sugar: `m.rows()` → `matrix.rows(m)`.

Maps (`map.*`)

Function	Role
`map.new`	Empty map
`put` / `put_all` / `get` / `remove` / `clear`	Mutation
`contains` / `keys` / `values` / `size` / `copy`	Query

Keys follow Pine map rules as implemented (hashable Python keys). Compile object-mode lowers maps to ordinary Python `dict`s.

Internals

Path	Role
`src/pynescript/ast/evaluator/builtins/arrays.py`	Array builtins
`src/pynescript/ast/evaluator/builtins/matrix.py`	`Matrix` type
`src/pynescript/ast/evaluator/builtins/matrix_evaluator.py`	matrix.* handlers
`src/pynescript/ast/evaluator/builtins/map.py`	`Map` type
`src/pynescript/ast/evaluator/builtins/map_evaluator.py`	map.* handlers
`src/pynescript/ast/evaluator/names.py`	Method markers for list/Matrix
`tests/test_collections*.py`, `test_map_collections.py`, `test_matrix_*.py`	Coverage

Multi-dispatch type tags (`array.float`, `matrix.string`, ...) interact with method overloading and `na` exclusion lists so `na`-typed optionals do not bind to collection `tostring` overloads.

Invariants & edge cases

1. **Identity under `var`.** `var a = array.new_float(0)` keeps one list across bars; reassigning `a := array.new_float(0)` replaces it.
2. **Drawing-typed arrays.** `array.new_label` etc. store drawing object references; delete semantics remain drawing-registry concerns.
3. **Empty array overload match.** Empty arrays match any `array.*` element tag in dispatch (no sample element).
4. **Matrix unpack hazard.** StatementEvaluator refuses to treat matrices as general iterables for tuple unpack—prevents corrupting multi-assign from row iteration.
5. **Compile mode.** Maps/UDTs force **object mode**; pure array numeric work may still numeric-compile depending on visitor detection—verify generated code when performance-critical.

Worked examples

Rolling buffer

```
//@version=6
indicator("buf")
var floats = array.new_float(0)
array.push(floats, close)
if array.size(floats) > 20
    array.shift(floats)
plot(array.avg(floats))
```

Matrix element

```
var m = matrix.new<float>(2, 2, 0.0)
matrix.set(m, 0, 1, close)
plot(matrix.get(m, 0, 1))
```

Map counter

```
var counts = map.new<string, int>()
map.put(counts, "n", nz(map.get(counts, "n")) + 1)
```

Binary search on a UDT array

```
//@version=6
indicator("udt search")
type Item
    int id
    float v
var a = array.new<Item>()
if bar_index == 0
    array.push(a, Item.new(2, 20.0))
    array.push(a, Item.new(1, 10.0))
    array.push(a, Item.new(3, 30.0))
    array.sort(a, order.ascending, "id")
plot(array.binary_search(a, 2, "id"))
plot(a.binary_search_leftmost(2, sort_field="id"))
```

Failure modes

Symptom	Cause
`map.get requires map and key`	Wrong arity or non-Map receiver
Index errors	OOB get/set without Pine `na` guard in that handler
Method not found	Typo or non-list receiver after series unwrap failed
Shared mutation across runs	Reused evaluator context without reset

See also

- Expressions & statements
- Drawing & plotting — arrays of labels/lines
- Compiler object mode

DRAWING AND PLOTTING

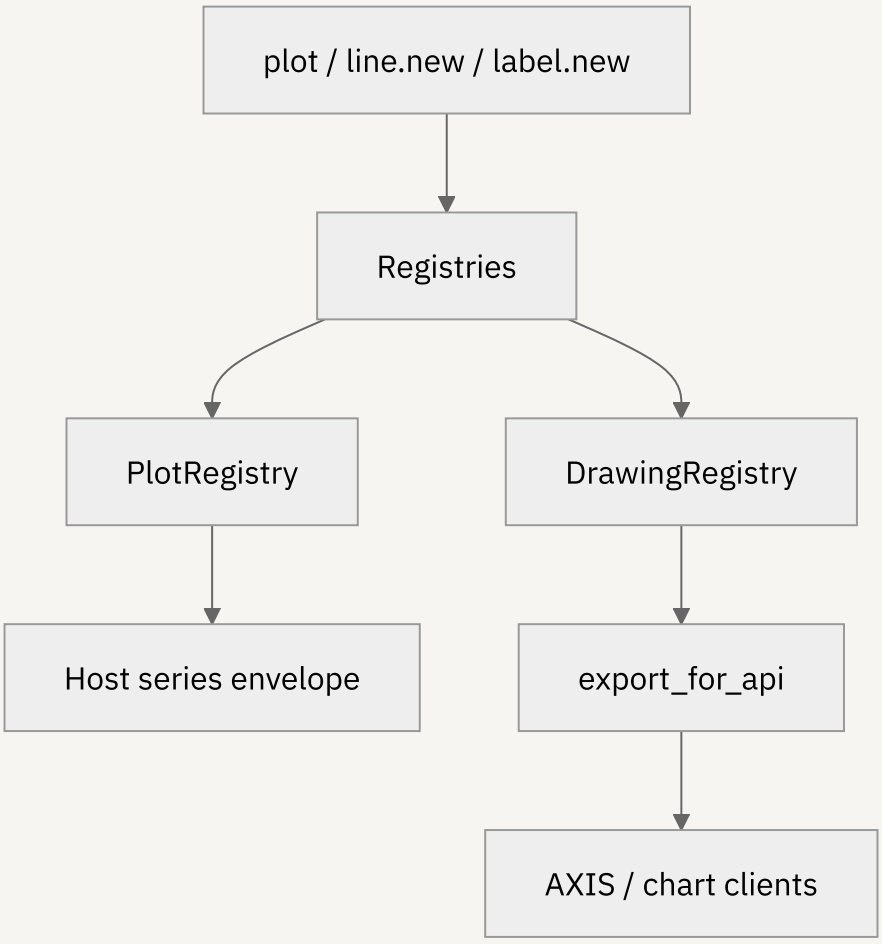
plot* side effects, DrawingRegistry objects, and export shapes for hosts and AXIS.

DRAWING AND PLOTTING

Abstract

Visual effects in Pine are **side channels**: they do not change pure arithmetic results, but they are first-class runtime outputs. PYNE records plots in `PlotRegistry` and drawing primitives (lines, boxes, labels, tables, polylines, linefills) in `DrawingRegistry`, then serializes them for the Pro API and optional AXIS. Evaluation never requires a UI—tests assert on registries alone.

Conceptual model



Interface surface

Plotting (`PlottingFunctionsMixin`)

Function	Kind	Notes
`plot`	`plot`	Returns `Plot` id for `fill(plot1, plot2)` ; series values in host export
`plotshape` / `plotchar` / `plotarrow`	markers	Condition series + `plot_meta.kind` (style/location/char). Visuals are **not** in `drawings`
`barcolor` / `plotbar` / `plotcandle`	overlays	Per-bar color / OHLC series + kind stamp (0.3.12 dual-host)
`hline`	horizontal	**Both** hosts export a **constant price series** keyed by title + `plot_meta.kind: "hline"` . Not in `drawings`
`bgcolor` / `barcolor`	coloring	Color-string series + `plot_meta.kind` . Not in `drawings`
`fill`	fill	**Both** export a **titled series key** (values often all-null); band color + plot refs live in `plot_meta`

Style constants: `plot.linestyle_solid` / `dashed` / `dotted` . Plot style tokens: `plot.style_line` , `linebr` , `stepline` , `steplinebr` , `stepline_diamond` , `histogram` , `columns` , `cross` , `area` , `areabr` , `circles` .

Each call constructs a `Plot` dataclass (`kind` , `series` , `title` , `color` , linewidth, text formatting, force_overlay, ...) and appends it to `PlotRegistry.plots` .

Hosts often **also** capture per-bar plot values on the evaluator (`plot_outputs` / `_plot_value_cols`) for time-aligned series maps—see `src/pynescript/runtime/evaluator.py` and packaging in `src/pynescript/runtime/host.py` (`backend.evaluator` / `backend.runtime` re-export those modules).

`PYNE_LIGHT_PLOTS=1` skips columnar capture, plot-registry handles, and `input.*` metadata (corpus OK/fail only).

Title defaults and series keys

Runtime packages plots into `result["series"]` / `result["plot_meta"]` on **both** hosts. `result["drawings"]` is **geometry-only** (line / box / label / polyline / table / linefill). Compile `__drawings` is an internal event list that Runtime **folds and strips** before JSON — hosts never need it. Empty or missing titles get defaults; collisions get `_2` , `_3` , ... suffixes so map keys stay unique.

Call	Default title	Notes
`plot`	`plot_0` , `plot_1` , ...	Compile emits `plot_{n}` at emit time; interpret empty title → `plot_{index}` at JSON packaging
`hline`	`hline`	Uniquified `hline` , `hline_2` , ... when untitled levels repeat
`fill`	`fill`	Same uniquify rule (`Background` / `Background_2` , ...)
`bgcolor`	`bgcolor`	Series + `plot_meta` (both hosts)
`plotshape`	`shape`	Series + `plot_meta` (both hosts)

Explicit `title=` (or positional title) wins when present.

`hline` — constant series (both modes)

Interpret (`src/pynescript/runtime/evaluator.py` + Runtime packaging):

- Captures kind `hline` with the price as the series cell each bar.
- JSON export **forward-fills** nulls with the known constant so AXIS can draw a full-width level.
- `plot_meta[title]` includes `kind: "hline"`, `price`, `color`, `linewidth`, ...

Compile:

- Emits a **plot array** (`safe_float(price)` every bar) under the uniquified title and stamps `plot_meta.kind: "hline"`.
- May also append an internal `__drawings` event; Runtime packaging **strips** visual kinds so host `drawings` stays geometry-only.
- Host wraps NaN → null; constant levels remain numeric across the bar range.

Parity tests: `tests/test_multi_plot_cross.py` (`test_hline_series_keys_compile_matches_interpret`), `tests/test_plot_drawing_dual_host.py`, `tests/test_compiler_objects.py`.

`fill` — titled series key for key-set parity (both modes)

Interpret:

- Meta stores sibling series titles as `plot1` / `plot2` (resolved from `plot()` handles).
- Capture may hold a color string on the column; **Runtime JSON packaging** typically emits **all-null** series cells for fills (non-numeric / non-bgcolor strings → `null`). Band **color** and plot refs stay in `plot_meta` (and registry fill objects when ids are enabled).

Compile:

- Registers a series key with expression `None` (float64 NaN → JSON `null`) under the uniquified title so **series key sets match interpret**.
- Band details (plot refs, color) land in `plot_meta`. An internal `__drawings` `fill` event, if emitted, is **stripped** before Runtime JSON.

Parity tests: `test_fill_background_series_keys_compile_matches_interpret`, `test_fill_title_series_key_and_drawings`, `tests/test_plot_drawing_dual_host.py`.

Import stubs must not leak as series strings

Unresolved `import …` libraries become chainable stubs marked `__pine_import_stub__ = True` (repr like `<PineImportStub namespace/name/version>`). If a script plots a stub member, Runtime's `_json_plot_value` maps those cells to **`null`**—never the stub repr or a color-like string—so AXIS and interpret/compile comparisons never see `"<PineImportStub …>"` as a plot series value.

Drawing objects (`DrawingBuiltinsMixin`)

Object types: `Line`, `Box`, `Label`, `Table`, `Polyline`, `LineFill`, `ChartPoint`.

Typical factories: `line.new`, `box.new`, `label.new`, `table.new`, `polyline.new`, `linefill.new`, plus getters/setters/deletes and `*.all` / last-bar helpers where implemented.

Instance methods resolve through namespace markers:

```
la.get_text()  →  label.get_text(la)
```

`DrawingRegistry.export_for_api(bar_times)` maps `xloc=bar_index` coordinates to wall times for chart clients, normalizes colors, and skips deleted objects. `xloc.bar_index` **past the last bar** (classic `bar_index + 1` on `barstate.islast`) is extrapolated from the series period (0.3.5); empty `bar_times` passes the bare bar index.

Line / box / label payloads include **`force_overlay`** for AXIS pane routing (0.3.6). `linefill.new` serializes as `type: "linefill"` quads (`t1` / `p1` … `t4` / `p4` from the two line endpoints, plus `color` / `bgcolor`).

Registry lifecycle

```
DrawingRegistry.reset()  # clears drawings + PlotRegistry
```

Hosts must reset at run start so labels from a previous evaluation do not leak.

Drawing GC (`max_*_count`)

Interpret path enforces TradingView®-style **garbage collection** on drawing factories (`line.new`, `label.new`, `box.new`, `polyline.new`, …):

Declaration kwargs (**`indicator`** / **`strategy`**)	Default	Hard cap
`max_lines_count`	50	500
`max_labels_count`	50	500
`max_boxes_count`	50	500
`max_polylines_count`	50	100

When more **active** objects of a type exist than the cap, the **oldest** are marked `deleted=True` so they leave `*.all` and `DrawingRegistry.export_for_api`. Caps are applied from the script declaration at run start (`DrawingRegistry.configure_from_declaration`); shrinking a cap immediately collects.

```
//@version=6
indicator("labels", overlay=true, max_labels_count=100)
if barstate.islast
    label.new(bar_index, high, "x")
```

Compile object-mode drawings are a simplified event list—GC caps apply on the **interpret** registry path; prefer interpret when you need full last-bar / `*.all` / cap behavior.

`fill` + **`plot_meta`** for AXIS

For dual `plot` + `fill` bands:

1. Each `plot()` contributes a **numeric series** under its title (default `plot_0`, …).
2. `fill(p1, p2, …)` contributes a **titled series key** (often all-null cells) so key-sets stay stable, plus `plot_meta` that carries band color and plot refs.
3. Hosts such as AXIS should read **`plot_meta[title]`** for `kind`, colors, and sibling plot titles (`plot1` / `plot2`) rather than treating fill series as prices.

Interpret packaging lives in `src/pynescript/runtime/host.py` / `evaluator.py`; compile emits matching series keys. Internal `__drawings` fill events are folded/stripped before Runtime JSON.

Internals

Path	Role
`src/pynescript/ast/evaluator/builtins/plotting.py`	`Plot`, `PlotRegistry`, plot* handlers
`src/pynescript/ast/evaluator/builtins/drawing.py`	Drawing types, registry, builtins, `export_for_api`, `fold_compile_drawing_mutations`
`src/pynescript/runtime/evaluator.py`	Host plot capture: hline / fill / bgcolor / plotshape series + meta
`src/pynescript/runtime/host.py`	JSON packaging: series keys, stub nulling, hline fill-forward, fill plot refs
`src/pynescript/compiler/compiler.py`	`hline` / `fill` series + internal `__drawings`; visual events lifted then stripped
`src/pynescript/ast/evaluator/names.py`	`_DRAWING_METHOD_NS`
`tests/test_plotting_effects.py`, `test_drawing_all_and_last_bar.py`, `test_multi_plot_cross.py`, `test_bgcolor_plotshape_export.py`, `test_plot_drawing_dual_host.py`, `test_compiler_objects.py`	Behavior / parity

Compile object-mode notes

Geometry factories and some marker-style calls force **object mode** (pure-Python bar loop with an append-only `__drawings` event list). Runtime packaging **lifts** visual events into `series` + `plot_meta.kind`, then **strips** them so host `drawings` is geometry-only — the same AXIS shape as interpret `export_for_api`.

Call	Host export (both modes)	Compile internals
`hline`	Constant series + `plot_meta.kind`	Series array; optional `__drawings` event stripped
`fill`	All-null series key + `plot_meta` color/refs	Series key; optional `__drawings` event stripped
`plotshape` / `plotchar` / `plotarrow`	Condition/signed series + `plot_meta.kind`	Visual events lifted into series, then stripped
`bgcolor` / `barcolor`	Color-string series + `plot_meta.kind`	Same lift/strip as other visuals
`plotbar` / `plotcandle`	OHLC series + kind stamp	Same lift/strip
`label` / `line` / `box` / `polyline` / `table` / `linefill`	Geometry objects in `drawings`	Handle dict + `__drawings`; `set_*` / `*.delete` fold via `DrawingRegistry.fold_compile_drawing_mutations`

Numeric mode still supports plain `plot` float64 arrays; geometry or marker calls can flip object mode for that script.

Invariants & edge cases

1. **`plot` returns an id.** Required for `fill`; treating it as "void" breaks band fills. Hosts that skip `PlotRegistry` unless `fill(` appears in source must still soft-fail fill gracefully.
2. **Deleted flag.** Soft-delete keeps history for debugging; `active()` filters deleted plots and GC-collected drawings.
3. **`max_*_count` GC.** Oldest active drawings are deleted when caps are exceeded (interpret registry).
4. **Series in drawings.** Prices may be series wrappers—export coerces `.current` and maps NaN → omit.
5. **Compile path.** Numeric mode supports `plot` arrays; geometry/markers force object mode with an internal `__drawings` event list. Runtime **folds** `set_*` / `*.delete`, **lifts** visual events into `series` + `plot_meta.kind`, then **strips** them so `drawings` is geometry-only. Prefer interpret when you need full last-bar / `*.all` / registry GC — not because visuals lack series.
6. **Import stubs → null.** `__pine_import_stub__` and `"<PineImportStub…>"` strings never appear as plot series cells after Runtime packaging.
7. **No GPU/canvas in-process.** Registries are data; AXIS/HOOX render elsewhere.

Worked examples

Dual plot + fill

```
//@version=6
indicator("BB mid")
basis = ta.sma(close, 20)
p1 = plot(basis)
p2 = plot(basis * 1.01)
fill(p1, p2, color=color.new(color.blue, 80))
```

Label on last bar

```
//@version=6
indicator("lbl", overlay=true)
if barstate.islast
    label.new(bar_index, high, str.tostring(close))
```

Hosts read `DrawingRegistry.labels` or API export after the run.

Failure modes

Symptom	Cause
Empty drawings in API	Forgot `DrawingRegistry.reset` timing / export; or script never called `*.new`
fill no-ops	plot ids not retained from `plot()` returns
Stale labels across HTTP runs	Missing registry reset between `Runtime.run` calls (Runtime resets—custom hosts must)
Object mode missing drawing	Compile still folds `set_*` / `delete`; leftover events usually mean fold did not run
Only 50 labels/lines visible	Default `max_*_count`; raise on `indicator` / `strategy` (≤ hard cap)
fill series all null	Expected for band keys; use `plot_meta` for color + plot refs

See also

- Builtins hub
- Pro API preview / chart renderer
- AXIS docs
- Compiler overview

REQUEST.* AND INPUT.*

Multi-symbol/timeframe requests, fundamental na semantics, host series bind, and input parameter resolution.

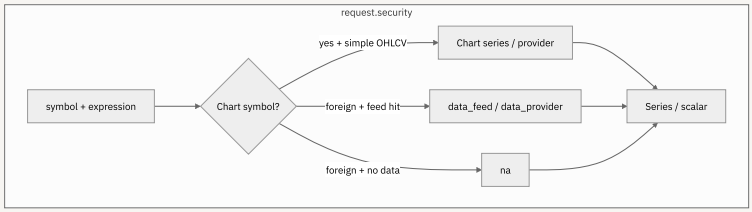
`request.*` AND `input.*`

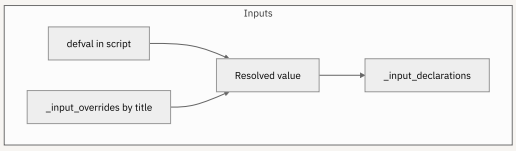
Abstract

Two namespaces couple scripts to the **host environment**: `input.*` declares parameters (with defaults and UI metadata), and `request.*` pulls data from other symbols, timeframes, or fundamental/economic sources. PYNE implements both as builtins with explicit extension points— `data_feed`, `data_provider`, and `_input_overrides` —so the same script can run offline or online with real market data.

Design rule: missing multi-symbol or fundamental data yields `na`, never silent substitution of chart OHLCV (no close-as-dividend, no chart volume as UPVOL).

Conceptual model





Interface surface

`input.*` (`InputBuiltinsMixin`)

Builtin	Purpose
`input`	Generic defval + title/tooltip/inline/group/confirm/active
`input.bool` / `int` / `float` / `string` / `color`	Typed scalars
`input.price` / `source` / `time` / `timeframe` / `session` / `symbol`	Domain inputs
`input.enum` / `input.text_area`	Enumerations and multiline text

Runtime value: each call returns the resolved value (override if title matches, else default). Metadata is appended to `_input_declarations` for settings panels and LSP-adjacent hosts.

Overrides live on the evaluator as `_input_overrides: dict[title, value]`.

`request.*` (`RequestBuiltinsMixin`)

Builtin	Role
`request.security`	Other symbol / timeframe expression
`request.security_lower_tf`	Lower-TF array expansion
`request.dividends` / `earnings` / `splits`	Corporate actions
`request.financial` / `economic` / `quandl`	Fundamentals / external series
`request.currency_rate`	FX conversion helper
`request.seed`	Deterministic pseudo-series
`request.footprint`	Volume footprint object (v6 surface)

`request.security` — interpret path

Resolution order (simplified):

1. Normalize symbol (series → last element; `ticker.*` → symbol string) and timeframe.
2. Classify **chart vs foreign** (`_is_chart_symbol` vs host `syminfo` / provider symbol).
3. Try `data_feed.fetch_latest_ohlcv` / ticker, then `data_provider.fetch` .
4. **Same-symbol coarser timeframe (HTF resample, 0.3.4+):** bucket chart OHLCV to the requested TF; **last completed** HTF bar by default (lookahead_off-style). Allowlisted simple TA on HTF series — `ta.sma` / `ta.ema` / `ta.rsi` / `ta.atr` / `ta.wma` / `ta.rma` (bare `close` / `high` /... source + const length; `ta.atr(n)` length-only) — evaluates on the unique completed HTF series. Nested `ta.*` , UDFs, and non-const lengths stay `na` . `barmerge.gaps_on` / `lookahead_on` are honored on these resample paths (0.6.0, finalized-bucket lookahead in 0.6.1): gaps deliver the value only on bucket-start bars, lookahead leaks each HTF bucket's final value from the period start on historical bars. Both stay unused on passthrough / provider / complex-na paths. Policy meta is exposed on `result["meta"]["request_security"]` (`gaps_supported` / `lookahead_supported` , `gaps_applied` / `lookahead_applied`).
5. **Foreign + pre-evaluated expression** (UDF result, list/tuple of chart values, non-string expr) without multi-symbol data → **`na`** (not chart close as “dividends”).
6. Fundamental / non-equity prefixes (`DIVIDEND` , `FACTSET` , `EARNINGS` , `ESD_`) with no feed hit → **`na`** (no mock OHLCV).
7. Bare equity-style string names may still use **legacy mock** prices for offline demos when no feed is wired.

`ChartOHLCVProvider` (wired by `Runtime` via `resolve_request_sources` when unset) **only serves the chart ticker**. Foreign tickers get empty series, so interpret returns `na` unless a real multi-symbol feed is configured.

`request.security` — compile path

Compile lowering is intentionally narrow (compiler overview):

Case	Emit
Same-symbol (`syminfo.ticker` / empty chart id) **and** simple OHLCV expr (`close` , `high[1]` , ...)	Passthrough chart array sample
Foreign tickers (`UPVOL.NY` , `ESD_FACTSET` , ...)	`np.nan`
Complex third arg (UDFs, `year_sum(close)` , non-allowlisted `ta.*` , arithmetic)	`np.nan`
Other `request.*` APIs	`np.nan` (object mode)

No inventing chart-close-as-dividends or chart-volume-as-advance/decline series.

Footprint (`FootprintBuiltinsMixin`)

Types `Footprint` and `VolumeRow` expose buy/sell volume, delta, VAH/VAL/POC rows, and per-row imbalance helpers—aligned with the v6 footprint surface inventory.

Mode selection (`mode=auto`)

Pro API `/run` defaults body `mode` to **`auto`**. `Runtime._compile_eligible` **rejects compile** when the source contains `request.` (or top-level `import`), so auto **prefers interpret** for any script that uses `request.*`:

```
request.* present → compile_fallback_reason = "request.* not supported in compile path"
                  → auto_backend = interpret
```

Forced `mode=compile` still runs the narrow lowering above (simple same-symbol only; else `na`). Input overrides also force interpret under auto.

See Compiler overview for warm compile, numeric vs object mode, and eligibility.

Host series bind (interpret)

Assigning host OHLCV / time series into a **user name** never aliases the host buffer by reference. `_bind_series_name` copies the current scalar into a fresh series for the user name so patterns like:

```
last_t := na(close) ? last_t[1] : time
```

cannot corrupt `time[j]` history. That bug previously broke TTM / `year_sum`-style windows (e.g. dividend-yield scripts).

Internals

Path	Role
`src/pynescript/ast/evaluator/builtins/input.py`	Input handlers + declarations
`src/pynescript/ast/evaluator/builtins/request.py`	`request.*` + chart/foreign / na policy
`src/pynescript/ast/evaluator/statements.py`	`_bind_series_name` (no OHLCV alias)
`src/pynescript/util/data.py`	`ChartOHLCVProvider`, `resolve_request_sources`
`src/pynescript/compiler/compiler.py`	Same-symbol OHLCV-only `security` lower
`src/pynescript/runtime/host.py`	Package Runtime SoT; feed wiring; `_compile_eligible` / `_run_auto`; copies policy onto `meta.request_security`
`backend/runtime.py`	Compat re-export of package Runtime (not the implementation)
`tests/test_dividend_yield_parity.py`, `test_datafeed_wiring.py`	na parity + chart provider

Invariants & edge cases

1. **Inputs are pure values at runtime.** Titles matter only for override keys and UI metadata—not for Pine type identity.
2. **Foreign without data → `na`.** Prefer honest missing data over inventing chart series as multi-asset results.
3. **Chart provider is chart-only.** Multi-asset accuracy needs a real `data_feed` / `data_provider`.
4. **Mocks are limited.** Equity-style bare symbols may still mock offline; fundamental prefixes and foreign pre-evaluated exprs do not.
5. **Dynamic symbols.** List/series symbols resolve to the latest element—supports loops constructing ticker ids.
6. **Lower TF.** `request.security_lower_tf` returns array-like structures; length scales with simulated lower-TF density when mocking.
7. **`mode=auto` + `request.*` → interpret.** Compile path is not a full multi-asset substitute (`_compile_eligible` rejects `request.`).
8. **HTF is last-completed only.** Forming HTF buckets are never returned; `gaps` / `lookahead` kwargs do not change that.

Worked examples

Parameterized length

```
//@version=6
indicator("len")
len = input.int(14, "Length", minval=1)
plot(ta.sma(close, len))
```

Host:

```
ev._input_overrides = {"Length": 21}
```

Multi-timeframe close / simple HTF TA (chart symbol)

```
//@version=6
indicator("HTF")
htf = request.security(syminfo.tickerid, "D", close)
htf_sma = request.security(syminfo.tickerid, "D", ta.sma(close, 20))
plot(htf)
plot(htf_sma)
```

Same-symbol coarser TF: interpret buckets chart OHLCV (last **completed** HTF bar) and evaluates allowlisted simple `ta.*` (`sma` / `ema` / `rsi` / `atr`) on that series. Compile may passthrough simple same-symbol OHLCV; complex foreign/security still → `na`. Inspect `meta.request_security.policies` (`htf_ohlcv_resample`, `htf_simple_ta_resample`, `foreign_na`, `complex_htf_na`, …).

Intentional `na` — dividend yield (fundamentals missing)

```
//@version=6
indicator("div")
year_sum(src) =>
    ta.cum(src)
div_ticker = ticker.new("ESD_FACTSET", "X;Y;DIVIDENDS")
div_ttm = request.security(div_ticker, "D", year_sum(close), barmerge.gaps_on, lookahead=barmerge.
plot(div_ttm)
```

Without a fundamentals feed: **interpret and compile both plot `na`** for `div_ttm`—not chart close as fake TTM dividends. Covered by `tests/test_dividend_yield_parity.py`.

Intentional `na` — CVI / UPVOL-style foreign OHLCV

```
//@version=6
indicator("cvi")
up = request.security("UPVOL.NY", "D", close)
plot(up)
```

Foreign ticker + no multi-symbol data → `na`. Compile never rewrites this as chart close; inventing advance/decline volume from the host series is forbidden.

Failure modes

Symptom	Cause
All-`na` multi-asset plots	Foreign symbol / fundamental expr; no feed (expected)
Always ~100 mock prices	Bare equity string + no feed; legacy mock path
Input override ignored	Title string mismatch (including empty title)
Footprint fields zero	`request.footprint` without configured footprint data
`time[1]` / history wrong after assign	Should be fixed: host bind no longer aliases OHLCV
Lookahead surprises	Host data alignment / gaps—not automatic TV replay guarantees
`auto_backend=interpret` with `request.*`	Eligibility prefilter; use interpret or wire multi-symbol feed

See also

- Compiler overview — auto eligibility, same-symbol security lower
- Builtins hub
- Series & history
- Pro API run endpoint
- Runtime hub

LIBRARIES

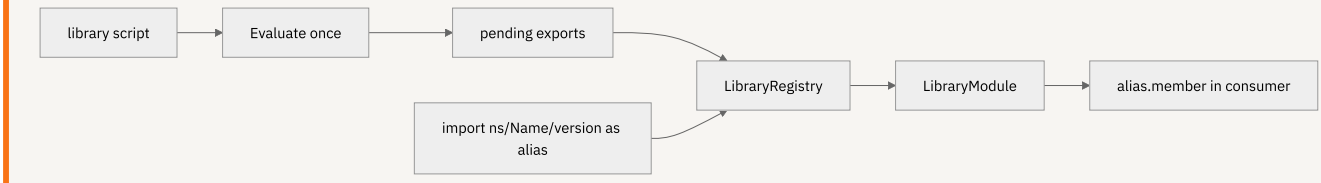
In-process library export/import, LibraryRegistry, and evaluate-order requirements.

LIBRARIES

Abstract

TradingView libraries publish as `namespace/Name/version` with `export`ed members. PYNE implements an **in-process** analogue: evaluate a `library("Title")` script (or register source by path), collect exports (`export const`, exported functions, types), then `import` binds an alias to a `LibraryModule` for `alias.member` access. There is no network publish step—the registry is the host's responsibility.

Conceptual model



Interface surface

Library script

```
//@version=6
library("MyMath")

    a + b
```

On script completion, `StatementEvaluator` finalizes `_pending_library_exports` onto the active `LibraryModule` and calls `LibraryRegistry.register`.

Consumer import

```
//@version=6
indicator("use")

plot(M.add(close, M.PHI))
```

Resolution (`LibraryRegistry.lookup`):

1. Exact `(namespace, name, version)` if provided and registered.
2. Else title-only match (`MyMath`) for local evaluation workflows without publisher path.

API on the evaluator

```
ev.register_library_source(namespace, name, version, source_str)
mod = ev.lookup_library(namespace=..., name=..., version=...)
```

`register_library_source` stores Pine text for **lazy** load on the first matching `import`. `_load_library_source` evaluates definitions only (skips showcase body) and **finalizes** `_pending_library_exports` onto the `LibraryModule` so `export const` / exported functions bind (0.3.8). Evaluating a library script on the same evaluator still populates exports eagerly.

Host evaluate (`0.3.7+`)

Hosts pass published sources without a pre-eval step:

```
Runtime(symbol="TEST").run(
    consumer_source,
    ohlcv,
    mode="interpret",  # or "auto" — import is compile-ineligible
    libraries=[
        {"namespace": "ns", "name": "Lib", "version": 1, "source": library_source},
    ],
)
```

The same list is accepted on Pro API and **pyne-worker** `POST /run` as `libraries`. pyne-worker also stores the array on deployed scripts so cron / `script_id` reuse it. Pro API slices to **32** entries (`backend/app.py`); `Runtime.run` itself does not cap. `mode=compile` has no import path — `mode=auto` treats `import` as compile-ineligible and **forwards** **`libraries=`** **into interpret fallback** (0.3.9).

`LibraryModule`

Dataclass with `title`, optional `namespace` / `version`, and `exports: dict[str, Any]`. Attribute access reads exports; missing members raise `AttributeError` with a clear message. `NameEvaluator` routes `alias.member` when the base value is a `LibraryModule`.

Internals

Path	Role
`src/pynescript/ast/evaluator/libraries.py`	`LibraryModule` , `LibraryRegistry`
`src/pynescript/ast/evaluator/statements.py`	export registration, import visit, library finalize
`src/pynescript/ast/evaluator/base.py`	registry fields on `BaseEvaluator`
`src/pynescript/ast/evaluator/__init__.py`	`register_library_source` , `lookup_library`
`src/pynescript/runtime/host.py`	`Runtime.run(..., libraries=)` + auto-mode forward
`tests/test_library_export_import.py`	Round-trip coverage

Exportable values include callables (user functions), constants, and type-related bindings depending on declaration paths. Exported methods participate in the same call machinery as local functions once retrieved from the module.

Invariants & edge cases

1. **Evaluate library before consumer** (or register source for lazy import). Empty registry → import failure.
2. **Same evaluator instance** (or shared registry) must see both scripts—registries are not process-global unless the host shares them.
3. **Version is part of the path key.** Mismatched version does not fall back to another version automatically when namespace is specified.
4. **Title registration** always updates `_by_title` so simple local titles work without publisher metadata.
5. **No TV account auth.** Publishing/versioning outside the process is out of scope.

Worked examples

Explicit registration

```
from pynescript.ast.evaluator import NodeLiteralEvaluator

ev = NodeLiteralEvaluator()
ev.evaluate_script(library_source)   # registers by title
ev.evaluate_script(consumer_source)  # import by title or path
```

Path registration without pre-eval

```
ev.register_library_source("user", "MyMath", 1, library_source)
# import user/MyMath/1 loads definitions and finalizes exports on first visit
```

Failure modes

Symptom	Cause
Import resolves to nothing	Library never registered / wrong title
`Library 'X' has no exported member 'y'`	Missing `export` or typo
Stale exports	Re-evaluated library without re-import; host cache
Version conflict	Multiple libs same title—last `register` wins for title key

See also

- Expressions & statements
- Builtins declarations
- End-user library API guide

STRATEGY EVENTS

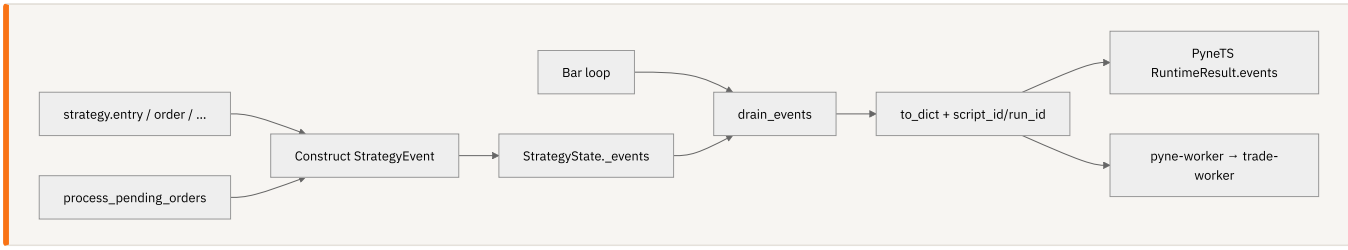
StrategyEvent shape, emission points, parity corpus, and host serialization.

STRATEGY EVENTS

Abstract

A **strategy event** is the immutable, structured record of one `strategy.*` action (or fill side-effect) at a specific bar. Events are the wire format between PYNE's Python evaluator, **PyneTS** (`StrategyEvent` on `RuntimeResult`), and downstream HOOX trade workers (via **pyne-worker**). Changing the dataclass without updating the TS library and tests is a contract break.

Conceptual model



Interface surface

`StrategyEvent` fields

Defined in `src/pynescript/ast/evaluator/events.py`:

Field	Type (logical)	Notes
`kind`	`entry` | `exit` | `close` | `close_all` | `cancel` | `cancel_all` | `order`	Discriminator
`id`	`str | None`	Order / entry id
`direction`	`long` | `short` | `None`	
`qty`	`float | None`	
`order_type`	`market` | `limit` | `stop` | `None`	Stop-limit may appear as composite handling elsewhere
`limit` / `stop`	`float | None`	Prices when relevant
`oca_name`	`str | None`	OCA group
`comment`	`str | None`	Includes system comments (`risk_blocked`, `fill:…`)
`bar_index`	`int`	From context at emit
`bar_time`	`int`	From context
`ohlc`	4-tuple → list in JSON	Bar snapshot
`script_id`	`str`	Host-stamped (source hash)
`run_id`	`str`	Host-stamped run uuid

Frozen dataclass → `to_dict()` always includes keys (JSON nulls for unspecified fields) so parity diffs are stable.

Emission sources

1. **Explicit builtins** — `strategy.entry`, `exit`, `close`, `close_all`, `cancel`, `cancel_all`, `order` push events as they execute.
2. **Broker fills** — `process_pending_orders` emits fill-related `order` / `entry` / `cancel` (OCA) events.
3. **Risk gates** — blocked entries may emit with `comment="risk_blocked"`.

Host collection pattern

```
evaluator.reset_events()
# ... process_pending_orders + visit(tree)
for ev in evaluator._strategy_state.drain_events():
    d = ev.to_dict()
    d["script_id"] = script_id
    d["run_id"] = run_id
    all_events.append(d)
```

`Runtime.run` implements this and returns events in the response envelope.

Compile path

`CompileStrategyBroker` accumulates plain dict events (`__events` in the return mapping) with the same field names for kinds it supports. Exit/trail/OCA/risk halt comments (`risk_blocked`) are on both hosts for the 0.3.4+ surface; remaining metric accessors still trail the interpreter.

Internals

Path	Role
`src/pynescript/ast/evaluator/events.py`	`StrategyEvent`
`src/pynescript/ast/evaluator/builtins/strategy.py`	Emit sites + `drain_events`
`src/pynescript/compiler/strategy_broker.py`	Compile-mode `_emit`
`tests/test_strategy_events.py`	Unit behavior
`tests/test_parity.py` + `tests/fixtures/parity/`	Cross-port oracle
`pynets/src/runtime/interpret.ts` `StrategyEvent`	TS library twin

Parity corpus scripts (`strategy_01_entry_long.pine` , ...) regenerate expected JSON via:

```
python tests/fixtures/parity/generate_fixtures.py
```

Tests strip `script_id` / `run_id` before compare.

Invariants & edge cases

1. **Immutability.** Do not mutate events after construction; drain copies and clears the buffer.
2. **Per-bar reset.** `reset_events` / drain prevents cross-bar leakage in tests that share evaluators.
3. **OHLC list vs tuple.** `to_dict` converts to list for JSON round-trips.
4. **Kind vocabulary is closed.** New kinds require TS + fixtures + this doc.
5. **Not alerts.** `alert()` / `alertcondition()` use a separate alerts channel—not `StrategyEvent` .

Worked examples

Expected single entry

Script enters long on a fixed bar; fixture JSON lists one event:

```
{
  "kind": "entry",
  "id": "L",
  "direction": "long",
  "qty": 1.0,
  "order_type": "market",
  "bar_index": 5
}
```

(plus nullables and ohlc—see real fixtures for full keys).

Cancel after OCA

Fill of one OCA sibling yields `cancel` events for others with `comment` reflecting `oca_cancel` / `oca_reduce` .

Failure modes

Symptom	Cause
Empty events with live strategy	Forgot drain; or strategy never called
Parity flake on ids	Comparing `script_id` / `run_id`
TS/Python mismatch	Field rename without dual update
Duplicate events	Not clearing buffer; double `process_pending_orders`

See also

- Strategy builtins
- Strategy broker
- PyneTS
- pyne-worker
- HOOX live trading
- HOOX docs

37 · RUNTIME

ALERTS

alert() / alertcondition() engine, host export, and L2 HTTP webhooks (Pro API + pyne-worker).

ALERTS

Abstract

Pine `alert()` and `alertcondition()` fire into a **host side-channel**, not strategy events. The evaluator records structured firings with TradingView-style frequency rules. Hosts (Pro API and **pyne-worker**) export them on evaluate responses and can **POST** them to HTTP webhooks (roadmap **L2**).

Conceptual model

Layer	Module
Engine	`src/pynescript/ast/evaluator/builtins/alerts.py` (`export_alerts` / `export_alerts_from_evaluator`)
Package host	`src/pynescript/runtime/host.py` (`clear_alerts` per run, `alerts` on the envelope)
Pro API webhook	`backend/alert_forwarder.py` (`backend/runtime.py` is only a Runtime re-export)
Edge host	sibling pyne-worker `alert_engine.py` + `alert_forwarder.py`

Pine surface

`alert(message, freq)`

- Default `freq`: `alert.freq_once_per_bar` (canonical token `once_per_bar`).
- Constants (also bare strings): `alert.freq_once_per_bar`, `alert.freq_once_per_bar_close`, `alert.freq_all`.
- **once_per_bar** — first fire per bar for the same (source, title, message, freq) key.
- **once_per_bar_close** — only when the host marks the bar confirmed (`barstate.isconfirmed` / `islast`; hosts without barstate treat bars as confirmed).
- **all** — every call.

`alertcondition(condition, title, message)`

- Records each evaluation in `alert_conditions` (debug / UI).
- When `condition` is true, also **fires** a host alert with `source: "alertcondition"` and once-per-bar semantics.

Event shape (`to_dict` / API)

```
{
  "message": "cross up",
  "freq": "once_per_bar",
  "bar_index": 42,
  "time": 1700000000000,
  "title": "optional",
  "source": "alert"
}
```

`source` is `"alert"` or `"alertcondition"`. Hosts stamp `script_id` / `run_id` (and edge cron may add `symbol` / `timeframe`).

Host export

Pro API (`POST /run`)

- Interpret path: full history of firings in `alerts` (and optional `alert_conditions`).
- Compile path: `alerts: []` today (alert side effects are interpret-only).
- See POST /run.

pyne-worker

- `POST /run` returns the same `alerts` array.
- Cron filters to the **last closed bar** before webhook delivery so historical bars do not re-fire every minute.
- Health: `features.alerts`, `features.alert_webhooks`.

L2 webhooks

Both hosts POST JSON to an HTTPS endpoint.

Host	Default URL	Per-request / per-job
Pro API	env `ALERT_WEBHOOK_URL`	body `webhook_url`
pyne-worker	secret/var `ALERT_WEBHOOK_URL`	script/job `webhook_url`

Pro API request flags

Field	Default	Meaning
`webhook_url`	`""`	Override destination
`forward_alerts`	`true`	Skip POST when false
`alert_last_bar`	`true`	Only firings on the last OHLCV bar
`alert_batch`	`true`	One batch body vs one POST per alert

Optional timeout: env `ALERT_WEBHOOK_TIMEOUT` (seconds, default 10).

Batch body

```
{
  "type": "pine_alert_batch",
  "source": "pyne-pro-api",
  "count": 1,
  "content": "cross up",
  "alerts": [
    {
      "type": "pine_alert",
      "source": "pyne-pro-api",
      "message": "cross up",
      "freq": "once_per_bar",
      "alert_source": "alert",
      "bar_index": 42,
      "time": 1700000000000
    }
  ]
}
```

Edge worker uses `"source": "pyne-worker"`. Discord-friendly `content` is set when a message is present.

Response meta

Successful `/run` may include:

```
"alert_forward": {
  "forwarded": 1,
  "failed": 0,
  "filter": "last_bar",
  "url": "https://hooks.example.com/pine",
  "batch": true,
  "count": 1
}
```

Webhook failures never fail the evaluate status; inspect `alert_forward` / `alert_forward_error`.

Worked example

```
curl -sS -X POST http://127.0.0.1:5002/run \
  -H 'Content-Type: application/json' \
  -d '{
    "script": "//@version=6\nindicator(\"a\")\nif bar_index == last_bar_index\n    alert(\"fire\")
    "mode": "interpret",
    "webhook_url": "https://hooks.example.com/pine",
    "data": [
      {"time": 1, "open": 1, "high": 2, "low": 0.5, "close": 1.5, "volume": 10},
      {"time": 2, "open": 1.5, "high": 2.5, "low": 1.0, "close": 2.0, "volume": 12}
    ]
  }'
```

Invariants

1. **Not strategy events** — trade mesh uses `events[]`; alerts are independent.
2. **Per-run clear** — hosts call `clear_alerts()` so runs do not leak firings.
3. **Last-bar default for webhooks** — full-history still available in the `alerts` response field.
4. **Compile path** — prefer `mode: "interpret"` (or auto fallback) when you need alerts.

See also

- Strategy events
- POST /run
- Pro API usage
- Runtime bridge
- Roadmap (L2 ✓)

COMPILER OVERVIEW

Source-to-source compilation: CompilerVisitor, numeric vs object mode, run(time=...), and engine API.

COMPILER OVERVIEW

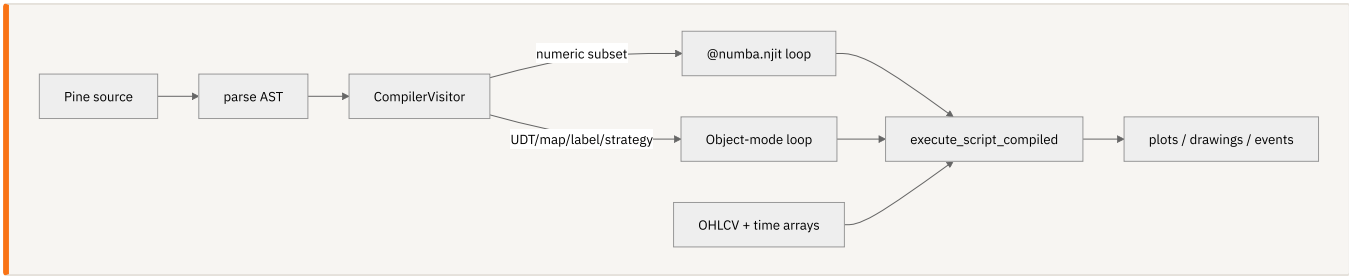
Abstract

The compile path lowers Pine's ASDL AST to **executable Python** that walks bars with an explicit index (`__bar_idx`) over contiguous `numpy` arrays—avoiding per-node visitor overhead. Pure numeric scripts wrap the loop in `@numba.njit`; scripts that touch UDTs, maps, drawings, or strategy select **object mode** (a tight Python/numpy loop, no AST walk). **Numba is required only for numeric mode**; object mode runs with numpy alone. **once** (0.4.4) lowers to a persistent `__once_fired_N` flag; UDF-scoped `once` forces object mode.

The public façade is `pynescript.compiler.engine` (`transpile`, `compile_script`, `run_script`), also reachable as `Runtime.run(..., mode="compile")` or `mode="auto"`.

This is an MVP with growing surface—not a full substitute for the interpreter on every script (notably multi-asset `request.*` and full strategy analytics). **0.3.16–0.3.17** recovered a large object-mode residual class: UDF locals, nopython `None`/unicode helpers, matrix handles, UDT returns, statement `switch` assign, drawing/`chart.point` copy, `round()` extras, import stubs as `na`, color type dispatch, and nested UDT method fields. Official TV builtins are **0 MISMATCH** vs interpret (**P1p** closed on that set); optional set0x community corpus is unmeasured, not a missing language surface.

Conceptual model



Generated entry point is always:

```
execute_script_compiled(open_arr, high_arr, low_arr, close_arr, vol_arr, time_arr)
```

Host packing lives in `engine.CompiledScript.run` (and `pynescript.runtime.Runtime._run_compiled`).

Interface surface

```
from pynescript.compiler.engine import transpile, compile_script, run_script, has_numba

src = transpile(pine_text)           # generated Python string
cs = compile_script(pine_text)       # CompiledScript
result = cs.run(open, high, low, close, volume, time=None)
# result: dict of plot title → float64 array, optional __drawings, __events, …
```

API	Role
`transpile`	Parse + visit → source string (inspect/debug)
`compile_script`	`exec` generated code, warm-up call, return `CompiledScript`
`run_script`	One-shot compile+run (prefer cache `CompiledScript` for batches)
`has_numba`	Whether numeric (njit) mode can load
`prewarm_numba_builtins` / `prewarm_scripts`	Host cold-start (H2 warm path)
`compile_cache_stats` / `compile_deploy_config`	Cache + deploy diagnostics
`clear_compile_cache`	In-process LRU only
`clear_disk_compile_cache`	Disk IR / index under cache dir
`clear_numba_function_caches`	Purge Numba `.nbi` / `.nbc` under known dirs

`CompiledScript` fields: `source`, `generated_code`, `execute`, `plot_titles`, `plot_kinds` (per-plot `plot` / `hline` / `fill` tags used to synthesize numeric-mode `__drawings`), `object_mode`, optional `nopython_fallback_reason`.

`CompiledScript.run` signature

```
def run(
    self,
    open_, high, low, close,
    volume=None,   # default: ones
    time=None,     # bar-open Unix ms; optional 6th series
) -> dict[str, Any]:
```

Argument	Behavior
OHLCV	Coerced to contiguous `float64`; lengths must match (`ValueError` otherwise)
`volume=None`	Filled with ones
`time=None`	Synthetic `bar_index * 60_000` ms (unit-test / pure-compile default)
`time=arr`	Real bar-open timestamps; length must match OHLCV

`Runtime._run_compiled` packs OHLCV dicts and **passes real bar times** so calendar/`time` / `timestamp` / `time[n]` stay aligned with interpret. Pure `cs.run(o,h,l,c)` without `time=` is fine for pure-price scripts and fixtures.

When `time` is omitted, the engine synthesizes `np.arange(n) * 60000.0` (1-minute bar opens from epoch). Hosts that care about `year` / `month` / `time[n]` must pass bar-open Unix ms.

Warm compile (product defaults)

Pro API `/run` defaults to **`mode=auto`** (compile when eligible, interpret on fallback with `compile_fallback_reason`). Disk IR cache is **on** (`PYNE_COMPILE_DISK_CACHE=1`); Docker sets `PYNE_COMPILE_CACHE_DIR=/data/compile-cache`. Operators call `POST /compile/prewarm` or `pynescript prewarm` so first interactive latency skips Numba cold JIT. Design notes live in repo `docs/COMPILER_PLAN.md` (not a product page).

`Runtime._compile_eligible` (used by `mode=auto`) **skips compile** when the source contains a top-level `import` or any `request.` token — those stay interpret even though the emitter can lower same-symbol `request.security`. Explicit `mode=compile` still attempts the visitor.

Mode selection

`CompilerVisitor` sets `object_mode = True` when it sees:

- User-defined types / enums
- Map / array / matrix handles that cannot stay float64
- Drawing constructors (`label.new`, `line.new`, ...) and assigned `hline` handles
- Strategy usage (object mode + `CompileStrategyBroker`)
- String/color series, library imports, or other non-numeric constructs

Statement-form `hline(price)` / `fill(...)` with a proven-numeric price **stay nopython**: they emit a constant (or all-`nan`) series plus `plot_kinds` metadata; the engine synthesizes `__drawings` after the JIT loop. Mixed scripts that later flip object mode replay those events into the body.

Otherwise **numeric mode** is preferred. Numeric mode **requires Numba** at load/warm-up; missing Numba raises `CompileNumbaRequiredError`. Object mode does **not** need Numba (still star-imports `numba_builtins` for `safe_*` / non-jitted helpers).

On numeric warm-up **TypingError** / nopython failure, the engine re-emits with `force_object_mode=True` and records `nopython_fallback_reason` —so `mode="compile"` can still run without hard-failing on partial numeric coverage.

Numba is **not** a hard requirement for all compile. Object-mode scripts (strategy, drawings, UDT, maps) execute without it. Only a pure-numeric emit that never flips to object mode needs Numba installed.

What numeric mode lowers today

Series assigns, arithmetic/logic, history (`close[n]`), `if` / `for` / `while`, selected `ta.*` (and bare TA aliases), scalar math helpers, `plot`, `input.*` defaults, `var` / `varip` carry—executed under `@numba.njit(cache=False)` on the generated entry (`cache=False` because code is `exec`'d from a string without a stable disk locator). Disk-cached IR modules may rewrite to `cache=True` for cross-process reuse of machine code.

Object mode extras

- UDT instances as field dicts
- Maps as Python dicts
- `__drawings` list of structured events
- Optional `__strategy` broker: pending fills, position, `__events`, equity snapshots

Compile-path limits (notable)

Construct	Compile behavior
`request.security` / bare `security` / `request.security_lower_tf` / `request.seed`	**Same-symbol, simple OHLCV expression only** (chart identity + bare `close` / `high` /... style). Foreign symbol or complex expression → `np.nan` (no inventing chart close as foreign data). Other `request.*` → object-mode `np.nan` stub
Bare builtins vs user series	User-defined series arrays win in `visit_Name` : `ad = ta.cum(...)` / `tr = …` shadow bare `ad` / `tr` formulas so `plot(ad)` is not silently re-bound to the builtin
`hline(...)`	Statement-form + numeric price → **nopython** constant series (`hline` , `hline_2` , ...) + synthesized `__drawings` . Assigned / non-numeric handles → object mode
`fill(..., title=…)`	Statement-form → **nopython** all- `nan` series key (band color in `plot_meta` / drawings). Expression-form / mixed object scripts emit `__drawings` in-loop

Compile-path `request.security` never invents foreign OHLCV from chart close. Foreign tickers and complex expressions (UDFs, `ta.*` inside the third arg, multi-value constructs beyond simple chart series) lower to `np.nan` . Use interpret + a real `data_feed` / `data_provider` for multi-asset work.

Internals

Path	Role
`src/pynescript/compiler/compiler.py`	`CompilerVisitor` , emit numeric/object, security/name/fill/hline lowering
`src/pynescript/compiler/numba_builtins.py`	JIT kernels + object-mode `safe_*`
`src/pynescript/compiler/strategy_broker.py`	Compile broker
`src/pynescript/compiler/engine.py`	Façade, caches, `CompiledScript.run` , Numba cache recovery
`src/pynescript/runtime/host.py`	Package SoT `Runtime._run_compiled` envelope; packs OHLCV + `time=`
`backend/runtime.py`	Compat re-export of `pynescript.runtime.host`
`scripts/compare_interp_compile.py`	Interpret↔compile series parity harness
`tests/test_compiler_numba.py` , `test_compiler_objects.py` , `test_compiler_strategy.py`	Coverage
`docs/COMPILER_PLAN.md`	Design history and remaining work (repo, not hosted)

Data layout

Unlike interpreter `PineSeries` deques:

```
open_arr, high_arr, low_arr, close_arr, vol_arr, time_arr
user_arr = np.full(n_bars, np.nan)                # or dtype=object for UDT
plot_i   = np.full(n_bars, np.nan)
for __bar_idx in range(n_bars):
    user_arr[__bar_idx] = …
    plot_i[__bar_idx] = …
```

- Bare `time` / `time_close` / `last_bar_time` / calendar extractors read `time_arr` (synthetic when omitted).
- `var` lowers to "init only when still na / first write" patterns so carry matches Pine without declaration sets.
- Numeric mode returns a **tuple** of plot arrays (host maps titles); object mode returns a **dict** (plots + `__drawings` / strategy extras).

Caches and corrupt-Numba recovery

Three layers:

1. **In-process source LRU** (sha256 of raw and/or sanitized source) + secondary IR cache keyed by generated-code hash.
2. **Disk IR** (default on): modules under `PYNE_COMPILE_CACHE_DIR` or `$XDG_CACHE_HOME/pynescript/compile`. The source→IR index JSON carries `"v": 9` (`engine._DISK_META_VERSION`). Bump that integer when generated IR semantics change so stale modules are ignored (source hash alone is stable across emitter fixes).
3. **Numba function cache** (`.nbi` / `.nbc`) next to `numba_builtins` and under disk-IR `__pycache__/`.

Truncated/corrupt Numba pickle files raise `EOFError` / `pickle.UnpicklingError` on load. The engine wraps execute/warm paths with **purge + single retry** via `clear_numba_function_caches` (does not fail the script for a bad cache alone). Manual clean slate:

```
from pynescript.compiler import (
    clear_compile_cache,
    clear_disk_compile_cache,
    clear_numba_function_caches,
)
clear_compile_cache()
clear_disk_compile_cache()
clear_numba_function_caches()
# shell: rm -rf ~/.cache/pynescript/compile \
#   src/pynescript/compiler/__pycache__/numba_builtins*.nb*
```

Clear disk + Numba caches after compiler emitter or kernel edits if results look stale.

Invariants & edge cases

1. **Same AST front-end.** No second parser—compile bugs are lowering bugs.
2. **OHLCV (+ time when provided) length equality** enforced in `CompiledScript.run`.
3. **Warm-up** ignores exceptions on a dummy 16-bar series (JIT or first-run); non-nopython failures may surface on the first real run.
4. **Result normalization** converts Numba typed maps to plain dicts; `__drawings` / `__events` stay Python lists.
5. **User series shadow bare builtins** in `visit_Name` (`ad`, `tr`, `n`, …) once `*_arr` is allocated.
6. **`request.security` is not multi-asset on compile.** Only chart-symbol simple OHLCV passthrough; everything else is `na`.

7. **Interpret still owns the full surface.** Exotic builtins, real multi-symbol feeds, and rich strategy metrics remain interpret-first until lowered.

Worked example

Pine:

```
//@version=6
indicator("c")
s = ta.sma(close, 14)
plot(s, "sma")
```

Generated shape (illustrative; numeric mode returns a tuple of series):

```
@numba.njit(cache=False)
def execute_script_compiled(open_arr, high_arr, low_arr, close_arr, vol_arr, time_arr):
    n_bars = len(close_arr)
    s_arr = np.full(n_bars, np.nan)
    plot_0 = np.full(n_bars, np.nan)
    for __bar_idx in range(n_bars):
        s_arr[__bar_idx] = numba_sma(close_arr, 14, __bar_idx)
        plot_0[__bar_idx] = s_arr[__bar_idx]
    return (plot_0,)
# Host packs titles → {'sma': plot_0}
```

Interpret↔compile parity

Full contract (tolerances, buckets, intentional diffs): Interpret ↔ compile parity.

```
# From repo root — nan-aware allclose on common series keys
python scripts/compare_interp_compile.py --bars 1000 --limit 50
python scripts/compare_interp_compile.py --glob 'ta_*.pine' --bars 200
python scripts/compare_interp_compile.py --files tests/fixtures/parity/pine/strategy_01_entry_long
python scripts/compare_interp_compile.py --ignore-hline-keys --ignore-fill-keys --strict-keys
```

Writes `.cache/interp_compile_parity.json`. First-party `hline` / `fill` / `bgcolor` / `plotshape` / `barcolor` / `plotarrow` / `plotbar` / `plotcandle` keys match interpret ↔ compile (0.3.12); `drawings` is geometry-only. Ignore flags are for leftover corpus noise.

Failure modes

Symptom	Cause
`numba is required for numeric compile mode` (`CompileNumbaRequiredError`)	Install numba **or** force/land in object mode (drawings/UDT/strategy already do)
Empty generated code	Visitor returned blank—unsupported top-level shape (`CompileEmitError`)
Missing `execute_script_compiled`	Emit bug or failed `exec` (`CompileLoadError`)
`EOFError` / `UnpicklingError` during JIT load	Corrupt Numba `.nb*` —engine purges and retries; if stuck, call `clear_numba_function_caches`
Silent semantic drift	Kernel seed differences—run `scripts/compare_interp_compile.py`
All- `nan` from `request.security`	Foreign symbol or non-simple expression on compile path (by design)
Wrong values for `ad` / `tr` after assignment	Fixed by user-series shadowing; if stale, clear compile/Numba caches
Strategy metrics missing	Use interpret or inspect `__events` / broker fields only
Calendar/ `time` off by hours	Host omitted `time=` ; synthetic 1m bar opens used

Remaining work (summary)

Expand Numba `ta.*` surface, richer drawings, nested UDT methods, broader `request.*` (still interpret-first for multi-asset; `mode=auto` already skips any `request.` source), and grow systematic interpret↔compile parity coverage per lowered construct. In-process + disk IR cache (`_DISK_META_VERSION = 9`) and product prewarm are landed. See repo `docs/COMPILER_PLAN.md` for the living checklist.

See also

- Numba path
- Strategy broker
- Interpret ↔ compile parity
- Runtime hub
- request.* / input.* (interpret)
- Numerical validation

NUMBA PATH

JIT bar loops, numba_builtins kernels, NA-as-NaN, and performance characteristics.

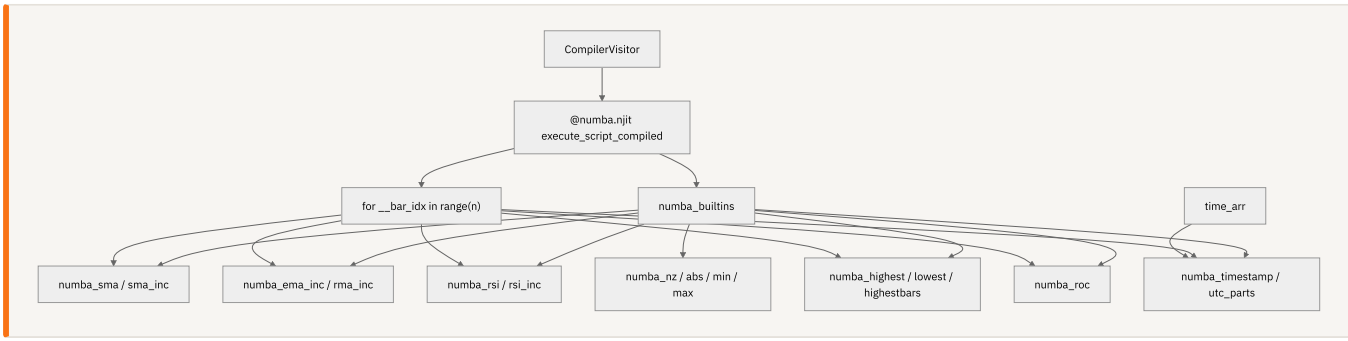
NUMBA PATH

Abstract

Numeric compile mode emits a single `@numba.njit` function whose body is the script's bar loop. Pine builtins that cannot cross the JIT boundary are replaced by **hand-written kernels** in `numba_builtins.py` that take full arrays plus the current index (or a small float state vector for `*_inc` forms). The result is large speedups on multi-thousand-bar series after a one-time JIT warm-up—at the cost of a restricted language subset and `np.nan` as the missing-value sentinel.

Kernel contracts below match interpret `ta.*` / calendar helpers where parity was fixed; intentional deviations should be documented on the kernel docstring and here.

Conceptual model



Interface surface

Kernels are defined with `@numba.njit(cache=True)` in `numba_builtins.py`. Generated entry points use `@numba.njit(cache=False)` in the in-memory IR string; when the engine writes a **disk IR** module it rewrites that decorator to `cache=True` so Numba can store machine code next to a real file path (see Cache).

Generated code imports `from pynescript.compiler.numba_builtins import *` so call sites are bare names. Prefer `*_inc` when the visitor allocates fixed state (`__ema0_st` , ...).

Kernel table (interpret-parity highlights)

Kernel	Semantics
`numba_sma(arr, period, i)`	Mean of last `period` bars ending at `i`; `nan` if warm-up or any window `nan`
`numba_ema` / `numba_ema_inc`	SMA seed, then EMA (`α = 2/(period+1)`). **Inc seed is NaN-safe**: while the accumulator is still `nan` and `j >= period-1`, try SMA over `arr[j-period+1 : j+1]` only when every sample is finite. Nested EMA / DEMA no longer stuck all-NaN after leading NaNs
`numba_rma_inc(arr, period, i, st)`	Wilder RMA with the **same NaN-window-safe SMA seed** as `numba_ema_inc`. After seed, NaN inputs hold the previous RMA. Required for `rma(tr)` / expanded ADX so compile is not all-NaN or ~0 after warm-up
`numba_rsi` / `numba_rsi_inc`	**Wilder RSI**, not rolling simple-window RSI. SMA of first `period` deltas (bars `1..period`), then RMA of gain/loss (`α = 1/period`). First valid bar is **`i == period`** (`period` deltas need `period+1` prices). `avg_loss == 0` → `100.0`
`numba_highest` / `numba_lowest`	Window extrema; full window required (`i >= period-1`); NaN samples skipped
`numba_highestbars` / `numba_lowestbars` (+ `_inc`)	TV **negative** bars-back offsets: `0` if current is extreme, `-1` one bar ago, ..., `-(length-1)`. Returns **`-1.0`** when window not full, invalid length, or all-NaN. **Oldest** extreme wins ties (strict `>` / `<`)
`numba_roc(arr, length, i)`	`100 * (src - src[length]) / src[length]` at bar `i`; `nan` if `i < length`, baseline 0/NaN, or src NaN
`numba_timestamp(y, m, d, h=0, mi=0, s=0)`	UTC epoch **ms** from calendar components; month/day overflow normalized (e.g. `month=0` → prior Dec) for TTM-style windows
`numba_utc_parts(ms)`	`(year, month, dayofmonth, hour, minute, second, dayofweek)` floats; Pine `dayofweek` is `1=Sun … 7=Sat`
`numba_nz(val, replacement)`	Replace `nan`
`numba_abs` / `numba_min` / `numba_max`	Scalar helpers

Full inventory lives in `src/pynescript/compiler/numba_builtins.py` (njit batch + `*_inc` + pure-Python object-mode helpers).

Calendar / `time_arr`

Bar open time is the host `time_arr` series (synthetic when the runner omits timestamps).

Pine form	Lowering
bare `year` / `month` / `dayofmonth` / `hour` / `minute` / `second` / `dayofweek`	`numba_utc_parts(time_arr[__bar_idx])[idx]`
`year(t)` ... (optional time arg)	`numba_utc_parts(src)[idx]` with `src` defaulting to `time_arr[__bar_idx]`
`timestamp(y, m, d[, h, mi, s])`	`numba_timestamp(...)` (leading timezone string args dropped → UTC stub)
`time` / `time_close` (approx)	`time_arr[__bar_idx]` / `+ 59999` ms

History access

`close[n]` lowers to array indexing at `__bar_idx - n` with bounds → `nan` (parallel to interpreter `None`, different sentinel).

Control flow

`if` / `for` / `while` / `once` emit Python control flow **inside** the njit function—supported Numba subset only (no heap objects, no Python lists of mixed types). `once` lowers to a persistent `__once_fired_N` bool set when the body runs (compile is historical: `barstate.isconfirmed` is always true). A `once` inside a UDF forces object mode so the flag is module-level.

Cache

Three layers interact:

Layer	Location / key	Role
Source LRU	in-process, sha256 of source	`CompiledScript` reuse
IR LRU	in-process, sha256 of generated Python	share warm callable across comment-only variants
Disk IR	`PYNE_COMPILE_CACHE_DIR` or `$XDG_CACHE_HOME/pynescript/compile`	persist generated modules; disable with `PYNE_COMPILE_DISK_CACHE=0`. Index JSON `"v"` must equal `engine._DISK_META_VERSION` (**9**)

Numba function cache (`.nbi` / `.nbc`):

- Hand-written kernels: `@njit(cache=True)` → artifacts under `…/pynescript/compiler/__pycache__/`.
- Generated entry: IR is emitted with `cache=False` (safe for `exec` from `<string>`). Disk write rewrites `@numba.njit(cache=False)` → `cache=True` so Numba's file locator can cache machine code under the disk IR `__pycache__/`.
- Truncated/corrupt pickle loads raise `EOFError` / `UnpicklingError`. The engine purges known `.nbi` / `.nbc` via `clear_numba_function_caches()` and recompiles once instead of failing the script.

```
from pynescript.compiler import clear_numba_function_caches, clear_disk_compile_cache
clear_numba_function_caches()
clear_disk_compile_cache()
```

Internals

Path	Role
`src/pynescript/compiler/numba_builtins.py`	Kernels (`cache=True`)
`src/pynescript/compiler/compiler.py`	`_emit_numeric_mode`, call lowering, `time_arr` calendar
`src/pynescript/compiler/engine.py`	Disk IR rewrite, Numba cache recovery, prewarm
`tests/test_compiler_numba.py`	Correctness / parity

Expanding coverage is primarily **new kernels + visitor call mapping**, not changes to the bar-loop skeleton.

Invariants & edge cases

1. **No Python objects in numeric mode.** Strings, UDTs, maps, drawings force object mode before njit is applied.
2. **Warm-up cost.** First `compile_script` invokes a dummy run; production servers should cache `CompiledScript` and prewarm (`PYNE_COMPILE_PREWARM` default **on**, `POST /compile/prewarm`).
3. **EMA/RMA seed** uses sliding all-finite SMA windows—not "seed once at bar `period-1` only if that fixed slice is clean." Leading NaN on `tr` / nested EMA sources must eventually produce values.
4. **RSI is Wilder**, first valid at `i == period`. Do not compare against a simple N-bar average-gain RSI.
5. **highestbars/lowestbars** return negative offsets and `-1` on incomplete windows; ties keep the oldest bar.
6. **Disk entry cache** requires the rewrite to a real file path; pure `exec` of `cache=False` IR does not populate Numba's on-disk function cache for the entry function.
7. **Anaconda static libpython.** Packaging notes in AGENTS/build docs apply when shipping JIT-heavy binaries.

Worked examples

Benchmark mental model

```
interpret:  O(bars × ast_nodes × python_dispatch)
numeric:    O(bars × lowered_ops) in machine code after JIT
```

For simple SMA scripts, internal notes cite order-of-magnitude speedups versus list-based interpretation on long series (see repo `docs/COMPILER_PLAN.md` qualitative claims).

Using from Pro API

```
Runtime(...).run(source, ohlcv, mode="compile")
```

Converts bar dicts → arrays (including `time=` bar-open ms), compiles, reshapes plots into the standard envelope, flags `object_mode` when applicable. `generated_code` is omitted unless `PYNESCRIPT_RETURN_GENERATED_CODE=1`.

Failure modes

Symptom	Cause
TypingError from Numba	Unsupported construct leaked into numeric emit
All-`nan` plots	Period longer than series; history index bugs; or (fixed) NaN-poisoned EMA/RMA seed on nested sources
Nested DEMA / `rma(tr)` all-NaN	Pre-fix seed; current kernels sliding-window all-finite SMA seed
RSI diverges vs interpret	Expect Wilder seed/RMA, not rolling window RSI
`highestbars` sign / magnitude off	TV negative offsets; incomplete window → `-1`
`EOFError` / UnpicklingError at load	Corrupt `.nbi` / `.nbc` — engine purges and retries; manual `clear_numba_function_caches()`
ImportError numba	Environment missing dependency
Object mode unexpectedly	Visitor detected drawing/UDT/map—inspect `visitor.object_mode`

Performance notes

- Prefer **one compile, many runs** with different OHLCV.
- Keep scripts in the numeric subset for realtime ticks.
- Host prewarm of `numba_builtins` + common scripts shifts cold JIT off the first user request.
- Object mode still wins over AST walking for UDT-heavy scripts but will not match peak njit throughput.

See also

- Compiler overview
- Technical builtins (interpret)
- Series & history

COMPILE STRATEGY BROKER

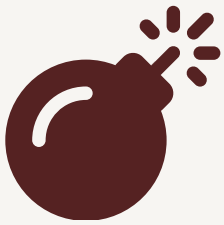
CompileStrategyBroker: pending orders, OHLC fills, OCA, commission, and event export.

COMPILE STRATEGY BROKER

Abstract

When `CompilerVisitor` detects strategy usage, object-mode emission instantiates **`CompileStrategyBroker`**—a lightweight broker aligned with the interpreter's fill-before-script cadence, but designed for generated loops (no AST, no full `StrategyState` metric surface). It supports market entry/close and pending limit/stop/stop-limit orders with per-bar OHLC triggers, OCA cancel/reduce, commission, and slippage.

Conceptual model



Syntax error in text

mermaid version 11.16.0

Interface surface

Construction (from `strategy()` kwargs when lowered)

```
CompileStrategyBroker(
    initial_capital=100_000.0,
    commission_value=0.0,
    commission_type="percent",  # percent | cash_per_order | cash_per_contract
    slippage_ticks=0,
    mintick=0.01,
    pyramiding=0,
    default_qty_type="fixed",
    default_qty_value=1.0,
    avg_price_model="stock",  # stock | futures | inverse (pynescript extension)
    leverage=1.0,             # futures UI: buying power / margin divisor
)
```

`avg_price_model` is lowered from `strategy(..., avg_price_model=...)`. The compile broker is a **single net lot**: partial reduce already keeps `position_avg_price` sticky (futures-like). Multi-leg stock reweight on partial close is interpret-first until the compile path grows an open-trade list.

`leverage` scales percent/cash default qty (`margin × leverage / price`) and free `cash` (equity – notional/leverage).

Bar context

```
# Preferred generated hot path (one call; skips pending walk when empty)
broker.begin_bar(bar_index, open_, high, low, close, bar_time=…)
# Equivalent two-step still exists:
broker.set_bar(bar_index, bar_time, mark, open_=…, high=…, low=…, close=…)
broker.process_pending_orders(open_=…, high=…, low=…, close=…)
```

Orders

Method	Role
`entry(...)`	Market or pending entry; reverse flattens opposite
`close` / `close_all`	Reduce/flat with optional price
Pending book	`PendingOrder` with type, direction, qty, limit/stop, OCA, `max_fill_per_bar`, `is_entry`

Fill logic (`_trigger_price`)

Type	Long trigger	Short trigger	Fill price idea
market	—	—	close (immediate path)
limit	`low <= limit`	`high >= limit`	gap-aware vs open
stop	`high >= stop`	`low <= stop`	gap-aware vs open
stop-limit	stop traded and limit available	symmetric	limit price

Partial fills: `max_fill_per_bar` caps quantity per bar; residual stays pending.

OCA

After a fill, siblings in the same `oca_name`:

- **`cancel`** — remove sibling, emit `cancel`
- **`reduce`** — decrease sibling qty by fill size
- **`none`** — no interaction

Economics

- **Slippage:** `± slippage_ticks * mintick` by direction on fills.
- **Commission:** percent of notional, cash per order, or cash per contract.
- **Position:** signed `position_size`, `position_avg_price`, `netprofit`, `equity` helpers for return payload.

Events

`_emit` appends dicts with keys compatible with strategy event serialization (`kind`, `id`, `direction`, `qty`, `order_type`, `limit`, `stop`, `oca_name`, `comment`, `bar_index`, `bar_time`, `ohlc`). Object-mode return includes `'__events': __strategy.to_events()` plus position/netprofit/equity snapshots.

Internals

Path	Role
`src/pynescript/compiler/strategy_broker.py`	Broker implementation
`src/pynescript/compiler/compiler.py`	Emits `__strategy` wiring in object mode
`src/pynescript/ast/evaluator/builtins/strategy.py`	Interpreter counterpart (richer)
`tests/test_compiler_strategy.py`	Compile strategy coverage

NA helpers treat `None`, blank/`na` strings, and float NaN as missing prices when classifying order types.

Invariants & edge cases

1. **Same order as interpreter:** `begin_bar` (or `set_bar` → `process_pending_orders`) → script body.
2. **Closed-trade list is partial.** The compile broker now keeps `closed_trade_records` and some `closedtrades_*` accessors (profit, comments, max drawdown/runup). Prefer interpret for the full `strategy.closedtrades.*` / `opentrades.*` metric surface.
3. **Direction normalization** accepts `strategy.long`, `long`, `1`, `buy`, etc.
4. **Reverse on opposite entry** via `close_all(comment="reverse")` then open.
5. **Event dicts are mutable lists**—fine for run output; not frozen dataclasses.

Worked examples

Generated prologue (illustrative)

```
__strategy = CompileStrategyBroker(initial_capital=100000.0)
for __bar_idx in range(n_bars):
    __strategy.begin_bar(__bar_idx,
        float(open_arr[__bar_idx]), float(high_arr[__bar_idx]),
        float(low_arr[__bar_idx]), float(close_arr[__bar_idx]),
        bar_time=int(time_arr[__bar_idx]))
    # lowered strategy.entry / close calls
return {..., '__events': __strategy.to_events(),
        '__position_size': __strategy.position_size,
        '__netprofit': __strategy.netprofit,
        '__equity': __strategy.equity}
```

Limit buy pending

```
strategy.entry("L", strategy.long, limit=low - syminfo.mintick)
```

Object-mode lowers to a pending limit; subsequent bars fill when `low` trades through.

Failure modes

Symptom	Cause
No fills	Triggers never met; or market path not invoked
OCA sibling remains	`oca_type` none / name mismatch
PnL vs interpret drift	Commission/slippage/mintick defaults differ
Missing events in API envelope	Host only maps plots—not `__events`

See also

- Strategy builtins (interpret)
- Events
- Compiler overview
- Numba path

41 · RUNTIME / COMPILER

INTERPRET ↔ COMPILE PARITY

Plot-series parity between Runtime interpret and compile modes: harness, tolerances, and known sentinel differences.

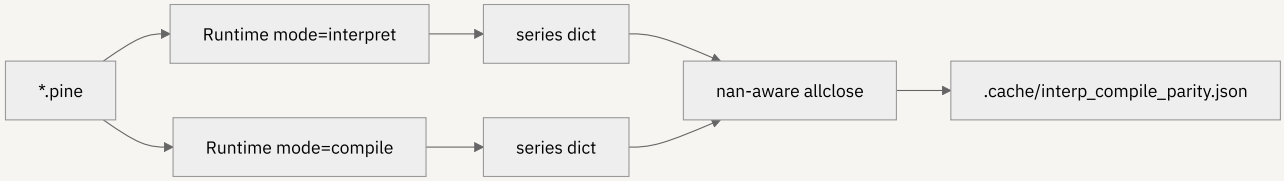
INTERPRET ↔ COMPILE PARITY

Abstract

PYNE ships two bar engines that should agree on **numeric plot series** for scripts in the compile surface: the AST interpreter and the generated Numba/object-mode loop. This page is the short contract for that agreement—how to measure it, what “equal” means, and where intentional differences live (`None` vs `nan`, host `time_arr`, structural keys).

Strategy **event** parity (Python vs TypeScript worker) is a separate oracle (`tests/test_parity.py` + fixtures). Here the focus is **plot values** on one host: `pynescript.runtime.Runtime`.

Conceptual model



Both modes receive the **same** synthetic (or host) OHLCV. Compile additionally builds **`time_arr`** from bar `time` fields so `time` / calendar expressions line up with the interpret `time` series.

Interface surface

Corpus harness

```
# From repo root — first-party fixtures by default
python scripts/compare_interp_compile.py --bars 1000 --limit 50
python scripts/compare_interp_compile.py --files tests/fixtures/parity/pine/strategy_01_entry_long
```

Flag / output	Role
Default inputs	`tests/fixtures/parity/pine/*.pine` (first-party)
Tolerances	`rtol=1e-5`, `atol=1e-6` (nan-aware)
Report	`.cache/interp_compile_parity.json`
Exit 0	No value/ `nan` mismatches on common series keys

Buckets: `OK`, `fill_background_only`, `both_error_same`, `expected_error`, `both_error`, `MISMATCH`, `interp_error`, `compile_error`, `structural_only`. Matched errors on both backends (`both_error_same`) and intentional demo failures (`expected_error`, e.g. auto-fib / pivot-depth) count as success unless `--strict-`

`errors`. First-party `hline` / `fill` / `bgcolor` / `plotshape` / `barcolor` / `plotarrow` / `plotbar` / `plotcandle` keys match interpret ↔ compile (0.3.12); `drawings` is geometry-only. `--ignore-hline-keys` / `--ignore-fill-keys` remain optional for leftover **corpus** key noise; `--strict-keys` fails on any one-sided key.

Focused unit tests

Test	Scope
`tests/test_dividend_yield_parity.py`	Host-series copy-on-assign + time history (interpret/compile)
`tests/test_compiler_numba.py`	Kernel-level numeric correctness
`tests/test_parity.py`	Strategy **event** fixtures (not plot allclose)

Product path

Pro API `/run` defaults to **`mode=auto`**: compile when `_compile_eligible` passes (no top-level `import`, no `request.` token), otherwise interpret with `compile_fallback_reason`. Parity harness always forces explicit `interpret` vs `compile` so fallbacks do not hide drift.

Official TradingView® builtins (0.6.2): interpret UDF series-parameter history (`s[1]` per call site) closed the remaining P1p plot MISMATCH tail on that set — **0 MISMATCH** vs compile. Optional set0x community corpus is unmeasured, not a missing language surface.

What must match

1. **Common plot keys** — float series after `None` ↔ `nan` normalization.
2. **Warm-up `na`** — leading missing samples align (interpret `None` / compile `nan`).
3. **`time` / bar-open ms** — when OHLCV includes `time`, or both use the synthetic `i * 60_000` fallback.
4. **Structured errors** — scripts that `runtime.error` (e.g. insufficient pivots) should fail on both with messages that normalize equal (`both_error_same`).

Intentional / soft differences

Topic	Policy
Sentinels	Interpret: `None`; Numba: `np.nan`. Compare with nan-aware equality, not `==`.
Hline / fill / bgcolor keys	May appear on one mode only; ignore with harness flags unless you need strict key sets.
Unresolved imports / stubs	Non-numeric plot cells serialize as `na` (null)—not string stubs.
Outside compile surface	UDT-heavy / full `request.*` / rich strategy analytics stay interpret-first; `auto` falls back.
Disk / Numba caches	Stale IR after kernel edits can fake mismatches—clear compile + Numba caches (see harness docstring).

Internals

Path	Role
`scripts/compare_interp_compile.py`	End-to-end series compare
`tests/test_interp_compile_parity.py`	Always-on smoke + harness flag tests
`src/pynescript/runtime/host.py`	`_run_compiled`, OHLCV + `time_arr` packing
`backend/runtime.py`	Compat re-export of the package host
`src/pynescript/compiler/engine.py`	Compile façade + result normalize
`src/pynescript/compiler/numba_builtins.py`	JIT kernels under test

Worked example

```
python scripts/compare_interp_compile.py \
  --files tests/fixtures/parity/pine/strategy_01_entry_long.pine \
  --bars 500
# → OK  (or MISMATCH with first differing bar/key in the JSON report)
```

Failure modes

Symptom	Likely cause
Systematic offset after warm-up	EMA/RSI seed or window clamp drift in a kernel
Only-compile empty plots	Script still object-mode / plot not lowered; or cache corruption
`time` diverges	Host omitted times on one path only; check `time_arr` packing
Structural-only keys	hline/fill/background naming — use ignore flags or align collectors

See also

- Compiler overview
- Numba path
- Series & history
- Runtime hub
- Numerical validation
- PyneTS parity — TypeScript vs Python plots

LANGUAGE SERVER

PYNE Language Server Protocol — diagnostics, completion, hover, formatting for .pyne / .pine and editor clients.

LANGUAGE SERVER

The PYNE **Language Server** is the editor-facing half of the evaluation stack: a pygls process that speaks LSP over STDIO (or TCP in tooling), parses Pine with the same ANTLR4 → ASDL pipeline as the runtime, and never requires a proprietary chart host. First-class file associations include `.pyne` (product extension) plus `.pine` / `.pinev5` / `.pinev6` / `.pinescript`.

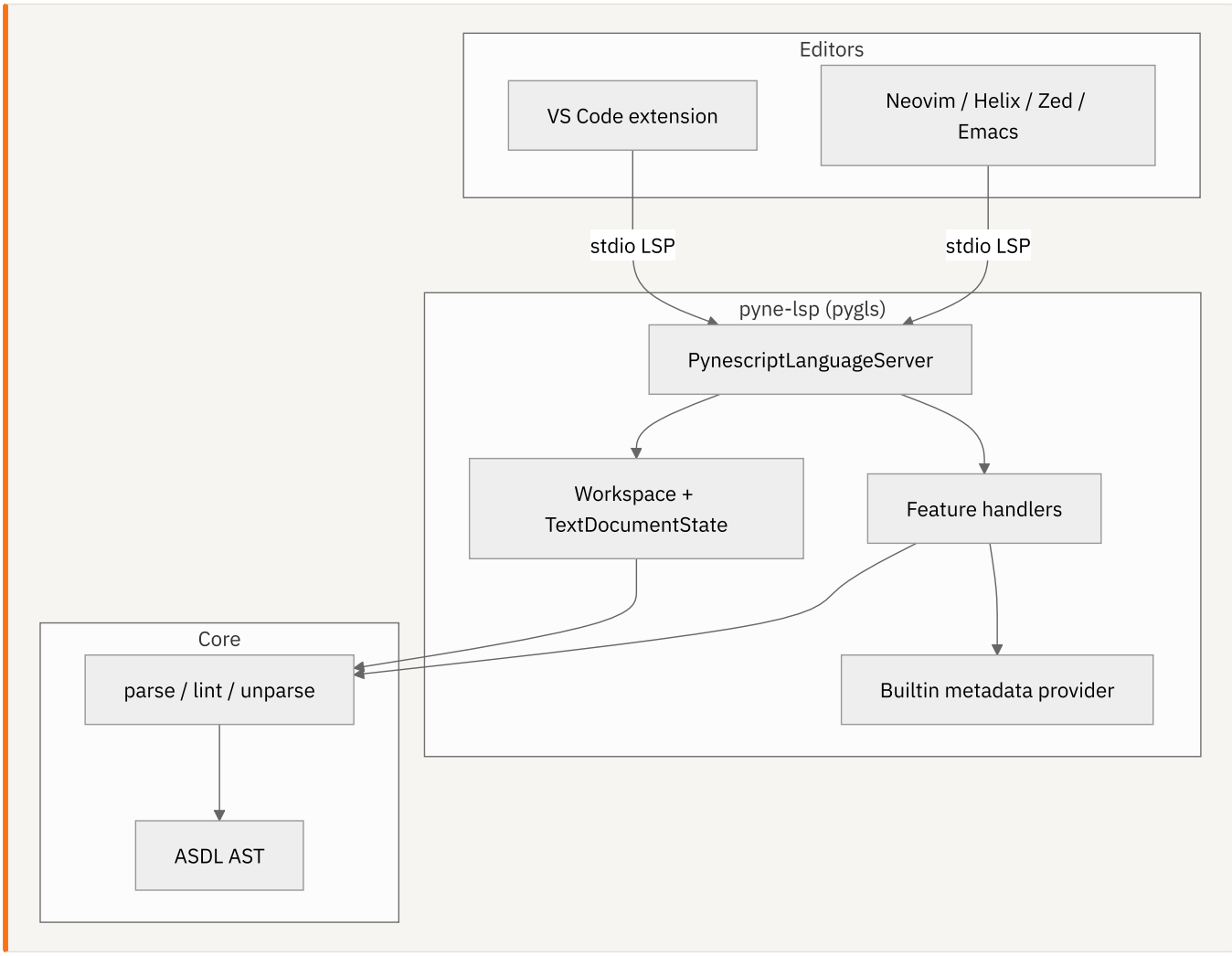
Abstract

Editors want *static* intelligence at typing cadence. Evaluation over OHLCV is *dynamic* and bar-loop expensive. PYNE therefore splits the stack:

Process	Cadence	Shared substrate
LSP (`pyne-lsp`)	keystroke / open / save	parse, lint, unparse, builtin metadata
Pro API (`backend.app`)	request / batch	parse, evaluate, plots / events / alerts
CLI / library	batch / embed	same AST + evaluator

The LSP publishes diagnostics from the linter and parse errors, completes and hovers from Fernet-ready builtin metadata (plus keywords and user enums), navigates user symbols via AST visitors, emits semantic tokens from the cached tree, and formats by round-tripping through the unparser. Feature modules live under `src/pynescript/langserver/features/`; the VS Code extension (`hoox-sh.pyne` **0.4.4**) and `clients/` configs are thin protocol hosts — VS Code maps `.pyne` **first**, then legacy Pine suffixes, to language id `pinescript`. The stdio image is `ghcr.io/hoox-sh/pyne/lsp`.

Conceptual model



Invariant: open/change/close always re-parse and re-lint the in-memory buffer. Features that need an AST re-parse on demand if the workspace cache is missing source.

Interface surface

Capability	LSP method(s)	Module
Sync	`textDocument/didOpen	didChange
Diagnostics (push + pull)	`publishDiagnostics`, `textDocument/diagnostic`	`workspace.py`, `features/diagnostics.py`
Completion (+ resolve)	`textDocument/completion`, `completionItem/resolve`	`features/completion.py`
Hover	`textDocument/hover`	`features/hover.py`
Definition / references	`textDocument/definition`, `textDocument/references`	`definitions.py`, `references.py`
Symbols	`textDocument/documentSymbol`, `workspace/symbol`	`symbols.py`, `server.py`
Formatting	`textDocument/formatting`, `rangeFormatting`	`features/formatting.py`
Inlay hints	`textDocument/inlayHint`	`features/inlay_hints.py`
Semantic tokens	`textDocument/semanticTokens/full`	`features/semantic_tokens.py`

Declared capabilities: `src/pynescript/langserver/config.py` (`get_server_capabilities`).

Internals

Path	Role
`src/pynescript/langserver/server.py`	`PynescriptLanguageServer`, method registration
`src/pynescript/langserver/workspace.py`	Document map, incremental edits, parse+lint
`src/pynescript/langserver/config.py`	ServerCapabilities
`src/pynescript/langserver/features/*`	Per-method handlers
`src/pynescript/langserver/providers/*`	Metadata + completion builders
`src/pynescript/langserver/__main__.py`	STDIO entry (`pyne-lsp` / alias `pynescript-lsp`)
`vscode-extension/`	Language client + TextMate grammar (`hoox-sh.pyne` 0.4.4)
`clients/`	Neovim, Zed, Emacs snippets (Helix / Sublime are documented, not files)

Console scripts are separate: `pyne` (Click CLI; alias `pynescript`) vs `pyne-lsp` (pygls; alias `pynescript-lsp`). Do not conflate them — see architecture.

Tracks in this tab

1. Architecture — server lifecycle, workspace, capabilities
2. Diagnostics through Inlay hints — feature manuals
3. Builtin metadata — generation + Fernet `.enc` / `CRYPTO_KEY`
4. VS Code extension — client packaging
5. Clients — Neovim, Zed, Emacs, Helix, Sublime

See also

- Pro API — HTTP evaluate path sharing the same AST
- Evaluate contract
- Linter
- Unparser
- Editors guide (end user)

LSP ARCHITECTURE

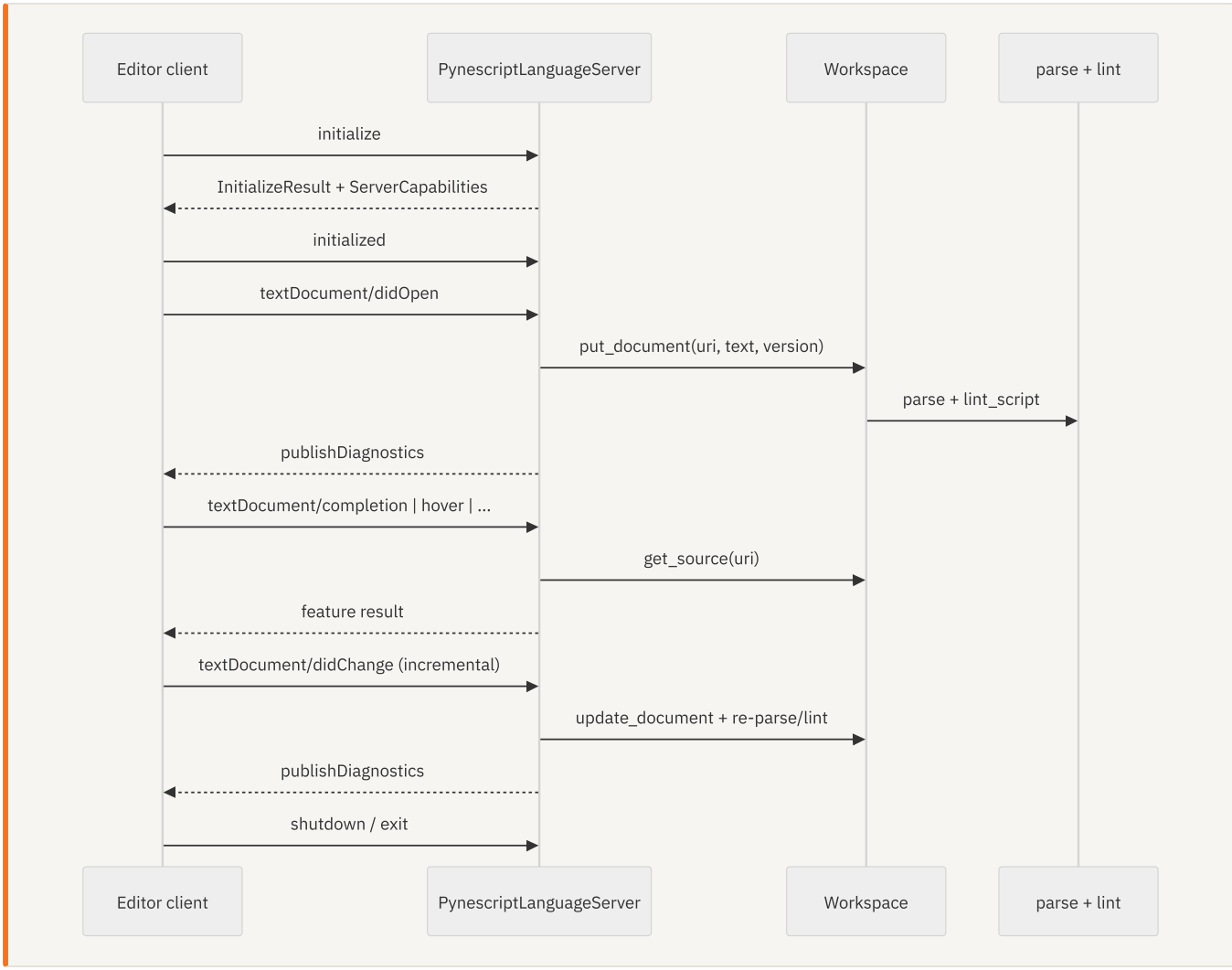
PynescriptLanguageServer lifecycle, Workspace document model, capability declaration, and STDIO transport.

LSP ARCHITECTURE

Abstract

The language server is a single-process pygls application. It owns a `Workspace` of open `TextDocumentState` records, re-parses on every mutation, and dispatches LSP methods to pure-ish feature handlers that take `(params, source)` rather than holding server globals. Capabilities are declared once at `initialize`; transport defaults to STDIO for editor integration.

Conceptual model



Interface surface

Process entry

```
# Editable install
pip install -e ".[lsp]"
python -m pynescript.langserver   # same as pyne-lsp / pynescript-lsp
make run-lsp
```

Entry: `src/pynescript/langserver/__main__.py` constructs `PynescriptLanguageServer()` and calls `server.start_io()` (STDIO JSON-RPC).

`pyproject.toml` registers:

```
pyne-lsp = "pynescript.langserver.__main__:main"
pynescript-lsp = "pynescript.langserver.__main__:main"   # alias
```

Server class

`PynescriptLanguageServer` (`server.py`):

- Subclasses `pygls.lsp.server.LanguageServer` with `name="Pynescript"` and `version=__version__` from `src/pynescript/__about__.py` (**0.3.14**).
- Instantiates `self.pine_workspace = Workspace()`.
- Registers handlers in `setup_method_handlers()` via `@self.feature(...)`.

Document sync

Event	Behavior
`didOpen`	`put_document` → parse/lint → push diagnostics
`didChange`	Incremental or full-text apply → re-parse/lint → push diagnostics
`didClose`	Remove document; publish empty diagnostic list
`didSave`	Re-publish diagnostics for current buffer

Sync options (`config.get_server_capabilities`):

- `TextDocumentSyncKind.Incremental`
- `open_close=True`
- `save` with `include_text=True`
- `will_save` / `will_save_wait_until` off

Capability set (declared)

From `config.py`:

- Diagnostic provider (`identifier="pynescript-diagnostics"`, `workspace_diagnostics=False` in options; workspace pull handler still exists)
- Completion: trigger `"."`, `resolve_provider=True`
- Hover, definition, references

- Document + workspace symbols
- Full + range formatting
- Code action (Convert to Pine v6) + executeCommand `pynescript.convertToV6`
- Inlay hints (`resolve_provider=False`)
- Semantic tokens full (legend of standard token types/modifiers; range=false)
- Signature help is **not** advertised.

File filters (`get_filter_options`): `*.pine` , `*.pinev5` , `*.pinev6` under language id `pinescript` . **`.pyne` is not in this helper** — VS Code still maps it via the extension contribution (`extensions: .pyne` first).

Internals

Workspace

`Workspace` (`workspace.py`) maps `uri → TextDocumentState` :

```
TextDocumentState
  uri, source, version
  ast | None
  diagnostics: list[LintWarning]
  parse_error, parse_error_line
```

`_parse_and_lint` :

1. `parse(source, filename=uri)` — on success, `lint_script(source, filename=uri)` .
2. On exception: clear AST, store `parse_error` string, extract line via regex `line[:\s]+(\d+)` .

Incremental edits: `_apply_text_edit` splits on `\n` , pads the line list when the range sits at/past EOF (append without a trailing newline), clamps columns, then replaces the range. Out-of-range positions no longer leave a stale buffer. Whole-document change events replace `source` wholesale. Identical text after apply skips re-parse/lint (version still updates).

Diagnostics conversion (`_lint_warnings_to_diagnostics`):

- Map severity strings → `DiagnosticSeverity` .
- Source tag `"PineScript"` , code from lint rule.
- Append synthetic **E001** error for parse failures at the extracted line.

Feature dispatch pattern

Handlers are functions, not server methods, e.g.:

```
@self.feature(lsp.TEXT_DOCUMENT_COMPLETION)
def text_completion(params):
    source = self.pine_workspace.get_source(params.text_document.uri)
    return completion_feature.handle_completion(params, source)
```

Definition / references / symbols / hover / completion / inlay / semantic tokens receive the **workspace-cached AST** (`tree=doc.ast`) so they skip a redundant parse. Pass `tree=None` when the last parse failed. Omit `tree` only in isolated tests (handler then parses from `source`).

Pull diagnostics

- `textDocument/diagnostic` → `RelatedFullDocumentDiagnosticReport` with `result_id=f"{uri}-{version}"`.
- `workspace/diagnostic` → one full report per open document from `get_all_diagnostics()`.

Workspace symbols

`workspace/symbol` walks each open document's AST via `_collect_workspace_symbols`: `FunctionDef` → Function, `TypeDef` → Class, `EnumDef` → Enum, `Assign` to `Name` → Variable. Filter is case-insensitive substring on `params.query`.

Invariants and edge cases

1. **Parse failure is non-fatal for the process.** AST-dependent features return `None` or `[]`; diagnostics still surface E001.
2. **Source of truth is the open buffer**, not disk. Save only re-publishes; it does not re-read files.
3. **Incremental sync pads/clamps rather than dropping the edit.** Ranges at EOF grow the line list; columns are clamped to the current line length.
4. **Version** is stored and used in pull-diagnostic `result_id`; the server does not reject out-of-order versions itself.
5. **Nuitka onefile** embeds providers data and may rely on encrypted metadata — see builtin metadata. Architecture of handlers is identical to the pure-Python path.

Worked example — minimal initialize

Client → server (conceptual):

```
{
  "method": "initialize",
  "params": {
    "capabilities": {},
    "clientInfo": { "name": "example", "version": "1" }
  }
}
```

Server returns `InitializeResult` with `serverInfo.name = "Pynescript Language Server"` and the full `ServerCapabilities` table from `get_server_capabilities()`.

Failure modes

Symptom	Likely cause
No diagnostics on open	Client did not send `didOpen` or language id is not `pinescript`
Completions empty	Metadata file missing/undecryptable; see builtin metadata
Format no-ops	Parse error (handler returns `[]`) or unparsed text equals source
Extension can't spawn server	`pynescript-lsp` not on `PATH`; check VS Code extension

See also

- Diagnostics

- Builtin metadata
- VS Code extension
- Nuitka build

DIAGNOSTICS

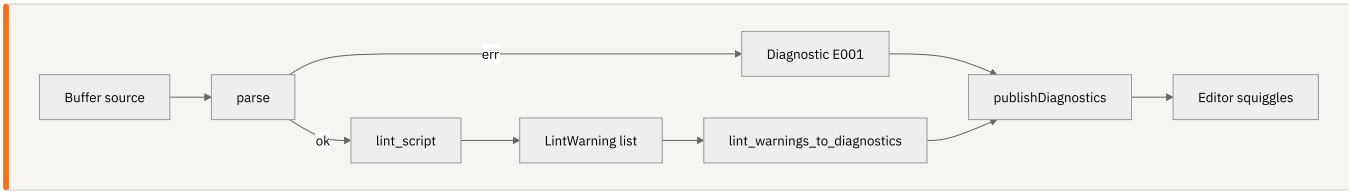
Push and pull diagnostics: parse errors, lint rules, severity mapping, and code-action stubs.

DIAGNOSTICS

Abstract

Diagnostics are the primary static feedback channel. The workspace re-parses and re-lints on every open/change/save, converts `LintWarning` objects (and parse failures) into `lsp.Diagnostic`, then **pushes** them with `textDocument/publishDiagnostics`. Clients that support LSP 3.16+ pull diagnostics can also request `textDocument/diagnostic` or `workspace/diagnostic`.

Conceptual model



Interface surface

Push (always on)

Triggered from `server.py` on `didOpen`, `didChange`, `didSave`. Close clears diagnostics (`items=[]`).

Pull

Method	Response
`textDocument/diagnostic`	`RelatedFullDocumentDiagnosticReport` with `kind=full`, `result_id="{uri}-{version}"`
`workspace/diagnostic`	One `WorkspaceFullDocumentDiagnosticReport` per open URI

Capability: `diagnostic_provider` with `identifier="pynescript-diagnostics"`, `inter_file_dependencies=False`.

Diagnostic shape

Field	Value
`range`	0-based line; column from warning or 0; end spans remainder of line (capped)
`severity`	error / warning / information / hint
`message`	Human-readable lint or parse message
`source`	`"PineScript"`
`code`	Lint code (e.g. `W001`) or `E001` for parse errors

The richer conversion in `features/diagnostics.py` also supports:

- `code_description` hrefs for codes starting with `E` / `W` (docs URLs)
- `tags`: `W001` → Unnecessary, `W002` → Deprecated

Push and pull both call `Workspace._lint_warnings_to_diagnostics`, which delegates lint rows to `features/diagnostics.lint_warnings_to_diagnostics` (noise filters + tags) and then appends synthetic **E001** for parse failures.

Internals

Path	Role
`src/pynescript/langserver/workspace.py`	`_parse_and_lint`, E001 append, incremental edits
`src/pynescript/langserver/features/diagnostics.py`	Lint conversion, C001 noise filter, tags, `create_quick_fix`
`src/pynescript/ast/linter.py`	Rule engine producing `LintWarning`

Severity map

```
error        → DiagnosticSeverity.Error
warning      → Warning
info|information → Information
hint         → Hint
(default)    → Warning
```

Parse errors

On exception from `parse`:

- `ast = None`, `diagnostics = []` (lint skipped)
- Message stored as `parse_error`
- Synthetic diagnostic code **E001**, severity Error, line from regex on the exception text

Quick fixes (feature module)

`create_quick_fix` (not yet fully wired through `workspace/executeCommand` / codeAction handlers in `server.py`):

- **W001** → insert `//@version=6\n` at document start
- **C002** (long line) → suggest format document command

Capability advertises QuickFix / Refactor / SourceOrganizeImports kinds for future wiring.

Invariants and edge cases

1. **Lint does not run when parse fails** — only E001 is shown.
2. Warnings without a line number are dropped (`None` conversion).
3. End column heuristic uses the remainder of the line and clamps to 2000 characters — not a precise token span.
4. `inter_file_dependencies=False`: no cross-file analysis; each URI is independent.
5. Push and pull should agree for a given buffer version; both read the same `TextDocumentState`.

6. **Noise filters:** `C001` on names that already lack `_` (camelCase `fastMA`) is dropped. The `C003` block-`if` noise filter was removed with the rule (retired 0.6.0).

Worked example

Source:

```
indicator("x")
plot(close)
```

If the linter emits a missing-version warning (code `W001`), the client receives a Warning diagnostic on the relevant line with `source: "PineScript"` and may offer the version-header quick fix when code actions are connected.

On a hard parse failure at line 3, expect:

```
{
  "severity": 1,
  "code": "E001",
  "source": "PineScript",
  "message": "<exception text>"
}
```

Failure modes

Symptom	Cause
Stale squiggles after edit	Client using full-document sync incorrectly, or version mismatch
Empty diagnostics on broken file	Unexpected: parse errors should still emit E001 — check client filter on `source`
No workspace pull results	Only **open** documents are tracked; closed files are absent

See also

- Linter (core)
- Error model
- Architecture

COMPLETION

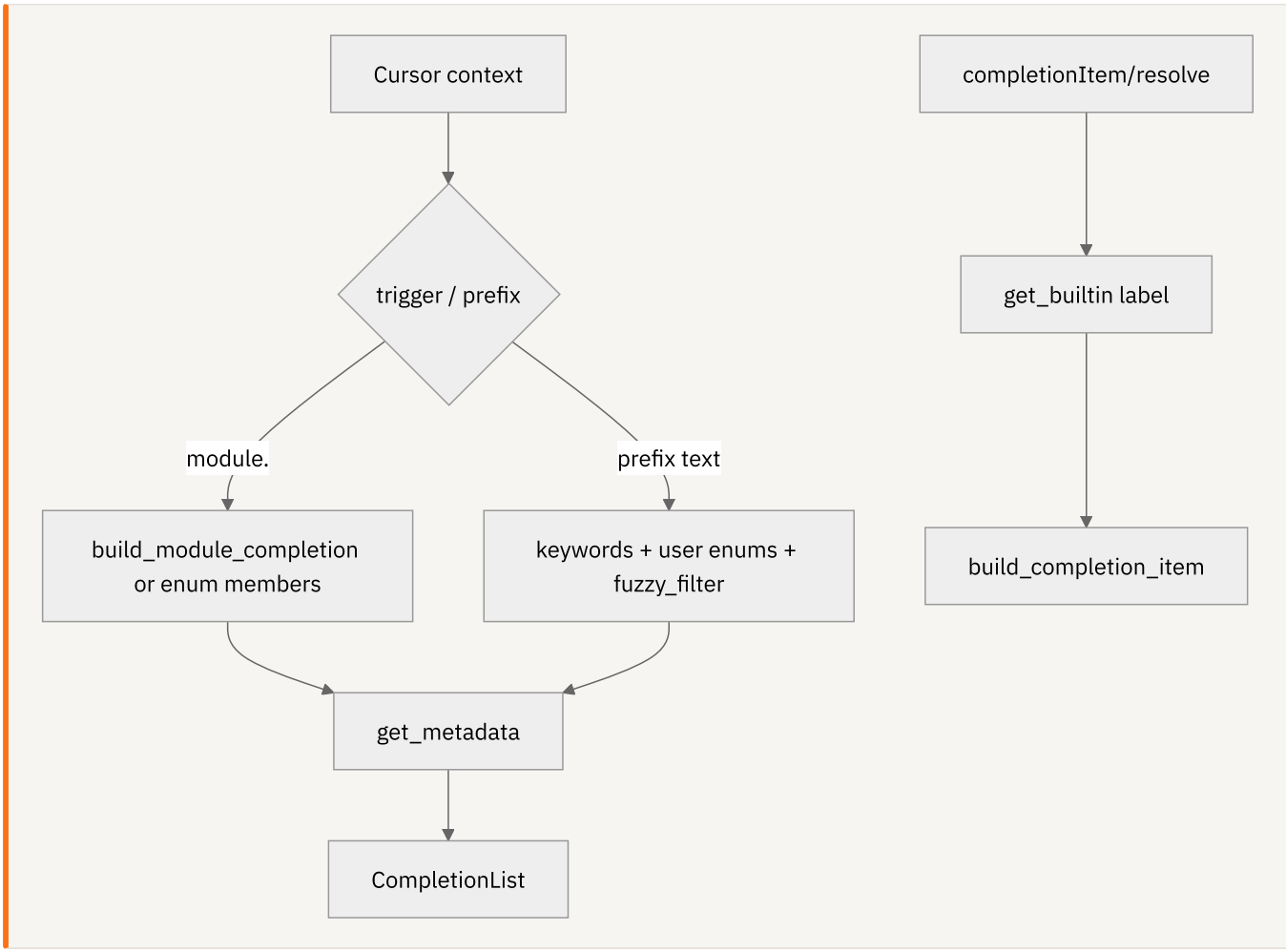
textDocument/completion and completionItem/resolve from builtin metadata, module-dot triggers, and fuzzy filter.

COMPLETION

Abstract

Completion is catalog-driven, not evaluator-driven. The server loads builtin metadata (plaintext JSON in development, Fernet-encrypted in compiled binaries), then also offers **Pine keywords** (`PINE_KEYWORDS`) and **user** `enum` **types / members** collected from the workspace AST. Resolve re-hydrates documentation for a single item.

Conceptual model



Interface surface

Method	Capability
`textDocument/completion`	`trigger_characters=["."]` , returns `CompletionList`
`completionItem/resolve`	`resolve_provider=True`

Handler: `features/completion.py` → `handle_completion` / `handle_completion_resolve`.

Trigger logic

1. Compute `text_before_cursor` on the current line.
2. `get_trigger_char` — if the previous character is `.` (or `(`, `,`, space), treat as trigger.
3. Collect user enums from the cached AST (`collect_user_enums`).
4. If the last token contains `.` (e.g. `ta.` or `Side.`):
   - user-enum prefix → `build_enum_member_completion`
   - else `build_module_completion(module)` for catalog namespaces
5. Else merge `build_keyword_items`, `build_enum_name_items`, and `build_completion_list(prefix=prefix)`.

Word boundaries for Pine identifiers: `[a-zA-Z_][a-zA-Z0-9_.]*` (`protocol/utils.py`).

Item fields

Built by `providers/completion_items.py`:

Field	Source
`label`	metadata `label` (e.g. `ta.sma`)
`kind`	`CompletionItemKind.Function`
`detail`	signature / detail string
`documentation`	Markdown from metadata
`insert_text`	Snippet if `${...}` present, else plain label
`insert_text_format`	Snippet or PlainText
`filter_text`	dotted parts + brief
`sort_text`	modules first (`\x01…`), root second (`\x02…`)

Category headers may appear as Folder-kind items with empty insert text and labels like `--- Technical Analysis (ta.*) (N) ---`.

Internals

Path	Role
`features/completion.py`	Request context + dispatch
`providers/completion_items.py`	List / item / module builders
`providers/builtin_metadata.py`	Load cache, `fuzzy_filter`, `get_builtin`
`protocol/utils.py`	Word + trigger helpers

Fuzzy filter scores

`fuzzy_filter(query, items, limit=50)`:

Match	Score
Exact label	1000
Prefix	500
Substring in label	100
Category	50
Brief	25

Results sorted by score descending, capped at `limit`.

Resolve

`handle_completion_resolve` looks up `params.label` in metadata; if found, rebuilds a full `CompletionItem`. Unknown labels return unchanged.

Invariants and edge cases

1. **No local function / variable completion yet** — user **enums** and **keywords** are included; local `myFunc` is not, unless it appears in metadata.
2. Empty metadata dict yields an empty list (failed decrypt / missing files).
3. Module completion uses `startswith(module + ".")` — nested namespaces beyond one dot still work if labels are fully qualified.
4. Snippets are generated at metadata build time (`scripts/generate_builtin_metadata.py`); incompleteness of placeholders is a generator concern, not the LSP handler.

Worked example

User types `ta.` → module completion for all `ta.*` labels.

User types `sma` without a module → fuzzy filter may return `ta.sma` among others (substring score).

Resolve on `ta.sma` reloads full documentation markup for the detail pane.

Failure modes

Symptom	Cause
Zero items always	Metadata not loaded — check builtin metadata
Dot trigger ignored	Client did not set completion trigger characters from server capabilities
Snippets inserted raw	Client disabled snippet support; extension setting `pynescript.completion.snippets`

See also

- Hover — shares metadata
- Builtin metadata
- Inlay hints — uses return types from `detail`

HOVER

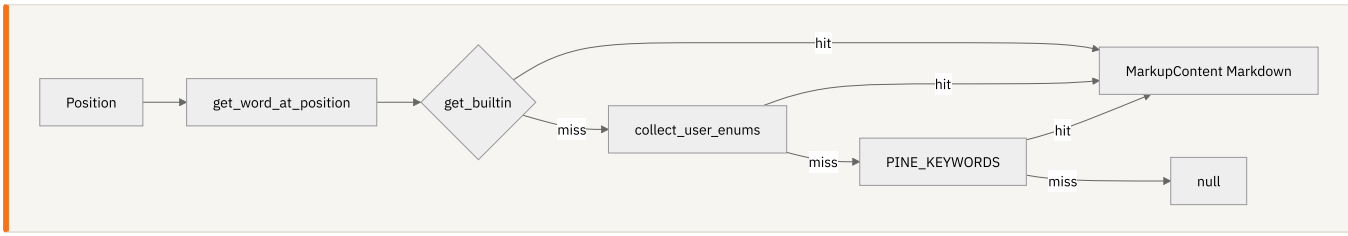
textDocument/hover — Markdown documentation cards for Pine builtins from metadata.

HOVER

Abstract

Hover answers “what is this identifier?” for **builtins**, then **user enums**, then **keywords**. The handler extracts the word under the cursor (including dotted forms), looks up metadata / enum members / `PINE_KEYWORDS`, and returns a Markdown card. Local functions and variables still return `null` (use definition).

Conceptual model



Interface surface

- Method: `textDocument/hover`
- Capability: `hover_provider=True`
- Handler: `features/hover.py` → `handle_hover`

Lookup order

1. Exact word at cursor via `get_word_at_position` (identifiers may include `.`).
2. Builtin lookup: full word, then leaf after `.`, then `module.word` if the preceding token ends with `.`.
3. Else user-enum type or `Enum.member` from the cached AST.
4. Else keyword brief from `PINE_KEYWORDS`.
5. Build hover with range covering `[start, end)` of the word on the line.

Card structure

{'{'}detail{'}'}

```
{brief}

---
{first documentation paragraph, ≤500 chars}

**Example:** …
**See also:** `ta.ema`, …
```

Examples and related maps are **hardcoded** for a small high-value set (`ta.sma` , `ta.ema` , `ta.rsi` , `ta.macd` , strategy entry/exit, etc.). Others show signature + brief only.

External TradingView doc links are intentionally omitted (`_build_docs_link` returns empty) to keep hover self-contained.

Internals

Path	Role
`features/hover.py`	Word resolution + Markdown assembly
`providers/builtin_metadata.py`	`get_builtin`
`protocol/utils.py`	Word extraction

Invariants and edge cases

1. **No source → null.** Missing buffer aborts early.
2. **Line OOB → null.** Cursor past last line is ignored.
3. **Empty word → null.** Whitespace / punctuation yields no hover.
4. **User functions / locals / UDTs** are not inferred for hover (enums are). Use definition / outline instead.
5. Documentation truncation is first-paragraph / 500-char only — full docs live in metadata and completion resolve.

Worked example

Cursor on `sma` in `plot(ta.sma(close, 14))` :

- Word may be `ta.sma` if the dotted identifier is one match, or `sma` with module recovery from `ta.` .
- Response `contents.kind = markdown` with fence of the metadata `detail` line.

Failure modes

Symptom	Cause
Hover never appears	Metadata empty or word not in catalog
Wrong symbol	Identifier regex swallowed neighboring text — rare with `_.` pattern
Stale docs after adding builtin	Regenerate metadata — builtin metadata

See also

- Completion
- Builtin metadata
- Navigation

NAVIGATION

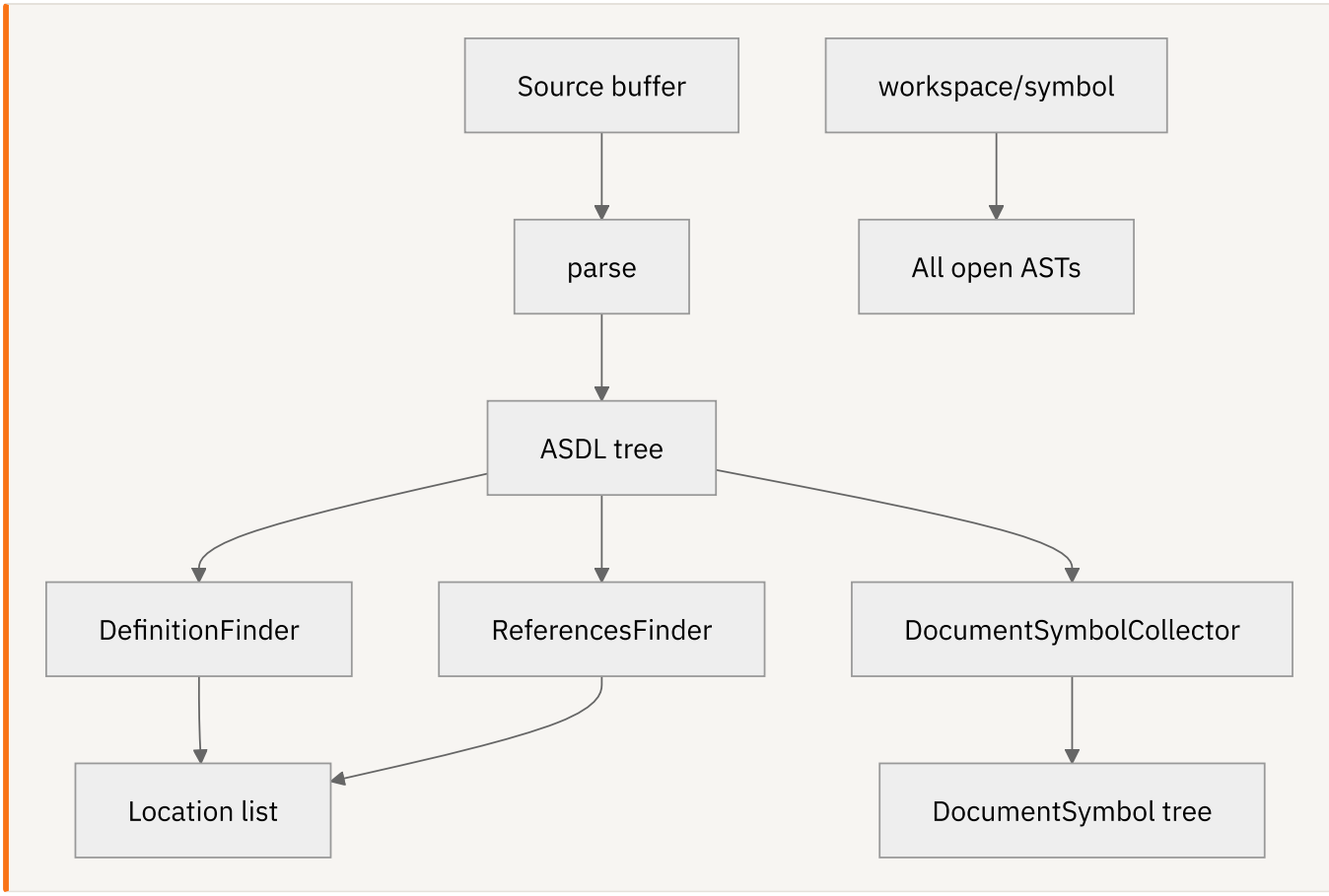
Go-to-definition, find-references, document outline, and workspace symbol search over the ASDL AST.

NAVIGATION

Abstract

Navigation features operate on **user AST symbols**, not the builtin catalog. Definition and references walk a freshly parsed tree with `NodeVisitor` subclasses; document symbols build a hierarchical outline; workspace symbols scan all **open** documents. Locations currently anchor to line starts (coarse ranges) — enough for jump lists, not character-precise rename.

Conceptual model



Interface surface

Method	Handler	Returns
`textDocument/definition`	`definitions.handle_definition`	`list[Location] | None`
`textDocument/references`	`references.handle_references`	`list[Location]` (possibly empty)
`textDocument/documentSymbol`	`symbols.handle_document_symbols`	`list[DocumentSymbol]`
`workspace/symbol`	`server._collect_workspace_symbols`	`list[SymbolInformation]`

Capabilities: `definition_provider`, `references_provider`, document + workspace symbol options with work-done progress flags.

Definition (`DefinitionFinder`)

Matches target word against:

- `FunctionDef.name`
- `TypeDef.name`
- `Assign` targets (`Name` or nested names)

Ignores mere call sites (`visit_Call` returns early when the callee name matches).

References (`ReferencesFinder`)

- `Name` with Load / Store contexts
- Function and type definitions when `include_declaration` is true
- Call sites of bare `Name` callees

`params.context.include_declaration` controls whether declarations appear.

Document symbols

Hierarchical:

AST	SymbolKind	Children
`FunctionDef`	Function	Variables assigned inside body
`TypeDef`	Class	Fields (`Assign` targets)
`EnumDef`	Enum	Members
Top-level `Assign`	Variable	—

Function `detail` is a coarse `function name()` string; assignments may show callee module attribute as detail when RHS is a call.

Workspace symbols

Flat `SymbolInformation` for functions, types, and assigned names across open buffers; filtered by lowercase query substring.

Internals

Path	Role
`features/definitions.py`	Go-to-definition visitor
`features/references.py`	Find-all-references visitor
`features/symbols.py`	Document outline
`server.py`	Workspace symbol aggregation

Handlers accept the **workspace-cached AST** (`tree=doc.ast` from `server.py`). They re-parse from `source` only when `tree` is omitted (tests / isolated calls). Pass `tree=None` after a failed parse so the handler does not retry.

Invariants and edge cases

1. **Parse failure → empty / null** (definition `None`, others `[]`).
2. **Builtin names** have no definition in-file; clients should not expect `ta.sma` to resolve to a library file URI.
3. **Ranges are line-coarse** for many locations (`character=0`); selection ranges for document symbols use `col_offset` when present.
4. **Scope is naive** — same identifier in nested functions may collect multiple definitions; no shadowing analysis.
5. **Closed documents** disappear from workspace symbol search.

Worked example

```
//@version=6
indicator("nav")
length = 14
mySma(src, len) =>
    ta.sma(src, len)
v = mySma(close, length)
```

- Definition on `mySma` at the call site → function line.
- References on `length` → assign + call argument (and declaration if included).
- Outline → `length` variable, `mySma` function (with locals), `v` variable.

Failure modes

Symptom	Cause
Jump lands column 0	Expected with current Location encoding
Missing references inside function when targeting another function	`visit_FunctionDef` only walks body when the def name **differs** from target — intentional for declaration handling; calls inside same-named functions are a known edge
Empty outline	Parse error or empty source

See also

- Architecture
- Hover — builtins vs user symbols

- Diagnostics

FORMATTING

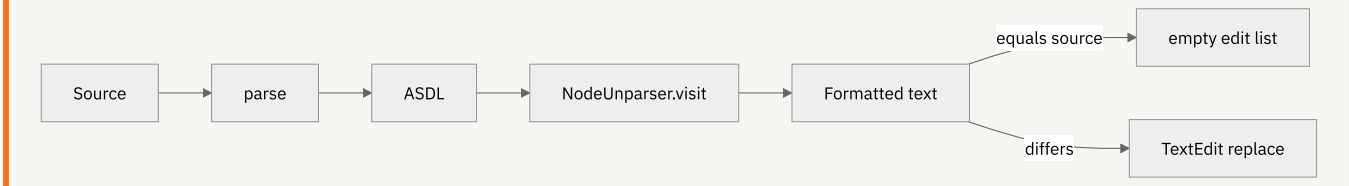
Document and range formatting via AST parse → NodeUnparser round-trip.

FORMATTING

Abstract

Formatting is **canonical unparse**, not a style linter. The handler parses the buffer, runs `NodeUnparser`, and if the result differs from the source, replaces the document (or the requested line range) with a single `TextEdit`. There is no prettier-like option matrix; `FormattingOptions` from the client are currently unused.

Conceptual model



Interface surface

Method	Handler
`textDocument/formatting`	`handle_formatting`
`textDocument/rangeFormatting`	`handle_range_formatting`

Capabilities: both providers `True` in `config.py`.

Full document

1. Parse with filename `"<format>"`.
2. `formatted = NodeUnparser().visit(tree)`.
3. If identical to source → `[]`.
4. Else one `TextEdit` spanning `(0,0)` → last line/col with `new_text=formatted`.

Range

1. Unparse the **entire** document.
2. Slice **formatted** lines and **source** lines by the client range's start/end line indices.
3. If the in-range slices match → `[]`.
4. Else replace `params.range` with the formatted line slice.

This is line-aligned, not AST-node-aligned: partial mid-line ranges may produce surprising splices when unparse changes line structure.

Internals

Path	Role
`features/formatting.py`	LSP handlers
`src/pynescript/ast/helper.py`	`parse`
`src/pynescript/ast/unparser.py`	`NodeUnparser`

VS Code command `pynescript.formatDocument` delegates to `editor.action.formatDocument`, which hits this provider when the language client is active.

Invariants and edge cases

1. **Parse failure →** `[]` (no partial format; silent no-op from the client's perspective).
2. **Idempotence goal:** format twice should stabilize if unparse is canonical; any non-idempotent unparse is a core bug, not an LSP bug.
3. **Comments / trivia:** fidelity is whatever the unparser preserves — not a full CST pretty-printer.
4. **Range formatting** still requires a full successful parse of the whole file.
5. Tab size / insert spaces from `FormattingOptions` are ignored today.

Worked example

Before:

```
//@version=6
indicator("x")
plot(  close  )
```

After full format (illustrative — exact whitespace follows `NodeUnparser`):

```
//@version=6
indicator("x")
plot(close)
```

If source already matches unparse output, the server returns an empty edit list and the editor leaves the buffer alone.

Failure modes

Symptom	Cause
Format does nothing on broken syntax	Expected: exception → `[]`
Range format rewrites wrong lines	Unparse shifted line count; prefer full-document format
Client "format on save" loops	Unparse not idempotent — fix unparser, not client

See also

- Unparser

- Architecture
- VS Code extension

SEMANTIC TOKENS

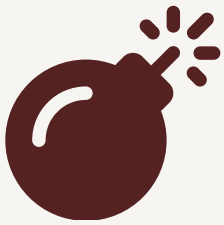
textDocument/semanticTokens/full — AST visitor, custom legend, and LSP delta encoding.

SEMANTIC TOKENS

Abstract

Semantic tokens let the server paint identifiers with roles richer than TextMate scopes (library namespaces vs user variables, function defs vs properties). PYNE advertises a **full** semantic-tokens provider with a **short custom legend** (not the full LSP standard set) and implements `textDocument/semanticTokens/full` by walking the workspace AST.

Conceptual model



Interface surface

Capability (`config.py` → `semantic_tokens_provider`):

```
legend.token_types (index order is contract):
  namespace, type, class, function, method, variable,
  parameter, property, keyword, string, number, operator, comment

legend.token_modifiers (bit 0 = declaration):
  declaration, definition, readonly, defaultLibrary

range: false
full: true
```

Handler: `features/semantic_tokens.py` → `handle_semantic_tokens(params, source, tree=…)` .

Response type: `lsp.SemanticTokens(data=[...])` — LSP-encoded five-tuples as a flat `int` array (relative line/character deltas).

What is emitted

AST	Token type	Modifiers
`FunctionDef.name`	function	definition | declaration
`TypeDef.name`	class	definition | declaration
`EnumDef.name`	type	definition | declaration
`Assign` to `Name`	variable	declaration
`Attribute` whose value is a builtin namespace (`ta`, `math`, `strategy`, …)	namespace + method	defaultLibrary (+ readonly on ns)
Other `Attribute.attr`	property	—

Builtin namespace set (`_BUILTIN_NS`): `ta`, `math`, `str`, `array`, `matrix`, `map`, `strategy`, `request`, `input`, `color`, `line`, `label`, `box`, `table`, `polyline`, `log`, `ticker`, `timeframe`, `chart`, `runtime`, `syminfo`, `barstate`, `session`, `time`.

Internals

Path	Role
`features/semantic_tokens.py`	Collector + `_encode`
`config.py`	Legend + capability — **indices must match** `_TT`
`server.py`	`TEXT_DOCUMENT_SEMANTIC_TOKENS_FULL`; passes cached `doc.ast`

Attribute columns prefer `end_col_offset - len(attr)` when present; otherwise parent `col_offset` (approximate).

TextMate from `vscode-extension/syntaxes/pinescript.tmLanguage.json` still colors keywords, strings, and comments — the visitor does not emit those yet even though the legend reserves the slots.

Invariants and edge cases

1. **Never throws** — missing source, `tree=None`, or parse exception yields empty `data`.
2. **No range provider** — clients must not request `semanticTokens/range`.
3. **Legend is fixed** at initialize; changing indices later would break cached clients.
4. Empty `data` is valid protocol (empty file or parse failure), not an error.
5. Tokens are sorted by `(line, col)` before delta encoding.

Worked example

```
//@version=6
indicator("tok")
length = 14
plot(ta.sma(close, length))
```

Expect tokens for `length` (variable/declaration), `ta` (namespace/defaultLibrary), `sma` (method/defaultLibrary). Keywords such as `indicator` remain TextMate-only.

Failure modes

Symptom	Cause
Semantic highlighting off despite tokens	Client `editor.semanticHighlighting.enabled` false
Colors look “shifted” after a legend edit	Client cached old type indices — restart the language client
Client errors on range request	`range=False` in capabilities; client should not call it
Attribute token sits on the wrong column	Missing `end_col_offset` on the AST node — approximate path

See also

- VS Code extension — TextMate grammar
- Architecture
- Inlay hints — complementary visual annotations

INLAY HINTS

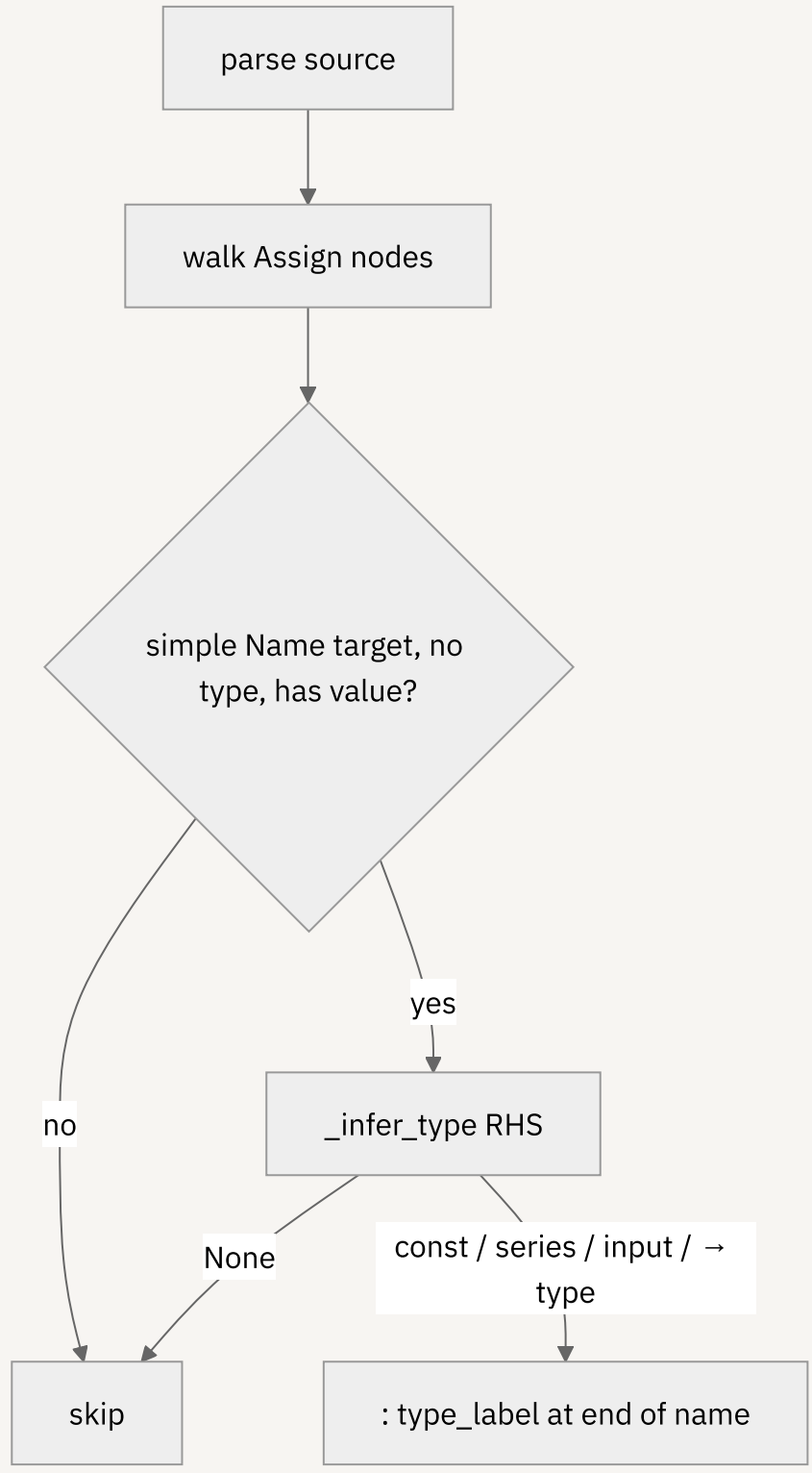
Inferred type annotations on simple assignments — const, series, and input kinds.

INLAY HINTS

Abstract

Inlay hints surface **inferred types** next to simple `name = expr` assignments when no explicit type annotation is present. Inference is deliberately shallow: constants, bare series builtins (`close` , …), `input.*` constructors, and `ta` / `math` / `str` / `color` calls whose metadata `detail` contains a `→` return type. Ambiguous operators are skipped rather than guessed wrong.

Conceptual model



Example mental model (from module docstring):

```
length = 14                →  length: const int
rsi = ta.rsi(close, 14)    →  rsi: series float   (from metadata detail)
n = input.int(5, "n")      →  n: input int
```

Interface surface

- Method: `textDocument/inlayHint`
- Capability: `InlayHintOptions(resolve_provider=False)`
- Handler: `features/inlay_hints.py` → `handle_inlay_hints` (receives cached `tree=doc.ast`)

Each hint:

Field	Value
`position`	End of target identifier (`lineno-1`, `col_offset + len(id)`)
`label`	`": {type_label}"`
`kind`	`InlayHintKind.Type`
`tooltip`	`Inferred type: {type_label}`

Returns `[]` for empty source, `None` if parse fails (client typically treats as no hints).

Internals

Constant mapping

Python value	Label
`bool`	`const bool`
`int`	`const int`
`float`	`const float`
`str`	`const string`

Builtin bare names

`open`, `high`, `low`, `close`, `volume`, `hl2`, `hlc3`, `ohlc4` → `series float`; time / bar index vars → `series int`; `na` → `na`.

Calls

- `input.{int,float,bool,string,color,symbol,session,source,time,timeframe,price}` → `input {attr}`
- `ta|math|str|color.{attr}` → parse `detail` after `→`, else fallback `"series"`

Walk

Recursive over `_fields` (same pattern as workspace symbol collection) — not limited to script top-level; nested function assigns can receive hints.

Path	Role
`features/inlay_hints.py`	Collection + inference
`providers/builtin_metadata.py`	Return-type detail for modules

Invariants and edge cases

1. **Explicit `name: type = …` suppresses hints** (`node.type is not None`).
2. **Tuple / attribute targets** are ignored.
3. **BinOp / Compare / Conditional** return `None` — no speculative arithmetic types.
4. **Metadata quality bounds accuracy** — missing `→` in detail yields coarse `series`.
5. **No resolve step** — tooltips are final.

Worked example

```
//@version=6
indicator("hints")
length = 14
src = close
avg = ta.sma(src, length)
period = input.int(20, "Period")
```

Expected hints (when parse + metadata succeed):

- `length` → `: const int`
- `src` → `: series float`
- `avg` → type from `ta.sma` detail (typically series float) or `series`
- `period` → `: input int`

`avg` will not get a hint derived from `length`'s type through the call graph — only the call's known return shape.

Failure modes

Symptom	Cause
No hints on file	Parse error → `None`
`ta.*` always `: series`	Metadata detail lacks `→` segment
Hints mid-identifier	Incorrect `col_offset` on AST node — builder issue

See also

- Completion
- Builtin metadata
- Type system

BUILTIN METADATA

JSON catalog for LSP completion and hover; generation pipeline; Fernet encryption, CRYPTO_KEY / METADATA_KEY, and decrypt path.

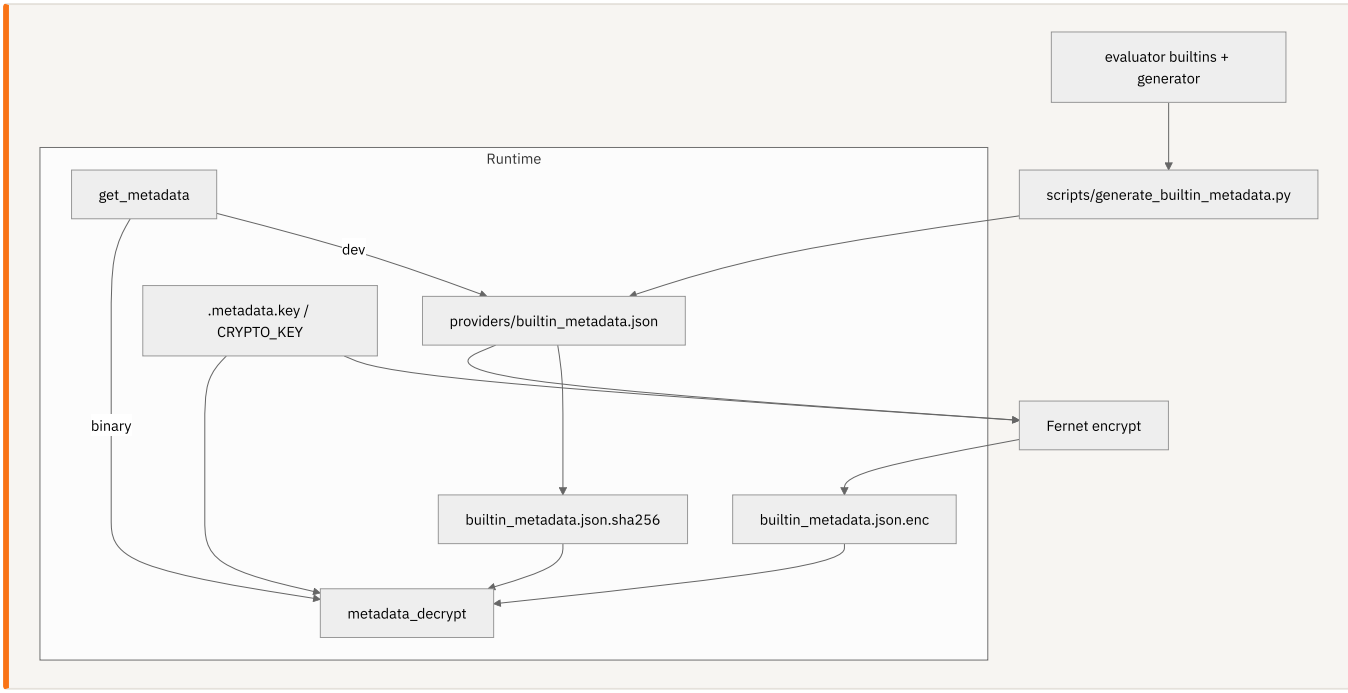
BUILTIN METADATA

Abstract

Builtin metadata is the **LSP documentation corpus**: a map from fully qualified names (`ta.sma`, `strategy.entry`, ...) to labels, signatures, briefs, snippets, and categories. It is generated from code (not hand-edited as source of truth), loaded by the language server for completion/hover/inlay, and optionally **Fernet-encrypted** for Nuitka onefile distribution so casual string extraction does not dump the full catalog.

This is obscurity for distribution hygiene, not a cryptographic access-control boundary.

Conceptual model



Interface surface (metadata record)

Typical entry in `builtin_metadata.json`:

```
{
  "ta.sma": {
    "label": "ta.sma",
    "kind": "function",
    "detail": "ta.sma(series, int) → series float",
    "brief": "…",
    "documentation": "…",
    "snippet": "ta.sma(${1:param1}, ${2:param2})",
    "category": "ta.technical_analysis"
  }
}
```

Categories drive completion headers (`ta.technical_analysis` → “Technical Analysis (ta.*)”, `builtin` → “Built-in Variables”, etc.).

Loader API

`providers/builtin_metadata.py` :

Function	Behavior
`get_metadata()`	Singleton cache; **plaintext JSON first**, else `.enc` decrypt, else `{}`
`get_builtin(name)`	Single entry or `None`
`get_builtins_by_category` / `get_all_categories`	Category queries
`fuzzy_filter`	Scored substring match for completion

Consumers

- Completion list / module / resolve
- Hover cards
- Inlay return-type extraction from `detail` after `→`

Internals

Path	Role
`scripts/generate_builtin_metadata.py`	Introspect builtins → JSON
`src/pynescript/langserver/providers/builtin_metadata.json`	Dev plaintext
`…/builtin_metadata.json.enc`	Encrypted blob for release
`…/builtin_metadata.json.sha256`	First 16 hex chars of SHA-256 of **plaintext**
`…/metadata_decrypt.py`	Fernet load + integrity check
`scripts/build/compile.py`	`encrypt_metadata()` , Nuitka include of providers dir
`scripts/build/ci_build.py`	`stage_metadata()` for CI
`scripts/build/.metadata.key`	**Gitignored** Fernet key material

Generation

After adding evaluator builtins:

```
python scripts/generate_builtin_metadata.py
```

Do not hand-edit the JSON as the long-term source of truth; regenerate from code.

Encryption (build)

`scripts/build/compile.py` → `encrypt_metadata()` / `_resolve_fernet_key()`:

1. Require `cryptography` and plaintext JSON.
2. Resolve key (first hit wins): env `CRYPTO_KEY` → `PYNESCRIPT_METADATA_KEY` → `METADATA_KEY` → existing `scripts/build/.metadata.key` → generate a new Fernet key and write that file (mode `0o600`).
3. Encrypt → `builtin_metadata.json.enc`.
4. Write truncated SHA-256 of plaintext to `.sha256`.

A random key is generated **only on first local encrypt** when no env/file key exists. For **byte-stable** encrypted artifacts across CI runs, supply a stable secret:

Context	Mechanism
GitHub Actions	`secrets.METADATA_KEY` → env (documented as `CRYPTO_KEY` in workflows / AGENTS)
Cloud Build	substitution `${_METADATA_KEY}` → `CRYPTO_KEY`
Runtime decrypt	File `.metadata.key` next to providers **or** env `PYNESCRIPT_METADATA_KEY`

Without a stable key, each CI encrypt produces a different `.enc` blob — not a functional bug, but noisy diffs and cache misses.

Decrypt path

`metadata_decrypt.py`:

1. Resolve key: PyInstaller/Nuitka data dir `.metadata.key`, else providers dir, else `PYNESCRIPT_METADATA_KEY`.
2. Fernet-decrypt `.enc`.
3. If `.sha256` present, compare `sha256(plaintext).hexdigest()[:16]`.
4. `json.loads` → dict.

Both `get_metadata()` and `get_metadata_cached()` prefer **plaintext** when both artifacts exist (developer-friendly). Difference: `get_metadata()` returns `{}` on total failure; `get_metadata_cached()` raises `FileNotFoundError` if neither file is present.

Invariants and edge cases

1. **Regenerate after builtin changes** or completion/hover/inlay drift.
2. **Empty dict on failure** fails soft (no completions) rather than crashing the server.
3. **Integrity mismatch** raises in decrypt — compiled binary should not silently serve corrupt docs.
4. **Key not in git** — never commit `.metadata.key`.
5. **Fernet is symmetric** — anyone with the binary + key material can recover plaintext; design goal is casual reverse-engineering friction and future paid-metadata swap hooks.

Worked example — local dev

```
python scripts/generate_builtin_metadata.py
# edit-free check
python -c "from pynescript.langserver.providers.builtin_metadata import get_builtin; print(get_bui
```

Worked example — encrypted binary path

```
# build (encrypts + Nuitka) — see DevOps
python scripts/build/compile.py

# or embed .metadata.key in the onefile data dir during compile
```

Failure modes

Symptom	Cause
Completions empty in release binary	Missing key / wrong `PYNESCRIPT_METADATA_KEY`
`Metadata integrity check failed`	`.enc` and `.sha256` out of sync
CI churn on `.enc`	New random key each build without `CRYPTO_KEY` / `METADATA_KEY`
Stale docs for new `ta.*`	Forgot to re-run generator

See also

- Metadata crypto (DevOps)
- Nuitka build
- Completion
- Hover

VS CODE EXTENSION

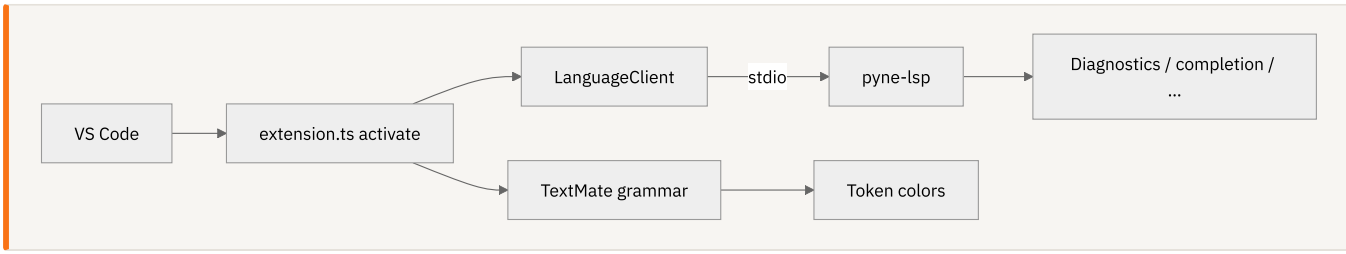
PYNE VS Code extension: HOOX branding, TextMate grammar, LanguageClient wiring, settings, and VSIX build.

VS CODE EXTENSION

Abstract

PYNE (`hoox-sh.pyne` **0.4.4**) is a **LanguageClient host** plus a rich TextMate grammar, branded as part of the **HOOX open trading stack**. It does not reimplement diagnostics or completion in TypeScript; it resolves `pyne-lsp` (then alias `pynescript-lsp`, then `python -m pynescript.langserver`) or a configured command over STDIO via `vscode-languageclient`, maps `.pyne` **(first-class)** plus `*.pine` / `*.pinev5` / `*.pinev6` / `*.pinescript` to language id `pinescript`, and exposes a small settings surface for enablement and feature toggles.

Conceptual model



Interface surface

Package identity

From `vscode-extension/package.json`:

Field	Value
name	`pyne`
displayName	`PYNE Language Support`
version	`0.4.4`
publisher	`hoox-sh`
extension id	`hoox-sh.pyne`
icon	`media/icon.png` (HOOX mark)
engines.vscode	`^1.91.0`
main	`./out/extension.js` (esbuild bundle)
dependency	`vscode-languageclient` ^10
galleryBanner	`#050505` dark

Language contribution

- **id:** `pinescript`
- **aliases:** `PYNE`, `pyne`, `pinescript`, `Pine`
- **extensions:** **`.pyne`** (first), `.pine`, `.pinev5`, `.pinev6`, `.pinescript`
- **configuration:** `language-configuration.json`
- **grammar:** `syntaxes/pinescript.tmLanguage.json` (`scopeName: source.pinescript`) — namespaces, annotations, hex colors, UDT/enum, multiline strings, plot/strategy builtins

Activation

```
onLanguage:pinescript
workspaceContains:**/*.pyne
workspaceContains:**/*.pine
workspaceContains:**/*.pinev5
workspaceContains:**/*.pinev6
```

Settings (`pynescript.*`)

Key	Default	Meaning
`lsp.enabled`	`true`	Skip activation when false
`lsp.command`	`auto`	`auto` = PATH `pyne-lsp` → `pynescript-lsp` → `python -m pynescript.langserver`. Or an absolute path. `"docker"` plus `lsp.args` can launch `ghcr.io/hoox-sh/pyne/lsp` (`${workspaceFolder}` is expanded).
`lsp.python`	`python3`	Interpreter for module launch when neither binary is on PATH (`pip install "hoox-pyne[lsp]"`)
`lsp.args`	`[]`	Extra server args
`formatting.enabled`	`true`	Passed as init option
`diagnostics.enabled`	`true`	Passed as init option
`completion.snippets`	`true`	Passed as init option

Initialization options object:

```
{
  "formattingEnabled": true,
  "snippetsEnabled": true,
  "diagnosticsEnabled": true
}
```

The Python server does **not** read these flags today. They are client-side documentation for hosts; capability advertisement is unconditional in `config.py`.

Commands

Command palette	ID	Action
PYNE: Restart Language Server	`pynescript.restartServer`	Stop then start the language client
PYNE: Format Document	`pynescript.formatDocument`	`editor.action.formatDocument` (`.pyne` / `.pine`)
PYNE: Convert to Pine v6	`pynescript.convertToV6`	Rewrite v1-v5 source toward v6 (LSP command or local Python)
PYNE: Show Language Server Output	`pynescript.showLspOutput`	Open Output channel / status-bar target
PYNE: Show Resolved LSP Launch Command	`pynescript.showLspCommand`	Show and copy resolved `pyne-lsp` launch

Client options

- Document selector: `pinescript` for `file` and `untitled` schemes
- File watcher: `**/*.{pyne,pine,pinev5,pinev6,pinescript}`
- Diagnostic collection name: `pynescript`
- Default formatter id: `hoox-sh.pyne`

Internals

Path	Role
`vscode-extension/src/extension.ts`	Activate / deactivate / client
`vscode-extension/package.json`	Contributes + scripts
`vscode-extension/syntaxes/pinescript.tmLanguage.json`	Grammar
`vscode-extension/language-configuration.json`	Brackets / comments
`out/extension.js`	Compiled JS

Server launch (`resolveLspLaunch` in `extension.ts`):

1. If `pynescript.lsp.command` is not `auto` , spawn that command with `lsp.args` .
2. Else try `pyne-lsp` , then `pynescript-lsp` , on `PATH` .
3. Else try `pynescript.lsp.python` (then `python3` / `python`) with `-m pynescript.langserver` .
4. Else status-bar error: install `hoox-pyne[lsp]` .

No `--parent-dir` / `PYTHONPATH` rewrite. Transport is STDIO. Release packaging expects `pyne-lsp` on `PATH` (pip, Nuitka onefile, or `docker run -i ghcr.io/hoox-sh/pyne/lsp:0.4.4`).

Build

```
cd vscode-extension
npm ci
npm run compile          # esbuild.mjs → out/extension.js
npm run package          # pyne-vscode-0.4.4.vsix
# or monorepo:
make build-vscode
```

Requires Node 22. `compile:tsc` is typecheck-only (`tsc --noEmit`).

Site download: https://hoox.sh/pyne-vscode.vsix (unversioned alias). Versioned `pyne-vscode-0.4.4.vsix` is built in-tree (`vscode-extension/`) and on GitHub Releases.

Invariants and edge cases

1. **`lsp.enabled === false` short-circuits activate** — no client, no restart command registration beyond early return (commands not registered).
2. **Binary vs module:** production users should install `hoox-pyne[lsp]` or the Nuitka onefile (`pynescript-lsp-*` on GitHub Releases) and set `pynescript.lsp.command` if not on PATH.
3. **Grammar works without LSP** — open a `.pyne` or `.pine` file offline and still get TextMate highlighting.
4. **`.pyne` is first-class** in `package.json` extensions + activation; legacy `.pine*` / `.pinescript` remain fully supported.
5. File watcher covers every associated suffix (`pyne`, `pine`, `pinev5`, `pinev6`, `pinescript`).

Worked example — extension development

```
pip install -e ".[lsp]"
cd vscode-extension && npm ci && npm run compile
# Launch Extension Development Host pointing at vscode-extension
```

Open a fixture `.pine` file; confirm Problems panel shows parse/lint diagnostics from the Python server.

Failure modes

Symptom	Cause
“Language client failed to start”	`pyne-lsp` missing / wrong command
No squiggles, highlighting OK	LSP disabled or server crash; check Output → PYNE Language Server
Restart does nothing	Client never started (`lsp.enabled` false)

See also

- Clients — non-VS Code hosts
- Architecture
- Nuitka build
- Editors guide

EDITOR CLIENTS

Wire pyne-lsp into Neovim, Zed, Emacs, Helix, Sublime, and generic LSP hosts.

EDITOR CLIENTS

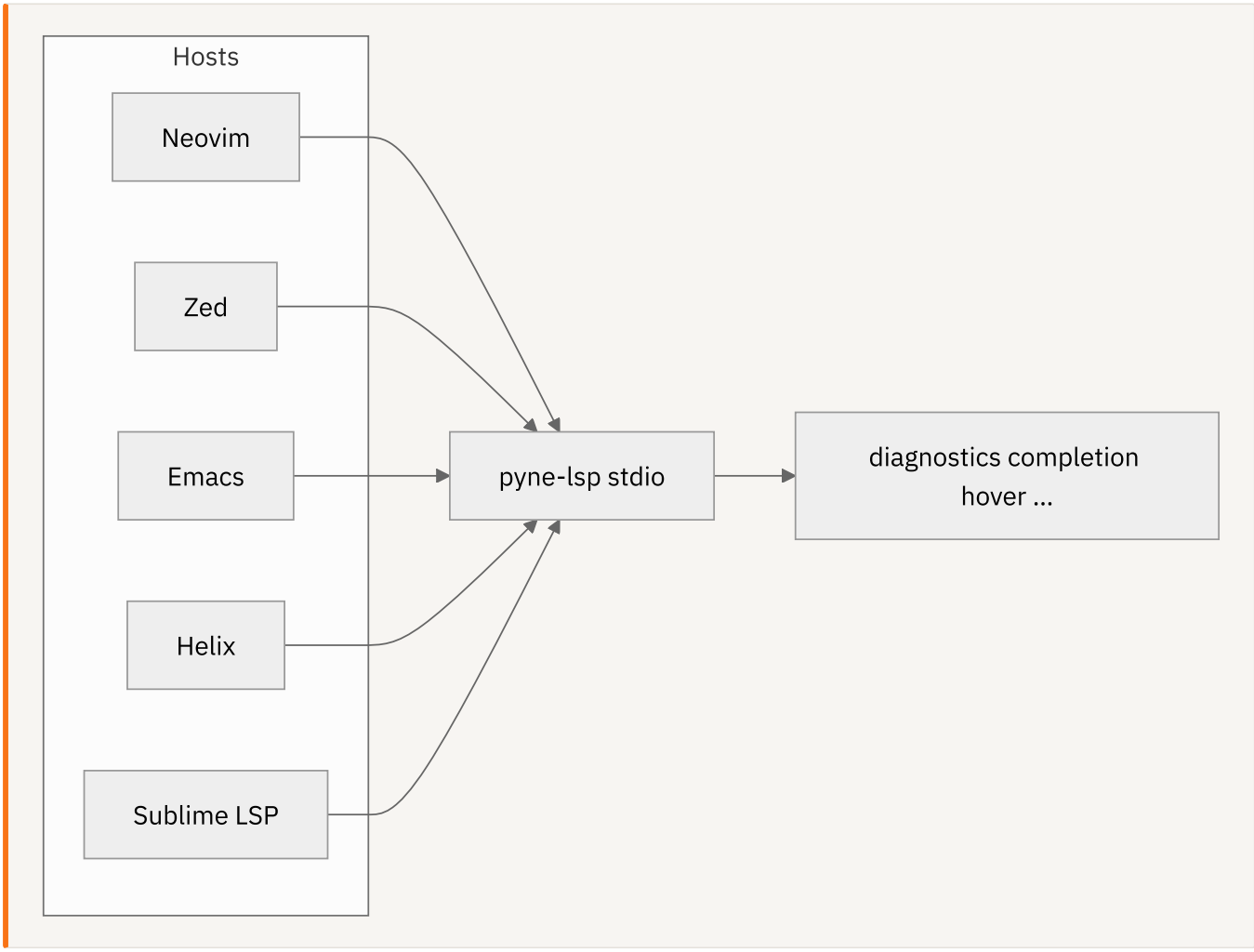
Abstract

Any editor that speaks LSP over STDIO can host PYNE intelligence. First-class snippets live in `clients/` (Neovim Lua, Zed JSON, Emacs Lisp) plus documented Helix/Sublime fragments. The VS Code path is a dedicated extension (vscode-extension); this page covers the rest.

Prerequisite: `pyne-lsp` (or `pynescript-lsp`) on `PATH` (Python 3.10+):

```
pip install "hoox-pyne[lsp]"
# development
pip install -e ".[lsp]"
```

Conceptual model



Interface surface — repo files

File	Editor
`clients/neovim.lua`	Neovim (`nvim-lspconfig` or manual)
`clients/zed.json`	Zed `settings.json` fragment
`clients/emacs.el`	Emacs `lsp-mode` + `pinescript-mode`
`clients/README.md`	Human-oriented install notes
`vscode-extension/`	VS Code / Cursor-compatible

Neovim

Return table from `clients/neovim.lua`:

```
return {
  cmd = { 'pyne-lsp' },
  filetypes = { 'pinescript' },
  root_dir = function(fname)
    return vim.fs.root(fname, { '.git', '*.pine', '*.pinev5', '*.pinev6', 'pyproject.toml' })
      or vim.fn.getcwd()
  end,
  settings = {
    pinescript = {
      formatting = { enabled = true },
      diagnostics = { enabled = true },
      completion = { snippets = true },
    },
  },
}
```

Suggested maps when attaching: `gd` definition, `gr` references, `K` hover, format via `vim.lsp.buf.format`.

Register filetype if needed:

```
au BufRead,BufNewFile *.pyne,*.pine,*.pinev5,*.pinev6,*.pinescript setfiletype pinescript
```

Zed

Merge `clients/zed.json` into `~/.config/zed/settings.json`:

```
{
  "languages": {
    "Pine Script": {
      "language_servers": ["pynescript"]
    }
  },
  "language_servers": {
    "pynescript": {
      "command": "pyne-lsp",
      "arguments": ["--stdio"],
      "languages": ["Pine Script"]
    }
  }
}
```

Zed does not ship a first-party Pine grammar in this repo. Map `*.pyne` / `*.pine` to the `"Pine Script"` language name (or whatever your grammar extension registers) so the `language_servers` key attaches.

Emacs

`clients/emacs.el` registers an `lsp-mode` client:

```
(lsp-register-client
 (make-lsp-client
  :new-connection (lsp-stdio-connection '("pyne-lsp" "--stdio"))
  :major-modes '(pinescript-mode)
  :server-id 'pynescript))
```

Defines a minimal `pinescript-mode` with font-lock keywords and `auto-mode-alist` for `\\.pine\\'` / `\\.pinev[0-9]+\\'` (add `\\.pyne\\'` locally if you use the product suffix). Optional keys: `C-c C-c` format, `M-.` definition, `M-?` references.

Helix

`~/.config/helix/languages.toml`:

```
[[language]]
name = "pinescript"
scope = "source.pinescript"
file-types = ["pyne", "pine", "pinev5", "pinev6"]
roots = ["pyproject.toml"]
command = "pyne-lsp"
args = ["--stdio"]
```

Sublime Text

With Package Control **LSP**:

```
{
  "clients": {
    "pynescript": {
      "command": ["pyne-lsp", "--stdio"],
      "selector": "source.pinescript",
      "initializationOptions": {}
    }
  }
}
```

Generic / Cursor

```
pyne-lsp
# aliases:
pynescript-lsp --stdio
# container (stdio):
docker run --rm -i -v "$PWD:/work" -w /work ghcr.io/hoox-sh/pyne/lsp:0.6.4
```

Point the host's "external language server" command at that binary. Cursor users can also load the VS Code extension (`hoox-sh.pyne`) in compatible mode when packaged as VSIX.

Internals

Server entry always uses STDIO (`start_io` in `__main__.py`). Capability negotiation is identical across hosts — differences are client UI only (how inlay hints render, whether pull diagnostics are used, etc.).

Tests: `tests/test_lsp_features.py` (handlers with fake params), `tests/test_langserver.py` (e2e with pytest-asyncio) — not editor-specific.

Invariants and edge cases

1. **Language id** should be `pinescript` for VS Code parity; some editors invent their own names — map filetypes carefully.
2. **Root directory** heuristics vary; wrong root rarely breaks single-file Pine editing (workspace is URI-keyed open buffers).
3. **Settings keys** under `pinescript` in Neovim are conventional; the Python server does not currently enforce a rich `workspace/configuration` schema for all of them.
4. Nuitka onefile can replace the module entry for air-gapped machines — same command name if installed on PATH.

Failure modes

Symptom	Cause
Server exits immediately	Missing pygls / incomplete install `.[lsp]`
Filetype never attaches	Extension not mapped to `pinescript`
Features missing in Helix	Older Helix without inlay / semantic support — grammar still useful

See also

- VS Code extension
- Architecture
- End-user editors guide

PRO API

Flask Pro API — /run with mode=auto, /optimize strategy search, plot_meta/fill, alerts + L2 webhooks, preview, backtest, and shared evaluate contract.

PRO API

The PYNE **Pro API** is a Flask HTTP service that runs Pine scripts over caller-supplied OHLCV, returns plots / series / `plot_meta` (incl. `fill` bands) / strategy events / drawings / **alerts**, optional **L2 alert webhooks**, and gates thumbnail + backtest routes behind API keys. Default evaluate path is `mode=auto` (compile when safe, interpret on fallback). It is intentionally **not** the language server: different auth, scaling, and failure domains.

Abstract

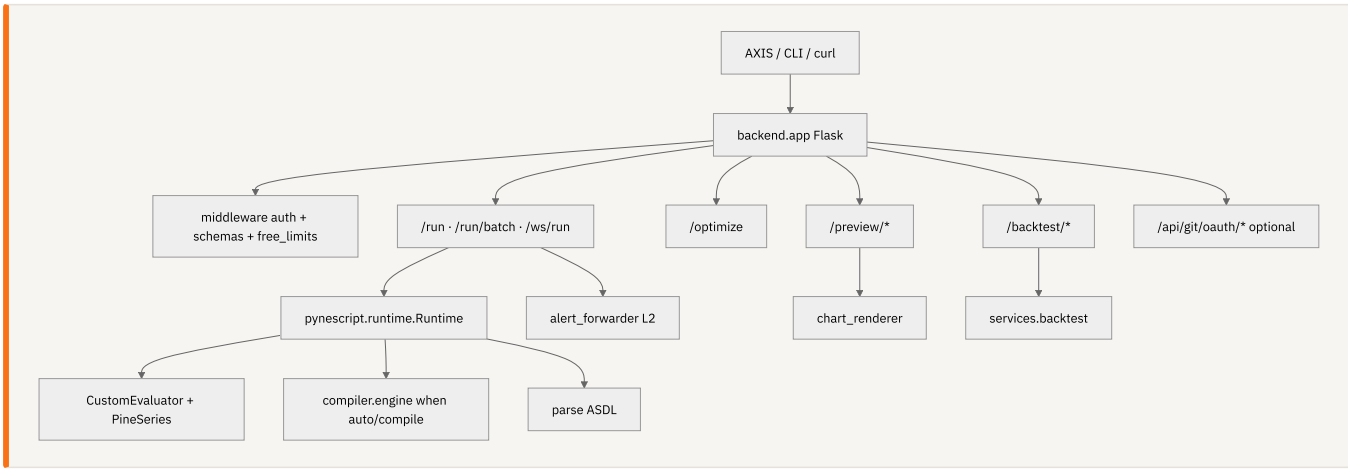
Where the LSP answers *static* editor questions, the Pro API answers *dynamic* evaluate questions:

```
POST JSON { script, data: OHLCV[], mode?, libraries?, webhook_url?, … }
  → pynescript.runtime.Runtime.run  (Pro API re-export; mode default: auto)
  → { plots, series, plot_meta, events, drawings, alerts,
      mode, auto_backend?, compile_fallback_reason?,
      script_id, run_id, alert_forward?, … }
```

Free-tier evaluate (`POST /run` , `POST /run/batch`) validates bodies with hand-rolled schemas, enforces a 5 MB body cap, and restricts CORS. Optional **bar/script/rate/concurrency** guards (`backend/middleware/free_limits.py`) are **off unless** `FREE_TIER_LIMITS` is `1` / `true` / `yes` / `on` . `GET /` and `GET /health` are the same unauthenticated readiness payload (`features.free_tier_limits` reports the switch). Optional **L2 webhooks** POST last-bar `alert()` / `alertcondition()` firings to `webhook_url` or env `ALERT_WEBHOOK_URL` (see Alerts). Pro routes under `/preview/*` and `/backtest/*` use `track_usage` (Bearer / ApiKey / query key). Chart PNGs are matplotlib Agg renders encoded as base64.

The same **evaluate contract** is mirrored by **pyne-worker** so AXIS and HOOX can swap hosts without redesigning payloads — see contract.

Conceptual model



Interface surface

Method	Path	Auth	Role
`GET`	`/` · `/health`	none	Health + endpoint map + compile cache section
`POST`	`/run`	none (free)	Single-script evaluate (`mode` default `auto` ; `libraries[]` ; alerts + optional webhooks)
`POST`	`/run/batch`	none (free)	≤8 scripts, shared OHLCV
`POST`	`/scripts` · `/cron/*`	optional runner	Hosted scripts + bar-close — **off** unless `PYNE_RUNNER=1` (runner)
`POST`	`/optimize`	none (free)	Strategy `input.*` search — N interpret Runtime runs (docs)
`WS`	`/ws/run`	none (free)	Same evaluate contract over JSON frames (`flask-sock` optional)
`POST`	`/preview/chart`	API key + usage	Line / OHLCV PNG
`POST`	`/preview/indicator`	API key + usage	Expression series PNG
`POST`	`/backtest/quick`	API key + usage	Quick strategy metrics + equity PNG
`POST`	`/auth/create_key`	admin decorator	Mint `pyn_…` key (fail-closed `ADMIN_TOKEN`)
`GET`	`/auth/usage`	API key	Usage counters
`POST`	`/auth/validate`	none (body key)	Validate without consuming quota
`POST`	`/compile/prewarm`	none (free, rate-gated)	Product warm-compile — skip cold Numba JIT
`POST`	`/lsp/*`	none (free)	AXIS editor completion / hover / diagnostics
`POST`	`/api/git/oauth/device/*`	none (device flow)	Optional GitHub/GitLab device OAuth for SPA git (blueprint)

`/run` response highlights

Field	Role
`series` / `plots` / `plot_meta`	Bar values + display meta (color, linewidth, `kind` , `fill` band refs)
`events`	Strategy broker events
`drawings`	Drawing registry / compile `__drawings`
`alerts`	Structured `alert()` / `alertcondition()` firings
`mode` / `auto_backend`	Effective path (`interpret` | `compile`)
`compile_fallback_reason`	Why `auto` chose interpret (eligibility or compile error)
`alert_forward`	Present when a webhook URL was configured
`error_kind`	On failure: `parse` | `compile` | `runtime` | `data` | `order` | `mode`

Local runner: `make run` → `python -m backend.app` (default `127.0.0.1:5002`).

Tracks in this tab

1. App lifecycle — Flask app, CORS, limits, blueprints
2. Auth and keys — tiers, stores, decorators
3. Run · Optimize · Preview · Backtest
4. Chart renderer
5. Runtime bridge — `Runtime` / series / evaluator glue
6. Contract — shared evaluate schema across Flask and workers

Internals (map)

Path	Role
`backend/app.py`	App object, `/run*`, `/health`, `/compile/prewarm`, `/auth*`, CORS
`backend/api/preview.py`	Preview + backtest blueprints
`backend/api/git_oauth.py`	Optional device OAuth for SPA git
`backend/api/lsp_http.py`	Free AXIS editor LSP-HTTP
`backend/middleware/*`	Auth, schemas, free_limits, optional Redis/SQLite stores
`src/pynescript/runtime/host.py`	Package SoT bar-loop (`mode=auto` , alerts pack)
`backend/runtime.py`	Compat re-export of the package host
`backend/alert_forwarder.py`	L2 HTTP webhook delivery
`backend/evaluator.py` / `backend/series.py`	Compat shims
`backend/services/*`	Charts + backtest simulation

See also

- Evaluate scripts (end user)
- Pro API usage guide
- Alerts
- Runtime hub
- LSP hub — editor path, not HTTP

APP LIFECYCLE

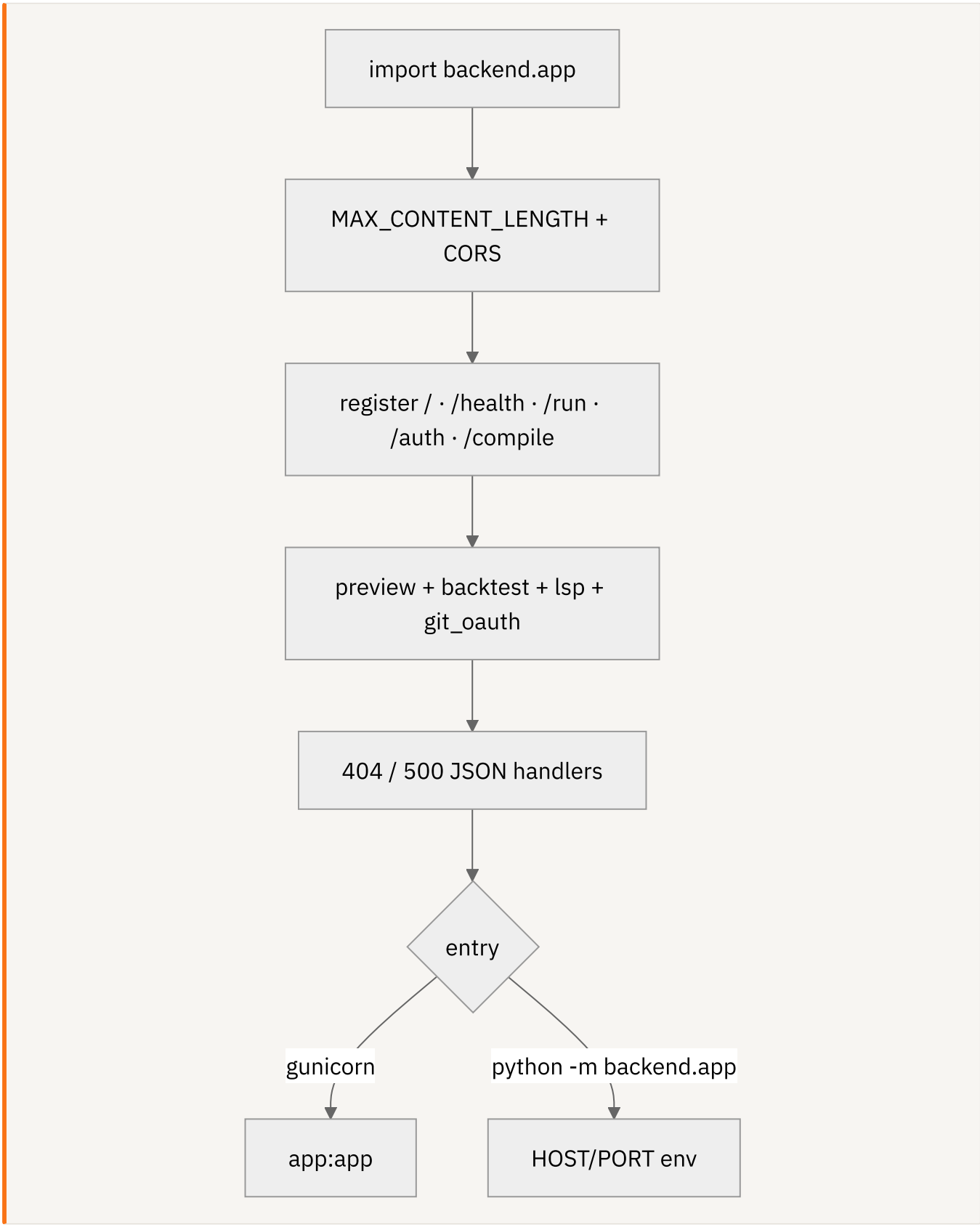
Flask application setup: MAX_CONTENT_LENGTH, CORS policy, blueprints, error handlers, and process entry.

APP LIFECYCLE

Abstract

`backend/app.py` constructs a single global `Flask` app, applies security-minded defaults (body size, CORS allowlist), registers free and Pro routes, and exposes a module `__main__` for local development. There is no multi-app factory pattern today — import `app` for WSGI (gunicorn) or run the module for the built-in server.

Conceptual model



Interface surface

Process entry

```
make run
# equivalent
python -m backend.app
```

Environment:

Variable	Default	Role
`HOST`	`127.0.0.1`	Bind address (localhost-first for dev)
`PORT`	`5002`	Listen port
`ALLOWED_ORIGINS`	`https://pynescript.ai`, `https://app.pynescript.ai`, localhost regex	CORS origins (comma-separated; regex allowed). `*` / `any` opens all. Code **always appends** localhost, private-LAN, and product (`hoox.sh` / `pynescript.ai` / `pynescript-axis.pages.dev`) regexes unless the value is `*`
`API_KEY_STORE`	`/data/api_keys.json`	JSON key-store path (`auth.APIKeyStore` default)
`ADMIN_TOKEN`	unset	Fail-closed admin minting (see auth). When set, hosted-runner writes also require it
`PYNE_RUNNER`	off	Optional hosted script registry + `/cron/run` (runner)
`PYNE_RUNNER_SCHEDULER`	off	In-process bar-close poll (requires `PYNE_RUNNER`)
`PYNE_RUNNER_POLL_SECONDS`	`60`	Scheduler interval
`PYNE_RUNNER_DB`	`/data/runner.db` or temp	SQLite registry path

`debug=False` always on the dev runner.

Health

`GET /` and **`GET /health`** are the same handler. AXIS Settings probes `/health`. Payload includes compile-cache diagnostics:

```
{
  "status": "healthy",
  "service": "pynescript-pro-api",
  "version": "1.0.0",
  "timestamp": 0,
  "websocket": true,
  "features": {
    "alerts": true,
    "alert_webhooks": true,
    "warm_compile": true,
    "default_run_mode": "auto",
    "script_runner": false,
    "script_runner_scheduler": false
  },
  "compile": {
    "has_numba": true,
    "disk_cache_enabled": true,
    "prewarm_enabled": true,
    "default_mode": "auto"
  },
  "endpoints": { "…": "…" }
}
```

`version` here is the **API service** label (`1.0.0`), not the `hoox-pyne` package version.

Hard limits

- **`MAX_CONTENT_LENGTH = 5 * 1024 * 1024`** — reject oversized bodies before JSON parse fills memory (audit 2026-07-05 / S1).
- Free compute is optionally gated by `backend/middleware/free_limits.py` when `FREE_TIER_LIMITS` is truthy (default **off**): max bars (`FREE_MAX_BARS` , 5000), script chars (`FREE_MAX_SCRIPT_CHARS` , 256 KiB), per-IP rate (`FREE_RATE_LIMIT` / `FREE_RATE_WINDOW_SEC` , 60 / 60s), concurrency (`FREE_MAX_CONCURRENT` , 4). Chart/ `mock` data sources only. Production compose sets `FREE_TIER_LIMITS=1` .
- CORS methods: `GET` , `POST` , `OPTIONS` , `HEAD` .
- Allow headers: `Content-Type` , `Authorization` , `X-Admin-Token` , `Accept` .
- `supports_credentials=False` .

Localhost regex used in the default origin list:

```
^https?://(?:localhost|127\.0\.0\.1)(?::\d+)?$
```

Free CORS prefixes (`/` , `/health` , `/run` , `/compile` , `/lsp/` , `/ws/`) reflect the request `Origin` even when it is not in `ALLOWED_ORIGINS` so AXIS VPS UI → local pyne preflight succeeds.

Same-origin / no `Origin` header (curl, server-to-server) remains usable under flask-cors behavior.

Blueprints

```
app.register_blueprint(preview_bp)   # url_prefix=/preview
app.register_blueprint(backtest_bp)  # url_prefix=/backtest
app.register_blueprint(lsp_bp)       # url_prefix=/lsp
app.register_blueprint(git_oauth_bp) # /api/git/oauth/*
```

Preview + backtest live in `backend/api/preview.py`. Optional `flask-sock` registers `WS /ws/run` when installed.

Error handlers

Code	Body code	Message
404	`NOT_FOUND`	`Endpoint {path} not found.`
500	`INTERNAL_ERROR`	`Internal server error.`

Both return JSON with `status: "error"`.

Internals

Path	Role
`backend/app.py`	App object, free routes, auth routes, CORS, health
`backend/api/preview.py`	Pro preview + backtest blueprints
`backend/api/lsp_http.py`	Free AXIS LSP-HTTP
`backend/middleware/schemas.py`	Request validation for `/run*` and auth
`backend/middleware/free_limits.py`	Optional bar / script / rate / concurrency caps (`FREE_TIER_LIMITS`)
`backend/requirements.txt`	Flask, flask-cors, numpy, matplotlib, ...

Recommended production: gunicorn/uvicorn-class WSGI with multiple workers; for multi-worker key consistency prefer SQLite/Redis stores over the default in-memory/file singleton assumptions — see auth.

Invariants and edge cases

1. **Import side effects:** creating `app` configures CORS immediately from env.
2. **Body too large** → Flask 413 before route logic (not custom JSON).
3. **Unknown fields** on schema-validated routes → `UNKNOWN_FIELDS` 400 (`/run` , auth); preview routes currently use looser `get_json()` (schemas exist but are not always applied in handlers).
4. **Dev bind default is loopback** — set `HOST=0.0.0.0` only when intentional.

Worked example — smoke test

```
curl -s http://127.0.0.1:5002/health | jq '.status, .features.default_run_mode'
# "healthy"
# "auto"
```

Failure modes

Symptom	Cause
Browser CORS errors	Origin not in `ALLOWED_ORIGINS`
413 on large OHLCV	Exceeded 5 MB — downsample or paginate
Import errors for matplotlib	Missing backend deps — `pip install -r backend/requirements.txt`

See also

- Auth and keys
- Run endpoint
- Docker / CI

AUTH AND KEYS

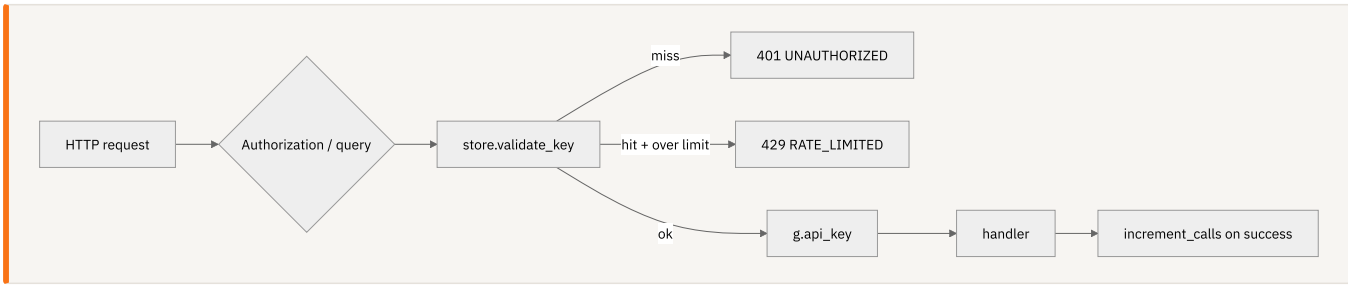
API key model, tiers, require_api_key / track_usage decorators, JSON/SQLite/Redis stores, and admin create_key.

AUTH AND KEYS

Abstract

Pro routes authenticate with a **raw API key** presented as `Authorization: Bearer …`, `Authorization: ApiKey …`, or `?api_key=`. Keys are minted as `pyn_` + url-safe secret, stored with a short `key_id` and SHA-256 hash metadata, and tracked against **monthly-ish call limits** by tier. Free `/run` does not require a key.

Conceptual model



Interface surface

Tiers (`_TIER_LIMITS`)

Tier	`calls_limit`
`free`	`0` (treated as unlimited in `is_rate_limited` — limit 0 means “no cap”)
`hobby`	5 000
`pro`	50 000
`team`	200 000
`enterprise`	`inf`

`calls_remaining()` returns `inf` when limit is 0.

Endpoints

`POST /auth/create_key`

Body schema `CREATE_KEY_SCHEMA`: optional `tier` (default `"hobby"`).

Response:

```
{
  "status": "success",
  "api_key": "pyn_…",
  "key_id": "12hexchars",
  "tier": "hobby",
  "message": "Store this API key securely. It will not be shown again."
}
```

Decorated with `@require_admin_token`:

- Requires non-empty `ADMIN_TOKEN` env (unset → **403**, minting disabled)
- Client must send matching `X-Admin-Token` (or `Authorization: Bearer …` / `AdminToken …`)
- Comparison uses `hmac.compare_digest`

`GET /auth/usage`

`@require_api_key` — returns `key_id`, `tier`, and usage block (`calls_used`, `calls_limit`, `calls_remaining`, timestamps).

`POST /auth/validate`

Body: `{ "api_key": "…" }` — validates without incrementing usage. Returns tier info, `active`, `rate_limited`.

Decorators

Decorator	Behavior
`require_api_key`	Resolve key → 401 / 429 or set `g.api_key`
`track_usage`	Wraps `require_api_key`; increments calls when response status < 400 (tuple responses) or always for bare Response
`require_admin_token`	Fail-closed admin gate (`ADMIN_TOKEN` + `X-Admin-Token`)

Client presentation

```
curl -H "Authorization: Bearer $API_KEY" \
  -H "Content-Type: application/json" \
  -d @body.json \
  http://127.0.0.1:5002/preview/chart
```

Internals

`APIKey` dataclass

Fields: `key_id`, `key_hash`, `tier`, `calls_used`, `calls_limit`, `created_at`, `last_used`.

Helpers: `is_active()` (always True today), `is_rate_limited()`, `get_tier_info()`, `increment_calls()`.

Stores

Module	Use
`middleware/auth.py` → `APIKeyStore` + `get_key_store()`	Facade; selects backend via `STORE_BACKEND`
`middleware/key_store_sqlite.py`	Hash-only rows, WAL, multi-worker friendly
`middleware/key_store_redis.py`	Hash-only Redis; multi-replica

`get_key_store()` backends (`STORE_BACKEND`):

Value	Persistence
`json` (default)	File at `API_KEY_STORE` (default `/data/api_keys.json`); **hash-only** object keys. Legacy files keyed by the raw `pyn_…` secret are migrated on load and the raw secret is dropped
`sqlite`	`API_KEY_STORE_SQLITE` (or `.db` beside JSON path); hash-only
`redis`	`REDIS_URL` required; hash-only

Prod compose defaults to `sqlite` on `/data/api_keys.db` so gunicorn workers share state.

Key minting:

```
raw_key = "pyn_" + secrets.token_urlsafe(32)
key_id  = sha256(raw_key).hexdigest()[:12]
key_hash = sha256(raw_key).hexdigest()
```

Schemas

`CREATE_KEY_SCHEMA`, `VALIDATE_KEY_SCHEMA` in `middleware/schemas.py` — strict types, reject unknown fields.

Invariants and edge cases

1. **`/run` is unauthenticated** — rate-limit at reverse proxy if exposed publicly.
2. **Limit 0 means unlimited**, not “zero calls” — free tier limit value is 0 with that semantics.
3. **Usage increments only after handler returns** — failed auth never bills; handler 4xx under `track_usage` skips increment when returned as `(response, status)`.
4. **Revocation** exists on the store API (`revoke_key`); no public HTTP revoke route in `app.py`.
5. **Admin minting is fail-closed** — unset `ADMIN_TOKEN` or wrong `X-Admin-Token` → 403.

Worked example

```
# create (requires ADMIN_TOKEN env + matching header)

curl -s -X POST http://127.0.0.1:5002/auth/create_key \
  -H "X-Admin-Token: $ADMIN_TOKEN" \
  -H 'Content-Type: application/json' \
  -d '{"tier":"hobby"}'

# validate
curl -s -X POST http://127.0.0.1:5002/auth/validate \
  -H 'Content-Type: application/json' \
  -d '{"api_key":"pyn_…"}'
```

Failure modes

Code	Meaning
`UNAUTHORIZED`	Missing/invalid key
`RATE_LIMITED`	`calls_used >= calls_limit` (finite limits)
`FORBIDDEN`	`ADMIN_TOKEN` unset or `X-Admin-Token` mismatch (fail-closed)
`INVALID_TIER`	create_key with unknown tier string
`INVALID_KEY`	validate endpoint miss

See also

- App lifecycle
- Preview
- Security (DevOps)

POST /RUN

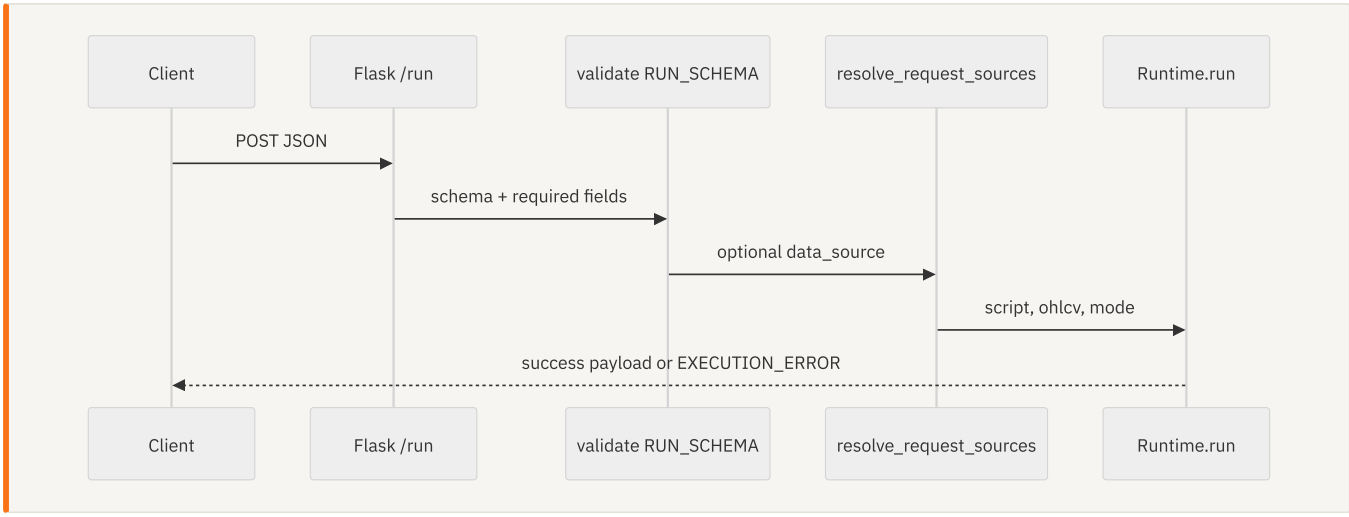
Free evaluate endpoints: single-script /run and multi-script /run/batch over shared OHLCV.

POST `/run` AND `/run/batch`

Abstract

`/run` is the primary free evaluate entry: accept Pine source + bar list, optionally wire `request.*` data sources, execute via package **`pynescript.runtime.Runtime`** (Pro API uses `backend.runtime` re-export), and return plots/series/events/drawings/**alerts**. `/run/batch` runs up to eight scripts on the **same** OHLCV (AXIS multi-indicator), isolating per-script errors so one failure does not discard siblings. Optional **free-tier guards** (`FREE_TIER_LIMITS`, default off) enforce bar/script caps, rate limit, concurrency, and chart/mock-only data (`backend/middleware/free_limits.py`). Optional **L2 webhooks** POST last-bar (default) `alert()` / `alertcondition()` firings to `webhook_url` or env `ALERT_WEBHOOK_URL` (SSRF-safe).

Conceptual model



Interface surface

`POST /run`

Request (`RUN_SCHEMA`)

Field	Type	Required	Default	Notes
`script`	string	yes		Pine source
`data`	list	yes		OHLCV bar objects
`symbol`	string	no	`"CHART"`	Syminfo / request wiring
`data_source`	string	no	`""`	`mock` / `ccxt` / `yahoo` / …
`data_options`	object	no	`{}`	Provider options
`mode`	string	no	`"auto"`	`"interpret"` | `"compile"` | `"auto"`
`inputs`	object	no	`{}`	Pine `input.*` overrides by title
`profiler`	bool	no	`false`	Per-line timing on interpret
`webhook_url`	string	no	`""`	L2 alert webhook (overrides `ALERT_WEBHOOK_URL`)
`forward_alerts`	bool	no	`true`	When false, skip outbound webhook
`alert_last_bar`	bool	no	`true`	Webhook only last OHLCV bar firings
`alert_batch`	bool	no	`true`	One batch POST vs per-alert
`libraries`	list	no	`[]`	`[{namespace, name, version, source}]` — AXIS git-publish emulator (max 32)
`timeout_seconds`	number	no	omit / `null`	Optional interpret wall-clock budget. Passed to `Runtime.run` only when set and `> 0`. Omit = no timeout.

Query `?mode=` is accepted when the body omits `mode`.

Unknown fields → `UNKNOWN_FIELDS` 400. Empty script/data still checked after schema with `NO_SCRIPT` / `NO_DATA`. `import` and any `request.` token are compile-ineligible; `mode=auto` falls back to interpret and **keeps** `libraries` on that path.

Bar shape (runtime expectation):

```
{ "time": 0, "open": 1, "high": 1, "low": 1, "close": 1, "volume": 1 }
```

Optional `bid` / `ask` per bar update runtime bid/ask state.

Success response

```
{
  "status": "success",
  "plots": [],
  "series": {},
  "plot_meta": {},
  "events": [],
  "drawings": [],
  "alerts": [],
  "script_id": "16hex",
  "run_id": "16hex",
  "count": 0,
  "mode": "compile",
  "auto_backend": "compile",
  "data_source": "chart",
  "alert_forward": {
    "forwarded": 0,
    "failed": 0,
    "filter": "last_bar",
    "url": "https://hooks.example.com/pine",
    "batch": true
  }
}
```

Field	Meaning
`plots`	Primary series (first plot) — backward compatible list
`series`	Named multi-plot map title → values per bar
`plot_meta`	Per-title color, linewidth, index
`events`	Strategy events with `script_id` / `run_id` stamps
`drawings`	Exported line/label/box registry for AXIS
`alerts`	`alert()` / true `alertcondition()` firings (interpret; empty on compile)
`alert_conditions`	Optional full condition evaluations (when present)
`alert_forward`	Present when a webhook URL was configured; delivery summary
`script_id`	`sha256(source)[:16]`
`run_id`	Per-`Runtime` instance id
`auto_backend`	`compile` | `interpret` when requested `mode` was `auto`
`compile_fallback_reason`	Why `auto` used interpret (eligibility or compile error)
`object_mode` / `compile_cached` / `compile_ms`	Compile diagnostics when that path ran
`nopython_fallback_reason`	Numeric JIT failed; engine re-emitted object mode (still compile)

Errors

HTTP	code	When
400	`MISSING_FIELD` / `INVALID_FIELD` / `UNKNOWN_FIELDS` / `INVALID_BODY`	Schema
400	`NO_SCRIPT` / `NO_DATA`	Empty after defaults
400	`DATA_SOURCE_ERROR`	`resolve_request_sources` failed
400	`WEBHOOK_URL_BLOCKED`	`webhook_url` is private/loopback/metadata (SSRF denylist)
403	`DATA_SOURCE_FORBIDDEN`	Free path rejected live provider (`ccxt` / `yahoo` / …) — only when `FREE_TIER_LIMITS` is on
413	`TOO_MANY_BARS` / `SCRIPT_TOO_LARGE`	Exceeded `FREE_MAX_BARS` (5000) or `FREE_MAX_SCRIPT_CHARS` (256 KiB) — only when `FREE_TIER_LIMITS` is on
429	`RATE_LIMITED` / `TOO_MANY_REQUESTS`	Free IP rate (60 / 60s) or concurrency (4) — only when `FREE_TIER_LIMITS` is on
500	`EXECUTION_ERROR`	Runtime returned `error` key; body may include `error_kind` / `error_type` / `error_bar`

`POST /run/batch`

Request (`RUN_BATCH_SCHEMA`)

Field	Type	Required	Notes
`scripts`	list	yes	strings or `{id, script}` objects
`data`	list	yes	Shared OHLCV
`symbol` / `data_source` / `data_options` / `mode` / `profiler`	optional	same as `/run` (no `inputs`)	
`libraries`	list	no	same AXIS git-publish list as `/run` (max 32; applied to every script)
`timeout_seconds`	number	no	same optional interpret budget as `/run` (applied to every script)
`webhook_url` / `forward_alerts` / `alert_last_bar` / `alert_batch`	optional	same L2 webhook fields as `/run` (applied per script result)	

Hard cap: `RUN_BATCH_MAX_SCRIPTS = 8` → `TOO_MANY_SCRIPTS` .

Response

```
{
  "status": "success" | "partial",
  "results": [ { "id": "…", "status": "success|error", … } ],
  "count": 2,
  "ok": 1,
  "data_source": "chart"
}
```

HTTP **200** for partial success (per-script errors inside `results`). Envelope validation failures still 400.

Internals

```
runtime = Runtime(symbol=str(symbol))
result = runtime.run(script, ohlcv, data_feed=…, data_provider=…, mode=…)
```

See runtime bridge for bar-loop details and compile mode.

Paths: `backend/app.py` (`run_pine_script`, `run_pine_script_batch`, `compile_prewarm`, `WS /ws/run`), `backend/middleware/schemas.py`, `backend/middleware/free_limits.py`, `backend/alert_forwarder.py`.

Invariants and edge cases

1. **No API key** on these routes — free-tier guards apply only when `FREE_TIER_LIMITS` is on (see Pro API usage); protect further at the edge if needed.
2. **Batch isolates exceptions** per script (`try/except` around `runtime.run`).
3. **Whitespace-only `data_source`** coerced to `None` → chart default; free paths reject live providers (ccxt/yahoo/…) when `FREE_TIER_LIMITS` is on.
4. **Compile mode** may omit alert side effects (`alerts: []`); use interpret/`auto` for alerts.
5. Schema rejects extras — clients must not send AXIS-only fields without updating schema.
6. **Webhook delivery is best-effort** — evaluate still returns `status: success` if the hook fails; check `alert_forward`.
7. Structured errors may include `error_kind` (`parse|compile|runtime|data|order|mode`), `error_type`, `error_bar`. `timed_out` is set on the HTTP body when Runtime exceeds `timeout_seconds`.

Worked example

```
curl -s http://127.0.0.1:5002/run \
  -H 'Content-Type: application/json' \
  -d '{
    "script": "//@version=6\nindicator(\"t\")\nplot(close)",
    "data": [
      {"time": 1, "open": 1, "high": 2, "low": 0.5, "close": 1.5, "volume": 10},
      {"time": 2, "open": 1.5, "high": 2.5, "low": 1.0, "close": 2.0, "volume": 12}
    ],
    "symbol": "TEST:DEMO"
  }'
```

Failure modes

Symptom	Cause
`UNKNOWN_FIELDS`	Typo'd property (e.g. `ohlcv` instead of `data`)
Parse errors in message	Invalid Pine — same engine as CLI
Batch `status: partial`	At least one script failed; inspect `results[i].message`

`POST /compile/prewarm`

Free readiness hook (same rate/concurrency gates). Body optional:

```
{ "scripts": ["//@version=6\nindicator(\"x\")\nplot(close)"], "force": false }
```

Caps at 16 sources. Returns `has_numba`, `builtins_warmed`, `scripts_ok` / `scripts_failed`, `prewarm_ms`. Does **not** execute on OHLCV.

`WS /ws/run`

When `flask-sock` is installed: JSON text frames `{type: "run", id, script, data, mode?}`. Reply `{type: "result", …}` or `{type: "error"}`. Ping/pong supported. Same `execute_run_payload` as HTTP `/run`.

Alert webhooks (L2)

```
# Server default (optional)

curl -s http://127.0.0.1:5002/run \
  -H 'Content-Type: application/json' \
  -d '{
    "script": "//@version=6\nindicator(\"a\")\nif bar_index == last_bar_index\n    alert(\"fire\")
    "mode": "interpret",
    "webhook_url": "https://hooks.example.com/pine",
    "data": [
      {"time": 1, "open": 1, "high": 2, "low": 0.5, "close": 1.5, "volume": 10},
      {"time": 2, "open": 1.5, "high": 2.5, "low": 1.0, "close": 2.0, "volume": 12}
    ]
  }'
```

Full engine rules and batch payload shape: Alerts.

See also

- Alerts
- Contract
- Runtime bridge
- App lifecycle

HOSTED RUNNER

Optional Flask/VPS/container script registry + bar-close scheduler (PYNE_RUNNER). Off by default.

HOSTED RUNNER

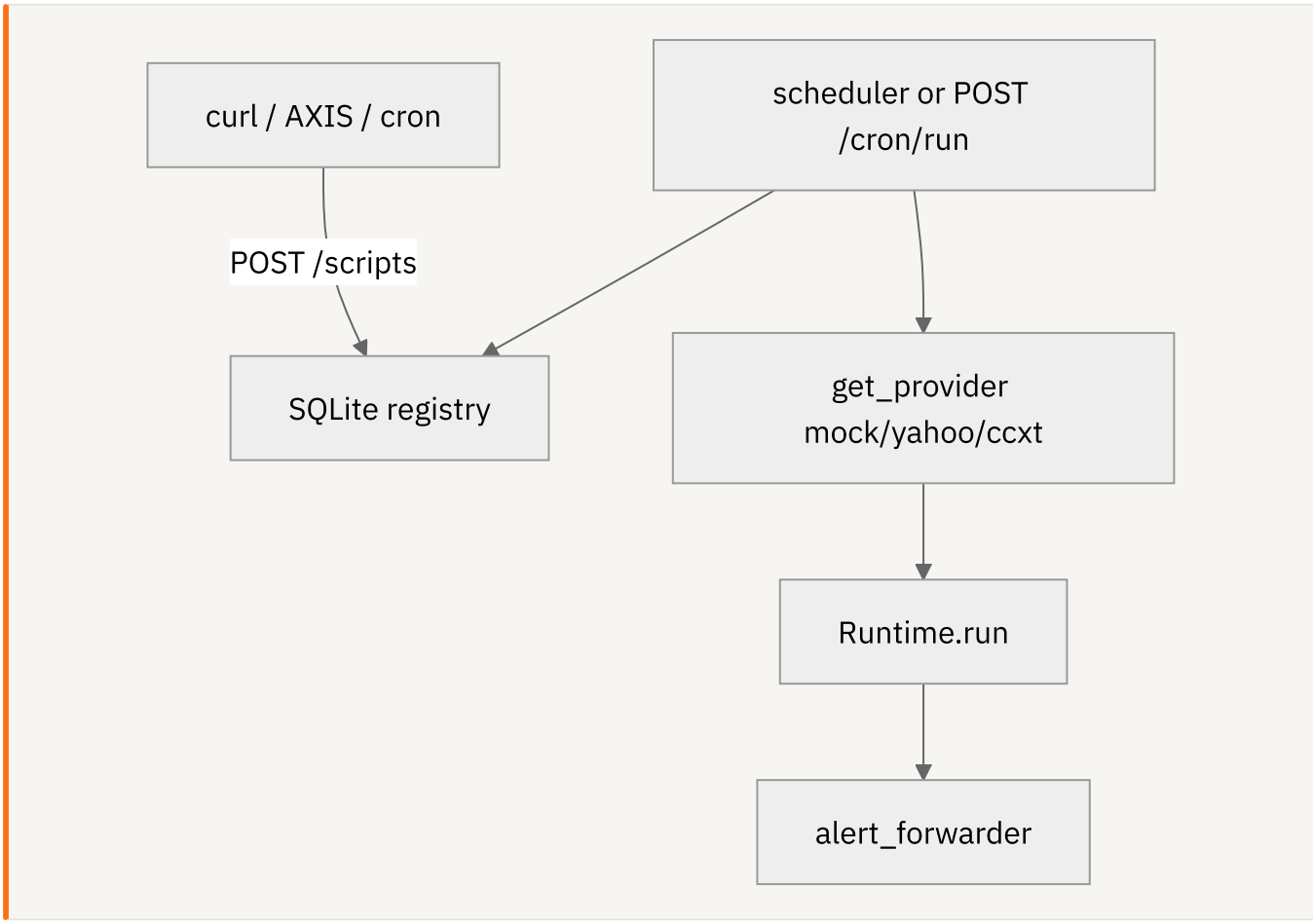
Abstract

The Pro API is request/response: AXIS (or curl) POSTs bars, gets `alerts[]` in that reply, and nothing runs after the client disconnects. **`PYNE_RUNNER`** is an optional Flask-side loop that matches pyne-worker's "upload then keep going" model on **local, Docker, VPS, and Cloudflare Containers**.

Off unless `PYNE_RUNNER=1` (or `true` / `yes` / `on`). Routes then 404 with `RUNNER_DISABLED`. Health reports `features.script_runner`.

This is **not** HOOX order routing. Alerts leave via the same L2 webhooks as `POST /run`. Strategy events stay on the tick JSON unless you forward them yourself.

Conceptual model



Each tick fetches OHLCV for the script's `symbol` / `timeframe` / `data_source`. If last bar time did not advance, the job is skipped. Otherwise a full `Runtime.run` (same evaluate contract as `/run`) and last-bar alert webhooks.

Interface surface

Environment

Variable	Default	Role
`PYNE_RUNNER`	off	Master switch — HTTP registry + `/cron/run`
`PYNE_RUNNER_SCHEDULER`	off	In-process poll loop (needs `PYNE_RUNNER`)
`PYNE_RUNNER_POLL_SECONDS`	`60`	Poll interval (floor 5s)
`PYNE_RUNNER_DB`	`/data/runner.db` if `/data` exists	SQLite file
`ADMIN_TOKEN`	unset	When set, writes require `X-Admin-Token`

Gunicorn multi-worker: every worker may start a scheduler thread; ticks take a flock on `PYNE_RUNNER_DB.lock`. Prefer `GUNICORN_WORKERS=1` when the scheduler is on.

Routes (`PYNE_RUNNER=1`)

Method	Path	Auth	Role
`POST`	`/scripts`	admin if `ADMIN_TOKEN`	Deploy / replace
`GET`	`/scripts`	none	List (no source)
`GET`	`/scripts/:id`	none	Source + cron state
`DELETE`	`/scripts/:id`	admin if `ADMIN_TOKEN`	Remove
`GET`	`/cron/jobs`	none	Scripts + last tick
`PUT`	`/cron/jobs`	admin if `ADMIN_TOKEN`	`{jobs:[{script_id, enabled}]}`
`POST`	`/cron/run`	admin if `ADMIN_TOKEN`	Tick now (`force`, optional `script_id`)

Deploy body


```
{
  "id": "btc-sma",
  "script": "//@version=6\nindicator(\"t\")\nplot(close)\nalertcondition(close>open,\"up\",\"greer
  "symbol": "BTCUSDT",
  "timeframe": "1d",
  "mode": "interpret",
  "data_source": "mock",
  "period": "6mo",
  "max_bars": 500,
  "enabled": true,
  "webhook_url": "https://hooks.example.com/pine",
  "forward_alerts": true
}
```


`data_source`: `mock` (deterministic, no network), `yahoo`, `ccxt` (`data_options.exchange`), `alphavantage`. Same provider factory as CLI `pyne data`.

Worked examples

Local (no Docker):

```
python -m backend.app   # :5002
pyne runner deploy script.pine --id demo --symbol BTCUSDT --source mock
pyne runner tick --id demo --force
pyne runner list
```

Same HTTP without the CLI:

```
curl -sS -X POST http://127.0.0.1:5002/scripts -H 'Content-Type: application/json' \
  -d '{"id":"demo","script":"//@version=6\nindicator(\"t\")\nplot(close)","data_source":"mock","ti
curl -sS -X POST http://127.0.0.1:5002/cron/run -d '{"force":true}' -H 'Content-Type: application/
```

Docker:

```
PYNE_RUNNER=1 PYNE_RUNNER_SCHEDULER=1 docker compose up --build api
```

VPS: set the same env on `pynescript-api.service` (or the process environment `scripts/deploy_vps.sh` restarts). SQLite path should sit on a persistent disk (`PYNE_RUNNER_DB=/data/runner.db`).

Cloudflare Container: keep `PYNE_RUNNER=0` in `cf/src/index.ts` unless you want the runner inside the isolate; one gunicorn worker is already the default.

Failure modes

Symptom	Cause	Fix
`RUNNER_DISABLED` 404	Switch off	`PYNE_RUNNER=1`
`FORBIDDEN` 403	`ADMIN_TOKEN` set, header missing	`X-Admin-Token`
`skipped` / `no_new_bar`	Last close unchanged	Wait, or `force: true`
`fetch_failed`	yahoo/ccxt error	Use `mock` or fix `data_options`
Double ticks	Several gunicorn workers	`GUNICORN_WORKERS=1` or rely on flock

See also

- POST /run
- Alerts
- pyne-worker cron — edge equivalent
- Docker
- App lifecycle

POST /OPTIMIZE

Strategy hyperparameter search — N interpret Runtime runs with input.* overrides, TPE/random/grid, holdout or walk-forward.

POST `/optimize`

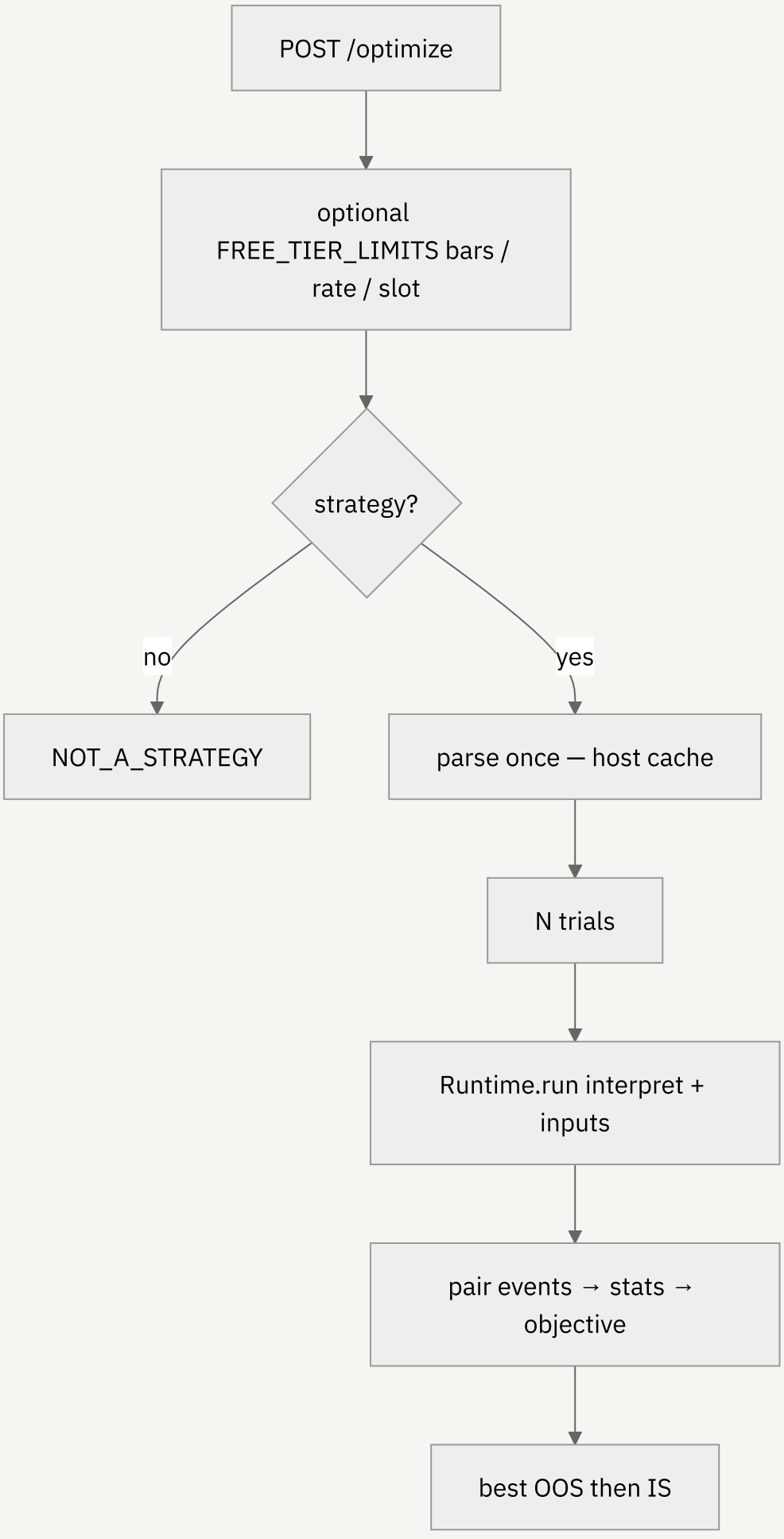
Abstract

`/optimize` searches Pine `input.*` values for a `strategy()` script. It loops the package **Runtime** (same interpret path as `/run`) with different overrides, scores closed trades, and returns the best assignment plus every trial.

This is **not** `/backtest/quick` (that route is a demo SMA simulator). It is **not** a Pine language surface — there is no `optimize.*` builtin.

AXIS Hyperparameter Optimisation calls this when the active engine is the Pro API (`server`). Pyodide / edge engines fall back to a client-side loop.

Conceptual model


POST /optimize
optional FREE_TIER_LIMITS bars / rate / slot
strategy?
no
yes
NOT_A_STRATEGY
parse once — host cache
N trials
Runtime.run interpret + inputs
pair events → stats → objective
best OOS then IS

Request

Field	Type	Required	Default	Notes
`script`	string	yes		Must declare `strategy(`
`data`	list	yes		Same OHLCV objects as `/run`
`space`	object	yes		`{ params: [{ name, kind, min, max, step?, choices? }] }`
`n_trials`	int	no	30	Capped at 200
`sampler`	string	no	`auto`	`auto` | `random` | `tpe` | `grid` (`auto` → random if N<20 else TPE)
`objective`	string	no	`composite`	`net_pnl` | `profit_factor` | `calmar` | `composite`
`validation`	object	no	`{ mode: "holdout" }`	`holdout` | `walk-forward` | `in-sample`
`min_trades`	int	no	5	Trials below this score `-inf`
`seed`	int	no		RNG seed
`symbol`	string	no	`CHART`	
`oos_every_trial`	bool	no	true	Holdout: score every trial on the test slice
`fixed_inputs`	object	no	`{}`	Current `input.*` values for names **not** in `space` (trial params win on overlap)
`libraries`	list	no	`[]`	Same AXIS git-publish list as `/run`

`space.params[].kind`: `int` | `float` | `bool` | `categorical`. Numeric axes **require** `min` and `max`.

`validation` extras: `holdout_frac` (default 0.3), `train_bars` / `test_bars` / `step_bars` for walk-forward.

Estimated engine runs are capped at 400 (`n_trials * folds`).

Success


```
{
  "status": "success",
  "sampler": "tpe",
  "objective": "composite",
  "validation": { "mode": "holdout", "holdout_frac": 0.3, "train_bars": 200, "test_bars": 50, "ste
  "n_trials": 30,
  "trials": [
    {
      "index": 0,
      "params": { "Fast": 8 },
      "is_score": 12.4,
      "oos_score": 3.1,
      "is_stats": { "total_pnl": 10, "trades": 8, "max_dd": 0.1 },
      "oos_stats": { "total_pnl": 3, "trades": 6, "max_dd": 0.08 },
      "error": null,
      "engine_runs": 2,
      "ms": 210.4
    }
  ],
  "best_index": 4,
  "best_params": { "Fast": 6 },
  "best_is_score": 15.0,
  "best_oos_score": 4.2,
  "engine_runs": 60,
  "ms": 8421.2,
  "warning": null
}
```


Full event arrays are **not** returned (memory). Re-run the winner via `/run` + `inputs`. Success includes `validation` and per-trial `oos_stats` / `engine_runs` / `ms`.

Errors

HTTP	code	When
400	`MISSING_FIELD` / `INVALID_FIELD` / `UNKNOWN_FIELDS` / `INVALID_BODY`	Schema — empty/`{}` body, missing required fields, wrong types, extras
400	`NO_SCRIPT` / `NO_DATA`	`script` / `data` present but empty after schema
400	`NOT_A_STRATEGY`	Script is not `strategy()`
400	`INVALID_SPACE`	Missing bounds / empty params
400	`INVALID_VALIDATION`	`validation.mode` not `holdout` | `walk-forward` | `in-sample`
400	`TOO_MANY_RUNS`	Estimated runs exceed 400
400	`GRID_TOO_LARGE`	Grid sampler product exceeds 200 cells
413 / 429	same as `/run`	Free-tier caps — only when `FREE_TIER_LIMITS` is on

Invariants

1. Always **interpret** (input overrides skip compile).
2. Train and test are **bar slices**, not trade filters on a full-sample run.
3. In-sample mode is allowed and **warned**.
4. Ranking prefers **OOS** score when validation is holdout or walk-forward.

CLI

```
pyne optimize strat.pine --trials 20 --sampler tpe \
  --space '{"params":[{"name":"Fast","kind":"int","min":3,"max":12}]}'
# walk-forward: --validation walk-forward  (derives train/test from --bars)
# override: --train-bars 80 --test-bars 20 --step-bars 20
```

See also

- POST /run
- Runtime bridge
- AXIS end-user guide: Hyperparameter Optimisation

PREVIEW ENDPOINTS

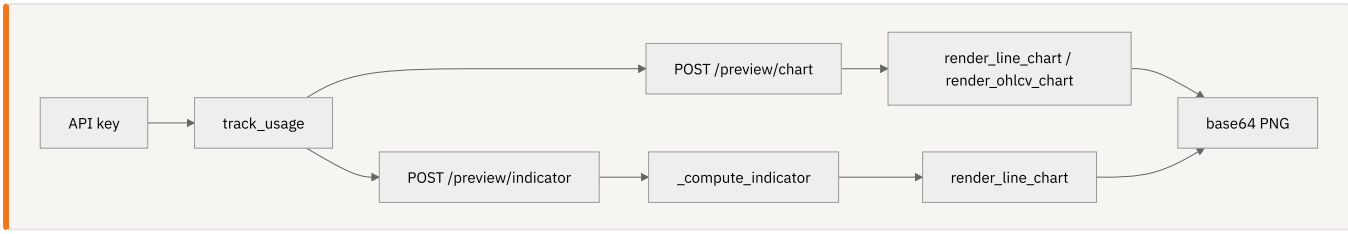
Pro chart and indicator thumbnail routes under /preview with usage tracking.

PREVIEW ENDPOINTS

Abstract

Preview routes render **PNG thumbnails** (base64) for desk UIs and docs. They require an API key via `@track_usage`, accept OHLCV-shaped **objects** (column arrays), and do **not** run the full `Runtime` bar loop for `/preview/chart` (data is plotted directly). `/preview/indicator` evaluates a small set of closed-form `ta.*` expressions in pure Python for the series, then plots.

Conceptual model



Blueprint: `preview_bp` with `url_prefix="/preview"` in `backend/api/preview.py`.

Interface surface

Auth

Both routes: `@track_usage` → must pass `require_api_key` and consume one call on success.

`POST /preview/chart`

Body (documented; schemas exist as `PREVIEW_CHART_SCHEMA`)

```
{
  "script": "optional string",
  "data": {
    "open": [], "high": [], "low": [], "close": [], "volume": []
  },
  "options": {
    "type": "line",
    "color": "#2196F3",
    "show_volume": false,
    "width": 600,
    "height": 300
  }
}
```

Option	Default	Clamp
`type`	`"line"`	`"ohlcv"` selects candlestick renderer
`color`	`#2196F3`	line chart only
`width`	600	max 1200
`height`	300	max 600
`show_volume`	false	twin-axis volume on line chart

Success

```
{
  "status": "success",
  "chart": "<base64 png>",
  "meta": {
    "type": "line",
    "bars": 252,
    "last_value": 0,
    "first_value": 0,
    "change": 0,
    "change_pct": 0,
    "rendered_at": 0
  },
  "tier_info": {}
}
```

Errors

code	When
`NO_DATA`	Missing `data` (both preview routes)
`NO_CLOSE_DATA`	Empty `close` array (chart **and** indicator)
`RENDER_ERROR`	matplotlib / renderer exception (`/preview/chart` only; indicator parse/render exceptions fall back to plotting `close`)

Note: `script` is accepted for API symmetry but **not executed** on this path today — the chart is pure data visualization.

`POST /preview/indicator`

Body

```
{
  "expression": "ta.sma(close, 14)",
  "data": { "close": [/* … */] },
  "options": { "color": "#FF9800", "width": 600, "height": 300 }
}
```

Supported expression prefixes in `_compute_indicator`:

Expression	Series
`ta.sma(…, period)`	SMA of close
`ta.ema(…, period)`	EMA
`ta.rsi(…, period)`	RSI
`ta.macd(…)`	MACD histogram (12/26/9)
other	falls back to raw `close`

Parse failures in the helper fall back to plotting `close`.

Internals

Path	Role
`backend/api/preview.py`	Route handlers + indicator math
`backend/services/chart_renderer.py`	PNG encode
`backend/middleware/auth.py`	`track_usage`

Schemas `PREVIEW_*` in `schemas.py` document intended validation; current handlers use `request.get_json() or {}` directly.

Invariants and edge cases

1. **Data shape differs from `/run`:** preview uses **columnar dicts**; `/run` uses **list of bar dicts**.
2. **Not a full Pine interpreter** for indicator expressions — string prefix parsing only.
3. **Width/height clamps** protect worker memory/CPU.
4. Successful responses include `tier_info` from the authenticated key.

Worked example

```
curl -s http://127.0.0.1:5002/preview/chart \
  -H "Authorization: Bearer $API_KEY" \
  -H 'Content-Type: application/json' \
  -d '{
    "data": {"close": [1,2,3,4,5], "volume": [10,10,10,10,10]},
    "options": {"type": "line", "show_volume": true}
  }' | jq '.meta'
```

Failure modes

Symptom	Cause
401	Missing key
429	Tier limit
Empty-looking PNG	Renderer empty-chart path when all values null

See also

- Chart renderer
- Auth and keys
- Run

BACKTEST ENDPOINT

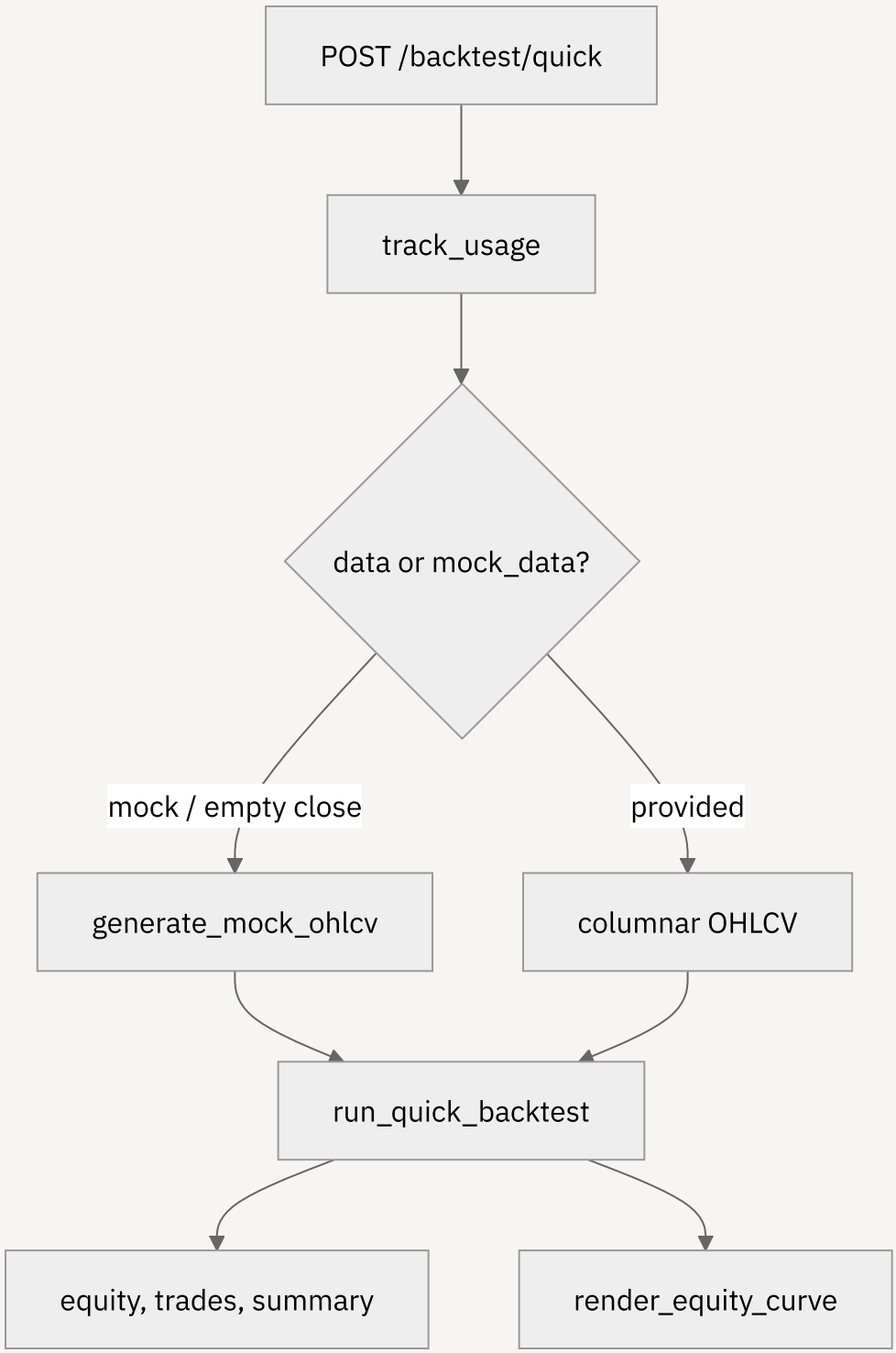
POST /backtest/quick — usage-tracked strategy metrics, equity curve, and optional mock OHLCV.

BACKTEST ENDPOINT

Abstract

`POST /backtest/quick` is a Pro, usage-tracked route that runs a **simplified** strategy simulation over columnar OHLCV (or generated mock bars), returns trade lists + summary metrics, and embeds a base64 equity-curve PNG. It is optimized for speed/demo UX; it is not a full fidelity substitute for event-aware `Runtime` strategy evaluation on `/run`.

Conceptual model



Blueprint: `backtest_bp`, prefix `/backtest`.

Interface surface

Request


```
{
  "script": "//@version=6\nstrategy(\"s\")…",
  "data": {
    "open": [], "high": [], "low": [], "close": [], "volume": []
  },
  "initial_capital": 10000.0,
  "mock_data": false,
  "mock_bars": 252
}
```


Field	Default	Notes
`script`	required	Empty → `NO_SCRIPT`
`data`	`{}`	Columnar; may be omitted if `mock_data`
`initial_capital`	`10000`	Starting equity
`mock_data`	`false`	Force synthetic bars
`mock_bars`	`252`	Length of mock series

If neither usable `close` nor `mock_data` → `NO_DATA`.

Success


```
{
  "status": "success",
  "result": {
    "equity_curve": [10000, …],
    "trades": [
      {
        "entry_time": 0,
        "entry_price": 0,
        "exit_time": 0,
        "exit_price": 0,
        "direction": "long",
        "pnl": 0,
        "pnl_pct": 0,
        "size": 1
      }
    ],
    "summary": {
      "total_pnl": 0,
      "total_pnl_pct": 0,
      "sharpe_ratio": 0,
      "max_drawdown": 0,
      "max_drawdown_pct": 0,
      "win_rate": 0,
      "profit_factor": 0,
      "total_trades": 0,
      "winning_trades": 0,
      "losing_trades": 0,
      "avg_win": 0,
      "avg_loss": 0
    },
    "equity_chart": "<base64 png>"
  },
  "tier_info": {},
  "meta": {
    "bars": 252,
    "initial_capital": 10000,
    "completed_at": 0
  }
}
```


Errors

code	HTTP	When
`NO_SCRIPT`	400	Empty script
`NO_DATA`	400	No data and not mock
`BACKTEST_ERROR`	500	Exception in simulation
`UNAUTHORIZED` / `RATE_LIMITED`	401 / 429	Auth

Internals

`run_quick_backtest` wraps `run_backtest(..., plot_chart=True)` and returns `BacktestResult.to_dict()` (`backend/services/backtest.py`). `avg_bars_in_trade` exists on the dataclass but is **not** exported in `summary`.

MVP simulation characteristics:

1. Optionally `parse(script)` (errors soft-ignored for MVP path).
2. Precompute long/short entry signals from dual SMA cross (10 vs 20) starting at bar 20.
3. Exit heuristics via RSI-like average and opposite signals / end of series.
4. Equity curve + trade PnL with optional commission/slippage parameters on the lower-level API.
5. Metrics: Sharpe (√252 scaling), max drawdown, win rate, profit factor.
6. Chart via `render_equity_curve`.

`generate_mock_ohlcv(n_bars)` produces synthetic columns for demos.

Path	Role
`backend/api/preview.py`	`quick_backtest` route
`backend/services/backtest.py`	Simulation + metrics
`backend/services/chart_renderer.py`	Equity PNG

Invariants and edge cases

1. **Script content is lightly used** in the MVP sim — do not treat results as broker-accurate for arbitrary strategies. Prefer `/run` events for engine-faithful strategy traces.
2. **Columnar data**, same family as preview, not `/run` bar lists.
3. **Mock path** ignores incomplete user data when `mock_data` or empty close.
4. One usage increment per successful HTTP response under `track_usage`.

Worked example

```
curl -s http://127.0.0.1:5002/backtest/quick \
  -H "Authorization: Bearer $API_KEY" \
  -H 'Content-Type: application/json' \
  -d '{
    "script": "//@version=6\nstrategy(\"demo\")\n// body optional for MVP",
    "mock_data": true,
    "mock_bars": 120,
    "initial_capital": 25000
  }' | jq '.result.summary'
```

Failure modes

Symptom	Cause
Implausible trades vs script logic	MVP signal model, not full evaluator
Empty trade list	No crosses in series / short history
Missing `equity_chart`	Renderer exception swallowed → empty string

See also

- Runtime bridge — faithful strategy events via `/run`
- Chart renderer
- Strategy builtins

CHART RENDERER

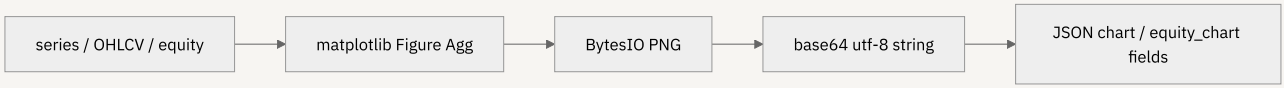
Matplotlib Agg PNG services: line charts, OHLCV candles, equity curves, base64 encoding.

CHART RENDERER

Abstract

`backend/services/chart_renderer.py` is a headless plotting façade. It forces `matplotlib` into **Agg**, draws dark-theme charts for desk thumbnails, and returns **base64-encoded PNG** strings suitable for JSON transport. Callers are the preview and backtest routes; the module has no Flask dependency.

Conceptual model



Interface surface

`render_line_chart`

Param	Default	Role
`values`	required	Y series; `None` → NaN gaps
`dates`	`None`	Optional x tick labels (≤ ~6 ticks)
`title`	`"Chart"`	Title string
`color`	`#2196F3`	Line + fill
`height` / `width`	300 / 600	`width` → figsize inches (`width/100`); **`height` is unused** for figsize — line charts use 3.0″ (3.5″ with volume), OHLCV 3.5″, empty 2.5″. dpi 100
`show_volume`	false	Twin axis bars from `ohlcv["volume"]`
`ohlcv`	`None`	Volume source when enabled

Empty / all-null values → `_render_empty_chart` placeholder.

`render_ohlcv_chart`

Candlestick + volume subplot pair from dict keys `open` , `high` , `low` , `close` , `volume` . Green/red body colors (`#4CAF50` / `#F44336`). Shared x-axis. Figsize `(width/100, 3.5)` — the `height` argument is ignored.

`render_equity_curve`

Single series with profit/loss fill relative to zero baseline and a dashed reference at the first equity point. Title fixed `"Equity Curve"` .

Return type

All public renderers → `str` (base64 PNG without data-URI prefix). Clients typically use:

```
{chart}
```

Internals

Detail	Implementation
Backend	`matplotlib.use("Agg")` before pyplot import
Theme	fig `#1E1E1E`, axes `#252526`, muted grids
Cleanup	`plt.close(fig)` after `savefig`
Empty state	Centered “No data available” text

Path	Role
`backend/services/chart_renderer.py`	Renderers
`backend/api/preview.py`	Consumers
`backend/services/backtest.py`	Equity chart hook

Invariants and edge cases

1. **Not interactive** — no GUI backend; safe for servers.
2. **NaNs** break lines at gaps (numpy nan).
3. **Candle body height** uses `+ 1e-9` to keep zero-range bodies visible.
4. **Thread safety:** matplotlib global state is historically process-local; multi-threaded gunicorn workers should avoid concurrent pyplot use on one worker or serialize renders.
5. Exceptions in callers often become HTTP `RENDER_ERROR` or empty equity chart strings.

Worked example

```
from backend.services.chart_renderer import render_line_chart

b64 = render_line_chart([1.0, 1.2, 1.1, 1.4], title="demo", color="#A3E635")
assert isinstance(b64, str) and len(b64) > 100
```

Failure modes

Symptom	Cause
ImportError Agg	Incomplete matplotlib install
Huge base64	High width/height — routes clamp preview sizes
Blank image	Empty input path succeeded with placeholder

See also

- Preview endpoints
- Backtest

RUNTIME BRIDGE

pynescript.runtime.Runtime — package SoT bar-loop evaluate façade, PineSeries context, compile mode, and result packaging.

RUNTIME BRIDGE

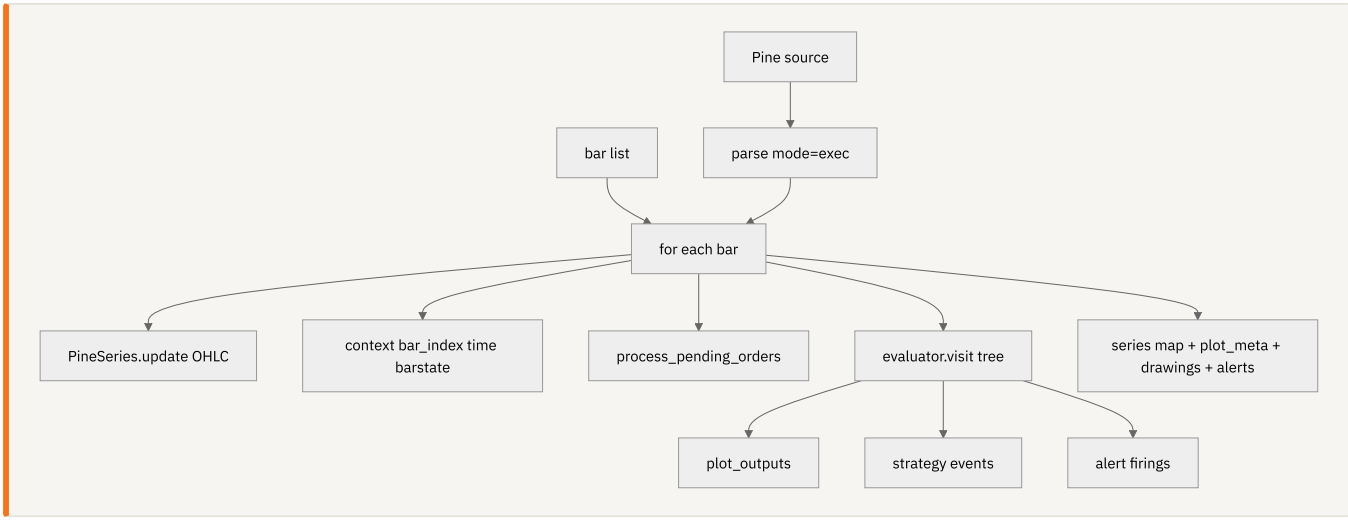
Abstract

Package Runtime SoT: the bar-loop host, series, and `CustomEvaluator` live under **`pynescript.runtime`** (`src/pynescript/runtime/host.py`). The Pro API still imports `backend.runtime.Runtime`, which **re-exports** the package implementation (`sys.modules` alias) so monorepo and wheel installs share one host.

`Runtime` owns symbol metadata namespaces (`syminfo`, `timeframe`, `barstate`, …), walks OHLCV bars, updates `PineSeries` for OHLC, visits a shared `CustomEvaluator` on the parsed AST each bar, drains strategy events, and packages multi-plot series for AXIS. Compile mode optionally swaps the AST walker for a Numba-backed subset engine. Optional **`timeout_seconds`** provides a wall-clock circuit breaker on the interpret path (checked every 32 bars) for edge/cron budgets. Flask `/run` and `/run/batch` accept the same field (omit / `null` / `≤ 0` → no timeout).

This is the bridge between Flask handlers and the language core — not a second semantics.

Conceptual model



Interface surface

Construction

```
Runtime(symbol: str = "AAPL", run_id: str | None = None)
```

- Sets `syminfo.tickerid` / `name` / `prefix` (prefix from `EXCHANGE:SYMBOL` form).
- Generates `run_id` as `uuid4().hex[:16]` when omitted.

Helpers: `configure_footprint`, `update_bid_ask`.

`run(source_code, ohlcv_data, data_feed=None, data_provider=None, mode="interpret", …)`

Arg	Role
`source_code`	Pine text
`ohlcv_data`	`list[dict]` with open/high/low/close/time[/volume/bid/ask]
`data_feed` / `data_provider`	`request.*` wiring; auto-resolved from chart bars when unset
`mode`	`"interpret"` (default for direct calls) · `"compile"` · `"auto"` (Pro API schema default)
`timeout_seconds`	Optional wall-clock budget (seconds); interpret path only — partial results + `timed_out` + `error_kind=runtime` when exceeded. Flask `/run` and `/run/batch` accept it (omit / `≤ 0` = no budget)
`libraries`	Optional `[{namespace, name, version, source}]` registered via `register_library_source` before `import ns/Name/ver`. `mode=auto` forwards libs into interpret fallback. Flask `/run` caps at 32
`inputs` / `profiler`	`input.*` overrides (interpret); non-empty `inputs` or `profiler` force interpret under `auto` (`compile_fallback_reason`)
`realtime_*`	Optional host simulation for `varip` multi-tick tests (`realtime_last_bar`, `realtime_ticks`, `realtime_bars`, `realtime_from_bar`)

Interpret path (default)

1. Resolve request sources (non-fatal on failure).
2. `parse(source_code, mode="exec")` — on failure `{ "error": "Parse Error: …" }`.
3. Init `PineSeries` for OHLC + context dict (timeframe daily defaults, barstate, chart colors).
4. Construct `CustomEvaluator(context=…, data_feed=…, data_provider=…)`, reset var declarations + drawing registry + `clear_alerts()`.
5. Per bar:
   - Update series + calendar fields from timestamp
   - Update barstate flags (`isfirst` / `islast` / history confirmed)
   - `process_pending_orders` when available
   - `evaluator.visit(tree)` — on failure `{ "error": "Runtime Error at bar …" }`
   - Lock pine defs after first bar (`_pine_defs_locked`) to avoid multi-dispatch blow-ups
   - Drain strategy events; stamp `script_id` / `run_id`
   - Capture plot outputs
6. Build `series` / `plot_meta` from all plot indices (disambiguate duplicate titles with `_2` suffixes).
7. Export drawings via `DrawingRegistry.export_for_api(bar_times)`.
8. Export alerts via `export_alerts_from_evaluator` (optional `alert_conditions`).

Return keys: `plots`, `series`, `plot_meta`, `events`, `drawings`, `alerts`, `count`, `script_id`, `run_id`, `mode` (+ optional `alert_conditions`, `timed_out`, `error_kind`).

`mode=auto`

1. Non-empty `inputs` → interpret immediately (`compile_fallback_reason = "input.* overrides require interpret path"`).

2. `_compile_eligible`: reject top-level `import` and any `request.` token (cached per source hash).
3. Eligible → `_run_compiled`. On success set `auto_backend=compile`. On compile/runtime error, fall back to interpret and set `compile_fallback_reason`.
4. Ineligible → interpret + `compile_fallback_reason` (e.g. `"import statements not supported in compile path"`).

Value mismatch never triggers a backend switch — that is the parity harness's job.

Compile path

1. Import `pynescript.compiler.engine` — missing → `error_kind=compile`.
2. **Do not** require `has_numba()`. Object-mode scripts (strategy, UDT, drawings) compile to a Python/numpy loop. Missing Numba only fails pure-numeric emit (`CompileNumbaRequiredError`); `auto` caches that failure.
3. Host LRU by raw-source sha256 → `compile_script` on miss.
4. Pack OHLCV to float64 columns **including** `time=` bar-open ms (`_ohlcv_pack_cached`).
5. `compiled.run(opens, highs, lows, closes, volumes, time=times)`.
6. Lift `__drawings` / `__events`; GC compile drawings by declaration caps; merge visual series (bgcolor / plotshape / …) into titled keys.
7. Envelope: `mode: "compile"`, `alerts: []`, `object_mode`, `compile_cached`, `compile_ms`. `generated_code` only when `PYNESCRIPT_RETURN_GENERATED_CODE=1`.

`PineSeries` (`pynescript.runtime.series`)

History deque (default maxlen **1000**, raised by `max_bars_back` / `PYNE_SERIES_MAX`). `PYNE_SERIES_CAP` default **on** (list-length trim). `PYNE_SERIES_RING` default **off** (chronological O(1) lookback when `1`). `series[0]` current / `series[n]` lookback, arithmetic on **current** values, `None` propagates as na-like absence. Negative indices raise — Pine Script™ does not allow them. `backend.series` re-exports the package type.

Context namespaces

Defined on the host: `Syminfo`, `Chartinfo`, `Timeframe`, `Barstate`, `Chart` — attribute names aim at Pine Script™ v5–v6 surface (`timeframe.isdaily`, `syminfo.isin`, …).

Internals

Path	Role
`src/pynescript/runtime/host.py`	Package SoT `Runtime` bar loop
`src/pynescript/runtime/evaluator.py`	`CustomEvaluator` specialization
`src/pynescript/runtime/series.py`	`PineSeries`
`backend/runtime.py`	Compat shim → `pynescript.runtime`
`backend/evaluator.py` / `backend/series.py`	Compat shims
`src/pynescript/ast/helper.py`	`parse`
`src/pynescript/ast/evaluator/**`	Builtin + strategy semantics
`src/pynescript/compiler/engine.py`	Compile mode

Flask wiring: `backend/app.py` constructs a **fresh** `Runtime` per request (and per batch job) so `run_id` and drawing registries do not leak across users.

Invariants and edge cases

1. **Parse once, visit many** — AST is not re-parsed per bar.
2. **Defs locked after bar 0** — performance invariant for large function tables.
3. **Drawing registry reset** at run start — isolation between requests.
4. **Primary `plots` list** is plot index 0 for backward compatibility; AXIS should prefer `series` + `plot_meta`.
5. **Errors are dicts**, not exceptions, for control-flow at the HTTP boundary (`"error" in result`).
6. Compile mode supports a **subset**; `mode=auto` falls back to interpret with `compile_fallback_reason` rather than failing the request.

Worked example

```
from pynescript.runtime import Runtime
# or: from backend.runtime import Runtime  # Pro API monorepo path

rt = Runtime(symbol="NASDAQ:AAPL")
out = rt.run(
    '//@version=6\nindicator("x")\nplot(close, "C")',
    [{"time": i, "open": 1, "high": 2, "low": 0.5, "close": 1.0 + i * 0.01, "volume": 1} for i in
)
assert "error" not in out
assert out["count"] == 50
assert "C" in out["series"] or "plot_0" in out["series"]
```

Failure modes

Message prefix	Cause
`Parse Error:`	Grammar / syntax
`Runtime Error at bar`	Evaluator exception
`Order fill error at bar`	Broker sim pending orders
`Compile mode requires numba` / `CompileNumbaRequiredError`	Numeric emit without Numba (object-mode scripts still run)
`Compile Error:` / `Compiled Runtime Error:`	Engine path (`error_kind=compile` / `runtime`)
`Script execution timed out`	`timeout_seconds` exceeded (`timed_out`, `error_kind=runtime`)

See also

- Contract
- Run endpoint
- Series and history

- Strategy events

EVALUATE CONTRACT

Shared request/response evaluate contract across Flask Pro API, pyne-worker, and in-process Runtime.run (Python + PyneTS).

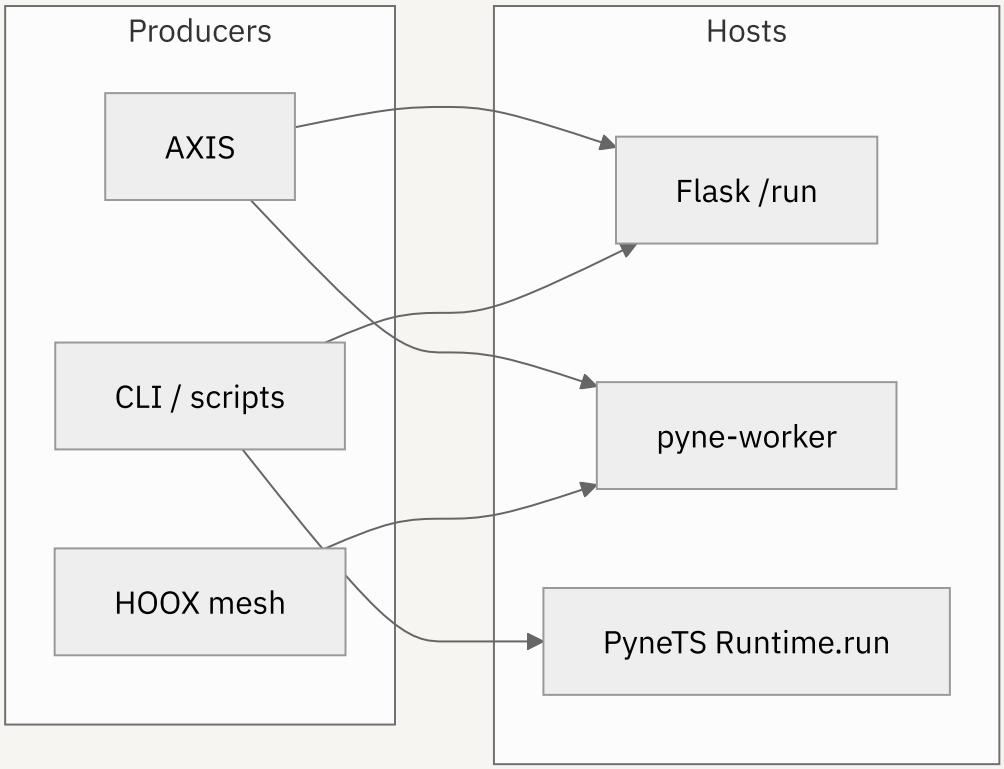
EVALUATE CONTRACT

Abstract

The **evaluate contract** is the JSON dialect that lets AXIS, HOOX, CLI tools, and edge workers call *any* PYNE-compatible host interchangeably. Flask `POST /run` is the reference implementation; Python **pyne-worker** aims for the same HTTP fields. **PyneTS** speaks the in-process form (`RuntimeResult`).

This page freezes the semantic envelope — not transport auth headers.

Conceptual model



Invariant: given the same script + OHLCV + mode, hosts should agree on plot series and strategy events within known parity limits (see numerical validation and PyneTS parity). AXIS engines call Python hosts today, not PyneTS.

Interface surface

Request (single evaluate)

Field	Type	Required	Description
`script`	string	yes	Full Pine source
`data`	`Bar[]`	yes	Chronological OHLCV (`ohlcv` alias on pyne-worker)
`symbol`	string	no	Default `CHART` / host-specific
`mode`	`"interpret"` | `"compile"` | `"auto"`	no	`Runtime.run` default when omitted: `PYNE_RUNTIME_MODE` or `interpret` . Flask `RUN_SCHEMA` default **`auto`**
`inputs`	object	no	`input.*` overrides keyed by title (forces interpret under `auto`)
`libraries`	`{namespace, name, version, source}[]`	no	In-process `import ns/Name/ver` . Flask `/run` and `/run/batch` accept this (max 32).
`profiler`	bool	no	Per-line interpret timings (forces interpret)
`timeout_seconds`	number	no	Optional wall-clock interpret budget (checked every 32 bars; sets `timed_out`). Flask `/run` and `/run/batch` accept it; omit / `null` / `≤ 0` → no timeout.
`webhook_url`	string	no	Flask L2 alert webhook (overrides `ALERT_WEBHOOK_URL`)
`data_source`	string	no	Optional external history provider id (free paths: chart / mock only)
`data_options`	object	no	Provider configuration

`Bar`

Field	Type	Required	Notes
`time`	number	recommended	Unix ms preferred
`open`	number	yes	
`high`	number	yes	
`low`	number	yes	
`close`	number	yes	
`volume`	number	no	Default host-dependent
`bid` / `ask`	number	no	Runtime bid/ask hooks

Response (success)

Field	Type	Description
`status`	`"success"`	Flask sets this; workers may omit and use HTTP 200 only — **prefer including it**
`plots`	`(number|null)[]`	Primary series (plot 0)
`series`	`Record<string, (number|null)[]>`	Named multi-series
`plot_meta`	`Record<string, PlotMeta>`	Display metadata
`events`	`StrategyEvent[]`	Strategy / trade-like events
`drawings`	`Drawing[]`	line/label/box export
`alerts`	`AlertEvent[]`	`alert()` / true `alertcondition()` firings (interpret; `[]` on compile)
`alert_conditions`	object[]	optional condition evaluations
`alert_forward`	object	optional L2 webhook delivery summary
`count`	number	Bars evaluated
`script_id`	string	Stable hash prefix of source
`run_id`	string	Per-invocation id
`mode`	string	Effective mode (`interpret` | `compile`)
`auto_backend`	string	`auto` only: which path ran
`compile_fallback_reason`	string	`auto` only: why compile was skipped or failed
`object_mode`	bool	Compile path: pure-Python bar loop (no njit)
`timed_out`	bool	Interpret budget exceeded (`Runtime.run(timeout_seconds=…)`)
`data_source`	string	Flask echo of resolved source label

`PlotMeta` (Flask)

```
{
  "title": "C",
  "color": "#2196F3",
  "linewidth": 1,
  "index": 0,
  "kind": "plot"
}
```

`kind` is `plot` | `hline` | `fill` | `bgcolor` | `plotshape` | `plotchar` | `plotarrow` . Fill bands add `plot1` / `plot2` title refs; hline may add `price` . Extra style keys (`style` , `linestyle` , `location` , `text` , `char` , `size`) appear when the interpret collector has them.

`StrategyEvent` (minimum)

Hosts should preserve:

```
{
  "type": "string",
  "bar_index": 0,
  "script_id": "…",
  "run_id": "…"
}
```

Additional fields follow evaluator `to_dict()` / TS port parity — treat unknown keys as forward-compatible.

`AlertEvent` (minimum)

```
{
  "message": "string",
  "freq": "once_per_bar",
  "bar_index": 0,
  "time": 0,
  "source": "alert",
  "script_id": "…",
  "run_id": "…"
}
```

See Alerts for frequency rules and webhooks.

Response (failure)

Flask `/run`:

```
{
  "status": "error",
  "code": "EXECUTION_ERROR",
  "message": "Runtime Error at bar …",
  "error_kind": "runtime"
}
```

Optional `error_type`, `error_bar`, `logs`, `profile`, `meta` pass through when `Runtime` set them. `error_kind` is `parse` | `compile` | `runtime` | `data` | `order` | `mode`.

Workers may use equivalent `{ error: string }` or HTTP 4xx/5xx with message body. Clients should accept:

1. Envelope `status === "error"` with `message`, or
2. Presence of top-level `error` string (raw `Runtime.run` shape).

Batch evaluate (Flask extension)

`POST /run/batch`:

Request: `scripts: (string | {id, script})[]` (max 8) + shared `data` + optional `symbol` / `data_source` / `data_options` / `mode` / `profiler` / `libraries` / `timeout_seconds` / webhook fields. **Not** on the batch schema: `inputs`.

Response:

```
{
  "status": "success" | "partial",
  "results": [ /* per-script success or error object with id */ ],
  "count": 0,
  "ok": 0,
  "data_source": "chart"
}
```

Workers may implement batch as N single evaluates; AXIS should not assume batch exists on every host.

Internals — reference mapping

Contract field	Flask source
`plots` / `series` / `plot_meta`	`pynescript.runtime.host.Runtime.run` packing loop
`events`	`evaluator._strategy_state.drain_events()` → `to_dict()` (compile: `__events`)
`drawings`	`DrawingRegistry.export_for_api` (compile: `__drawings`)
`alerts`	`export_alerts_from_evaluator` (interpret; compile returns `[]`)
`alert_forward`	`backend/alert_forwarder.maybe_forward_run_alerts`
`script_id`	`sha256(source)[:16]`
`run_id`	`Runtime._run_id`
`compile_fallback_reason`	`Runtime._run_auto`
`timed_out`	Interpret bar loop vs `timeout_seconds` (Flask `/run` and `/run/batch` when the field is set and `> 0`)

Schema enforcement on Flask free tier: `RUN_SCHEMA` / `RUN_BATCH_SCHEMA` in `backend/middleware/schemas.py` (strict, reject extras). Webhook fields: `webhook_url`, `forward_alerts`, `alert_last_bar`, `alert_batch`. `libraries` and `timeout_seconds` are on both `/run` and `/run/batch`.

Tests: `tests/test_backend.py` (`test_run_success`, `test_run_exports_alerts`, webhook cases), `tests/test_alert_forwarder.py`.

Divergences to remember

Surface	Divergence
Preview / backtest	Columnar `data` dict, not `Bar[]`; not the evaluate contract
Compile mode	Extra keys `object_mode`, `compile_cached`, `compile_ms`, `nopython_fallback_reason`. `generated_code` only if `PYNESCRIPT_RETURN_GENERATED_CODE=1`
`timeout_seconds`	In-process `Runtime.run`, edge workers, and Flask `/run` / `/run/batch` (optional; omit = no budget)
Health	Flask `GET /` and `GET /health` (CORS-free); not part of evaluate
Auth	Flask free `/run` open (bar/rate/concurrency caps); workers may require HOOX mTLS / API gateways
Error codes	Flask uses `code` enums; workers should map to stable strings when possible

Invariants and edge cases

1. **Bar order is chronological ascending** — hosts do not sort for you.
2. `na` **/ missing** appear as JSON `null` in series arrays.
3. **Duplicate plot titles** get suffixes (`_2`, …) on Flask; clients should key on returned map keys, not assumed titles.
4. **Idempotent script_id** for identical source text across hosts.
5. **run_id is not stable** across retries — use for log correlation only.
6. Parity is best-effort on floating edges; pin fixtures when testing hosts against each other.

Worked example — host-agnostic client sketch

```
type EvaluateRequest = {
  script: string;
  data: Array<{
    time: number;
    open: number;
    high: number;
    low: number;
    close: number;
    volume?: number;
  }>;
  symbol?: string;
  mode?: "interpret" | "compile" | "auto";
  libraries?: Array<{
    namespace: string;
    name: string;
    version: number;
    source: string;
  }>;
};

type EvaluateSuccess = {
  plots: Array<number | null>;
  series: Record<string, Array<number | null>>;
  plot_meta: Record<string, Record<string, unknown>>;
  events: Array<Record<string, unknown>>;
  drawings: Array<Record<string, unknown>>;
  alerts: Array<Record<string, unknown>>;
  count: number;
  script_id: string;
  run_id: string;
  mode: string;
  auto_backend?: "compile" | "interpret";
  compile_fallback_reason?: string;
};
```

POST that body to Flask `/run` or the worker's evaluate route; branch on `status` / `error`.

Failure modes

Client bug	Symptom
Sending columnar preview data to `/run`	Schema `data must be a list`
Assuming batch on worker	404 / unknown field
Ignoring `series` and only reading `plots`	Multi-plot scripts look single-series

See also

- Run endpoint
- Runtime bridge
- pyne-worker
- pyne-agent-worker
- Pro API usage (end user)
- Events

DEVOPS

CI matrices, containers, Nuitka LSP binaries, metadata crypto, Cloud Run, and operational invariants for PYNE.

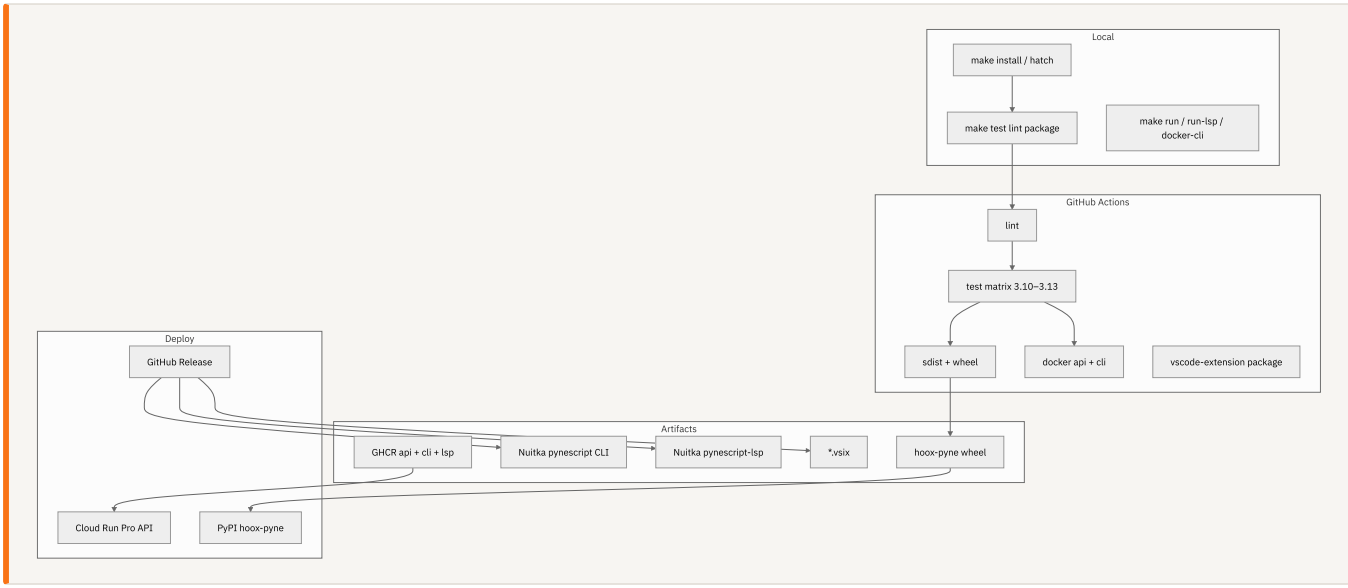
DEVOPS

Abstract

PYNE is a **multi-surface product**: pure-Python library + Click CLI (`pyne` / alias `pynescript`), pygls Language Server (`pyne-lsp`), Flask Pro API, VS Code extension (`hoox-sh.pyne` 0.3.14), and Docker targets (`api` / `cli` / `lsp`). Edge workers and the AXIS charting PWA are **sister repos** — not built in this tree.

This tab documents the **operational graph**: local loops, GitHub Actions, release tags, Docker images, Nuitka CLI/LSP binaries, Fernet-encrypted LSP metadata, GCP Cloud Build/Run, observability hooks, and security controls.

Conceptual model



Invariant: CI green on `main` is necessary but not sufficient for a full release. Tag `v*` fires **Build & Release** (binaries + VSIX + GitHub Release), **Publish** (`hoox-pyne` 0.3.14 to PyPI), and **GHCR** (`ghcr.io/hoox-sh/pyne/{api,cli,lsp}`). Cloud Build deploys the API image separately.

Interface surface

Concern	Entry	Doc
Local install & loops	`Makefile`, hatch envs	Local development
PR / push CI	`.github/workflows/ci.yml` (`name: CI`)	CI
Versioned CLI + LSP + VSIX	`.github/workflows/release.yml` (`name: Build & Release`)	Release
PyPI (`hoox-pyne`)	`.github/workflows/publish.yml` (`name: Publish`)	PyPI publish
GHCR images	`.github/workflows/ghcr.yml` (`name: GHCR`)	Docker
First ship checklist	personal PyPI + API token	Publish checklist
Containers	`Dockerfile` targets api/api-dev/lsp/**cli**	Docker
Compiled CLI / LSP	`scripts/build/compile.py --target`	Nuitka build
Builtin metadata crypto	Fernet key + `.enc`	Metadata crypto
Cloud Run	`cloudbuild.yaml`	GCP
Logs / health	Flask `/`, gunicorn, Cloud Logging	Observability
Auth, CORS, secrets	`backend/middleware/auth.py`	Security

Internals (repo map)

Path	Role
`Makefile`	Human-facing orthography: install, test, lint, build, docker, worker
`pyproject.toml`	Hatch envs (`test`, `lint`, `docs`), optional extras (`lsp`, `data`)
`.github/workflows/`	`ci.yml`, `release.yml`, `publish.yml`, `ghcr.yml`
`scripts/build/`	Nuitka compile + CI build + Fernet metadata stage
`scripts/generate_builtin_metadata.py`	Regenerates LSP `builtin_metadata.json` from live builtins
`Dockerfile` / `docker-bake.hcl`	Multi-target Buildx images (api, api-dev, lsp)
`docker-compose.yml`	Local API (+ optional Redis / LSP profiles)
`cloudbuild.yaml`	Build → GCR push → Cloud Run deploy; optional LSP compile
`vscode-extension/`	Node 22 extension package / vsce

Invariants & edge cases

1. **Two console scripts, two entrypoints.** Preferred: `pyne` (Click) ≠ `pyne-lsp` (pygls). Aliases: `pynescript` / `pynescript-lsp`. Ops scripts must not conflate them.
2. **Generated artifacts are not hand-edited.** ANTLR/ASDL under `generated/` and `builtin_metadata.json` are code-derived.
3. **Fernet key is gitignored.** Without a stable `CRYPTO_KEY` / `METADATA_KEY` secret, every CI encrypt produces a different `.enc` blob (harmless functionally, bad for reproducibility).
4. **Python matrix vs Nuitka pin.** CI tests 3.10–3.13; Nuitka release currently pins **3.11** (`release.yml` `PYTHON_VERSION`, `nuitka>=2.5.1,<2.8`).

5. **AXIS CI is not in this repo.** Playwright / PWA security gates live in hoox-sh/axis. There is no `axis-nightly.yml` here.

Worked examples

```
# Fast local confidence loop
make install
make lint
make test-lsp
make build-check   # import check only, ~30s, no Nuitka compile

# API in Docker — local *dev* stack (api-dev + source mounts)
make docker-up
make docker-smoke

# Production image bake (load local) + optional prod compose overlay
make docker-build
# export ADMIN_TOKEN=… && make docker-prod
```

Failure modes

Symptom	Likely cause	Fix
CI lint red, local green	Different ruff/mypy versions	Match CI install pins in workflow
Metadata decrypt fails in binary	Missing key at runtime	Set `PYNESCRIPT_METADATA_KEY` or embed key at build
Cloud Run 502 after deploy	Image missing deps / gunicorn bind	Confirm `Dockerfile` target `api` ENTRYPOINT + `PORT` / `GUNICORN_BIND`
VSIX empty / missing	`npm ci` / vsce not run	Use `make build-vscode` or release job artifacts
Nuitka Anaconda link error	Static libpython missing	`conda install libpython-static` or keep `--static-libpython=no`

See also

- Contributing
- LSP architecture
- Pro API contract
- pyne-worker
- pyne-agent-worker

LOCAL DEVELOPMENT

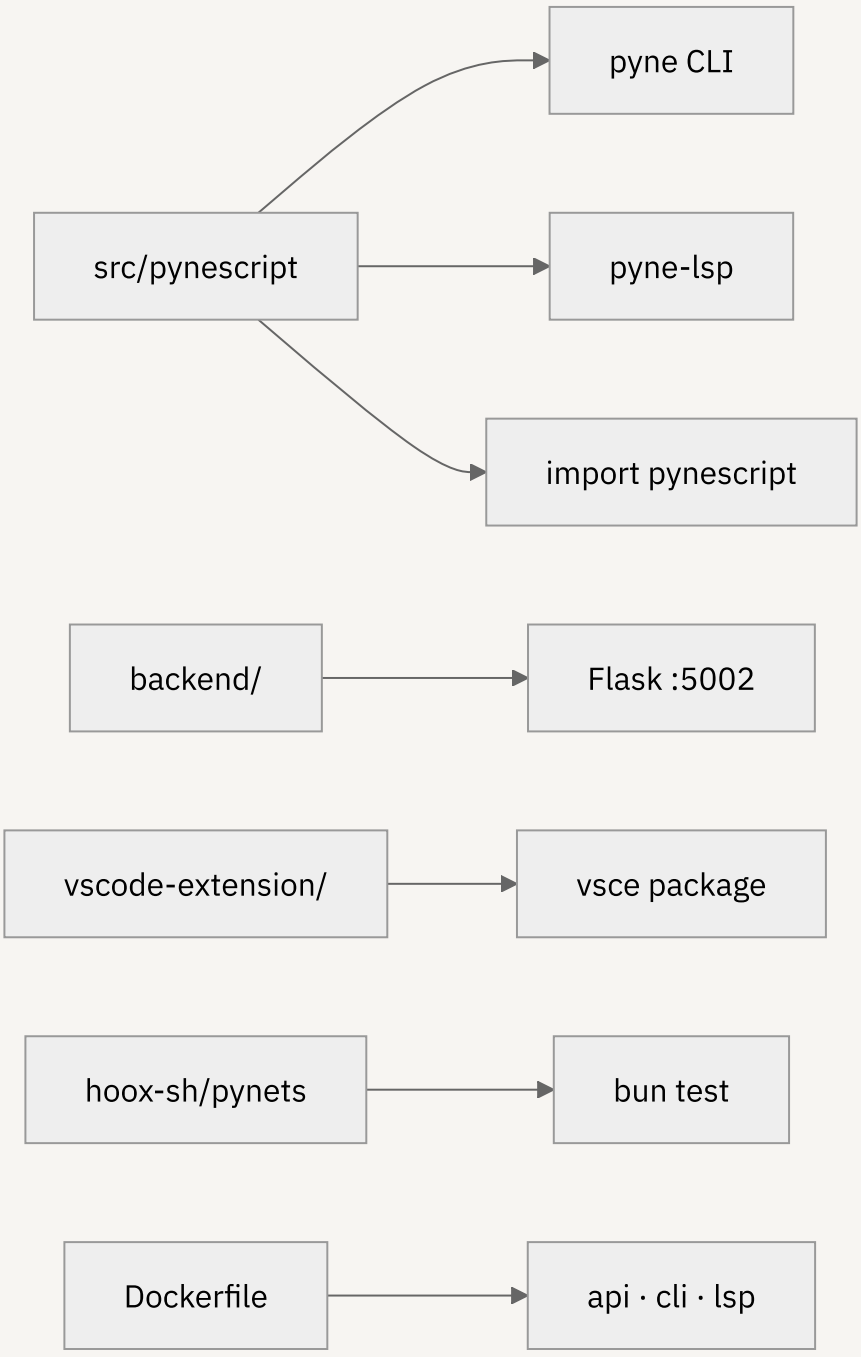
Install, test, lint, package, and run PYNE — Make targets after AXIS extraction.

LOCAL DEVELOPMENT

Abstract

A PYNE checkout is a **Python-first workspace**: src-layout package + Flask Pro API, optional VS Code extension (Node). PyneTS is the standalone hoox-sh/pynets / `@hoox-sh/pynets` package. The AXIS charting PWA is a **sister repo** (axis) — do not recreate `frontend/` here.

Conceptual model



Interface surface

Make targets (primary)

Target	Effect
`make install`	`pip install -e ".[lsp,pro]"`
`make test`	full `pytest tests/`
`make test-cli`	`tests/test_cli.py`
`make test-lsp`	langserver + lsp_features
`make test-backend`	`tests/test_backend.py`
`make lint` / `make fmt`	Ruff on `src/`, `tests/`, `backend/`
`make package`	sdist + wheel (`python -m build`)
`make build`	Nuitka **LSP** binary
`make build-cli`	Nuitka **CLI** binary
`make build-check`	import check for lsp+cli (~30s)
`make build-vscode`	compile + package VSIX
`make run`	Flask Pro API (`:5002` typical)
`make run-lsp`	`python -m pynescript.langserver`
`make docker-build`	bake **api**
`make docker-build-cli`	bake **cli**
`make docker-build-lsp`	bake **lsp** (`pynescript-lsp:latest`; GHCR name `ghcr.io/hoox-sh/pyne/lsp`)
`make docker-build-all`	api + api-dev + lsp + cli
`make docker-buildx`	multi-platform release bake (`api` + `cli` + `lsp`)
`make docker-push-ghcr`	`gh workflow run ghcr.yml`
`make docker-cli ARGS="…"`	compose profile cli (ephemeral)
`make deploy-vps`	rsync + restart API; health **API :5002**, **AXIS :80** (VPS; local AXIS is :8081)
`make docker-up`	compose API on `:5002`
`make docker-up-full`	+ redis profile
`make docker-prod`	gunicorn overlay (`ADMIN_TOKEN` required)
`make docker-smoke`	curl health on `:5002`
`make docker-down` / `docker-logs`	stop / logs

Hatch

```
hatch run test:test
hatch run test:test-cov
hatch run lint:style
hatch run lint:typing
hatch run lint:gen-parser
```

Quick desk loop

```
git clone https://github.com/hoox-sh/pyne.git
cd pyne
python3 -m venv .venv && source .venv/bin/activate
pip install -e ".[lsp,compile,pro]"
pyne info
pyne check examples/rsi_strategy.pine
make test-cli
make run   # Pro API
```

From PyPI (no checkout)

```
pip install "hoox-pyne[lsp,compile]"
pyne --version   # 0.3.14
```

Internals

Path	Why it matters
`src/pynescript/`	Editable package root
`tests/conftest.py`	Optional `--example-scripts-dir` only; no third-party corpus shipped
`backend/requirements.txt`	Pro API deps (also covered by `[pro]` extra)
hoox-sh/pynets	Standalone TS library (`@hoox-sh/pynets`); not an in-tree submodule
`vscode-extension/`	PYNE VS Code extension
`Dockerfile`	targets `api` , `api-dev` , `lsp` , **`cli`**

Python

- **Requires-Python:** `>=3.10`
- **CI matrix:** 3.10–3.13
- **Hatch test matrix:** 3.10–3.12 (no 3.13 cell)
- **Recommended:** 3.12+ with venv

Optional extras

Extra	Adds
`lsp`	pygls / lsprotocol → `pyne-lsp` (alias `pynescript-lsp`)
`compile`	numpy / numba
`data` / `datafeed`	ccxt
`pro`	Flask + gunicorn + matplotlib + redis + compile

Invariants

1. **No `frontend/` in this repo** — AXIS is external.
2. `from __future__ import annotations` required on new Python modules.
3. **Grammar edits** only under `resource/`; regenerate via hatch, don't hand-edit generated parser.
4. **Corpus fixture** runs hundreds of cases — prefer focused unit tests for CI paths.
5. **CLI Docker** runs as uid 1000; bind-mounts must be readable by others.

Failure modes

Symptom	Fix
`No module named pynescript`	activate venv; `pip install -e .` or `hoox-pyne`
Pro API ImportError flask	`pip install "hoox-pyne[pro]"` or backend requirements
Nuitka missing	`pip install nuitka`; `make build-check` first
Docker CLI can't read file	chmod host path; don't use 0700-only dirs

See also

- Installation
- CI
- Docker
- Nuitka build

CONTINUOUS INTEGRATION

GitHub Actions for lint, Python 3.10–3.13 tests, package build, Docker api+cli smoke, and VS Code extension.

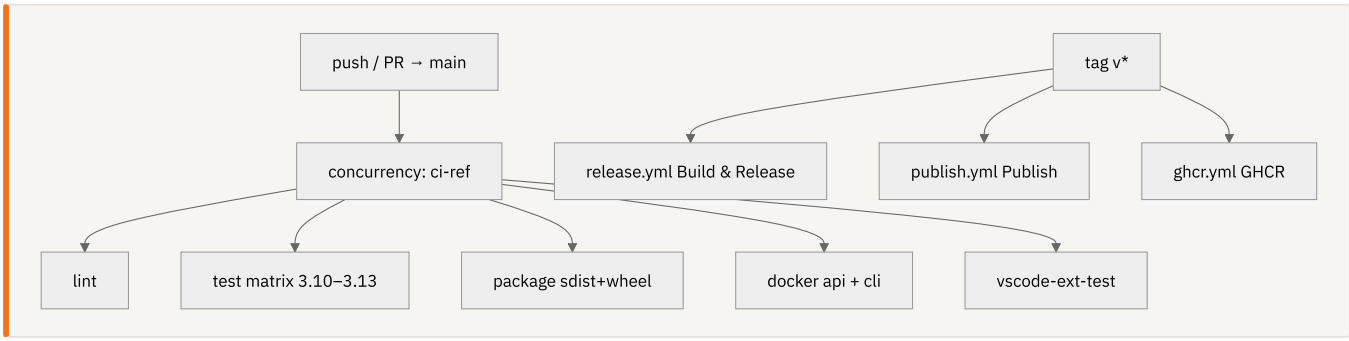
CONTINUOUS INTEGRATION

Abstract

CI is the **public invariant check** for the **pyne** repo after AXIS extraction. Workflows live under `.github/workflows/`. AXIS PWA/e2e CI lives in the sister axis repo — **not** here.

Workflow file	`name:`	Role
`ci.yml`	**CI**	Lint, test matrix, package, Docker api+cli, VSIX
`release.yml`	**Build & Release**	Tag `v*` + dispatch: wheels, CLI/LSP Nuitka binaries, CLI image tarball, VSIX, GitHub Release
`publish.yml`	**Publish**	Tag `v*` + dispatch: PyPI `hoox-pyne` (token or OIDC)
`ghcr.yml`	**GHCR**	Tag `v*` + dispatch: `ghcr.io/hoox-sh/pyne/{api,cli,lsp}` multi-arch
`docs-versions.yml`	**Docs versions**	Daily: refresh PyPI / npm / Open VSX / GitHub release stamps. Commits only on change. No docs pageload.

Conceptual model



Interface surface

Triggers

Workflow	On	Purpose
`CI`	push/PR to `main`	Required confidence
`Build & Release`	tags `v*` + dispatch	Binaries + VSIX + GitHub Release
`Publish`	tags `v*` + dispatch (`dry_run` default true)	PyPI `hoox-pyne`
`GHCR`	tags `v*` + dispatch	Public images `api` / `cli` / `lsp`

`ci.yml` jobs (current)

Job	Runtime	What it asserts
lint	Python 3.13	Ruff E/F/W gate on `src/` `tests/` (+ auth middleware); mypy non-blocking
test	Matrix 3.10–3.13	`.[lsp,pro,compile]` + backend deps; core (`test_linter` / `test_evaluator` / `test_cli`); runtime (parity, strategy, series, incremental TA, package Runtime); LSP (`test_langserver` + `test_lsp_features`); backend; Codecov on 3.13
package	3.13	`python -m build` + twine; wheel smoke (`pynescript --help`); artifact
docker	buildx	Build **api** + **cli** targets; API health + admin fail-closed; CLI `--help` / `info` / `check` (`--network=none`)
vscode-ext-test	Node 22	`npm ci` → compile → `vsce package` → VSIX artifact

Concurrency: `group: ci-${{ github.ref }}` , `cancel-in-progress: true` . Default permissions: `contents: read` .

Not in this repo's CI

- AXIS PWA / Playwright (axis repo — no `axis-nightly.yml` here)
- GHCR push (separate `ghcr.yml` , tag-triggered)
- Cloudflare Worker deploy
- Full `tests/` corpus parametrized suite (run locally: `make test`)

Internals

Path	Role
`.github/workflows/ci.yml`	PR/main fan-out (`name: CI`)
`.github/workflows/release.yml`	Multi-artifact tag release (`name: Build & Release`)
`.github/workflows/publish.yml`	PyPI (token preferred) (`name: Publish`)
`.github/workflows/ghcr.yml`	Multi-arch image push (`name: GHCR`)
`tests/test_cli.py`	CLI unit suite (CI-critical)

Secrets

Secret	Used by	Notes
`PYPI_API_TOKEN`	`publish.yml`	Personal PyPI account (`jango-blockchained`)
`METADATA_KEY` / `CRYPTO_KEY`	`release.yml` LSP encrypt	Fernet metadata
`VSCE_PAT`	optional Marketplace	VSIX still attaches without it
`GITHUB_TOKEN`	release upload	default

Invariants

1. **Fail-fast is off** on the Python matrix.

2. **Codecov only on 3.13**.
3. **CLI Docker image is non-root** — smoke mounts must be world-readable (`chmod` on mktemp).
4. **Corpus parametrized tests are intentionally skipped** in CI for time; local `make test` is broader.
5. **Package smoke** installs the wheel and runs `pynescript --help` .

Worked examples

```
# Lint as CI (correctness gate)
ruff check src/ tests/ backend/middleware/auth.py --select E,F,W --ignore E501,E741,F841,W293,E402

# Core unit slice
pip install -e ".[lsp,pro,compile]"
python -m pytest tests/test_linter.py tests/test_evaluator.py tests/test_cli.py -q

# Docker CLI smoke
docker buildx bake cli
docker run --rm --network=none pynescript-cli:latest info
```

Failure modes

Symptom	Cause	Fix
Docker CLI PermissionError on smoke.pine	mktemp 0700 + non-root user	CI chmods 755/644 (already fixed)
Wheel missing `pynescript`	wrong package name	install `hoox-pyne` / built wheel
Ruff F821 on main	incomplete imports	fix before push; gate is E,F,W

See also

- Release
- PyPI publish
- Docker

RELEASE

Tag-driven CLI + LSP multi-platform Nuitka binaries, wheels, Docker CLI image, VSIX packaging, GitHub Releases, and Marketplace publish.

RELEASE

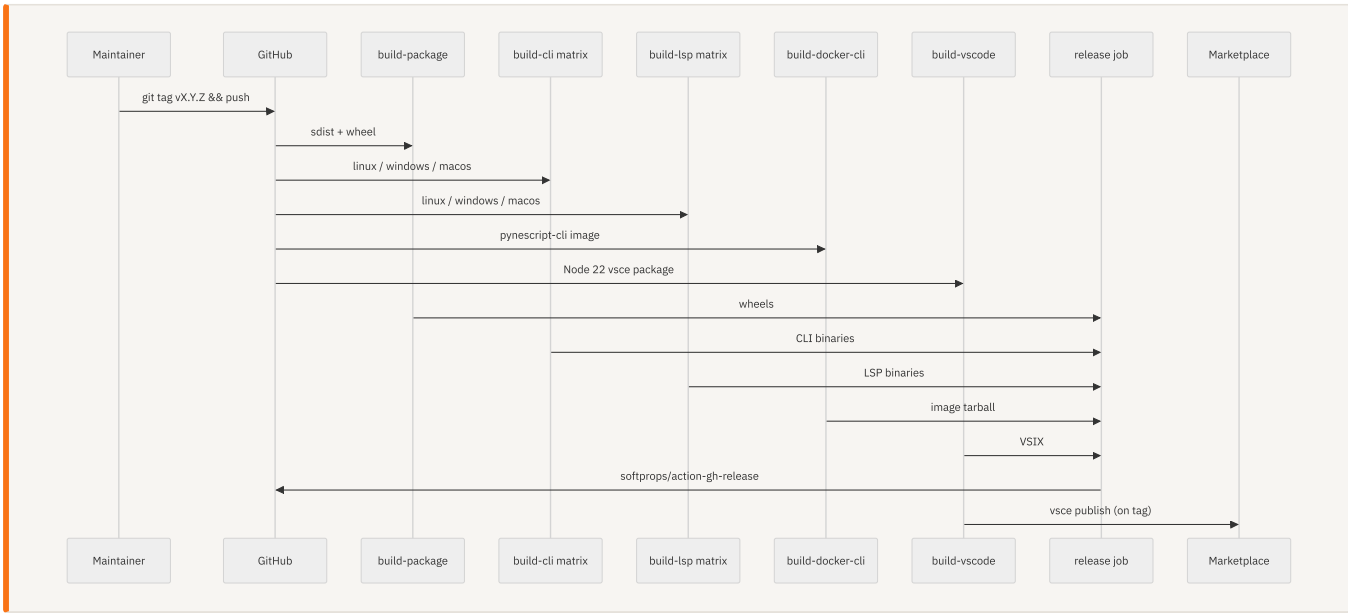
Abstract

A PYNE release is a **multi-artifact event**, not a single wheel upload. The primary automation (`.github/workflows/release.yml`) builds:

- **PyPI sdist + wheel** (`hoox-pyne`) with CLI smoke
- **CLI** platform-native Nuitka binaries (`pynescript`)
- **LSP** platform-native Nuitka binaries (`pynescript-lsp`)
- **CLI Docker image** tarball (`pynescript-cli-image.tar.gz`)
- **VS Code** VSIX

Assets attach to a GitHub Release; PyPI upload is handled by **Publish** (`publish.yml`). Multi-arch container images are a **separate** workflow (**GHCR** / `ghcr.yml`) — not attached as tarballs except the CLI image save in this workflow. Marketplace / Open VSX publish is optional via `VSCE_PAT` / `OVSX_PAT`.

Conceptual model



Interface surface

Triggers

Event	Behavior
Push tag matching `v*`	Full build + create release + marketplace/Open VSX publish
`workflow_dispatch` with `version` input	Same jobs; **Create Release** also runs on dispatch (not tag-only)

Jobs

Job	Matrix / OS	Output
`build-package`	ubuntu-latest	`hoox-pyne-dist` (sdist + wheel)
`build-cli`	ubuntu-latest, windows-latest, macos-latest	`pynescript-cli-linux-x86_64` , `…-windows-x86_64.exe` , `…-macos-arm64`
`build-lsp`	ubuntu-latest, windows-latest, macos-latest	`pynescript-lsp-linux-x86_64` , `…-windows-x86_64.exe` , `…-macos-arm64`
`build-docker-cli`	ubuntu-latest	`pynescript-cli-docker` (gzipped `docker save`)
`build-vscode`	ubuntu-latest	`pynescript-vscode-extension` VSIX
`release`	needs package + vscode success; tag **or** dispatch	GitHub Release body + staged assets (`name: pyne vX.Y.Z`)
`publish-vscode`	needs VSIX; tag **or** dispatch	`vsce publish` if `VSCE_PAT` ; `ovsx publish` if `OVSX_PAT`

Environment

```
env:
  PYTHON_VERSION: "3.11"   # Nuitka 2.5–2.7 graph; comment in workflow
  NUITKA_JOBS: 4
```

Build steps rely on:

```
# LSP
pip install -e ".[lsp]" "nuitka>=2.5.1,<2.8" cryptography
python scripts/build/compile.py --target lsp --no-encrypt --check
python scripts/build/ci_build.py --target lsp --jobs 4
# CRYPTO_KEY from secrets.METADATA_KEY

# CLI
pip install -e . "nuitka>=2.5.1,<2.8"
python scripts/build/compile.py --target cli --check
python scripts/build/ci_build.py --target cli --jobs 4 --skip-metadata --skip-vsix
```

Local make-side packaging

```
make package        # sdist + wheel (python -m build)
make build          # Nuitka LSP
make build-cli      # Nuitka CLI
make build-vscode   # npm install && compile && vsce package
make docker-build-cli
```

Artifacts land under `dist/`, `dist/lsp/`, `dist/cli/`, `dist/vsix/`, and `vscode-extension/*.vsix` depending on script path.

Internals

Path	Role
`.github/workflows/release.yml`	Orchestration
`scripts/build/ci_build.py`	CI-oriented Nuitka + metadata encrypt
`scripts/build/compile.py`	Local/full compile options (`--onefile`, `--standalone`, `--check`)
`vscode-extension/package.json`	Extension version / engines
`src/pynescript/__about__.py`	Hatch dynamic version for Python package

Release asset contract (from release body)

Artifact	Install sketch
`hoox_pyne-*.whl` / `*.tar.gz`	`pip install ./hoox_pyne-*.whl`
`pynescript-cli-linux-x86_64`	`chmod +x` → `/usr/local/bin/pynescript`
`pynescript-cli-windows-x86_64.exe`	Place on PATH as `pynescript.exe`
`pynescript-cli-macos-arm64`	Place on PATH (Apple Silicon runners)
`pynescript-cli-image.tar.gz`	`gunzip -c … | docker load` → `pynescript-cli:latest`
`pynescript-lsp-linux-x86_64`	`chmod +x` → `/usr/local/bin/pynescript-lsp`
`pynescript-lsp-windows-x86_64.exe`	Place on PATH
`pynescript-lsp-macos-arm64`	Place on PATH (Apple Silicon runners)
`pyne-vscode-*.vsix` (`hoox-sh.pyne` 0.4.4)	`code --install-extension pyne-vscode-*.vsix` · site: hoox.sh/pyne-vscode.vsix (unversioned)

Invariants & edge cases

1. **`CRYPTO_KEY` must be stable across builds** if you care about byte-identical `builtin_metadata.json.enc`. Supply `secrets.METADATA_KEY`.
2. **Tag name is the version source** for the GitHub Release title (`v` prefix stripped).
3. **Marketplace publish needs `VSCE_PAT`.** Without it, packaging may still succeed while publish fails.
4. **Artifact retention is short** (5 days on build jobs) — releases must attach files immediately.

5. **macOS artifact is ARM64** from `macos-latest` — Intel Mac users may need separate builds if required.
6. **Draft vs prerelease:** workflow currently sets `draft: false`, `prerelease: false` for tag releases.

Worked examples

Cut a release

```
# ensure main is green
git checkout main && git pull
# bump versions in extension / __about__ as needed
git tag -a v0.3.14 -m "v0.3.14"
git push origin v0.3.14
# watch Actions → Build & Release, Publish, GHCR
```

Smoke-test a downloaded binary

```
chmod +x pynescript-cli-linux-x86_64
./pynescript-cli-linux-x86_64 --help
./pynescript-cli-linux-x86_64 check script.pine

chmod +x pynescript-lsp-linux-x86_64
./pynescript-lsp-linux-x86_64 --help
# or wire stdio into an editor client
```

Failure modes

Symptom	Cause	Fix
Binary not found after Nuitka	Path layout drift in `ci_build.py`	Inspect `dist/`; fix finder step
Decrypt errors in field	Key mismatch between encrypt and runtime	Align `METADATA_KEY` with embedded key strategy
Empty GitHub Release assets	Artifact download path mismatch	Match `files:` globs to download-artifact layout
Marketplace 401	Bad/expired `VSCE_PAT`	Rotate PAT with Marketplace scopes
Windows `.exe` not marked executable	Finder uses `-type f -executable`	Fallback non-executable find branch in workflow

See also

- Nuitka build
- Metadata crypto
- CI
- VS Code extension

69 · DEVOPS

PYPI PUBLISH

Ship hoox-pyne to PyPI under the personal jango-blockchained account (API token or Trusted Publishing). Import package stays pynescript.

PYPI PUBLISH

Abstract

The Python distribution name on PyPI is **`hoox-pyne`**. After install, the import path stays **`pynescript`**; preferred CLIs are **`pyne`** / **`pyne-lsp`** (aliases: `pynescript` / `pynescript-lsp`).

PyPI ownership is the personal account **jango-blockchained** (not a PyPI organization — org approval is not required). Automation still runs on GitHub **`hoox-sh/pyne`** via `.github/workflows/publish.yml`.

Auth modes (first match wins):

1. **API token** — repo secret `PYPI_API_TOKEN` from the personal PyPI account (recommended while no PyPI org is needed).
2. **Trusted Publishing (OIDC)** — pending publisher created while logged in as `jango-blockchained`, with GitHub Owner = `hoox-sh` (where Actions run).

> **Why not `pynescript` or `pyne` on PyPI?** `pynescript` is owned by elbakramer/pynescript; plain `pyne` / `PyNE` is an unrelated 0.1.0 process-networking library. We ship as **`hoox-pyne`**; the import package stays `pynescript`.

Install surface

```
pip install hoox-pyne
pip install "hoox-pyne[lsp]"
pip install "hoox-pyne[pro]"

python -c "import pynescript; print(pynescript.__version__)"
pyne --help
pyne-lsp   # stdio language server (alias: pynescript-lsp)
```

One-time setup (personal PyPI account)

A. Recommended — API token (no org, no OIDC)

1. Sign in at pypi.org as **`jango-blockchained`** (2FA required).
2. **Account settings → API tokens → Add API token**.
   - Name: e.g. `hoox-sh-pyne-gha`
   - Scope: **Entire account** for the first upload of `hoox-pyne`; after the project exists you can rotate to a **project-scoped** token.
3. Copy the token (`pypi-…`) once.

4. On GitHub:

```
# Environment (optional reviewers / wait timer)
gh api -X PUT repos/hoox-sh/pyne/environments/pypi

# Never commit the token — Actions secret only
gh secret set PYPI_API_TOKEN -R hoox-sh/pyne
# paste pypi-... token
```

5. First successful tag (or `workflow_dispatch` with `dry_run=false`) creates **`hoox-pyne`** under your personal account.

B. Optional — Trusted Publishing (OIDC)

Use this when you prefer no long-lived token. Log into PyPI as **`jango-blockchained`** (personal user), then:

Field	Value
PyPI project name	`hoox-pyne`
Owner	**`hoox-sh`** ← GitHub **repo** owner (workflow host), **not** your PyPI username
Repository	`pyne`
Workflow name	`publish.yml`
Environment name	`pypi`

Leave `PYPI_API_TOKEN` **unset** so the publish job uses OIDC.

> Do **not** put `jango-blockchained` in the GitHub Owner field unless the workflow actually runs under `github.com/jango-blockchained/pyne`. OIDC is bound to the repo that executes the job (`hoox-sh/pyne`).

GitHub environment

Repo **Settings → Environments →** **`pypi`** on `hoox-sh/pyne`. Optional: required reviewers on production uploads.

Cut a release

```
# 1. Version + changelog on main
#    src/pynescript/__about__.py  →  0.3.14
#    vscode-extension/package.json version (same pin when shipping VSIX)
#    CHANGELOG.md

# 2. Local smoke (no upload)
pip install build twine
rm -rf dist/
python -m build
twine check dist/*
# expect: hoox_pyne-0.3.14-*.whl  and  hoox_pyne-0.3.14.tar.gz

# 3. Push main, then tag
git tag -a v0.3.14 -m "v0.3.14"
git push origin v0.3.14

# 4. Watch Actions → Publish
```

If a tag already exists and the wheel was never uploaded, re-run:

Actions → Publish → Run workflow → `dry_run=false` (builds from that workflow's default branch / checked-out tag context as configured), or retag a new patch version.

Dry-run without upload: **Actions → Publish → Run workflow →** `dry_run=true`.

Workflow map

Job	When	What
`build`	tag `v*` or dispatch	`python -m build` + `twine check` + artifact
`publish-pypi`	tag `v*` **or** dispatch with `dry_run=false`	Token if `PYPI_API_TOKEN` set, else OIDC

Failure modes

Symptom	Cause	Fix
403 invalid token	Wrong/revoked `PYPI_API_TOKEN`	Create new token as `jango-blockchained`; re-`gh secret set`
403 / no trusted publisher	OIDC path without pending publisher	Add pending publisher **or** set API token
Wrong GitHub Owner in pending publisher	Used PyPI username as Owner	Set Owner to `hoox-sh` (GitHub org of the repo)
Environment not found	Missing GitHub `pypi` env	`gh api -X PUT repos/hoox-sh/pyne/environments/pypi`
File already exists	Version already on PyPI	Bump `__about__.__version__` and tag a new version
Tag publish skipped	`dry_run` dispatch only	Use a real `v*` tag or `dry_run=false`

See also

- Release — CLI/LSP binaries + VSIX
- Publish checklist
- CI
- Root `CONTRIBUTING.md`

70 · DEVOPS

PUBLISH CHECKLIST

Publish hoox-pyne under personal PyPI account jango-blockchained from hoox-sh/pyne Actions.

PUBLISH CHECKLIST

Abstract

Ship **`hoox-pyne`** from GitHub **`hoox-sh/pyne`** while owning the PyPI project under the **personal** account `jango-blockchained`. A PyPI **organization is not required** (and is not used while org approval is pending).

0. Identity map

Surface	Value
GitHub org / repo	`hoox-sh/pyne` (Actions host)
PyPI account (owner)	`jango-blockchained` (personal)
PyPI project	`hoox-pyne`
Import / CLIs	import `pynescript`; CLIs `pyne` / `pyne-lsp` (aliases `pynescript` / `pynescript-lsp`)
Package version	`src/pynescript/__about__.py` → **0.6.4**
VS Marketplace / Open VSX publisher	`hoox-sh` (`hoox-sh.pyne` 0.4.4)

1. Confirm repo host (`hoox-sh/pyne`)

The GitHub repo **already lives** at `hoox-sh/pyne`. If a checkout still points at the old personal remote, retarget it:

```
git remote set-url origin https://github.com/hoox-sh/pyne.git
git remote -v
gh repo view --json nameWithOwner,url
# expect: hoox-sh/pyne
```

Automation secrets

Item	Action
Actions enabled	Org/repo Settings → Actions allowed for this repo
Environment `pypi`	Recreate if missing: `gh api -X PUT repos/hoox-sh/pyne/environments/pypi`
Secrets	Re-set if missing: `METADATA_KEY`, `CRYPTO_KEY` (same Fernet material as `scripts/build/.metadata.key`)
Optional	`VSCE_PAT` for Marketplace publish on tag
Workflow permissions	Read/write as needed for `release.yml` (contents: write on release job)

```
gh secret set METADATA_KEY -R hoox-sh/pyne < scripts/build/.metadata.key
gh secret set CRYPTO_KEY   -R hoox-sh/pyne < scripts/build/.metadata.key
# optional marketplace:
# gh secret set VSCE_PAT -R hoox-sh/pyne
```

2. PyPI under personal account `jango-blockchained`

Preferred — API token

1. pypi.org as **`jango-blockchained`** → **API tokens** → add token.
2. `gh secret set PYPI_API_TOKEN -R hoox-sh/pyne`
3. First upload creates project **`hoox-pyne`** owned by your user.

Optional — Trusted Publishing

Log in as **`jango-blockchained`** (personal), then pending publisher:

Field	Value
PyPI project name	`hoox-pyne`
Owner	**`hoox-sh`** (GitHub **repo** owner — Actions host)
Repository	`pyne`
Workflow name	`publish.yml`
Environment name	`pypi`

Leave `PYPI_API_TOKEN` unset to force OIDC. Do **not** put your PyPI username in the GitHub Owner field.

Details: PyPI publish.

3. Local package smoke (no upload)

```
# version already 0.6.4 in src/pynescript/__about__.py
pip install build twine
rm -rf dist/
python -m build
twine check dist/*
# expect: hoox_pyne-0.6.4-py3-none-any.whl  hoox_pyne-0.6.4.tar.gz

python -m venv /tmp/pyne-smoke
/tmp/pyne-smoke/bin/pip install dist/*.whl
/tmp/pyne-smoke/bin/python -c "import pynescript; print(pynescript.__version__)"
/tmp/pyne-smoke/bin/pyne --help
```

4. GitHub Actions dry-run

1. Push `main` with publish-ready tree to **`hoox-sh/pyne`**.
2. Confirm **CI** green (lint, test matrix, package, docker, vscode-ext).
3. Actions → **Publish** → Run workflow → `dry_run=true` (build + twine only, no upload).

5. Cut `v0.6.4`

1. Move `CHANGELOG.md` **Unreleased** notes into `## [0.6.4] - YYYY-MM-DD`.
2. Align VSIX version in `vscode-extension/package.json` (already **0.4.4**) if shipping Marketplace/VSIX with this tag.
3. Tag and push:

```
git tag -a v0.6.4 -m "v0.6.4"
git push origin v0.6.4
```

4. Watch in parallel:
   - **Publish** → PyPI `hoox-pyne`
   - **Build & Release** → GitHub Release assets + optional Marketplace
   - **GHCR** → `ghcr.io/hoox-sh/pyne/{api,cli,lsp}:0.6.4`

6. Post-publish verify

```
pip index versions hoox-pyne   # or: pip install hoox-pyne==0.6.4
python -c "import pynescript; print(pynescript.__version__)"
# GitHub Release: https://github.com/hoox-sh/pyne/releases/tag/v0.6.4
# PyPI: https://pypi.org/project/hoox-pyne/0.6.4/
```

Hardcoded owner map (repo automation)

Workflows use relative `github.*` context (no hard-coded owner) except documentation/comments. Project metadata and docs must point at **`hoox-sh/pyne`**:

Location	Role
`pyproject.toml` `[project.urls]`	PyPI package links
`.github/workflows/publish.yml` header	Trusted Publisher fields
`Dockerfile` `org.opencontainers.image.source`	Image source label
`package.json` / `vscode-extension/package.json` repository	Clone / issues URLs
`docs/pyne/docs.json` navbar GitHub	Docs site link
`scripts/build/README.md` `gh secret set -R`	Secret target repo
`CONTRIBUTING.md` + this page	Human publish runbook

Intentionally not changed

Item	Why
`.github/FUNDING.yml`	GitHub Sponsors username (personal)
Sister repos `jango-blockchained/{axis,hoox}`	Separate transfers
Author strings `jango_blockchained`	Copyright / author field
PyPI account `jango-blockchained`	Package owner; not the VS Code namespace

See also

- PyPI publish
- Release — LSP + VSIX
- CI
- Root `CONTRIBUTING.md`

DOCKER

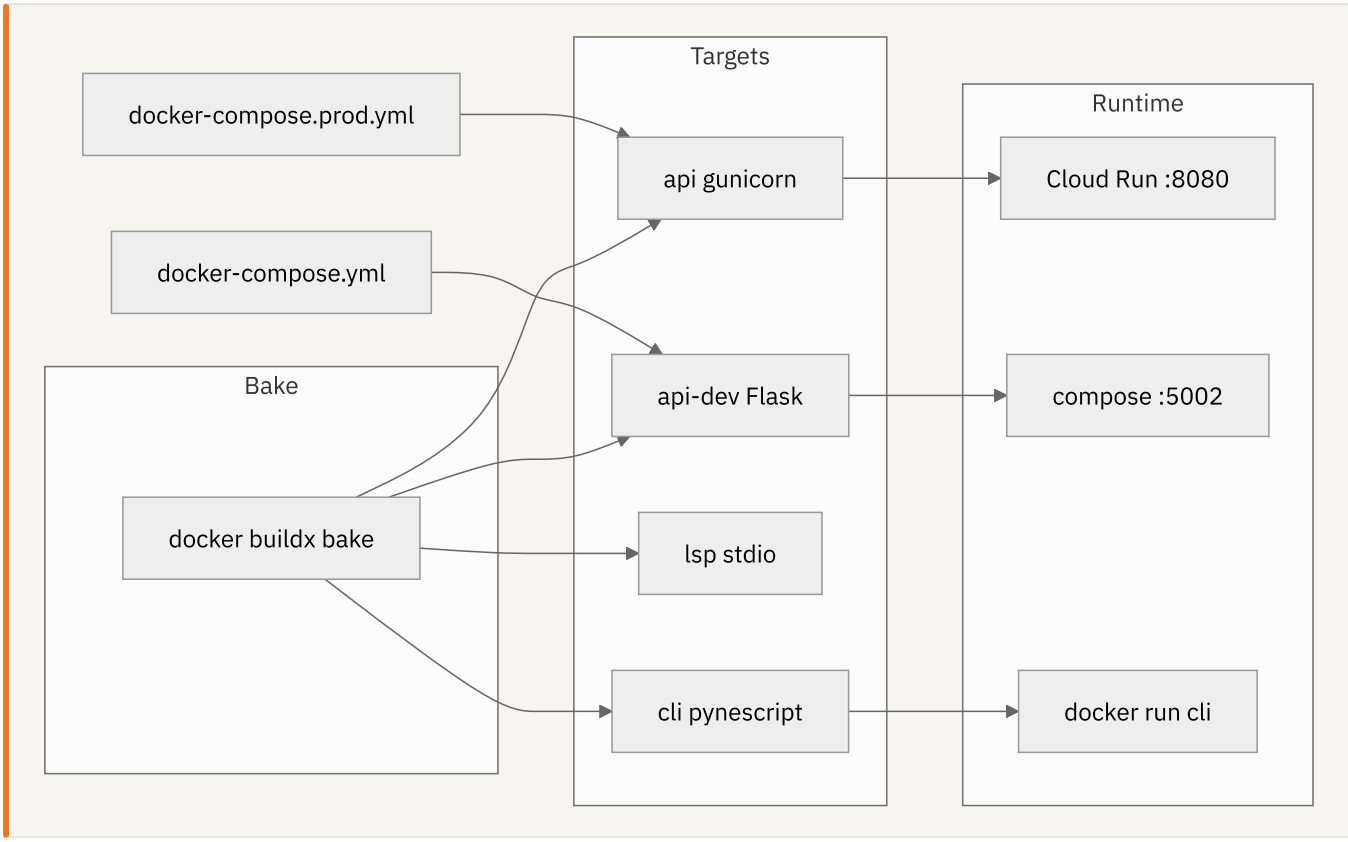
Buildx multi-target images, Compose profiles, and Cloud Run production contract for the PYNE Pro API.

DOCKER

Abstract

Containers package the **Pro API** (Flask + evaluator), the **CLI** (`pynescript`), and the **LSP** for reproducible local runs and Cloud Run deploys. A single multi-stage `Dockerfile` exposes named targets; **Docker Buildx Bake** (`docker-bake.hcl`) drives local loads and multi-platform release builds. Compose wires volume mounts and optional Redis/LSP/CLI profiles for desk development, with a production overlay for gunicorn.

Conceptual model



Interface surface

Dockerfile targets

Target	Process model	Audience
`api`	`entrypoint-api.sh` → gunicorn (`0.0.0.0:$PORT`)	Production / Cloud Run / compose prod
`api-dev`	`python -m backend.app` with `HOST=0.0.0.0`	Local compose (source mounts)
`lsp`	`ENTRYPOINT pyne-lsp` (stdio)	Compose profile `lsp` ; GHCR `ghcr.io/hoox-sh/pyne/lsp`
`cli`	`entrypoint-cli.sh` → `pyne` (fallback `pynescript` ; ephemeral; `/work` CWD)	CI gates, one-shot parse/lint/compile/run

All targets: Python 3.12-slim, non-root `appuser` (uid 1000). API targets also set `API_KEY_STORE=/data/api_keys.json` , `MPLBACKEND=Agg` , healthcheck via `curl` . The **cli** image installs `.[compile,data]` only (no Flask/matplotlib) and is smaller/faster to build than `api` .

Buildx Bake (`docker-bake.hcl`)

```
# Local load (host platform)
docker buildx bake api                # production image → pynescript-api:latest
docker buildx bake cli                # CLI image → pynescript-cli:latest (ENTRYPOINT: pyne)
docker buildx bake lsp                # LSP image → pynescript-lsp:latest (ENTRYPOINT: pyne-lsp)
docker buildx bake                    # default group: api + api-dev
docker buildx bake all                # api + api-dev + lsp + cli

# Multi-platform (amd64 + arm64). Set REGISTRY to push:
REGISTRY=ghcr.io/hoox-sh/pyne TAG=0.6.4 docker buildx bake release
# release group = api-release + cli-release + lsp-release
# → ghcr.io/hoox-sh/pyne/{api,cli,lsp}:0.6.4
# Prefer: make docker-push-ghcr  (workflow GHCR on v* tags)
```

Make wrappers:

```
make docker-build       # bake api
make docker-build-cli   # bake cli
make docker-build-lsp   # bake lsp
make docker-build-all   # bake all targets
make docker-buildx      # bake release (multi-platform)
make docker-push-ghcr   # gh workflow run ghcr.yml
make docker-cli ARGS="check script.pine"   # compose profile cli
```

One-time multi-platform builder (if needed):

```
docker buildx create --use --name pynescript
```

Compose

```
cp .env.example .env   # optional overrides

make docker-up         # api-dev on host :5002
make docker-up-full    # + redis profile (api + redis)
make docker-prod       # requires ADMIN_TOKEN; gunicorn, no source mounts
make docker-smoke      # curl http://127.0.0.1:5002/
make docker-down

# stdio LSP (not a long-running compose service)
docker compose --profile lsp run --rm lsp

# CLI (ephemeral; mounts repo at /work)
docker compose --profile cli run --rm cli check script.pine
# ≡ make docker-cli ARGS="check script.pine"
```

Service	Profile	Notes
`api`	default	Port `${API_PORT:-5002}:8080`, ro mounts of `src/` + `backend/`, volume `api_data` → `/data`
`redis`	`redis`	redis:7-alpine, AOF, `${REDIS_PORT:-6379}`
`lsp`	`lsp`	stdio; use `docker compose run --rm lsp` (not detached `up`)
`cli`	`cli`	`entrypoint-cli.sh` → `pyne`; mounts `.:/work` + `src` + `cli_data`

Network name: `pynescript-dev`.

Production overlay (`docker-compose.prod.yml`): target `api`, **`volumes: !override`** so only `api_data:/data` remains (dev source binds are dropped), SQLite key store on `/data`, mem/cpu caps, requires `ADMIN_TOKEN`.

Environment variables

Variable	Default	Meaning
`PORT`	`8080`	Listen port inside the container
`HOST`	`0.0.0.0`	Bind address (dev Flask runner)
`API_PORT`	`5002`	Host port published to container `8080`
`FLASK_ENV`	`production` / `development`	App mode by target
`STORE_BACKEND`	`json` (prod overlay: `sqlite`)	`json` | `sqlite` | `redis`
`API_KEY_STORE`	`/data/api_keys.json`	JSON key store path (`STORE_BACKEND=json`)
`API_KEY_STORE_SQLITE`	`/data/api_keys.db`	SQLite path (`STORE_BACKEND=sqlite`)
`ALLOWED_ORIGINS`	compose default includes `https://hoox.sh`, `https://hoox.sh/axis`, and **local AXIS `:8081`**	CORS allow-list. VPS AXIS is **:80** — do not copy 8081 into VPS env
`ADMIN_TOKEN`	unset	Required for `POST /auth/create_key` (`X-Admin-Token`) and prod compose
`PYNE_RUNNER`	`0`	Optional hosted runner (docs). `1` enables `/scripts` + `/cron/*`
`PYNE_RUNNER_SCHEDULER`	`0`	In-process bar-close poll. Prefer `GUNICORN_WORKERS=1` when on
`PYNE_RUNNER_DB`	`/data/runner.db`	SQLite on the `api_data` volume
`REDIS_URL`	empty in prod; compose dev default `redis://redis:6379/0`	Required when `STORE_BACKEND=redis`
`GUNICORN_WORKERS` / `GUNICORN_THREADS` / `GUNICORN_TIMEOUT`	`2` / `4` / **`120`** (entrypoint default; prod compose overlay defaults to `60`)	Prod entrypoint knobs
`GUNICORN_BIND`	`0.0.0.0:${PORT}`	Optional gunicorn bind override

Published image names

GitHub Container Registry (workflow **GHCR**, tags `v*`):

```
ghcr.io/hoox-sh/pyne/api:0.6.4
ghcr.io/hoox-sh/pyne/cli:0.6.4
ghcr.io/hoox-sh/pyne/lsp:0.6.4
```

Cloud Build (`cloudbuild.yaml`, **`--target api`** only):

```
gcr.io/$PROJECT_ID/pynescript/pynescript-pro-api:$COMMIT_SHA
gcr.io/$PROJECT_ID/pynescript/pynescript-pro-api:latest
```

Substitution `_PYNESCRIPT_VERSION` in that YAML currently defaults to `"0.3.0"` — override at submit time to match `__about__.py` (**0.6.4**).

Internals

Build stages

1. **base-os / base** — slim image, non-root `appuser`; API/LSP `base` also installs `curl` + freetype/png
2. **builder** — `build-essential`, pip cache mounts, install `backend/requirements.txt` then `pip install ".[lsp]"` into `/install`
3. **lsp-builder** — `pip install ".[lsp]"` only (no Flask / matplotlib)
4. **cli-builder** — `pip install ".[compile,data]"` only (no Flask stack)
5. **runtime / api / api-dev / lsp / cli** — final CMDs / ENTRYPOINTs and healthchecks (`lsp` has none — stdio)

`.dockerignore`

Keeps context small by excluding `tests/` (corpus), `docs/`, `brand/`, node modules, compiler `*.nbc` / `*.nbi` caches, venvs, and IDE cruft.

Compose healthchecks

API: `curl -fsS http://127.0.0.1:8080/`. Redis: `redis-cli ping`.

Invariants & edge cases

1. **Host port 5002 vs container 8080.** Local compose publishes API on **:5002**. Cloud Run uses **:8080**. VPS deploy (`scripts/deploy_vps.sh`) health-checks **API :5002** and **AXIS :80**. Local AXIS PWA is **:8081** — do not treat 8081 as the VPS PWA port.
2. **Read-only mounts in compose** mean dependency changes need image rebuild; source edits are visible via `PYTHONPATH` precedence.
3. **Production target is immutable** — no live mounts; use the prod overlay or Cloud Run.
4. **Non-root `appuser`** cannot write arbitrary paths; keep `API_KEY_STORE` under `/data`.
5. **Cloud Build always builds `--target api`** (gunicorn). Do not point it at `api-dev`.
6. **LSP is stdio** — not an HTTP service; use `docker compose run --rm lsp` rather than expecting a published port.
7. **CLI is ephemeral** — `entrypoint-cli.sh` execs `pyne` (fallback `pynescript`); pass subcommands after the image name. Workdir defaults to `/work` (compose mounts the repo there).
8. **Prod overlay is fail-closed** — `ADMIN_TOKEN` is required (`:?` interpolation); `volumes: !override` drops the dev source binds so only `api_data:/data` remains.

Worked examples

Production-like local API (bake)

```
docker buildx bake api
docker run --rm -p 8080:8080 \
  -e FLASK_ENV=production \
  -e ADMIN_TOKEN=change-me \
  -e ALLOWED_ORIGINS=https://hoox.sh,https://hoox.sh/axis \
  pynescript-api:latest
curl -s http://127.0.0.1:8080/ | jq .
```

Compose with Redis

```
docker compose --profile redis up --build
```

Optional LSP container

```
docker compose --profile lsp run --rm lsp
```

CLI image

```
docker buildx bake cli
docker run --rm pynescript-cli:latest --help
docker run --rm -v "$PWD:/work" -w /work pynescript-cli:latest check script.pine
docker run --rm -v "$PWD:/work" -w /work pynescript-cli:latest lint script.pine --json
# or via compose:
make docker-cli ARGS="run script.pine --bars 50"
```

Production overlay

```

docker compose -f docker-compose.yml -f docker-compose.prod.yml up --build -d
```

Failure modes

Symptom	Cause	Fix
Healthcheck fails	process not listening / curl missing	Use image targets from this Dockerfile; check `docker compose logs api`
Connection refused on host port	Flask bound to `127.0.0.1`	Dev target sets `HOST=0.0.0.0`; do not override to localhost
Permission denied writing keys	Running as `appuser` outside `/data`	Set `API_KEY_STORE=/data/...` and mount `api_data`
Import errors after mount	PYTHONPATH / stale site-packages	`PYTHONPATH=/app/src:/app`; rebuild after dependency changes
OOM under load	Small memory cap	Raise `API_MEM_LIMIT` / Cloud Run memory; reduce concurrency
CORS failures from PWA	Origin not allowed	Set `ALLOWED_ORIGINS`
Huge build context	Missing `.dockerignore`	Ensure `.dockerignore` is present (corpus/docs excluded)

See also

- GCP
- Security
- Pro API lifecycle
- Local development

NUITKA BUILD (LSP + CLI)

Compile pyne-lsp and pyne CLI with Nuitka — onefile vs standalone, CI build script, Anaconda quirks, and artifact layout.

NUITKA BUILD (LSP + CLI)

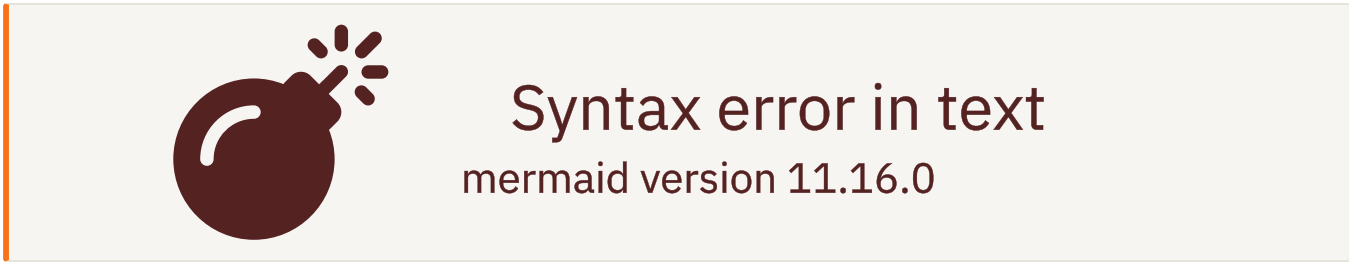
Abstract

Both the **Language Server** and the **Click CLI** can run as pure Python or as **Nuitka-compiled onefile binaries** for offline distribution. Compilation freezes the entry module and followed imports into artifacts under `dist/`.

Binary	Entry	Typical use
`pynescript-lsp` (PATH name; product CLI is `pyne-lsp`)	`langserver/__main__.py`	Editors / VSIX
`pynescript` (PATH name; product CLI is `pyne`)	`pynescript/__main__.py`	Shell / CI gates

Primary tools: `scripts/build/compile.py` (local) and `scripts/build/ci_build.py` (CI/Cloud Build). Both accept `--target lsp|cli|all`.

Conceptual model



Interface surface

Commands

```
# Prerequisites
pip install nuitka cryptography

# Full LSP onefile build (default path via Make)
make build
# ≡ python scripts/build/compile.py --target lsp --jobs=4

# CLI onefile binary
make build-cli
# ≡ python scripts/build/compile.py --target cli --jobs=4

# Fast import check only (~30s) — both targets
make build-check
# ≡ python scripts/build/compile.py --target all --check

# Explicit modes
python scripts/build/compile.py --target cli --standalone
python scripts/build/compile.py --target all --jobs 8
python scripts/build/ci_build.py --target cli --jobs=4 --skip-metadata --skip-vsix
```

Build modes (approx. wall time)

Mode	Time	Notes
`--check`	~30s	Import resolution / no full compile
`--standalone`	5–15 min	Directory distribution, faster iteration
`--onefile`	10–30 min	Self-extracting single binary
CI 8 cores	5–15 min	High parallelism

Expected layout

```
dist/
├── lsp/
│   └── pynescript-lsp          # onefile (path may vary by script)
├── vsix/
│   └── pynescript-lsp.vsix     # optional bundle
└── pynescript-lsp              # ci_build final move target (script-dependent)
```

Also see `scripts/build/README.md`.

Internals

Key Nuitka flags (`compile.py`)

- `--static-libpython=no` — default; critical for many Linux/Anaconda envs
- `--python-flag=no_site,no_docstrings`
- `--include-data-dir=…/providers=pynescript/langserver/providers`

- `--lto=auto`, `--jobs=N`
- `--onefile` or `--standalone`
- `--follow-imports` on local compile path

Entry module

Compilation targets `src/pynescript/langserver` (package entry), product name `pynescript-lsp`.

Metadata stage

Before compile, scripts may:

1. Run `scripts/generate_builtin_metadata.py` if plaintext metadata missing
2. Fernet-encrypt to `builtin_metadata.json.enc`
3. Write truncated SHA256 sidecar
4. Write `scripts/build/.metadata.key` (gitignored) or consume `CRYPTO_KEY` env in CI

Anaconda

If static libpython is unavailable:

```
conda install libpython-static
# or rely on --static-libpython=no (already the default in compile.py)
```

Invariants & edge cases

1. **C compiler required** (gcc/clang/MSVC). Containers need `build-essential` or equivalent if compiling inside Docker.
2. **Python 3.10+**; release workflow pins **3.11** with `nuitka>=2.5.1,<2.8`. Local/Cloud Build sketches often use 3.12.
3. **Providers data dir must ship** — completion/hover collapse without metadata.
4. **Onefile startup cost** includes extract-to-temp; standalone is snappier for local soak tests.
5. **Do not commit** `scripts/build/.metadata.key`.

Worked examples

Local onefile with explicit jobs

```
python scripts/build/compile.py --jobs "$(nproc)"
find dist -name 'pynescript-lsp*' -type f
```

CI-shaped build

```
python scripts/build/ci_build.py --jobs=4
```

Verify without compile

```
python scripts/build/compile.py --check
```

Failure modes

Symptom	Cause	Fix
Link error on libpython	Anaconda static lib missing	Install static lib or keep `--static-libpython=no`
Binary missing builtins	Data dir not included	Confirm `--include-data-dir` providers path
Huge binary	Full stdlib pull-in	Review Nuitka plugins / unused deps
CI timeout	Low cores / onefile	Raise jobs; use larger runner; prefer standalone intermediate
Decrypt fail at runtime	Key not embedded / env unset	See Metadata crypto

See also

- Metadata crypto
- Release
- LSP builtin metadata

METADATA ENCRYPTION

Fernet-encrypted LSP builtin_metadata.json — key lifecycle, CI secrets, integrity hashes, and runtime decrypt.

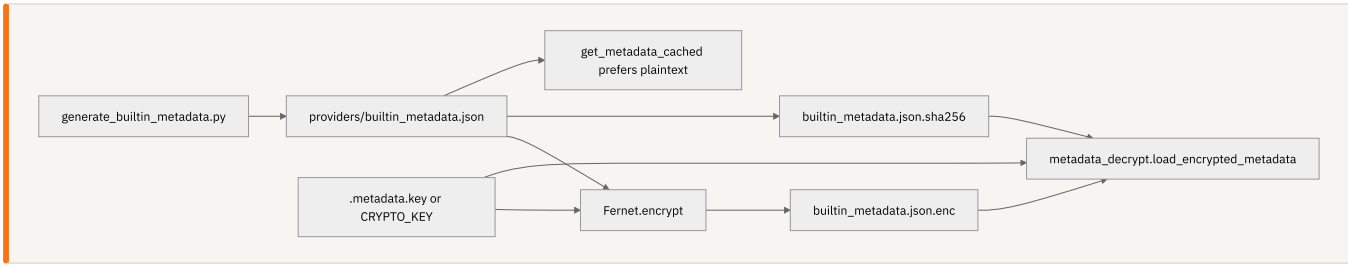
METADATA ENCRYPTION

Abstract

LSP completion, hover, and signature help depend on a large **builtin metadata** document derived from the live evaluator surface. In open development the JSON is plaintext and git-tracked. For **compiled Nuitka binaries**, the build pipeline can ship an encrypted blob so the binary does not leave a trivially scrapable metadata file on disk.

Mechanism: **Fernet** (symmetric, `cryptography` package) + optional SHA256 integrity sidecar.

Conceptual model



Interface surface

Files

Path	Tracked?	Role
`src/pynescript/langserver/providers/builtin_metadata.json`	yes	Plaintext for dev / hatch
`…/builtin_metadata.json.enc`	often yes	Encrypted payload for binaries
`…/builtin_metadata.json.sha256`	yes	Truncated SHA256 of plaintext
`scripts/build/.metadata.key`	**no** (gitignored)	Local Fernet key material
`scripts/generate_builtin_metadata.py`	yes	Regenerates JSON from code

Environment variables

Variable	Context	Meaning
`CRYPTO_KEY`	Build scripts / CI / Cloud Build	Key bytes for encrypt stage
`PYNESCRIPT_METADATA_KEY`	Runtime decrypt fallback	Env-supplied key if file missing
GitHub `secrets.METADATA_KEY`	Actions	Stable secret → `CRYPTO_KEY`
Cloud Build `_METADATA_KEY`	`cloudbuild.yaml` substitution	`CRYPTO_KEY=${_METADATA_KEY}`

Code entrypoints

- Encrypt stage: `scripts/build/compile.py` (`encrypt_metadata` / `_resolve_fernet_key`), `scripts/build/ci_build.py` (`stage_metadata`)
- Decrypt: `src/pynescript/langserver/providers/metadata_decrypt.py`
- Loader preference: plaintext if present, else encrypted (`get_metadata` returns `{}` on miss; `get_metadata_cached` raises)

Internals

Generation (always code-derived)

```
python scripts/generate_builtin_metadata.py
```

Do **not** hand-edit the JSON for permanent feature work — re-run the generator after adding builtins.

Encrypt (local sketch)

```
from cryptography.fernet import Fernet
key = Fernet.generate_key()  # or load stable secret
fernet = Fernet(key)
encrypted = fernet.encrypt(plaintext_bytes)
```

Build scripts chmod the key file to `0o600` when writing `.metadata.key`.

Decrypt integrity check

After Fernet decrypt, if a `.sha256` sidecar exists, the loader compares `sha256(plaintext)[:16]` and raises on mismatch.

Dev vs binary

Mode	Behavior
Editable install / repo checkout	Prefer plaintext JSON
Nuitka onefile	Encrypted data dir + key resolution via MEIPASS / env
Missing both	`FileNotFoundError` directing to compile script

Invariants & edge cases

1. **Stable key ⇒ reproducible `.enc`.** Random key every CI run yields a different ciphertext even if plaintext is identical — not a functional bug, but pollutes diffs and caches.
2. **Plaintext remains in git** for open-source transparency of the *language surface*. Encryption protects the *bundled binary layout*, not the secret of the API catalog.
3. **Truncated hash is integrity, not secrecy.** It detects bitrot / wrong plaintext, not attackers with the key.
4. **Never commit `.metadata.key`.** Rotate if leaked; regenerate `.enc` with the new key for binary builds.
5. **After adding builtins:** regenerate metadata, re-encrypt with the **same** CI key, commit updated JSON (and `.enc` if tracked).

Worked examples

Regenerate + local encrypt

```
python scripts/generate_builtin_metadata.py
pip install cryptography
python scripts/build/compile.py --check   # stages metadata paths as configured
```

CI snippet

```
- name: Build LSP binary
  run: python scripts/build/ci_build.py --jobs 4
  env:
    CRYPTO_KEY: ${{ secrets.METADATA_KEY }}
```

Runtime with env key

```

pynescript-lsp
```

Failure modes

Symptom	Cause	Fix
`No metadata decryption key found`	Missing file + env	Supply `PYNESCRIPT_METADATA_KEY` or rebuild with key
`Metadata integrity check failed`	Stale hash or wrong ciphertext	Re-encrypt from current plaintext; commit both
Completion empty in binary only	Data dir not included in Nuitka	Fix `--include-data-dir`
Divergent `.enc` every CI	Unset `CRYPTO_KEY`	Configure `METADATA_KEY` secret

See also

- Nuitka build
- LSP builtin metadata
- Release
- Security

74 · DEVOPS

GCP & CLOUD RUN

Cloud Build pipeline, Cloud Run Pro API sizing, cost model, and substitution secrets for PYNE deployments.

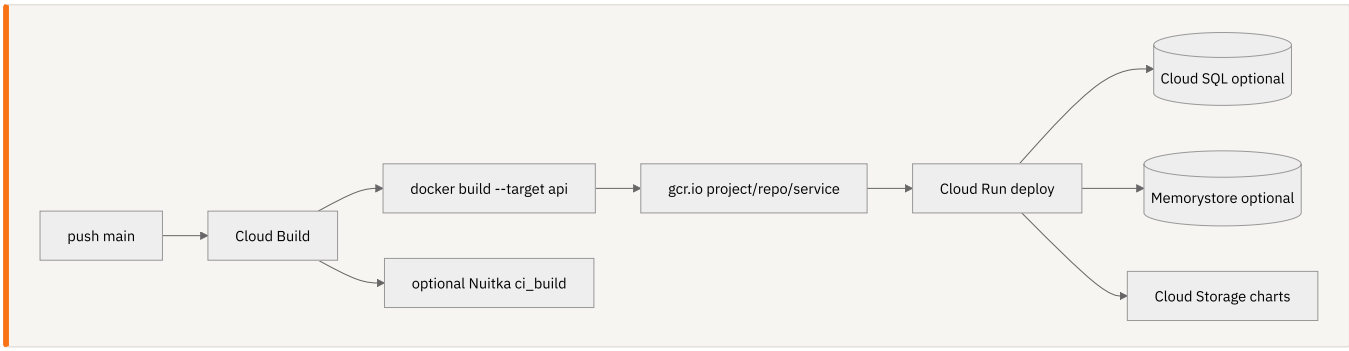
GCP & CLOUD RUN

Abstract

Production-shaped hosting for the Pro API targets **Google Cloud Run**, fed by **Cloud Build** (`cloudbuild.yaml`). The pipeline builds a container, pushes multi-tags to Container Registry, deploys a managed service, and can optionally compile the LSP binary in the same build graph.

This page codifies the as-checked-in config and the cost envelope from `docs/gcp_cost_estimate.md` — treat dollar figures as planning estimates, not invoices.

Conceptual model



Interface surface

Substitutions (`cloudbuild.yaml`)

Substitution	Default	Meaning
`_SERVICE_NAME`	`pynescript-pro-api`	Cloud Run service
`_REGION`	`us-central1`	Deploy region
`_REPOSITORY`	`pynescript`	Image path segment
`_METADATA_KEY`	(secret/subst)	Fernet key → `CRYPTO_KEY` for LSP stage
`_PYNESCRIPT_VERSION`	`"0.3.0"` (stale default in YAML)	OCI label; override to **0.3.14** at submit

Build steps

1. **build** — `docker build` with BuildKit; tags `$COMMIT_SHA` + `latest`; cache-from `latest`
2. **push** — `docker push --all-tags`
3. **deploy** — `gcloud run deploy` with managed platform flags
4. **build-lsp** — `python:3.12-slim` image runs `scripts/build/ci_build.py --jobs=4` with `CRYPTO_KEY=${_METADATA_KEY}`

Cloud Run flags (as configured)

Flag	Value	Rationale
`--allow-unauthenticated`	on	Public HTTP API (auth via API keys at app layer)
`--min-instances`	0	Scale to zero
`--max-instances`	10	Cap spend / blast radius
`--memory`	512Mi	Evaluator headroom
`--cpu`	1	Single vCPU per instance
`--concurrency`	10	Parallel requests per instance
`--timeout`	60s	Platform kill. Entrypoint gunicorn default is **120s** (`GUNICORN_TIMEOUT`); set the env to **60** on Cloud Run or long evals die at the platform first.
`--set-env-vars`	`FLASK_ENV=production`	App mode

Machine / budget knobs

```
options:
  machineType: E2_HIGHCPU_8
  logging: CLOUD_LOGGING_ONLY
timeout: 1200s
```

Internals

Path	Role
`cloudbuild.yaml`	Pipeline definition
`Dockerfile` target `api`	Production image recipe used by Cloud Build
`backend/app.py`	Health `/` , CORS, body size limit
`docs/gcp_cost_estimate.md`	Scenario costs (April 2026 draft)

Cost envelope (planning)

Scale	Rough monthly	Notes
Free / hobby	~$0	Within Cloud Run free tier if tiny
MVP (~10 pro users)	~$50	Cloud Run + micro SQL
Growth (~50)	~$120	Add Redis caching
Scale (~200)	~$350	Dominated by evaluator CPU-sec

Dominant cost driver: **script execution CPU time** (~2–5 CPU-sec per typical backtest over 1k bars). Memory ~200–400MB per concurrent Python worker.

Free-tier leverage (GCP)

- Cloud Run: 2M requests, 400K CPU-sec, 200K GB-sec / month
- Cloud Build: 120 build-minutes / day
- Artifact Registry / GCS small free slices

Invariants & edge cases

1. **App-layer auth still required** when Cloud Run allows unauthenticated invoke — see Security.
2. **Scale-to-zero cold starts** add latency; min-instances=1 trades money for snappiness.
3. **LSP build step does not deploy the binary** to Cloud Run; it is a release-adjacent compile in the same YAML.
4. **512Mi may be tight** for large OHLCV payloads — watch OOM kills; raise memory before raising concurrency.
5. **Cost estimate doc** also lists Railway / Render / Fly / self-host alternatives if GCP is overkill.

Worked examples

Manual deploy sketch (operator)

```
gcloud builds submit --config cloudbuild.yaml \
  --substitutions=_METADATA_KEY="$METADATA_KEY"
```

Smoke health after deploy

```
curl -s "https://SERVICE-URL/"
# expect JSON status healthy, service pynescript-pro-api
```

Failure modes

Symptom	Cause	Fix
Deploy timeout	Nuitka step + docker on 20 min budget	Split LSP compile to release pipeline
429 / queue	max-instances or concurrency	Raise caps carefully; add queue
High bill	No cache; chatty clients	Redis TTL; Cloud Tasks for backtests
Auth bypass illusion	Unauthenticated Cloud Run	Enforce API keys in Flask
Image pull errors	Wrong project / registry perms	IAM on GCR + Cloud Run service agent

See also

- Docker
- Observability
- Security
- Pro API

OBSERVABILITY

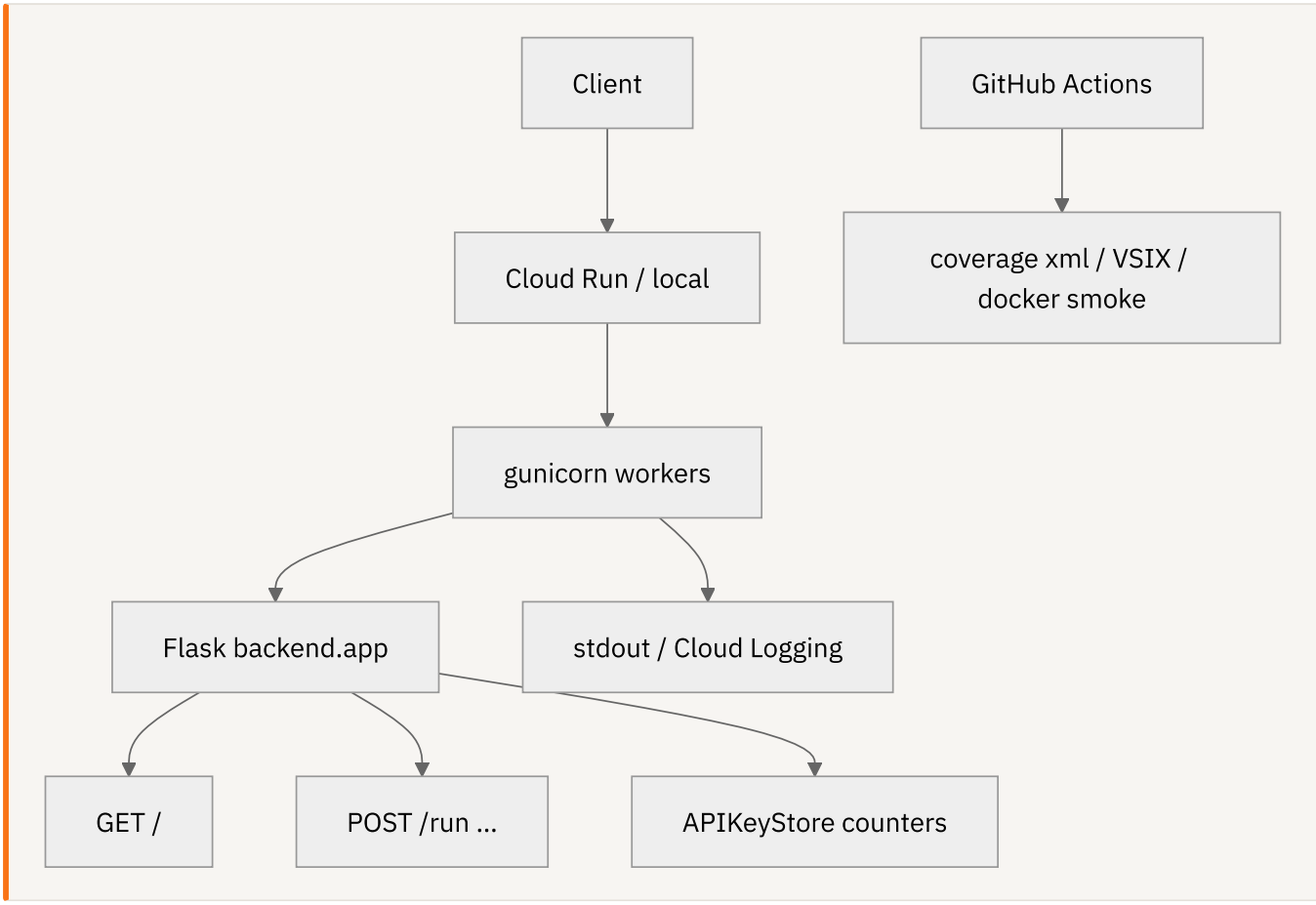
Health endpoints, gunicorn process model, Cloud Logging, coverage artifacts, and operational signals for PYNE.

OBSERVABILITY

Abstract

PYNE's production surface today optimizes for **simple, hostable signals** rather than a full OpenTelemetry mesh: HTTP health checks, process managers (gunicorn), cloud platform logs, CI artifacts (coverage, Playwright reports), and application-level API key usage counters. Treat this page as the map of *what exists in-repo*, not a claim of enterprise APM parity.

Conceptual model



Interface surface

Health

`backend/app.py` exposes:

```
GET /
→ { "status": "healthy", "service": "pynescript-pro-api", … }
```

Used by:

- `Dockerfile` target `api` `HEALTHCHECK` (`curl -fsS http://127.0.0.1:8080/`)
- `docker-compose.yml` api healthcheck
- Load balancers / Cloud Run startup-ish probes (platform-dependent)

Process model (production)

```
gunicorn --bind :8080 --workers 2 --threads 4 --timeout 120 backend.app:app
# docker/entrypoint-api.sh: GUNICORN_TIMEOUT default 120
# docker-compose.prod.yml: GUNICORN_TIMEOUT default 60
```

Knob	Effect on signals
workers	Process isolation; multiplies memory
threads	Concurrent requests within worker
timeout	Entrypoint default **120s**. Cloud Run `--timeout=60s` is tighter unless you raise the platform flag or lower `GUNICORN_TIMEOUT`.

Logs

Environment	Where logs go
Local `make run`	Process stdout/stderr
Docker	Container logs (`docker logs`, compose)
Cloud Build	`logging: CLOUD_LOGGING_ONLY`
Cloud Run	Cloud Logging (request + container)

Application modules use standard library / Flask logging patterns; there is no mandatory structured-log schema enforced repo-wide. `SLOT0` if a future branch introduces OpenTelemetry exporters.

CI / quality signals

Signal	Source
Ruff / mypy	`CI` lint job
Pytest results	runtime + LSP + backend steps in `CI`
Codecov (optional)	Python 3.13 matrix cell
VSIX	`CI` `vscode-ext-test` artifact
Docker smoke	`CI` api health + admin 403; CLI `--help` / `info` / `check`

Usage metering (app-layer)

`backend/middleware/auth.py` tracks per-key:

- `calls_used` / `calls_limit` / tier
- `last_used` timestamps

These are **business metrics** for rate limits, not Prometheus time series — export them if you need dashboards.

Internals

Path	Role
`backend/app.py`	Health route, CORS, `MAX_CONTENT_LENGTH`
`backend/middleware/auth.py`	Key store, rate limit counters
`backend/services/backtest.py`	Backtest metrics objects for API responses
`Dockerfile` (`api` target)	HEALTHCHECK + gunicorn entrypoint
`cloudbuild.yaml`	Cloud Logging-only build logs
`logs/` (repo)	Local combined/error log files if generated by tools — not the production contract

Invariants & edge cases

1. **Health ≠ readiness of evaluator warm caches.** A 200 on `/` does not mean the first `/run` will be fast (cold import / JIT / numba).
2. **Timeouts double-bind.** Cloud Run 60s vs gunicorn default 120s — the **platform** limit wins on Cloud Run unless you set `GUNICORN_TIMEOUT=60`.
3. **Scale-to-zero hides idle metrics.** No traffic ⇒ no samples; use synthetic uptime checks if SLOs matter.
4. **Coverage artifacts expire** (7–14 days retention on Actions uploads).
5. **Do not scrape secrets from logs.** API keys and admin tokens must never be logged at info level.

Worked examples

Local health loop

```
make run
curl -s http://127.0.0.1:5002/ | python -m json.tool
```

Docker health

```
docker inspect --format='{{json .State.Health}}' CONTAINER
```

Tail Cloud Run (gcloud)

```
gcloud run services logs read pynescript-pro-api --region us-central1 --limit 50
```

Failure modes

Symptom	Cause	Fix
Health green, `/run` 500	Exception in evaluator path	Inspect request logs; reproduce with payload
Worker kills	Soft timeout	Optimize script / raise timeout consistently
Missing CI artifact	Path wrong / tests passed	`if-no-files-found: ignore` hides absence
Rate limit false positives	Shared key / limit too low	Adjust tier limits; multi-key
Silent deploy failure	Build logging-only + unread	Check Cloud Build history UI

See also

- GCP
- Docker
- Security
- API lifecycle

SECURITY

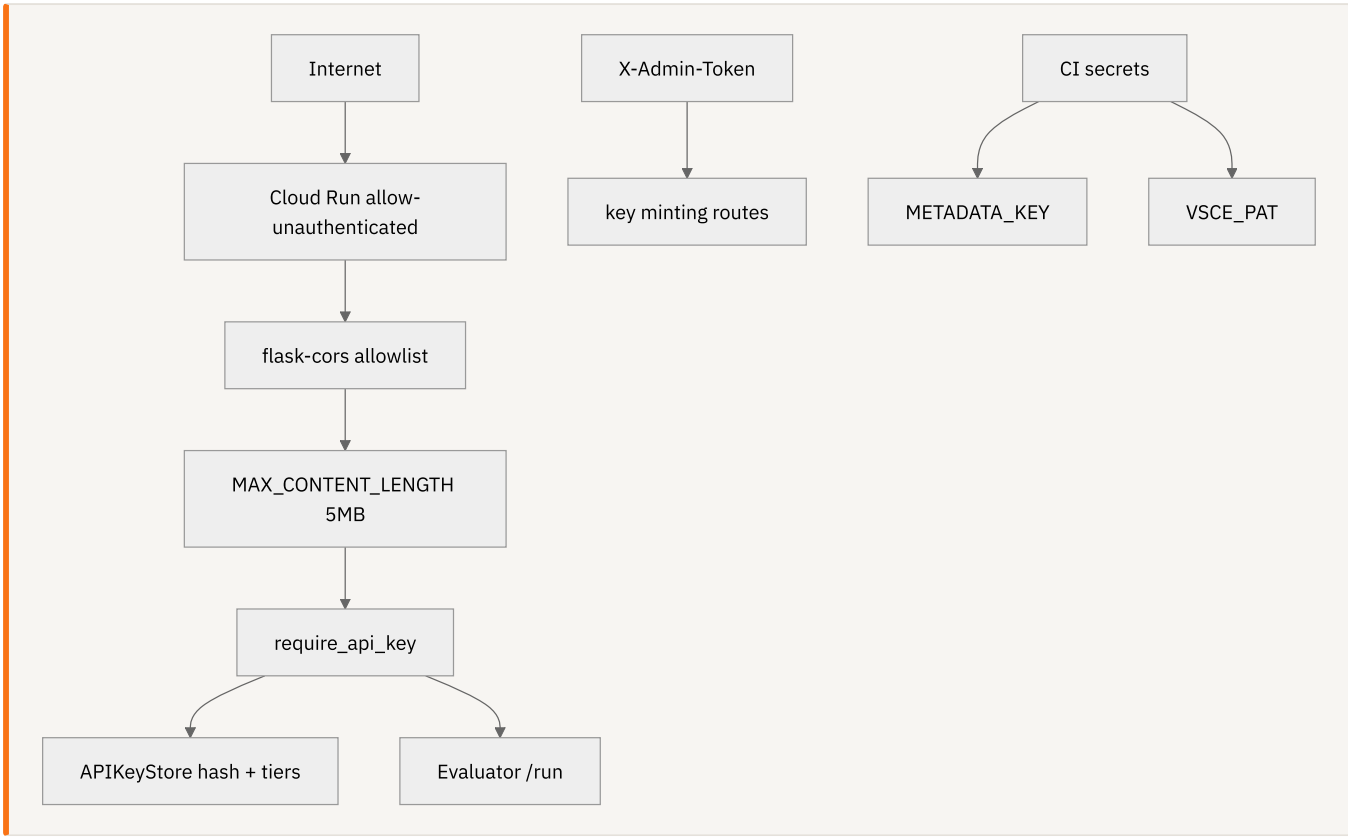
API keys, CORS, body limits, secrets hygiene, Fernet metadata keys, and threat-aware defaults for PYNE ops.

SECURITY

Abstract

PYNE security is layered: **transport/hosting** (Cloud Run, containers), **application** (API keys, admin tokens, CORS, body size caps), **build secrets** (Fernet metadata keys, VSCE PAT), and **supply chain** (CI dependency installs, extension packaging). This page documents controls that exist in code and workflows — not a full threat model certification.

Conceptual model



Interface surface

HTTP application controls (`backend/app.py`)

Control	Default / mechanism
Body size	`MAX_CONTENT_LENGTH = 5 * 1024 * 1024` (5 MiB) — DoS/OOM mitigation
CORS	`ALLOWED_ORIGINS` env (comma-separated); default includes product hosts + localhost regex
Methods	GET, POST only via flask-cors config
Headers	`Content-Type` , `Authorization` , `X-Admin-Token`
Credentials	`supports_credentials=False`

API keys (`backend/middleware/auth.py`)

- Keys stored as **hashes**, not raw secrets (store path via `API_KEY_STORE`)
- Tiers: `free` , `hobby` (5k), `pro` (50k), `team` (200k), `enterprise` (∞)
- Rate limit by monthly call counters (`calls_used` / `calls_limit`)
- Decorators: `require_api_key` , `require_admin_token`

Build / release secrets

Secret	Purpose
`METADATA_KEY` / `_METADATA_KEY`	Stable Fernet key for LSP metadata encrypt
`VSCE_PAT`	VS Code Marketplace / vsce package
`GITHUB_TOKEN`	Release asset upload
`PYNESCRIPT_METADATA_KEY`	Runtime decrypt env fallback

Container hardening (`Dockerfile` target `api`)

- Multi-stage build (no compiler toolchain in final image)
- Non-root `appuser` (uid 1000)
- Production env flags + gunicorn entrypoint
- Healthcheck without privileged ops
- Intended mutable path `/data` (key store); prod compose uses SQLite there for multi-worker

Admin minting

- `POST /auth/create_key` requires `ADMIN_TOKEN` env **and** matching `X-Admin-Token` header (constant-time compare)
- Unset `ADMIN_TOKEN` → 403 (fail closed, not open access)

Key store backends (`STORE_BACKEND`)

Value	Use
`json` (default)	Single-process / tests; file may hold raw keys
`sqlite`	Multi-worker single host (shared volume); **hash-only**
`redis`	Multi-replica; needs `REDIS_URL` ; **hash-only**

Internals

Path	Role
`backend/app.py`	CORS, body limit, blueprint registration
`backend/middleware/auth.py`	Key store selection, tiers, decorators, admin gate
`backend/middleware/key_store_redis.py` / `key_store_sqlite.py`	Hash-only shared stores
`backend/middleware/schemas.py`	Request validation schemas
`src/pynescript/langserver/providers/metadata_decrypt.py`	Encrypted metadata load
`.github/workflows/*.yml`	Secret consumption (`METADATA_KEY` , `PYPI_API_TOKEN` , `VSCE_PAT` , `OVSX_PAT`)
AXIS PWA security tests	Sister repo hoox-sh/axis — not in this tree

Audit notes encoded in source

Comments in `app.py` reference a **2026-07-05 audit** (findings S1 body size, S7/S8 CORS). Prefer reading the code over assuming defaults never change.

Invariants & edge cases

1. **Cloud Run `--allow-unauthenticated` does not mean open data.** It means IAM does not gate invoke; the Flask app must.
2. **Free tier with `calls_limit=0` is special-cased** as unlimited remaining in some helpers — verify tier semantics before production billing.
3. **Localhost CORS regex** permits any port on localhost/127.0.0.1 — fine for desk, wrong for prod `ALLOWED_ORIGINS` .
4. **Fernet protects binary metadata packaging**, not user script confidentiality.
5. **Admin token routes are high privilege** — rotate and store separately from user API keys.
6. **Never log raw API keys.** Hash lookups only.

Worked examples

Harden CORS for a single origin

```
python -m backend.app
```

Point key store at a volume

```
# ensure process user can write /data
```

Rotate metadata key (ops procedure sketch)

1. Generate new Fernet key; store as secret `METADATA_KEY`
2. Rebuild LSP with `CRYPTO_KEY` set
3. Ship new binary; retire old key after client upgrade window
4. Do not leave dual-key decrypt unless you implement it

Failure modes

Symptom	Cause	Fix
Browser CORS error	Origin not listed	Set `ALLOWED_ORIGINS`
413 Payload Too Large	Body > 5MB	Reduce OHLCV batch; raise limit knowingly
401/403 on `/run`	Missing/invalid key	Check `Authorization` header scheme
Rate limited	Tier exhausted	Upgrade tier / wait reset
Secret in fork PR logs	Misconfigured workflow echo	Never `echo` secrets; use masked env
Root-owned files in volume	Old image ran as root	Align UID with `appuser`

See also

- Auth and keys (API)
- Metadata crypto
- Docker
- GCP

77 · REFERENCE

ECOSYSTEM

PYNE, PyneTS, pyne-worker, pyne-agent-worker, AXIS, and HOOX — what each one is and is not.

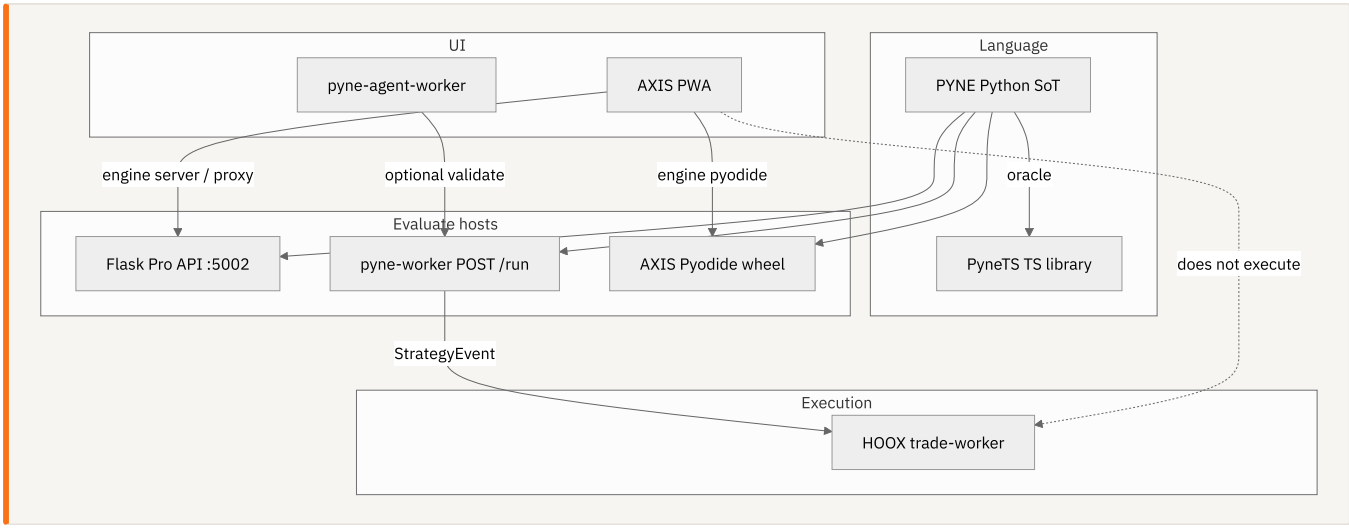
ECOSYSTEM

Abstract

The HOOX family splits **language**, **chart**, and **execution**. Mixing the names is the most common docs failure: `pyne-worker` is not `pynets`, and AXIS does not place HOOX orders.

Product	Role	Docs
PYNE (`hoox-pyne` , import `pynescript`)	Python language SoT: parse, LSP, Pro API, `Runtime.run`	this site
PyneTS (`@hoox-sh/pynets`)	TypeScript / Bun **library + CLI**	PyneTS
pyne-worker	Python Cloudflare isolate — `POST /run`	pyne-worker · HOOX isolate
pyne-agent-worker	NL → Pine (Workers AI)	agent · AXIS plugin
AXIS	Charting PWA (engines call Python)	AXIS
HOOX	Edge trade mesh	HOOX

Conceptual model



Invariant: evaluation never requires a proprietary chart host. AXIS is an optional chart. HOOX is an optional execution mesh. PyneTS is an optional TypeScript import.

Interface surface

Published versions (static; a daily cron rewrites the table when a registry moves):

Checkouts (this repo vs sisters)

Checkout	What you open
hoox-sh/pyne	PYNE SoT (`hoox-pyne` **0.6.4**)
hoox-sh/pynets	Standalone `@hoox-sh/pynets` **0.2.0** (JS compile + stream)
hoox-sh/pyne-worker	Python edge `/run`
hoox-sh/pyne-agent-worker	NL authoring (**not** in this tree)
hoox-sh/axis	AXIS PWA
hoox-sh/hoox	Mesh monorepo

Evaluate contract

Flask `POST /run`, pyne-worker `POST /run`, and in-process `Runtime.run` share the evaluate contract: `script` + OHLCV + `mode` → `plots` / `events` / `alerts`. PyneTS speaks the **library** form of that envelope (`RuntimeResult`), not HTTP.

Internals

Docs are authored in product trees and synced to hoox.sh:

Tree	Public base
`pynescript/docs/pyne/**`	`/pyne/docs`
`axis/docs/**`	`/axis/docs`
`hoox/docs/**`	`/docs`

Do not author in `hoox-landing-page/content/*` — that is a sync artifact.

Invariants & edge cases

1. **Python wins** language semantics. PyneTS does not invent TradingView behaviour.
2. **AXIS ≠ engine.** AXIS never embeds a closed interpreter; engines are plugins.
3. **AXIS ≠ execution.** Drawing a strategy on AXIS does not call `trade-worker`.
4. **Alerts ≠ orders.** `alert()` webhooks are not `StrategyEvent` forwards unless you point them at the HOOX gateway yourself.
5. **Do not** `pip install pyne` or `pip install pynescript` for this project. Dist name is **`hoox-pyne`**.
6. **Do not** `npm install pynets` for this project. The published package is **`@hoox-sh/pynets`** from the standalone hoox-sh/pynets repo. Python Runtime remains the oracle.
7. **`pine-worker`** is a leftover TypeScript Cloudflare experiment (hoox-sh/pine-worker). It is not a product evaluate host and has no pages on this site.

Worked examples

Pick a surface

You want	Use
Parse / lint / LSP on a laptop	`pip install "hoox-pyne[lsp]"`
Embed evaluate in TypeScript	`bun add @hoox-sh/pynets`
HTTP evaluate at the edge	Deploy **pyne-worker**
Upload then keep running (Flask/VPS/container)	`PYNE_RUNNER=1` — hosted runner
Chart + editor	AXIS PWA + Flask or Pyodide
Live CEX orders from strategy events	pyne-worker `TRADE_SERVICE` → HOOX
Chat → script	**pyne-agent-worker**

Failure modes

Mix-up	What goes wrong
Opening `pyne-worker/` or `pine-worker/` inside PYNE	Directories **do not exist** — clone the sister repos; TS library work is `@hoox-sh/pynets`
Expecting AXIS to flatten a position	No trade-worker hop
Pointing AXIS at pynets as an engine	No such engine. Use Flask / Pyodide / Worker proxy
`pyne run` vs `Runtime.run` vs `pynets run`	Different defaults — see modes

See also

- PYNE index
- PyneTS
- pyne-worker
- pyne-agent-worker
- Modes
- Evaluate contract
- AXIS
- HOOX
- HOOX live trading
- AXIS and HOOX

78 · REFERENCE

COMPATIBILITY GUARANTEE

Grouped map of what PYNE implements for Pine Script™ v5/v6 — working, partial, residual, and by-design out of scope. hoox-pyne 0.6.4.

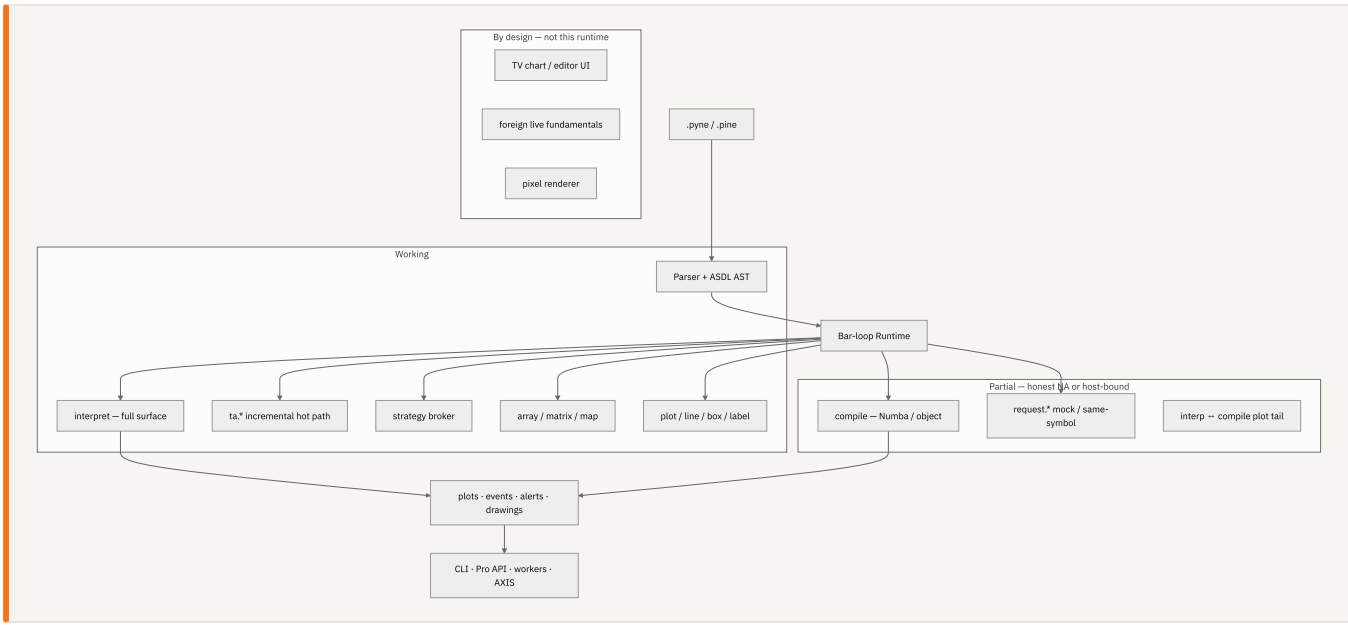
COMPATIBILITY GUARANTEE

Abstract

PYNE implements Pine Script™ **v5/v6 language core** as an inspectable pipeline: parse → AST → bar-loop evaluate. The map below is the product-facing compatibility surface for **hoox-pyne 0.6.4**. It is **not** TradingView® platform identity (chart host, proprietary data, editor-only UI) and **not** bit-identical results vs the hosted platform on every script or bar.

Name-level dispatch lives in Pine v6 surface (`docs/pine_v6_full_surface_inventory.md`). Gaps: Missing features. Checkmarks: Implementation status.

Conceptual model



Legend

Mark	Meaning
Working	Implemented, usable, covered by first-party tests
Partial	Callable but mock, host-bound, or incomplete vs hosted Pine
Residual	Known open hole on an otherwise landed surface
Out of scope	Intentionally not a TV platform clone

Dispatch snapshot (2026-07-25): **640** callables, **0 missing** vs the public TV v6 function list (434 symbols). A registered name is not the same as hosted-platform semantics.

```
Language, series, TA hot path, collections, strategy broker, drawings, alerts, libraries, interpre

`request.*` feeds (honest `na`), compile eligibility, compile alerts, PyneTS vs Python oracle.

Plot MISMATCH corpus tail (P1p), `PYNE_SERIES_RING` default off.

Pixel chart, editor UX, foreign live data, bit-identical TV bars.
```

Interface surface

Corpus (set01–04 · 2477 scripts · 2026-08-09)

Open-source sets, **not shipped** in git. Not a TradingView® platform score.

Suite	Rate	Notes
Parse + unparse	**99.96%** (2476/2477)	Residual: one intentional invalid line-wrap demo
Runtime interpret (50 bars)	**100%** excl. EXPECTED_FAIL	2466 OK + 11 listed demos
set01 Runtime	**249/249**	Interpret (and prior compile sweep)

EXPECTED_FAIL are path-listed demos (`runtime.error` library guards, lower-TF security, pathological loops) — not silent suppression.

Grouped map

```
| Area | Status | Notes |
| --- | --- | --- |
| ANTLR4 grammar v5/v6 | Working | Multiline strings, soft keywords, bitwise, typed UDF returns (`
| Parse → unparse | Working | Structural fidelity; whitespace may differ |
| `var` / `varip` / `:=` | Working | Execution-site init; start-of-bar carry |
| Functions, methods, type annotations | Working | |
| Control flow, `switch`, `once`, enums | Working | Strict bool; `na` is false in conditions. `onc
| UDTs + `.new` | Working | Field defaults on compile (0.3.8) |
| Libraries `export` / `import ns/Name/ver` | Working | In-process `register_library_source`; `Run
| String interpolation, tuple unpack, method chaining | Working | |
| Series `[]` history | Working | OOB / negative → `na` (not wrap) |
| `timeframe.change` | Working | Intraday fixed-width buckets; D/W/M on the exchange calendar (0.6



| Area | Status | Notes |
| --- | --- | --- |
| `pynescript.runtime.Runtime` | Working | Package SoT; `backend.runtime` re-exports |
| Modes `interpret` / `compile` / `auto` | Working | Defaults differ by surface — [modes](/pyne/do
| Series caps | Working | `PYNE_SERIES_CAP` default **on** |
| Incremental `ta.*` | Working | Default **on**; `PYNE_TA_INCREMENTAL=0` full recompute. **0.3.12*
| Derived OHLCV skip | Working | Unused `hl2`/`hlc3`/`ohlc4`/`tr` not written; `ta.vwap` still upd
| `timeout_seconds` | Working | Library / edge / Flask `/run` + `/run/batch` (omit = no budget) |
| Ring buffer | Residual | `PYNE_SERIES_RING` default **off** |



| Area | Status | Notes |
| --- | --- | --- |
| MAs | Working | `sma`/`ema`/`rma`/`wma`/`hma`/`vwma`/`kama`/`dema`/`tema`/`alma` incremental |
| Oscillators | Working | `rsi`/`macd`/`stoch`/`cci`/`cmo`/`tsi`/`roc`/`wpr`; **0.3.12** increment
| Volatility / structure | Working | `atr` (Wilder `rma(tr)`), `bb`, `kc`, `stdev`, `highest`/`low
| Volume | Working | Incremental `obv`/`wad`/`wvad`/`cmf`/`klinger`/`mfi`/`vwap`/`nvi`/`pvi` |
| Supertrend | Working | Locked mid±factor·ATR (inc ≡ compile ≡ Numba). TV band ratchet is out of



| Area | Status | Notes |
| --- | --- | --- |
| `array.*` | Working | Negative indices; UDT `sort_field` + `binary_search*` |
| `matrix.*` | Working | LA (`det`/`inv`/`eigen`…), predicates, `sort_field` |
| `map.*` | Working | |



| Area | Status | Notes |
| --- | --- | --- |
| `entry` / `close` / `cancel` / events | Working | `StrategyEvent` stream |
| `exit` stop/limit + OCA | Working | OHLC path, not tick path |
| `profit` / `loss` | Working | Ticks × mintick from entry avg |
| `from_entry` / `qty_percent` | Working | Interpret + compile |
| Trail | Working | `trail_offset` / `trail_points` / `trail_price` on OHLC high/low (dual-host).
| Risk cascade | Working | `allow_entry_in`, max size/drawdown/cons-loss-days/intraday |
| Open/closed trade fields + MAE/MFE | Working | Extremes from bar high/low |
| Pending-fill average (pyramiding ≤ 0) | Working | F2 |
| Licensed broker / exchange fills | Out of scope | In-process model only |
```

Area	Status	Notes
`plot` / `plotshape` / `plotchar` / `plotarrow` / `hline` / `bgcolor` / `barcolor` / `fill` / `p
`line` / `box` / `label` / `table` / `polyline` / `linefill`	Working	Coordinates snapshotted
`max_*_count` GC	Working	Interpret registry
`force_overlay`	Working	Export payload
Pixel paint	Out of scope	AXIS / clients consume series + drawings
Compile `hline`/`fill`/`bgcolor`/`plotshape` keys	Working	First-party key sets match interpr

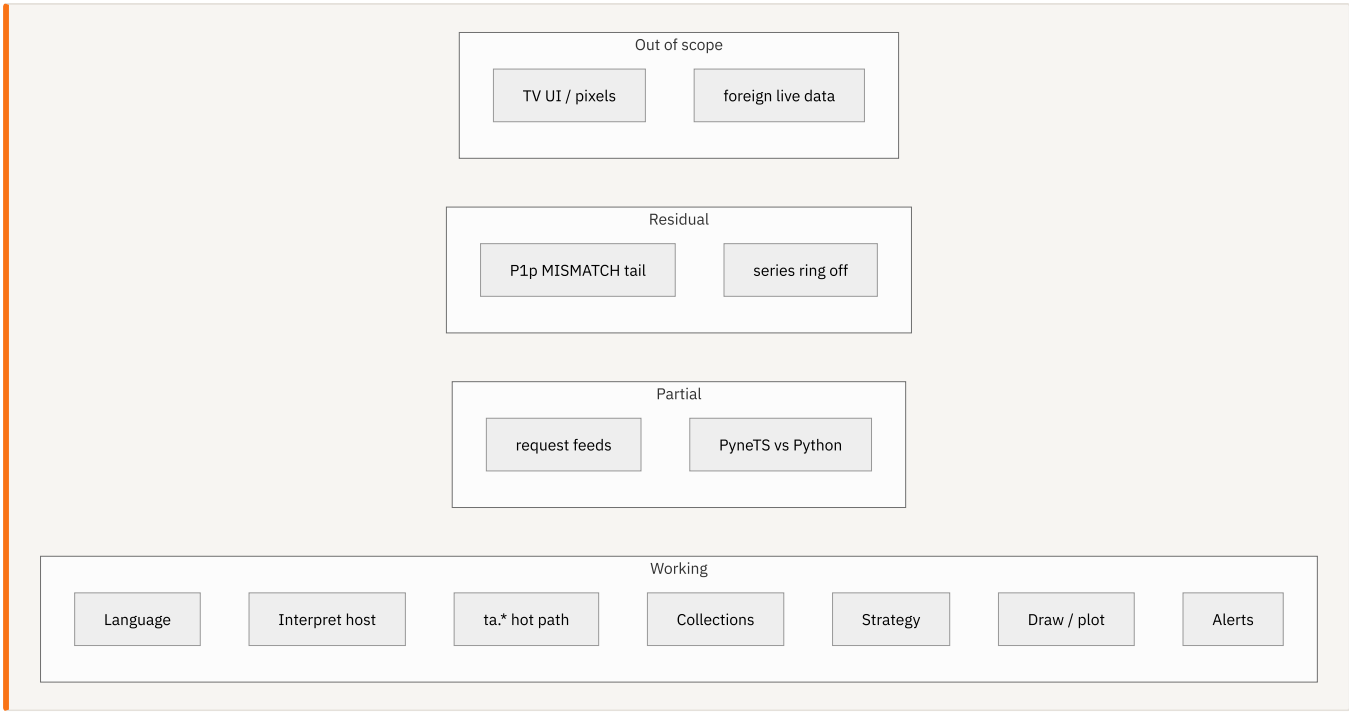
Area	Status	Notes
Same-symbol simple OHLCV / HTF	Working	Allowlisted `ta.sma`/`ema`/`rsi`/`atr`/`wma`/`rma` or
Foreign / complex `request.security`	Partial	Resolves to **`na`** (no invented chart-close)
`request.footprint` / financial / economic	Partial	Mock / feed-scaled; not a live vendor
`input.*` including `active` / `enum`	Working	Values + metadata
Bid/ask omitted	Partial	`na` when host does not supply
Real multi-symbol market data (B1)	Out of scope	Needs a host adapter

Area	Status	Notes
`alert()` / `alertcondition()`	Working	Frequency rules; `/run` export
L2 webhooks	Working	`webhook_url` / `ALERT_WEBHOOK_URL`
Compile alerts	Partial	Interpret is the alert oracle
Numba + object-mode compile	Working	`mode=auto` falls back; `import` / `request.*` / `inputs
Warm compile / disk IR	Working	Prewarm CLI/API; cache meta v16
Interp ↔ compile plot parity	Working	Harness + first-party goldens + official TV builtins **
CLI / LSP / Pro API / Docker	Working	`pyne` / `pyne-lsp`; Flask `/run`
Flask `/run` `libraries`	Working	Max 32; also `/run/batch`
Flask `timeout_seconds`	Working	Optional on `/run` and `/run/batch`; omit / `≤ 0` = no timeo
`timeframe.change`	Working	Intraday fixed-width; D/W/M exchange calendar. Interpret + compil
`POST /optimize` / `pyne optimize`	Working	Strategy `input.*` search — not a Pine builtin

Surface	Status	Notes
Python Runtime	Working	Language oracle
PyneTS `@hoox-sh/pynets` 0.2.0	Partial	Interpret + JS compile on npm; Python Runtime is the
`pynets/` submodule in this repo	Partial	Pin **v0.2.0** — same surface as npm (interpret + J
pyne-worker	Working	Thin Python CF host; vendors engine (may lag 0.3.16). Docs: [pyne-worker
pyne-agent-worker	Working	NL authoring; optional `/run` validate. Docs: [agent](/pyne/docs/a

Residual and out of scope (compact)

Item	Class
Interpret↔compile value `MISMATCH` (optional set0x community corpus)	Residual (P1p official builtins closed)
`PYNE_SERIES_RING` default-on	Residual (flagged off)
Pixel chart / editor word-wrap / TV UI	Out of scope
Foreign live fundamentals without a feed	Out of scope
TV Supertrend band ratchet	Out of scope
Tick-path trail / broker	Out of scope
Bit-identical every recursive smoother vs TV	Out of scope
Parallel bars / whole-script vectorization	Out of scope



Dual-host plot parity

Same script + OHLCV under `mode="interpret"` and `mode="compile"`; series compared with nan-aware allclose (`rtol=1e-5`, `atol=1e-6`).

Artifact	Role
`scripts/compare_interp_compile.py`	Corpus harness → `.cache/interp_compile_parity.json`
`tests/test_interp_compile_parity.py`	Always-on smoke
`tests/test_first_party_ta_goldens.py`	ATR / Supertrend / Keltner dual-host

Known **contract** differences (not silent value bugs):

1. Foreign / complex `request.*` → `na` on both hosts when no feed.
2. First-party `hline` / `fill` / `bgcolor` / `plotshape` keys match; harness `--ignore-hline-keys` / `--ignore-fill-keys` remain optional for leftover corpus noise.

3. Pivot-starved Auto Fib → matching `runtime.error` (`both_error_same`), not invented pivots.

Internals

Artifact	Role
`docs/compatibility_guarantee.md`	Historical long-form; this page is the map
`COMPATIBILITY.md`	Repo-root distillation of this map
`docs/pine_v6_full_surface_inventory.md`	Name-level dispatch tables
`docs/known_divergences.md`	Semantic deltas vs public Pine docs
`tests/fixtures/parity/`	First-party strategy fixtures

Invariants & edge cases

1. **Parse success ≠ evaluate success.** Round-trip can still depend on `request.*` data.
2. **Dispatch success ≠ hosted semantics.** 0 missing vs the public function list does not mean TV-identical bars.
3. **Interpret is the oracle.** Compile matches where eligible; `mode=auto` falls back.
4. **Plots are not a renderer.** Registry + series export; pixels are AXIS / clients.
5. **Floating-point is statistical** for some recursive smoothers — Numerical validation.
6. **Libraries** need in-process registration; there is no TV CDN.

Worked examples

Round-trip

```
from pynescript.ast.helper import parse, unparse

src = open("strategy.pine").read()
assert unparse(parse(src))  # structural; whitespace may differ
```

Interpret ↔ compile

```
python scripts/compare_interp_compile.py --bars 1000 --limit 50
python scripts/compare_interp_compile.py --ignore-hline-keys --ignore-fill-keys
pytest tests/test_interp_compile_parity.py -q
```

Failure modes

Observation	Read as
Parse error on a new TV release-notes feature	Surface gap — Missing features
Numerical drift on a custom smoother	Bar-mode vs list-mode; `na` ; known divergences
Interpret vs compile series mismatch	Foreign `request.*` , UDF last-assign `na` , leftover corpus keys
Strategy equity ≠ TV	Commission / slippage / OHLC vs tick path
`import` unresolved	Library not in `libraries=` / registry
`/run` 400 `UNKNOWN_FIELDS`	Extra key not in `RUN_SCHEMA`

See also

- Implementation status
- Missing features
- Pine v6 surface inventory
- Numerical validation
- Runtime modes
- Roadmap

IMPLEMENTATION STATUS

Checklist-style status of Pine Script™ series, strategy, builtins, and language constructs in PYNE.

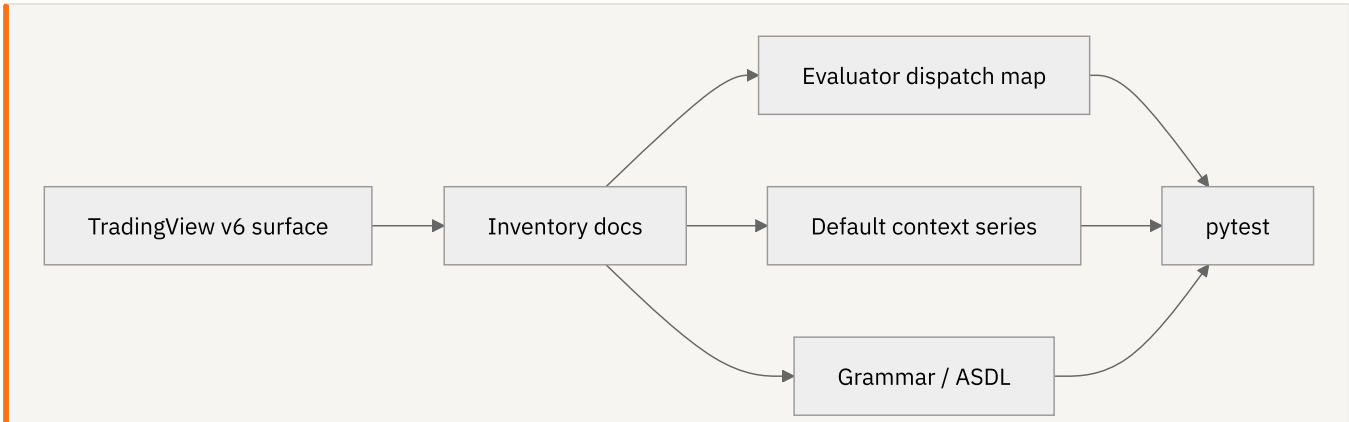
IMPLEMENTATION STATUS

Abstract

This annex summarizes the **feature matrix** maintained in `docs/pinescript_implementation_status.md`: a large ✓/✗/◐ inventory of series variables, strategy fields, and related surfaces. It is the human-readable twin of the full surface inventory (dispatch-oriented, auto-countable).

Status here is **implementation judgment**, not marketing completion percentage.

Conceptual model



Interface surface

Legend

Mark	Meaning
✓	Implemented and usable
◐	Partial / stub / mock
✗	Missing
⚙	By design out of scope
□	Editor/platform N/A

Component rollup

Component	Posture (2026-07 consolidation)
Parser (ANTLR4)	Complete for v5/v6 core + multiline / export / `once` **(0.4.4)**
Evaluator	Full bar-loop with var/varip, reassignment
Builtins	224+ historical count; **640 callables** on live dispatch inventory (2026-07-25)
TA	150+ `ta.*` keys in inventory
Collections	array / matrix / map complete on exercised surface
Strategy events	`StrategyEvent`, parity corpus, risk enforcement
Drawing / plot	Registry effects; AXIS is external
Linter	Present (`pynescript lint`)
Data providers	Mock, Yahoo, AlphaVantage, CCXT + datafeed wiring
LSP	Diagnostics, completion, hover, format, symbols, defs/refs, …
PyneTS	`@hoox-sh/pynets` **0.2.0** (npm + this repo's `pynets/` pin **v0.2.0**) — interpret + JS compile. Python Runtime remains the oracle — see PyneTS
pyne-worker	Python Cloudflare evaluate host (sister repo) — docs
pyne-agent-worker	NL authoring host (sister repo) — docs
Numba compile path	MVP + object-mode fallback; **0.3.16–0.3.17** UDT/switch/matrix/drawing/round/color residual recovery
Incremental `ta.*`	Default on (`PYNE_TA_INCREMENTAL`); **0.3.10** volume `obv` / `wad` / `cmf` / `klinger` ; **0.3.11** `nvi` / `pvi` ; **0.3.12** `aroon` / `dpo` / `donchian` / `kst`
Interpret bar-loop	**0.3.10** dispatch inlining + unused derived-series skip

Series & context (illustrative ✓ families)

Fully listed in the source doc; representative groups:

- **Price:** `open` / `high` / `low` / `close` / `volume` / `hl2` / `ohlc4` /…
- **Time:** `year` … `second` , `time_close` , `timenow` , …
- **Barstate / chart:** `bar_index` , `barstate.*` , `last_bar_*`
- **syminfo.*** ticker, mintick, session, fundamentals fields tracked as ✓
- **session.*** / **dividends.*** / **earnings.*** context fields tracked as ✓
- **strategy.*** position, trades, equity, risk, OCA, commission constants

July 2026 enhancements called out in source

- Full strategy event emission with bar context
- `strategy.long` / `strategy.short` constants
- `var` / `varip` + `:=` reassignment
- Parity fixtures for cross-implementation validation
- Risk helpers and performance series expansion

Internals

Path	Role
`docs/pinescript_implementation_status.md`	Exhaustive checklist (do not paste wholesale here)
`docs/pine_v6_full_surface_inventory.md`	Dispatch-centric inventory
`src/pynescript/ast/evaluator/builtins/`	Per-namespace implementations
`scripts/regenerate_inventory_summary.py`	Inventory regeneration aid

Invariants & edge cases

1. **Checklist lag:** a ✅ may trail a commit by a day; prefer tests when shipping.
2. **Context fields may be defaults** rather than live exchange feeds.
3. **Partial stubs still appear callable** — inventory marks ⚠ when docstrings/heuristics say mock.
4. **Namespace counts ≠ quality.** Many drawing setters are thin mutators; TA is heavier.

Worked examples

Smoke a strategy series field

```
from pynescript.ast.helper import parse
# evaluate via Runtime / library API with OHLCV — see runtime docs
```

Regenerate inventory summary

```
python scripts/regenerate_inventory_summary.py
```

Failure modes

Symptom	Action
Doc says ✅, runtime AttributeError	File bug; fix dispatcher or demote status
Partial mock accepted silently	Assert data_feed wiring in tests
Status doc conflicts inventory	Prefer live dispatch + tests

See also

- Compatibility map — grouped working / partial / residual / out of scope
- Missing features
- Pine v6 surface
- Roadmap

MISSING FEATURES

Known gaps and closed Pine Script™ v6 items — 0.6.4 convert-to-v6; 0.6.3 hosted runner; 0.6.2 P1p official builtins 0 MISMATCH; 0.6.1 correctness follow-up on 0.6.0 parity.

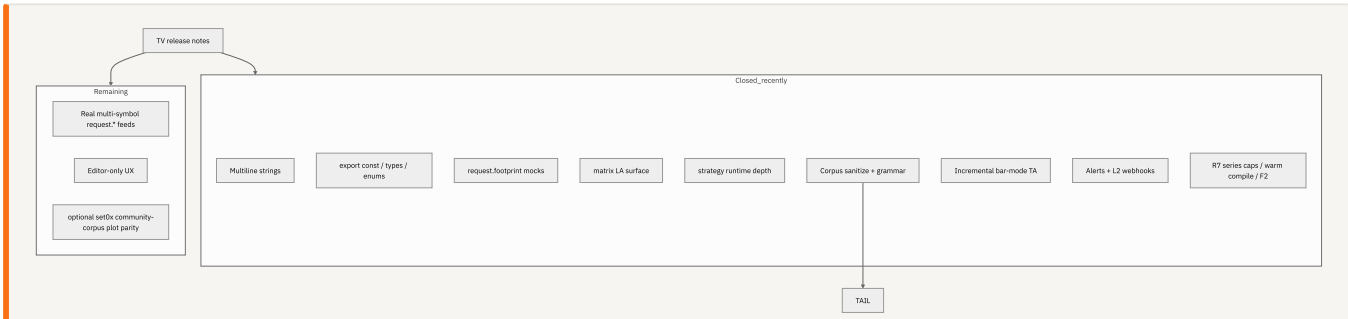
MISSING FEATURES

Abstract

“Missing” in PYNE is a **moving frontier**. As of **2026-09-14** (`hoox-pyne` **0.6.4**), core v6 support remains very high (~99%+ for core language), with post-launch TV items closed (multiline strings, library `export const`, footprint mocks, risk enforcement, matrix LA, compile-mode strategy, drawing GC, …) **plus** open-source corpus parse/Runtime hardening, **incremental bar-mode TA** (including 0.3.10–0.3.12 kernels), **alert engine + L2 webhooks**, **series caps**, **warm-compile**, **pending-fill** averaging, **`timeframe.change` (intraday buckets + 0.6.0 D/W/M exchange calendar)**, **package Runtime SoT** (`pynescript.runtime`; `backend.runtime` is a shim), **compile object-mode residual recovery** (UDF locals, nopython `None`/unicode, matrix handles, UDT returns, `x = switch`, drawing copy, `int(na)`, `round()` extras, import stubs, color type dispatch, nested UDT methods), **type-aware intraday risk enforcement on both brokers**, **intrabar `var` rollback on realtime ticks**, **`request.security` barmerge on HTF resample paths**, and **linter legacy-rule retirements**. Remaining gaps cluster into **optional set0x community-corpus plot parity (P1p official builtins closed)**, **by-design platform differences**, and **editor-only features**. Incremental `ta.nvi` / `ta.pvi` / `aroon` / `dpo` / `donchian` / `kst` and Supertrend mid±factor·ATR goldens have landed.

Authoritative narrative: `docs/missing_features.md`. Canonical IDs: `docs/ROADMAP.md` (**2026-08-17**).

Conceptual model



Interface surface

Recently completed (do not re-open as “missing”)

Summarized — full prose in source doc:

Area	Notes
Multiline `"""` / `'''`	Lexer + LexerBase + unparser
Library `export const` + types/enums	Parse/AST/runtime registry
UDT `sort_field` on array/matrix sort	Implemented
UDT `sort_field` on `array.binary_search*` (August 2026)	Implemented (interpret + compile, 0.3.6)
`once` conditional structure (August 2026)	Implemented in **0.4.4** — parse/unparse, interpret (confirmed-bar latch + realtime re-fire), compile historical flags. Statement-only.
`input.*.active`	Metadata-driven
Dynamic `request.*`	Shared resolvers; mock/feed scaling
Enums runtime + input.enum	Type kind + LSP partial
Strategy open/closed trades, risk, OCA, commission	Golden tests
Drawing `*.all`, plot registry, **`max_*_count` GC**	Effects recorded; GC on package + Pro + AXIS
Numba compile + object mode + **warm-compile (H2)**	Disk IR cache, prewarm API/CLI, deploy defaults
Compile object-mode residuals (0.3.16–0.3.17)	UDF locals / free-series, nopython `None` / `timestamp`, matrix handles, UDT returns, `x = switch`, drawing copy, `pine_int(na)`, sanitize chrome, `round()` extras, import stubs as `na`, color type dispatch, nested UDT methods
Official matrix LA & predicates	det/inv/eigen/...
0 missing vs public TV v6 function reference list (434 symbols, 2026-07-25)	
Runtime `_pine_defs_locked` + append-only `current_series`	Host hygiene (backend + pyne-worker)
Incremental `ta.*` (MAs, oscillators, `bb` / `kama` / `stochrsi`, volume `obv` / `wad` / `cmf` / `klinger` / `nvi` / `pvi`)	Bar mode; `PYNE_TA_INCREMENTAL=0` to disable
Series caps (T1)	`PYNE_SERIES_CAP` default ON + goldens
Pending-fill averaging (F2)	Pyramiding ≤ 0; interpret + compile broker goldens
Alert engine + **L2 webhooks (L2)**	Dual-host `/run` export, `ALERT_WEBHOOK_URL` / `webhook_url`
`request.security` foreign-na policy	Foreign/complex → `na` both backends; same-symbol OHLCV passthrough
Plot parity harness (P1p)	`scripts/compare_interp_compile.py`, first-party goldens + official TV builtins **0 MISMATCH**
`fill()` export for AXIS	series + `plot_meta` / compile `__drawings`

Still incomplete / by design

Item	Status posture
Real (non-mock) multi-symbol `request.*` market data (**B1**)	⚙ By design for library core; feeds optional. **Foreign-na policy landed** (no invent).
`request.security` foreign data	⚙ **Not filled with real series** — emits `na` for foreign/complex (honest). Same-symbol simple OHLCV may lower.
Auto Fib / pivot-heavy scripts	⚙ Need real pivot structure (or registered ZigZag lib). Flat synthetic bars → intentional `runtime.error` (insufficient pivots); both hosts should agree (`both_error_same` in parity harness).
Pixel chart host	External (AXIS / clients)
Some unlimited-history / trim edge cases	High-level support; exotic TV behaviors may differ
Editor word-wrap / UI-only release notes	□ N/A
PyneTS full builtin parity (**L3**)	⚙ interpret + JS compile landed in `@hoox-sh/pynets` 0.2.0; Python remains oracle
Bit-identical every recursive smoother vs TV	Numerical bounds, not bits
Dual Runtime host copies (**H1**)	✓ **largely done** — package `pynescript.runtime` SoT + `backend.*` shims + pyne-worker thin wrap. Residual = worker-only extras (logs/profile, CF first-plot), not a forked bar loop
Warm-compile product path (**H2**)	✓ SLOs / prewarm / deploy IR cache defaults (2026-08)
Runtime residual (**C1**)	✓ set01–04 Runtime interpret **100%** excl. EXPECTED_FAIL (2026-08-09); third-party corpus not shipped
Series cap (**T1**)	✓ `PYNE_SERIES_CAP` default ON
Residual TA inc (**T2**)	✓ R7 bb/kama/cmo/stochrsi + wma/hma/linreg; **0.3.10** `obv` / `wad` / `cmf` / `klinger` ; **0.3.11** `nvi` / `pvi` ; **0.3.12** `aroon` / `dpo` / `donchian` / `kst`
Pending-fill (**F2**)	✓ R7 interpret + compile goldens
Plot parity residual (**P1p**)	✓ Harness + first-party goldens + official TV builtins **0 MISMATCH**; optional set0x community corpus unmeasured
Compile leftover library stubs	⚙ Some `float(None)` / UDF arity on missing `import` libraries; INV expected-fail fixtures stay fail
TV Wilder RMA-ATR / supertrend (**F1**)	✓ interpret ATR is `ta.rma(ta.tr)` ; Supertrend mid±factor·ATR locked. TV ratchet is out of scope
Alert webhooks (**L2**)	✓ Pro API + pyne-worker; see Alerts

Post-v6 launch themes (historical gap list)

v6 launch items (dynamic requests, strict bool, enums, polylines, `log.*` , negative indices, truediv, …) are largely supported or intentionally stubbed where platform-bound.

2025–2026 monthly TV updates introduced footprint types, `active` inputs, multiline strings, `export const` , UDT sort fields — tracked and largely closed in this repo's July 2026 work.

Corpus + performance snapshot (2026-08-09)

Open-source set01–04 (**2477** scripts; measured locally, **not shipped** in-repo):

Metric	Value
Parse + unparse	**99.96%** (2476 / 2477) — residual: 1 intentional invalid line-wrap docs demo
Runtime interpret	**100%** excl. EXPECTED_FAIL — 2466 OK + **11** intentional demos
set01 Runtime	**249 / 249 (100%)**
EXPECTED_FAIL class	library `runtime.error` demos, lower-TF security guards, pathological loops, truncated mid-call scrapes
`ta_sma` bar loop (≈3.2k bars)	**~9.5×** vs full recompute
`ta_atr` bar loop	**~10×**
macd+atr+rsi+sma combo	**~8.5×**

Not TradingView® platform parity. OK% excludes only path-listed intentional demos (never soft-suppress unknown failures).

Plans: `.opencode/plans/2026-07-28-runtime-performance.md`. Skill: `.grok/skills/pynescript-perf/`. Golden: `tests/test_ta_incremental.py`, `tests/test_series_cap.py`.

Internals

Path	Role
`docs/missing_features.md`	Living gap analysis
`tests/test_v6_features.py`	v6 feature coverage
`tests/test_pine_surface_gaps.py`	Surface gap probes
`tests/test_ta_incremental.py`	Incremental TA ≡ full recompute
`tests/test_series_cap.py`	Series cap goldens (T1)
`scripts/compare_interp_compile.py`	Plot parity harness (P1p)
`docs/pine_v6_full_surface_inventory.md`	Full name-level inventory
`resource/*.g4` + builder	Grammar-level feature work
`.../technical_submodules/core.py`	`_sma_inc_update`, `_macd_inc_update`, `_atr_inc_update`, …

Invariants & edge cases

1. **Closing a syntax gap requires grammar + builder + unparser + tests**, not only an evaluator stub.
2. **Mock data can green tests while production feeds differ** — declare data source explicitly.
3. **Compile does not fill foreign `request.security` data** — multi-asset plots stay `na` on compile unless same-symbol simple OHLCV; do not treat invent-as-close as a fix.
4. **Auto Fib needs pivots** — without swing structure (or ZigZag library registration), expect structured insufficient-pivot errors, not fib lines.
5. **“0 missing vs reference list” ≠ identical semantics for every edge case.**

6. **Perf changes must keep bar-by-bar semantics** — no whole-script vectorization; prefer golden tests vs current oracle.
7. Update this annex when `missing_features.md` changes major status lines.

Worked examples

Detect multiline string support

```
from pynescript.ast.helper import parse, unparse
src = '''//@version=6
indicator("m")
s = """a
b"""
plot(1)
'''
tree = parse(src)
assert '"""' in unparse(tree) or "'''" in unparse(tree)
```

Footprint mock call

Prefer tests in `tests/test_v6_features.py` as executable specification for `request.footprint` methods.

Incremental TA parity smoke

```
.venv/bin/python -m pytest tests/test_ta_incremental.py -q
# Disable hot path if needed:
PYNE_TA_INCREMENTAL=0 python -c "from pynescript.runtime import Runtime; ..."
```

Failure modes

Symptom	Likely gap class
Lexer error on triple quotes	Stale generated lexer (should be fixed on main)
Import member missing	Library export registration
Empty footprint rows	Mock seed / data_feed not configured
Risk not blocking entry	Old runtime path; ensure risk kwargs applied
Runtime TIMEOUT on method-heavy scripts	Missing `_pine_defs_locked` on host (fixed 2026-07-28)
Corpus PARSE_FAIL mid-block	Truncated scrape — not a grammar hole

See also

- Implementation status
- Pine v6 surface
- Roadmap
- Compatibility
- Technical analysis (`ta.*`)

- Alerts
- Interpret ↔ compile parity

PINE V6 SURFACE INVENTORY

Summary of the full Pine Script™ v6 surface inventory — dispatch counts, namespaces, status schema — with pointer to the large source document.

PINE V6 SURFACE INVENTORY

Abstract

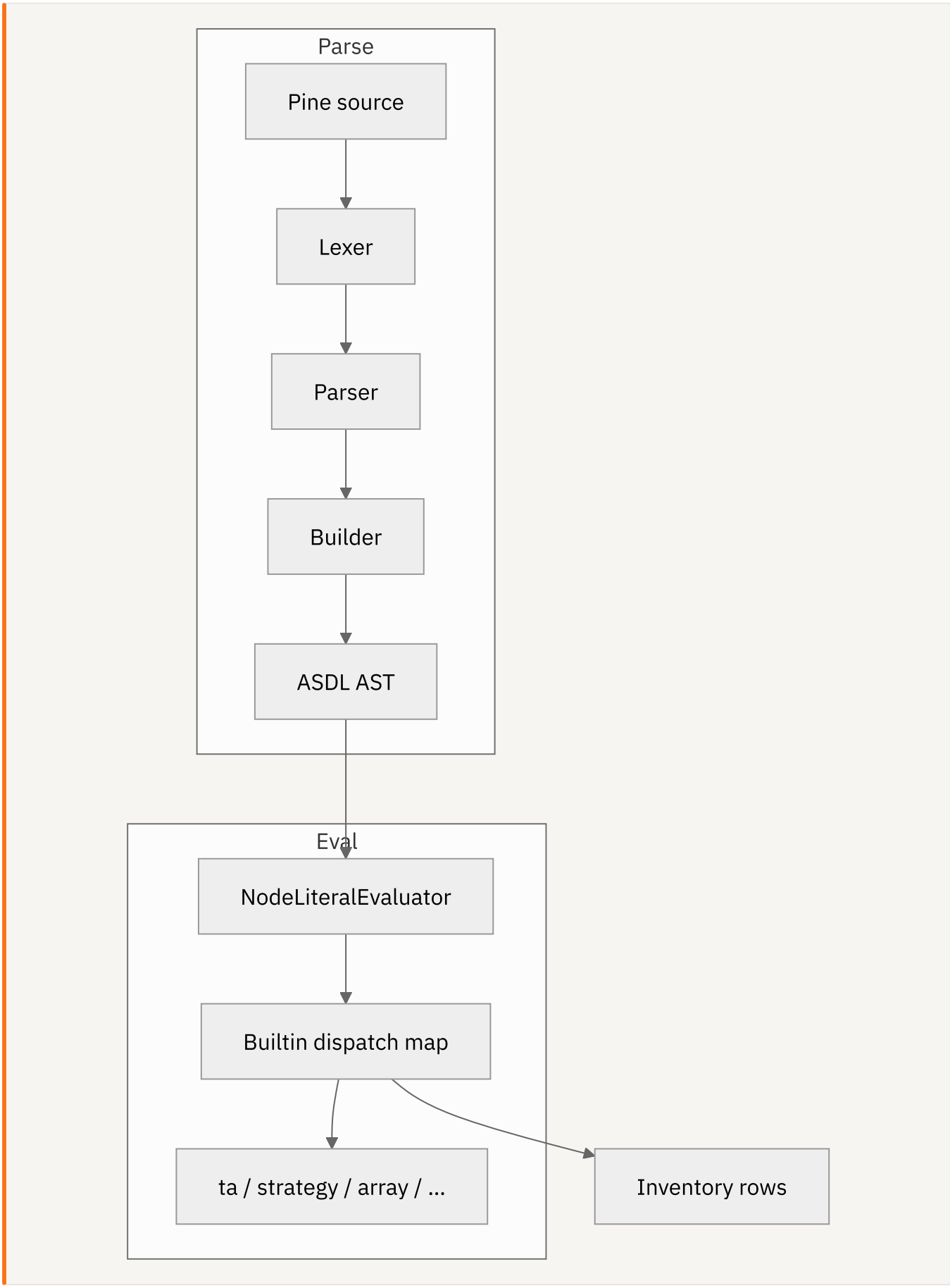
The **full surface inventory** enumerates every registered evaluator builtin, major series/variable, language construct, and known gap against Pine Script™ v6. It is deliberately large (tens of thousands of tokens when fully expanded). **This page is the navigable summary**; the exhaustive tables live in-repo at:

`docs/pine_v6_full_surface_inventory.md`

Generated snapshot referenced here: **2026-07-25**, from live `NodeLiteralEvaluator` dispatch, `builtin_metadata.json`, and design docs.

Do **not** paste the entire inventory into product docs — regenerate and link.

Conceptual model



Interface surface

Status schema

Status	Definition
✅ implemented	Handler/parser path exists and is usable
⚠️ partial	Mock/stub, incomplete semantics, or metadata-only
❌ missing	Expected on TV v6; not resolving
⚙️ by design	Intentionally not full TV platform
⬜ N/A	Editor/UI-only

Each inventory row also carries: **Name**, **Namespace**, **Kind**, **Metadata** (LSP JSON present?), **Source** (e.g. `dispatch`), **Notes**.

Summary counts (2026-07-25)

Metric	Count
Dispatch builtins (callable)	**640**
Dispatch partial-heuristic (stub/mock docstring)	8
Top-level namespaces	60
Official TV v6 function reference list	434 symbols — **0 missing** in dispatch

Top namespaces by dispatch keys

Namespace	Count
`ta`	159
`matrix`	74
`strategy`	70
`array`	56
`box`	31
`label`	28
`table`	26
`math`	24
`line`	22
`str`	19
`input`	14
`map`	11
`request`	11
`footprint`	9
`ticker`	9
(remaining namespaces)	smaller

Kinds covered

`function` · `series/var` · `constant` · `declaration` · `control` · `operator` · `type system` · `literal` · `semantics` · `export` · `import`

Internals

Path	Role
`docs/pine_v6_full_surface_inventory.md`	Full tables + architecture diagrams
`scripts/regenerate_inventory_summary.py`	Regeneration helper
`scripts/generate_builtin_metadata.py`	LSP metadata from live surface
Evaluator `_build_builtin_map()`	Source of dispatch truth

How to use the full file

1. Open `docs/pine_v6_full_surface_inventory.md` in the repo or Sphinx/site mirror.
2. Search by namespace (`## ta`, `strategy`, …).
3. Trust **dispatch** source rows for “is there a callable?”; read **Notes** for mock/feed caveats.
4. After large builtin PRs, regenerate summary counts before quoting them publicly.

Invariants & edge cases

1. **Inventory size is a feature** — summarization must not drop □ rows when claiming coverage.
2. **Metadata presence ≠ semantic completeness.**
3. **Partial heuristic count (8)** is docstring-based — real partials may be higher.
4. **Language constructs** (control/export) may be parser-complete without every runtime edge.

Worked examples

Quote coverage honestly

> "Against the public Pine v6 function reference (434 symbols), pynescript registered **0 missing** dispatch entries as of 2026-07-25; 640 total callables across 60 namespaces. See `docs/pine_v6_full_surface_inventory.md`."

Programmatic check (sketch)

```
# regenerate after evaluator changes
python scripts/regenerate_inventory_summary.py
```

Failure modes

Mistake	Consequence
Pasting 134KB inventory into Mintlify page	Unreadable docs; broken UX
Citing counts without date	Stale marketing
Equating dispatch hit with TV gold	False precision claims

See also

- Implementation status
- Missing features
- Compatibility
- Runtime builtins

NUMERICAL VALIDATION

Methodology and error bounds for PYNE technical and math builtins versus reference Pine Script™ implementations.

NUMERICAL VALIDATION

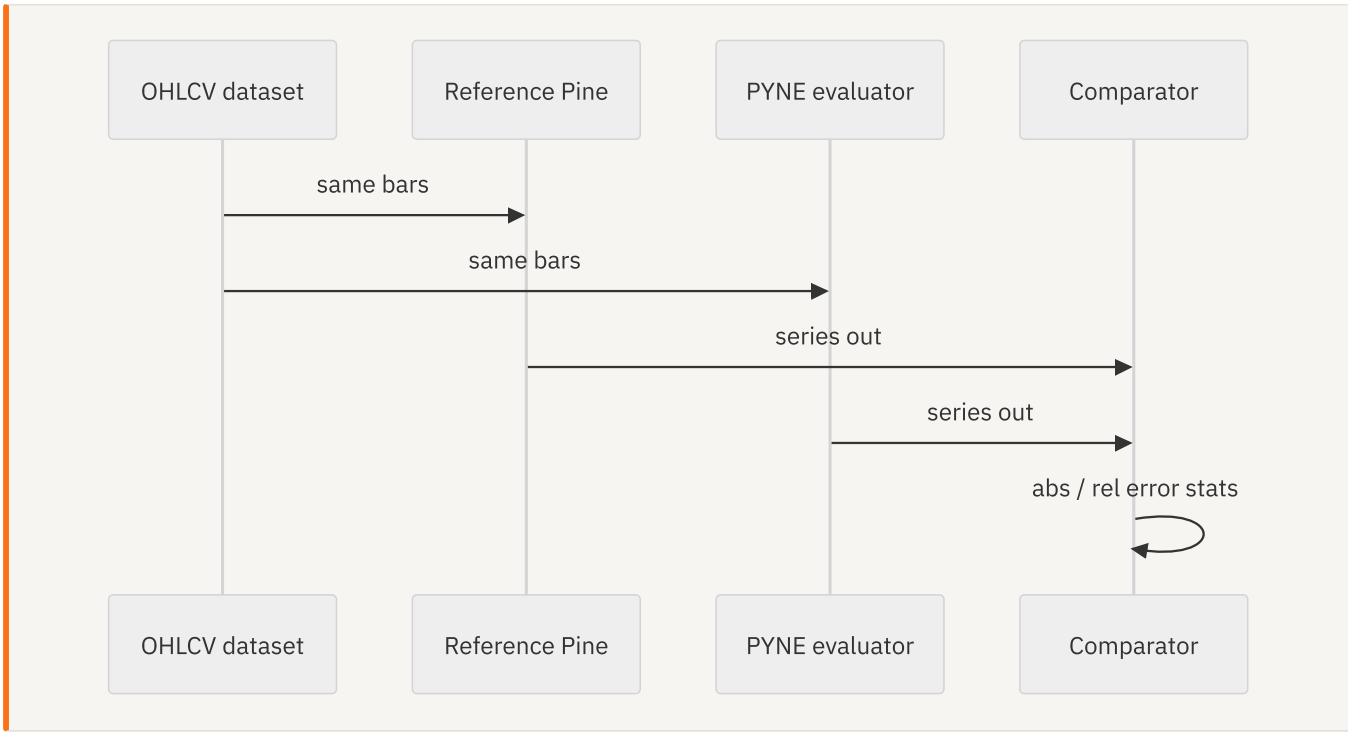
Abstract

PYNE's technical and mathematical builtins are validated for **numerical agreement** with reference Pine Script™ implementations under IEEE 754 realities. The long-form report (`docs/numerical_validation_report.md`, v1.0, Nov 2025) documents methodology, per-family error tables, and edge cases.

Headline (from that report): max observed relative error **0.0022%** (ADX, extreme volatility); average error **< 0.0005%**; deterministic ops (SMA, OBV, discrete math) often **exact**.

Treat percentages as **report-era measurements**, not eternal SLAs — re-validate after changing smoothers or bar-mode semantics.

Conceptual model



Interface surface

Acceptable thresholds (report)

Band	Relative error	Verdict
Excellent	< 0.001%	Pass
Good	< 0.01%	Pass
Acceptable	< 0.1%	Review
Unacceptable	≥ 0.1%	Fail

Metric definitions

- **Absolute error:** `|ref − pyne|`
- **Relative error:** `|ref − pyne| / |ref| × 100%` (guard zero refs)
- **Max / mean / std** over bars and scenarios

Category highlights

Family	Notes
Moving averages	SMA exact; EMA/WMA/... tiny FP error
Oscillators	RSI/stoch slightly higher smoothing error still ≪ 0.01%
Trend	ADX highest in report (0.0022% max)
Volume	OBV exact cumulative; MFI inherits typical-price FP
Stats	percentrank exact; variance/stdev excellent
Math	Discrete exact; transcendentals ~1e-15 class

Dual-host formula alignment (interpret ↔ compile)

Where fixed, these kernels **share one formula contract** across interpret (evaluator / incremental) and compile (`numba_builtins`) so plot series match bit-identical or maxdiff ≈ 0 on dual-run goldens:

Builtin	Shared contract (summary)
`ta.rsi`	Wilder: SMA seed of first `period` deltas, then RMA gain/loss; first valid at bar `period`
`ta.roc`	Standard lookback % change; `na` until lookback (no early `0.0`)
`ta.wma`	Full non-`na` window required; no partial-window reweight (nested ROC/WMA safe)
`ta.cum`	Running sum; NaNs as 0; user series (`ad = ta.cum(...)`) must not alias builtin A/D
`ta.highestbars` / `lowestbars`	Negative bars-back offset; short/all-NaN → `-1`; oldest extreme on ties

Goldens: `tests/test_compiler_numba.py` (`TestInterpCompilePlotParityFixes`) and `scripts/compare_interp_compile.py`. Residual dual-host drift (if any) tends to cluster in **nested EMA seed families** (DEMA/TEMA SMA-seed vs first-value) rather than these aligned kernels — see Compatibility.

Error distribution (report aggregate)

Error range	Share (approx.)
Exact 0	~38%
< 0.0001%	~46%
0.0001–0.001%	~14%
0.001–0.01%	~2%
≥ 0.1%	0%

Internals

Path	Role
`docs/numerical_validation_report.md`	Full tables + bias analysis
`tests/test_ta_indicators_*.py`	Executable TA tests
`tests/test_indicators.py` / builtins tests	Additional numeric guards
`tests/test_compiler_numba.py`	Interpret ↔ compile formula goldens (RSI/ROC/WMA/...)
`scripts/compare_interp_compile.py`	Corpus plot-series parity harness
Bar-mode vs list-mode (`_pine_bar_mode`)	Scalar-per-bar vs full series — compare like with like

Methodology (report)

1. Generate synthetic 1k-bar OHLCV across regimes (trend, range, vol, gaps)
2. Validate with real histories (equities, crypto, FX samples)
3. Export reference outputs; run PYNE; element-wise compare
4. Search systematic bias; inspect tails

Invariants & edge cases

1. **na propagation** must match Pine truthiness — numerical compare should mask `na` pairs.
2. **Bar-mode scalars** vs full-series list mode can look like “bugs” if misaligned.
3. **Seeded mocks** (`request.seed`) stabilize stochastic feeds for regression.
4. **Recursive smoothers** accumulate FP differently across languages; bounds > bits.
5. **Extreme prices** (near-zero, huge) tested; still within excellent band in report.

Worked examples

Relative error helper

```
def rel_err(a: float, b: float) -> float:
    if a == 0 and b == 0:
        return 0.0
    denom = abs(a) if a != 0 else abs(b)
    return abs(a - b) / denom * 100.0
```

Run TA tests

```
pytest tests/test_ta_indicators_1.py tests/test_ta_indicators_2.py -q
```

Failure modes

Symptom	Investigation
Sudden ADX drift	Check Wilder smoothing / seed bars
SMA not exact	Off-by-one window or float input casts
Good unit tests, bad TV export compare	Timezone/session alignment of bars
Only bar 0 differs	Warm-up / `na` initialization

See also

- Compatibility
- Runtime technical builtins
- Implementation status

ROADMAP

PYNE forward plan — dual-host residual, runtime residual, plot parity harness; H2/T1/F2/L2 landed (2026-08).

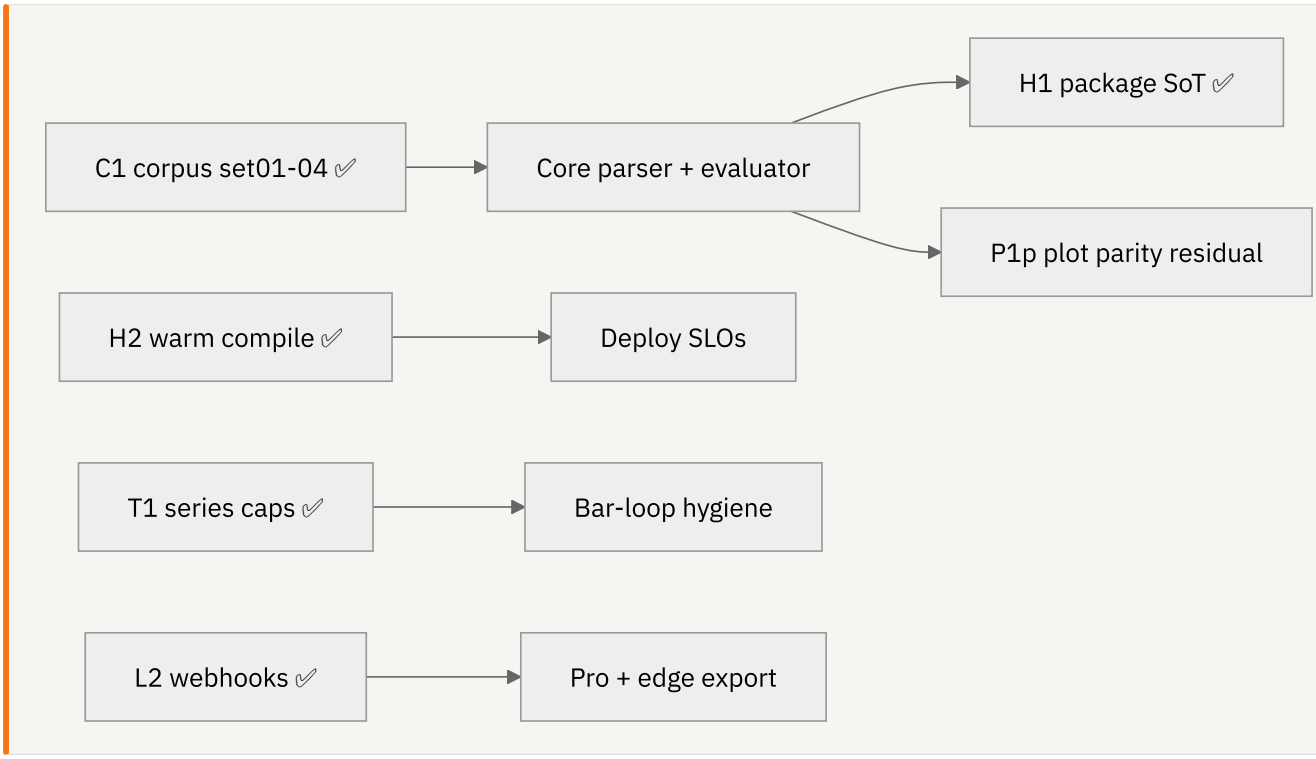
ROADMAP

Abstract

The roadmap is a **living prioritization document**, not a contract. Source: `docs/ROADMAP.md` (updated **2026-08-19**). Core v6 language/builtins are essentially closed. Product **warm-compile (H2)**, **series caps (T1)**, **pending-fill (F2)**, **L2 webhooks**, **corpus Runtime residual (C1)**, **package Runtime SoT (H1 largely done)**, **interpret Round 9 (0.3.10)**, and **compile object-mode residuals (0.3.16–0.3.17)** have **landed** (set01–04 parse **99.96%**, Runtime interpret **100%** excl. intentional demos). What remains: **H1 worker-only extras**. **P1p** official TV builtins are **0 MISMATCH** (optional set0x community corpus unmeasured). Incremental `nvi` / `pvi` (**T2**) and Supertrend goldens (**F1**) have landed.

Conceptual model

```
P0 docs honesty → P1 dual-host H1 ✅ (package SoT + shims + worker thin wrap)
                → P1 plot parity residual (P1p)
                → H2 warm compile ✅ · T1 series caps ✅ · F2 pending-fill ✅ · C1 corpus ✅
                              ↘ T2 nvi/pvi ✅ · F1 Supertrend goldens ✅
                                  → P6 long-horizon (L1 / L3; L2 ✅)
```



Interface surface

Status snapshot

Component	Status
Parser ANTLR4	✅ v5/v6 + multiline / export const
Evaluator	✅ incl. full var/varip, ReAssign
Builtins / TA / collections	✅ broad (0 missing vs public TV ref list)
Strategy events + broker (OCA, commission, risk)	✅
Numba + object-mode compile	✅ MVP+ (disk IR cache, `mode=auto`, `time_arr`, cache recovery; **0.3.16–0.3.17** UDT/switch/matrix/drawing/round/color residuals)
Pro API + auth + Docker	✅ (`mode` default `auto`; targets api / api-dev / lsp / **cli**)
PyPI **`hoox-pyne`**	✅ live **0.6.4** (pypi.org/project/hoox-pyne); CLIs `pyne` / `pyne-lsp`
CLI surface (`check` / `format` / `compile` / `run` / `prewarm` /…)	✅ + Nuitka binaries + CLI Docker image
Alert engine + L2 webhooks	✅ dual-host export + outbound webhooks
Drawing `max_*_count` GC	✅ package + Pro API + AXIS Pyodide
Package Runtime SoT (`pynescript.runtime`)	✅ 0.3.4 — Pro API + worker thin wrap
Strategy exits (`from_entry`, `qty_percent`, trail, risk cascade)	✅ 0.3.4
`request.security` HTF simple TA + foreign-na	✅ same-symbol OHLCV/HTF allowlist; foreign → `na`
Free-tier Pro API guards	✅ bar/script caps, rate, concurrency, mock-only free data
Dual-host TA goldens (ATR / Supertrend / Keltner)	✅ first-party + `test_ta_incremental` CI
Interpret dispatch + unused derived skip + volume inc (Round 9)	✅ **0.3.10** — bench @ 2000 bars: minimal 2.7×, `ta_sma` 2.0×, `ta_combo` 1.45×
Series caps (`PYNE_SERIES_CAP`)	✅ T1
Warm-compile product path	✅ H2 (SLOs, prewarm, deploy IR cache)
Pending-fill averaging (pyramiding ≤ 0)	✅ F2
Interpret↔compile plot parity	⚙ harness + goldens landed; residual MISMATCH tail
Runtime `timeout_seconds`	✅ 0.3.6 — wall-clock circuit breaker on interpret
UDT `array.binary_search*` `sort_field`	✅ 0.3.6 — interpret + compile (Pine August 2026)
`Runtime.run(..., libraries=)` / `POST /run libraries`	✅ 0.3.7+ — AXIS `import ns/Name/ver`; export finalize in 0.3.8; auto fallback keeps libs
pyne-worker 0.6.0	✅ edge `/run` libraries + plot_meta/logs; engine vendor 0.3.8
PyneTS (`@hoox-sh/pynets`)	✅ standalone TS library 0.2.0 — interpret + JS compile; Python remains oracle
pyne-agent-worker	✅ sister NL authoring host (`POST /v1/chat`)
Full TV platform identity	⚙ out of scope

Completed themes (do not re-plan as greenfield)

- Error message / typing hygiene
- `%%pinescript` Jupyter magic + sample data helpers
- `pynescript lint`
- Yahoo / AlphaVantage / CCXT + realtime datafeed wiring
- StrategyEvent emission + parity fixtures
- Colocated TS extra tool extracted (0.3.7). Product TypeScript library is PyneTS.
- Alert engine + dual-host export + **L2 webhooks**
- Round 7: series caps, incremental TA (bb/kama/cmo/stochrsi), parse cache, warm compile, pending-fill, drawing GC
- Round 9 / **0.3.10**: interpret dispatch inlining, unused derived OHLCV skip, incremental `obv` / `wad` / `cmf` / `klinger`

Open backlog (IDs stable)

ID	Item	Pri	Status
H1	Dual-host Runtime (package SoT + backend shims + pyne-worker thin wrap ✅; residual = worker-only extras)	P1	✅ largely done
H2	Product warm-compile (SLOs, prewarm API/CLI, IR cache on in deploy)	P1	✅
C1	Corpus set01–04 residual (parse 99.96%; Runtime 100% excl. EXPECTED)	P1	✅ 2026-08-09
P1p	Compile/interpret **plot parity** residual	P1	✅ first-party + official TV builtins **0 MISMATCH**; optional set0x unmeasured
T1	Cap `current_series` to `max_bars_back` / `_SERIES_MAX`	P2	✅ `PYNE_SERIES_CAP` default ON
T2	Incremental TA for remaining heavy kernels	P2	✅ R7 + 0.3.10 volume inc + 0.3.11 nvi/pvi + **0.3.12 aroon/dpo/donchian/kst**
F1	ATR Wilder / TV supertrend re-baseline only with dedicated goldens	P2	✅ mid±factor·ATR locked; TV ratchet out of scope
F2	Pending-fill averaging when pyramiding ≤ 0	P2	✅
L1	v5↔v6 converter maturity	P3	open
L2	Webhook alerts productization	P3	✅ Pro + pyne-worker
L3	Legacy TS Worker builtin parity	P3	sister `hoox-sh/pine-worker` only — not a product-docs target; new TS work is PyneTS
B1	Real (non-mock) `request.*` market data	—	⚙ by design; **foreign-na policy landed**

Longer horizon (not next): ML wrappers, automatic refactor, parallel/distributed eval experiments.

Priority recommendation

Horizon	Focus
Short	H1 worker extras (logs/profile); optional set0x P1p sweep
Medium	Further nested incremental TA where profiled
Long	**L1** converter maturity; TypeScript library work is PyneTS (**L2** ✅)

Landed residual notes (2026-08)

- **Plot parity:** Always-on smoke in `tests/test_interp_compile_parity.py`. Full corpus compare: `python scripts/compare_interp_compile.py` → `.cache/interp_compile_parity.json`. Flags `--ignore-hline-keys` / `--ignore-fill-keys` drop structural residuals.
- **Corpus residual (C1):** set01–04 parse **99.96%** (2476/2477); Runtime interpret **100%** excl. EXPECTED_FAIL (2466 OK + 11 intentional demos); set01 **249/249**. Third-party corpora are not shipped in git.
- **Foreign-na:** `request.security` on foreign/complex expressions → `na` both backends; `ChartOHLCVProvider` refuses non-chart symbols.

Internals

Path	Role
`docs/ROADMAP.md`	Canonical roadmap prose
`docs/PROGRESS_REPORT.md`	Historical completion narrative
`docs/COMPILER_PLAN.md`	Compile/numba plan
`scripts/compare_interp_compile.py`	Plot parity harness
`.opencode/plans/`	Engineering plans (strategy events, consolidation)
sister `hoox-sh/pyne-worker`	Python edge evaluate host — docs
sister `hoox-sh/pyne-agent-worker`	NL authoring host — docs
`pynets/` submodule	TS library pin **v0.2.0** (interpret + JS compile); Python remains oracle

Invariants & edge cases

1. **Roadmap dates lag code** — always diff against `missing_features.md` and tests.
2. **Sister repos** (pyne-worker, pyne-agent-worker) may absorb “integration” items.
3. **AXIS is a separate product tree** (`docs/axis`) — chart UX roadmap is not PYNE-core.
4. Avoid reintroducing deleted backups (`builder.py.bak`, etc.) as “plan items.”
5. Do **not** re-open **H1 / H2 / T1 / T2 / F1 / F2 / L2 / C1** as greenfield — they are marked ✅ (H1 residual = worker extras only; P1p is corpus tail) in `docs/ROADMAP.md` (2026-08-16).

Worked examples

Track a roadmap item with a test

```
# pick a gap, write failing test first, implement, update missing_features.md
pytest tests/test_v6_features.py -k footprint -v
```

Plot parity smoke

```
pytest tests/test_interp_compile_parity.py -q
python scripts/compare_interp_compile.py   # opt-in full compare
```

TypeScript library gate (PyneTS)

```
# standalone checkout — not present as pine-worker/ in this repo
cd ../pynets && bun test && bun run typecheck
```

See PyneTS.

Failure modes

Anti-pattern	Why it hurts
Planning features already ✅	Duplicated work (H2/T1/F2/L2 are done)
Roadmap without tests	Unverifiable “done”
Ignoring by-design limits	Infinite platform chase

See also

- Missing features
- Implementation status
- Interpret ↔ compile parity
- Alerts
- pyne-worker
- pyne-agent-worker
- Contributing

CONTRIBUTING

Hard constraints, workflow, and style rules for contributors and agents working on PYNE.

CONTRIBUTING

Abstract

Contributions to PYNE are welcome when they respect the repo's **mechanical invariants**: generated code boundaries, src-layout packaging, dual console scripts, test corpus costs, and encryption secrets. This page distills `CONTRIBUTING.md` and `AGENTS.md` into an operational contract for humans and agents.

Conceptual model

Interface surface

Baseline workflow

1. Fork and branch from `main`
2. Install: `make install` or hatch envs
3. Run tests: `make test` / `hatch run test:test`
4. Implement with tests (prefer TDD for non-trivial behavior)
5. Lint/format: `make lint` / `make fmt` or `hatch run lint:style` + `lint:typing`
6. Open a PR with rationale + test evidence

Official short form also lives in root `CONTRIBUTING.md`.

Everyday commands

Goal	Command
Editable install + LSP	`make install`
Full tests	`make test`
LSP only	`make test-lsp`
Backend only	`make test-backend`
Lint / format	`make lint` / `make fmt`
Fast build sanity	`make build-check`
API	`make run`
LSP process	`make run-lsp`

Hard constraints

Grammar & generated code

- **Edit only** `src/pynescript/ast/grammar/antlr4/resource/*.g4` for grammar work
- **Never hand-edit** `…/antlr4/generated/` or `…/asdl/generated/`
- Regenerate via hatch `lint:gen-parser` / project scripts; see grammar guides under `.opencode/context/`

No stale backups

Do not recreate removed backups such as `builder.py.bak` or `evaluator/builtins/technical_refactored.py`. Live `builder.py` and `technical.py` are sole sources of truth.

Future annotations

Every new Python file must start with:

```
from __future__ import annotations
```

Enforced by ruff isort `required-imports`.

Console scripts

Preferred	Alias	Role
`pyne`	`pynescript`	Click CLI
`pyne-lsp`	`pynescript-lsp`	Language Server (pygls)

Import package remains `pynescript`. Do not conflate CLI and LSP entrypoints in docs, packaging, or process managers.

Optional example-script fixture

`tests/conftest.py` only expands `pinescript_filepath` when `--example-scripts-dir` points at a local directory of `*.pine` files. **No third-party corpus is shipped** in the repository.

`tests/data/library/` is reference material, not auto-parametrized.

Builtin metadata

`builtin_metadata.json` is **generated from code**. After adding builtins:

```
python scripts/generate_builtin_metadata.py
```

Do not hand-maintain the catalog long-term.

Interpret / compile parity

When changing `pynescript.runtime.Runtime`, the compiler, or plot/series export, check **series parity** between `mode="interpret"` and `mode="compile"`:

```
# Local corpus harness (default 50 scripts × 1000 bars)
python scripts/compare_interp_compile.py --bars 1000 --limit 50

# Broader pass from a path list; ignore one-sided hline/fill keys
python scripts/compare_interp_compile.py --file-list path.txt --limit 0 --workers 4 --timeout-sec
```

Always-on unit coverage lives in `tests/test_interp_compile_parity.py` (harness helpers + a small smoke subset). Optional longer path:

```
pytest tests/test_interp_compile_parity.py -m interp_compile_full
# or: PYNE_INTERP_COMPILE_FULL=1 pytest tests/test_interp_compile_parity.py
```

Report artifact: `.cache/interp_compile_parity.json`. Value mismatches on shared `series` keys are regressions. First-party `hline` / `fill` / `bgcolor` / `plotshape` / `barcolor` / `plotarrow` / `plotbar` / `plotcandle` keys match (0.3.12); `drawings` is geometry-only. `--ignore-hline-keys` / `--ignore-fill-keys` are optional for leftover corpus noise. End-user mode semantics: Evaluate scripts — Runtime modes.

Secrets & crypto

- `scripts/build/.metadata.key` is **gitignored**
- CI must supply stable `CRYPTO_KEY` from `METADATA_KEY` / `_METADATA_KEY` for reproducible `.enc` blobs
- Never commit Fernet keys or API admin tokens

Internals (where to change what)

Want to...	Look at
Parse / unparse	`src/pynescript/ast/helper.py`
Add builtin	`src/pynescript/ast/evaluator/builtins/<ns>.py` + metadata generator
LSP feature	`src/pynescript/langserver/features/` + `server.py`
CLI subcommand	`src/pynescript/__main__.py`
Grammar	`resource/*.g4` then regenerate
Pro API route	`backend/api/preview.py`, `backend/app.py`
Runtime modes / compile fallback	`src/pynescript/runtime/host.py` (`Runtime.run`); `backend/runtime.py` is a shim
Interp/compile series parity	`scripts/compare_interp_compile.py`, `tests/test_interp_compile_parity.py`
TS library	sister repo hoox-sh/pynets / `@hoox-sh/pynets`

Style

- **Ruff** line length 120; broad rule set in `pyproject.toml`
- **Black** target `py310` (via toolchain config)
- **Mypy** strict-ish; exemptions for generated grammar, `evaluator.builtins.*`, tests
- Voice for docs: academic nerdy-cool per `docs/WRITING.md` — no empty marketing adjectives

Release (this repo)

Match root `CONTRIBUTING.md` and `.github/workflows/publish.yml` / `release.yml` / `ghcr.yml`:

1. Bump `__version__` in `src/pynescript/__about__.py` and update `CHANGELOG.md`.
2. Align `vscode-extension/package.json` when shipping the VSIX together.
3. Ensure CI is green on `main`.
4. Tag and push: `git tag vX.Y.Z && git push origin vX.Y.Z`.
5. **Publish** (`publish.yml`) builds sdist/wheel (`hoox_pyne-*.whl`) and uploads to PyPI (`PYPI_API_TOKEN` or Trusted Publishing OIDC, environment `pypi`).
6. **Build & Release** (`release.yml`) attaches Nuitka CLI/LSP binaries + VSIX to the GitHub Release.
7. **GHCR** (`ghcr.yml`) publishes `ghcr.io/hoox-sh/pyne/{cli,lsp,api}:X.Y.Z` on `v*` tags.

Dry-run PyPI: Actions → **Publish** → Run workflow → `dry_run=true`. Local: `python -m build && twine check dist/*`.

AXIS charting releases live in hoox-sh/axis (historical fork: `jango-blockchained/axis`).

PyneTS contributions

- Work in hoox-sh/pynets, not a `pine-worker/` directory in this repo
- Bun for install / test / typecheck; `bun run generate` after PYNE `.g4` changes
- Python remains the oracle — see PyneTS parity
- Do not copy PyneTS sources into this tree; work in the standalone hoox-sh/pynets repo / `@hoox-sh/pynets` package
- Python Runtime remains the oracle.

Failure modes for contributors

Mistake	Fallout
Editing generated ANTLR/ASDL	Overwritten / review hell
Skipping corpus cost awareness	10-minute “unit” tests
Hand-editing metadata JSON	Drift from dispatch
Changing compile/export without parity check	Silent chart/series drift vs interpret
Committing `.metadata.key`	Secret leak; rotate immediately
PR without lint	CI red on ruff/mypy

See also

- Local development
- CI
- PyneTS
- pyne-worker
- pyne-agent-worker
- Roadmap

- Root `AGENTS.md`, `CONTRIBUTING.md`, `CODE_OF_CONDUCT.md`

CHANGELOG

Release notes for PYNE (hoox-pyne / pynescript).

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog, and this project adheres to Semantic Versioning.

[Unreleased]

0.6.4 - 2026-09-14

Convert older Pine to v6 in the editor, LSP, CLI, and HTTP.

Added

- Editor **Convert to Pine v6**: AXIS overflow menu + command palette, VS Code `PYNE: Convert to Pine v6` / lightbulb, LSP code action, and `POST /lsp/convert`. Source-level rewrite from any older version (missing pragma = v1) — colors / `n` / timeframe idents, `ta.*` / `math.*` / `request.*` namespaces, `study(` → `indicator(`, typed `input.*()`, `iff` / `offset`. Not a semantic v6 migrator.

Changed

- `pyne convert --to 6` now runs the full v1-v5 pipeline, not only leftover `study(` / `request.*` names.

0.6.3 - 2026-09-14

Hosted script runner on Flask / VPS / container, plus `pyne runner` CLI.

Added

- Optional Flask hosted runner (`PYNE_RUNNER=1`): SQLite script registry, `POST/GET/DELETE /scripts`, `GET/PUT /cron/jobs`, `POST /cron/run`. Background bar-close polling is a second switch (`PYNE_RUNNER_SCHEDULER=1`). Docker / VPS / Cloudflare® Container pick it up via env; default remains off.
- CLI `pyne runner` talks to that API: `deploy`, `list`, `show`, `delete`, `tick`, `jobs`, `enable`, `disable` (`--url` / `PYNE_API_URL`).

0.6.2 - 2026-09-12

P1p plot-parity tail on official TV builtins.

Fixed

- Interpret UDF series parameters: `s[1]` inside a function now tracks the per-call-site argument stream when the caller passed a scalar (`src = close; updown(src)`). Official Connors RSI and the TV builtin script set are 0 MISMATCH vs compile.

0.6.1 - 2026-09-12

Correctness follow-up on 0.6.0 dual-host risk, InterpretSession, and HTF resample, plus the session API that landed after the `v0.6.0` tag.

Added

- Persistent `InterpretSession`: delta `append_bars`, realtime `update_last_bar` ticks, TA rollback, confirm-on-append.

Fixed

- `strategy.cash` as a qty-type identifier now selects cash mode on both interpret and compile (string `"cash"` was the only working form).
- Compile `begin_bar` receives bar time so day-scoped fill caps and consecutive-loss days roll on `Runtime.run(mode="compile")`.
- Pending fills are risk-gated at fill time on both brokers; covering leftover cannot reverse-open after an intraday halt.
- Session `barstate` no longer leaks across appends; confirm overwrites strategy history; `islast` is false on a confirmed bar when more bars follow; `_once_fired` rolls back with the bar-open snapshot.
- HTF `lookahead_on` leaks each bucket's **final** value from the period start on historical bars (not developing-as-of-this-bar); `close[1]` resamples previous HTF bars (same-TF falls back to chart history); bare D/W/M buckets follow `timeframe.change` (Monday weeks, `syminfo.timezone`).
- Codecov per-flag uploads set `disable_search` so reports stay isolated.

Changed

- Disk IR cache v16 (`begin_bar` bar time, calendar `chart_timezone()`, `strategy.cash` qty-type emit).
- Docs: PyneTS as standalone package (no `pynets/` submodule); retired linter codes C003/W102/W103; version stamps **0.6.1**.

Security

- wrangler 4.131.0 (sharp 0.35.4, GHSA-rgj7-g3m4-5g8c).

0.6.0 - 2026-09-10

Parity increments across strategy risk, realtime scope, timeframes, `request.security` barmerge, and the linter.

Added

- Strategy risk: `strategy.risk.max_intraday_loss(value, type)` honors percent-of-equity vs cash on both brokers; interpret enforces intraday loss plus the day-scoped `max_intraday_filled_orders` cap, with order-level fill counting on both hosts.
- Realtime: intrabar rollback of the `var` scope plus series currents on interpret realtime ticks (`varip` persists); `once` fires once from confirmed state.
- Timeframes: `timeframe.change` D/W/M on the exchange calendar (`syminfo.timezone`, UTC default; DST-aware days, Monday weeks, calendar months); compile routes calendar TFs through object mode.
- `request.security`: `gaps_on` / `lookahead_on` honored on the HTF resample paths (bucket-start na-gaps, forming-bucket reads); presence-vs-effect policy tags (`gaps_lookahead_provided` vs `gaps_applied` /

`lookahead_applied`).

Changed

- Docs: `once` on grammar/ASDL/builder/unparser, interpret + compile pages, compatibility, glossary, FAQ, missing-features. Version table **0.6.0**.
- `docs/known_divergences.md`: F-12…F-16 closed; D-02/D-04/D-06 narrowed to their documented residuals.
- Disk IR cache v14 (calendar routing changed codegen).

Removed

- Linter: retired legacy rules `C003` (indented `if` without "braces"), `W102` (histogram → plotcandle), `W103` (`var int x = na` → 0); codes stay retired so history stays greppable. LSP `C003` noise filter removed with the rule.

[0.5.0] - 2026-09-07

Performance release across the whole stack, plus several correctness fixes surfaced by the work.

Changed

- **Slotted AST nodes.** ASDL generator emits `@dataclass(slots=True)`; the shared `AST` base carries `__slots__` for evaluator call-site stamps. Removes the per-node `__dict__` (large scripts allocate tens of thousands of nodes). Regenerate via `python src/pynescript/ast/grammar/asdl/tool/generate.py`.
- **LSP typing latency (~3-5×).** `didChange` no longer runs two full ANTLR parses (the linter accepts a pre-built `tree=`), and parse+lint now run debounced (250ms, `PYNESCRIPT_LSP_DEBOUNCE_MS`) on a worker thread so the event loop stays responsive. Semantic tokens, inlay hints, hover decl maps, and workspace/symbol results are memoized per document version; completion items, fuzzy-filter fields, and categories are cached per metadata load; references/definitions dedupe is O(1) per line.
- **TA incremental coverage.** `ta.max` / `ta.min` / `ta.range` now use the incremental kernels; `ta.highest` / `ta.lowest` / `ta.wpr` / `ta.stoch` incremental use monotonic-deque sliding extremes (amortized O(1)/bar instead of O(period) scans); `ta.rci` rank computation is O(n log n) (was O(n²) per bar); `ta.vwma` incremental drops its O(period) na-scan.
- Per-bar `import` statements hoisted out of hot paths (`+=` assignment, subscript call-sites, `switch`, overload dispatch); `_commit_unwritten_history` iterates a cached `history n var` intersection.
- Parse-cache hit scrub is a specialized stack walk (was generator BFS); `Runtime._parse_script` no longer performs a second redundant full-tree walk.
- Builder: single-line token fast path skips the per-node `stop.text` substring allocation; simple string literals skip `ast.literal_eval`.
- Backend: `/run` applies a server-side default timeout (`PYNE_RUN_TIMEOUT`, 30s) so pathological scripts cannot kill a gunicorn worker; PBKDF2 key hashing is LRU-cached (~50-100ms → O(1) per authenticated request); SSRF DNS verdicts are TTL-cached; optional off-thread webhook delivery (`ALERT_WEBHOOK_ASYNC=1`); chart renderer uses the thread-safe OO matplotlib API (pyplot globals raced under `--threads 4`).
- OHLCV pack/list caches are bounded by tracked bars (`PYNE_OHLCV_CACHE_MAX_BARS`, default 2M), not just entry count.
- Strategy `position_size` / `position_avg_price` / `closedtrades` end-of-bar history is capped at 512 entries.

Fixed

- Five compiler/evaluator subtree walks used `node.__dict__.values()`, which silently collected nothing on slotted nodes and broke UDF `var`-local and history-series detection; traversal now goes through `_fields`.
- `_as_series` same-bar cache could false-hit on a different series (single interned-int fingerprint): fingerprint uses the two newest samples and the host clears the cache every bar.
- OHLCV pack cache evicted from the *list* cache, so the pack cache was never bounded.
- `didChange` without a prior `didOpen` crashed the LSP handler (now treated as an empty open); closed documents leaked into the workspace parse-error set.

0.4.4 - 2026-09-05

Added

- **`once` conditional structure** (Pine Script™, August 2026). Executes its block the first time the optional condition is true on a closed bar, then never again. Soft keyword so existing `once` identifiers still parse. Interpret honors realtime rollback (unconfirmed ticks can re-fire); compile is historical (`isconfirmed` always true).

0.4.3 - 2026-09-04

Added

- **`pynescript convert --to 5|6`** — source-level v5 ↔ v6 rewriter (`//@version`, `study(` → `indicator(` on the way to v6, bare `security(` / `financial(` / … ↔ `request.*`). Comments and string literals are left alone. Not a semantic migrator (bool-as-int, `na` tightening).

Fixed

- Compile `plot(..., display=)` bitfield is folded into `plot_attrs` so `plot_meta` carries it (follow-up to 0.4.2).
- UDF `var` + history (`x[1]`) is per call site; interpret no longer drops every-other-bar `na` vs compile (e.g. `moon_phases.pine`).
- `table.merge_cells` no-ops on `na` coordinates instead of `int(None)` on the last bar.
- Remaining CodeQL: stack traces not returned in alert-forward errors; ReDoS regexes rewritten as linear scans.

Changed

- **Pro API key store (breaking for SHA-256-only records):** keys are PBKDF2-HMAC-SHA256 (`v2:`). JSON files keyed by the raw secret are rehashed on load. Orphaned 64-hex SHA-256 records are skipped — remint those keys.
- CI: ruff E/F/W over `backend/`; VS Code VSIX packaged with `@vscode/vsce` (`@types/vscode` pinned to `engines.vscode` 1.91); `download-artifact@v8`; PyPI publish `skip-existing` so a duplicate tag run does not 400.
- Dependabot weekly updates landed (Actions, docs, vscode-languageclient 10, TypeScript 7, wrangler).

0.4.2 - 2026-08-26

Fixed

- Compile `plot(..., display=)` is stored on `plot_attrs` so `plot_meta` exports the display bitfield.

- First CodeQL pass: errors, high, and medium alerts on the tagged tree.

0.4.1 - 2026-08-26

Fixed

- Compile `plot_meta` backfills hline linestyle from `__drawings`.

0.4.0 - 2026-08-25

Changed

- **VPS origin is Hetzner `pynescript.online` (`204.168.138.51`)** — `scripts/deploy_vps.sh` defaults to SSH host `pynescript` and skips AXIS dist (PWA is on Cloudflare® Pages). API-only gunicorn behind nginx; `axis-pwa.service` is optional.

[0.3.18] - 2026-08-25

Added

- AXIS contract parity: `plot_meta` exports Pine visual params (`offset`, `histbase`, `trackprice`, `join`, `editable`, `show_last`) on interpret and compile hosts; omitted keys equal Pine defaults.
- `plotcandle` / `plotbar` export OHLC as close-primary series plus `<title>.open/.high/.low` siblings referenced from `plot_meta` (`open` / `high` / `low` / `close`), with `wickcolor` / `bordercolor`.
- Drawing/table exports gain stored-but-unexported styling: box extend/border_style/text styling, label tooltip/alignment/formatting, polyline curved/fill_color/force_overlay, table frame_width/border_color/border_width plus `merged_cells`.
- LSP builtin metadata carries canonical Pine v6 signatures for plot-family builtins.

Fixed

- Positional `plot(...)` calls misread linewidth/style indices (now linewidth = 3rd arg, style = 4th per Pine v6 signature).
- `table.clear(t, r0, c0, r1, c1)` clears only the requested range; `table.merge_cells` records ranges and exports them.
- Compile-mode `plot_meta` no longer hardcodes `linewidth=1` when source declares constant linewidth/style/offset/histbase.
- Compile IR cache no longer shares entries whose titles/kinds/attrs diverge (metadata bleed across sources).
- Positional wickcolor/bordercolor args on plotcandle/plotbar are captured (previously kwargs-only).

[0.3.17] - 2026-08-19

Fixed

- **Compile object-mode residuals** — user `round()` UDF call sites pass free/chart formals; missing library `alias.method(...)` stubs are `na` not Python `None`; same-arity `negate(float|bool|color)` dispatches by argument type so hex colors stay off float64 series; method overloads on UDT fields (`this.wins.getAvgProfit()`) pick the field's type, not the outer receiver.

[0.3.16] - 2026-08-19

Fixed

- **Compile object-mode corpus residuals (set06)** — UDF `int m = if` stays a local (not script `m_arr`); nested UDFs still receive script-level `a_arr` when a param shadows `a`; UDF `var vol` does not steal chart `vol_arr` (`get_counts` arity); `Type.new()` locals mark UDT returns so `s.strength` is not `float(dict)`; statement-form `x = switch` no longer emits invalid `x = if`; `chart.point.copy` / `box.copy` clone dicts instead of `list(None)`.
- **Numba object-mode helpers** — `None` no longer TypingErrors `numba_max` / `min` / `abs` and related scalars; `timestamp` skips a leading timezone string; `valuewhen` and running max/min fall back on object arrays; Pine `int(na)` is `na` (`pine_int`, not Python `int(nan)`); `ta.pivothigh` na lengths; `ta.cum` uses `safe_float`.
- **Matrix handles** — UDF/`kron` results stay list-of-lists objects; builtins do not `len()` a numpy scalar.
- **Corpus sanitize** — keep trailing commas on wrapped `=>` calls; tab-indented bodies after `) =>`; drop Hugo/markdown tails after a complete script; keep nested `/* */` comments; stub jinja/markdown planning docs.
- **Grammar** — `NEWLINE+` before indented `=>` / if / for / while / switch bodies.

Changed

- Docs: missing-features / implementation-status / roadmap / progress report aligned with compile object-mode residual recovery (P1p plot MISMATCH tail still open).

[0.3.15] - 2026-08-19

Added

- **`pynescript download-builtins`** — catalog + download TradingView® builtin Pine templates via pine-facade (`--list` / `--yes` / `--limit`).
- **`scripts/search_clone_pine_pool.py`** — search and shallow-clone GitHub Pine repos for corpus collection.
- **`cf/`** — Cloudflare® Containers Worker that fronts the Pro API and keeps a named instance warm.

Improved

- **LSP hover** — types/qualifiers (`series` / `float`), namespaces (`ta` in `ta.sma`), user functions/assignments/`type` / `enum`, and longer keyword briefs. Builtin cards keep signature fence + params when metadata has them.

Fixed

- **Compile set06 residuals** — missing `array` / `map` / `matrix` helpers, `for` + `continue` increment, for/while-as-expression emit, multi-value and default-only `switch`, `/* */` comments, map `for-in` pairs, `ta.dmi` scalar assign, `matrix.set` expression form, `np` unpack shadowing, nopython `TypingError` object-mode retry, tuple/UDT stores off float64 series.
- **Corpus sanitize** — RST/Hugo/MDX chrome, mustache `{{IDENT}}`, indented `study` bodies, column-1 ternary tails, torn string fixtures.

[0.3.14] - 2026-08-18

Fixed

- **`strategy.position_avg_price`** — repeating `strategy.entry("L")` no longer overwrites a filled position (avg was tracking last close). Extra same-direction entries only when `pyramiding` allows; interpret and compile.
- **`strategy()` leverage vs margin** — when `leverage=` is set it wins (margins derived from it). `margin_long` / `margin_short` only apply when leverage is omitted. Compile broker accepts those kwargs the same way.

Changed

- Docs / comments: PYNE open-entry cap is `pyramiding + 1` (TV `pyramiding` default 1 ≡ PYNE 0). Unused compile-broker `replace_same_id` kwarg removed.
- Tests: leverage-vs-margin (both set), compile margin-only 5× cash sizing, same-id pyramid VWAP (third fill blocked), pending same-id limit/stop replacement; free-tier rate/slot no-ops when the switch is off.
- **Free-tier `/run` guards are opt-in** — `FREE_TIER_LIMITS` (default off). Bar/script/rate/concurrency and chart/mock-only `data_source` apply only when set to `1` / `true` / `yes` / `on`. Production compose still enables them (`FREE_TIER_LIMITS=1`). Health `features.free_tier_limits` reports the flag. Pro API / endpoint abstracts no longer read as always-on.
- Product landing and end-user hub list `pyne optimize` / `POST /optimize`.

[0.3.13] - 2026-08-17

Fixed

- Optimize scores broker fills (not `strategy.exit` placements); trail-only exits; warmup pairing; JSON-safe study scores.
- Walk-forward train+test run cap; HTTP `min_trades=0` / 400 mapping; CLI walk-forward windows; continuous Random/TPE sampling; grid product cap.
- Incremental `ta.dpo` / `ta.kst` dispatch; `timeframe.change("1M")` is monthly; compile table cells/colors and deleted linefills.

[0.3.12] - 2026-08-17

Added

- **Strategy hyperparameter search** — `pynescript.optimize` (TPE / random / grid, holdout + walk-forward) scores real `Runtime` strategy events. `POST /optimize` (same free-tier gates as `/run`) and `pyne optimize`. Not `/backtest/quick`; not a Pine builtin.
- Real **`timeframe.change`** on interpret and compile (UTC fixed-width buckets; bar 0 is a new period).
- HTF `request.security` allowlist includes **`ta.wma`** and **`ta.rma`**.
- Incremental interpret kernels for **`ta.aroon`**, **`ta.dpo`**, **`ta.donchian`**, and **`ta.kst`**.
- Dual-host plot/drawing identity: compile geometry-only AXIS export, plot_meta kinds, coordinate snapshot at create/set.

[0.3.11] - 2026-08-16

Added

- Incremental interpret **`ta.nvi`** / **`ta.pvi`** plus matching compile `numba_nvi_inc` / `numba_pvi_inc`.
- Flask `POST /run` `timeout_seconds` and `POST /run/batch` `libraries`.
- Dual-host goldens: Supertrend mid±factor·ATR, trail OHLC, foreign `request.*` → `na`, compile plot keys.

Changed

- Interpret **`ta.pvt`** / **`ta.vpt`** is cumulative (matches compile / TV). Bar 0 is `0.0`.
- `pynets/` submodule pin **v0.2.0** (interpret + JS compile). Python Runtime remains the oracle.

Fixed

- Compile MACD signal SMA-seeds the MACD line; OBV skips the first close change.
- Compile `ta.ao` / `ta.aroon` kernels (were all-`na`).
- Compile foreign `request.security` tuple unpack no longer invents chart close.
- Compile titled `bgcolor` / empty `plot(..., title="")` keys match interpret (disk IR v10).
- Compile `strategy.exit` keeps the Pine exit id for trail events.

[0.3.10] - 2026-08-16

Added

- Incremental interpret kernels for **`ta.obv`**, **`ta.wad`** / **`wvad`**, **`ta.cmf`**, and **`ta.klinger`** (`PYNE_TA_INCREMENTAL`).

Changed

- Runtime interpret hot path (Round 9): direct Assign/`plot`/Call dispatch, skip unused derived OHLCV series, plot bar-reuse. Same-machine bench @ 2000 bars: minimal **2.7×**, `ta_sma` **2.0×**, `ta_combo` **1.45×**. `PYNE_SERIES_RING` still default off.
- `ta.vwap()` with no source still updates `hlc3` when that identifier is absent from the script.

[0.3.9] - 2026-08-16

Added

- **`FunctionDef.returns`** — UDF/method return types survive parse → unparse (`int ilog2(...)`).
- Incremental **`ta.median`** / **`ta.cmo`** Numba kernels; statement **`hline`** / **`fill`** stay nopython with synthesized drawings.
- Thread-local ANTLR lexer/parser reuse (warm SLL DFA across distinct sources).
- **LSP Docker image** `ghcr.io/hoox-sh/pyne/lsp` (stdio `pyne-lsp`, slim `[lsp]` only). Published on `v*` tags with api/cli.

Changed

- Grammar left-factors typed names; bare `x =` is `Assign`, `=` reassignment only on attribute/subscript.

- Evaluator: `var` history carry at bar start; series maxlen matches host ≥1000; `_pine_site_id` for UDF/call-expr history.
- **pyne-vscode** publisher namespace is **`hoox-sh`** (extension id `hoox-sh.pyne`).

Fixed

- **`Runtime.run(mode="auto", libraries=)`** forwards `libraries` into interpret fallback.
- Windows CLI `--help` no longer crashes on cp1252. Open VSX unknown-publisher no longer fails Build & Release.
- **CORS** — AXIS product Origins + `/health` free CORS path (also in 0.3.8 notes).
- **Build & Release:** Create Release no longer downloads Docker Buildx cache artifacts.

[0.3.8] - 2026-08-15

Added

- Compile UDT field defaults on `Type.new()` (omitted `bool x = false` stays `False`; `bool(np.nan)` no longer trips flags). Fixes interpret↔compile `071` Entry plot.
- `register_library_source` finalizes library exports so `Runtime.run(..., libraries=)` / `POST /run` `import ns/Lib/1` actually binds members.
- Supertrend first-party goldens lock the simplified mid±factor·ATR contract (interpret full/inc + Numba).
- LSP: user-enum completion/hover/outline; soft-keyword hover; `plot(ta.` leaf insert; drop C001/C003 false positives.

Fixed

- `trail_points=0` / `na` no longer disables a valid `trail_offset` (interpret + compile).
- VPS deploy AXIS health probe uses `:80` (not stale `:8081`).

Changed

- Compile disk IR cache meta version **7 → 8** (UDT defaults).
- Docs honesty: tick-offset `profit` / `loss`, omitted bid/ask `na`, H1 Runtime unify done.

[0.3.7] - 2026-08-15

Added

- **`Runtime.run(..., libraries=)`** — optional `[{namespace, name, version, source}]` registered via `register_library_source` before interpret so AXIS `import ns/Name/ver` resolves published git folders. Pro API `POST /run` accepts the same `libraries` field.

Changed

- Runtime hot path: skip series dual-write when `PYNE_SERIES_RING=1`; tick-offset exits, incremental MA, plot pack, kernels.

Removed

- Colocated **`pine-worker/`** TypeScript extra tool (parser/evaluator port). Lives in the sibling `pine-worker` checkout, not this repo.

[0.3.6] - 2026-08-13

Added

- **Binary search in UDT arrays** (Pine August 2026): `array.binary_search()`, `array.binary_search_leftmost()`, and `array.binary_search_rightmost()` accept `sort_field` (const int field index, default 0, or const string field name) so they can search arrays of user-defined types. The array must be sorted by the same field in ascending order. Interpret + compile object-mode, with dual-host parity tests.
- **Runtime `timeout_seconds`**: optional wall-clock circuit breaker on interpret (every 32 bars); partial results with `timed_out` + `error_kind=runtime` for edge/cron budgets (shared by Pro API and pyne-worker).
- **`linefill.new` export** — `DrawingRegistry.export_for_api` serializes line fills as `type: "linefill"` quads (line1 + line2 endpoints) so AXIS can paint the band.
- *Compile drawing set_ fold** — `fold_compile_drawing_mutations` applies `line.set_*` / `label.set_*` / … onto live handles and drops set events so compile `/run` drawings match interpret final state.
- **Compile drawing `*.delete`** — `line.delete` / `label.delete` / `box.delete` / `polyline.delete` / `table.delete` emit `kind: 'delete'` events on `__drawings` (target = `*.new` handle); `fold_compile_drawing_mutations` drops deleted objects (was MVP no-op).
- **`force_overlay` on export** — line/box/label payloads include `force_overlay` for AXIS pane routing.

Changed

- **H1 dual-host:** pyne-worker `pynescript_backend` is a thin wrap over package `pynescript.runtime` (strict OHLCV validation only); vendor via `./scripts/sync_vendor.sh`.

[0.3.5] - 2026-08-12

Fixed

- **Drawing export (`line.new`)** — `export_for_api` no longer drops lines whose `xloc.bar_index` is past the last bar (classic `bar_index + 1` on `barstate.islast`). Extrapolates from series period; empty `bar_times` passes bare bar index for AXIS logical mapping.

[0.3.4] - 2026-08-10

Added

- **Package Runtime SoT** (`pynescript.runtime`): bar-loop host, series, CustomEvaluator live in the installable package; `backend.*` re-exports for Pro API.
- **First-party dual-host TA goldens** (ATR / Supertrend / Keltner) and full `test_ta_incremental` CI gate.
- **Strategy exit surface**: pending stop/limit + OCA; `from_entry` multi-leg (interpret + compile); `qty_percent`; minimal trail (interpret + compile); compile risk halt cascade (`allow_entry_in`, `max_position_size`,

`max_drawdown`, `max_cons_loss_days`, `max_intraday_loss`, `max_intraday_filled_orders`); open/closed trade fields + MAE/MFE.

- **`request.security` honesty + HTF resample**: policy meta on Runtime; same-symbol coarser TF OHLCV bucketing; allowlisted simple `ta.sma` / `ema` / `rsi` / `atr` on HTF.
- **Realtime host simulation**: `realtime_last_bar` / `realtime_ticks` / `realtime_bars` / `realtime_from_bar` for `varip` multi-tick tests.
- Free-tier Pro API guards (bar/script caps, rate, concurrency, SSRF-safe webhooks, chart/mock-only free data).
- Hash-only JSON API key store; first-party fixtures under `tests/fixtures/first_party/`.
- pine-worker series lookback polarity + parity smoke (`series[1]` previous bar).
- Plot host: `plot.style_*` constants, style/linestyle capture, plotshape size export.

Fixed

- EMA dual-host seed (full-list SMA seed matches incremental/Numba).
- `ta.atr` Wilder RMA on interpret + Numba.
- AugAssign / tuple unpack series binding; unparser `visit_Simple`; linter C004/W002.
- Enum/string series history so `d[1]` works for plotshape flips.
- vscode-extension transitive `brace-expansion` Dependabot high (2.1.4).

Changed

- CI Core runtime gates parity, strategy, series, TA incremental, first-party goldens, runtime package, free-limits.
- Docs: `docs/known_divergences.md`, audit sprint status notes.

[0.3.3] - 2026-08-09

Fixed

- **Corpus residual (C1, set01–04):** dual-namespace user methods vs bare ta aliases (`method dmi` / `float dmi = dmi(...)` no longer mis-routes to `ta.dmi` after series assign); `line.get_price` / line getters soft-na on `na` line or endpoints and coerce PineSeries samples; `request.security` OHLCVT + `time` list unpack on different TF; sanitize dangling binary ops inside unclosed trailing calls (truncated docs scrapes).
- `ta.dmi` soft-na on unresolved / `na` length (aligned with other TA period soft-na).

Changed

- Local open-source corpus set01–04 (2477 scripts, not shipped in git): **parse 99.96%** (2476/2477) and **Runtime interpret 100% excl. EXPECTED_FAIL** (2466 OK + 11 intentional demos); set01 Runtime **249/249**. Harness classifies path-listed intentional demos (library `runtime.error`, lower-TF guards, pathological loops) as EXPECTED_FAIL.
- Docs / landing: README, compatibility, roadmap, missing-features, and marketing numbers updated to the 2026-08-09 corpus snapshot (not TradingView® platform parity).

[0.3.2] - 2026-08-09

Changed

- **Console scripts**: preferred names are **`pyne`** and **`pyne-lsp`**. Legacy **`pynescript`** / **`pynescript-lsp`** remain installed as aliases (same callables). Import package is still `pynescript`; PyPI dist is `hoox-pyne`. Docs, VS Code auto-detect, and editor client configs prefer `pyne*` with fallback to `pynescript*`.

[0.3.1] - 2026-08-09

Added

- **Strategy average-price models** (pynescript extension): `strategy(..., avg_price_model=)`
  - `"stock"` (default) — multi-leg FIFO reweight of `position_avg_price` on partial close (TV-like)
  - `"futures"` — sticky net AEP until flat (linear USDT-M / BTCUSDT-style)
  - `"inverse"` — accepted; sticky reduce (harmonic add deferred)
  - Tokens: `strategy.avg_price_stock` / `avg_price_futures` / `avg_price_inverse`
- **Leverage for futures UI** (pynescript extension): `strategy(..., leverage=N)`
  - Scales percent/cash default qty: `qty = margin × leverage / price`
  - Margin held = notional / leverage; free cash adjusts accordingly
  - Simple `strategy.margin_liquidation_price` when leverage > 1
  - Derives TV-style `margin_long` / `margin_short` % as `100/leverage`
  - Read-back: `strategy.leverage`
- Dual-path wiring: interpret + compile broker + compiler emit for both options
- Docs: stock vs futures avg semantics; TV vs PYNE notes on `input` before `strategy()`

Fixed

- Compile broker ctor no longer emits `name_arr[__bar_idx]` for strategy kwargs (undefined at ctor time). Const-like input/literal defvals fold into ctor (e.g. `lev = input.float(10)` then `strategy(..., leverage=lev)` → `leverage=10`).

Changed

- Compile emit allowlist also wires `default_qty_type` / `default_qty_value` into `CompileStrategyBroker(...)`.

[0.3.0] - 2026-08-06

First public **PYNE** release. PyPI distribution name is **`hoox-pyne`** (plain `pyne` / `PyNE` is taken by an unrelated package; import/CLIs remain `pynescript` / `pynescript-lsp`). Install: `pip install "hoox-pyne[lsp]"`.

Added

- **Alert engine** (`AlertsMixin`): TV-style `alert.freq_*` normalization, once-per-bar dedup, once-per-bar-close gating, `alertcondition` fire-on-true, host helpers `export_alerts` / `export_alerts_from_evaluator`.
- **Dual-host alerts export**: Pro API + pyne-worker `Runtime` clear/export `alerts` (and optional `alert_conditions`) on interpret `/run`.

- **L2 alert webhooks**: Pro API `backend/alert_forwarder.py` (`webhook_url`, `ALERT_WEBHOOK_URL`, last-bar filter, batch JSON); pyne-worker edge webhooks + cron last-bar delivery. Docs: `docs/pyne/runtime/alerts.mdx`.
- Roadmap residual backlog IDs **H1–L3** (dual-host, corpus tail, warm compile, series/TA, optional fidelity, long-horizon); see `docs/ROADMAP.md` (2026-08-01). **L2 closed** (Pro API + edge).
- Corpus C1 residual goldens: bare `tonumber`, `math.isfinite`, strategy `closedtrades` / `opentrades` entry_id/comment/max_drawdown/max_runup accessors; tests in `tests/test_corpus_runtime_residuals.py`.
- Interpret↔compile plot parity harness: `scripts/compare_interp_compile.py` (multiprocess series compare with `--file-list`, `--timeout-sec`, `--ignore-hline-keys`, `--ignore-fill-keys`, and wave workers).
- Parity tests: `tests/test_interp_compile_parity.py`, `tests/test_dividend_yield_parity.py`, and R8/R9 kernel suites.
- Compiler: `time_arr` on the compiled execute signature; `numba_timestamp` / `numba_utc_parts`; titled `fill()` series export; `clear_numba_function_caches` with corrupt-cache recovery.
- Compile path for `numba_rci` / `ta.rci`.
- Strategy events system: StrategyEvent dataclass, full emission from strategy.* builtins, parity test corpus.
- pine-worker/ as colocated extra tool: TypeScript evaluator port + Python to TS converter script.
- var / varip declaration modes and ReAssign support.
- Multi-target Docker image (`api` / `api-dev` / `lsp`), `docker-bake.hcl`, prod compose overlay, Makefile docker helpers.
- Key-store backends selectable via `STORE_BACKEND` (`json` | `sqlite` | `redis`) for multi-worker / multi-replica deploys.
- PyPI publish workflow (Trusted Publishing on `v*` tags) and package build job in CI.
- Compiler disk IR/module cache (`PYNE_COMPILE_DISK_CACHE`, default on) + typed `CompileError*` hierarchy + `prewarm_numba_builtins()`.
- Runtime structured errors: `error_kind` (`parse|compile|runtime|data|order|mode`), `error_type`, `error_bar` (API `/run` forwards).

Changed

- GitHub / package metadata for org **`hoox-sh/pyne`**: project URLs, Docker image source label, docs, CONTRIBUTING, and PyPI Trusted Publisher owner. See `docs/pyne/devops/publish-checklist.mdx`.
- **PyPI distribution name is `hoox-pyne`** (import/CLIs remain `pynescript` / `pynescript-lsp`). Avoids collision with upstream elbakramer `pynescript` and the unrelated pre-existing `pyne` / `PyNE` project on PyPI.
- Default image version labels / bake / Cloud Build substitution aligned to **0.3.0**.
- Compile `request.security` policy: same-symbol simple OHLCV only; other foreign tickers and complex expressions resolve to `na`.
- Disk IR meta version bumped when titled fill series export landed (and again for R8/R9 parity kernels).
- Interpret bar-loop residual wins (visit/Call arg plans, series last-sample, residual TA): ~1.24× `ta_combo` vs Round 5 (~137 ms @ 2k bars).
- Compiler stays numeric for more history/math/chart surface; viewport right bar time is `(n_bars-1)*60000` in compile mode.
- Runtime `mode=auto`: non-empty `inputs` forces interpret with clear fallback reason; object-mode compile no longer requires Numba.
- VS Code extension rebrand: **PYNE — Pine Script™ for VS Code** (`pyne` 0.3.0), HOOX logo icon, marketplace description as part of the HOOX open trading stack, TradingView® / Pine Script™ trademark disclaimer, and

richer TextMate syntax highlighting.

- VS Code extension packaging: esbuild-bundle `vscode-languageclient` into the VSIX, explicit `onCommand` activation, resilient command handlers, and shared output channel.
- CI rewritten for the post-AXIS-extract repo: Python lint/test matrix, package build, Docker smoke; removed dead `frontend/` jobs.
- `require_admin_token` enforces `ADMIN_TOKEN` + `X-Admin-Token` (fail-closed); prod compose drops host source mounts via `volumes: !override`.
- `pyproject.toml` project URLs, Alpha classifier, Python 3.13, `pro` extra includes `redis`.
- Updated documentation (ROADMAP, missing_features, implementation status, LSP plan, devops Docker) to reflect current state.

Fixed

- **R8/R9 interpret↔compile corpus parity**: foreign-na / MTF `request.security`, Heikin-Ashi security, TA call-site slots, UDF series locals per Call site, series assign bar key `(site, name)`, PineSeries snapshot for typed float/ `nz` / `array.set` (BBI ring buffer), `ta.vwap(hlc3[, anchor])`, `time(...)[1]` call-expr history, strategy `position_size` / `avg` / `closedtrades[n]` end-of-bar history, Pine `==` / `!=` with `na` (`na==na` True; other na compares False via `numba_pine_eq` / `ne`).
- Builtin kwargs merge no longer drops explicit trailing `None` / Pine `na` (e.g. `array.push(id=a, value=na)`), unblocking many corpus library scripts.
- Corpus C1 8-agent residual pass: `str.replace` 4-arg/occurrence + coerce; richer `timestamp` date strings + TZ-first; series negative/na/OOB index → `na` ; TA float/series periods; color hex/string channels; `syminfo.prefix` / `ticker` dual-mode; array get/set soft index + `index_2d_to_1d` stub polyfill; `hour` / `dayofmonth` /... series+timezone arity.
- set05 6-agent pass: `timestamp(9999,…)` year-first + calendar overflow; `strategy.initial_capital =` reassignment; `timenow` / `dayofweek.*` ; sanitize keeps `type == "SMA"` ternary chains; v4 `random` / `offset` / `round_to_mintick` + `ticker.pointfigure` full arity; call-site cache stored on AST node (fixes id reuse collisions).
- set05 residual round 2 (6 agents): bare TA series `obv` / `accdist` / `vwap` ; `ta.linreg` length<2→na; `ta.kama` 2-arg; UDF name clobber restore; array.push soft arity + `newcolor` aliases; multi-island sanitize merge for UDF defs; missing UDF soft-na; int()/tonumber soft coerce; ticker.modify `adjustment=` ; kwargs merge + timestamp const-fold + static for-to.
- Round 6 multi-agent pass (perf + correctness + error handling + compiler coverage).
- Compiler nopython kernels: `ta.dmi` / `ta.adx` / `ta.supertrend` / `ta.alma` / `ta.percentrank` (match interpret oracle).
- Incremental interpret TA for residual full-history paths: `mfi` , `sar` , `kc` / `kcw` , `alma` , `correlation` , percentiles (behind `PYNE_TA_INCREMENTAL`).
- Strategy exit commission and exit slippage on interpret + compile paths; bad order args emit events instead of silent fills.
- Stable `CRYPTO_KEY` / `METADATA_KEY` resolution for Fernet metadata encryption (GitHub secrets wired).
- User series no longer shadows bare builtins (`ad` / `tr` / `obv` / `pvt`): `plot(ad)` emits the user series instead of re-emitting the Chaikin A/D stub.
- `highestbars` / `lowestbars` use negative TradingView®-style offsets (Aroon parity).
- Compile Wilder RSI matches interpret (was a rolling SMA of gains).
- NaN-safe EMA/RMA seed for ATR custom `ma_function` and nested DEMA.

- UDF last assign of `na` no longer retains a prior tuple unpack (ADX early zeros).
- `math.avg` propagates `na`; `ta.roc` uses the standard formula; `wma` requires a full non-`na` window; `ta.cum` treats `na` / IEEE NaN as 0.
- `request.security`: foreign tickers and complex expressions resolve to `na` instead of inventing chart close; `dividend_yield` / CVI interpret↔compile parity.
- Import stub plot cells are not serialized as color strings.
- `auto_fib` interpret and compile raise the same insufficient-pivot errors; `for` / `to` auto step; `chart.point` object dtype.
- Interpreter: `ta.rising` / `ta.falling` / `ta.highestbars` / `ta.lowestbars` no longer raise `TypeError` on Pine `na` (e.g. VIDYA warmup on MA-STER).
- Crossover equality aligned with TV/numba (`<=` / `>=` on previous bar); short series rising/falling in bar mode.
- Body `TypeError` in call dispatch no longer soft-fails to `na` (signature mismatches still soft); strategy `strategy()` apply fails closed.
- Compile object-mode: `array.sort` / `sort_indices` honor `sort_field`; `array.fill` range; `map.keys` / `values` not coerced via `safe_float`.
- Corpus sanitize: multi-version islands, UI chrome, safer ternary/call/type-field repairs without FP on `?` in input titles.
- CI Ruff E,F,W gate: F821/`Any` imports and F401 unused imports cleaned for green CI on main.

Removed

- Stale `Dockerfile.api` (folded into multi-target `Dockerfile`).
- Dead `technical_refactored.py` and internal refactoring notes from the published package tree.
- Broken AXIS-only GitHub workflows (`axis-nightly`, PWA/e2e jobs) — AXIS CI lives in jango-blockchained/axis.

PYNETS

TypeScript / Bun library port of PYNE — parse, unparse, Runtime.run. Python remains the semantic source of truth.

PYNETS

PyneTS (`@hoox-sh/pynets` , currently **0.2.0**) is the TypeScript / Bun library for Pine Script: `parse` , `unparse` , and `Runtime.run` . Public names match Python `pynescript` . Python `pynescript.runtime` is the **source of truth** — when semantics disagree, Python wins.

This is **not** a Cloudflare Worker, **not** a Numba compile path, and **not** a TradingView platform substitute.

This PYNE checkout pins the `pynets/` submodule at annotated **v0.2.0** (`@hoox-sh/pynets` **0.2.0** — interpret + JS compile, matching npm). Python `pynescript.runtime` remains the semantic oracle.

```
bun add / npm install / esm.sh. CLI still needs Bun.

na is null. Lookback, TA call-sites, foreign security → na.

check / format / run / dump / info. Pipes stay JSON.

JS bar-loop emit. mode interpret | compile | auto.

Compare plots against Python Runtime.run on the same bars.

PYNE · PyneTS · pyne-worker · pyne-agent-worker · AXIS · HOOX.
```

Abstract

Most TypeScript Pine ports rewrite the language. PyneTS does not. It is the library-shaped port of **PYNE**: the same ANTLR grammar, the same ASDL field names (`kind` , `lineno` , `col_offset` , …), the same interpret contract.

```
Source (.pine / .pyne)
  → shared PYNE .g4  (TS ANTLR target — not forked here)
  → ASDL AST
  → bar-loop  (interpret | JS compile | auto)
  → plots / series / events / drawings / alerts
```

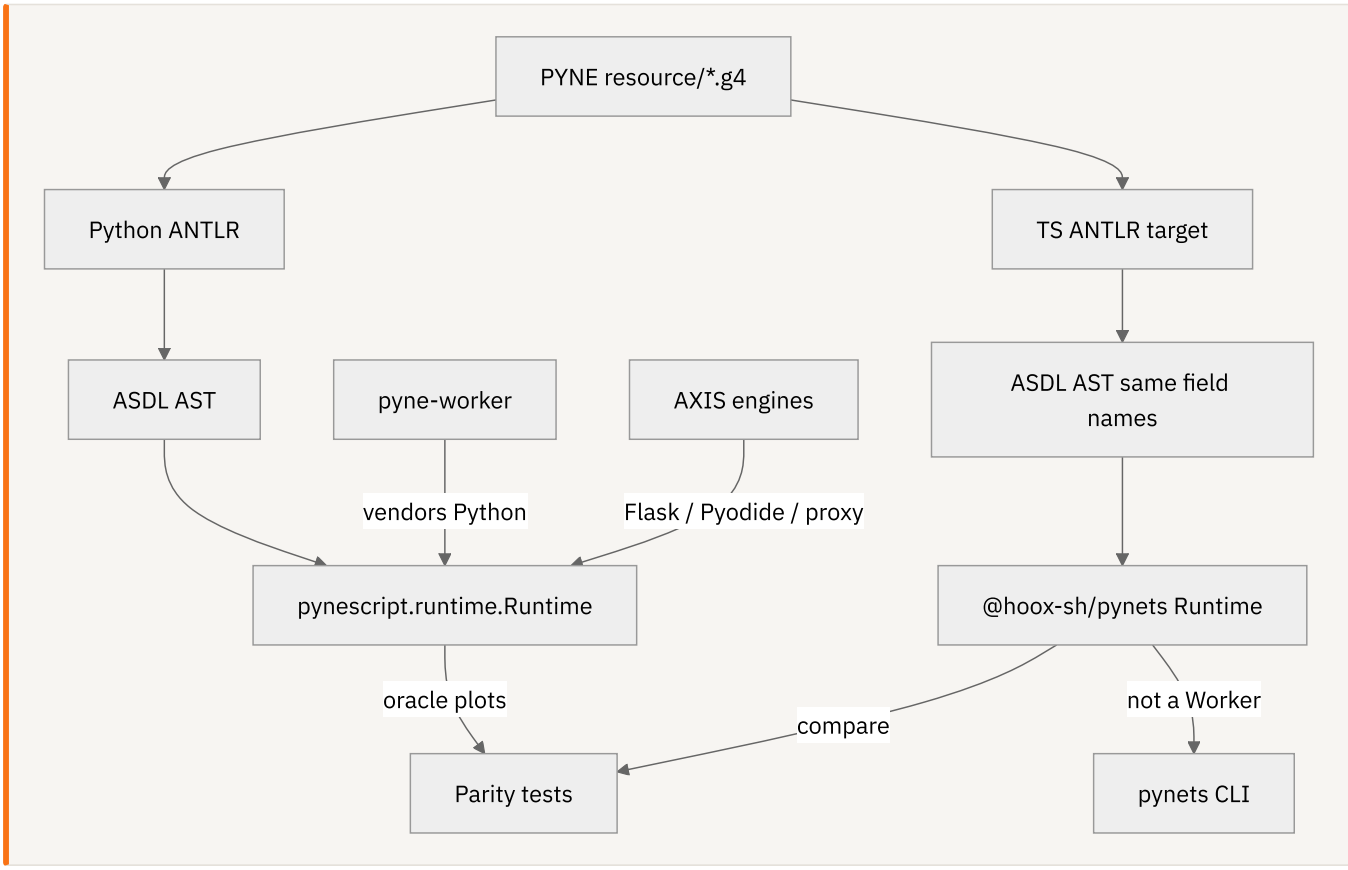
Standalone checkout: github.com/hoox-sh/pynets. PYNE consumes it **only** as the `pynets/` git submodule — never copy sources into `hoox-sh/pyne` .

```
const src = `//@version=5
indicator("sma")
plot(ta.sma(close, 3))
`;

const tree = parse(src);
unparse(tree);

const out = new Runtime("AAPL").run(src, [
  { close: 1 }, { close: 2 }, { close: 3 }, { close: 4 },
]);
// out.plots, out.series, out.count
// new Runtime("AAPL", { mode: "compile" }) — JS bar-loop emit (not Numba)
// mode: "auto" tries compile, then interpret
```

Conceptual model



Invariant: PyneTS is a **library you import**. pyne-worker is the production **Python** Cloudflare isolate for `POST /run`. AXIS engines still call Python (Flask, Pyodide wheel, Worker proxy) — they do not import `@hoox-sh/pynets` today.

Interface surface

Name	Role
`parse(source)`	ANTLR → ASDL tree. Throws `PinescriptSyntaxError`
`unparse(tree)`	Tree → normalized source
`dump(tree)`	Debug dump
`tokenize(source)`	Lexer tokens
`new Runtime(symbol, options).run(source, bars, extra?)`	Bar-loop evaluate
`Runtime.stream(source)`	Push-driven re-eval
`Runtime.runProvider(source, provider)`	Fetch bars then `run`
`compileScript` / `transpile`	JS emit (not Numba)
CLI `pynets`	`check` `format` `run` `dump` `info`

Default `mode` is **`interpret`**. Constructor and per-call `extra` accept `mode: "interpret" | "compile" | "auto"`.

What this is not

Confused with	Reality
pyne-worker	Python Cloudflare Worker. Vendors `pynescript.runtime`. `POST /run`. See pyne-worker.
pyne-agent-worker	NL authoring host. Not an evaluate library. See agent.
Python `pyne run`	Compile-only smoke on synthetic bars. PyneTS `pynets run` calls `Runtime.run`.
Numba	Python compile backend. PyneTS compile is **JS emit** (object-mode analog).
AXIS engine	AXIS does not load this package.

Internals (repo paths)

Standalone (`/home/jango/Git/pynets` or the published package):

Path	Role
`src/index.ts`	Public barrel — do not re-export generated ANTLR
`src/ast/helper.ts`	`parse` / `unparse` / `dump` / `tokenize`
`src/ast/nodes.ts`	Hand-written ASDL nodes
`src/generated/`	ANTLR TS — **do not hand-edit**; `bun run generate`
`src/runtime/interpret.ts`	`Runtime`, interpret host
`src/runtime/compile/`	JS emit (`transpile` / `compileScript`)
`src/cli.ts`	TTY CLI
`test/interpret_*.test.ts`	Focused interpret tests

PYNE checkout: `pynets/` submodule. Grammar SoT stays in PYNE `resource/*.g4`.

Invariants & edge cases

1. **Python wins.** Do not invent TradingView platform behaviour that Python does not implement.
2. **`na` is `null`.** Non-finite in/out is `na`. `na == na` is true. Any other comparison involving `na` is false.
3. **Lookback:** `x[0]` current, `x[1]` previous, out of range or negative → `na`.
4. **TA is one sample per bar**, state keyed by call-site (Python incremental kernels).
5. **`request.security` foreign / HTF without a feed → `na`.** The chart series is never silently reused as another symbol.
6. **Do not fork the grammar.** Edit `.g4` only in PYNE, then regen here.
7. **Do not edit** `src/runtime/compile/*` and `src/runtime/interpret.ts` in the same agent turn.
8. License: **AGPL-3.0-or-later**.

Worked examples

Library (Bun)

```
const src = `indicator("t")
plot(close[1])
plot(na == na ? 1 : 0)`;

const tree = parse(src);
console.log(unparse(tree));

const out = new Runtime("TEST").run(src, [
  { close: 10 },
  { close: 20 },
]);
// last bar: plots[0] = 10 (previous close), plots[1] = 1
```

Node 20+ / browser

```

```

```
<script type="module">
  import { Runtime } from "https://esm.sh/@hoox-sh/pynets";
</script>
```

Bun imports TypeScript source (`package.json` `"bun": "./src/index.ts"`). Node and browsers use `dist/` (`bun run build`; `prepack` runs it). The CLI still needs Bun on `PATH`.

Failure modes

Symptom	Cause	Fix
Plots disagree with Python	Port bug or invented host behaviour	Fix TS. Python is the oracle.
`request.security` returns `na`	No foreign/HTF feed	Expected. Do not invent bars.
`mode: "compile"` misses a builtin	JS emit surface lag	Use `mode: "auto"` or `interpret`
Importing in a Worker and expecting `/run`	This is a library	Use pyne-worker
Editing `src/generated/*`	Overwritten on regen	`bun run generate` after PYNE `.g4` change
`pynets` CLI on Node-only host	`bin` is `src/cli.ts` (Bun)	Run under Bun, or call `Runtime` from Node via `dist/`

See also

- Install
- Runtime contract
- CLI
- JS compile
- Parity
- Evaluate scripts (Python)
- Modes matrix
- Ecosystem
- pyne-worker
- pyne-agent-worker

INSTALL PYNETS

Add @hoox-sh/pynets with Bun, npm, or esm.sh. CLI requires Bun. Node and browsers use the ESM bundle.

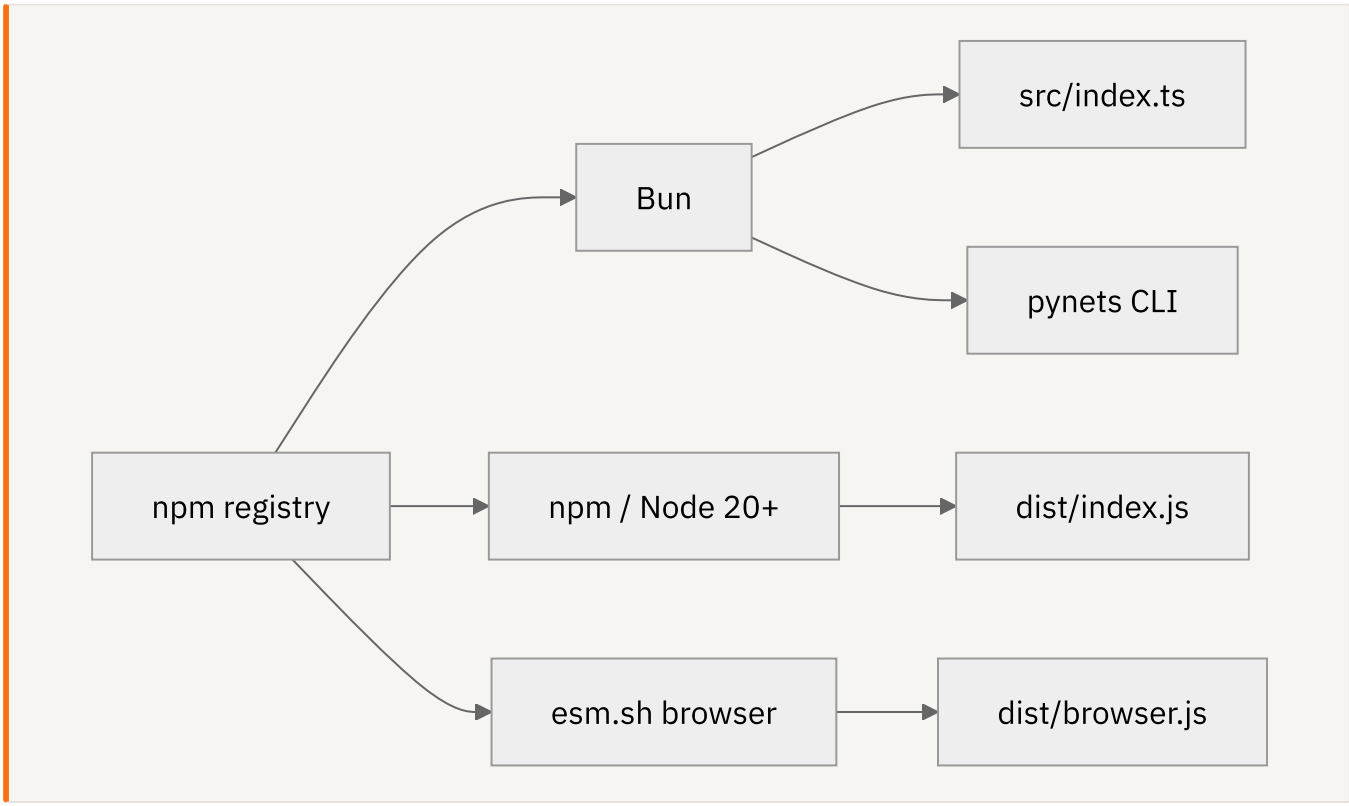
INSTALL PYNETS

Abstract

`@hoox-sh/pynets` is published to the npm registry. Bun imports TypeScript source. Node 20+ and browsers consume the ESM bundle in `dist/`. The `pynets` CLI is a Bun script.

Package: npmjs.com/package/@hoox-sh/pynets · source: github.com/hoox-sh/pynets. Current version **0.2.0**.

Conceptual model



Consumer	Entry	Notes
Bun `>= 1.2`	`./src/index.ts`	No bundle required
Node `>= 20`	`./dist/index.js`	Run `bun run build` / `prepack`
Browser	`./dist/browser.js`	Same package; `exports.browser`
CLI	`./src/cli.ts`	Needs Bun on `PATH`

Interface surface

Bun (recommended)

```
bun add @hoox-sh/pynets
bunx pynets -- help
bunx pynets info
```

```

```

Node 20+

```
npm install @hoox-sh/pynets
```

```

```

`npm pack` / publish runs `prepack` → `bun run build`. If you clone the repo and import from Node without building, resolution fails.

Browser (esm.sh)

```
<script type="module">
  import { Runtime } from "https://esm.sh/@hoox-sh/pynets";
</script>
```

This is a library import, not a hosted evaluate API.

From a PYNE checkout

```
git clone --recurse-submodules https://github.com/hoox-sh/pyne.git
cd pyne
git submodule update --init --recursive   # after a plain clone
cd pynets
bun install
bun test
```

This PYNE pin is annotated **v0.2.0** (`@hoox-sh/pynets` **0.2.0**, interpret + JS compile). That matches npm. `dist/` is gitignored — run `bun run build` in `pynets/` if you need the Node/browser ESM bundle from the submodule. Python Runtime remains the oracle.

Develop the library

```
git clone https://github.com/hoox-sh/pynets.git
cd pynets
bun install
bun test
bun run typecheck
bun run build
```

Grammar regen (needs Java + PYNE `.g4`):

```
bun run generate
```

Resolution order in `scripts/generate-antlr.ts`: `PYNETS_GRAMMAR` → `PYNESCRIPT_ROOT` → parent PYNE checkout → sibling `../pynescript` or `../pyne`.

Internals

Path	Role
`package.json` `exports`	bun → `src/`, types → `dist/index.d.ts`, browser → `dist/browser.js`
`package.json` `bin.pynets`	`./src/cli.ts`
`scripts/build.ts`	Node + browser ESM → `dist/` (gitignored)
`scripts/generate-antlr.ts`	ANTLR TS regen
`.github/workflows/publish.yml`	npm publish on `v*` tags

Local tooling is **Bun only**. Do not add npm/yarn/pnpm lockfiles. The npm registry is used only to publish.

Invariants & edge cases

1. The CLI is **not** a Node-native binary. `npx pynets` without Bun will fail.
2. This package is **not** a Cloudflare Worker. There is no `wrangler` here.
3. Do not `npm install pynets` (unscoped). The name is **`@hoox-sh/pynets`**.
4. Do not confuse with `pip install hoox-pyne` (Python).
5. Engines field: `"bun": ">=1.2.0"`, `"node": ">=20"`.

Worked examples

Smoke the CLI

```
bunx pynets info --json
# { "name": "pynets", "package": "@hoox-sh/pynets", "version": "0.2.0", ... }

echo 'indicator("x")
plot(close)' > /tmp/x.pine
bunx pynets check /tmp/x.pine
# TTY: panel; pipe: "ok"
```

Library one-liner

```
const out = new Runtime("AAPL").run(
  `indicator("sma")\nplot(ta.sma(close, 3))`,
  [{ close: 1 }, { close: 2 }, { close: 3 }, { close: 4 }],
);
if (out.error) throw new Error(out.error);
console.log(out.plots);
```

Failure modes

Symptom	Cause	Fix
`Cannot find module '@hoox-sh/pynets'`	Typo / old `@hoox/pynets` name	Use `@hoox-sh/pynets`
Node import fails in a fresh clone	`dist/` not built	`bun run build`
`pynets: command not found`	No Bun / not on PATH	`bunx pynets` or `bun run src/cli.ts`
Grammar regen fails	No Java / no PYNE `.g4`	Set `PYNESCRIPT_ROOT` or `PYNETS_GRAMMAR`
AXIS still evaluates via Python	Expected	AXIS does not consume this package

See also

- PyneTS hub
- CLI
- Python install
- Ecosystem

PYNETS CLI

pynets check / format / run / dump / info — Rich TTY, JSON on pipes. run calls Runtime.run, unlike Python pyne run.

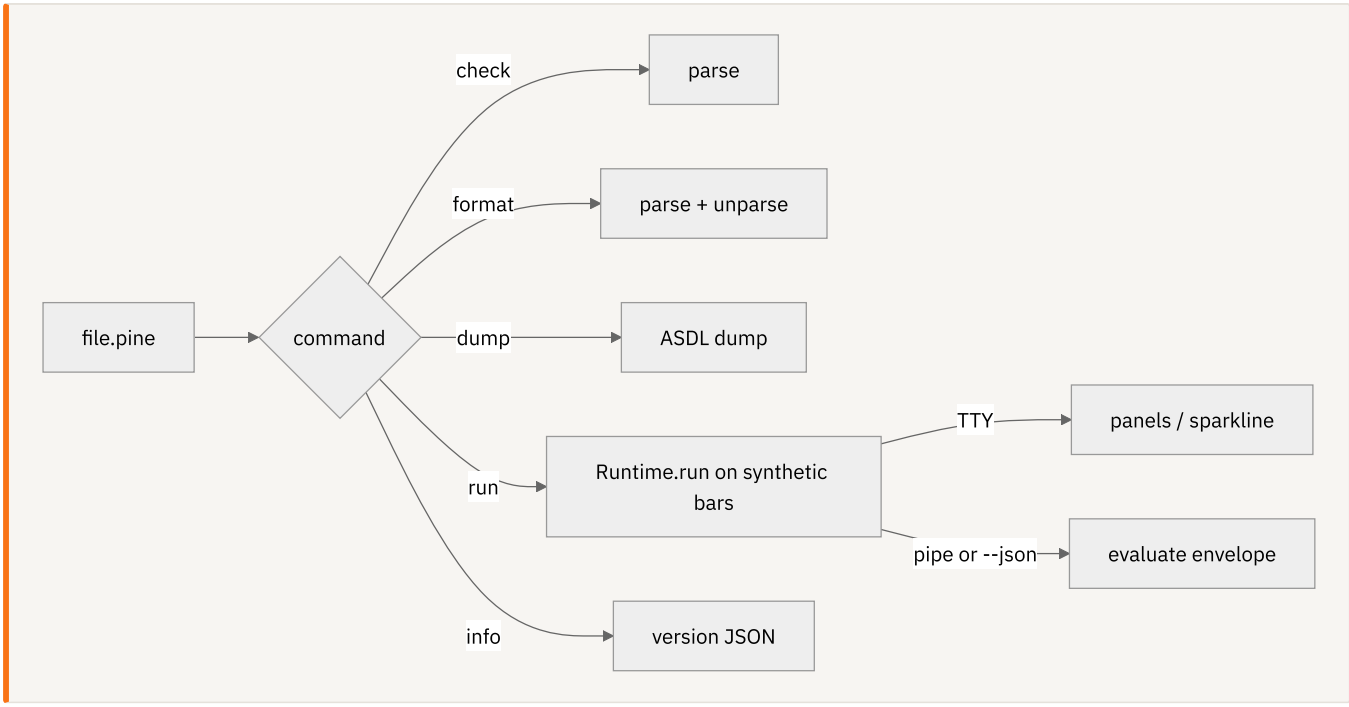
PYNETS CLI

Abstract

`pynets` is a Bun CLI for parse, format, AST dump, and evaluate. TTY output is a Rich-inspired panel (PYNE volt). **Pipes and `--json` stay machine-readable.** Exit codes: `0` ok, `1` syntax/runtime, `2` usage.

Unlike Python `pyne run` (compile-only smoke), **`pynets run` calls `Runtime.run`**. Default `--mode interpret`. `--mode compile` / `auto` select JS emit (`@hoox-sh/pynets` **0.2.0**, including this repo's `pynets/` pin).

Conceptual model



Interface surface

```
usage: pynets <check|format|run|dump|info> [file] [--bars N]
              [--mode interpret|compile|auto]
              [--commission N] [--slippage N] [--pyramiding N]
              [--json] [--plain] [--rich] [--full] [--indent N]
```

Command	Action	File required
`check`	Parse-only validation	yes
`format`	Parse → unparse (pretty on TTY)	yes
`run`	`Runtime.run` on synthetic OHLCV	yes
`dump`	ASDL-shaped AST	yes
`info`	Version and runtime extras	no
`help` / `-h`	Usage	no

Flag	Meaning
`--bars N`	Synthetic bar count for `run` (default **20**, max **100000**)
`--mode MODE`	`interpret` (default) | `compile` | `auto`
`--commission N`	Broker commission fraction (e.g. `0.001`)
`--slippage N`	Broker slippage in price units
`--pyramiding N`	Max same-direction adds
`--json`	Machine-readable `run` / `info`
`--indent N`	AST dump indent (default 2)
`--full`	Print the full dump (no 80-line cap)
`--plain`	Disable Rich (wins over `FORCE_COLOR`)
`--rich`	Force Rich (put **after** the command)
`-h, --help`	Help

Path resolution

`resolveUserFile` takes **only** the path you typed (cwd-relative or absolute). No search path. Missing file → exit 2 (`error: file not found`).

Synthetic bars

`run` builds a ramp: `close = 100 + i`, `high/low = close ± 0.5`, `volume = 1`, `time = 1700000000000 + i * 60000`. Symbol is hardcoded `AAPL`. This is a desk smoke, not a data provider.

Internals

Path	Role
`src/cli.ts`	Arg parse, commands, exit codes
`src/cli/rich.ts`	TTY paint, help, tables, sparkline
`src/index.ts`	`parse` / `unparse` / `dump` / `Runtime`

`run` constructs `new Runtime("AAPL", { broker, mode }).run(source, syntheticBars(bars))`.

Invariants & edge cases

1. **Pipe-safe.** Non-TTY `check` prints `ok`. Non-TTY `run` prints JSON.
2. `--rich` must come **after** the command (`pynets run x.pine --rich`), because flags before the command are stripped for Rich detection.
3. Unknown flags and extra positional args are usage errors (exit 2).
4. `run` default mode is **`interpret`**, matching `new Runtime()` — not Pro API `auto`.
5. `--bars` is clamped; negative / non-integer → usage error.

Worked examples

```
pynets check script.pine
pynets format script.pine
pynets dump script.pine --full
pynets run script.pine --bars 50
pynets run script.pine --mode compile --json
pynets run strat.pine --commission 0.001 --pyramiding 1
pynets info --json
```

CI:

```
pynets check script.pine
# stdout: ok
# exit 0 / 1 / 2
```

Failure modes

Symptom	Cause	Fix
Exit 2 `missing file`	No path / ENOENT	Pass a real file
Exit 1 on `check`	Syntax error	Read the ANTLR message
`run` JSON has `error` / `error_kind: parse`	Script does not parse	`pynets check` first
Expected compile-only like `pyne run`	Different CLI	See modes
Rich garbage in logs	TTY detected	`--plain` or pipe

See also

- Install
- Runtime
- Python CLI
- Modes

PYNETS RUNTIME

Runtime.run envelope: na is null, series lookback, call-site TA, request.security without data → na, stream and providers.

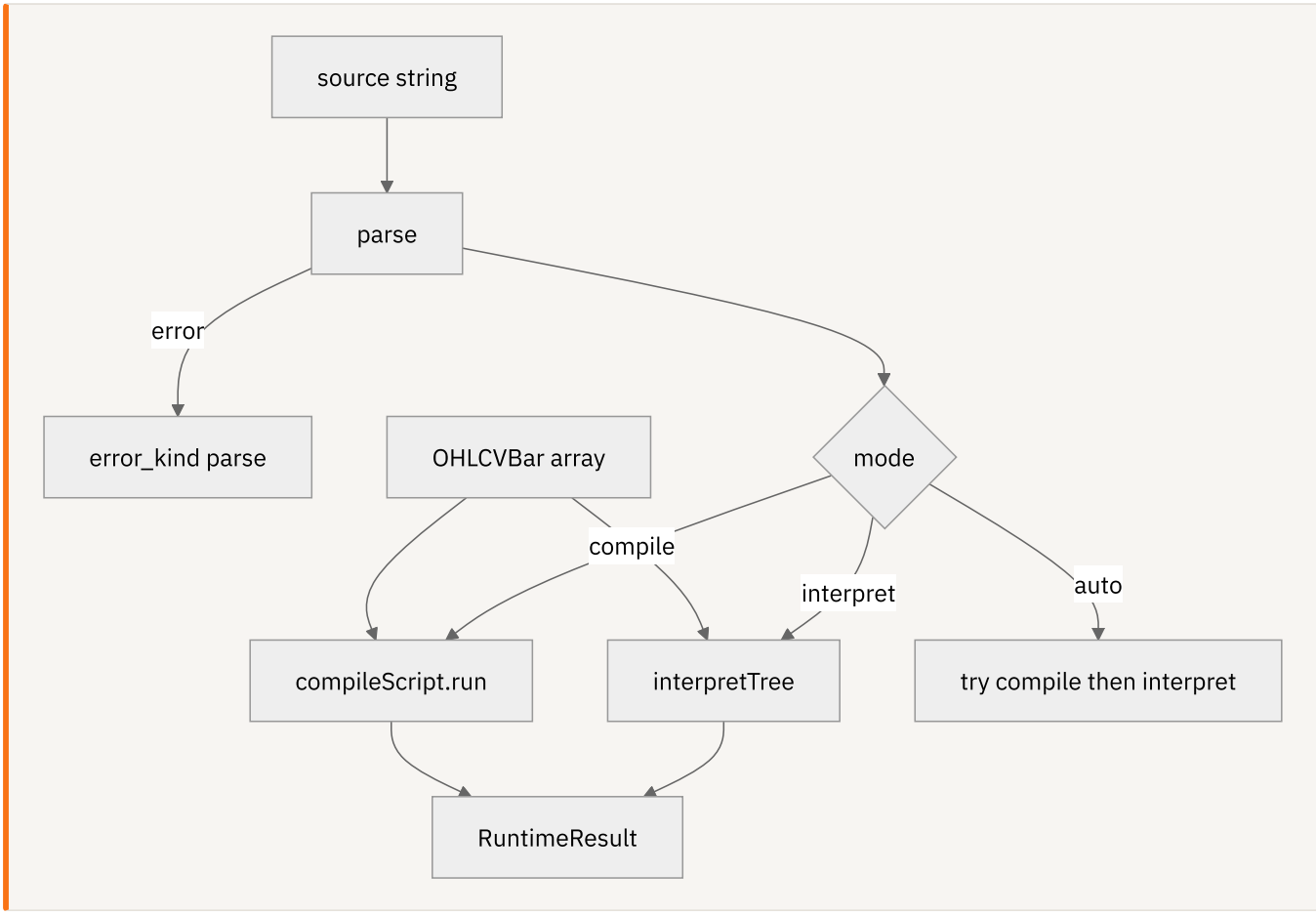
PYNETS RUNTIME

Abstract

`Runtime` is the TypeScript counterpart of `pynescript.runtime.Runtime`. It is **not** a port of `runtime/host.py` line-by-line; it matches the **public envelope**: `run(source, ohlcv) → plots / series / events / drawings`. Default `mode` is `interpret` (AST walk). `compile` / `auto`, `stream`, `runProvider`, and `registerLibrarySource` are on **`@hoox-sh/pynets` 0.2.0**, including this repo's `pynets/` pin. Python Runtime remains the oracle.

There is **no** `Runtime.evaluate`.

Conceptual model



Interface surface

```
new Runtime(symbol = "AAPL", options?: RuntimeOptions)

runtime.run(source, ohlcv, extra?: RuntimeOptions | InputOverrides): RuntimeResult
runtime.stream(source): RuntimeStream
runtime.runProvider(source, provider, extra?): Promise<RuntimeResult>
runtime.registerLibrarySource(namespace, name, version, source): void
```

`RuntimeOptions`

Field	Meaning
`inputs`	`input.*` overrides (`Record<string, number | string | boolean>`)
`broker`	`{ commission?, slippage?, pyramiding? }`
`timeframe`	Chart TF string, or `null`
`libraries`	`LibraryRegistry` instance
`mode`	`interpret` (default) | `compile` | `auto`

Per-call `extra` merges over the constructor. A plain object of input values (no `mode` / `broker` / …) is treated as `InputOverrides`.

`OHLCVBar`

All fields optional: `open`, `high`, `low`, `close`, `volume`, `time`. Missing numeric fields become `na` at use.

`RuntimeResult`

Field	Meaning
`series`	Titled plot series → `Array<number | null>`
`plots`	First / default plot column
`plot_meta`	`{ title }[]`
`count`	Bar count consumed
`script_name` / `script_type`	From `indicator` / `strategy`
`mode`	`"interpret"` or `"compile"` (what actually ran)
`auto_backend`	When `mode: "auto"`
`compile_fallback_reason`	Why auto fell back
`events`	`StrategyEvent[]`
`fills`	Broker fills
`drawings`	`line` / `label` / `box` / …
`logs`	`log.*` records
`strategy`	Book scalars when the script is a strategy
`error` / `error_kind`	Soft failure (`parse` | `runtime` | …)

Parse and runtime failures **return** an error envelope; they do not throw from `run` (except unexpected host bugs). `interpret(source, bars)` (helper) **does** throw if `out.error`.

Stream

```
const s = new Runtime("AAPL").stream(src);
s.on("bar", (out) => { /* full re-eval on bars so far */ });
s.on("error", (err) => {});
s.push({ close: 10 });
s.push({ close: 11 });
s.close();
```

Each `push` re-runs the script on **all bars so far** (Python-shaped, not incremental-only).

Providers

```
const provider = new MemoryProvider({
  AAPL: [{ close: 1 }, { close: 2 }, { close: 3 }],
});
const out = await new Runtime("AAPL").runProvider(src, provider, { limit: 3 });
```

Also: `StaticMapProvider`, `JsonBarProvider`. `request.security` without a matching feed stays `na`.

Internals

Path	Role
`src/runtime/interpret.ts`	Host, `Runtime`, `interpretTree`
`src/runtime/series.ts`	`NA`, `PineSeries`
`src/runtime/ta.ts`	Incremental `TaEngine`
`src/runtime/strategy.ts`	Broker, fills, risk, OCA
`src/runtime/request.ts`	`request.security` policy
`src/runtime/library.ts`	In-process `import ns/Name/ver`
`src/runtime/provider.ts`	Bar providers
`src/runtime/drawings.ts`	Drawing book + GC

Invariants & edge cases

1. **`na` is `null`.** A non-finite number in or out becomes `na`.
2. **`na == na` is true.** Any other comparison involving `na` is false.
3. **Lookback:** `close[1]` is the previous bar; OOB → `na`.
4. **Call-site TA.** Two `ta.sma(close, 14)` at different AST nodes do **not** share state.
5. **Foreign / HTF `request.security` without data → `na`.** Same-symbol simple OHLCV may passthrough; the chart is never invented as another ticker.
6. **v3/v4 bare aliases** (`sma`, `ema`, `rsi`, …) resolve when Python does.

7. Libraries: `registerLibrarySource` then `import namespace/Name/version`. Unresolved aliases stub to `na` (soft), they do not crash the run.

Worked examples

na and lookback

```
const out = new Runtime("TEST").run(
  `indicator("t")
plot(close[1])
plot(na == na ? 1 : 0)`,
  [{ close: 10 }, { close: 20 }],
);
// last bar: plots[0] === 10, plots[1] === 1
```

Inputs and strategy

```
const out = new Runtime("AAPL", {
  inputs: { len: 5 },
  broker: { commission: 0.001, pyramiding: 1 },
}).run(src, bars);

if (out.error) throw new Error(out.error);
console.log(out.events, out.fills, out.strategy);
```

Failure modes

Symptom	Cause	Fix
`error_kind: "parse"`	Grammar failure	`parse(source)` / `pynets check`
All `request.security` values `null`	No foreign feed	Expected. Pass a provider or accept `na`
Plots off-by-one vs TV	Lookback / `na` policy	Compare to **Python**, not to TradingView
Stream looks "slow"	Full re-eval each push	Expected host model
`mode` in the result is `interpret` after `auto`	Compile ineligible / error	Read `compile_fallback_reason`

See also

- JS compile
- Parity
- Python series and history
- request.*
- Evaluate scripts

PYNETS COMPILE

JS bar-loop emit for mode compile | auto. Python object-mode analog — not Numba. auto falls back to interpret.

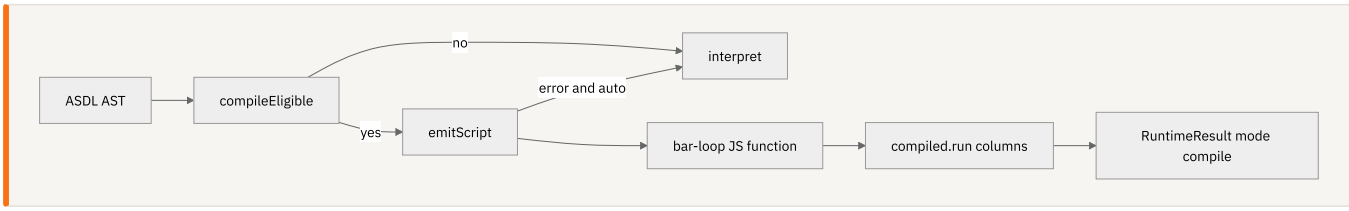
PYNETS COMPILE

Abstract

PyneTS compile (`mode: "compile"` / `"auto"`) **emits a JavaScript bar-loop function**. It is the analog of Python **object-mode** compile, **not** Numba nopython. There is no LLVM, no `numba` , no disk IR cache like `PYNE_COMPILE_CACHE_DIR` .

Landed in `@hoox-sh/pynets` **0.2.0**. Interpret is unchanged. This PYNE checkout's `pynets/` pin is **v0.2.0** and includes `src/runtime/compile/` . Python Runtime remains the oracle.

Conceptual model



	Python compile	PyneTS compile
Backend	Numba nopython **or** object-mode Python loop	**JS emit only**
Default `Runtime.run`	`interpret` (env override)	`interpret`
`auto`	try compile, fallback interpret	same
`import` / foreign `request.*`	limited; often fallback	inline registered libs; foreign → `na`
Alerts / drawings / strategy	object-mode path	`__h.strategy` , drawings, `alertcondition`

Interface surface

```
  compileScript,
  transpile,
  compileEligible,
  clearCompileCache,
  compileCacheStats,
  runScript,
  CompileError,
} from "@hoox-sh/pynets";

new Runtime("AAPL", { mode: "compile" }).run(src, bars);
new Runtime("AAPL", { mode: "auto" }).run(src, bars);
```

Name	Role
`compileEligible(tree)`	`{ ok, reason? }`
`transpile` / `compileScript`	Emit + wrap `CompiledScript`
`compiled.run(open, high, low, close, volume, time, extras)`	Columnar execute
`clearCompileCache` / `compileCacheStats`	Process-local emit cache

CLI: `pynets run script.pine --mode compile` or `--mode auto`.

What emit covers (0.2.0)

Strategy object-mode (`__h.strategy` / fills / events), UDF series state (`src[1]` , `var` locals), array / map / matrix, UDT (`Type.new` , field get-set, `method`), drawings (`label` / `line` / `box`), named UDF kwargs, input overrides, `color.*` , enums, `request.security` same-symbol passthrough (foreign → `na`), `barstate.*` / `syminfo` stubs, calendar, `str.*` / `str.format` , `import` (inline registered sources; unresolved → `na`), `session.*` / `chart.*` , `log.*` , `ticker.new` / `standard` / `heikinashi` , `request.currency_rate` , `alertcondition` .

`mode: "auto"` no longer skips compile when inputs are set.

Internals

Path	Role
`src/runtime/compile/engine.ts`	`compileScript` , `runScript` , cache
`src/runtime/compile/emit.ts`	JS text emit
`src/runtime/compile/emit_call.ts`	Builtin / UDF calls
`src/runtime/compile/emit_udf.ts`	User functions
`src/runtime/compile/runtime.ts`	Host helpers used by emitted code
`src/runtime/interpret.ts` `runCompiled` / `runAuto`	`Runtime.run` wiring

Do not edit `src/runtime/compile/*` and `src/runtime/interpret.ts` in the same agent turn. Parent wires `Runtime.run({ mode })` .

Invariants & edge cases

1. **Not Numba.** Do not document `nopython`, `njit`, or `PYNE_COMPILE_CACHE_DIR` for this package.
2. **Python still wins** if compiled plots disagree with Python interpret on the same script+bars — unless Python compile is the thing being compared, in which case compare apples to apples (`object` vs `object`).
3. **`import` / foreign `request.*` stay interpret-or-na.** No invented bars.
4. `result.mode` reports what **ran** (`interpret` or `compile`). After `auto` fallback, read `compile_fallback_reason`.
5. Compile failures in `mode: "compile"` return an error envelope; `auto` falls back.

Worked examples

```
const interpret = new Runtime("AAPL", { mode: "interpret" }).run(src, bars);
const compiled = new Runtime("AAPL", { mode: "compile" }).run(src, bars);
const auto = new Runtime("AAPL", { mode: "auto" }).run(src, bars);

if (compiled.error) {
  // inspect compiled.error / error_kind
}
console.log(auto.mode, auto.auto_backend, auto.compile_fallback_reason);
```

```
pynets run script.pine --mode auto --json
```

Failure modes

Symptom	Cause	Fix
`CompileIneligibleError` / fallback	Surface not emitted	`interpret`, or extend emit (parity first)
Plots differ interpret vs compile	Emit bug	Add `test/compile_*.test.ts`; Python is still SoT
Expecting Numba speedups	Wrong mental model	This is JS object-mode
Edited compile + interpret together	Review / merge hazard	Split the change

See also

- PyneTS runtime
- Parity
- Python compiler overview
- Python Numba path
- Modes

PYNETS PARITY

How to compare @hoox-sh/pynets Runtime.run plots against Python pynescript.runtime. Python wins ties.

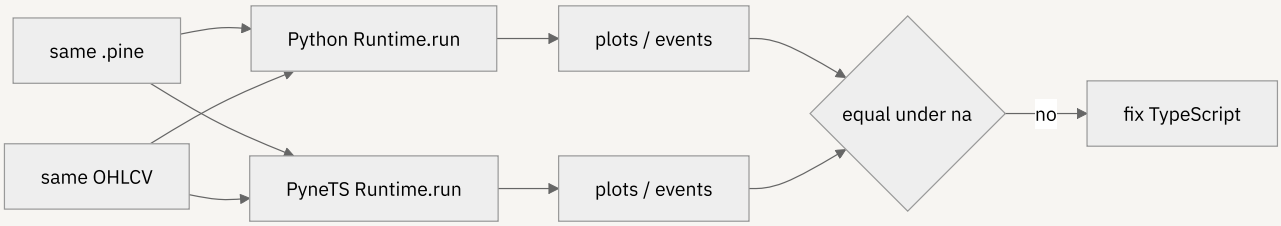
PYNETS PARITY

Abstract

Parity here means **TypeScript `Runtime.run` vs Python `Runtime.run`** on the same source and the same bars — not vs TradingView. Interpret↔compile alignment **inside** PyneTS is a second, narrower check (`test/compile_*.test.ts`).

Workflow when adding or fixing a builtin: read the Python handler, port `na` / lookback / call-site semantics, wire the name, add a focused `test/interpret_*.test.ts`, compare plots if PYNE is present.

Conceptual model



Interface surface

Commands

```
# PyneTS
cd /path/to/pynets
bun test
bun test test/interpret_ta.test.ts
bun run typecheck

# Python oracle (sister checkout)
cd /path/to/pynescript
python -c "from pynescript.runtime import Runtime; ..."
```

Use `/pynets-parity` when adding or fixing builtins. Use `/pynets-generate` when touching grammar output.

What to compare

Field	Rule
`plots` / `series`	Equal length; `null` ↔ Python `None`; finite numbers match
`events`	Kind, direction, bar, id — same contract as strategy events
`error_kind`	Same class of failure (`parse` vs `runtime`)
TradingView screenshots	**Out of scope** as an oracle

`na` encoding: Python interpret uses `None`; JSON / TS uses `null`; Python Numba uses `nan`. Compare after normalizing non-finite → `na`.

Internals

Path	Role
`test/interpret_*.test.ts`	Focused interpret cases
`test/compile_*.test.ts`	JS emit vs interpret
`test/helpers/first_party.ts`	First-party fixtures (live in PYNE)
PYNE `src/pynescript/ast/evaluator/builtins/`	Handler SoT
PYNE `tests/test_first_party_ta_goldens.py`	Python goldens

Do **not** treat historical `pine-worker/test/parity/` JSON as the PyneTS harness. That tree lives in the leftover TypeScript Worker sister (hoox-sh/pine-worker), not here.

This PYNE `pynets/` pin (**v0.2.0**) includes interpret tests and compile-vs-interpret cases (`test/compile_*.test.ts`). Python Runtime remains the oracle.

Invariants & edge cases

1. **Python wins ties.** If TS and Python disagree, fix TS (unless Python is proven wrong — then fix both + tests).
2. **Do not "improve" semantics.** Port `na`, lookback, and call-site keys as Python implements them.
3. Same-symbol `request.security` may passthrough; **foreign without data is `na` on both sides**.
4. Compile-vs-interpret inside PyneTS can pass while TS-vs-Python still fails — check both.
5. First-party fixtures live in PYNE; do not copy corpora into `hoox-sh/pyne`.

Worked examples

Minimal Python vs TS

Python:

```
from pynescript.runtime import Runtime

src = 'indicator("t")\nplot(close[1])'
bars = [{"close": 10}, {"close": 20}]
print(Runtime(symbol="TEST").run(src, bars))
```

TypeScript:

```
const src = 'indicator("t")\nplot(close[1])';
const bars = [{ close: 10 }, { close: 20 }];
console.log(new Runtime("TEST").run(src, bars));
```

Last-bar plot should be `10` on both.

Adding a builtin

1. Read `src/pynescript/ast/evaluator/builtins/…` (Python).
2. Port into `src/runtime/ta.ts` / `math.ts` / … and `evalCall` in `interpret.ts`.
3. Add `test/interpret_<area>.test.ts`.
4. If PYNE is present, compare `Runtime.run` plots on the same bars.
5. `bun test` and `bun run typecheck`.

Failure modes

Symptom	Cause	Fix
Off-by-one vs Python	Lookback / bar_index	Match `PineSeries` to Python
`null` vs `0`	`na` collapsed	Keep `null`; do not coerce
Foreign security equals chart	Invented bars	Return `na`
Compile matches interpret, both wrong	Shared host bug	Fix interpret first (Python SoT)
Copied sources into PYNE	Submodule rule	PYNE consumes `pynets/` only

See also

- PyneTS hub
- Runtime
- Python interpret↔compile parity
- Compatibility
- Contributing

PYNE-WORKER

Production Python Cloudflare Worker for POST /run. Thin wrap over pynescript.runtime. Sister repo — not in this checkout.

PYNE-WORKER

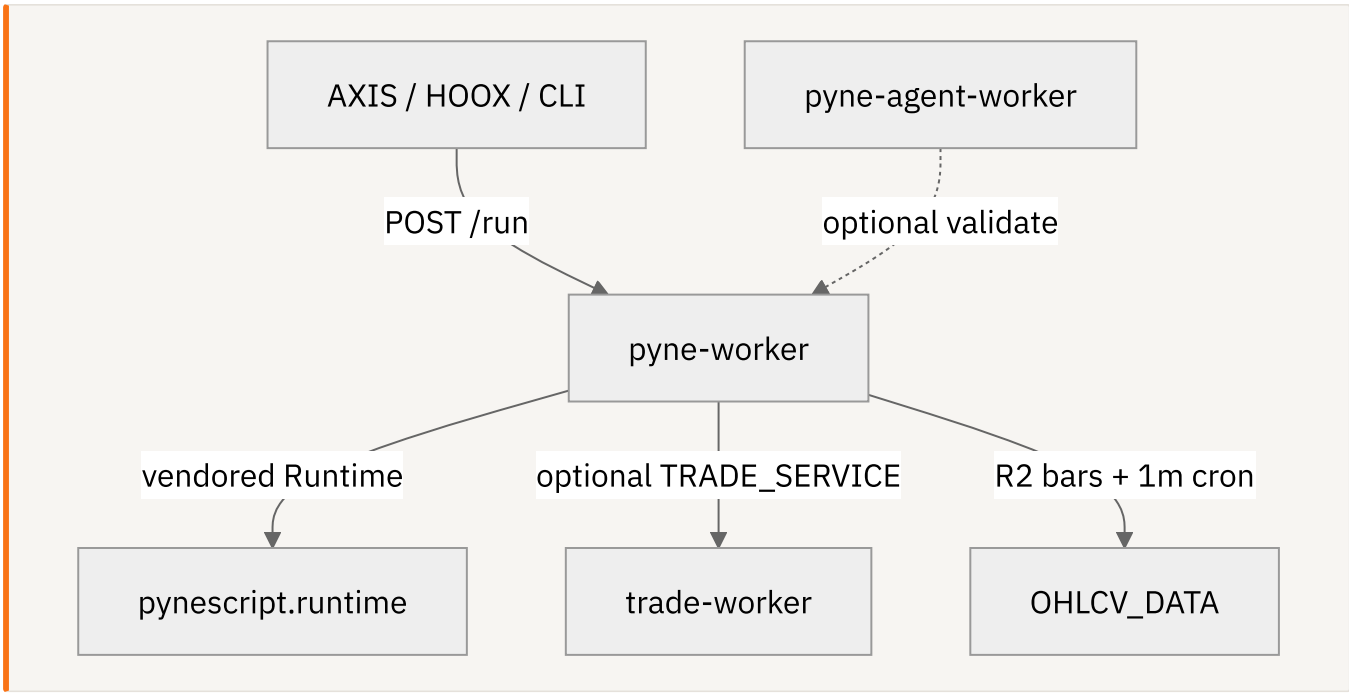
Abstract

pyne-worker is the production **edge evaluate host**: a Python Cloudflare Worker that vendors `pynescript.runtime` and speaks the same evaluate contract as Flask `POST /run`. It is a **sister repository** — it does **not** live in this PYNE checkout.

Repo	hoox-sh/pyne-worker
Role	Edge `POST /run` + alerts + libraries + cron
Runtime	Cloudflare Workers (Python)
Engine	Vendored `hoox-pyne` (`pynescript.runtime`)
In this repo?	**No**

This Worker is a **thin wrap**. CLI, LSP, Flask Pro API, Numba desk compile, and the language SoT stay in **hoox-sh/pyne**.

Conceptual model



Interface surface

This Worker	Use PYNE (`hoox-pyne`) instead
`POST /run` + alerts + `libraries`	`pyne` CLI, `pyne-lsp`, VS Code
R2 bars, 1m cron, L2 webhooks	Flask Pro API, Docker, `/run/batch`
Vendored engine (`./scripts/sync_vendor.sh`)	Live package, Numba, corpus harness
30s / 100KB / 100K bars / 5MB envelope	Uncapped research on your machine

Taxonomy

Name	What it is
PYNE	Language SoT (this repo)
PyneTS	TypeScript **library** (`@hoox-sh/pynets`)
pyne-worker	Python Cloudflare **evaluate** host
pyne-agent-worker	NL **authoring** host (optional validate via this Worker)

Do **not** create `pynescript/pyne-worker/` or `pynescript/pine-worker/`.

The leftover name **pine-worker** is a TypeScript Cloudflare experiment (hoox-sh/pine-worker). It is not this host and is not documented on this site. TypeScript library work is PyneTS.

Relationship to PYNE

- Python Runtime remains the **oracle**. The Worker vendors a snapshot; it can lag `__about__.py`.
- After a PYNE release, operators re-run `./scripts/sync_vendor.sh` in the Worker checkout before `wrangler deploy`.
- Shared JSON: evaluate contract. Alerts: Alerts.

Quick links

- Evaluate (`POST /run`)
- Deploy
- GitHub README

See also

- pyne-agent-worker
- PyneTS
- Pro API usage
- Ecosystem

PYNE-WORKER EVALUATE

POST /run, libraries, alerts, scripts, ingest, and 1m cron on the Python edge host.

PYNE-WORKER EVALUATE

Abstract

The Worker's evaluate path is `POST /run` with the same envelope as Flask: `script` + OHLCV + `mode` → `plots` / `series` / `events` / `alerts`. Auth is `X-API-Key` (`API_KEY` secret). Wall timeout defaults to **30s**.

Sister repo: hoox-sh/pyne-worker. Contract: evaluate contract.

Endpoints

Method	Path	Auth	Role
`GET`	`/health`	no	Liveness + binding flags
`POST`	`/run`	yes	Evaluate (`script` or `script_id`)
`POST`	`/ingest`	yes	Upload OHLCV to R2
`POST`	`/scripts`	yes	Deploy a script
`GET`	`/scripts`	yes	List deployed scripts
`GET`	`/scripts/:id`	yes	Get deployed script
`DELETE`	`/scripts/:id`	yes	Delete deployed script
`GET`	`/cron/jobs`	yes	List bar-close jobs
`PUT`	`/cron/jobs`	yes	Replace job list
`POST`	`/cron/run`	yes	Trigger scheduler
`POST`	`/feed/refresh`	yes	Pull klines into R2

`POST /run`

Accepts `data` (Pro API name) or `ohlcv`. Omit bars and pass `symbol` + `timeframe` to load R2 history.

```
{
  "script": "//@version=5\nindicator(\"t\")\nplot(close)",
  "ohlcv": [
    {"open": 100, "high": 105, "low": 95, "close": 102, "time": 1000, "volume": 1000}
  ],
  "symbol": "BTCUSDT",
  "mode": "auto"
}
```

Field	Notes
`script`	Inline Pine
`script_id`	Load a deployed script instead of `script`
`ohlcv` / `data`	Bars; or omit and use R2
`mode`	`interpret` (default) · `compile` · `auto`
`inputs`	`input.*` overrides (forces interpret under `auto`)
`libraries`	`[{namespace, name, version, source}]` — max 32
`profiler`	Per-line interpret timings
`timeout_seconds`	Wall budget (default 30, cap 30)
`webhook_url`	Per-job L2 alert URL (else `ALERT_WEBHOOK_URL`)

Success includes `status`, `plots`, `series`, `plot_meta`, `alerts`, `events`, `drawings`, `inputs`, `logs`, `meta`. Failures use `error` / `error_kind`.

Deployed scripts can store `libraries` and `inputs`. `script_id` and cron reuse them.

Alerts and trades

- `alert()` / `alertcondition()` export on `/run` (same engine as Pro API).
- L2 HTTP POST: secret `ALERT_WEBHOOK_URL` or per-job `webhook_url`.
- Optional `TRADE_SERVICE` binding forwards strategy events to **trade-worker**. Remove the `services` block if that Worker is not in the account (fails closed).

Limits

Cap	Value
Script	100 KB
Bars	100K
Envelope	5 MB
`/run` wall	30 s
Rate	100 req / 60 s

See also

- pyne-worker
- Deploy
- Alerts
- Flask `/run`
- PyneTS — in-process `RuntimeResult`, not HTTP
- pyne-agent-worker — optional generate → `/run` loop

PYNE-WORKER DEPLOY

Vendor the PYNE engine, wrangler secrets, R2 bars, and 1m cron. Sister checkout only.

PYNE-WORKER DEPLOY

Abstract

Deploy from the **hoox-sh/pyne-worker** checkout. Wrangler packages `python_modules/pynescript`, **not** an editable PYNE install. After every engine bump, run `./scripts/sync_vendor.sh`.

Quick start

```
cd ~/Git/pyne-worker
pip install -e ".[dev]"
pytest -v
./scripts/sync_vendor.sh
npx wrangler deploy
echo "my-secret-key" | npx wrangler secret put API_KEY
```

Deploy to Cloudflare deploys **this host only**.

Dashboard prompts: `API_KEY` (required), optional `ALERT_WEBHOOK_URL`, `INTERNAL_KEY_BINDING`. R2 bucket `OHLCV_DATA` comes from `wrangler.jsonc`.

If `TRADE_SERVICE` points at a missing `trade-worker`, **delete the `services` block** and retry.

Vendor sync

```
./scripts/sync_vendor.sh
npx wrangler deploy
```

Skip sync and the edge still runs the last snapshot (may lag `hoox-pyne` 0.3.14+).

Script + 1m cron

```
python scripts/fetch_and_ingest.py \
  --symbol BTCUSDT --timeframe 1m \
  --ingest-url "$WORKER/ingest" --api-key "$API_KEY"

curl -sS -X POST "$WORKER/scripts" \
  -H "Content-Type: application/json" -H "X-API-Key: $API_KEY" \
  -d '{"id":"btc-sma","script":"//@version=5\nindicator(\"s\")\nplot(close)","symbol":"BTCUSDT","t

curl -sS -X POST "$WORKER/cron/run" \
  -H "Content-Type: application/json" -H "X-API-Key: $API_KEY" \
  -d '{"force":true}'
```

Cron in `wrangler.jsonc` is `* * * * *`. Each tick pulls **closed** klines (Bybit, Binance fallback) and evaluates only when the last closed bar advanced.

See also

- pyne-worker
- Evaluate
- pyne-agent-worker — optional generate → `/run` loop
- PyneTS

95 · AGENT

PYNE-AGENT-WORKER

Natural-language PYNE Agent on Cloudflare Workers AI. Standalone chat, optional pyne-worker validate loop, AXIS plugin. Sister repo.

PYNE-AGENT-WORKER

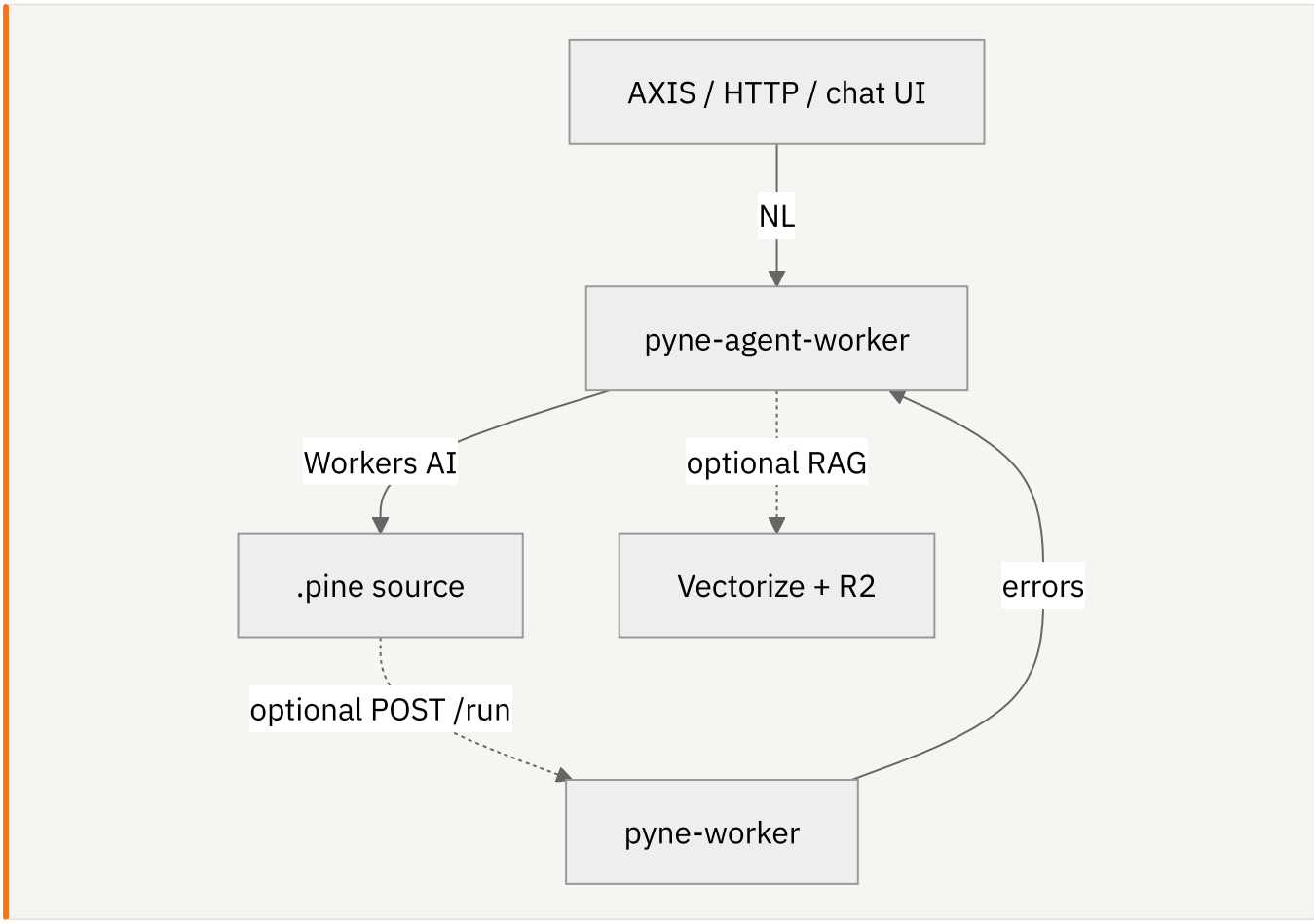
Abstract

pyne-agent-worker is the edge **authoring** host: natural language in, Pine source out. It is a **sister repository** — it does **not** live in this PYNE checkout.

Repo	hoox-sh/pyne-agent-worker
Role	Write scripts via chat (PYNE Agent) + optional RAG
Runtime	Cloudflare Workers (TypeScript) + Workers AI
AXIS	`GET /plugin/axis-pine-agent.js`
In this repo?	**No**

You do **not** need pyne-worker, trade-worker, or the HOOX mesh to chat. The agent **writes** source. It does not import PyneTS and it does not run the bar-loop.

Conceptual model



Mode	Requirement	Behavior
Standalone	AI binding (+ optional KB)	`POST /v1/chat` → Pine; validation skipped
HOOX-enhanced	+ pyne-worker URL or service binding	generate → `POST /run` → fix retries

`GET /health` reports `"mode": "standalone" | "hoox"`.

Taxonomy

Name	What it is
PYNE	Language SoT (this repo)
PyneTS	TypeScript **library** (`@hoox-sh/pynets`) — `parse` / `Runtime.run`
pyne-worker	Python Cloudflare **evaluate** host
pyne-agent-worker	NL **authoring** host (this Worker)

Hard rule

This git tree **never** contains TradingView built-in Pine sources. Knowledge is operator-ingested into private R2 + Vectorize (`ingest:docs` / `ingest:corpus` / `ingest:builtins`). `scripts/legal-check.sh` refuses tracked `*.pine` on deploy.

Relationship to PYNE

- PYNE remains the **evaluate** oracle. The agent **writes**.
- Optional validate uses the evaluate contract against pyne-worker or the Pro API.
- AXIS install URL: `https://<pyne-agent-worker>/plugin/axis-pine-agent.js`

Pages

- Chat API
- Deploy

See also

- pyne-worker
- PyneTS
- Evaluate contract
- AXIS plugin
- Ecosystem

96 · AGENT

PYNE-AGENT-WORKER CHAT

POST /v1/chat, search, sessions, and the optional pyne-worker validate loop.

PYNE-AGENT-WORKER CHAT

Abstract

Chat is `POST /v1/chat`. When `API_KEY` is unset, open mode is allowed for **local dev only**. Admin embed/index always require `API_KEY`.

Sister repo: hoox-sh/pyne-agent-worker.

Endpoints

Method	Path	Auth	Role
`GET`	`/health`	no	Liveness + `mode`
`GET`	`/`	no	Demo chat UI
`POST`	`/v1/chat`	yes*	NL → Pine
`GET` / `POST`	`/v1/search`	yes*	Vectorize search
`POST`	`/v1/sessions`	yes*	Create session
`GET`	`/v1/sessions/:id`	yes*	Session + messages
`POST`	`/v1/admin/embed`	yes	Batch embeddings
`POST`	`/v1/admin/index`	yes	Embed + R2 + Vectorize
`GET`	`/plugin/axis-pine-agent.js`	no	AXIS plugin module

* Required when `API_KEY` is set.

`POST /v1/chat`

```
{
  "message": "v6 RSI strategy with ATR trailing stop",
  "pine_version": "v6",
  "style": "strategy",
  "session_id": null,
  "validate": true,
  "max_retries": 2
}
```

Response includes `reply`, extracted `pine`, `validation` (attempts / pyne-worker errors), `rag` hits, and a trademark disclaimer.

Validate loop (optional)

Only when `PYNE_SERVICE` or `PYNE_WORKER_URL` is set:

```
generate (Workers AI)
  → extract ```pine
  → POST pyne-worker /run (synthetic OHLCV)
  → ok? return
  → else fix prompt + retry (max_retries)
```

Without a binding, `validation.available` is `false` and the Worker returns the first draft.

See also

- pyne-agent-worker
- Deploy
- pyne-worker `/run`
- PyneTS — in-process TS evaluate, not this chat API

PYNE-AGENT-WORKER DEPLOY

Standalone Workers AI deploy, optional RAG, and optional pyne-worker validate binding.

PYNE-AGENT-WORKER DEPLOY

Abstract

Deploy from **hoox-sh/pyne-agent-worker**. Minimum is Workers AI. Vectorize / R2 / D1 improve quality; they are not required to chat.

Standalone

```
cd ~/Git/pyne-agent-worker
bun install

# Optional RAG / sessions
npx wrangler vectorize create pyne-agent-kb --dimensions=768 --metric=cosine
npx wrangler r2 bucket create pyne-agent-kb
npx wrangler d1 create pyne-agent-sessions
npx wrangler d1 execute pyne-agent-sessions --remote --file=schemas/sessions.sql

cp .env.example .dev.vars
bun run dev
echo "your-secret" | npx wrangler secret put API_KEY
bun run deploy
```

No HOOX mesh and no pyne-worker binding.

Optional validate loop

If you already run pyne-worker:

```
// wrangler.jsonc — uncomment services, OR:
// "vars": { "PYNE_WORKER_URL": "https://pyne-worker.<you>.workers.dev" }
```

```
echo "pyne-api-key" | npx wrangler secret put PYNE_WORKER_API_KEY
```

Knowledge (private)

```
bun run ingest:docs -- --dir /secure/pine-docs-v6 --version v6
bun run ingest:docs -- --pyne-docs ../pynescript/docs/pyne
bun run ingest:corpus -- --dir /secure/open-pine-corpus --max 1000
bun run ingest:builtins -- --metadata ../pynescript/src/pynescript/langserver/providers/builtin_me
bun run scripts/build-index.ts \
  --embed-endpoint https://pyne-agent-worker.<you>.workers.dev/v1/admin/embed
```

Never commit TradingView built-in sources. `scripts/legal-check.sh` runs on predeploy.

AXIS

```
https://pyne-agent-worker.<you>.workers.dev/plugin/axis-pine-agent.js
```

Set **endpoint** + **API key** in the plugin config.

See also

- pyne-agent-worker
- Chat API
- pyne-worker — optional validate host
- PyneTS
- AXIS plugin