HOOX · DOCUMENTATION

PLATFORM ENGINEERS · SRES · CONTRIBUTORS

DEVOPS MANUAL

Architecture, workers, deployment, security, and API

2026-07-22 DIN A4 · HTTPS://HOOX.SH/DOCS OPEN SOURCE

CONTENTS

32 API Endpoint Directory

33 Request Payload Schemas

34 Standard Response Schemas

35 CLI Architecture & Features

36 TUI Code Architecture

37 Workers Pattern Consolidation

01 · DEVOPS MANUAL

DEVOPS MANUAL

Infrastructure provisioning, deployment sequences, secret management, and edge operations for HOOX administrators.

This manual is the primary, production-grade reference for system administrators, security engineers, and platform operators responsible for provisioning edge databases, orchestrating secure V8 service bindings, deploying multi-exchange execution workers, auditing secrets, and monitoring system health.

```
graph TD
    subgraph "Public Entryways"
        GW[hoox Gateway Isolate]
        DASH[dashboard Next.js OpenNext]
    end
    subgraph "Private V8 Compute Layer"
        TW[trade-worker Execution Engine]
        D1W[d1-worker SQL Hub]
        TGW[telegram-worker Alerts]
        AW[agent-worker AI Risk Monitor]
    end
    subgraph "Edge Data Storage Layer"
        D1[(D1 SQLite Database)]
        KV[(CONFIG_KV Namespace)]
        R2[(R2 Log offload Bucket)]
    end

    GW -->|Service Binding| TW
    GW -->|Service Binding| TGW
    TW -->|Service Binding| D1W
    AW -->|Service Binding| TW
    D1W -.->|SQLite read/write| D1
    TW -.->|Config Read| KV
    TW -.->|Log Offload| R2
    DASH -->|Service Binding| GW
```

Operator's System Directory

The DevOps manual is structured into five core architectural operational layers:

1. Operations & Setup Runbooks

Complete guides for bootstrapping environments, managing interactive terminal dashboards, and diagnosing local runner configurations:

- **Operations & Troubleshooting** — Core operations manual, 31-key environment variable matrices, and diagnostic codes.
- **Core Installation Flow** — Guided, step-by-step local machine and edge provision setup.
- **Terminal UI Cockpit Development** — Code architectures, stores, and operations of the OpenTUI monitor.

2. Architectural Specifications

Deep dives into latency structures, isolation parameters, and bindings linkages:

- **System Topology Overview** — Architectural layout, regional edge clustering, and scaling limits.
- **Isolate Communication** — Deep dive into Service Bindings, zero-TCP routing, and V8 engines.
- **Data Flow Architecture** — Flowcharts for trade execution, backup queues, and cron risk monitoring.
- **Bindings Matrix** — Exact mapping of D1, KV, Queues, R2, and Service Bindings.
- **Storage Engineering** — Persistent storage boundaries, SQLite DDL parameters, and R2 buckets.
- **Internal Endpoints Map** — Sub-millisecond binding paths and routing maps.
- **Visual Tokens & Design System** — Monochromatic token mappings and visual SVGs catalog.

3. Edge Worker Microservices (10 Profiles)

Individual developer profiles for each running isolate, cataloging bindings, custom middlewares, and API formats:

- **hoox Gateway** · **trade-worker** · **agent-worker** · **telegram-worker**
- **d1-worker** · **web3-wallet-worker** · **email-worker** · **analytics-worker**
- **report-worker** · **dashboard (Next.js OpenNext)**

4. Deployment, WAF, & CI/CD Pipelines

Rollout manuals, Access gates, and GitHub Actions telemetry:

- **Production Deployment** — Wrangler commands, Account provisioning, and production variables.
- **CI/CD Workflow pipelines** — GitHub Actions secrets, syntax tests, and automated edge uploads.
- **Monitoring & Telemetry** — Live wrangler logs streaming and custom Analytics Engine metrics.
- **Cloudflare Zero Trust Corridor** — Setting up Access client corridors, IP firewalls, and WAF rules.

5. Developer & API Reference

TypeScript interfaces, compiler settings, Bun test specs, and HTTP schemas:

- **Wrangler Dev Setup** · **Testing Standards** · **Debugging Runbook**
- **Exposed API Routes** · **Request Payloads** · **Standard Responses**
- **CLI Commands Engine** — Command-line argument parsing, binary execution, and JSON flags.

First time deploying a Hoox workspace to production? Start with the **Production Deployment Manual** to verify your Cloudflare Account permissions and execute sequential deployments seamlessly.

Quick Links

- **End-User Documentation Hub** — Standard setup, cURL webhooks, and TradingView Pine Scripts.
- **Hoox Git Submodules** — Central monorepo codebase.

Enterprise (Commercial Layer)

Full institutional architecture documentation (Workers for Platforms, Workflows, Logpush audit, Enterprise security) for the **closed-source commercial layer** lives at docs/enterprise/.

See root OPEN_CORE.md and OPEN_CORE_FEATURE_SPLIT.md. This public repo only includes the open core + public design docs.

02 · DEVOPS MANUAL

CLOUDFLARE® WORKERS SETUP FLOW

Detailed system onboarding, toolchain validation steps, wrangler.jsonc schemas, and Secret Store binding architectures.

This document details the step-by-step installation, bootstrapping, and validation workflows executed by the Hoox CLI during project initialization.

The Hoox CLI enforces strict type validation for all configuration files via the `Config` and `WorkerConfig` TypeScript interfaces defined in `packages/cli/src/core/types.ts`. Avoid using `as any` or type bypasses when extending wrangler parameters to prevent build-time CLI crashes.

Onboarding Wizard (`hoox onboard`)

> **v0.8.0 update:** The recommended entry point is now `hoox onboard`, which chains `init` (configuration) and `setup` (infrastructure) into a single command. Running `hoox` with no arguments on an uninitialized workspace will auto-launch `onboard`.

To start the system bootstrap, run the one-shot wizard from the monorepo root:

```
hoox onboard
```

For fine-grained control, run the two steps separately:

```
hoox init    # 1. Write wrangler.jsonc, collect integration secrets
hoox setup   # 2. Generate keys, apply D1 schema, push secrets, deploy dashboard
```

The setup wizard guides you through these critical onboarding phases:

Phase 1: Cloudflare Authentication

Prompts for your Cloudflare API token (with `Account.D1`, `Account.KV`, `Account.Workers` permissions) and Account ID.

Phase 2: Microservice Profile Selection

Lets you selectively enable or disable specific workers from the `workers/` directory based on your trading intent (e.g., cross-margin futures execution vs. Web3 DeFi wallet swaps). Disabling unnecessary workers saves deployment bandwidth and keeps resource bindings clean.

Phase 3: Integration Secrets

Collects any integration secrets required by the selected workers (exchange API keys, Telegram bot tokens, AI provider keys) and writes them to local `.dev.vars` files.

Phase 4: Configuration Write

Consolidates all chosen parameters and writes your central `wrangler.jsonc` file, mapping out variables and bindings for every worker.

Phase 5: Infrastructure Provisioning

Generates internal auth keys, creates D1 databases, applies the schema, and pushes secrets to Cloudflare. Builds and deploys the Next.js dashboard.

Phase 6: Verification

Runs `hoox check setup` to verify that all components are operational.

Configuration Files Spec

The Hoox platform uses a dual configuration file architecture to track workspace states:

A. wrangler.jsonc (Central Settings)

This file represents the declarative single source of truth for your monorepo's active workers:

```
{
  "global": {
    "cloudflare_account_id": "debc6545e63bea36be059cbc82d80ec8",
    "subdomain_prefix": "hoox",
  },
  "workers": {
    "d1-worker": {
      "enabled": true,
      "path": "workers/d1-worker",
      "vars": { "database_name": "trade-data-db" },
    },
    "trade-worker": {
      "enabled": true,
      "path": "workers/trade-worker",
      "secrets": ["BYBIT_API_KEY", "BYBIT_API_SECRET", "TELEGRAM_BOT_TOKEN"],
    },
  },
}
```

B. .install-wizard-state.json (Onboarding State)

During the interactive setup, the CLI caches your current step and intermediate inputs inside `.install-wizard-state.json` at your project root.

- **State Recovery**: If your terminal session is disconnected or wrangler login prompts timeout, you can run `hoox onboard` (or `hoox init --resume`) again. The CLI will detect the state file and seamlessly resume your onboarding from the last incomplete step.
- **Auto-Cleanup**: Upon final completion of Phase 7, the state file is automatically purged to keep your root directory clean.

Secret Bindings Architecture

Hoox utilizes Cloudflare's hardware-secured **Secret Store** to bind environment credentials to V8 isolates without exposing them in git history.

Local Mocking (`.dev.vars`)

During local development, wrangler dev looks for a local, gitignored file called `.dev.vars` inside each worker's directory to simulate secrets:

```
# workers/trade-worker/.dev.vars
BYBIT_API_KEY=mock_bybit_development_key
BYBIT_API_SECRET=mock_bybit_development_secret
```

Production Secret Bindings

When deploying to production, wrangler binds these variables using direct encrypted environments in your worker's wrangler configuration:

```
{
  "secrets_store": {
    "bindings": [
      {
        "binding": "BYBIT_API_KEY_BINDING",
        "store_id": "48433bc559a943f09d9d6c622e188fd5",
        "secret_name": "BYBIT_API_KEY"
      }
    ]
  }
}
```

This guarantees that secrets are never logged, never cached in plain text on disk, and are only accessible inside your worker's sandboxed execution isolate memory.

Made a configuration mistake or changed your subdomain? You can re-run `hoox check-setup` at any time to execute high-integrity type validation and ensure all bindings and configurations match production examples perfectly!

Next Steps

- **DevOps Setup & Operations Manual** — Dive into complete operations, variable matrices, and troubleshooting.
- **Terminal UI Cockpit** — Run, hot-reload, and monitor your local workers via TUI.

03 · DEVOPS MANUAL

HOOX TRADING SYSTEM — COMPLETE SETUP & OPERATIONS GUIDE

Version: 2.0.0

> **Version:** 2.0.0
> **Last Updated:** 2026-05-12
> **Classification:** Internal Operations Manual
> **Audience:** DevOps Engineers, Senior TypeScript Developers, System Administrators

---

Table of Contents

---

1. System Overview

1.1 Architecture

Hoox is an edge-deployed cryptocurrency trading system built on Cloudflare Workers. It consists of:

Layer	Components
Gateway	`hoox` (webhook entrypoint, DO idempotency, KV-backed rate limiting)
Execution	`trade-worker` (multi-exchange), `web3-wallet-worker` (DeFi)
Intelligence	`agent-worker` (AI risk manager, 5min cron, multi-provider AI gateway)
Data	`d1-worker` (centralized D1 database service)
Notifications	`telegram-worker` (Telegram bot), `email-worker` (email signal parsing)
Analytics	`analytics-worker` (Cloudflare Analytics Engine, cross-worker observability)
Reporting	`report-worker` (Automated PDF reports via Browser Rendering, 2x daily cron)
Dashboard	`workers/dashboard` (Next.js 16 + OpenNext on Cloudflare Workers)
CLI	`packages/cli` (management tool)
Shared	`packages/shared` (types, router, middleware, utilities)

1.2 Communication Flow

```
External Webhook → hoox → [Queue] → trade-worker → D1 (via d1-worker)
                    ↓
                telegram-worker (notifications)
                    ↓
                analytics-worker (metrics)

agent-worker (cron) → trade-worker / d1-worker / telegram-worker
email-worker (cron) → trade-worker
```

1.3 Infrastructure Components

Service	Instance Name	Binding	Used By
D1 Database	`trade-data-db`	`DB`	trade-worker, d1-worker, agent-worker
KV Namespace	`CONFIG_KV`	`CONFIG_KV`	ALL workers + dashboard (config + rate limiter)
KV Namespace	`SESSIONS_KV`	`SESSIONS_KV`	hoox
R2 Bucket	`trade-reports`	`REPORTS_BUCKET`	trade-worker, report-worker
R2 Bucket	`hoox-system-logs`	`SYSTEM_LOGS_BUCKET`	trade-worker
R2 Bucket	`user-uploads`	`UPLOADS_BUCKET`	telegram-worker
Queue	`trade-execution`	`TRADE_QUEUE`	hoox (producer), trade-worker (consumer)
Vectorize	`my-rag-index`	`VECTORIZE_INDEX`	hoox, trade-worker, telegram-worker
Analytics Engine	`hoox-analytics`	(REST API)	analytics-worker (cross-worker data collection)
Durable Objects	`IdempotencyStore`	`IDEMPOTENCY_STORE`	hoox (SQLite-backed, TTL+alarm cleanup)
Browser Rendering	—	(REST API)	report-worker (PDF generation via CF API)
AI	—	`AI`	hoox, trade-worker, telegram-worker, agent-worker
Smart Placement	—	(wrangler config)	hoox, trade, agent, d1, telegram, report

2. Pre-Flight Requirements

2.1 Required Accounts

Account	Purpose	URL
Cloudflare Account	Worker hosting, D1, KV, R2, Queues	https://dash.cloudflare.com
Telegram Bot	Notifications	Via @BotFather
Exchange APIs	Trading execution	Binance, MEXC, Bybit
AI Providers (optional)	Agent intelligence	OpenAI, Anthropic, Google

2.2 Required Tools

Tool	Version	Installation Command	Verification
Bun	>=1.2	`curl -fsSL https://bun.sh | bash`	`bun --version`
Git	>=2.40	`apt install git`	`git --version`
Wrangler CLI	latest	`bun add -g wrangler`	`wrangler --version`
Node.js	>=18 (for some tools)	—	`node --version`

2.3 Required Cloudflare Permissions

Your Cloudflare API Token needs these permissions:

Permission	Scope	Why
Cloudflare Workers Scripts	Edit	Deploy workers
Account Workers Scripts	Edit	Deploy workers
Account Workers KV Storage	Edit	Manage KV
Account D1	Edit	Manage databases
Account R2	Edit	Manage buckets
Account Queues	Edit	Manage queues
Account AI	Read	Use Workers AI
Zone Settings	Read	DNS management
Zone DNS	Edit	Custom domains

2.4 Required Repository Access

You need access to clone with submodules:

```
# Main repository
git clone --recursive https://github.com/jango-blockchained/hoox-setup.git

# Or via CLI
hoox clone my-hoox-app
```

---

3. Complete Environment Matrix

3.1 Secret Inventory

All production secrets must be set via `wrangler secret put` or `hoox secrets update-cf`. Never commit secrets to version control.

Secret Name	Worker(s)	Set Via	Required For	Description
`CLOUDFLARE_API_TOKEN`	analytics-worker, CLI	`wrangler secret put`	Production	CF API token for Analytics SQL queries
`WEBHOOK_API_KEY_BINDING`	hoox	`wrangler secret put`	Production	External webhook auth key
`INTERNAL_KEY_BINDING`	hoox, trade-worker, telegram-worker	`wrangler secret put`	Production	Inter-worker auth
`AGENT_INTERNAL_KEY`	agent-worker	`wrangler secret put`	Production	Agent worker auth
`API_SERVICE_KEY`	trade-worker	`wrangler secret put`	Production	Trade worker service key
`BINANCE_API_KEY`	trade-worker	`wrangler secret put`	Optional	Binance exchange API
`BINANCE_API_SECRET`	trade-worker	`wrangler secret put`	Optional	Binance exchange secret
`MEXC_API_KEY`	trade-worker	`wrangler secret put`	Optional	MEXC exchange API
`MEXC_API_SECRET`	trade-worker	`wrangler secret put`	Optional	MEXC exchange secret
`BYBIT_API_KEY`	trade-worker	`wrangler secret put`	Optional	Bybit exchange API
`BYBIT_API_SECRET`	trade-worker	`wrangler secret put`	Optional	Bybit exchange secret
`TG_BOT_TOKEN_BINDING`	telegram-worker	`wrangler secret put`	Optional	Telegram bot token
`TG_CHAT_ID_BINDING`	telegram-worker	`wrangler secret put`	Optional	Default Telegram chat ID
`TELEGRAM_SECRET_TOKEN`	telegram-worker	`wrangler secret put`	Optional	Telegram webhook secret
`AUTHORIZED_CHAT_IDS`	telegram-worker	`wrangler secret put`	Optional	Comma-separated authorized chat IDs
`WALLET_PK_SECRET`	web3-wallet-worker	`wrangler secret put`	Optional	Wallet private key
`WALLET_MNEMONIC_SECRET`	web3-wallet-worker	`wrangler secret put`	Optional	Wallet mnemonic phrase
`EMAIL_HOST`	email-worker	`wrangler secret put`	Optional	Email IMAP host
`EMAIL_USER`	email-worker	`wrangler secret put`	Optional	Email username
`EMAIL_PASS`	email-worker	`wrangler secret put`	Optional	Email password

Secret Name	Worker(s)	Set Via	Required For	Description
`INTERNAL_KEY_BINDING`	email-worker	`wrangler secret put`	Optional	Email worker auth
`D1_INTERNAL_KEY`	d1-worker (header check)	`wrangler secret put`	Optional	D1 worker API auth
`HA_TOKEN_BINDING`	hoox	`wrangler secret put`	Optional	Home Assistant token

3.2 Environment Variables by File

`.env.local` (Project Root)

```
# === CLOUDFLARE ACCOUNT ===
CLOUDFLARE_API_TOKEN="cfut_..."
CLOUDFLARE_ACCOUNT_ID="debc6545e63bea36be059cbc82d80ec8"
CLOUDFLARE_SECRET_STORE_ID="48433bc559a943f09d9d6c622e188fd5"
SUBDOMAIN_PREFIX="hoox"

# === INTERNAL AUTH KEYS ===
D1_INTERNAL_KEY="<generate-secure-random-string>"
TRADE_INTERNAL_KEY="<generate-secure-random-string>"
AGENT_INTERNAL_KEY="<generate-secure-random-string>"

# === TELEGRAM ===
TELEGRAM_BOT_TOKEN="<your-bot-token>"

# === AI PROVIDERS (optional) ===
AGENT_OPENAI_KEY="sk-..."
AGENT_ANTHROPIC_KEY="sk-ant-..."
AGENT_GOOGLE_KEY="..."

# === EXCHANGE API KEYS (optional) ===
BINANCE_API_KEY="..."
BINANCE_API_SECRET="..."
MEXC_API_KEY="..."
MEXC_API_SECRET="..."
BYBIT_API_KEY="..."
BYBIT_API_SECRET="..."

# === DASHBOARD AUTH ===
DASHBOARD_USER="admin"
DASHBOARD_PASS="<secure-password>"
SESSION_SECRET="<32-character-secure-random-string>"
```

`workers/dashboard/.env.local` (Dashboard Local Dev)

```
DASHBOARD_USER=admin
DASHBOARD_PASS=admin
```

`workers/dashboard/.dev.vars` (Wrangler Dev Mode)

```
DASHBOARD_USER=admin
DASHBOARD_PASS=admin
```

`wrangler.jsonc` (Central Configuration)

```
{
  "global": {
    "cloudflare_api_token": "<USE_WRANGLER_SECRET_PUT>",
    "cloudflare_account_id": "debc6545e63bea36be059cbc82d80ec8",
    "cloudflare_secret_store_id": "48433bc559a943f09d9d6c622e188fd5",
    "subdomain_prefix": "hoox",
  },
  "workers": {
    "d1-worker": {
      "enabled": true,
      "path": "workers/d1-worker",
      "vars": { "database_name": "my-database" },
    },
    "telegram-worker": {
      "enabled": true,
      "path": "workers/telegram-worker",
      "vars": {},
      "secrets": ["TELEGRAM_BOT_TOKEN"],
    },
    "trade-worker": {
      "enabled": true,
      "path": "workers/trade-worker",
      "vars": {},
      "secrets": [
        "API_SERVICE_KEY",
        "BINANCE_API_KEY",
        "BINANCE_API_SECRET",
        "MEXC_API_KEY",
        "MEXC_API_SECRET",
        "BYBIT_API_KEY",
        "BYBIT_API_SECRET",
      ],
    },
    "web3-wallet-worker": {
      "enabled": true,
      "path": "workers/web3-wallet-worker",
      "vars": {},
      "secrets": ["WALLET_MNEMONIC_SECRET", "WALLET_PK_SECRET"],
    },
    "hoox": {
      "enabled": true,
      "path": "workers/hoox",
      "vars": {},
      "secrets": ["WEBHOOK_API_KEY_BINDING"],
    },
    "agent-worker": {
      "enabled": true,
      "path": "workers/agent-worker",
      "vars": {},
      "secrets": ["AGENT_INTERNAL_KEY"],
    },
    "email-worker": {
      "enabled": true,
      "path": "workers/email-worker",
      "vars": { "USE_IMAP": "false" },
      "secrets": ["EMAIL_HOST", "EMAIL_USER", "EMAIL_PASS", "INTERNAL_KEY"],
    },
    "analytics-worker": {
      "enabled": true,
      "path": "workers/analytics-worker",
```

```
      "vars": {},
      "secrets": ["CLOUDFLARE_API_TOKEN"],
    },
  },
  "dev": {
    "runtime": "native", // "native" (wrangler) or "docker" (compose) — hoox dev start preference
  },
}
```

3.3 KV Configuration Keys

These keys must be set in `CONFIG_KV` namespace:

Key	Type	Default	Set By	Used By
`webhook:tradingview:ip_check_enabled`	boolean	`false`	Manual	hoox
`webhook:allowed_ips`	string	`""`	Manual	hoox
`routing:dynamic:enabled`	boolean	`false`	Manual	hoox
`trade:max_daily_drawdown_percent`	number	`10`	Manual	agent-worker
`trade:kill_switch`	boolean	`false`	Manual	agent-worker, hoox
`trade:watermark:{exchange}:{symbol}:{side}`	number	—	agent-worker	agent-worker
`agent:openai_key`	string	—	Manual	agent-worker
`agent:anthropic_key`	string	—	Manual	agent-worker
`agent:google_key`	string	—	Manual	agent-worker
`agent:azure_api_key`	string	—	Manual	agent-worker
`agent:azure_endpoint`	string	—	Manual	agent-worker
`email:scan_subject`	string	—	Manual	email-worker
`email:coin_pattern`	string	—	Manual	email-worker
`email:action_pattern`	string	—	Manual	email-worker
`email:quantity_multiplier`	number	`1`	Manual	email-worker
`email:use_imap`	boolean	`false`	Manual	email-worker

4. Development Setup

4.1 Step 1: Clone Repository

```
# Option A: Via CLI (Recommended)
hoox clone my-hoox-app
cd my-hoox-app

# Option B: Direct git clone
git clone --recursive https://github.com/jango-blockchained/hoox-setup.git my-hoox-app
cd my-hoox-app

# If submodules are missing
git submodule update --init --recursive
```

4.2 Step 2: Verify Submodules

```
# Check all worker directories exist
bun run check:worker-submodules

# Expected directories:
# workers/hoox
# workers/trade-worker
# workers/agent-worker
# workers/d1-worker
# workers/telegram-worker
# workers/web3-wallet-worker
# workers/email-worker
# workers/analytics-worker
# workers/report-worker
```

4.3 Step 3: Install Dependencies

```
# Install all workspace dependencies
bun install

# Verify installation
bun run lint        # ESLint check
bun run typecheck   # TypeScript check
```

4.4 Step 4: Configure Local Environment

```
# Copy environment template
cp .env.example .env.local

# Edit .env.local with your values
# At minimum, set:
# - CLOUDFLARE_API_TOKEN
# - CLOUDFLARE_ACCOUNT_ID
# - SUBDOMAIN_PREFIX
```

4.5 Step 5: Authenticate Wrangler

```
# Login to Cloudflare
wrangler login

# Verify authentication
wrangler whoami
```

4.6 Step 6: Create Infrastructure (Local)

For local development, some infrastructure is optional. You need:

Required:

- D1 database (for trade data)
- KV namespace (for config)

Optional for local dev:

- R2 buckets
- Queues
- Vectorize
- Analytics Engine

```
# Create D1 database
wrangler d1 create trade-data-db

# Create KV namespace
wrangler kv:namespace create CONFIG_KV
wrangler kv:namespace create SESSIONS_KV

# Note the IDs and update wrangler.jsonc files
```

4.7 Step 7: Apply Database Schema

```
# Apply trade worker schema to D1
wrangler d1 execute trade-data-db --file=workers/trade-worker/schema.sql

# Apply tracking schema
bun run migrate:tracking
```

4.8 Step 8: Start Development

```
# Start all workers (prompts for runtime: Native or Docker)
hoox dev start

# Or force a specific runtime
hoox dev start --runtime docker   # docker compose
hoox dev start --runtime native   # wrangler dev

# Docker Compose directly (with profiles)
docker compose --profile workers up        # workers only
docker compose --profile full up           # workers + dashboard
docker compose --profile dashboard up      # dashboard only

# Or start individual workers
hoox dev worker <name> [--runtime native|docker]
hoox dev dashboard                         # dashboard only

# TUI (interactive terminal UI)
./hoox-tui
```

4.9 Step 9: Dashboard Local Dev

```
# Start dashboard dev server
hoox dev dashboard

# Or manually
cd workers/dashboard && bun run dev
```

The dashboard runs at `http://localhost:3000`.

---

5. Production Setup

5.1 Phase 1: Account & Tooling

1. Create Cloudflare account
2. Generate API Token with required permissions (see Section 2.3)
3. Install Bun, Wrangler CLI
4. Authenticate: `wrangler login`

5.2 Phase 2: Repository Setup

```
# Clone with submodules
git clone --recursive https://github.com/jango-blockchained/hoox-setup.git
cd hoox-setup

# Install dependencies
bun install

# Verify structure
bun run check:worker-submodules
```

5.3 Phase 3: Infrastructure Provisioning

See Section 6 for detailed commands.

5.4 Phase 4: Configuration

```
# Copy and edit environment
cp .env.example .env.local
# Set all required values

# Update wrangler.jsonc
# - Set your account_id
# - Set your secret_store_id
# - Set your subdomain_prefix
# - Enable/disable workers as needed
```

5.5 Phase 5: Secret Deployment

```
# Push all secrets to Cloudflare
hoox secrets update-cf

# Or set individually per worker:
wrangler secret put WEBHOOK_API_KEY_BINDING --config workers/hoox/wrangler.jsonc
wrangler secret put INTERNAL_KEY_BINDING --config workers/hoox/wrangler.jsonc
wrangler secret put AGENT_INTERNAL_KEY --config workers/agent-worker/wrangler.jsonc
# ... etc for all secrets
```

5.6 Phase 6: Database Setup

```
# Apply schema
wrangler d1 execute trade-data-db --file=workers/trade-worker/schema.sql --remote

# Apply tracking schema
bun run migrate:tracking
```

5.7 Phase 7: KV Configuration

```
# Set required KV keys
wrangler kv:key put --namespace-id=c5917667a21745e390ff969f32b1847d \
  "webhook:tradingview:ip_check_enabled" "false"

wrangler kv:key put --namespace-id=c5917667a21745e390ff969f32b1847d \
  "trade:kill_switch" "false"

wrangler kv:key put --namespace-id=c5917667a21745e390ff969f32b1847d \
  "trade:max_daily_drawdown_percent" "10"
```

5.8 Phase 8: Worker Deployment

See Section 7 for the exact sequence.

5.9 Phase 9: Dashboard Deployment

```
cd workers/dashboard

# Build with OpenNext
bun run opennext:build

# Deploy to Cloudflare Workers
bun run opennext:deploy
```

5.10 Phase 10: Verification

See Section 10 for validation procedures.

---

6. Infrastructure Provisioning

6.1 D1 Database

```
# Create database
wrangler d1 create trade-data-db

# Note the database_id from output
# Update in:
# - workers/trade-worker/wrangler.jsonc
# - workers/d1-worker/wrangler.jsonc

# Apply schema
wrangler d1 execute trade-data-db --file=workers/trade-worker/schema.sql --remote
```

Schema Tables:

- `trade_signals` — Incoming signal tracker
- `trades` — Executed trades log
- `positions` — Active & closed positions
- `balances` — Exchange balance snapshots
- `system_logs` — System observability logs

6.2 KV Namespaces

```
# Create CONFIG_KV (shared across all workers)
wrangler kv:namespace create CONFIG_KV
# ID: c5917667a21745e390ff969f32b1847d

# Create SESSIONS_KV (for hoox gateway)
wrangler kv:namespace create SESSIONS_KV
# ID: ff70a58b492e45d79880a7a8213c745c

# Update all wrangler.jsonc files with these IDs
```

6.3 R2 Buckets

```
# Create trade reports bucket
wrangler r2 bucket create trade-reports

# Create system logs bucket
wrangler r2 bucket create hoox-system-logs

# Create user uploads bucket
wrangler r2 bucket create user-uploads
```

6.4 Queue

```
# Create trade execution queue
wrangler queues create trade-execution
```

6.5 Vectorize Index

```
# Create RAG vector index
wrangler vectorize create my-rag-index --dimensions=768 --metric=cosine
```

6.6 Analytics Engine

```
# Create analytics dataset
# Via Cloudflare Dashboard: Workers & Pages > Analytics Engine
# Name: hoox-analytics
```

6.7 Durable Objects Migration

The `hoox` worker requires a Durable Object migration:

```
// Already defined in workers/hoox/wrangler.jsonc
"durable_objects": {
  "bindings": [
    {
      "name": "IDEMPOTENCY_STORE",
      "class_name": "IdempotencyStore"
    }
  ]
},
"migrations": [
  {
    "tag": "v1",
    "new_sqlite_classes": ["IdempotencyStore"]
  }
]
```

This is automatically applied on first deploy.

---

7. Worker Deployment Sequence

Deploy order matters due to service bindings. A worker must be deployed before another worker can bind to it.

7.1 Deployment Order

```
1. analytics-worker    (no dependencies)
2. report-worker       (depends on: analytics-worker)
3. d1-worker           (depends on: analytics-worker)
4. telegram-worker     (depends on: trade-worker, hoox, analytics-worker)
5. web3-wallet-worker  (depends on: telegram-worker, analytics-worker)
6. email-worker        (depends on: trade-worker, analytics-worker)
7. trade-worker        (depends on: d1-worker, telegram-worker, analytics-worker)
8. agent-worker        (depends on: d1-worker, trade-worker, telegram-worker, analytics-worker)
9. hoox                (depends on: trade-worker, telegram-worker, analytics-worker)
10. dashboard          (depends on: all services being live)
```

7.2 Deployment Commands

```
# Deploy all workers in correct order
hoox workers deploy analytics-worker
hoox workers deploy report-worker
hoox workers deploy d1-worker
hoox workers deploy telegram-worker
hoox workers deploy web3-wallet-worker
hoox workers deploy email-worker
hoox workers deploy trade-worker
hoox workers deploy agent-worker
hoox workers deploy hoox

# Or deploy all enabled workers (CLI handles order)
hoox workers deploy --all
```

7.3 Post-Deployment: Telegram Webhook

```
# Set Telegram webhook after telegram-worker is deployed
curl -X POST "https://api.telegram.org/bot<TELEGRAM_BOT_TOKEN>/setWebhook" \
  -H "Content-Type: application/json" \
  -d '{
    "url": "https://telegram-worker.<SUBDOMAIN_PREFIX>.workers.dev/webhook",
    "secret_token": "<TELEGRAM_SECRET_TOKEN>"
  }'
```

7.4 Post-Deployment: Update Internal URLs

```
# Update service URLs in dashboard wrangler.jsonc
hoox workers update-internal-urls
```

---

8. Dashboard Setup & Deployment

8.1 Configuration Files

File	Purpose
`workers/dashboard/next.config.ts`	Next.js config (OpenNext init)
`workers/dashboard/wrangler.jsonc`	Worker deployment config
`workers/dashboard/open-next.config.ts`	OpenNext adapter config
`workers/dashboard/.env.local`	Local dev credentials
`workers/dashboard/.dev.vars`	Wrangler dev credentials

8.2 Wrangler Configuration

```
// workers/dashboard/wrangler.jsonc
{
  "name": "hoox-dashboard",
  "main": ".open-next/worker.js",
  "account_id": "debc6545e63bea36be059cbc82d80ec8",
  "compatibility_date": "2026-04-17",
  "compatibility_flags": ["nodejs_compat"],
  "assets": {
    "directory": ".open-next/assets",
    "binding": "ASSETS",
  },
  "kv_namespaces": [
    {
      "binding": "CONFIG_KV",
      "id": "c5917667a21745e390ff969f32b1847d",
    },
  ],
  "vars": {
    "D1_WORKER_URL": "https://d1-worker.hoox.workers.dev",
    "AGENT_SERVICE_URL": "https://agent-worker.hoox.workers.dev",
    "TRADE_SERVICE_URL": "https://trade-worker.hoox.workers.dev",
    "TELEGRAM_SERVICE_URL": "https://telegram-worker.hoox.workers.dev",
  },
}
```

8.3 Local Development

```
cd workers/dashboard

# Install dashboard dependencies (if not done at root)
bun install

# Set credentials
cp .env.local.example .env.local
# Edit: DASHBOARD_USER, DASHBOARD_PASS

# Start dev server
bun run dev
# Access: http://localhost:3000
```

8.4 Production Build & Deploy

```
cd workers/dashboard

# Install dependencies
bun install

# Build with OpenNext
bun run opennext:build
# Output: .open-next/worker.js and .open-next/assets

# Deploy to Cloudflare Workers
bun run opennext:deploy

# Or from root:
bun run pages:deploy
```

8.5 Dashboard Environment Variables

Variable	Required	Description
`DASHBOARD_USER`	Yes	Login username
`DASHBOARD_PASS`	Yes	Login password
`SESSION_SECRET`	Yes	Cookie signing secret (32+ chars)
`AUTH_TYPE`	No	`basic`, `cf-access`, or `none`
`CF_ACCESS_TEAM_NAME`	No	CF Access team (if using cf-access)
`D1_WORKER_URL`	Yes	D1 worker service URL
`TRADE_SERVICE_URL`	Yes	Trade worker service URL
`AGENT_SERVICE_URL`	Yes	Agent worker service URL
`TELEGRAM_SERVICE_URL`	Yes	Telegram worker service URL
`D1_INTERNAL_KEY`	Yes	Auth key for D1 worker
`AGENT_INTERNAL_KEY`	Yes	Auth key for agent worker
`TELEGRAM_INTERNAL_KEY_BINDING`	No	Auth key for telegram worker
`API_SERVICE_KEY`	No	General API service key

9. Secret Management Reference

9.1 Setting Secrets via CLI

```
# Set a secret for a specific worker
wrangler secret put <SECRET_NAME> --config workers/<worker>/wrangler.jsonc
# You will be prompted to enter the value (hidden input)

# Set secret via hoox CLI
hoox secrets update-cf <SECRET_NAME> <WORKER_NAME>

# Set all secrets from wrangler.jsonc
hoox secrets update-cf
```

9.2 Local Development Secrets

For local development with `wrangler dev`, create `.dev.vars` in each worker directory:

```
# workers/hoox/.dev.vars
WEBHOOK_API_KEY_BINDING=dev-webhook-key
INTERNAL_KEY_BINDING=dev-internal-key

# workers/trade-worker/.dev.vars
INTERNAL_KEY_BINDING=dev-internal-key
MEXC_KEY_BINDING=dev-mexc-key
MEXC_SECRET_BINDING=dev-mexc-secret
# ... etc
```

9.3 Secret Security Best Practices

1. **Never commit secrets** — Use `.gitignore` for `.env.local`, `.dev.vars`, `.keys/`
2. **Use `wrangler secret put`** — Never pass secrets as CLI arguments
3. **Rotate regularly** — Exchange API keys every 90 days
4. **Use least privilege** — Create exchange API keys with minimal permissions
5. **Enable IP restrictions** — Restrict exchange API keys to Cloudflare IP ranges
6. **Monitor usage** — Review analytics-worker logs for unusual patterns

---

10. Validation & Health Checks

10.1 Automated Validation Commands

```
# Check overall setup
hoox check-setup

# Check secrets
hoox secrets list

# Check worker health
hoox check health

# Run tests
bun test
bun run tests:coverage

# Type checking
bun run typecheck
bun run build

# Lint
bun run lint
```

10.2 Manual Health Checks

10.2.1 Gateway Health

```
# Check hoox gateway
curl https://hoox.<SUBDOMAIN_PREFIX>.workers.dev/health

# Expected: {"status":"ok"}
```

10.2.2 Trade Worker Health

```
# Check trade worker
curl https://trade-worker.<SUBDOMAIN_PREFIX>.workers.dev/health

# Check signals endpoint
curl https://trade-worker.<SUBDOMAIN_PREFIX>.workers.dev/api/signals \
  -H "Authorization: Bearer <API_SERVICE_KEY>"
```

10.2.3 Agent Worker Health

```
# Check agent health
curl https://agent-worker.<SUBDOMAIN_PREFIX>.workers.dev/health

# Check status
curl https://agent-worker.<SUBDOMAIN_PREFIX>.workers.dev/status
```

10.2.4 D1 Worker Health

```
# Test D1 query
curl -X POST https://d1-worker.<SUBDOMAIN_PREFIX>.workers.dev/query \
  -H "Content-Type: application/json" \
  -H "X-Internal-Auth-Key: <D1_INTERNAL_KEY>" \
  --data '{"query":"SELECT 1"}'
```

10.2.5 Telegram Worker Health

```
# Check telegram worker
curl https://telegram-worker.<SUBDOMAIN_PREFIX>.workers.dev/health
```

10.2.6 Analytics Worker Health

```
# Track a test event
curl -X POST https://analytics-worker.<SUBDOMAIN_PREFIX>.workers.dev/track/test \
  -H "Content-Type: application/json" \
  --data '{"event":"test","value":1}'
```

10.2.7 Dashboard Health

```
# Check dashboard
curl https://hoox-dashboard.<SUBDOMAIN_PREFIX>.workers.dev/api/health
```

10.3 Database Validation

```
# List tables
wrangler d1 execute trade-data-db --command="SELECT name FROM sqlite_master WHERE type='table'" -

# Check trade_signals count
wrangler d1 execute trade-data-db --command="SELECT COUNT(*) FROM trade_signals" --remote

# Check recent logs
wrangler d1 execute trade-data-db --command="SELECT * FROM system_logs ORDER BY timestamp DESC LI
```

10.4 KV Validation

```
# List KV keys
wrangler kv:key list --namespace-id=c5917667a21745e390ff969f32b1847d

# Check kill switch
wrangler kv:key get --namespace-id=c5917667a21745e390ff969f32b1847d "trade:kill_switch"
```

10.5 Complete System Validation Checklist

- [ ] All workers deployed and returning HTTP 200 on `/health`
- [ ] D1 database has all required tables
- [ ] KV namespace has required configuration keys
- [ ] Secrets are set for all enabled workers
- [ ] Service bindings resolve correctly (no 502/503 errors)

- [ ] Queue is configured and consumer is registered
- [ ] Telegram webhook is set and responding
- [ ] Dashboard accessible and authenticated
- [ ] Analytics Engine receiving data points
- [ ] Cron triggers scheduled (agent-worker: every 5min, email-worker: every 5min)

---

11. Repair & Recovery Procedures

11.1 Complete System Repair Checklist

```
# 1. Verify repository integrity
bun run check:worker-submodules
bun run lint:scripts

# 2. Verify dependencies
bun install

# 3. Verify TypeScript
bun run typecheck

# 4. Verify tests
bun test

# 5. Verify infrastructure exists
wrangler d1 list
wrangler kv:namespace list
wrangler r2 bucket list
wrangler queues list
wrangler vectorize list

# 6. Verify secrets
hoox secrets list

# 7. Verify worker health
hoox check health

# 8. Redeploy if needed
hoox deploy workers
```

11.2 Individual Worker Repair

```
# Redeploy a single worker
hoox workers deploy <worker-name>

# Check worker logs
hoox workers logs <worker-name>

# Tail logs in real-time
wrangler tail --config workers/<worker-name>/wrangler.jsonc
```

11.3 Database Repair

```
# Reset database schema (WARNING: Destructive)
wrangler d1 execute trade-data-db --file=workers/trade-worker/schema.sql --remote

# Apply tracking schema
bun run migrate:tracking

# Check for missing tables
wrangler d1 execute trade-data-db --command="SELECT name FROM sqlite_master WHERE type='table'" -
```

11.4 Secret Repair

```
# If secrets are missing, re-upload all from wrangler.jsonc
hoox secrets update-cf

# Or set individual secrets
wrangler secret put <SECRET_NAME> --config workers/<worker>/wrangler.jsonc
```

11.5 KV Configuration Repair

```
# Reset critical KV keys
wrangler kv:key put --namespace-id=c5917667a21745e390ff969f32b1847d "trade:kill_switch" "false"
wrangler kv:key put --namespace-id=c5917667a21745e390ff969f32b1847d "webhook:tradingview:ip_check
wrangler kv:key put --namespace-id=c5917667a21745e390ff969f32b1847d "trade:max_daily_drawdown_per
```

11.6 Dashboard Repair

```
cd workers/dashboard

# Rebuild
bun run opennext:build

# Redeploy
bun run opennext:deploy

# Clear browser cache and cookies if auth issues
```

11.7 Complete Rebuild from Scratch

```
# 1. Backup any important data from D1/R2

# 2. Delete and recreate D1
wrangler d1 delete trade-data-db
wrangler d1 create trade-data-db

# 3. Delete and recreate KV (note: data loss)
# KV namespaces cannot be renamed, create new ones if needed

# 4. Redeploy all workers
hoox workers deploy --all

# 5. Re-apply schema
wrangler d1 execute trade-data-db --file=workers/trade-worker/schema.sql --remote

# 6. Reconfigure KV
# Set all required keys (see Section 3.3)

# 7. Reconfigure Telegram webhook
# (see Section 7.3)

# 8. Redeploy dashboard
cd workers/dashboard && bun run opennext:build && bun run opennext:deploy
```

---

12. Operational Runbook

12.1 Daily Operations

```
# Check system health
hoox check health

# Check recent trades
wrangler d1 execute trade-data-db --command="SELECT * FROM trades ORDER BY timestamp DESC LIMIT 5

# Check system logs
wrangler d1 execute trade-data-db --command="SELECT * FROM system_logs ORDER BY timestamp DESC LI
```

12.2 Kill Switch Operations

```
# Emergency stop all trading
wrangler kv:key put --namespace-id=c5917667a21745e390ff969f32b1847d "trade:kill_switch" "true"

# Resume trading
wrangler kv:key put --namespace-id=c5917667a21745e390ff969f32b1847d "trade:kill_switch" "false"
```

12.3 Updating Workers

```
# Pull latest code
git pull --recurse-submodules

# Update dependencies
bun install

# Run checks
bun run lint
bun run typecheck
bun test

# Deploy updated workers
hoox deploy workers

# Verify deployment
hoox check health
```

12.4 Rotating Secrets

```
# Generate new key
hoox keys generate <SECRET_NAME>

# Update in Cloudflare
hoox secrets update-cf <SECRET_NAME> <WORKER_NAME>

# Update any local .dev.vars files

# Verify functionality
bun test
```

12.5 Monitoring

Metric	How to Check
Worker errors	`wrangler tail --config workers/<worker>/wrangler.jsonc`
Trade volume	D1 `SELECT COUNT(*) FROM trades WHERE timestamp > unixepoch() - 86400`
Queue depth	Cloudflare Dashboard > Queues
Analytics	Cloudflare Dashboard > Analytics Engine
System logs	D1 `system_logs` table
Uptime	Cloudflare Dashboard > Workers

12.6 Backup Procedures

```
# Export D1 database
wrangler d1 export trade-data-db --output=backup-$(date +%Y%m%d).sql --remote

# Export KV (manual script needed)
# R2 buckets can be synced with rclone
```

---

13. Complete File Inventory

13.1 Required Configuration Files

File	Purpose	Required
`wrangler.jsonc`	Central worker configuration	**Yes**
`.env.local`	Local environment variables	**Yes**
`package.json`	Root workspace manifest	**Yes**
`bunfig.toml`	Bun test configuration	**Yes**
`tsconfig.json`	TypeScript configuration	**Yes**
`vitest.config.ts`	Integration test config	**Yes**

13.2 Worker Configuration Files

Worker	Wrangler Config	Main Entry	Schema
`hoox`	`workers/hoox/wrangler.jsonc`	`workers/hoox/src/index.ts`	—
`trade-worker`	`workers/trade-worker/wrangler.jsonc`	`workers/trade-worker/src/index.ts`	`workers/trade-worker/schema.sql`
`agent-worker`	`workers/agent-worker/wrangler.jsonc`	`workers/agent-worker/src/index.ts`	—
`d1-worker`	`workers/d1-worker/wrangler.jsonc`	`workers/d1-worker/src/index.ts`	—
`telegram-worker`	`workers/telegram-worker/wrangler.jsonc`	`workers/telegram-worker/src/index.ts`	—
`web3-wallet-worker`	`workers/web3-wallet-worker/wrangler.jsonc`	`workers/web3-wallet-worker/src/index.ts`	—
`email-worker`	`workers/email-worker/wrangler.jsonc`	`workers/email-worker/src/index.ts`	—
`analytics-worker`	`workers/analytics-worker/wrangler.jsonc`	`workers/analytics-worker/src/index.ts`	—
`report-worker`	`workers/report-worker/wrangler.jsonc`	`workers/report-worker/src/index.ts`	—

13.3 Dashboard Files

File	Purpose
`workers/dashboard/next.config.ts`	Next.js configuration
`workers/dashboard/wrangler.jsonc`	Cloudflare Workers deployment config
`workers/dashboard/open-next.config.ts`	OpenNext adapter configuration
`workers/dashboard/src/middleware.ts`	Edge middleware (auth)
`workers/dashboard/.env.local`	Local dev credentials
`workers/dashboard/.dev.vars`	Wrangler dev credentials

13.4 Package Files

Package	Main Export	Purpose
`packages/cli`	`bin/hoox.js`	CLI management tool
`packages/shared`	`src/index.ts`	Shared types, router, middleware

13.5 Script Files

Script	Purpose
`scripts/migrate-tracking.sh`	D1 tracking schema migration
`scripts/check-script-paths.ts`	Validate script paths
`scripts/check-worker-submodules.ts`	Verify worker directories exist
`scripts/purge-credentials.sh`	Git history credential purge
`hoox-tui`	Terminal UI for local dev (if exists)

13.6 Documentation Files

Document	Purpose
`docs/home.md`	Project home
`docs/getting-started/installation.md`	Installation guide
`docs/getting-started/configuration.md`	Configuration guide
`docs/deployment/production.md`	Production deployment
`docs/development/local-dev.md`	Local development
`docs/workers/*.md`	Per-worker documentation
`docs/architecture/*.md`	Architecture documentation
`openapi.yaml`	OpenAPI REST specification
`asyncapi.yaml`	AsyncAPI event specification

---

14. Troubleshooting Matrix

Symptom	Likely Cause	Solution
`bun install` fails	Missing submodules	`git submodule update --init --recursive`
`wrangler login` fails	Browser/auth issue	Try `wrangler login --browser=false`
Worker deploy fails	Missing secrets	`hoox secrets update-cf`
502 Bad Gateway	Service binding not found	Deploy dependency workers first (Section 7)
401 Unauthorized	Wrong API key	Check secret values with `wrangler secret list`
429 Too Many Requests	Rate limiting	Check KV rate limit keys; increase limits
D1 query fails	Schema not applied	Run `wrangler d1 execute trade-data-db --file=workers/trade-worker/schema.sql --remote`
Telegram not receiving	Webhook not set	Run webhook setup (Section 7.3)
Dashboard 500 error	Missing env vars	Check `.env.local` and `.dev.vars`
TypeScript errors	Missing types	`bun install` to refresh `@cloudflare/workers-types`
Tests fail	Missing test env	Check `bunfig.toml` test.env settings
Queue not processing	Consumer not bound	Check `trade-worker` wrangler.jsonc queue consumer config
Analytics missing	Dataset not created	Create `hoox-analytics` in Cloudflare Dashboard
Kill switch not working	KV key missing	Set `trade:kill_switch` in CONFIG_KV
Exchange API errors	Invalid keys	Regenerate and re-upload exchange secrets
Build fails	TypeScript errors	`bun run typecheck` to identify issues
OpenNext build fails	Missing assets	Ensure `next.config.ts` has `initOpenNextCloudflareForDev()`

Appendix A: Quick Reference Commands

```
# Setup
bun install
hoox onboard                  # One-shot full bootstrap (recommended)
hoox secrets sync             # Push .dev.vars to Cloudflare

# Development
hoox dev start                # Start all workers (choose runtime)
hoox dashboard dev            # Dashboard only
hoox tui                      # Interactive TUI
hoox workers dev <name>       # Dev single worker
bun run dev                   # Dashboard dev

# Testing
bun test                      # Unit tests
bun run tests:coverage        # Coverage
bun run test:integration      # Integration tests

# Deployment
hoox deploy workers           # Deploy all workers
hoox deploy worker <name>     # Deploy one worker
hoox dashboard deploy         # Deploy dashboard
# OR (equivalent): hoox deploy dashboard

# Operations
hoox check health             # Check worker health (single source of truth)
hoox workers logs <name>      # View worker logs
hoox check setup              # Validate setup
hoox secrets list <worker>    # Check secrets

# Database
wrangler d1 execute trade-data-db --file=workers/trade-worker/schema.sql --remote
bun run migrate:tracking

# KV
wrangler kv:key put --namespace-id=<ID> <key> <value>
wrangler kv:key list --namespace-id=<ID>

# Secrets
wrangler secret put <NAME> --config workers/<worker>/wrangler.jsonc
hoox secrets update-cf

# Health Checks
curl https://<worker>.<prefix>.workers.dev/health
```

Appendix B: Directory Structure

```
hoox-setup/
├── .opencode/                    # Project intelligence hub
├── .env.local                    # Local environment (gitignored)
├── .env.example                  # Environment template
├── wrangler.jsonc                 # Central worker config
├── package.json                  # Root workspace manifest
├── bunfig.toml                   # Bun config
├── tsconfig.json                 # TypeScript config
├── vitest.config.ts              # Vitest config
├── hoox-tui                      # TUI launcher (if exists)
│
├── packages/
│   ├── cli/                 # CLI tool
│   │   ├── bin/hoox.js           # CLI entry
│   │   └── src/
│   │       ├── index.ts          # Command dispatcher
│   │       ├── commands/         # CLI commands
│   │       ├── adapters/         # Cloudflare/Bun adapters
│   │       └── core/             # Engine, observer, types
│   └── shared/                   # Shared types/utilities
│       └── src/
│           ├── types.ts          # Core types
│           ├── router.ts         # Custom router
│           ├── middleware/       # Auth, rate-limit, logger
│           └── errors.ts         # Error factories
│
├── workers/
│   ├── hoox/                     # Gateway worker
│   ├── trade-worker/             # Trading execution
│   │   └── schema.sql            # D1 schema
│   ├── agent-worker/             # AI risk manager
│   ├── d1-worker/                # Database service
│   ├── telegram-worker/          # Telegram notifications
│   ├── web3-wallet-worker/       # DeFi operations
│   ├── email-worker/             # Email signal parsing
│   ├── analytics-worker/         # Analytics collection
│   ├── report-worker/            # PDF reports
│   └── dashboard/                # Next.js 16 dashboard
│       ├── next.config.ts
│       ├── wrangler.jsonc
│       ├── src/middleware.ts
│       └── src/app/              # Next.js app routes
│
├── scripts/
│   ├── migrate-tracking.sh       # D1 tracking migration
│   ├── check-worker-submodules.ts
│   └── purge-credentials.sh      # Emergency credential purge
│
└── docs/                         # Documentation
    ├── getting-started/
    ├── deployment/
    ├── development/
    ├── workers/
    └── architecture/
```

Document Version: 1.0.0
Last Updated: 2026-05-05
Maintainer: Hoox Development Team

CLOUDFLARE® WORKERS BINDINGS & ENVIRONMENT VARIABLES

Comprehensive reference for Cloudflare bindings (D1, R2, KV, DO, Queues, Workers AI, Vectorize, Analytics Engine) used across Hoox workers.

This document provides a comprehensive reference for all bindings, environment variables, and secrets used in the Cloudflare® Workers project.

Table of Contents

Secrets & Environment Variables

Variable	Type	Workers	Description
`INTERNAL_KEY_BINDING`	Secret	telegram-worker, hoox, trade-worker	Internal authentication key for worker-to-worker communication
`TG_BOT_TOKEN_BINDING`	Secret	telegram-worker	Telegram Bot API token
`TG_CHAT_ID_BINDING`	Secret	telegram-worker	Telegram chat ID for notifications
`TELEGRAM_SECRET_TOKEN`	Secret	telegram-worker	Webhook verification token for Telegram
`AUTHORIZED_CHAT_IDS`	Secret	telegram-worker	Comma-separated chat IDs authorized to send bot commands
`WEBHOOK_API_KEY_BINDING`	Secret	hoox	API key for webhook endpoints
`WALLET_PK_SECRET`	Secret	web3-wallet-worker	Private key for Web3 wallet operations
`WALLET_MNEMONIC_SECRET`	Secret	web3-wallet-worker	Mnemonic phrase for wallet generation
`MEXC_KEY_BINDING`	Secret	trade-worker	MEXC exchange API key
`MEXC_SECRET_BINDING`	Secret	trade-worker	MEXC exchange API secret
`BINANCE_KEY_BINDING`	Secret	trade-worker	Binance exchange API key
`BINANCE_SECRET_BINDING`	Secret	trade-worker	Binance exchange API secret
`BYBIT_KEY_BINDING`	Secret	trade-worker	Bybit exchange API key
`BYBIT_SECRET_BINDING`	Secret	trade-worker	Bybit exchange API secret
`CLOUDFLARE_API_TOKEN`	Secret	analytics-worker	CF API token for Analytics SQL queries

Service Bindings

Binding	Worker	Connected Service	Description
`TRADE_SERVICE`	hoox, agent-worker, email-worker	trade-worker	Access to trading functionality
`TELEGRAM_SERVICE`	hoox, trade-worker, agent-worker, web3-wallet-worker, report-worker	telegram-worker	Telegram notifications
`D1_SERVICE`	trade-worker, agent-worker, report-worker, dashboard	d1-worker	Centralized D1 database service
`AGENT_SERVICE`	dashboard	agent-worker	Agent worker access

KV Namespace Bindings

Binding	Worker	Description
`CONFIG_KV`	hoox, trade-worker, agent-worker, telegram-worker, d1-worker, email-worker, dashboard	Configuration + rate limiter state
`SESSIONS_KV`	hoox	Webhook session storage

R2 Bucket Bindings

Binding	Worker	Bucket Name	Description
`UPLOADS_BUCKET`	telegram-worker	user-uploads	Store user uploaded files
`REPORTS_BUCKET`	trade-worker	trade-reports	Store trading reports and data

D1 Database Bindings

Binding	Worker	Database Name	Description
`DB`	d1-worker	trade-data-db	Main database for trading data
`DB`	trade-worker	trade-data-db	Database for trade operations
`DB`	agent-worker	trade-data-db	Portfolio queries

AI Bindings

Binding	Worker	Description
`AI`	hoox, agent-worker, telegram-worker, dashboard	Access to Cloudflare® Workers AI capabilities

Vectorize Bindings

Binding	Worker	Index Name	Description
`VECTORIZE_INDEX`	hoox, telegram-worker	my-rag-index	Vector database for RAG applications

Analytics Engine Bindings

Binding	Worker	Dataset	Description
`ANALYTICS_ENGINE`	analytics-worker	hoox-analytics	Time-series metrics from all workers

Browser Rendering

Report-worker uses the Cloudflare Browser Rendering **REST API** (no wrangler binding needed):

```
POST https://api.cloudflare.com/client/v4/accounts/{id}/browser-rendering/pdf
Authorization: Bearer {CF_API_TOKEN}
Body: { html: "...", options: { format: "A4" } }
```

Local Development URLs

For local development with `wrangler dev`, the following service URLs are used:

Service	Local URL	Used By
hoox (Gateway)	http://localhost:8787	Webhook clients
Trade Worker	http://localhost:8789	hoox, agent-worker, email-worker
Telegram Worker	http://localhost:8791	hoox, trade-worker, agent-worker, web3-wallet, report
d1-worker	http://localhost:8792	trade-worker, agent-worker, report, dashboard
Web3 Wallet Worker	http://localhost:8793	trade-worker
dashboard	http://localhost:8794	Users (browser)
agent-worker	http://localhost:8795	dashboard
email-worker	http://localhost:8796	— (cron triggered)
report-worker	http://localhost:8797	— (cron triggered)

Configuration

Each worker contains a `.dev.vars` file for local development which should be populated with the appropriate values. These files should not be committed to the repository.

Example setup for `.dev.vars` files can be found in the corresponding `.dev.vars.example` files in each worker directory.

Setting Up Secrets

For production deployment, secrets should be set using the Wrangler CLI:

```
wrangler secret put SECRET_NAME
```

This will securely store the secret and make it available to the worker at runtime.

---

05 · ARCHITECTURE

SYSTEM TOPOLOGY & OVERVIEW

High-level architectural blueprint of the Hoox trading ecosystem, detailing edge microservice layouts, and multi-layered security models.

Hoox is an enterprise-grade, serverless algorithmic trading platform built entirely on **Cloudflare's Edge V8 isolates** and globally distributed resources. By using a modular, service-oriented architecture, Hoox decomposes complex trading processes into ten highly specialized micro-workers.

These workers communicate privately in microseconds, auto-scale globally near exchange servers, and store transaction logs in localized databases—all while running within Cloudflare's $0 free tiers.

High-Level System Architecture

The ecosystem splits public-facing ingress points from private internal compute layers:

```
graph TB
    %% ── Styling ──────────────────────────────────────────────────
    classDef external fill:#f5f5f5,stroke:#999,stroke-width:2,color:#333
    classDef waf fill:#fff8e1,stroke:#f9a825,stroke-width:2
    classDef worker fill:#e8f5e9,stroke:#43a047,stroke-width:2,color:#1b5e20
    classDef storage fill:#e3f2fd,stroke:#1e88e5,stroke-width:2,color:#0d47a1
    classDef compute fill:#f3e5f5,stroke:#8e24aa,stroke-width:2,color:#4a148c
    classDef dash fill:#fff3e0,stroke:#ef6c00,stroke-width:2,color:#bf360c

    %% ── Ingress Layer ────────────────────────────────────────────
    subgraph Ingress["Public Ingress Layer"]
        TV["TradingView Webhooks"]:::external
        TG["Telegram Bot Commands"]:::external
        EM["Email Signal Senders"]:::external
        WAF["Cloudflare WAF / Firewall"]:::waf
        GW["hoox Gateway Isolate"]:::worker
    end

    %% ── Private Compute Layer ────────────────────────────────────
    subgraph Compute["Private Internal Edge Compute"]
        TW["trade-worker (Execution Engine)"]:::worker
        D1W["d1-worker (SQL Hub)"]:::worker
        AW["agent-worker (AI Risk Manager)"]:::worker
        TGW["telegram-worker (Notifications)"]:::worker
        EMW["email-worker (Email Parser)"]:::worker
        W3W["web3-wallet (DeFi Swaps)"]:::worker
        ANW["analytics-worker (observability)"]:::worker
        RPW["report-worker (PDF Generator)"]:::worker
    end

    %% ── Storage & Resource Layer ─────────────────────────────────
    subgraph Storage["Persistent Edge Storage"]
        KV[("KV Config Namespace")]:::storage
        DB[("D1 SQLite Database")]:::storage
        R2[("R2 Logs & PDF Bucket")]:::storage
        VEC[("Vectorize RAG Index")]:::storage
        DO{{" Durable Objects mutex"}}:::compute
        Q{{" Queues Queue"}}:::compute
        BR{{" Browser Rendering Chrome"}}:::compute
    end

    %% ── Flow Connections ─────────────────────────────────────────
    TV --> WAF
    WAF --> GW
    EM --> EMW
    TG --> TGW

    GW -->|Service Binding| TW
    GW -->|Service Binding| TGW
    GW -->|Service Binding| ANW
    GW -->|Durable Object Lock| DO
    GW -->|Queue Failover| Q

    TW -->|Service Binding| D1W
    TW -->|Service Binding| TGW
    TW -->|Service Binding| ANW
    TW -->|DeFi Execution| W3W
    TW -.->|Config Read| KV
    TW -.->|Write Trade Logs| R2
```

```
    D1W -.->|SQLite queries| DB
    TGW -.->|Semantic RAG search| VEC
    TGW -->|Service Binding| ANW

    AW -->|Cron 5m / Position Scale| TW
    AW -->|Service Binding| TGW
    AW -->|Service Binding| D1W

    RPW -->|Cron 2x/day / Render PDFs| BR
    RPW -.->|Save Reports| R2
    RPW -->|Push PDF Link| TGW
```

Comprehensive Micro-Worker Catalog

Worker Name	Runtime Scope	Cron Trigger	Public Routing	Smart Placement	Primary Observability
`hoox`	Gateway Router	No	**Yes** (`/webhook`)	**Yes** (Fast path)	Time-series Telemetry
`trade-worker`	Order Execution	No	No (Isolated)	**Yes** (Exchange Proxied)	Execution Logs
`agent-worker`	Risk Management	Cron `*/5`	No (Isolated)	**Yes** (Account Auditing)	Alert Logs
`telegram-worker`	Alerts & Chat	No	No (Isolated)	**Yes** (Telegram APIs)	Command Logs
`d1-worker`	SQLite Manager	No	No (Isolated)	**Yes** (SQLite Bound)	Query Latency
`report-worker`	Puppeteer PDF	Cron `06,18`	No (Isolated)	**Yes** (Rendering APIs)	Print Status
`email-worker`	IMAP Parsing	Cron `*/5`	No (Isolated)	No	Parse Statistics
`web3-wallet`	DeFi Swap Engine	No	No (Isolated)	No	Tx Sign Logs
`analytics-worker`	Observability	No	No (Isolated)	No	Metrics Dataset

The 5-Layer Security Architecture

Security is designed as concentric protective corridors:

```
[ WAF: IP Range Allow-list ] -> [ Gateway: Webhook Passkey ] -> [ Isolation: Service Bindings ] -
```

Layer 1: Edge-Level Firewall & WAF

Cloudflare's global WAF drop connections immediately at the edge if:

- The payload does not originate from verified TradingView webhook IP ranges.
- The request rate exceeds threshold ceilings (10 requests/minute).

Layer 2: Webhook Passkey Authentication

The `hoox` gateway validates that the payload `apiKey` string exactly matches the encrypted `webhooks:api_key` stored inside your `CONFIG_KV` namespace. Mismatched signals are instantly dropped with a `401 Unauthorized` response.

Layer 3: Service Binding Encrypted Isolation

Internal workers (`trade-worker`, `d1-worker`, `agent-worker`) expose **zero** public HTTP endpoints. They cannot be targeted or accessed from the public internet. They can only be invoked internally by other V8 isolates using Cloudflare **Service Bindings**.

Layer 4: Standardized Internal Authorization

To prevent internal bypass or privilege escalation, all internal microservice boundaries enforce a strict **bearer authorization check**:

- All internal workers (`hoox`, `trade-worker`, `d1-worker`, `agent-worker`, `telegram-worker`) are bound to the same `INTERNAL_KEY_BINDING` secret.
- Every service-to-service invocation is audited by the shared `requireInternalAuth` middleware from `@jango-blockchained/hoox-shared/middleware`, dropping unauthorized calls.

Layer 5: Durable Object Idempotency Locks

If the network drops after an order fill, TradingView will resend the webhook. The gateway uses a single-threaded **Durable Object** to lock the request trace ID. If the transaction ID has already been logged, the duplicate is dropped before hitting exchange APIs, preventing double-ordering.

Smart Placement is enabled across all critical execution paths. This ensures that even though your webhook might hit a Cloudflare edge node in London, the actual transaction logic automatically shifts to Frankfurt or Tokyo (wherever the exchange APIs reside), eliminating network slippage entirely.

Codebase Dependency Graph

Hoox ships with an automated **function-level dependency graph extractor** that maps every exported symbol, import, call, type reference, and service binding across all 917 source files and 14 workspaces.

Generating the Graph

```
# From repository root
bun run graph
```

This runs `scripts/extract-graph.ts` (powered by **ts-morph**) and scans 917 source files across all 14 workspaces in **~20–25s**.

During extraction, an **8-phase live progress bar** reports each stage with timing:

```
   [██▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒] 13%  exports: 1667 nodes            10.6s
   [██████████▒▒▒▒▒▒▒▒▒▒] 50%  type refs: 2536 edges            1.7s
   [████████████▒▒▒▒▒▒▒▒] 63%  calls: 445 edges                 9.7s
   [████████████████████] 100% Done                             22.0s

⏱  Total time: 22.0s
```

File	Size	Format	Purpose
`graph.json`	2.5 MB	JSON (997 nodes, 5536 edges)	Machine-readable — AI context, programmatic queries
`graph.dot`	1.3 MB	Graphviz DOT (14 clusters)	Visual rendering — workspace-clustered architecture diagram

Node Types & Color Coding (DOT)

Shape	Kind	Example
`box`	Function / Const	`executeTrade` , `CONFIG_KV`
`note`	Interface / Type	`TradeRequest` , `Env`
`component`	Class	`IdempotencyDO`
`Mrecord`	Enum	`ExitCode` , `OrderStatus`
Penwidth=2	Entry Point	`main` , `export default`

Entry points (top-level workspace exports) are highlighted with double borders.

Edge Color Legend

Color	Edge Kind	Count
Blue `#4A90D9`	Imports	2,524
Orange `#FF9800`	Type references	2,536
Pink `#E91E63`	Cross-file calls	445
Green `#4CAF50`	Extends	7
Light Green `#8BC34A`	Implements	2
Purple `#673AB7`	Service bindings	11
Cyan `#00BCD4`	Workspace deps	11

Rendering to SVG

```
# Requires Graphviz installed
dot -Tsvg graph.dot -o graph.svg

# Or paste graph.dot into edotor.net / viz-js.com
```

Key Extraction Rules

- **Export-only nodes**: Only exported symbols appear in the graph (684 unexported symbols filtered out).
- **Call edge fallback**: Method calls like `router.post` resolve via TypeChecker; when the symbol name (`post`) doesn't match a top-level export, the edge falls back to the target file's first container node.
- **Self-reference filtering**: Calls within the same file are excluded.
- **Deduplication**: Duplicate edges are collapsed into a single entry.
- **Service bindings**: Parsed from each worker's `wrangler.jsonc` configuration.

Next Steps

- **Worker Communication Specifications** — Deep dive into service bindings, zero-TCP routing, and V8 engines.
- **Data Flow Maps** — Step-by-step sequence charts of trade executions and cron risk evaluations.
- **Codebase Dependency Graph** — Generate a live architecture map of all exports, calls, and bindings.

Enterprise / Institutional Scale

For the full bleeding-edge Enterprise architecture (Workers for Platforms multi-tenancy, Workflows, Logpush audit, AI Gateway, Bot Management, etc.) — the commercial layer on top of the open core — see:

- `docs/enterprise/architecture.mdx`
- `docs/enterprise/multi-tenancy.mdx`
- `docs/enterprise/observability-audit.mdx`
- `docs/enterprise/security-compliance.mdx`

See also root `OPEN_CORE.md` and `OPEN_CORE_FEATURE_SPLIT.md`.

These build directly on the current retail architecture while unlocking Cloudflare Enterprise features and 2025-2026 primitives.

06 · ARCHITECTURE

SYSTEM DATA ROUTING SPEC

Comprehensive data routing, transaction sequence diagrams, time-series metrics pipelines, and global storage mapping.

Hoox operates as a highly orchestrated **distributed event loop**. Because execution logic is split into isolated compute nodes, data flows recursively through multiple V8 transitions, asynchronous queues, time-series datasets, and database ledgers.

This document provides complete, low-level technical specifications and Mermaid sequence diagrams for our four primary data routing pipelines: Webhook Trade execution, AI Risk management, PDF browser rendering, and Observability tracking.

1. Webhook to Trade Execution Flow (High-Speed Path)

This is the primary transaction pipeline. When a trade signal is received, the system validates the payload, locks the trace ID, executes the order at the edge closest to the exchange, records the fill, and alerts the user.

```
sequenceDiagram
    autonumber
    actor TV as Signal Source (TradingView)
    participant GW as hoox Gateway
    participant DO as IdempotencyStore (Durable Object)
    participant TW as trade-worker
    participant D1 as d1-worker
    participant TG as telegram-worker
    participant AE as analytics-worker

    TV->>GW: POST /webhook (Ingest raw JSON signal)
    Note over GW: Extract Trace ID & check KV Rate Limiter
    GW->>DO: lockTransaction(traceId) (SQLite atomic mutex check)
    alt Mutex Lock Denied (Trace ID Already Logged)
        DO-->>GW: Fail (Conflict: Duplicate Request)
        GW-->>TV: 409 Conflict (Duplicate Request)
    else Mutex Lock Granted (Unique Request)
        DO->>GW: Lock Acquired (Success)
        GW->>TW: Service Binding invoke: POST /webhook (X-Internal-Auth-Key)
        Note over TW: Fetch encrypted Secrets & calculate HMAC-SHA256 signature
        TW->>D1: Service Binding invoke: POST /query (Write "TRADE_PENDING" row)
        TW->>TW: Send Order to Centralized Exchange API
        Note over TW: Exchange fills order & returns status details
        TW->>D1: Service Binding invoke: POST /query (Update Status to "Filled")
        TW->>TG: Service Binding invoke: POST /alert (Format alert template)
        TG-->>TV: Dispatch Telegram Push Alert to User's phone
        TW-->>GW: Return Executed Order JSON
        GW->>AE: Service Binding invoke: POST /track (Log latency & status metrics)
        GW-->>TV: 200 OK (Filled order metadata)
    end
```

2. Autonomous AI Risk Monitoring Flow (Cron Cycle)

Running on a strict **5-minute Cron schedule**, the risk management loop queries SQLite records, audits active exposures, calculates trailing stop deviations, and manages emergency halts.

```
sequenceDiagram
    autonumber
    participant Cron as Cloudflare Cron Trigger
    participant AW as agent-worker (AI Risk Manager)
    participant D1 as d1-worker (SQL Hub)
    participant TW as trade-worker (Execution)
    participant TG as telegram-worker (Alerts)

    Cron->>AW: Fire trigger: */5 * * * *
    AW->>D1: Service Binding invoke: POST /query (Get open margin positions)
    D1-->>AW: Return position matrices (Average price, leverage, unrealized P&L)
    Note over AW: Evaluate Max Daily Drawdown threshold against balance
    alt Drawdown Limit Hit (USDT Daily loss > max_daily_drawdown_percent)
        Note over AW: EMERGENCY HALT TRIGGERED
        AW->>AW: Set trade:kill_switch = true in CONFIG_KV
        AW->>TW: Service Binding invoke: POST /close-all (Flatten all active exposure)
        TW-->>AW: All positions flattened successfully
        AW->>TG: Service Binding invoke: POST /alert (Dispatch Emergency Alert)
        TG-->>AW: User notified on phone
    else Normal Operation (Within Safe Margins)
        Note over AW: Calculate Trailing Stop-Loss price deviations
        alt Deviation Triggered (Price crossed dynamic stop-loss boundary)
            AW->>TW: Service Binding invoke: POST /order (Close target position)
            TW-->>AW: Order filled
            AW->>TG: Service Binding invoke: POST /alert (Stop-Loss Triggered Notification)
        end
    end
```

---

3. PDF Portfolio Report Rendering Flow

Runs twice daily to automate HTML dashboard rendering, compile PDFs via Puppeteer on the edge, offload to R2 storage, and dispatch download corridors.

```
sequenceDiagram
    autonumber
    participant Cron as Cloudflare Cron Trigger
    participant RP as report-worker
    participant BR as Browser Rendering API (Chrome Isolate)
    participant R2 as R2 Object Storage Bucket
    participant TG as telegram-worker

    Cron->>RP: Fire trigger: 06:00 & 18:00 UTC
    RP->>RP: Query D1 P&L history & compile HTML5 template
    RP->>BR: POST /browser-rendering/pdf (HTML string payload)
    Note over BR: Spin up headless Puppeteer Chrome & render viewport
    BR-->>RP: Return compiled PDF Buffer stream
    RP->>R2: PutObject: reports/daily-report-{timestamp}.pdf
    RP->>TG: Service Binding invoke: POST /alert (P&L report link)
    TG-->>RP: Dispatch report download link to user
```

4. Observability & Time-Series Analytics Flow

To maintain complete cross-worker telemetry without blocking critical order threads, Hoox routes analytics data points asynchronously to a dedicated metrics warehouse.

```
sequenceDiagram
    autonumber
    participant W as Any Compute Worker (hoox, trade, agent, etc.)
    participant AN as analytics-worker (SQL Analytics Ingest)
    participant AE as Cloudflare Analytics Engine

    W->>AN: Service Binding invoke: POST /track/api-call (Trace ID, latencyMs, success)
    Note over AN: Sanitize metrics inputs & format structural time-series logs
    AN->>AE: writeDataPoint({ blobs: [worker, endpoint], doubles: [latencyMs] })
    Note over AE: Persistent logging in time-series warehouse for Next.js dashboard
```

---

5. Global Data Persistence Mapping

Storage Platform	Namespace / Database Name	Data Payload Details	Associated Compute Workers
D1 Database	`trade-data-db` (SQLite)	Executed fills, open position matrices, Drizzle tracking logs.	`d1-worker`, `trade-worker`, `agent-worker`
CONFIG_KV	`CONFIG_KV` (Key-Value)	16-key global runtime manifest, emergency Kill Switch.	**All Workers** + Next.js Dashboard
SESSIONS_KV	`SESSIONS_KV` (Key-Value)	Session access states and API authorization cookies.	`hoox` Gateway
R2 Storage	`trade-reports` (S3 Bucket)	Compiled PDF portfolio reports.	`report-worker`
R2 Storage	`hoox-system-logs` (S3 Bucket)	Verbose JSON exchange API payloads (REST & WebSocket logs).	`trade-worker`
Vectorize	`my-rag-index` (Vector DB)	Semantic chat and history vector embeddings.	`telegram-worker`

Next Steps

- **Bindings Catalog** — Check wrangler settings and resource declarations.
- **Storage Engineering Manual** — Dive into Drizzle database schemas and SQLite properties.

07 · ARCHITECTURE

ISOLATE COMMUNICATION SPEC

Detailed specification of Cloudflare Service Bindings, zero-overhead V8 isolate routing, and standard cross-worker auth middleware.

In a traditional server-based monorepo or Docker cluster, microservices communicate over TCP/IP connections using protocols like REST, gRPC, or WebSockets. These introduce significant **networking overhead**: DNS resolution, TCP handshakes, TLS negotiation, and data serialization.

Hoox completely bypasses the networking stack by leveraging Cloudflare's **Service Bindings**. This document details the low-level V8 routing mechanics, internal authentication protocols, and diagnostic mocking configurations of the Hoox communication layer.

---

1. Service Bindings: Zero-Overhead V8 Routing

Cloudflare Service Bindings allow one edge worker to call another **without ever hitting the public internet**.

The Under-the-Hood V8 Mechanics

- **Direct Execution**: When `hoox` calls `env.TRADE_SERVICE.fetch()`, the Cloudflare runtime does not construct a TCP packet or route it through a virtual network. Instead, the runtime **spawns the target worker's V8 isolate in the same physical memory thread** and executes its entry point function directly.
- **Zero Serialization Overhead**: Payloads are passed directly as active V8 memory pointers, cutting JSON serialization and parsing costs.
- **Latency Guarantee**: Internal isolate transitions are completed in **under 1 microsecond**, making microservice communication practically instant.

```
// Declarative bindings inside workers/hoox/wrangler.jsonc
{
  "services": [
    { "binding": "TRADE_SERVICE", "service": "trade-worker" },
    { "binding": "TELEGRAM_SERVICE", "service": "telegram-worker" },
  ],
}
```

---

2. Complete Service Invocations Implementation

Below is the standard, production-grade template used to route authenticated, structured HTTP payloads between workers:

```
  TRADE_SERVICE: Fetcher; // Service Binding Fetcher
  INTERNAL_KEY_BINDING: string; // Authorized Internal Key secret
}

  async fetch(request: Request, env: Env): Promise<Response> {
    // 1. Build and validate payload
    const payload = {
      exchange: "bybit",
      action: "LONG",
      symbol: "BTCUSDT",
      quantity: 0.002,
    };

    // 2. Invoke the internal trade-worker V8 isolate
    try {
      const tradeResponse = await env.TRADE_SERVICE.fetch(
        "https://trade-worker/webhook",
        {
          method: "POST",
          headers: {
            "Content-Type": "application/json",
            "X-Internal-Auth-Key": env.INTERNAL_KEY_BINDING, // Bearer Auth
          },
          body: JSON.stringify(payload),
        }
      );

      if (!tradeResponse.ok) {
        throw new Error(
          `Internal binding returned HTTP status: ${tradeResponse.status}`
        );
      }

      const result = await tradeResponse.json();
      return new Response(JSON.stringify({ success: true, result }), {
        status: 200,
        headers: { "Content-Type": "application/json" },
      });
    } catch (error: any) {
      return new Response(
        JSON.stringify({ success: false, error: error.message }),
        {
          status: 502, // Bad Gateway
          headers: { "Content-Type": "application/json" },
        }
      );
    }
  },
};
```

---

3. Internal Authorization Middleware Standard

To prevent unauthorized, cross-tenant, or direct internal calls, every internal endpoint is secured using the shared `requireInternalAuth` middleware from `@jango-blockchained/hoox-shared/middleware`:

```
// Inside target worker's router or fetch handler:

  request: Request,
  env: Env
): Promise<Response> {
  // Returns 401 Unauthorized Response if the header is invalid or missing
  const authError = requireInternalAuth(request, env, "INTERNAL_KEY_BINDING");
  if (authError) return authError;

  // Key is valid — continue execution
  return new Response("Authorized", { status: 200 });
}
```

Every single internal worker (`hoox` , `trade-worker` , `d1-worker` , `agent-worker` , `telegram-worker` , `email-worker` , `analytics-worker` , `report-worker` , `web3-wallet-worker` , `dashboard`) binds the exact same secret name: `INTERNAL_KEY_BINDING` . This eliminates variable footprint drift and simplifies secret deployments across your workspace.

---

4. Testing & Mocking Service Bindings

During local testing (via native `bun test`), you can mock the Service Binding `Fetcher` object cleanly to run full-coverage unit tests without provisioning real Cloudflare APIs:

```
test("Should mock internal trade-worker service bindings", async () => {
  const mockEnv = {
    INTERNAL_KEY_BINDING: "secret_local_test_key",
    TRADE_SERVICE: {
      fetch: async (url: string, init?: RequestInit) => {
        // Confirm headers are present and valid
        const headers = init?.headers as Record<string, string>;
        if (headers["X-Internal-Auth-Key"] !== "secret_local_test_key") {
          return new Response(JSON.stringify({ success: false }), {
            status: 401,
          });
        }

        return new Response(
          JSON.stringify({
            success: true,
            orderId: "mock_order_10482",
          }),
          { status: 200 }
        );
      },
    } as Fetcher,
  };

  // Run call test assertions
  const res = await mockEnv.TRADE_SERVICE.fetch(
    "https://trade-worker/webhook",
    {
      headers: { "X-Internal-Auth-Key": "secret_local_test_key" },
    }
  );

  const data = await res.json();
  expect(res.status).toBe(200);
  expect(data.orderId).toBe("mock_order_10482");
});
```

Next Steps

- **Data Flow Mapping** — Step-by-step sequence charts of trade executions, backups, and metrics.
- **Bindings Catalog** — Review complete environment namespaces and bound objects.

08 · ARCHITECTURE

SEE /DOCS/DEVOPS/BINDINGS

Stub — the canonical Cloudflare Workers bindings registry lives at /docs/devops/bindings.

> **This document is now a stub. For the complete bindings registry, see `/docs/devops/bindings`.**

In Cloudflare's serverless architecture, **bindings** represent the declarative bridges linking your isolate compute logic to other internal microservices and storage platforms. This page intentionally keeps only the high-level architectural framing; the canonical registry — every Service Binding, KV, R2, D1, Workers AI, Vectorize, Queue, Durable Object, and Browser Rendering entry — lives in **`docs/devops/bindings`** and is the single source of truth.

> **Tip:** Adding new bindings to your workers? Always update your `wrangler.jsonc` manifest at the workspace root, and then execute `hoox deploy update-internal-urls` to sync bindings and URLs globally!

Next Steps

- **Canonical Bindings Reference** — Complete, up-to-date bindings table (service bindings, KV, R2, D1, AI, Vectorize, queues, DOs).
- **Storage & SQLite DDL** — Drizzle schemas, R2 bucket configurations, and database rules.
- **Production Deployments** — How Wrangler compiles and maps these bindings to the live edge.

09 · ARCHITECTURE

STORAGE & DATA ENGINEERING SPEC

High-integrity architecture specifications for edge SQLite D1 databases, zero-egress R2 object stores, and eventually consistent KV cache namespaces.

Hoox operates a **multi-tier edge storage topology** to balance high-speed read/write performance, relational transactional safety, and long-term analytical capacity. By separating high-frequency dynamic states from static parameters and verbose event logs, Hoox ensures that latency remains in the single-digit milliseconds while keeping storage costs at $0/month.

---

1. The Multi-Tier Storage Architecture

Hoox segregates data into four distinct edge storage tiers based on consistency, read/write latency, and size constraints:

```
graph TD
    Pipeline["Incoming Request Pipeline"] -->|Route| DO["Durable Objects

Strong Consistency

(Idempotency Locks)"]
    Pipeline -->|Route| KV["CONFIG_KV

Eventual Consistency

(sub-ms reads, 10s write)"]
    Pipeline -->|Route| D1["D1 SQLite

Relational ACID

(5ms SQL queries)"]

    DO -->|Persist / Log| R2["R2 Object Storage

Zero-Egress Bucket

(Heavy JSON Fills / PDFs)"]
    KV -->|Persist / Log| R2
    D1 -->|Persist / Log| R2

    style Pipeline fill:#CCCCCC,stroke:#3b82f6,stroke-width:2
    style DO fill:#CCCCCC,stroke:#f59e0b,stroke-width:2
    style KV fill:#CCCCCC,stroke:#10b981,stroke-width:2
    style D1 fill:#CCCCCC,stroke:#ef4444,stroke-width:2
    style R2 fill:#CCCCCC,stroke:#a855f7,stroke-width:2
```

Storage primitive	Engine Technology	Consistency Model	Read Latency	Write Latency	Ideal Use Case
Durable Objects	In-Memory Mutex	Strong Consistency	**< 2ms**	**< 5ms**	Atomic transaction locks, high-frequency concurrency dedup.
Workers KV	Distributed Cache	Eventual Consistency	**< 1ms**	**< 10s**	Dynamic exchange routing paths, emergency Kill Switch.
D1 Database	Edge SQLite Isolate	Relational ACID	**< 5ms**	**< 12ms**	Fills ledger, open positions matrix, margin balance records.
R2 Storage	S3 Object Bucket	Strong (per-object)	**< 35ms**	**< 50ms**	Verbose exchange API request/response JSONs, PDF reports.

---

2. D1 Relational Engine & Drizzle Schema Specs

D1 runs a serverless SQLite engine embedded in the Cloudflare V8 worker isolate thread. This eliminates TCP handshake overhead when executing SQL queries.

Drizzle ORM Schema Standards

Hoox developers use Drizzle ORM to define strict type safety for D1 SQLite tables. Below is the active specification for our `trades` and `positions` schemas:

```
// 1. Transactional Trades Ledger Schema

  id: text("id").primaryKey(), // UUIDv4
  requestId: text("request_id").notNull(), // Distributed Trace ID
  exchange: text("exchange").notNull(), // "bybit" | "binance" | "mexc"
  symbol: text("symbol").notNull(), // Uppercase symbol: "BTCUSDT"
  action: text("action").notNull(), // "LONG" | "SHORT" | "CLOSE"
  side: text("side").notNull(), // "BUY" | "SELL"
  quantity: real("quantity").notNull(), // Executed contract quantity
  price: real("price").notNull(), // Execution fill price
  fee: real("fee").notNull(), // Exchange transaction fee paid in quote
  orderId: text("order_id").notNull(), // Exchange-provided order identifier
  status: text("status").notNull(), // "Filled" | "Failed"
  createdAt: integer("created_at", { mode: "timestamp" }).default(new Date()),
});

// 2. Real-Time Open Position Matrix Schema

  symbol: text("symbol").primaryKey(),
  exchange: text("exchange").notNull(),
  side: text("side").notNull(), // "LONG" | "SHORT"
  size: real("size").notNull(), // Current contract size
  entryPrice: real("entry_price").notNull(), // Volume-weighted average entry price
  leverage: integer("leverage").default(1),
  updatedAt: integer("updated_at", { mode: "timestamp" }).default(new Date()),
});
```

---

3. R2 Log Offloading: SQLite Conservation Strategy

SQLite engines are single-writer databases. If write concurrency is too high, transaction queues can lock the thread. To conserve D1 write limits and maintain high performance during high-frequency events:

1. **D1 Relational Logs**: Only high-value financial transactions (the structured fields in the `trades` table above) are written to D1.
2. **R2 Verbose Blobs**: Full, verbose JSON payload exchanges, network headers, and WebSocket stream records are serialized and saved to **R2 Storage** using date-based key paths: `logs/bybit/BTCUSDT/2026-05-19/order-18049284739.json`

This offloading strategy reduces D1 SQLite write volumes by **up to 75%**, keeping your database compact, fast, and fully within free-tier limits.

---

4. KV Eventual Consistency & Cache Performance

Cloudflare KV is a highly distributed key-value store optimized for high-read/low-write operations:

- **Eventual Consistency**: When you toggle a setting in KV (e.g. `trade:kill_switch = true`), the change is instantly cached at your local gateway node. However, it takes up to **10 seconds** to propagate to Cloudflare's other 330+ locations globally.
- **Operational Rule**: KV is perfect for global configurations, allowlists, and emergency kill switches. It is **never** used to track highly dynamic real-time states like position sizes, account drawdowns, or margin balances. High-frequency states must always be routed through D1 or Durable Objects.

Never attempt to run raw migration files directly on your production D1 database without first running a backup. Ensure your database status is fully tracked and synchronized using Drizzle migrations: `hoox db migrate --remote`.

Next Steps

- **Internal Endpoints Mapping** — Map out exposed ports, URLs, and service bindings.
- **Wrangler Setup & Tooling** — Configure Wrangler to bind local D1 and KV instances for dev testing.

10 · ARCHITECTURE

INTERNAL ENDPOINTS MAP

Architectural routing flow and service-binding topology for the Hoox microservice mesh, with the canonical endpoint directory living under the API reference.

> **This document is now a stub focused on the routing flow. For complete endpoint specifications, see `/docs/devops/api/endpoints`.**

This page captures the **architectural context** for routing across the Hoox microservice monorepo — the Mermaid routing diagram, the internal-auth header contract, and the request-trace propagation story. The exhaustive endpoint directory (per-worker route tables, JSON payloads, and success-response examples) lives in the canonical reference.

Interactive Compute & Routing Flow

All external webhooks flow through the public `hoox` gateway, which authenticates payloads and routes them to private workers inside localized V8 engine isolates:

```
graph TD
    %% Ingress
    Client[TradingView / User] -->|1. Public HTTPS Ingress| GW[hoox Gateway]

    %% Gateway Service Bindings
    GW -->|2. Service Binding| TW[trade-worker]
    GW -->|2. Service Binding| TGW[telegram-worker]
    GW -->|2. Async Failover Queue| Q[trade-execution Queue]

    %% Internal Workers Services
    TW -->|3. Service Binding| D1W[d1-worker]
    TW -->|3. Service Binding| W3[web3-wallet-worker]
    TGW -.->|RAG Queries| VEC[(Vectorize Index)]
    Q -->|4. Async Consumer| TW
    D1W -.->|SQLite read/write| DB[(D1 Database)]
```

Internal Authorization Contract

Every internal HTTP request between V8 isolates **must** transmit the standard bearer header:

```
X-Internal-Auth-Key: <INTERNAL_KEY_BINDING>
```

This shared secret is provisioned per-worker as an encrypted `wrangler secret` and validated by the `requireInternalAuth` middleware from `@jango-blockchained/hoox-shared/middleware`. Public webhook and dashboard endpoints use payload-embedded API keys or session cookies instead — see the canonical `/docs/devops/api/endpoints` for the full auth matrix.

Request-Trace Propagation

Every internal-to-internal transaction automatically inherits the `requestId` trace UUID generated by the gateway. This trace ID is attached as the `X-Request-Id` header, allowing you to trace a single webhook alert across D1 database writes, R2 log outputs, and Telegram alerts instantly!

Next Steps

- **Complete API Reference** — The canonical endpoint directory with full request/response examples for every worker.
- **Service Binding Mesh** — Deep dive into the Zero-Trust service-binding topology and the public/internal isolation boundary.
- **System Storage Architecture** — Review R2 log offloading, D1 SQLite ledger layout, and Vectorize RAG indexes.

11 · ARCHITECTURE

VISUAL TOKENS & DESIGN SYSTEM

Production design system and visual token specifications, covering OKLCH color palettes, typography, and visual SVG icon maps.

To maintain visual cohesion across the entire Hoox ecosystem (web landing pages, Astro documentation sites, Zustand Next.js dashboards, and terminal UIs), the platform implements a standardized, high-integrity design system.

This document catalogs our color tokens, spacing densities, border-radius behaviors, and provides the complete **SVG Icon Mapping Catalog** used inside the documentation navigation chrome.

---

1. OKLCH Color Palette

Hoox utilizes OKLCH color values to ensure perceptually uniform brightness and high contrast across all edge screens and operating systems:

Dark Mode (Primary Visual Standard)

- **Background (`--background`)**: `oklch(0.08 0 0)` — Deep charcoal/black.
- **Foreground (`--foreground`)**: `oklch(0.95 0 0)` — Bright neutral Zinc/white.
- **Card (`--card`)**: `oklch(0.12 0 0)` — Slightly elevated dark panel grey.
- **Accent (`--accent`)**: `oklch(0.70 0.20 45)` — High-contrast bright orange (used for focal actions and headers).
- **Muted Foreground (`--muted-foreground`)**: `oklch(0.68 0 0)` — Secondary readability text.
- **Borders (`--border`)**: `oklch(0.30 0 0)` — Low-contrast dark grey dividers.

---

2. Layout, Borders & Density

- **Border Radius (`--radius`)**: Restored globally to `0.5rem` (8px) to soften panel boundaries without losing the clean command-center aesthetic.
- **Shadows**: Cards and active modal prompts bind a highly contrasting deep shadow: `shadow-2xl` -> `box-shadow: 0 25px 50px -12px rgba(0, 0, 0, 0.5);`
- **Spacing Density**: Standardize grid gutters using Tailwind flex/grid spaces:
  - Cards Grid: `gap-6` (24px) for dashboard cards.
  - Sidebar Items: `gap-0.5` (2px) vertical padding between nav links.

---

3. Typography & Runtimes

Aesthetic Layer	Font Family	CSS Utility	Ideal Operational Use Case
Primary Titles	`"Bebas Neue", sans-serif`	`font-heading`	Main high-signal card headers, page titles.
Prose & Body	`"IBM Plex Sans Variable", sans-serif`	`font-sans`	Explanatory text, paragraphs, tables.
Technical Chrome	`"IBM Plex Mono", monospace`	`font-mono`	Navigation links, settings inputs, SQL queries, code.

---

4. SVG Icon Mapping Catalog

To ensure visual consistency and completely eliminate platform-inconsistent emojis, the navigation chrome (`Sidebar.astro` and `MobileNav.astro`) maps dynamic section keys to clean, flat, inline SVGs:

A. Rocket Icon (`getting-started`)

Used to represent system setup and installation paths.

```
<svg
  class="size-3.5"
  viewBox="0 0 24 24"
  fill="none"
  stroke="currentColor"
  stroke-width="2"
>
  <path
    d="M4.5 16.5c-1.5 1.26-2 5-2 5s3.74-.5 5-2c.71-.84.7-2.13-.09-2.91a2.18 2.18 0 0 0-2.91-.09z"
  />
  <path
    d="M12 15l-3-3a22 22 0 0 1 2-3.95A12.88 12.88 0 0 1 22 2c0 2.72-.78 7.5-6 11a22.35 22.35 0 0
  />
  <path d="M9 12H4s.55-3.03 2-4c1.62-1.08 5 0 5 0" />
  <path d="M12 15v5s3.03-.55 4-2c1.08-1.62 0-5 0-5" />
</svg>
```

B. Book Icon (`guides`)

Used to represent operational manuals and setup instructions.

```
<svg
  class="size-3.5"
  viewBox="0 0 24 24"
  fill="none"
  stroke="currentColor"
  stroke-width="2"
>
  <path d="M4 19.5A2.5 2.5 0 0 1 6.5 17H20" />
  <path d="M6.5 2H20v20H6.5A2.5 2.5 0 0 1 4 19.5v-15A2.5 2.5 0 0 1 6.5 2z" />
</svg>
```

C. Lightbulb Icon (`concepts`)

Used to represent structural theories and edge architectures.

```
<svg
  class="size-3.5"
  viewBox="0 0 24 24"
  fill="none"
  stroke="currentColor"
  stroke-width="2"
>
  <path d="M9 18h6" />
  <path d="M10 22h4" />
  <path
    d="M15.09 14c.18-.98.65-1.74 1.41-2.5A4.65 4.65 0 0 0 18 8 6 6 0 0 0 6 8c0 1 .23 2.23 1.5 3.5
  />
</svg>
```

D. Book-Open Icon (`reference`)

Used to represent dictionary definitions, configuration matrices, and API details.

```
<svg
  class="size-3.5"
  viewBox="0 0 24 24"
  fill="none"
  stroke="currentColor"
  stroke-width="2"
>
  <path d="M2 3h6a4 4 0 0 1 4 4v14a3 3 0 0 0-3-3H2z" />
  <path d="M22 3h-6a4 4 0 0 0-4 4v14a3 3 0 0 1 3-3h7z" />
</svg>
```

E. Target Icon (`tutorials`)

Used to represent step-by-step walkthroughs and integration guides.

```
<svg
  class="size-3.5"
  viewBox="0 0 24 24"
  fill="none"
  stroke="currentColor"
  stroke-width="2"
>
  <circle cx="12" cy="12" r="10" />
  <circle cx="12" cy="12" r="6" />
  <circle cx="12" cy="12" r="2" />
</svg>
```

F. Gear Icon (`devops` & `workers`)

Used to represent background cron engines, system setups, and deployment configs.

```
<svg
  class="size-3.5"
  viewBox="0 0 24 24"
  fill="none"
  stroke="currentColor"
  stroke-width="2"
>
  <circle cx="12" cy="12" r="3" />
  <path
    d="M12 1v2M12 21v2M4.22 4.22l1.42 1.42M18.36 18.36l1.42 1.42M1 12h2M21 12h2M4.22 19.78l1.42-1
  />
</svg>
```

G. Home House Icon (`home`)

Used to return to the parent portal.

```
<svg
  class="size-4"
  viewBox="0 0 24 24"
  fill="none"
  stroke="currentColor"
  stroke-width="2"
>
  <path d="M3 9l9-7 9 7v11a2 2 0 0 1-2 2H5a2 2 0 0 1-2-2z" />
  <polyline points="9 22 9 12 15 12 15 22" />
</svg>
```

Next Steps

- **System Topology Overview** — Analyze edge isolates and data layers integrations.
- **Isolate Communication Spec** — Check service bindings and standard Bearer Auth middleware.

12 · ARCHITECTURE

ARCHITECTURE DECISION RECORDS

Significant architectural choices for the Hoox trading platform — context, decision, and consequences.

This directory contains the Architecture Decision Records for the Hoox trading platform. Each ADR documents a significant architectural choice, its context, the decision made, and its consequences.

ADR Index

#	Title	Status	Date
001	Cloudflare Workers as Execution Platform	Accepted	2025-04
002	SQLite-backed Durable Objects for Idempotency	Accepted	2025-05
003	9-Microservice Architecture	Accepted	2025-06
004	Dual-Layer Configuration (Env + KV)	Accepted	2025-06
005	Multi-Provider AI Gateway with Fallback Chain	Accepted	2026-03
006	Bun Workspaces for Monorepo Management	Accepted	2025-04

ADR-001: Cloudflare Workers as Execution Platform

Status: Accepted

Context: Algorithmic trading requires ultra-low latency between signal generation and order execution. Traditional VPS-based bots suffer from 200ms+ round-trip latency due to fixed geographic locations and heavy runtime overhead.

Decision: Deploy all trading logic as Cloudflare Workers running on V8 isolates at the edge.

Consequences:

- **Positive:** 30-60% latency reduction vs. VPS
- **Negative:** 128MB memory limit per isolate
- **Mitigation:** Web3 wallet worker uses EVM-compatible RPC calls

ADR-002: SQLite-backed Durable Objects for Idempotency

Status: Accepted

Context: Webhook-based trade execution is inherently unreliable — network timeouts can cause TradingView to retry the same signal, potentially resulting in duplicate trades. Financial systems require exactly-once execution semantics.

Decision: Implement idempotency using Cloudflare Durable Objects with SQLite-backed state. Each incoming trade signal is hashed into a unique trace ID. A singleton DO instance per trace ID acts as a distributed mutex lock, atomically checking and recording whether the signal has been processed.

Consequences:

- **Positive:** Zero-race-condition dedup with single-threaded DO isolation
- **Negative:** Adds one network hop per request
- **Mitigation:** DO alarm scheduled at TTL for automatic garbage collection

---

ADR-003: 9-Microservice Architecture

Status: Accepted

Context: Monolithic trading systems are fragile — a crash in any component (portfolio calculation, notification, database) takes down the entire system. Microservices provide isolation but increase operational complexity.

Decision: Split the platform into 9 specialized workers, each with a single responsibility:

Worker	Responsibility
`hoox`	Gateway, validation, rate limiting, idempotency
`trade-worker`	Exchange order execution, leverage calculation
`agent-worker`	AI risk management, trailing stops, cron jobs
`d1-worker`	All SQLite read/write operations
`telegram-worker`	Push notifications, chat command handling
`web3-wallet-worker`	On-chain DeFi swap execution
`email-worker`	Email signal parsing (POP3/IMAP)
`analytics-worker`	Metrics collection and reporting
`report-worker`	PDF portfolio report generation

Consequences:

- **Positive:** Independent scaling; failure isolation; focused development; clean ownership boundaries
- **Negative:** More deployment units to manage; Service Binding dependency chain complexity; inter-worker communication overhead
- **Mitigation:** CLI automates dependency-ordered deployment; health check endpoints on every worker; TUI provides real-time status of all 9 workers

---

ADR-004: Dual-Layer Configuration (Environment + KV)

Status: Accepted

Context: Configuration needs fall into two categories.

Decision: Use two distinct configuration layers:

1. **Build-time (`.env.local` + Workers Secrets):** {" "} API keys, internal auth keys, dashboard credentials. Managed via{" "} `hoox secrets set` and `hoox config env`.
2. **Runtime (Cloudflare KV):** Kill switch, exchange routing, rate-limiter thresholds, email regex patterns. Managed via{" "} `hoox config kv set` and propagates globally in under 10 seconds.

Consequences:

- **Positive:** Kill switch takes effect on next request
- **Negative:** Config values are eventually consistent
- **Mitigation:** KV manifest with documented defaults; `hoox config kv apply-manifest` for one-command sync; documentation clearly separates the two layers

---

ADR-005: Multi-Provider AI Gateway with Fallback Chain

Status: Accepted

Context: The agent-worker needs LLM capabilities for risk assessment, market analysis, and user chat. Single-provider AI introduces single points of failure and cost concentration.

Decision: Implement a 5-provider fallback chain ordered by cost and availability:

```
Workers AI (free) -> Anthropic -> Google AI -> OpenAI -> Azure OpenAI
```

Each provider is health-checked before use. On failure, the gateway automatically retries with the next provider. Usage and costs are tracked per-provider.

Consequences:

- **Positive:** Zero single point of failure for AI; cost optimization (use free tier first); resilience against provider outages; vision + reasoning + chat in one gateway
- **Negative:** Complex provider abstraction layer; potential for inconsistent response quality across providers; harder to optimize prompts for all providers simultaneously
- **Mitigation:** Provider selection per-request (caller specifies preferred provider); standardized response schemas; usage dashboard for cost monitoring

---

ADR-006: Bun Workspaces for Monorepo Management

Status: Accepted

Context: The project contains multiple packages (CLI, TUI, shared library, 9 workers) that need to share types, utilities, and be tested/developed locally.

Decision: Use Bun Workspaces as the monorepo manager instead of npm/yarn/pnpm workspaces. Rationale:

- Native TypeScript compilation (no separate tsconfig complexity)
- Built-in test runner (no Jest/Mocha dependency)
- Fastest install times among JS package managers
- Single `bun.lockb` lockfile for all workspaces
- Native script runner for CI pipeline

Consequences:

- **Positive:** Zero-config TypeScript; fast CI; unified tooling; shared types resolve at compile time
- **Negative:** Bun is less mature than npm ecosystem; some packages may have compatibility issues; harder to onboard developers unfamiliar with Bun

- **Mitigation:** Comprehensive `hoox check prerequisites` validation; Docker runtime as fallback; documentation covers both native and Docker workflows

13 · WORKERS

HOOX GATEWAY ISOLATE PROFILE

Comprehensive engineering specification for the Hoox public gateway, covering ingress WAF rules, Durable Object idempotency stores, and Service Binding configurations.

The **hoox** gateway is the public-facing entry point of the trading ecosystem. Running as an ultra-low-latency Cloudflare Worker, the gateway is responsible for authorizing incoming trade signals (TradingView alerts, email routing, manual commands), executing rate-limiting checks, locking transaction trace IDs via Durable Objects to prevent duplicate fills, and routing validated events privately to background compute nodes.

Architectural Topology

```
graph TD
    Alert["External Webhook"] -->|Signal| WAF["Cloudflare WAF / Firewall"]
    WAF -->|IP & Auth Checks| Gateway["Gateway: hoox

Public Isolate"]
    Gateway -->|V8 Service Bindings

Private, Encrypted, Zero-TCP| Compute["Background Nodes"]

    subgraph Compute ["Background Compute Isolates"]
        Trade["trade-worker"]
        Tele["telegram-worker"]
        Anal["analytics-worker"]
    end

    style Alert fill:#CCCCCC,stroke:#3b82f6,stroke-width:2
    style WAF fill:#CCCCCC,stroke:#f59e0b,stroke-width:2
    style Gateway fill:#CCCCCC,stroke:#ef4444,stroke-width:2
    style Trade fill:#CCCCCC,stroke:#10b981,stroke-width:1
    style Tele fill:#CCCCCC,stroke:#10b981,stroke-width:1
    style Anal fill:#CCCCCC,stroke:#10b981,stroke-width:1
```

1. Declared Wrangler Configurations & Bindings

The gateway's `wrangler.jsonc` defines its private service binding links and resource bounds:

```
{
  "name": "hoox",
  "main": "src/index.ts",
  "compatibility_date": "2026-05-19",
  "compatibility_flags": ["nodejs_compat"],
  "account_id": "debc6545e63bea36be059cbc82d80ec8",
  "placement": {
    "mode": "smart",
  },
  "vars": {
    "ENVIRONMENT": "production",
  },
  "kv_namespaces": [
    {
      "binding": "CONFIG_KV",
      "id": "c5917667a21745e390ff969f32b1847d",
    },
    {
      "binding": "SESSIONS_KV",
      "id": "ff70a58b492e45d79880a7a8213c745c",
    },
  ],
  "services": [
    { "binding": "TRADE_SERVICE", "service": "trade-worker" },
    { "binding": "TELEGRAM_SERVICE", "service": "telegram-worker" },
  ],
  "queues": {
    "producers": [{ "queue": "trade-execution", "binding": "TRADE_QUEUE" }],
  },
  "durable_objects": {
    "bindings": [
      { "name": "IDEMPOTENCY_STORE", "class_name": "IdempotencyStore" },
    ],
  },
  "migrations": [
    {
      "tag": "v1",
      "new_sqlite_classes": ["IdempotencyStore"],
    },
  ],
}
```

---

2. Environmental Variables & Encrypted Secrets

For security, build-time credentials are never stored in plain text. They are bound at deploy time as encrypted secrets:

- **`WEBHOOK_API_KEY`**: The custom authentication passkey expected inside incoming signal payloads.
- **`INTERNAL_KEY_BINDING`**: The shared bearer authorization token used by `requireInternalAuth` middleware to invoke internal V8 isolates.

Local Development Mocking (`.dev.vars`)

When running tests or starting local Wrangler dev, create a gitignored `.dev.vars` file in the gateway directory:

```
WEBHOOK_API_KEY=dev_webhook_auth_passkey
INTERNAL_KEY_BINDING=dev_shared_internal_security_key
```

---

3. API Route Specifications

> **Note:** For the canonical endpoint directory with full request/response examples across all workers, see `/docs/devops/api/endpoints`.

The gateway exposes three public entryways:

A. Ingest Signal Webhook

- **Endpoint**: `/webhook`
- **Method**: `POST`
- **JSON Payload**:

```
{
  "apiKey": "dev_webhook_auth_passkey",
  "exchange": "bybit",
  "action": "LONG",
  "symbol": "BTCUSDT",
  "quantity": 0.01,
  "leverage": 10,
  "idempotencyKey": "uuid-9b1deb4d-3b7d"
}
```

- **Success Response (200 OK)**:

```
{
  "success": true,
  "requestId": "9b1deb4d-3b7d-4bad-9bdd-2b0d7b3dcb6d",
  "exchange": "bybit",
  "symbol": "BTCUSDT",
  "action": "LONG",
  "result": { "orderId": "18049284739", "status": "Filled" }
}
```

B. Gateway Health Diagnostics

- **Endpoint**: `/health`
- **Method**: `GET`
- **Success Response (200 OK)**:

```
{
  "status": "ok",
  "timestamp": 1779261050000,
  "bindings": {
    "d1": "connected",
    "kv": "connected",
    "queue": "active"
  }
}
```

If exchange APIs experience high latency or go offline, the gateway intercepts the timeout error, serializes the payload, and pushes it to the `TRADE_QUEUE` producer in less than **2 milliseconds**, returning a `"status": "Enqueued"` (202 Accepted) response to TradingView.

Next Steps

- **trade-worker Spec** — Review how trade executions, order math, and margin settings compile on the edge.
- **D1 Database Operations** — Manage schema migrations, query ledgers, and execute SQL scripts.

14 · WORKERS

TRADE-WORKER ISOLATE PROFILE

Comprehensive engineering specification for the Hoox Trade Execution Worker, covering multi-exchange client architectures, HMAC signing, and D1 database ledgers.

The **`trade-worker`** is the execution engine of the Hoox trading platform. Deployed as a private compute isolate behind the edge firewall, this worker is responsible for calculating order parameters, executing leverage scaling, performing cryptographic HMAC-SHA256 signature calculations, placing trades on Bybit, Binance, and MEXC, and offloading transactional metrics.

---

1. Declared Wrangler Configurations & Bindings

The `trade-worker` does not expose a public URL, communicating internally via V8 Service Bindings. Its `wrangler.jsonc` maps out its critical storage, queue, and database hooks:

```
{
  "name": "trade-worker",
  "main": "src/index.ts",
  "compatibility_date": "2026-05-19",
  "compatibility_flags": ["nodejs_compat"],
  "account_id": "debc6545e63bea36be059cbc82d80ec8",
  "placement": {
    "mode": "smart",
  },
  "d1_databases": [
    {
      "binding": "DB",
      "database_name": "trade-data-db",
      "database_id": "c5917667a21745e390ff969f32b1847a",
    },
  ],
  "r2_buckets": [
    { "binding": "REPORTS_BUCKET", "bucket_name": "trade-reports" },
    { "binding": "SYSTEM_LOGS_BUCKET", "bucket_name": "hoox-system-logs" },
  ],
  "kv_namespaces": [
    {
      "binding": "CONFIG_KV",
      "id": "c5917667a21745e390ff969f32b1847d",
    },
  ],
  "queues": {
    "consumers": [
      {
        "queue": "trade-execution",
        "max_batch_size": 10,
        "max_batch_timeout": 1,
      },
    ],
  },
  "secrets": [
    "INTERNAL_KEY_BINDING",
    "BYBIT_API_KEY",
    "BYBIT_API_SECRET",
    "BINANCE_API_KEY",
    "BINANCE_API_SECRET",
    "MEXC_API_KEY",
    "MEXC_SECRET_BINDING",
  ],
}
```

---

2. The Provider-Based `ExchangeRouter` Pattern

To support multiple centralized exchanges with different API schemas while maintaining a clean code structure, `trade-worker` implements a **Provider Composition Pattern**:

A. Generic Exchange Provider Interface

The client abstraction is defined inside the shared monorepo package `@jango-blockchained/hoox-shared/types`:

```
  readonly name: string;
  createClient(env: TEnv): TClient;
  hasCredentials(env: TEnv): boolean;
}
```

B. Dynamic Runtime Routing

The `ExchangeRouter` evaluates the incoming symbol and parses settings in `CONFIG_KV` in sub-milliseconds:

1. **Default Path**: Routes trades to the default CEX declared in `exchanges:default_routing` (typically `bybit`).
2. **Dynamic Symbol Redirects**: Parses overrides in KV (e.g. `exchanges:routing:SOLUSDT = binance`). If present, the router bypasses Bybit and instantiates the `BinanceProvider` instantly without redeploying code.

---

3. Internal REST API Specification

> **Note:** For the canonical endpoint directory with full request/response examples across all workers, see `/docs/devops/api/endpoints`.

A. Process Order Pipeline

Invoked by the `hoox` gateway or the `TRADE_QUEUE` consumer batch runner.

- **Endpoint**: `/process`
- **Method**: `POST`
- **Headers**: `X-Internal-Auth-Key: <INTERNAL_KEY_BINDING>`
- **JSON Payload**:

```
{
  "requestId": "9b1deb4d-3b7d-4bad-9bdd-2b0d7b3dcb6d",
  "payload": {
    "exchange": "bybit",
    "action": "LONG",
    "symbol": "BTCUSDT",
    "quantity": 0.005,
    "leverage": 10
  }
}
```

- **Success Response (200 OK)**:

```
{
  "success": true,
  "result": {
    "orderId": "18049284739",
    "status": "Filled",
    "price": 68425.5
  },
  "error": null
}
```

4. Standardized Exception Handling

All execution rejects and validation failures are intercepted by the `trade-worker` error middleware and formatted using the shared `Errors` factory from `@jango-blockchained/hoox-shared/errors`:

```
// 1. Parameter Validation Failure
if (quantity <= 0) {
  return Errors.badRequest("Quantity parameter must be greater than zero.");
}

// 2. Exchange Signature Timeout
if (timestampExpired) {
  return Errors.unauthorized(
    "Timestamp verification failed. Check system NTP sync."
  );
}

// 3. API Execution Rejects
try {
  await exchange.placeOrder(order);
} catch (err: any) {
  return Errors.internal(`Exchange API Reject: ${err.message}`);
}
```

If the exchange API rejects an order due to account rate-limiting, the queue consumer automatically returns a retry flag. Cloudflare Queues will back off and re-route the batch at intervals starting at 30 seconds, protecting your strategy from missed fills.

Next Steps

- **hoox Gateway Profile** — Review WAF rules and Durable Object idempotency locks.
- **D1 Database Operations** — Manage Drizzle schemas, migrations, and query operations.

15 · WORKERS

D1-WORKER ISOLATE PROFILE

Comprehensive engineering specification for the Hoox D1 SQLite Database Proxy Worker, covering SQL query interfaces, batch operations, and dashboard statistics aggregation.

The **`d1-worker`** is the data routing hub of the Hoox trading platform. Deployed as an isolated, private micro-worker, it acts as a centralized SQL execution proxy. By encapsulating database interactions behind secure Service Bindings, it allows other lightweight compute workers to execute parameterized queries, trigger transactional batch operations, and retrieve structured dashboard telemetry without direct database driver overhead.

---

1. Declared Wrangler Configurations & Bindings

The `d1-worker` binds directly to the production SQLite database (`trade-data-db`) and does not expose any public endpoints:

```
{
  "name": "d1-worker",
  "account_id": "debc6545e63bea36be059cbc82d80ec8",
  "main": "src/index.ts",
  "alias": {
    "@jango-blockchained/hoox-shared/errors": "../../packages/shared/src/errors.ts",
    "@jango-blockchained/hoox-shared/middleware": "../../packages/shared/src/middleware/index.ts"
    "@jango-blockchained/hoox-shared/router": "../../packages/shared/src/router.ts",
    "@jango-blockchained/hoox-shared/types": "../../packages/shared/src/types.ts",
    "@jango-blockchained/hoox-shared/analytics": "../../packages/shared/src/analytics.ts",
    "@jango-blockchained/hoox-shared/health": "../../packages/shared/src/health.ts",
    "@jango-blockchained/hoox-shared/types/router": "../../packages/shared/src/types/router.ts",
  },
  "compatibility_date": "2026-04-17",
  "compatibility_flags": ["nodejs_compat"],
  "placement": {
    "mode": "smart",
  },
  "observability": {
    "enabled": true,
    "head_sampling_rate": 1,
  },
  "d1_databases": [
    {
      "binding": "DB",
      "database_name": "trade-data-db",
      "database_id": "a682f084-594e-4bd8-be2d-40ea5f8cf42e",
    },
  ],
  "vars": {
    "INTERNAL_KEY_BINDING": "__SECRET__",
  },
  "kv_namespaces": [
    {
      "binding": "CONFIG_KV",
      "id": "c5917667a21745e390ff969f32b1847d",
    },
  ],
  "services": [
    {
      "binding": "ANALYTICS_SERVICE",
      "service": "analytics-worker",
    },
  ],
}
```


2. Internal REST API Specification

> **Note:** For the canonical endpoint directory with full request/response examples across all workers, see `/docs/devops/api/endpoints`.

Every endpoint is secured via `requireInternalAuth` and expects the `X-Internal-Auth-Key` header.

A. Execute Single SQL Query

- **Endpoint**: `/query`
- **Method**: `POST`
- **JSON Payload**:

```
{
  "query": "SELECT created_at, symbol, action, price FROM trades WHERE symbol = ? ORDER BY cre
  "params": ["BTCUSDT", 5]
}
```

- **SELECT Success Response (200 OK)**:

```
{
  "success": true,
  "results": [
    {
      "created_at": 1779261050000,
      "symbol": "BTCUSDT",
      "action": "LONG",
      "price": 68425.5
    }
  ]
}
```

- **Write Success Response (200 OK)** (INSERT/UPDATE/DELETE/REPLACE):

```
{
  "success": true,
  "lastRowId": 123,
  "changes": 1
}
```

B. Execute Transactional Batch Operations

Allows running multiple statements atomically in a single network trip, reducing latency.

- **Endpoint**: `/batch`
- **Method**: `POST`
- **JSON Payload**:

```
{
  "statements": [
    {
      "query": "INSERT INTO trades (id, symbol, price) VALUES (?, ?, ?)",
      "params": ["trade-1", "BTCUSDT", 68000]
    },
    {
      "query": "UPDATE positions SET size = size + ? WHERE symbol = ?",
      "params": [0.005, "BTCUSDT"]
    }
  ]
}
```

- **Success Response (200 OK)**:

```
{
  "success": true,
  "results": [
    { "success": true, "meta": { "last_row_id": 1, "changes": 1 } },
    { "success": true, "meta": { "changes": 1 } }
  ]
}
```

C. Dashboard Telemetry Statistics

Calculates aggregated win ratios, active positions size, and time-series P&L.

- **Endpoint**: `/api/dashboard/stats`
- **Method**: `GET`
- **Success Response (200 OK)**:

```
{
  "success": true,
  "stats": {
    "totalTrades": 1048,
    "winRate": 64.2,
    "totalPnlUSDT": 18340.5,
    "activePositionsCount": 2,
    "dailyTradesCount": 14
  }
}
```

---

3. Security & SQL Injection Protection

To protect financial transaction ledgers and portfolios against SQL injection attacks, `d1-worker` enforces strict development rules:

- **Parameterized Bindings**: All inputs must utilize parameterized placeholders (`?` or `?1`) mapped to `env.DB.prepare().bind()`. **Never** concatenate raw request strings directly into SQL statements.
- **Access Isolation**: The database does not exist publicly. By binding D1 solely to this worker and accessing it via V8 Service Bindings, you prevent external crawlers or bots from querying database nodes directly.

If you are extending schemas or adding tables, generate migration scripts locally using Drizzle: `hoox db migrate --remote`. This keeps edge schema histories atomic and securely tracked.

Next Steps

- **System Storage Architecture** — Review SQLite properties and R2 bucketing pipelines.
- **Database Operations Manual** — Learn commands to run query ledgers and restore backups.

16 · WORKERS

AGENT-WORKER ISOLATE PROFILE

Comprehensive engineering specification for the Hoox Agent Cron Worker, covering multi-provider AI Gateway configurations, vision analysis, and risk protection loops.

Last Updated: May 2026 (Post-Enhancement)

The `agent-worker` serves as the proactive intelligence layer of the Hoox trading ecosystem. Rather than waiting for webhooks, it runs continuously on a cron schedule to monitor portfolio health, enforce risk limits, and optimize position exits.

Core Capabilities

Feature	Description
⏱ **Cron-Driven Observation**	Automatically runs every 5 minutes (`*/5 * * *`) to fetch live market data from Binance, Bybit, and MEXC.
Global Kill Switch	Calculates total account PnL and instantly locks out the `hoox` gateway from new entries if the `max_daily_drawdown_percent` is breached.
Dynamic Trailing Stops	Stores watermark prices in `CONFIG_KV` and automatically triggers `CLOSE` payloads if the market reverses.
Scale-Out Take Profits	Detects when a position reaches a specific profit target and automatically sends partial close commands to secure gains.
AI System Summarization	Periodically fetches `system_logs` from the `d1-worker` , analyzes them via LLAMA 3 8B, and sends natural language health reports to Telegram.
Multi-Provider AI	Seamlessly switches between Workers AI, OpenAI, Anthropic, Google AI, and Azure OpenAI with automatic fallbacks.
Advanced Models	Supports vision, embeddings, reasoning (extended thinking), and code generation models.

Architecture & Flow

1. **Trigger:** Cloudflare® Cron triggers the worker.
2. **State Sync:** Fetches active `OPEN` positions via the `d1-worker` .
3. **Market Pulse:** Pings public exchange APIs for the latest `markPrice` .
4. **Risk Evaluation:** Cross-references current price with KV-stored watermarks and global drawdown limits.
5. **AI Processing:** Uses configured AI provider with automatic fallback chain.
6. **Execution:** Dispatches actions to `trade-worker` (closing positions) and `telegram-worker` (alerts) via internal Service Bindings.

Endpoints & Interactions

> **Note:** For the canonical endpoint directory with full request/response examples across all workers, see `/docs/devops/api/endpoints` .

Management Endpoints

`GET /agent/config`

Returns current agent configuration including provider settings.

```
{
  "success": true,
  "config": {
    "defaultProvider": "workers-ai",
    "fallbackChain": ["workers-ai", "openai"],
    "modelMap": { ... },
    "trailingStopPercent": 0.05,
    "takeProfitPercent": 0.10
  }
}
```

`POST /agent/config`

Update agent configuration at runtime.

```
{
  "defaultProvider": "openai",
  "fallbackChain": ["openai", "workers-ai", "anthropic", "google", "azure"],
  "modelMap": {
    "workers-ai": "@cf/meta/llama-3.1-8b-instruct",
    "openai": "gpt-4o-mini-2024-07-18",
    "anthropic": "claude-3-haiku-20240307",
    "google": "gemini-1.5-flash-002",
    "azure": "gpt-4o-mini"
  },
  "timeoutMs": 30000,
  "retryCount": 3
}
```

`GET /agent/models`

Returns all available models from Cloudflare Workers AI and external providers.

`POST /agent/test-model`

Test a specific AI model.

```
{
  "prompt": "Say hello",
  "model": "@cf/meta/llama-3.1-8b-instruct",
  "provider": "workers-ai"
}
```

`GET /agent/health`

Returns health status of all configured AI providers.

```
{
  "success": true,
  "providers": {
    "workers-ai": { "healthy": true, "latency": 150 },
    "openai": { "healthy": true, "latency": 200 }
  }
}
```

AI Interaction Endpoints

`POST /agent/chat`

Send a chat request with automatic provider fallback and **SSE streaming support**.

Request:

```
{
  "messages": [{ "role": "user", "content": "Analyze BTC market sentiment" }],
  "systemPrompt": "You are a professional crypto trading analyst.",
  "temperature": 0.7,
  "maxTokens": 500,
  "stream": true
}
```

Streaming Response (SSE):

```
data: {"content": "Based on current market conditions..."}
data: {"content": " technical indicators suggest..."}
data: [DONE]
```

`POST /agent/vision` **(NEW!)**

Analyze images with AI vision models. Supports both URL and base64 input.

```
{
  "imageUrl": "https://example.com/chart.png",
  "prompt": "Analyze this price chart and identify key support/resistance levels",
  "model": "@cf/meta/llama-3.2-11b-vision-instruct"
}
```

Or with base64:

```
{
  "imageBase64": "iVBORw0KGgoAAAANSUhEUgAA...",
  "prompt": "What pattern do you see in this chart?"
}
```

`POST /agent/reasoning` **(NEW!)**

Extended thinking queries with reasoning models like OpenAI o1.

```
{
  "prompt": "Design a risk management strategy for a $100k portfolio",
  "model": "o1-preview",
  "reasoningEffort": "medium"
}
```

Response:

```
{
  "reasoning": "Let me think through this step by step...",
  "answer": "Here's a comprehensive risk management strategy...",
  "model": "o1-preview"
}
```

`GET /agent/usage` (NEW!)

Get AI API usage statistics across all providers.

```
{
  "success": true,
  "usage": {
    "workers-ai": { "requests": 150, "tokens": 45000 },
    "openai": { "requests": 75, "tokens": 22000 }
  }
}
```

`GET /agent/prompts` (NEW!)

List available prompt templates.

```
{
  "success": true,
  "prompts": ["trading-analyst", "risk-assessor", "market-scanner"]
}
```

`POST /agent/embedding`

Generate text embeddings using Workers AI embedding models.

```
{
  "text": "Bitcoin price analysis for position sizing",
  "provider": "workers-ai"
}
```

Legacy Endpoints

`POST /agent/risk-override`

Manually enforce or release risk locks.

```
{
  "action": "engage_kill_switch",
  "reason": "Manual override from dashboard"
}
```

`GET /agent/status`

Retrieve the real-time health of the agent and active trailing stops.

Configuration

KV Keys

All configuration is stored in `CONFIG_KV` for real-time adjustments.

KV Key	Default	Description
`agent:config`	JSON object	Main provider configuration
`agent:openai_key`	-	OpenAI API key
`agent:anthropic_key`	-	Anthropic API key
`agent:google_key`	-	Google AI API key
`agent:azure_api_key`	-	Azure OpenAI API key
`agent:azure_endpoint`	-	Azure OpenAI endpoint URL
`trade:max_daily_drawdown_percent`	`-5`	Account PnL % that triggers Kill Switch
`trade:kill_switch`	`false`	When `true`, halts all new trades
`trade:watermark:{exchange}:{symbol}:{side}`	N/A	High/low watermark

Default Agent Config

```
{
  "defaultProvider": "workers-ai",
  "fallbackChain": ["workers-ai", "openai", "anthropic", "google", "azure"],
  "modelMap": {
    "workers-ai": "@cf/meta/llama-3.1-8b-instruct",
    "openai": "gpt-4o-mini-2024-07-18",
    "anthropic": "claude-3-haiku-20240307",
    "google": "gemini-1.5-flash-002",
    "azure": "gpt-4o-mini"
  },
  "timeoutMs": 30000,
  "retryCount": 3,
  "maxDailyDrawdownPercent": -5,
  "trailingStopPercent": 0.05,
  "takeProfitPercent": 0.1
}
```

Supported Models

Workers AI Models

Task	Workers AI Model
Chat	`@cf/meta/llama-3.1-8b-instruct`
Vision	`@cf/meta/llama-3.2-11b-vision-instruct`
Reasoning	`@cf/deepseek-ai/deepseek-r1-distill-qwen-32b`
Code	`@cf/qwen/qwen2.5-coder-32b-instruct`
Embeddings	`@cf/baai/bge-base-en-v1.5`
Summarization	`@cf/facebook/bart-large-cnn`

External Providers

Provider	Models
OpenAI	GPT-4o, GPT-4o-mini, GPT-4 Turbo, o1
Anthropic	Claude 3 Haiku, Sonnet, Opus
Google	Gemini 1.5 Flash, Gemini 1.5 Pro
Azure	GPT-4o, GPT-4o-mini (custom deployment)

AI Gateway Features

The AI Gateway provides:

- **Fallback Chain**: Automatically tries providers in order on failure
- **Health Checks**: Providers self-report health status every 5 minutes
- **Retry Logic**: Exponential backoff with configurable max retries
- **Timeout Protection**: Configurable per-request timeout (default 30s)
- **Usage Tracking**: Automatic token and request counting per provider

Internal Service Bindings

The `agent-worker` requires the following bindings to operate:

- `D1_SERVICE` : To fetch open positions and system logs.
- `TRADE_SERVICE` : To execute trailing stops and profit-taking.
- `TELEGRAM_SERVICE` : To broadcast AI summaries and emergency alerts.
- `CONFIG_KV` : For dynamic configuration and state.
- `AI` : Workers AI binding for inference.

Testing

The agent-worker includes comprehensive tests (**171 tests across 33 files**):

```
# Run all tests
bun test

# Run specific test file
bun test tests/ai/providers/openai.test.ts

# Run with coverage
bun test --coverage
```

Test Coverage: >80% across all modules

Test Structure

```
tests/
├── ai/
│   ├── providers/       # Tests for each AI provider
│   ├── gateway.test.ts  # AI Gateway fallback tests
│   ├── streaming.test.ts # SSE streaming tests
│   └── prompts.test.ts  # Prompt template tests
├── handlers/           # Tests for all endpoints
├── middleware/          # Tests for auth, validation, logging
├── market/             # Tests for exchange clients
├── risk/               # Tests for risk manager/executor
├── routine/            # Tests for housekeeping
└── integration/        # End-to-end integration tests
```

---

17 · WORKERS

TELEGRAM-WORKER ISOLATE PROFILE

Comprehensive engineering specification for the Hoox Telegram Notification & Command Worker, covering RAG vectorized indexes, R2 uploads, and alert payloads.

The **`telegram-worker`** serves as the dynamic communicator of the Hoox trading ecosystem. Running as an isolated edge microservice, it parses incoming chat commands forwarded from Telegram's webhooks, executes retrieval-augmented generation (RAG) queries via Vectorize, analyzes screenshots using AI vision models, and dispatches real-time HTML/Markdown formatting order alerts to your mobile.

---

1. Declared Wrangler Configurations & Bindings

The `telegram-worker` mounts multiple storage and vector search services to implement AI-powered chatbot features. Its `wrangler.jsonc` specifies:

```
{
  "name": "telegram-worker",
  "main": "src/index.ts",
  "compatibility_date": "2026-05-19",
  "compatibility_flags": ["nodejs_compat"],
  "account_id": "debc6545e63bea36be059cbc82d80ec8",
  "placement": {
    "mode": "smart",
  },
  "kv_namespaces": [
    {
      "binding": "CONFIG_KV",
      "id": "c5917667a21745e390ff969f32b1847d",
    },
  ],
  "r2_buckets": [
    {
      "binding": "UPLOADS_BUCKET",
      "bucket_name": "user-uploads",
    },
  ],
  "vectorize": [
    {
      "binding": "VECTORIZE_INDEX",
      "index_name": "rag-index",
    },
  ],
  "ai": {
    "binding": "AI",
  },
  "secrets": [
    "INTERNAL_KEY_BINDING",
    "TG_BOT_TOKEN_BINDING",
    "TG_CHAT_ID_BINDING",
    "TELEGRAM_SECRET_TOKEN",
  ],
}
```

---

2. Environmental Variables & Encrypted Secrets

- **`TG_BOT_TOKEN_BINDING`**: Private HTTP bot token generated by `@BotFather`.
- **`TG_CHAT_ID_BINDING`**: Your default Chat ID for receiving notifications.
- **`TELEGRAM_SECRET_TOKEN`**: A secure, random token sent as the `X-Telegram-Bot-Api-Secret-Token` header to validate webhook requests from Telegram.
- **`INTERNAL_KEY_BINDING`**: Shared key used to validate calls from `hoox` or `trade-worker`.
- **`AUTHORIZED_CHAT_IDS`**: Optional comma-separated list of chat IDs allowed to send bot commands. If set, only these chat IDs can interact with the bot.

Local Development Mocking (`.dev.vars`)

```
TG_BOT_TOKEN_BINDING=mock_telegram_bot_token
TG_CHAT_ID_BINDING=987654321
TELEGRAM_SECRET_TOKEN=local_secure_route_token
INTERNAL_KEY_BINDING=dev_shared_internal_security_key
```

---

3. Internal REST API Specification

> **Note:** For the canonical endpoint directory with full request/response examples across all workers, see `/docs/devops/api/endpoints`.

A. Dispatch Notification Endpoint

- **Endpoint**: `/alert`
- **Method**: `POST`
- **Headers**: `X-Internal-Auth-Key: <INTERNAL_KEY_BINDING>`
- **JSON Payload**:

```
{
  "requestId": "9b1deb4d-3b7d-4bad-9bdd-2b0d7b3dcb6d",
  "payload": {
    "chatId": "987654321",
    "message": "<b> Bybit LONG Filled!</b>\nSymbol: <code>BTCUSDT</code>\nPrice: <code>$68,425
    "parseMode": "HTML"
  }
}
```

- **Success Response (200 OK)**:

```
{
  "success": true,
  "result": { "messageId": 482904 },
  "error": null
}
```

B. Telegram Ingress Webhook Route

Receives chat commands, `/start` signals, and image binaries.

- **Endpoint**: `/telegram/<TELEGRAM_WEBHOOK_SECRET>`
- **Method**: `POST`
- **JSON Payload**: Standard Telegram update JSON schema.
- **Access Control**: The worker extracts the incoming `message.chat.id`. If it does not match your authorized `TELEGRAM_CHAT_ID_DEFAULT` secret, the payload is silently dropped with a `200 OK` return to Telegram to prevent malicious commands.

---

4. AI-Powered Chatbot & RAG Mechanics

The bot does not just return static responses. When you send a natural language question (e.g., *"What is my average entry price on SOL?"*):

1. **Embedding Generation**: The worker calls `env.AI` to generate high-dimensional text embeddings for your prompt using the `@cf/baai/bge-base-en-v1.5` model.
2. **Vector Query**: Passes the vector to `env.VECTORIZE_INDEX` to retrieve semantically related transaction rows and market summaries from past trades.
3. **Context Construction**: Formats the matched history into a rich LLM context window.
4. **LLM Inference**: Invokes LLaMA-3 instruct via `env.AI` using your multi-provider API keys to generate a highly informed financial summary.

Secure your bot webhook instantly after deployment: `hoox deploy telegram-webhook`. The CLI automatically queries your secrets, makes the HTTP setup call to Telegram's servers, and validates the TLS tunnel!

Next Steps

- **hoox Gateway Profile** — Review how incoming commands route to the gateway.
- **Setting Up Telegram Bot Alerts** — Step-by-step tutorial for BotFather configuration.

EMAIL-WORKER ISOLATE PROFILE

Engineering specification for the Hoox email parsing worker, covering Mailgun webhook ingestion, direct JSON signal parsing, KV-configured regex patterns, and service bindings.

The **`email-worker`** ingests trading signals from email sources, parses them, and forwards them to `trade-worker` for execution. It receives emails via Mailgun webhook (`POST /webhook`) or direct JSON POST (`POST /email-signal`). No IMAP/SMTP polling — Cloudflare Workers edge runtime does not support the Node.js `net` / `tls` modules required for IMAP.

The worker also tracks signal ingestion metrics by forwarding event data to `analytics-worker` via service binding.

---

1. Endpoints

> **Note:** For the canonical endpoint directory with full request/response examples across all workers, see `/docs/devops/api/endpoints` .

Endpoint	Method	Auth	Purpose
`/webhook`	POST	Mailgun signature (HMAC-SHA256)	Inbound Mailgun webhook for forwarded emails
`/email-signal`	POST	`X-Internal-Auth-Key` header	Direct JSON signal ingestion
`/health`	GET	None	Health check

The Mailgun webhook endpoint is `/webhook` , not `/webhook/mailgun` . The direct JSON endpoint is `/email-signal` , not `/process` .

---

2. Mailgun Signature Verification

When a POST arrives at `/webhook` , the worker validates the Mailgun signature before processing the email body:

1. Reads `Mailgun-Signature` , `Mailgun-Timestamp` , `Mailgun-Token` from request headers
2. Computes HMAC-SHA256 hex digest of `timestamp + token` using `MAILGUN_API_KEY` as the signing key
3. Compares computed digest against the `Mailgun-Signature` header value
4. Returns `401 Unauthorized` if headers are missing or signature does not match

```
const dataToSign = timestamp + token;
const encoder = new TextEncoder();
const key = await crypto.subtle.importKey(
  "raw",
  encoder.encode(apiKey),
  { name: "HMAC", hash: "SHA-256" },
  false,
  ["sign"]
);
const signatureBytes = await crypto.subtle.sign(
  "HMAC",
  key,
  encoder.encode(dataToSign)
);
const expectedSignature = Array.from(new Uint8Array(signatureBytes))
  .map((b) => b.toString(16).padStart(2, "0"))
  .join("");

if (signature !== expectedSignature) {
  return Errors.unauthorized("Invalid signature");
}
```

On success, the worker extracts the email body from the `body-plain` or `stripped-text` field of the Mailgun form-data payload and passes it to the signal parser.

---

3. Signal Extraction

Two-phase parsing: JSON first, plaintext fallback.

Phase 1: JSON Parsing

`parseEmailSignal()` calls `JSON.parse()` on the body text. If the result contains `exchange`, `action`, and `symbol` fields, it returns a structured signal immediately:

```
{
  exchange: "binance",     // normalized lowercase
  action: "buy",           // buy/long → buy | sell/short → sell
  symbol: "BTCUSDT",       // uppercased
  quantity: 100,           // multiplied by quantityMultiplier from KV
  price?: 45000,           // optional
  leverage?: 3             // optional
}
```

Phase 2: Plaintext Fallback

If JSON parsing fails or required fields are missing, `extractFromPlaintext()` uses keyword matching:

- `extractField()` scans for `keyword:` prefixes (e.g. `exchange:`, `symbol:`, `action:`)
- Coin symbols and action keywords matched via regex from KV config (`coinPattern`, `actionPattern`)
- `normalizeExchange()` resolves exchanges: `binance`, `mexc`, `bybit`
- `normalizeAction()` maps: `buy` / `long` → `buy`, `sell` / `short` → `sell`

Example plaintext email body:

```
exchange: binance
symbol: BTCUSDT
action: buy
quantity: 1.5
```

Zod Validation

Incoming JSON payloads are validated using Zod schemas:

- `EmailSignalSchema` validates the signal structure (exchange, action as enum, symbol, quantity with default 100, optional price/leverage)
- `WebhookPayloadSchema` validates the wrapper payload (subject, text, body — all optional)
- Invalid payloads return `400 Bad Request` with structured error
- Extra fields are stripped via `.strip()`

Forwarding

Once parsed, the signal is sent to `trade-worker` via:

```
const response = await serviceFetch(env.TRADE_SERVICE, "/webhook", signal, {
  headers: {
    "X-Internal-Auth-Key": internalKey,
    "X-Source": "email-worker",
  },
});
```

Analytics events are sent non-blocking via `ctx.waitUntil()`:

```
ctx.waitUntil(
  trackAnalytics(env, "/track/signal", {
    data: {
      source: "email-worker",
      type: signal.action,
      symbol: signal.symbol,
      confidence: 0.5,
    },
  })
);
```

Cloudflare Email Routing

The worker also exposes an `email()` handler that processes incoming emails via Cloudflare Email Routing. When an email arrives, `postal-mime` parses the raw MIME content, extracts the text body, and feeds it through the same `parseEmailSignal()` pipeline. Validated signals are forwarded to `trade-worker` with analytics tracking.

---

4. Bindings

Service Bindings

Binding	Target Worker	Purpose
`TRADE_SERVICE`	trade-worker	Forward parsed trading signals for execution
`ANALYTICS_SERVICE`	analytics-worker	Track signal ingestion metrics

Send Email Binding

Binding	Purpose
`EMAIL`	Cloudflare Email Routing — receive inbound emails

KV Namespaces

Binding	ID	Purpose
`CONFIG_KV`	`c5917667a21745e390ff969f32b1847d`	Signal pattern configuration

---

5. Secrets

All secrets are set via `wrangler secret put <name>`:

Secret	Purpose
`INTERNAL_KEY_BINDING`	Shared internal auth key for service-to-service calls
`MAILGUN_API_KEY`	Mailgun webhook signature verification (HMAC-SHA256)
`EMAIL_HOST_BINDING`	Reserved for future email host configuration
`EMAIL_USER_BINDING`	Reserved for future email user configuration
`EMAIL_PASS_BINDING`	Reserved for future email password configuration

---

6. Environment Variables (Vars)

Variable	Value	Purpose
`TRADE_WORKER_NAME`	`trade-worker`	Service name of trade-worker (reference only)

---

7. KV Configuration Keys

The worker loads signal parsing patterns from `CONFIG_KV` via `loadSignalPatterns()`:

KV Key	Default	Purpose
`email:coin_pattern`	`BTC|ETH|SOL`	Regex for matching asset symbols in email body
`email:action_pattern`	`buy|sell|long|short`	Regex for matching trade action direction
`email:quantity_multiplier`	`1`	Coefficient applied to parsed quantity values

```
const [coinPattern, actionPattern, quantityMultiplier] = await Promise.all([
  env.CONFIG_KV?.get(KVKeys.KV_EMAIL_COIN_PATTERN).then(
    (v) => v || "BTC|ETH|SOL"
  ),
  env.CONFIG_KV?.get(KVKeys.KV_EMAIL_ACTION_PATTERN).then(
    (v) => v || "buy|sell|long|short"
  ),
  env.CONFIG_KV?.get(KVKeys.KV_EMAIL_QUANTITY_MULTIPLIER).then((v) =>
    v ? parseFloat(v) : 1
  ),
]);
```

---

8. Observability

Full observability enabled with 100% head sampling:

```
{
  "observability": {
    "enabled": true,
    "head_sampling_rate": 1,
    "logs": {
      "enabled": true,
      "head_sampling_rate": 1,
      "persist": true,
      "invocation_logs": true,
    },
  },
}
```

- **Head sampling rate**: 1 (all requests sampled)
- **Log persistence**: Enabled — logs stored for debugging and audit
- **Invocation logs**: Enabled — full invocation records available
- **Smart Placement**: Enabled for 30-60% latency reduction

---

9. Configuration (wrangler.jsonc)

```
{
  "name": "email-worker",
  "main": "src/index.ts",
  "compatibility_date": "2026-04-17",
  "compatibility_flags": ["nodejs_compat"],
  "account_id": "debc6545e63bea36be059cbc82d80ec8",
  "placement": {
    "mode": "smart",
  },
  "observability": {
    "enabled": true,
    "head_sampling_rate": 1,
    "logs": {
      "enabled": true,
      "head_sampling_rate": 1,
      "persist": true,
      "invocation_logs": true,
    },
  },
  "vars": {
    "TRADE_WORKER_NAME": "trade-worker",
  },
  "kv_namespaces": [
    {
      "binding": "CONFIG_KV",
      "id": "c5917667a21745e390ff969f32b1847d",
    },
  ],
  "services": [
    {
      "binding": "TRADE_SERVICE",
      "service": "trade-worker",
    },
    {
      "binding": "ANALYTICS_SERVICE",
      "service": "analytics-worker",
    },
  ],
  "send_email": [
    {
      "name": "EMAIL",
    },
  ],
}
```

Secrets (`INTERNAL_KEY_BINDING` , `MAILGUN_API_KEY` , `EMAIL_HOST_BINDING` , `EMAIL_USER_BINDING` , `EMAIL_PASS_BINDING`) are set via `wrangler secret put <name>` and are not present in the checked-in wrangler.jsonc.

---

10. Development

```
# Run email-worker unit tests
bun test workers/email-worker/

# Start local dev server
hoox dev worker email-worker

# Deploy
hoox workers deploy
```

Config tracking: `wrangler.jsonc` is tracked in git.

---

11. Architecture Context

The email-worker is one of 10 workers in the Hoox service binding mesh:

```
flowchart LR
  classDef worker fill:#2196F3,color:#fff,stroke:#1565C0
  classDef storage fill:#FF9800,color:#fff,stroke:#E65100
  classDef external fill:#4CAF50,color:#fff,stroke:#2E7D32

  Mailgun{{"Mailgun Webhook"}}:::external --> EW["email-worker"]:::worker
  Direct("Direct JSON POST"):::external --> EW
  CFEmail("Cloudflare Email Routing"):::external --> EW
  EW --> TW["trade-worker"]:::worker
  EW --> AW["analytics-worker"]:::worker
  EW --> KV[("CONFIG_KV")]:::storage
```

- **Mailgun flow**: Inbound email → Mailgun webhook POST → email-worker `/webhook` → trade-worker
- **Direct flow**: Internal tooling → `POST /email-signal` (authenticated) → email-worker → trade-worker
- **Email Routing flow**: Cloudflare Email Routing → `email()` handler → email-worker → trade-worker
- **Analytics**: Every parsed signal sends `{ source, type, symbol, confidence }` to analytics-worker

All active signal ingestion is via webhook, direct POST, or Cloudflare Email Routing.

Next Steps

- **trade-worker Profile** — How parsed signals become execution orders
- **analytics-worker Profile** — Signal tracking and observability
- **Architecture Overview** — Full system architecture and data flow

19 · WORKERS

WEB3-WALLET-WORKER ISOLATE PROFILE

Comprehensive engineering specification for the Hoox EVM & DeFi On-Chain Wallet Worker, covering mnemonics, private keys, and smart contract signature routing.

The **`web3-wallet-worker`** is the on-chain gateway of the Hoox trading ecosystem. Running as an isolated private micro-worker, this service is responsible for securely managing EVM mnemonics and private keys (bound as encrypted Workers Secrets), querying multi-chain gas limits and token balances, executing native/ERC-20 transfers, and signing smart contract swap payloads (e.g. Uniswap/1inch routers) via JSON-RPC providers.

---

1. Declared Wrangler Configurations & Bindings

The `web3-wallet-worker` does not expose a public URL, communicating internally via V8 Service Bindings. Its `wrangler.jsonc` specifies:

```
{
  "name": "web3-wallet-worker",
  "main": "src/index.ts",
  "compatibility_date": "2026-05-19",
  "compatibility_flags": ["nodejs_compat"],
  "account_id": "debc6545e63bea36be059cbc82d80ec8",
  "vars": {
    "DEFAULT_CHAIN": "ethereum",
  },
  "kv_namespaces": [
    {
      "binding": "CONFIG_KV",
      "id": "c5917667a21745e390ff969f32b1847d",
    },
  ],
  "secrets": [
    "INTERNAL_KEY_BINDING",
    "WALLET_MNEMONIC_SECRET",
    "WALLET_PK_SECRET",
    "RPC_PROVIDER_URL",
  ],
}
```

---

2. Environmental Variables & Encrypted Secrets

- **`WALLET_PK_SECRET`**: Encrypted private key used for single-account execution.
- **`WALLET_MNEMONIC_SECRET`**: Encrypted 12 or 24-word HD wallet seed phrase used to derive multiple accounts.
- **`RPC_PROVIDER_URL`**: High-availability HTTP Ethereum / EVM RPC provider (e.g., Infura, Alchemy, or QuickNode).
- **`INTERNAL_KEY_BINDING`**: Shared key used to validate calls from internal compute nodes.

Local Development Mocking (`.dev.vars`)

```
WALLET_PK_SECRET=0x0123456789abcdef0123456789abcdef0123456789abcdef0123456789abcdef
WALLET_MNEMONIC_SECRET="abandon abandon abandon abandon abandon abandon abandon abandon abandon a
RPC_PROVIDER_URL=http://localhost:8545
INTERNAL_KEY_BINDING=dev_shared_internal_security_key
```

---

3. Internal REST API Specification

> **Note:** For the canonical endpoint directory with full request/response examples across all workers, see `/docs/devops/api/endpoints` .

A. Execute On-Chain Transaction

- **Endpoint**: `/process`
- **Method**: `POST`
- **Headers**: `X-Internal-Auth-Key: <INTERNAL_KEY_BINDING>`
- **JSON Payload**:

```
{
  "requestId": "9b1deb4d-3b7d-4bad-9bdd-2b0d7b3dcb6d",
  "payload": {
    "action": "sendTransaction",
    "chain": "arbitrum",
    "to": "0x6b175474e89094c44da98b954eedeac495271d0f",
    "value": "0.05",
    "data": "0xa9059cbb000000000000000000000000...",
    "gasLimit": 100000
  }
}
```

- **Success Response (200 OK)**:

```
{
  "success": true,
  "result": {
    "txHash": "0x53a9284739ebfd10482da73cbcfd10482da73cbcfd10482da73cbcfd10482ab",
    "nonce": 142,
    "gasUsed": 64205,
    "effectiveGasPrice": "24000000000"
  },
  "error": null
}
```

B. Query Token Balance

- **Endpoint**: `/process`
- **Method**: `POST`

- **JSON Payload**:

```
{
  "requestId": "9b1deb4d-3b7d-4bad-9bdd-2b0d7b3dcb6d",
  "payload": {
    "action": "getBalance",
    "chain": "polygon",
    "address": "0x6b175474e89094c44da98b954eedeac495271d0f",
    "tokenAddress": "0xc2132d05d31c914a87c6611c10748aeb04b58e8f"
  }
}
```

- **Success Response (200 OK)**:

```
{
  "success": true,
  "result": {
    "balance": "1485.50",
    "symbol": "USDT",
    "decimals": 6
  },
  "error": null
}
```

---

4. On-Chain Security Best Practices

Operating hot wallets on public blockchain networks introduces extreme security vectors:

- **Harden Private Keys**: **Never** write keys to wrangler config files or print them in telemetry logs. Always provision keys via encrypted Cloudflare Secrets.
- **Gas Price Limit Traps**: To prevent severe loss during network congestion or flash crashes, the worker enforces a **gas limit trap**—if current network gas price exceeds your KV configured limit (`web3:max_gas_price_gwei`), transactions are dropped before signing to prevent massive fee consumption.
- **Isolate Access**: All calls must originate internally via Service Bindings. The wallet worker does not bind to public ports, meaning external scrapers cannot send raw transaction payloads or try to brute-force auth codes.

Testing on-chain logic locally? Use the Docker runtime stack (`hoox dev start --runtime docker`) to launch an isolated Hardhat/Anvil node container and test private wallet swaps on a simulated local EVM fork safely!

Next Steps

- **trade-worker Profile** — Review how execution orders route transactions to EVM wallet nodes.
- **D1 Database Operations** — Manage your SQLite schemas and sync positions logs.

20 · WORKERS

ANALYTICS-WORKER ISOLATE PROFILE

Comprehensive engineering specification for the Hoox Analytics Observability Worker, covering Analytics Engine datasets, latency metrics, and dashboard API queries.

The **`analytics-worker`** is the observability engine of the Hoox trading platform. Deployed as a private internal microservice, it aggregates time-series metrics, database query latencies, execution performance ratios, and API status codes across all V8 isolates. By translating incoming events and writing them to **Cloudflare Analytics Engine**, it provides the backend telemetry used to draw live charts in the Next.js Dashboard.

---

1. Declared Wrangler Configurations & Bindings

The `analytics-worker` binds directly to Cloudflare's **Analytics Engine** dataset and does not expose any public endpoints:

```
{
  "name": "analytics-worker",
  "main": "src/index.ts",
  "compatibility_date": "2026-04-17",
  "compatibility_flags": ["nodejs_compat"],
  "account_id": "debc6545e63bea36be059cbc82d80ec8",
  "analytics_engine_datasets": [
    {
      "binding": "ANALYTICS_ENGINE",
      "dataset": "hoox-analytics",
    },
  ],
  "placement": { "mode": "smart" },
  "observability": { "enabled": true, "head_sampling_rate": 1 },
}
```

> No `CLOUDFLARE_API_TOKEN` or `CLOUDFLARE_ACCOUNT_ID` are in `secrets` — these are set as `vars` in `wrangler.jsonc` and used for the Cloudflare SQL API query path.

---

2. Environmental Variables & Encrypted Secrets

- **`CLOUDFLARE_API_TOKEN`** (var): A secure token with `Account.Analytics` read permissions, allowing the worker to query stored datasets via the Cloudflare SQL API.
- **`CLOUDFLARE_ACCOUNT_ID`** (var): Your Cloudflare Account ID used in SQL API requests.

---

3. Internal REST API Specification

> **Note:** For the canonical endpoint directory with full request/response examples across all workers, see

> `/docs/devops/api/endpoints`.

All endpoints are reachable via Cloudflare Service Bindings only (no public URL). Each endpoint validates its payload with a Zod schema using `.strict()` to reject unknown fields.

A. Track Trade Event

- **Endpoint**: `POST /track/trade`
- **Payload**:

```
{
  "payload": {
    "exchange": "binance",
    "symbol": "BTCUSDT",
    "action": "LONG",
    "quantity": 0.5
  },
  "result": { "success": true },
  "latencyMs": 1200
}
```

- **Response**: `{ "success": true }`

B. Track API Call Event

Invoked by other workers (like `hoox` or `trade-worker`) immediately upon completing an action.

- **Endpoint**: `POST /track/api-call`
- **Payload**:

```
{
  "worker": "trade-worker",
  "endpoint": "/api/v3/order",
  "latencyMs": 250,
  "success": true
}
```

Under-the-Hood Analytics Engine Write

The worker formats and writes a time-series data point to the `hoox-analytics` dataset:

```
env.ANALYTICS_ENGINE.writeDataPoint({
  blobs: ["api-call", "trade-worker", "success", "/api/v3/order", ""],
  doubles: [250, 0, 0],
  indexes: ["550e8400-e29b-41d4-a716-446655440000"],
});
```

- **Response (200)** : `{ "success": true }`

C. Track Worker Performance

- **Endpoint**: `POST /track/worker-perf`
- **Payload**:

```
{
  "data": {
    "worker": "trade-worker",
    "requests": 100,
    "errors": 0,
    "duration": 25000
  }
}
```

D. Track Signal

- **Endpoint**: `POST /track/signal`
- **Payload**:

```
{
  "data": {
    "source": "agent-worker",
    "type": "BUY",
    "symbol": "ETHUSDT",
    "confidence": 0.85
  }
}
```

E. Track Notification

- **Endpoint**: `POST /track/notification`
- **Payload**:

```
{
  "data": { "type": "trade_executed", "target": "telegram", "success": true }
}
```

F. Health

- **Endpoint**: `GET /health`
- **Response**: `{ "status": "ok", "worker": "analytics-worker" }`

Query Methods (Exported Functions)

The analytics worker exports query functions that are called via Service Bindings (not HTTP). Each builds a SQL query and executes it against the Cloudflare Analytics Engine SQL API:

Function	Parameters	Purpose
`getTradeMetrics(timeRange)`	ISO date range	Trade counts by exchange
`getTradesByExchange(exchange, limit)`	Exchange name	Recent trades for an exchange
`getTradeSuccessRate(timeRange?)`	Optional ISO date	Overall trade success rate
`getWorkerPerformance(worker, timeRange?)`	Worker name	Request counts and latency
`getApiCallStats(exchange?)`	Optional exchange	API call statistics
`getSignalOutcomes(timeRange?)`	Optional ISO date	Signal analysis by source

All SQL queries are built via `sanitizeQueryInputs()` which validates parameters using allowlists and regex to prevent SQL injection.

---

4. Dashboard Visual Integrations

The metrics written to `hoox-analytics` are queried by the dashboard to render:

1. **System Health Indicators**: Real-time error spikes trigger warning banners in under 2 seconds.
2. **Isolate Latency Charts**: Visual line graphs showing V8 processing speeds vs exchange API transit speeds.
3. **Volume Load Heatmaps**: Visualizes peak trading hours and signal traffic volumes globally.

Testing analytics locally? Local Wrangler dev automatically intercepts `writeDataPoint` calls and logs the parsed Blobs and Doubles straight to your terminal standard output, ensuring easy debugging!

Next Steps

- **System Observability Guides** — Deepen your understanding of time-series logging and metrics.
- **trade-worker Profile** — Review how execution latency details are generated and offloaded.

21 · WORKERS

REPORT-WORKER ISOLATE PROFILE

Comprehensive engineering specification for the Hoox PDF Report Worker, covering Cloudflare Browser Rendering APIs, Puppeteer Chrome automation, and R2 bucketing.

The **`report-worker`** is the automated document compiler of the Hoox trading platform. Deployed as a scheduled cron isolate, this worker runs twice daily (at 06:00 and 18:00 UTC) to pull D1 transactional stats, construct a highly styled HTML5 portfolio report, spin up a serverless **Cloudflare Browser Rendering** Chrome instance to compile the page into a PDF buffer, offload the file to R2, and push a private download link to your Telegram.

---

1. Declared Wrangler Configurations & Bindings

The `report-worker` mounts storage buckets, notification bindings, and is triggered by Cloudflare's background cron engine:

```
{
  "name": "report-worker",
  "main": "src/index.ts",
  "compatibility_date": "2026-05-19",
  "compatibility_flags": ["nodejs_compat"],
  "account_id": "debc6545e63bea36be059cbc82d80ec8",
  "placement": {
    "mode": "smart",
  },
  "triggers": {
    "crons": ["0 6 * * *", "0 18 * * *"], // Runs twice daily
  },
  "r2_buckets": [
    {
      "binding": "REPORTS_BUCKET",
      "bucket_name": "trade-reports",
    },
  ],
  "services": [
    { "binding": "D1_SERVICE", "service": "d1-worker" },
    { "binding": "TELEGRAM_SERVICE", "service": "telegram-worker" },
  ],
  "secrets": [
    "INTERNAL_KEY_BINDING",
    "CF_API_TOKEN", // Required to call Browser Rendering REST API
  ],
}
```

---

2. Environmental Variables & Encrypted Secrets

- **`CF_API_TOKEN`**: Your Cloudflare API Token with `Account.Browser Rendering` and `Account.R2` write permissions.
- **`INTERNAL_KEY_BINDING`**: Shared key used to validate calls from other V8 isolates.

---

3. Browser Rendering & PDF Print Pipeline

When the Cron schedule triggers, `report-worker` runs the following programmatic pipeline:

Step 1: Data Aggregation & HTML Compilation

The worker pings `D1_SERVICE` to retrieve P&L, win rate, and total daily fees, inserting the metrics into a styled HTML5 template utilizing Tailwind CSS and CSS Grid styling:

```
<div class="card">
  <h2>Daily P&L</h2>
  <p class="pnl-green">+$1,234.50</p>
</div>
```

Step 2: Cloudflare Chrome Isolate Print

Pushes the HTML template to Cloudflare's headless Chrome rendering pool via the REST API:

```
const response = await fetch(
  `https://api.cloudflare.com/client/v4/accounts/${env.ACCOUNT_ID}/browser-rendering/pdf`,
  {
    method: "POST",
    headers: {
      Authorization: `Bearer ${env.CF_API_TOKEN}`,
      "Content-Type": "application/json",
    },
    body: JSON.stringify({
      html: htmlContent,
      options: {
        format: "A4",
        printBackground: true,
        margin: { top: "1cm", bottom: "1cm", left: "1cm", right: "1cm" },
      },
    }),
  }
);
```

Step 3: R2 Storage & Expiration

1. The API compiles the page and returns a raw PDF binary stream.
2. The worker uploads the stream to the `REPORTS_BUCKET` R2 storage bucket using a unique, date-hashed key: `reports/daily-pnl-2026-05-19.pdf`
3. A lifecycle rule on the R2 bucket automatically purges reports older than 30 days to maintain storage cleanliness.

Step 4: Dispatch Telegram Alert

Calls `TELEGRAM_SERVICE` to send the link:

```
await env.TELEGRAM_SERVICE.fetch("https://telegram-worker/alert", {
  method: "POST",
  headers: { "X-Internal-Auth-Key": env.INTERNAL_KEY_BINDING },
  body: JSON.stringify({
    chatId: env.TELEGRAM_CHAT_ID_DEFAULT,
    message:
      "<b> Daily P&L PDF Report Compiled!</b>\nDownload link: <a href='https://reports.hoox.examp
  }),
});
```

If `CF_API_TOKEN` is not configured, the worker gracefully fails by compiling a rich text-only P&L summary and pushing it directly via Telegram instead of generating a PDF, guaranteeing continuous operations without hard crashes.

Next Steps

- **telegram-worker Profile** — Review push alert configurations and webhook tunnels.
- **D1 Database Operations** — Manage Drizzle schemas and execute custom queries.

22 · WORKERS

PINE-WORKER ISOLATE PROFILE

Comprehensive engineering specification for the pine-worker Pine Script evaluator, covering ANTLR4 parsing, AST evaluation, strategy event emission, and cross-worker trade forwarding.

The **`pine-worker`** is the strategy evaluation engine of the Hoox trading platform. Deployed as a compute isolate, it runs TradingView Pine Script strategy scripts against historical or live OHLCV data and emits structured trade events to `trade-worker` via a V8 Service Binding.

> **Status:** worker design is documented here; the companion **`hoox pine` CLI group is upcoming** and not part of the current CLI release.

---

1. Declared Wrangler Configurations & Bindings

The `pine-worker` communicates internally via a Service Binding to `trade-worker` and reads OHLCV data from an R2 bucket. Its `wrangler.jsonc` maps out storage, service, and environment hooks:

```
{
  "name": "pine-worker",
  "main": "src/index.ts",
  "compatibility_date": "2026-06-08",
  "compatibility_flags": ["nodejs_compat"],
  "placement": { "mode": "smart" },
  "observability": {
    "enabled": true,
    "head_sampling_rate": 0.1,
    "logs": { "enabled": true, "head_sampling_rate": 1, "persist": true, "invocation_logs": true
  },
  "vars": {
    "TRADE_WORKER_NAME": "trade-worker"
  },
  "services": [
    { "binding": "TRADE_SERVICE", "service": "trade-worker" }
  ],
  "r2_buckets": [
    { "binding": "OHLCV_DATA", "bucket_name": "pine-worker-ohlcv", "preview_bucket_name": "pine-w
  ]
}
```

Secrets

Secret	Purpose
`INTERNAL_KEY_BINDING`	Shared internal auth key for cross-worker communication

Aliases

The worker uses `wrangler.jsonc` alias mappings to resolve the `@jango-blockchained/hoox-shared` monorepo package at build time — no npm link or workspace protocol needed at deploy time.

2. Architecture: Parser → AST → Evaluator → Events

The `pine-worker` processes Pine Script in four stages, following the **pynescript** reference implementation:

A. Parser (ANTLR4)

An ANTLR4-generated TypeScript parser (`PinescriptLexer.g4` / `PinescriptParser.g4`) tokenizes and parses Pine Script source into an ANTLR parse tree. The grammar supports the full TradingView Pine Script surface including expressions, statements, and type annotations.

```
# Regenerate parser from grammar
bun run gen:parser
```

B. AST Builder (Visitor + Zod)

A generated visitor (`PinescriptParserVisitor`) walks the ANTLR parse tree and builds a Zod-validated typed AST. Every AST node is defined with `.strict()` Zod schemas, catching malformed or unsupported constructs at the boundary before they reach the evaluator.

C. Evaluator

The evaluator is a composition-based engine that resolves AST nodes into Pine Series values. It handles:

- **Expression evaluation** — arithmetic, logical, comparison, ternary, subscript, function calls
- **Statement execution** — variable declarations (with history-based series semantics), if/else, for loops
- **Builtin resolution** — namespaced builtins (ta.*, math.*, strategy.*, etc.) registered in the builtin registry
- **Series semantics** — each variable maintains per-bar history, with `close[1]`-style offset lookups

D. Runtime (Per-Bar Loop)

The `Runtime` class orchestrates the full pipeline:

1. Parse the script via ANTLR
2. Build the AST via visitor
3. For each OHLCV bar (chronologically): set `open`, `high`, `low`, `close`, `volume`, `time`, and `bar_index` state
4. Evaluate the AST against the current bar's state
5. Collect strategy events (`strategy.entry`, `strategy.exit`, `strategy.close`)

```

const rt = new Runtime({ symbol: "BTCUSDT" });
const result = await rt.run("strategy.entry('long', strategy.long)", ohlcvBars);
```

E. Trade Forwarding

When `forward: true` is passed in the request, trade events are forwarded to `trade-worker` via the `TRADE_SERVICE` Service Binding. Each event is serialized into a `WebhookPayload` and sent to `trade-worker`'s `/webhook` endpoint with `INTERNAL_KEY_BINDING` authentication.

F. Chart Export

Every `/run` response includes a per-bar `sheet` with OHLCV columns plus all `plot()` / `hline()` values. The response can be requested in JSON (default) or CSV format — mirroring TradingView's "Export chart data" feature.

---

3. Internal REST API Specification

A. Execute Strategy

- **Endpoint**: `/run`
- **Method**: `POST`
- **Headers**: `X-Internal-Auth-Key: <INTERNAL_KEY_BINDING>`
- **JSON Payload**:

```
{
  "script": "indicator('My Strategy')\nplot(close)",
  "symbol": "BTCUSDT",
  "timeframe": "1d",
  "bars": 500,
  "forward": false,
  "sheet": true,
  "format": "json"
}
```

Payload fields:

Field	Type	Required	Default	Description
`script`	string		—	Pine Script source (max 100,000 chars)
`ohlcv`	OHLCVBar[]		—	Explicit bars (max 10,000). If omitted, fetched from DataProvider
`symbol`	string		`BTCUSDT`	Trading pair
`timeframe`	enum		`1d`	Bar size: `1m` , `5m` , `15m` , `1h` , `4h` , `1d` , `1w`
`bars`	integer		`500`	Number of bars to fetch (max 10,000)
`forward`	boolean		`false`	Forward trade events to `trade-worker`
`sheet`	boolean		`true`	Include per-bar OHLCV + plot data in response
`format`	enum		`json`	Response format: `json` or `csv`

- **Success Response (200 OK)**:

```
{
  "events": [
    {
      "type": "strategy.entry",
      "id": "long",
      "direction": "long",
      "bar_index": 50,
      "price": 68425.5
    }
  ],
  "bars": 500,
  "script_id": "abc123",
  "run_id": "def456",
  "symbol": "BTCUSDT",
  "timeframe": "1d",
  "sheet": {
    "bars": [
      {
        "plots": {
          "close": 68425.5,
          "sma20": 68100.3
        },
        "time": 1720000000000,
        "open": 68200.0,
        "high": 68500.0,
        "low": 68150.0,
        "close": 68425.5,
        "volume": 1234.5
      }
    ]
  },
  "export": {
    "columns": ["time", "open", "high", "low", "close", "volume", "close", "sma20"],
    "data": [[1720000000000, 68200.0, 68500.0, 68150.0, 68425.5, 1234.5, 68425.5, 68100.3]]
  }
}
```


When `format: "csv"`, the response is a `text/csv` download with `Content-Disposition: attachment`.

- **Error Response (404)** — no data for symbol/timeframe:


```
{
  "error": "No data for BTCUSDT 1d. Run `bun run scripts/download-data.ts --symbol BTCUSDT --tf 1
}
```


B. Health Check

- **Endpoint**: `/health`
- **Method**: `GET`
- **Response (200 OK)**:


```
{
  "status": "ok",
  "worker": "pine-worker"
}
```

4. Security Middleware Stack

Every request passes through a layered middleware pipeline (fail-closed):

1. **Top-level error boundary** — unhandled exceptions return sanitized 500 responses (no stack traces, no internal paths)
2. **Auth** — `requireInternalAuth` validates `X-Internal-Auth-Key` header with timing-safe comparison against `INTERNAL_KEY_BINDING`
3. **CSRF** — Origin vs Host header check on mutations. Service-binding calls (no Origin header) pass through
4. **Zod validation** — `.strict()` schema on `/run` rejects unknown fields, returns sanitized error messages
5. **Security headers** — `wrapWithSecurityHeaders` applies CSP, X-Content-Type-Options, etc. on every response
6. **Output sanitization** — `createJsonResponse` strips Error instances and stack traces from response bodies

---

5. DataProvider Architecture

The worker resolves OHLCV data through a layered provider chain:

```
LiveDataProvider            ← always wraps, adds live Binance bars
  └─ CompositeDataProvider  ← tries R2, then local files
       ├─ R2DataProvider    (production — reads from pine-worker-ohlcv R2 bucket)
       └─ LocalFileDataProvider  (dev — reads from data/ directory)
```

- **Local dev**: Use `bun run download:data` to populate `data/{SYMBOL}/{TIMEFRAME}/` with compressed JSON Lines files
- **Production**: Upload data to the `pine-worker-ohlcv` R2 bucket in the same directory structure

---

6. Parity Testing

The worker's correctness is verified against the Python reference implementation in pynescript. The parity corpus under `pynescript/tests/fixtures/parity/` defines the contract: every `.pine` script must produce identical event lists when run through both evaluators.

```
bun test              # Full test suite including parity tests
bun test --watch      # Watch mode for development
bun test --coverage   # Coverage report
```

---

7. CLI Scripts

> **Upcoming:** the `hoox pine` command group is **not shipped** in the current CLI release. Use `bun run` scripts inside `workers/pine-worker` until the group lands. TradingView **webhooks** (external Pine Script alerts → gateway) are available today and do not require `hoox pine`.

Worker-local scripts (from `workers/pine-worker`):

Download OHLCV Data

```
bun run download:data --symbol ETHUSDT --tf 1h --days 30
```

Fetches from Binance public API, stores as compressed JSON Lines: `data/{SYMBOL}/{TIMEFRAME}/{YYYY}.jsonl.gz`.

Export Chart Data

```
bun run export:chart ../grid.pine BTCUSDT 5m 100 --csv
```

Run Backtest

```
bun run run:backtest ../grid.pine BTCUSDT 5m 864
```

Bundle Libraries

```
bun run bundle:libraries
```

Planned CLI surface (not yet released)

```
# hoox pine download | backtest | export | bundle  — upcoming
```

Related Workers & Services

- **trade-worker** — Trade execution engine; receives forwarded events via `TRADE_SERVICE` binding
- **hoox Gateway** — Edge gateway that routes external requests to `pine-worker`
- **pynescript (Python reference)** — Reference evaluator for parity testing
- **hoox-setup** — Monorepo root

For development, always run `bun run download:data` first to populate local OHLCV data. Without it, `/run` requests that don't include explicit `ohlcv` bars will return a 404 with instructions.

23 · WORKERS

NEXT.JS DASHBOARD & OPENNEXT ISOLATE

Comprehensive engineering specification for the Next.js 16 Hoox Dashboard, covering Turbopack builds, OpenNext adapters, and Cloudflare Workers hosting.

The **Hoox Dashboard** is the web-based command center of the trading platform. Rather than running on a traditional server, the dashboard is a **Next.js 16 application** compiled using the **OpenNext adapter** and hosted natively on Cloudflare Workers. This edge-first layout allows for real-time portfolio monitoring, dynamic form-based settings editing, and interactive kill-switch controls at microsecond speeds.

Next.js to Cloudflare Edge Architecture

When deploying the dashboard, the monorepo uses `@opennextjs/cloudflare` to convert standard Node.js server-side features into Edge-compliant JavaScript:

```
graph TD
    Src["Next.js 16 App Source"] -->|bun run opennext:build| Bundler["Turbopack / Webpack Bundler

    Bundler -->|Server-Side Rendering| WorkerJS[".open-next/worker.js"]
    Bundler -->|Static Assets| Assets[".open-next/assets/"]

    WorkerJS -->|wrangler deploy| V8["Cloudflare V8 Isolate"]
    Assets -->|ASSETS binding| CDN["Cloudflare Global CDN"]

    style Src fill:#CCCCCC,stroke:#3b82f6,stroke-width:2
    style Bundler fill:#CCCCCC,stroke:#f59e0b,stroke-width:2
    style WorkerJS fill:#CCCCCC,stroke:#10b981,stroke-width:1
    style Assets fill:#CCCCCC,stroke:#10b981,stroke-width:1
    style V8 fill:#CCCCCC,stroke:#ef4444,stroke-width:2
    style CDN fill:#CCCCCC,stroke:#ef4444,stroke-width:2
```

- **Server Isolate** (`.open-next/worker.js`): Handles Server-Side Rendering (SSR), React Server Components (RSC) hydration, API routes, and cookies encryption.
- **Static Assets** (`.open-next/assets/`): Houses all static assets, which are bound to the worker via **ASSETS bindings** to serve static pages at sub-millisecond CDN latency.

---

1. Declared Wrangler Configurations & Bindings

The dashboard's `wrangler.jsonc` maps out its internal service links to extract D1 databases metrics and agent status updates:

```
{
  "name": "hoox-dashboard",
  "main": ".open-next/worker.js",
  "compatibility_date": "2026-05-19",
  "compatibility_flags": ["nodejs_compat"],
  "account_id": "debc6545e63bea36be059cbc82d80ec8",
  "assets": {
    "directory": ".open-next/assets",
    "binding": "ASSETS",
  },
  "kv_namespaces": [
    {
      "binding": "CONFIG_KV",
      "id": "c5917667a21745e390ff969f32b1847d",
    },
  ],
  "services": [
    { "binding": "D1_SERVICE", "service": "d1-worker" },
    { "binding": "AGENT_SERVICE", "service": "agent-worker" },
    { "binding": "TELEGRAM_SERVICE", "service": "telegram-worker" },
  ],
  "vars": {
    "D1_SERVICE_URL": "https://d1-worker.hoox.example.com",
    "AGENT_SERVICE_URL": "https://agent-worker.hoox.example.com",
    "TELEGRAM_SERVICE_URL": "https://telegram-worker.hoox.example.com",
  },
  "secrets": [
    "DASHBOARD_USER",
    "DASHBOARD_PASS",
    "SESSION_SECRET",
    "INTERNAL_KEY_BINDING",
  ],
}
```

---

2. Environmental Variables & Encrypted Secrets

- **`DASHBOARD_USER`** & **`DASHBOARD_PASS`**: Administrative credentials required to access the visual panel.
- **`SESSION_SECRET`**: A secure 32-character encryption key used to cryptographically sign session cookies at the edge.
- **`INTERNAL_KEY_BINDING`**: Shared key used to validate calls to the `d1-worker` and `agent-worker` service bindings.

Local Development Mocking (`.env.local`)

Create a gitignored `.env.local` file inside `workers/dashboard/` for local Next.js runs:

```
DASHBOARD_USER=admin
DASHBOARD_PASS=admin_dev_passkey_183
SESSION_SECRET=abandon_abandon_abandon_abandon_32
```

---

3. Schema-Driven Settings System

To allow operators to extend the platform's settings without touching React code, the dashboard implements a **schema-driven configuration manager** parsed dynamically from `config.schema.json`:

```
{
  "sections": [
    {
      "id": "risk",
      "title": "Risk Management",
      "fields": [
        {
          "key": "trade:max_daily_drawdown_percent",
          "label": "Max Daily Drawdown (%)",
          "type": "number",
          "default": 5,
          "category": "risk"
        }
      ]
    }
  ]
}
```

Form Submission Flow

1. When you save the settings form, the Next.js server actions intercept the payload.
2. The server action iterates over the JSON schema keys and writes the values directly to the bound `CONFIG\_KV` namespace.
3. The changes propagate globally to your edge executors in under 10 seconds.

---

4. Production Build & Rollout Runbook

To compile and deploy the Next.js dashboard to Cloudflare:

```
# 1. Navigate to the dashboard directory
cd workers/dashboard

# 2. Build the application using OpenNext
bun run opennext:build

# 3. Deploy the compiled bundles to Cloudflare Workers
bun run opennext:deploy
```

Framer Motion components and interactive visual elements used in Next.js pages **must** declare the `'use client'` directive at the top of the file. Under OpenNext, client-side pages cannot export metadata directly—move metadata definitions to separate `metadata.ts` files!

Security Headers

The dashboard applies **7 security headers** on every response via the shared `@jango-blockchained/hoox-shared/middleware` `secureHeaders()` factory:

Header	Value
`X-Content-Type-Options`	`nosniff`
`X-Frame-Options`	`DENY`
`X-XSS-Protection`	`1; mode=block`
`Referrer-Policy`	`strict-origin-when-cross-origin`
`Permissions-Policy`	All features disabled
`Strict-Transport-Security`	`max-age=31536000; includeSubDomains`
`Content-Security-Policy`	`default-src 'self'`

Headers are applied via the `withSecurityHeaders()` wrapper in `src/middleware.ts`, covering **all** return paths including static files, login pages, CSRF errors, and auth failures.

Next Steps

- **Security Testing & Hardening** — Auth hardening, security tests, and CI/CD scanning.
- **agent-worker Profile** — Review AI chat streaming and risk evaluation structures.
- **d1-worker Profile** — Check REST schemas that feed database metrics to the dashboard.

LOCAL DEVELOPMENT SETUP

Detailed operational guide for local development setups, covering Wrangler CLI native dev parameters, Docker Compose profiles, and local .dev.vars mocking.

Hoox provides a comprehensive local development workspace designed to emulate your production Cloudflare edge environment. Developers can choose between running microservices natively on local Wrangler V8 isolates or inside completely isolated, containerized Docker stacks, with real-time log tailing and TUI diagnostics.

1. Dev Runtime Selection: Native vs. Docker

The `hoox dev start` command orchestrates the boot sequence for all enabled workers and supports two execution runtimes:

Runtime Mode	Boot Command	Latency / Hot-Reload	Isolation Level	Ideal Use Case
Native	`wrangler dev`	**< 10ms**	Low (shares local host ports)	High-speed script tweaking, rapid feature iterations.
Docker	`docker compose up`	**~100ms**	High (isolated Linux containers)	Testing full integrations, database migrations, queue failovers.

The Guided Startup Flow

When you run `hoox dev start`:

1. **Wrangler Version Verification**: Probes your global/local Wrangler package. If Wrangler is outdated, it prompts an advisory upgrade warning. The warning shows the current vs minimum version and offers a one-line update command.
2. **Docker Environment Probing**: Checks if the Docker daemon is active and `docker-compose.yml` is present in the workspace.
3. **Runtime Preference Lock**: If both runtimes are available, the CLI prompts you to select your preference. This choice is written to your `wrangler.jsonc` file under the dev runtime block, bypassing future prompts.

```
# Override the saved runtime preference on the fly
hoox dev start --runtime native
hoox dev start --runtime docker
```

2. Docker Compose Orchestration & Profiles

For full stack containerized development, Hoox divides operations into three Docker Compose profiles in your root `docker-compose.yml`:

```
# Profile 1: All background workers (9 services — no dashboard UI)
docker compose --profile workers up

# Profile 2: Dashboard + its transitive worker dependencies (5 services:
# hoox, d1-worker, trade-worker, agent-worker, dashboard).
# trade-worker is pulled in because agent-worker depends on it.
docker compose --profile dashboard up

# Profile 3: Full Stack — all 10 services
docker compose --profile full up
```

Only `hoox` (8787) and `dashboard` (8794) are published to the host. The remaining workers are reachable only via service bindings on the `hoox-net` bridge network — the same topology as production Cloudflare Workers Service Bindings.

---

3. Local Port Mapping Registry

In the local environment, each V8 dev instance mounts to a dedicated host port. Ensure no other applications are occupying these ports before launching:

Microservice	Default Port	Local Binding URL	Purpose
hoox	8787	http://localhost:8787	Ingress Gateway
trade-worker	8789	http://localhost:8789	Order Execution Engine
telegram-worker	8791	http://localhost:8791	Notifications Hub
d1-worker	8792	http://localhost:8792	Relational SQLite Manager
web3-wallet	8793	http://localhost:8793	On-Chain DeFi Node
dashboard	8794	http://localhost:8794	Next.js Dashboard SSR
agent-worker	8795	http://localhost:8795	Cron Risk Monitor
email-worker	8796	http://localhost:8796	Email Signal Parser
report-worker	8797	http://localhost:8797	PDF Portfolio Generator
analytics-worker	8798	http://localhost:8798	Analytics & Reporting

---

4. Local Environmental Mocking (`.dev.vars`)

Because production secrets are locked inside Cloudflare's secured key vaults, Wrangler dev uses local `.dev.vars` files located inside each worker's directory to mock API keys and credentials:

```
# 1. Copy the secure template
cp workers/hoox/.dev.vars.example workers/hoox/.dev.vars

# 2. Inject local mock keys for testing
echo 'INTERNAL_KEY_BINDING="local_test_key"' >> workers/hoox/.dev.vars
```

`.dev.vars` files contain local plaintext keys and are excluded from git index tracking via `.gitignore`. **Never** remove these files from `.gitignore` or check them into your repository.

If local tests fail with `Time-Stamp Expired` errors when calling exchange APIs, check that your local machine's NTP system clock is synchronized: `sudo ntpdate pool.ntp.org` (or check date/time settings on macOS/Windows)!

---

5. Shell Helpers: Running Commands Across All Workers

The `scripts/helpers.sh` file provides a `wx()` function that runs any CLI command in every worker directory — useful for bulk operations like regenerating types, running typecheck, or installing dependencies.

Installation

```
# One-time install (adds to ~/.zshrc or ~/.bashrc)
bash scripts/install.sh

# Or source it manually in your current session
source scripts/helpers.sh
```

Usage

```
# Regenerate Wrangler types in every worker
wx "wrangler types"

# Run typecheck across all workers
wx "pnpm run typecheck"

# Inspect package names
wx "cat package.json | jq .name"

# List source files per worker
wx "ls -la src/"
```

Filtering

Use `WX_FILTER` to target specific workers by glob pattern:

```
# Only wrangler-based workers (excludes dashboard, hoox)
WX_FILTER='*-worker' wx "wrangler types"

# Single worker
WX_FILTER='trade-worker' wx "pnpm run typecheck"
```

Custom Path

Set `WX_WORKERS_DIR` if your workers live elsewhere:

```
WX_WORKERS_DIR=~/other-project/workers wx "some command"
```

6. Codebase Dependency Graph

Hoox includes a **function-level dependency graph extractor** that maps all exported symbols, imports, calls, type references, and service bindings across the entire monorepo.

```
# Regenerate graph.json and graph.dot
bun run graph
```

This runs `scripts/extract-graph.ts` (ts-morph) and scans 917 source files across all 14 workspaces. During extraction, a **live progress bar** shows each phase with elapsed time:

```
  Extracting exported declarations...
    [██▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒] 13%  exports: 1667 nodes            10.6s
  Extracting import relationships...
    [█████▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒] 25%  imports: 2524 edges             0.1s
  Extracting type references...
    [██████████▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒] 50%  type refs: 2536 edges           1.7s
  Extracting cross-file call expressions...
    [████████████▒▒▒▒▒▒▒▒▒▒▒▒▒] 63%  calls: 445 edges               9.7s
    [████████████████████████] 100% Done                          22.0s

⏱  Total time: 22.0s
```

The 8 extraction phases (exports → imports → extends/implements → type refs → calls → service bindings → workspace deps → filtering) complete in **~20–25s** on average.

Outputs

File	Contents
`graph.json`	997 nodes, 5536 edges — full machine-readable graph
`graph.dot`	Graphviz DOT with 14 workspace clusters, color-coded edges

Quick Render

```
# Install Graphviz if missing
# macOS: brew install graphviz
# Ubuntu: sudo apt install graphviz

# Render to SVG
dot -Tsvg graph.dot -o graph.svg && open graph.svg
```

Use `graph.json` as AI/LLM context for codebase-aware queries, or paste `graph.dot` into Edotor for instant browser visualization without installing Graphviz.

Next Steps

- **Testing Standards** — Run unit and integration tests using Bun's native test runner.
- **Debugging Runbook** — Debug local and remote isolates, tail logs, and trace queries.

25 · DEVELOPMENT

TESTING FRAMEWORK & QA STANDARDS

Detailed QA operations guide, covering Bun test suites, Miniflare integration testing, E2E cli lifecycles, and live Cloudflare resource audits.

To protect live capital and ensure robust order routing, Hoox mandates a rigorous testing pipeline. With money on the line, we verify every contract calculation, rate-limiting gate, and database query.

Our test suite is powered natively by **Bun's high-speed test runner**, comprising **~124 test files** and **~4,500 individual test assertions** split across five diagnostic layers.

The 5 QA Testing Layers

```
graph TD
    UT["Unit Tests

(~4,000 Assertions)"] -->|Next Layer| IT["Integration Tests

(34 Assertions)"]
    IT -->|Next Layer| ST["Security Tests

(40 Assertions)"]
    ST -->|Next Layer| E2E["E2E Smoke Tests

(5 Assertions)"]
    E2E -->|Next Layer| LT["Live Resource Tests

(77 Assertions)"]

    style UT fill:#CCCCCC,stroke:#3b82f6,stroke-width:2
    style IT fill:#CCCCCC,stroke:#10b981,stroke-width:2
    style ST fill:#CCCCCC,stroke:#f59e0b,stroke-width:2
    style E2E fill:#CCCCCC,stroke:#f59e0b,stroke-width:2
    style LT fill:#CCCCCC,stroke:#ef4444,stroke-width:2
```

Running Tests: CLI Commands

A. Core Platform Verification (Excluding Live)

```
# 1. Run all unit, integration, and E2E smoke tests in parallel
bun test

# 2. Run the suite and output a detailed V8 coverage report
bun test --coverage
```

B. Workspace-Specific Targeted Runs

To optimize developer feedback loops, you can target specific workspaces or workers:

```
# Run CLI commands tests only (packages/cli/)
bun run test:cli

# Run Terminal UI tests only (packages/tui/)
bun run test:tui

# Run shared helper tests only (packages/shared/)
bun run test:shared

# Run all edge workers unit tests (workers/*)
bun run test:workers

# Run a single specific test file with hot-reload watch mode
bun test workers/agent-worker/src/index.test.ts --watch
```

C. Security & Fuzz Testing

```
# Run all security tests (auth bypass, security headers, fuzz)
bun run test:security

# Run specific security test files
bun test tests/security/auth-bypass.test.ts
bun test tests/security/security-headers.test.ts
bun test tests/security/fuzz.test.ts
```

D. Advanced Integration & Live Runs

```
# Run Miniflare 3 gateway integration tests
bun run test:integration

# Run E2E CLI lifecycle smoke tests
bun run test:e2e

# Run live Cloudflare API integration tests (requires tests/live/.env credentials)
bun run test:live --jobs 1

# Run k6 performance/load tests (requires k6 CLI)
bun run test:load
```

Type-Safe Mocking Specifications (No `as any`)

To enforce strict TypeScript compiler safety, test files **must never** utilize `as any` to bypass types when mock-binding resources. Always cast stubs using `as unknown as Env`:

```
describe("trade-worker Gateway Router Mocking", () => {
  it("should securely mock internal service binding fetchers", async () => {
    // 1. Construct a type-safe mock environment structure
    const mockEnv = {
      INTERNAL_KEY_BINDING: "local_secret_token_183",
      TELEGRAM_SERVICE: {
        fetch: async (url: string, init?: RequestInit) => {
          // Verify auth headers exist
          const headers = init?.headers as Record<string, string>;
          if (headers["X-Internal-Auth-Key"] !== "local_secret_token_183") {
            return new Response(JSON.stringify({ success: false }), {
              status: 401,
            });
          }

          return new Response(
            JSON.stringify({
              success: true,
              messageId: 4829,
            }),
            { status: 200 }
          );
        },
      } as Fetcher,
    } as unknown as Env;

    // 2. Execute assertions
    const res = await mockEnv.TELEGRAM_SERVICE.fetch(
      "https://telegram-worker/alert",
      {
        method: "POST",
        headers: {
          "X-Internal-Auth-Key": "local_secret_token_183",
        },
      }
    );

    const data = await res.json();
    expect(res.status).toBe(200);
    expect(data.success).toBe(true);
    expect(data.messageId).toBe(4829);
  });
});
```

Continuous Integration Gates & Coverage Targets

Our GitHub Actions workflows enforce the following quality gates:

1. **TypeScript Type Safety**: All workspaces must compile without errors using `tsc --noEmit`.
2. **Coverage Thresholds**: The monorepo enforces a **minimum 80% coverage threshold** across all core execution paths (`packages/cli`, `packages/shared`, `workers/hoox`, `workers/trade-worker`).
3. **Dependency Audit**: `bun audit` runs after tests to detect known vulnerabilities (informational, doesn't block CI).
4. **Secret Scanning**: `gitleaks` scans all commits for hardcoded secrets on every push/PR (informational).

5. **CodeQL**: Weekly `security-and-quality` analysis for JavaScript/TypeScript.

```
# Check your local workspace coverage statistics
bun test packages/shared/ --coverage
```

Next Steps

- **Security Testing & Hardening** — Auth hardening, security tests, and CI/CD scanning.
- **Debugging Telemetry Runbook** — Learn how to trace active V8 memory, tail logs, and audit SQL execution.
- **Local Development Setup** — Configure Wrangler and Docker compose to run testbeds.

26 · DEVELOPMENT

EDGE DEBUGGING & TELEMETRY

Detailed system troubleshooting guide, covering local console logging, production wrangler tailing telemetry, and distributed trace ID tracking.

Debugging globally distributed serverless microservices presents unique operational challenges. Because code runs on Cloudflare's edge V8 isolates without a persistent server host, traditional step-through debugging is replaced by **structured logging, real-time edge telemetry tailing, and distributed tracing**.

This document is your engineering runbook for debugging and resolving runtime anomalies in the Hoox monorepo.

---

1. Real-Time Telemetry: Tailing Production Logs

Cloudflare's **`wrangler tail`** engine establishes an active, secure WebSocket stream straight from Cloudflare's global edge nodes to your terminal, printing every `console.log`, exception trace, and HTTP status code in real-time.

You can trigger this streaming telemetry via the Hoox CLI or Wrangler:

```
# A. Recommended: Stream live console logs for a specific worker via Hoox CLI
hoox logs tail trade-worker

# B. Stream gateway traffic in real-time
hoox logs tail hoox

# C. Alternatively, invoke Wrangler directly for custom configurations
npx wrangler tail hoox --config workers/hoox/wrangler.jsonc
```

---

2. Distributed Tracing: The `requestId` Standard

Because a single TradingView alert flows through multiple workers (Gateway → trade-worker → d1-worker → telegram-worker), locating a specific transaction log in a sea of concurrent entries is impossible without a unified trace parameter.

The RequestId Protocol

1. **Generation**: The `hoox` gateway generates a unique, cryptographically secure **UUIDv4** immediately upon receiving a webhook payload. This is set as the `requestId` in the transaction context.
2. **Propagation**: Every internal service binding call transmits this UUID as the `X-Request-Id` HTTP header.
3. **Structured Ingestion**: When logging error telemetry or writing trade fills to R2 and D1 databases, workers attach the `requestId` as a dedicated index column.
4. **Unified Auditing**: You can audit an entire transaction's lifecycle across all 9 workers by running a single database query filtering by the trace ID:

```
SELECT created_at, worker, message, severity
FROM system_logs
WHERE request_id = '9b1deb4d-3b7d-4bad-9bdd-2b0d7b3dcb6d'
ORDER BY created_at ASC
```

---

3. Operational Runbooks for Common Edge Failures

A. Symptom: `401 Unauthorized` on Internal Worker Calls

- **Diagnostics**: The calling worker gets a `401 Unauthorized` response when querying internal services like `d1-worker` or `trade-worker`.
- **Primary Root Cause**: The `INTERNAL_KEY_BINDING` secret does not match between the calling worker and the target worker.
- **Resolution**:
  1. Inspect the secret existence on both workers: `hoox secrets check`.
  2. If mismatched, re-inject the secret globally using one command:

```
hoox secrets set INTERNAL_KEY_BINDING "your_secure_shared_internal_key"
```

B. Symptom: `502 Bad Gateway` on Webhook Routes

- **Diagnostics**: An incoming signal returns a 502 error instantly.
- **Primary Root Cause**: A Service Binding target declared in the gateway's `wrangler.jsonc` has not been deployed yet.
- **Resolution**:
  1. Run the dependency check: `hoox check health`.
  2. Redeploy the entire stack in the correct dependency order:

```
hoox deploy all --auto
```

C. Symptom: `D1_ERROR: no such table`

- **Diagnostics**: Worker database transactions fail with table missing errors.
- **Primary Root Cause**: Relational SQLite schemas were never initialized on your production D1 instance.
- **Resolution**:
  1. Initialize database tables and run pending drizzle migrations:

```
hoox db apply --remote
hoox db migrate --remote
```

D. Symptom: KV Configuration Propagation Delays

- **Diagnostics**: You toggled a KV configuration (e.g. turned `trade:kill_switch = false`), but some incoming signals are still being rejected.
- **Primary Root Cause**: Cloudflare KV is eventually consistent. Updates can take up to 10 seconds to propagate to all global edge locations.
- **Resolution**: Wait 10-15 seconds for cache validation to settle globally. Do not trigger rapid test webhooks immediately after modifying configuration keys.

Next Steps

- **Testing Framework & QA Standards** — Run local unit and integration tests using Bun's native test runner.
- **Self-Healing & System Repair** — Run diagnostics, targeted component repairs, and full system rebuilds.

27 · DEPLOYMENT

PRODUCTION DEPLOYMENT RUNBOOK

High-integrity production deployment guide, covering API token configurations, custom routes mapping, and edge isolate hardening strategies.

Deploying the Hoox trading ecosystem to production requires absolute operational rigor. Because the platform executes real-world trades using private capital, every V8 isolate, environment binding, and routing domain must be locked down, optimized for speed, and insulated against external failures.

This runbook guides you through production prerequisites, manual and automated deployment commands, custom domain routing, and edge hardening strategies.

---

1. Production Rollout Pre-Flight Checklist

Before rolling out updates to Cloudflare's live edge networks:

- **API Token Scoping**: Confirm your active Cloudflare API Token inherits the strict minimal permission sets (`Account.D1: Edit`, `Account.KV: Edit`, `Account.Queues: Edit`, `Account.Workers: Edit`, and `Zone.DNS: Edit` if mapping custom routing paths).
- **Workspace Diagnostic Sweep**: Run the automated pre-flight check to verify that all git submodules, local dependencies, and TypeScript variables compile without warnings:

  ```
  hoox check prerequisites
  hoox test
  ```

- **Exchange Credentials Check**: Ensure you have created dedicated exchange API keys with **Withdrawal permissions turned OFF (disabled)** to enforce least-privilege security.

---

2. Production Rollout Invocations

The Hoox CLI automates the entire deployment sequence, compiling the shared libraries, deploying database schemas, synchronizing configuration manifests, and uploading workers in the correct dependency sequence:

```
# 1. Execute full sequenced deployment
hoox deploy all --auto

# 2. Deploy a single specific worker (e.g. after a trade execution update)
hoox deploy worker trade-worker
```

Dashboard OpenNext Compilation

```
# Build and deploy the Next.js Dashboard via OpenNext
hoox deploy dashboard
```

This command runs Turbopack, bundles server-side assets into `.open-next/worker.js`, maps static files to the `ASSETS` CDN binding, and uploads the dashboard to Cloudflare.

3. Custom Domain & Routing Mapping

By default, deployed workers are assigned subdomains under Cloudflare's shared domain (e.g., `https://hoox.alpha-trading.workers.dev`).

For production, it is highly recommended to map your public gateway and dashboard to a **custom domain** under your own Cloudflare zone. This reduces DNS lookup latencies and enables custom SSL and WAF configurations.

To map custom routes, add the `routes` array inside your worker's `wrangler.jsonc` file:

```
// In workers/hoox/wrangler.jsonc
{
  "routes": [
    {
      "pattern": "api.my-trading-empire.com/webhook",
      "custom_domain": true,
    },
  ],
}
```

Deploying this configuration automatically updates your Cloudflare DNS zone records, maps the domain to your V8 isolate, and registers a universal SSL certificate near-instantaneously.

---

4. Edge Isolate Hardening Strategies

To protect your live production deployments from attacks and latency spikes:

A. Enable Smart Placement

Verify that every execution worker contains the smart placement var inside its `wrangler.jsonc`. This shifts the CPU isolate geographically close to exchange servers (e.g. Frankfurt for Bybit, Tokyo for Binance):

```
"placement": {
  "mode": "smart"
}
```

B. Restrict CORS Origin Policies

The public `/webhook` route inside the `hoox` gateway must **only** accept POST requests from authorized endpoints. Disable all CORS headers to prevent cross-origin script injections from browsers.

C. Deploy WAF Rate Limit Traps

Use the Hoox CLI to provisionZone-level rate limiters on Cloudflare's global edge network, dropping packet floods before they load the V8 isolates:

```
# Provision a strict rate-limit rule (e.g., max 10 requests/minute to /webhook)
hoox waf configure --limit-requests
```

Made an emergency change? You don't need a full rebuild. If the change was a configuration setting or a trade symbol routing change, simply update the KV store: `hoox config kv set` . The update will propagate globally in under 10 seconds!

Next Steps

- **CI/CD Pipelines Reference** — Automate edge deployments using GitHub Actions.
- **Observability & Time-Series Monitoring** — Stream console logs and check analytics datasets.

28 · DEPLOYMENT

CI/CD PIPELINE AUTOMATION

How to automate the Hoox deployment pipeline using GitHub Actions, including code style checks, TypeScript type-checking, Bun unit testing, and Cloudflare edge uploads.

Automating your deployment pipeline ensures that every commit pushed to your repository is systematically vetted for code style, type safety, syntax regressions, and unit test coverage before being deployed to your live production trading nodes.

This guide provides the complete specification and production-grade YAML workflow for implementing **GitHub Actions** integration pipelines.

The Continuous Integration & Deployment Pipeline Flow

```
graph TD
    Push["Git Push: main"] -->|Step 1| Checkout["Checkout submodules"]
    Checkout -->|Step 2| Install["Bun Install & Cache"]
    Install -->|Step 3| Lint["Code Formatting & Lint"]
    Lint -->|Step 4| Typecheck["tsc Typecheck"]
    Typecheck -->|Step 5| Test["Bun Test Suite"]
    Test -->|Step 5b| Audit["bun audit \n(dependency vulns)"]
    Audit -->|Step 6| Deploy["Sequenced Edge Deployment"]
    Deploy -->|Step 7| Notify["Telegram Notification"]

    SecretScan["gitleaks \n(secret scanning)"] -.->|Parallel| Lint
    CodeQL["CodeQL \n(weekly SAST)"] -.->|Parallel| Lint

    style Push fill:#CCCCCC,stroke:#3b82f6,stroke-width:2
    style Checkout fill:#CCCCCC,stroke:#10b981,stroke-width:1
    style Install fill:#CCCCCC,stroke:#10b981,stroke-width:1
    style Lint fill:#CCCCCC,stroke:#f59e0b,stroke-width:1
    style Typecheck fill:#CCCCCC,stroke:#f59e0b,stroke-width:1
    style Test fill:#CCCCCC,stroke:#f59e0b,stroke-width:1
    style Audit fill:#CCCCCC,stroke:#ef4444,stroke-width:1
    style SecretScan fill:#CCCCCC,stroke:#a855f7,stroke-width:1
    style CodeQL fill:#CCCCCC,stroke:#a855f7,stroke-width:1
    style Deploy fill:#CCCCCC,stroke:#ef4444,stroke-width:2
    style Notify fill:#CCCCCC,stroke:#10b981,stroke-width:1
```

Complete GitHub Actions Workflow (`deploy.yml`)

Create a YAML workflow file inside your repository at `.github/workflows/deploy.yml` :

```yaml
name: "Hoox Production CI/CD Pipeline"

on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main

concurrency:
  group: ${{ github.workflow }}-${{ github.ref }}
  cancel-in-progress: true

jobs:
  verify-and-deploy:
    name: "Verify & Deploy Stack"
    runs-on: "ubuntu-latest"
    timeout-minutes: 15

    steps:
 # 1. Checkout repository with all worker submodules recursively
      - name: "Checkout Codebase"
        uses: "actions/checkout@v4"
        with:
          submodules: "recursive"
          fetch-depth: 0

 # 2. Install Bun JavaScript runtime environment
      - name: "Setup Bun Runtime"
        uses: "oven-sh/setup-bun@v2"
        with:
          bun-version: "latest"

 # 3. Cache Bun dependency modules to optimize pipeline speeds
      - name: "Cache Workspace Dependencies"
        uses: "actions/cache@v4"
        with:
          path: "~/.bun/install/cache"
          key: "${{ runner.os }}-bun-${{ hashFiles('**/bun.lockb') }}"
          restore-keys: |
            ${{ runner.os }}-bun-

 # 4. Install all monorepo dependencies
      - name: "Install Dependencies"
        run: "bun install"

 # 5. Check formatting and ESLint rules
      - name: "Run Lint Checks"
        run: "bun run lint"

 # 6. Verify TypeScript compile-time type-safety (tsc --noEmit)
      - name: "Run Type Checks"
        run: "bun run typecheck"

 # 7. Execute all unit and integration test assertions (excluding live tests)
      - name: "Run Bun Test Suite"
        run: "bun test"

 # 8. Deploy all database schemas and workers in correct sequence
```

```
      - name: "Deploy Entire Edge Stack"
        if: "github.ref == 'refs/heads/main' && github.event_name == 'push'"
        env:
          CLOUDFLARE_API_TOKEN: "${{ secrets.CLOUDFLARE_API_TOKEN }}"
          CLOUDFLARE_ACCOUNT_ID: "${{ secrets.CLOUDFLARE_ACCOUNT_ID }}"
          SUBDOMAIN_PREFIX: "${{ secrets.SUBDOMAIN_PREFIX }}"
        run: |
  # Bootstrap local path binaries
          bun run build:cli

  # Deploy D1 SQL schemas to production
          npx wrangler d1 execute trade-data-db --file=workers/trade-worker/schema.sql --remote -

  # Sequentially upload all enabled workers
          ./packages/cli/bin/hoox.js deploy all --auto --quiet

  # Post-deployment: update service bindings URLs globally
          ./packages/cli/bin/hoox.js deploy update-internal-urls --quiet

  # Post-deployment: apply KV manifest configurations
          ./packages/cli/bin/hoox.js deploy kv-config --quiet

  # Post-deployment: update Telegram bot webhook routing path
          ./packages/cli/bin/hoox.js deploy telegram-webhook --quiet
```

Managing Secrets & Variable Scopes

To authorize GitHub to interact with your Cloudflare account, navigate to your repository's **Settings > Secrets and variables > Actions** and register the following Action Secrets:

1. **`CLOUDFLARE_API_TOKEN`**: A secure, scoped token with Workers, D1, KV, and DNS write permissions.
2. **`CLOUDFLARE_ACCOUNT_ID`**: Your unique 32-character Cloudflare dashboard hash.
3. **`SUBDOMAIN_PREFIX`**: The subdomain namespace chosen for your deployments.
4. **`GITHUB_TOKEN`**: Auto-provided, used by gitleaks for PR annotations.
5. **`INTERNAL_AUTH_KEY`**, **`HOOX_API_KEY`**, **`LOAD_TEST_BASE_URL`**: Required for nightly k6 load tests (see Security Overview).

You **never** need to store worker-specific secrets (like Bybit API keys or Telegram Bot tokens) in GitHub Secrets. These are stored directly in Cloudflare's secured key vaults. The CI pipeline only deploys the code logic; the running edge isolates pull their credentials locally from Cloudflare's hardware-level Secret Store at runtime.

Pipeline Caching & Concurrency Controls

- **Concurrency Lock**: The `concurrency` block configured in the YAML ensures that if you push a new commit while a previous deployment pipeline is running, GitHub Actions will **automatically cancel** the older, redundant run to avoid race conditions or double-deploying.
- **Bun Cache**: Caching dependency directories reduces build-time installation overhead from minutes to **under 8 seconds**.

Next Steps

- **Production Deployment Manual** — Audit DNS zones and custom domain routing options.
- **System Observability & Monitoring** — Stream console logs and check analytics datasets.

29 · DEPLOYMENT

OBSERVABILITY & TELEMETRY

How to configure Cloudflare Analytics Engine, set up 100% head sampling observability, and deploy real-time alarm systems.

Operating a globally distributed algorithmic trading network demands extreme observability. A silent failure in an order loop or a transient API throttling event can lead to missed targets and unhedged exposures.

This guide outlines our **telemetry architecture**, covering Cloudflare Workers 100% request sampling, time-series SQL database queries, R2 logging, and automated alarm systems.

---

1. 100% Request Head Sampling

To monitor traffic at Cloudflare's edge compiler level, every worker in the Hoox monorepo enables native **observability tracing** inside its `wrangler.jsonc`:

```
{
  "observability": {
    "enabled": true,
    "head_sampling_rate": 1, // Captures and logs 100% of all incoming requests
  },
}
```

Metrics Tracked Automatically

Once enabled, Cloudflare captures and graphs these metrics near-instantaneously:

- **Requests velocity**: Total hits and requests/second rates.
- **CPU execution duration (ms)**: The exact CPU time consumed by the isolate thread.
- **Uncaught Exceptions**: Crashes or code errors that bypass middlewares.
- **Subrequest Counts**: Outbound HTTP calls made to exchange APIs or other workers.

---

2. Cloudflare Analytics Engine (SQL Telemetry)

While Cloudflare collects basic HTTP metrics, Hoox uses **Cloudflare Analytics Engine** inside `analytics-worker` to track trading-specific variables (e.g. execution slippage, exchange response delays, fee sums).

Under-the-Hood SQL Queries

The `analytics-worker` exposes a SQL interface to query the metrics dataset directly from your Next.js Dashboard:

```
/* Query 1: Calculate the 95th percentile latency of Bybit API order fills */
SELECT
  quantiles(0.95)(double1) as p95_latency_ms,
  count() as total_executions
FROM hoox_telemetry
WHERE blob1 = 'trade-worker'
  AND blob2 = 'bybit'
  AND timestamp >= now() - INTERVAL '1' HOUR

/* Query 2: Aggregate error velocities across all edge isolates */
SELECT
  blob1 as worker_name,
  count() as error_count
FROM hoox_telemetry
WHERE blob3 = '0' -- Success = false
  AND timestamp >= now() - INTERVAL '6' HOUR
GROUP BY worker_name
```

---

3. Logging & R2 Storage Offloading

Because transactional databases like SQLite D1 have write limits, Hoox implements a **dual-tier logging policy**:

- **High-Value Data**: Transaction statuses and fills are written to your D1 `trades` table.
- **Verbose Telemetry**: Full, verbose JSON payload exchanges, network headers, and WebSocket stream records are serialized and saved to **R2 Storage** using date-based key paths: `logs/bybit/BTCUSDT/2026-05-19/order-18049284739.json`

This offloading reduces database write volumes by **up to 75%**, keeping your database compact, fast, and fully within free-tier limits.

---

4. Standardized Alarm Systems

If a critical error or execution failure occurs (e.g., an order is rejected due to insufficient margin, or the kill switch is flipped due to drawdown limits):

1. The executing worker intercepts the exception using the shared error handler.
2. Direct V8 Service Binding pings `telegram-worker` instantly.
3. You receive an emergency alert on your phone.

```
// Standard alarm notification trigger
if (orderResponse.status === "Rejected") {
  await env.TELEGRAM_SERVICE.fetch("https://telegram-worker/alert", {
    method: "POST",
    headers: {
      "Content-Type": "application/json",
      "X-Internal-Auth-Key": env.INTERNAL_KEY_BINDING,
    },
    body: JSON.stringify({
      chatId: env.TELEGRAM_CHAT_ID_DEFAULT,
      message: ` <b>Emergency Order Reject!</b>\nExchange: Bybit\nSymbol: BTCUSDT\nReason: Insuff
    }),
  });
}
```

Need to tail live worker logs to debug a custom strategy? Run `hoox logs tail trade-worker` in your terminal to open a real-time WebSocket connection to Cloudflare's global edge logging console!

Next Steps

- **Debugging Runbook** — Diagnose local and remote V8 exceptions using diagnostic profiles.
- **Self-Healing & Repair** — Recover from degraded workers and provision missing bindings.

30 · DEPLOYMENT

ZERO TRUST & SECURITY HARDENING

How to configure Cloudflare Access (Zero Trust), enforce Multi-Factor Authentication (MFA), and configure WAF firewall allow-lists for TradingView webhook IPs.

While the Hoox dashboard features a highly secure, custom cookie-based authentication middleware, wrapping your dashboard and API endpoints inside **Cloudflare® Zero Trust (Access)** provides an enterprise-grade security perimeter.

By placing your deployment behind Cloudflare Access, you can enforce Multi-Factor Authentication (MFA), restrict access to specific GitHub/Google SSO identities, evaluate device posture, and drop malicious scanner payloads at the DNS level before they ever hit your workers.

The Zero Trust Protective Boundary

```
graph TD
    Client["Public Web Browser"] -->|Request| Gate["Cloudflare Edge

WAF / Access Gate"]
    Gate -->|MFA & GitHub SSO Validation| Dash["Next.js Dashboard Isolate"]

    style Client fill:#CCCCCC,stroke:#3b82f6,stroke-width:2
    style Gate fill:#CCCCCC,stroke:#f59e0b,stroke-width:2
    style Dash fill:#CCCCCC,stroke:#10b981,stroke-width:2
```

- **Zero Public Exposure**: The dashboard isolate does not evaluate public logins directly.
- **MFA Gate**: Users are intercepted by a secure Cloudflare authentication card at the nearest edge PoP.
- **Zero Cost**: Cloudflare's Zero Trust free tier includes up to 50 users, which is more than enough for a personal algorithmic trading desk.

---

1. Step-by-Step Dashboard Access Setup

Step 1: Enable Zero Trust on Your Account

1. Log in to the Cloudflare Dashboard and click **Zero Trust** on the sidebar.
2. If this is your first time, follow the onboarding prompts to register a unique **Team Name** (e.g. `alpha-trading.cloudflareaccess.com`).

Step 2: Create a Self-Hosted Application

1. In the Zero Trust dashboard, navigate to **Access > Applications** and click **Add an application**.
2. Select **Self-hosted**.
3. **Application Name**: `Hoox Dashboard Cockpit`.
4. **Session Duration**: Select your preference (e.g. `24 Hours` to prevent constant login prompts).

5. **Application Domain**: Enter the custom domain mapped to your dashboard worker (e.g., `hoox.my-trading-empire.com`).

Step 3: Configure Authorization Policies

1. Click **Next** to proceed to the Policies tab.
2. **Policy Name**: `Allow Admin Only`.
3. **Action**: `Allow`.
4. **Configure Rules**:
   - **Include**: Select **Emails** and enter your personal email address (enables Email OTP).
   - **Include (SSO)**: Alternatively, select **GitHub Org/Teams** or **Google Workspace** to enable SSO integrations.
5. In the **Require** block, you can optionally require a valid **security key (MFA)** or **device posture check** (e.g. verifying that your laptop runs a specific OS version).

Step 4: Map Identity Providers & Save

1. In **Settings > Authentication**, link your desired login providers (Google Workspace, GitHub OAuth, or Email OTP).
2. Save the application.
3. Open your browser and navigate to your custom domain (`https://hoox.my-trading-empire.com`). You will be intercepted by your Cloudflare Access card. Once authorized, you are passed cleanly to your Next.js dashboard.

---

2. Strict WAF Webhook IP Allow-listing

To ensure that **only** TradingView's official servers can fire signals to your `/webhook` entryway:

1. Under your Cloudflare DNS zone dashboard, navigate to **Security > WAF > Custom Rules**.
2. Click **Create Rule**.
3. **Rule Name**: `Restrict /webhook to TradingView IPs`.
4. **Field**: `URI Path` | **Operator**: `equals` | **Value**: `/webhook`.
5. **And**: `IP Source Address` | **Operator**: `is not in` | **Value**: (Paste TradingView's official IP ranges here, which are automatically synced by running the `hoox waf configure --TradingView-only` command).
6. **Action**: **Block** (or **Challenge**).
7. Save. All unauthorized traffic hitting `/webhook` is dropped instantly at the DNS edge, preventing any V8 compute load.

```
# Automated WAF setup via CLI
hoox waf configure --TradingView-only
```

---

3. Optional: Bypassing Local Dashboard Auth

Once your custom domain is wrapped inside Cloudflare Access, the dashboard's built-in login form (`DASHBOARD_USER` , `DASHBOARD_PASS`) becomes redundant.

To streamline access:

1. Edit the Next.js `middleware.ts` file inside `workers/dashboard/src/` .
2. Toggle the authentication checker to leverage Cloudflare's Access headers:

```
// Next.js Edge Middleware
export function middleware(request: Request) {
  // Verify the JWT payload injected in headers by Cloudflare Access
  const cfAccessJwt = request.headers.get("Cf-Access-Jwt-Assertion");
  if (cfAccessJwt) {
    // Cloudflare has already authenticated the session. Bypass local login.
    return NextResponse.next();
  }
  // Fallback to local cookie checks...
}
```

Next Steps

- **Next.js Dashboard worker Profile** — Review OpenNext compilation and asset bindings.
- **System Observability & Metrics** — Setup time-series logging and Analytics Engine tables.

31 · SECURITY

SECURITY TESTING & HARDENING

Comprehensive security testing infrastructure, auth hardening, and CI/CD security scanning for the Hoox trading platform.

Overview

Security work spans three layers:

1. **Code Hardening** — Fixing auth vulnerabilities (fail-closed middleware, timing-safe comparisons)
2. **Security Testing** — Auth bypass tests, fuzz tests, header verification
3. **CI/CD Scanning** — Dependency auditing, secret detection, CodeQL analysis

---

1. Auth Middleware Hardening

`requireInternalAuth` — Fail-Closed

File: `packages/shared/src/middleware/auth.ts`

Previously, `requireInternalAuth()` returned `null` (allowed the request through) when `INTERNAL_KEY_BINDING` was not configured. This meant workers with unconfigured auth keys were wide open.

Fix: The middleware now returns `401 Unauthorized` when the key is missing:

```
// Before: auth was optional when key not set
if (!expectedKey) return null;

// After: fails closed
if (!expectedKey) {
  return new Response(
    JSON.stringify({
      success: false,
      error: `Internal auth key ${keyName} not configured`,
    }),
    { status: 401, headers: { "Content-Type": "application/json" } }
  );
}
```

`timingSafeEqual` — Exported for Cross-Worker Use

File: `packages/shared/src/middleware/auth.ts`

The `timingSafeEqual()` function was a private helper used only internally by `requireAuth()` and `requireInternalAuth()`. It's now exported so all workers can use it for constant-time string comparisons.

Webhook API Key — Timing-Safe Comparison

File: `workers/hoox/src/index.ts`

The hoox gateway's webhook API key validation was changed from `===` to `timingSafeEqual()` to prevent timing side-channel attacks:

```
// Before (vulnerable to timing attack):
const isValid = apiKey === expectedKey;

// After (constant-time comparison):
const isValid = timingSafeEqual(apiKey, expectedKey);
```

---

2. Auth Coverage by Worker

All workers now have authentication on their internal endpoints:

Worker	Auth Method	Protected Endpoints	Unprotected
hoox	API key in body (timing-safe) + IP allowlist	`POST /webhook`	`GET /health`
d1-worker	`X-Internal-Auth-Key`	`POST /query`, `POST /batch`, `GET /api/*`	`GET /health`
trade-worker	`X-Internal-Auth-Key`	`POST /webhook`, `POST /process`	`GET /health`, `GET /api/signals`
agent-worker	`X-Internal-Auth-Key`	All `POST /agent/*`, `GET /agent/status|config|models|health`	`GET /`
analytics-worker	`X-Internal-Auth-Key`	All `POST /track/*` (5 endpoints)	`GET /health`
telegram-worker	`X-Internal-Auth-Key`	`POST /process`	`GET /health`, `POST /webhook` (Telegram secret)
email-worker	`X-Internal-Auth-Key`	`POST /email-signal`	`GET /health`, `POST /webhook` (Mailgun HMAC)
report-worker	`X-Internal-Auth-Key`	`GET /report`	`GET /health`
web3-wallet-worker	`X-Internal-Auth-Key`	`GET /` (wallet address)	`GET /health`
dashboard	Session cookie	All routes	`/login`, `/api/auth`, `/_next/*`

Health check endpoints (`GET /health`) are intentionally left unauthenticated for monitoring systems.

---

3. Shared Security Headers Middleware

File: `packages/shared/src/middleware/security-headers.ts`

A reusable middleware that provides 7 standard security headers following the same pattern as the existing `cors.ts` middleware.

API

```
// Get headers as a plain object
const headers = secureHeaders({ contentSecurityPolicy: "default-src 'self'" });

// Wrap an existing Response with security headers
const secure = wrapWithSecurityHeaders(response);

// Customize individual headers
const custom = secureHeaders({
  xFrameOptions: "SAMEORIGIN",
  contentSecurityPolicy: "", // Disable CSP
});
```

Default Headers

Header	Default Value
`X-Content-Type-Options`	`nosniff`
`X-Frame-Options`	`DENY`
`X-XSS-Protection`	`1; mode=block`
`Referrer-Policy`	`strict-origin-when-cross-origin`
`Permissions-Policy`	All features disabled
`Strict-Transport-Security`	`max-age=31536000; includeSubDomains`
`Content-Security-Policy`	`default-src 'self'`

Dashboard

File: `workers/dashboard/src/middleware.ts`

The dashboard was upgraded from 2 headers (`X-Content-Type-Options`, `X-Frame-Options`) to the full 7-header set via the shared middleware. The `withSecurityHeaders()` wrapper ensures headers are applied on **all** return paths (including early returns for static files, login pages, CSRF errors, and auth failures).

---

4. Security Test Suites

Auth Bypass Tests

File: `tests/security/auth-bypass.test.ts` **Tests:** 12

Validates:

- `requireInternalAuth` returns 401 when key is not configured (fail-closed)
- `requireInternalAuth` returns 401 with missing/wrong auth header
- `requireInternalAuth` passes with valid key
- `timingSafeEqual` correctness: identical strings, different lengths, different same-length strings, empty strings, special characters, long keys

Security Headers Tests

File: `tests/security/security-headers.test.ts` **Tests:** 9

Validates:

- All 7 default headers are present
- Individual header overrides work
- CSP can be disabled (set to empty string)
- `wrapWithSecurityHeaders` preserves body, status, and existing headers
- Custom options pass through

Fuzz Tests

File: `tests/security/fuzz.test.ts` **Tests:** 19

Validates:

- Auth enforcement on d1-worker `/query` (valid, missing, wrong key)
- SQL injection attempts via auth headers (`' OR '1'='1`, UNION injection)
- `timingSafeEqual` edge cases: Unicode homoglyphs, null bytes, 10k-char strings, regex special chars, emoji
- Request edge cases: missing Content-Type, GET to POST, OPTIONS preflight, extremely long headers
- Webhook payload edge cases: non-JSON body, empty body, unconfigured key (fail-closed)

Running Security Tests

```
# Run all security tests
bun test tests/security/

# Run specific test files
bun test tests/security/auth-bypass.test.ts
bun test tests/security/security-headers.test.ts
bun test tests/security/fuzz.test.ts
```

---

5. CI/CD Security Scanning

Dependency Audit

File: `.github/workflows/ci.yml`

A `bun audit` step runs in CI after unit tests to detect known vulnerabilities in dependencies. It's configured with `continue-on-error: true` so it doesn't block development, but findings are visible in workflow output.

Secret Scanning

File: `.github/workflows/secret-scan.yml` **Config:** `.gitleaks.toml`

Uses `gitleaks/gitleaks-action@v2` to detect hardcoded secrets (API keys, tokens, passwords) across the full git history. Runs on:

- Push/PR to `main` and `develop`
- Weekly (Sunday)

- Manual trigger via `workflow_dispatch`

Information only (`continue-on-error: true`) — findings don't block CI.

Secret Allowlist

The `.gitleaks.toml` allowlists:

- Test files (`.test.`, `.spec.`, `tests/`)
- Example/template files (`.example`, `.env.example`)
- Documentation and changelogs
- Generated files (`dist/`, `.next/`, `coverage/`)
- Lock files (`bun.lock`, `package-lock.json`)
- Placeholder patterns (`__SECRET__`, `YOUR_`, `<USE_WRANGLER_SECRET_PUT>`)

CodeQL

File: `.github/workflows/codeql.yml`

Weekly security analysis with `security-and-quality` query suite for JavaScript/TypeScript.

Dependabot

File: `.github/dependabot.yml`

Daily npm and GitHub Actions dependency checks with automatic PR creation.

---

6. CI Pipeline Security Flow

```
graph LR
    Lint["ESLint"] --> TC["TypeScript Check"]
    TC --> Test["Unit + Integration Tests"]
    Test --> Audit["bun audit (informational)"]
    Audit --> Build["Build"]

    SecretScan["gitleaks (informational)"] -.->|Parallel| Lint
    CodeQL["CodeQL (weekly)"] -.->|Parallel| Lint
    Dependabot["Dependabot (daily)"] -.->|PR| Lint
```

---

7. Performance & Load Testing

File: `tests/load/` **Tool:** k6 (separate from bun test)

See the `tests/load/` directory for full documentation.

Script	Target	Description
`webhook-flow.js`	hoox `POST /webhook`	TradingView alert simulation
`d1-query-load.js`	d1-worker `POST /query+batch`	Concurrent D1 query patterns
`agent-cron-sim.js`	agent-worker	5-minute cron cycle simulation
`system-mixed.js`	All combined	70/20/10 traffic distribution

CI: Nightly via `.github/workflows/load-test.yml` (informational thresholds).

---

8. Environment Setup

Required GitHub Secrets

For CI security scanning to work, add these to your repository:

Secret	Used By	Purpose
`GITHUB_TOKEN`	`secret-scan.yml`	gitleaks PR annotations (auto-provided)
`INTERNAL_AUTH_KEY`	`load-test.yml`	Internal worker auth for load tests
`HOOX_API_KEY`	`load-test.yml`	Webhook API key for load tests
`LOAD_TEST_BASE_URL`	`load-test.yml`	Target URL for load tests

---

9. Related Documentation

- Testing Framework & QA Standards — Unit/integration/E2E testing
- CI/CD Pipeline — Full deployment pipeline
- Isolate Communication Spec — Service binding auth
- Zero Trust Deployment — Cloudflare Access + WAF
- Internal Endpoints Map — All worker endpoints

API ENDPOINT DIRECTORY

Exhaustive HTTP REST API directory for the Hoox edge gateway, webhooks, analytics telemetry, and database queries.

This directory provides the complete HTTP REST API specification for all public and internal endpoints exposed across the Hoox edge microservices stack.

Security & Authorization Headers

All internal calls between workers (via V8 Service Bindings) must provide the standard internal authentication header:

```
X-Internal-Auth-Key: <INTERNAL_KEY_BINDING>
```

Public webhook and dashboard endpoints use cookie authorization or custom API keys:

- **Webhook Ingestion**: Authenticated using the `apiKey` property inside the JSON payload.
- **Telegram webhook**: Authenticated via the secure token path: `/telegram/<TELEGRAM_WEBHOOK_SECRET>`.

Ingress Webhook Endpoints (`workers/hoox`)

The public gateway acts as the primary firewall and entry entryway for all external trade signals.

A. Ingest Signal Webhook

Receives TradingView alerts or automated cURL signals.

- **Path**: `/webhook`
- **Method**: `POST`
- **JSON Payload**:

```
{
  "apiKey": "secure_webhook_key_18305",
  "exchange": "bybit",
  "action": "LONG",
  "symbol": "BTCUSDT",
  "quantity": 0.002,
  "leverage": 10,
  "idempotencyKey": "uuid-9b1deb4d-3b7d"
}
```

- **Success Response (200 OK)**:

```
{
  "success": true,
  "requestId": "9b1deb4d-3b7d-4bad-9bdd-2b0d7b3dcb6d",
  "exchange": "bybit",
  "symbol": "BTCUSDT",
  "action": "LONG",
  "result": {
    "orderId": "18049284739",
    "status": "Filled",
    "price": 68425.5
  }
}
```

B. Proactive Health Diagnostics

- **Path**: `/health`
- **Method**: `GET`
- **Success Response (200 OK)**:

```
{
  "status": "ok",
  "timestamp": 1779261050000,
  "bindings": {
    "d1": "connected",
    "kv": "connected",
    "queue": "active"
  }
}
```

Database Service Endpoints (`workers/d1-worker`)

The `d1-worker` serves as a private, internal SQL proxy database.

A. Execute Single SQL Statement

- **Path**: `/query`
- **Method**: `POST`
- **JSON Payload**:

```
{
  "requestId": "9b1deb4d-3b7d-4bad-9bdd-2b0d7b3dcb6d",
  "query": "SELECT * FROM trades WHERE symbol = ? LIMIT ?",
  "params": ["BTCUSDT", 5]
}
```

- **Success Response (200 OK)**:

```
{
  "success": true,
  "results": [{ "id": "trade-1", "symbol": "BTCUSDT", "price": 68400 }],
  "meta": { "rows_read": 1, "duration": 3.5 }
}
```

B. Execute Transactional Batch Statements

- **Path**: `/batch`
- **Method**: `POST`
- **JSON Payload**:

```
{
  "requestId": "9b1deb4d-3b7d-4bad-9bdd-2b0d7b3dcb6d",
  "queries": [
    {
      "query": "INSERT INTO trades (id, symbol) VALUES (?, ?)",
      "params": ["t-1", "ETHUSDT"]
    },
    {
      "query": "UPDATE positions SET size = size + 0.05 WHERE symbol = 'ETHUSDT'",
      "params": []
    }
  ]
}
```

- **Success Response (200 OK)**:

```
{
  "success": true,
  "results": [{ "meta": { "changes": 1 } }, { "meta": { "changes": 1 } }]
}
```

AI Risk & Chat Endpoints (`workers/agent-worker`)

Bridges background risk audits and multi-provider AI chat streams.

A. Conversational Chat Stream (Server-Sent Events)

- **Path**: `/agent/chat`
- **Method**: `POST`
- **Headers**: `Accept: text/event-stream`
- **JSON Payload**:

```
{
  "prompt": "Summarize my trade history and average win rate today.",
  "stream": true,
  "provider": "anthropic"
}
```

- **Success Response**: Emits standard text/event-stream updates, terminating with `data: [DONE]`.

B. Multimodal AI Vision Audit

- **Path**: `/agent/vision`
- **Method**: `POST`
- **JSON Payload**:

```
{
  "imageBase64": "iVBORw0KGgoAAAANSUhEUgAA...",
  "prompt": "Evaluate this trading chart image for support and resistance levels."
}
```

- **Success Response (200 OK)**:

```
{
  "success": true,
  "analysis": "Based on the provided chart screenshot, there is strong horizontal support at $
}
```

Next Steps

- **Request Payload Schemas** — Analyze the complete JSON request schemas and type rules.
- **Standard Response Schemas** — Check error factories and normal JSON envelopes.

REQUEST PAYLOAD SCHEMAS

High-integrity JSON request payload specifications, TypeScript interfaces, and validation rules across Hoox microservices.

To maintain robust data routing and prevent runtime failures, Hoox enforces a strict **payload contract** across all internal and public service boundaries. All incoming requests are validated by active JSON Schema or Zod middleware before entering V8 execution loops.

This document details the exact JSON schemas, TypeScript interfaces, and validation rules for all primary request payloads.

1. Standard Request Envelope

Every service-to-service invocation (via Service Bindings) wraps its business parameters inside a standardized **Request Envelope** containing distributed tracing indices:

```
  requestId: string; // Unique UUIDv4 distributed trace ID
  payload: T; // Service-specific payload
}
```

```
{
  "requestId": "9b1deb4d-3b7d-4bad-9bdd-2b0d7b3dcb6d",
  "payload": {
    "symbol": "BTCUSDT",
    "quantity": 0.005
  }
}
```

2. Trade Execution Payloads (`trade-worker`)

The execution engine processes two primary types of order payloads:

A. Centralized Exchange Order Payload (`/webhook` & `/process`)

```
  exchange: "binance" | "bybit" | "mexc"; // Target exchange
  action: "LONG" | "SHORT" | "CLOSE"; // Margin position action
  symbol: string; // Standardized asset ticker (e.g. "BTCUSDT")
  quantity: number; // Order size in contracts or tokens
  leverage?: number; // Optional margin leverage multiplier
  price?: number; // Optional execution limit price
  orderType?: "MARKET" | "LIMIT"; // Default: MARKET
}
```

```
{
  "exchange": "bybit",
  "action": "LONG",
  "symbol": "BTCUSDT",
  "quantity": 0.01,
  "leverage": 10,
  "orderType": "MARKET"
}
```

B. On-Chain DeFi Swap Payload (`/dex`)

```
  chain: "ethereum" | "arbitrum" | "polygon"; // EVM target network
  to: string; // Recipient address or Uniswap Router
  value: string; // Transaction value in Wei (as string)
  data: string; // Encoded contract call data bytes
  gasLimit?: number; // Optional transaction gas limit
}
```

```
{
  "chain": "arbitrum",
  "to": "0x6b175474e89094c44da98b954eedeac495271d0f",
  "value": "100000000000000000",
  "data": "0xa9059cbb000000000000000000000000..."
}
```

---

3. Database SQL Query Payloads (`d1-worker`)

The SQLite database proxy accepts parameterized SQL statements to protect against SQL injections:

A. Execute Single SQL Statement

```
  query: string; // Parameterized SQL statement
  params: Array<string | number | boolean | null>; // Binding values
}
```

```
{
  "query": "SELECT * FROM trades WHERE symbol = ? AND status = ?",
  "params": ["BTCUSDT", "Filled"]
}
```

---

4. Telegram Notification Payloads (`telegram-worker`)

Used to push structured alerts and charts to your mobile device:

```
  chatId: string; // Target Telegram Chat ID
  message: string; // Formatted message content
  parseMode?: "HTML" | "MarkdownV2"; // Default: HTML
}
```

```
{
  "chatId": "987654321",
  "message": "<b>Alert:</b> Position filled at 68,420.",
  "parseMode": "HTML"
}
```

Adding new fields to a payload schema? Remember to update the corresponding type interfaces in `@jango-blockchained/hoox-shared/types` to ensure complete type safety across all workers!

Next Steps

- **API Endpoint Directory** — Analyze REST routes, headers, and endpoints mappings.
- **Standard Response Schemas** — Check success envelopes and error models.

34 · API

STANDARD RESPONSE SCHEMAS

High-integrity JSON response templates, success envelopes, edge error codes, and shared error middleware specs.

To ensure predictable error handling and parsing across all microservices, Hoox enforces a **strict response contract**. Every worker responds with a standardized JSON envelope containing tracing details, operation status, results payload, and detailed error models.

This document details the exact JSON templates, types, and error factories utilized in the monorepo.

1. Standard Response Envelope

Every HTTP response returned by an edge worker is encapsulated within a unified JSON envelope:

```
  success: boolean; // Absolute state of the operation
  requestId: string; // UUIDv4 trace ID mapped from request
  result?: T; // Operation success metadata payload
  error?: {
    // Detail model present only if success = false
    code: string; // Unique machine-parseable error token
    message: string; // Human-readable error description
    details?: any; // Optional specific array of field issues
  };
}
```

2. Success Response Templates

A. Trade Execution Fill Success (`trade-worker` - 200 OK)

```
{
  "success": true,
  "requestId": "9b1deb4d-3b7d-4bad-9bdd-2b0d7b3dcb6d",
  "result": {
    "orderId": "18049284739",
    "status": "Filled",
    "executedQty": 0.005,
    "price": 68425.5,
    "timestamp": 1779261050000
  },
  "error": null
}
```

B. Telegram Push Success (`telegram-worker` - 200 OK)

```
{
  "success": true,
  "requestId": "9b1deb4d-3b7d-4bad-9bdd-2b0d7b3dcb6d",
  "result": {
    "messageId": 482904,
    "delivered": true
  },
  "error": null
}
```

---

3. Error Models & Edge Error Codes

When a worker operation fails, the `success` flag is set to `false`, the `result` field is omitted or set to `null`, and the `error` model is fully populated:

A. 400 Bad Request (JSON Validation Error)

```
{
  "success": false,
  "requestId": "9b1deb4d-3b7d-4bad-9bdd-2b0d7b3dcb6d",
  "error": {
    "code": "VALIDATION_ERROR",
    "message": "Invalid JSON payload structure.",
    "details": [{ "field": "quantity", "issue": "Must be greater than zero." }]
  }
}
```

B. 409 Conflict (Idempotency Mutex Intercept)

```
{
  "success": false,
  "requestId": "9b1deb4d-3b7d-4bad-9bdd-2b0d7b3dcb6d",
  "error": {
    "code": "DUPLICATE_REQUEST",
    "message": "Order rejected. Durable Object locked: trace ID already processed in the last 24
  }
}
```

---

4. Shared `Errors` Factory Middleware

To enforce this response contract without writing redundant JSON wrappers in every worker, we use the standardized `Errors` factory from the shared monorepo package `@jango-blockchained/hoox-shared/errors`:

```
// 1. Validation Failures (400 Bad Request)
if (!payload.symbol) {
  return Errors.badRequest("Symbol is a required parameter.");
}

// 2. Secret Key Mismatch (401 Unauthorized)
if (requestHeaderKey !== env.INTERNAL_KEY_BINDING) {
  return Errors.unauthorized("Access Denied: Invalid X-Internal-Auth-Key.");
}

// 3. System Exceptions (500 Internal Server Error)
try {
  await database.write(data);
} catch (err: any) {
  return Errors.internal(`Database write exception: ${err.message}`);
}
```

These factory methods automatically serialize the exception, attach the active `requestId` from the context, set the correct HTTP status code, and return a compliant `Response` object.

Next Steps

- **API Endpoint Directory** — Analyze REST routes, headers, and endpoints mappings.
- **Request Payload Schemas** — Check JSON request payload specifications and typescript interfaces.

CLI ARCHITECTURE & FEATURES

Detailed specification of the Hoox Command-Line Interface, monorepo workspaces compilation, and task-management engines.

The **hoox CLI** (`packages/cli`) is the unified lifecycle engine of the trading platform monorepo. It governs local sandboxes, compiles TypeScript structures, coordinates Cloudflare infrastructure resources, deploys edge isolates, manages encrypted secrets, and executes self-healing diagnostics.

Monorepo Workspace Design

The CLI is integrated into our monorepo using **Bun Workspaces**, which link local packages together. This design ensures that the CLI binary can resolve and load local shared types (`@jango-blockchained/hoox-shared`) and TUI components (`packages/tui`) instantly without network downloads or pre-compilation overhead:

```
hoox-setup/ (Monorepo Root)
├── packages/
│   ├── cli/       # CLI Source code (entry binary: bin/hoox.js)
│   ├── tui/       # OpenTUI dashboard source code
│   └── shared/    # Common libraries (auth, router, error models)
├── workers/
│   └── ...        # Edge V8 isolates
└── package.json   # Root workspace manager
```

---

1. Command-Line Core Architectures

The CLI binary parses terminal instructions using the following architectural layers:

A. Command Dispatcher (`packages/cli/src/index.ts`)

Intercepts all incoming arguments (e.g. `hoox infra provision`), evaluates global flags (`--json`, `--quiet`), validates configuration integrity, and delegates execution to target command modules under `src/commands/`.

B. Cloudflare API Adapters (`src/adapters/`)

Translates command intentions (like `hoox infra d1 create`) into parameterized Cloudflare API REST payloads, bypassing wrangler wrappers when high-performance execution is needed.

C. State Engine (`src/core/`)

Tracks active project profiles, enabled worker matrices, and workspace configuration formats (`wrangler.jsonc` vs. `wrangler.toml`).

---

2. Declarative Config Mapping & Validation

To prevent configuration drift, the CLI enforces strict type validation on `wrangler.jsonc` files:

1. **Config Interfaces**: Built-in parsers validate configuration files against type-safe TypeScript interfaces (`Config` and `WorkerConfig`) defined in `src/core/types.ts`.
2. **Setup Verification (`hoox check setup`)**: Compares active workspace profiles against example files, audits environment variables keys, and scans for missing bindings, outputting formatted terminal reports.

---

3. Self-Healing & Diagnostics Engine

One of the CLI's most powerful features is its **guided repair framework** (`hoox repair` command groups):

- **Diagnostic Probes**: The `hoox repair check` command runs a 5-step checklist (verifying submodule checkouts, NPM/Bun package resolutions, TypeScript variables, Cloudflare zone links, and encrypted credentials).
- **Automated Recovery**: If a missing resource is identified (e.g. a D1 database ID is bound in `wrangler.jsonc` but the database doesn't exist on your Cloudflare account), the repair engine prompts you and provisions it automatically:

```
# Provision missing Cloudflare bindings and repair system states
hoox repair infra
```

Every single command supports the `--json` global flag. This outputs machine-parseable JSON payloads (e.g. `hoox check health --json`), allowing you to integrate the CLI with external telemetry dashboards or alert scripts effortlessly!

---

4. Output Framework (v0.9.0+)

The CLI's terminal output is built on a shared framework of small, focused primitives that every command composes. The framework lives in `packages/cli/src/utils/` and spans five layers:

Theme tokens (`utils/theme.ts`)

A single source of truth for color and iconography. Refined in v0.9.0 to a **modern-minimal** aesthetic (zinc/slate text, single indigo-400 accent, de-saturated status colors). New tokens available to every command:

- **Text scale**: `theme.text` (zinc-200), `textMuted` (zinc-400), `textSubtle` (zinc-500), `textFaint` (zinc-600)
- **Status**: `success` (emerald-400), `error` (rose-400), `warning` (amber-400), `info` (sky-400), `accent` (indigo-400)
- **Borders**: `border` (zinc-700), `borderStrong` (zinc-600)
- **Spinner**: braille dots `["⠋","⠙","⠹","⠸","⠼","⠴","⠦","⠧","⠇","⠏"]`

All previous named tokens (`success`, `error`, `heading`, `value`, etc.) are preserved for backward compatibility — only the values changed.

Output formatters (`utils/formatters.ts`)

Every command uses one of these for consistent output:

Formatter	Purpose	Key options
`formatSuccess(msg)`	Single-line success indicator	`--json` / `--quiet`
`formatError(err, opts?)`	Card-style error with `[code]` badge, `did you mean` suggestion	`suggestions?` , `inCard?`
`formatTable(rows, opts?)`	Box-drawn table	`zebra` , `alignNumbers` , `colorizeStatus` , `compact`
`formatKeyValue(pairs)`	Aligned `key: value` pairs	—
`formatHeader(text, opts?)`	Section heading with rule	`subtitle?`
`formatList(items)`	Bulleted list	—
`formatJson(data)`	Always-JSON output	—
`formatBadge(level, text?)`	Inline status chip	—
`formatHint(text)`	Subtle hint line	—
`formatCompletion(msg, opts?)`	" Done in 1.2s" footer + next-step suggestion	`durationMs?` , `suggestion?`
`formatNumber(n)` / `formatBytes(n)`	Compact notation (`1.2K` / `1.5M` / `1.0 KiB`)	—

Rich mode gate (`utils/format-mode.ts`)

`isRichMode(opts)` is the single switch every formatter checks before emitting color. It returns `false` (i.e. suppresses color) when any of these is true:

- `--json` or `--quiet` flag is set
- `--no-color` flag is set
- `NO_COLOR` env var is set (https://no-color.org)
- `TERM=dumb` is set
- stdout is not a TTY (piped, redirected, or running under CI)

This is the single place that encodes "is rich output appropriate right now?" — every formatter asks it instead of duplicating TTY/env detection.

Custom help formatter (`utils/help-formatter.ts`)

Replaces Commander's default flat help layout with a sectioned modern-minimal layout for `hoox --help` and per-command `--help` :

```
hoox deploy all
Cloudflare Workers Platform
──────────────────────────────────────────────────

Usage
  hoox deploy all [options]

Options
  --auto              Skip confirmations
  --json              Output JSON

Examples
  $ hoox deploy all
  $ hoox deploy all --auto
```

Wired into the program via `program.configureHelp({ formatHelp: (cmd, helper) => renderHelp(cmd, helper) })`.

"Did you mean" + completion footer (`utils/completion.ts`, `utils/error-handler.ts`)

Two small touches that improve UX on errors and after successful commands:

- **Did you mean** — `hoox deplpy` → `did you mean 'hoox deploy' ?`. Uses Levenshtein distance with a threshold of 2; checks against all registered command names (top-level + nested).
- **Completion footer** — every successful command prints `<message> in 1.2s` and optionally `→ next: hoox <next-command> (<reason>)`. Wired into the global `program.hook("postAction", ...)` in `index.ts`.

Banner (`ui/banner.ts`)

`renderBanner()` is called when the CLI runs with no arguments. The version is read at module init by walking up from `import.meta.url` looking for the `@jango-blockchained/hoox-cli` `package.json` — this works in source, bundle, and global install layouts. Default variant is `minimal` (the cleanest of four); `legacy`, `horizon`, and `signal` are opt-ins.

Adding a new output

When writing a new command, the pattern is:

```
  formatSuccess,
  formatTable,
  formatHint,
  getFormatOptions,
} from "../../utils/formatters.js";

  program.command("my").action(
    withErrorHandling(async (cmd) => {
      const opts = getFormatOptions(cmd);
      const rows = await loadMyData();
      formatTable(rows, opts); // respects --json / --quiet / --no-color
      formatSuccess("Done", opts); // suppressed in --json/--quiet
      formatHint("next: hoox deploy", opts);
    })
  );
}
```

Every formatter handles the format modes for you — you never read `process.stdout.isTTY` or `process.env.NO_COLOR` yourself.

Next Steps

- **CLI Reference Manual** — Review the complete command tree, positional arguments, and flags.
- **Wrangler Setup & Tooling** — Configure Wrangler to bind local D1 and KV instances for dev testing.

TUI CODE ARCHITECTURE

High-integrity architecture specifications, Zustand store configurations, and connection state machines of the Hoox Terminal UI.

The **Hoox Terminal UI (TUI)** is a full-screen, keyboard-driven operations center built with **OpenTUI**, React 19, and Zustand. This document covers package layout, state, data flows, and crash recovery for engineers.

Package: `packages/tui` (`@jango-blockchained/hoox-tui`)
Entry: `src/main.tsx` · **Root:** `src/app.tsx` · **CLI launcher:** `hoox tui`

Architectural Blueprint

```
graph TD
    Renderer["OpenTUI Renderer (main.tsx)

alternate screen, mouse/Kitty keyboard"] -->
    AppRoot["AppRoot + CrashRecoveryApp (app.tsx)

Sidebar, StatusBar, Command Palette, quit modal"] -->
    ActiveView["Active view (15 views)

each sandboxed in ErrorBoundary"]

    style Renderer fill:#CCCCCC,stroke:#3b82f6,stroke-width:2
    style AppRoot fill:#CCCCCC,stroke:#f59e0b,stroke-width:2
    style ActiveView fill:#CCCCCC,stroke:#10b981,stroke-width:2
```

Directory map

```
packages/tui/src/
├── main.tsx                # createCliRenderer (TUI_FPS / TUI_MOUSE env)
├── app.tsx                 # VIEWS registry, shortcuts, session, SSE kickoff
├── components/
│   ├── views/              # 15 views (+ dashboard/, config-editor/ subpanels)
│   ├── layout/             # sidebar, statusbar
│   ├── shared/             # palette, crash screen, error boundary, …
│   └── ui/                 # dialog / toast wrappers
├── services/
│   ├── cli-bridge/         # typed hoox subprocess bridge
│   ├── hoox-path-service.ts # $HOME/.hoox paths
│   └── tui-storage.ts      # file-backed JSON (no localStorage in Bun)
├── hooks/                  # keyboard, polling, renderer-ref
└── stores/*.test.ts        # unit tests (stores live in hoox-shared)
```

Stores are **not** under `packages/tui/src/stores` for production code — they live in `@jango-blockchained/hoox-shared` :

Store	Path	Responsibility
UI	`packages/shared/src/stores/ui-store.ts`	activeView, sidebar, modal, palette
Service	`packages/shared/src/stores/service-store.ts`	workers, trades, logs, alerts, connection, SSE
Config	`packages/shared/src/stores/config-store.ts`	theme, refresh interval, notifications, shortcuts

Session persistence: `$HOME/.hoox/.tui-state/session.json` via shared `restoreSession` / `saveSession`.

View registry (15)

Must stay aligned with `ViewId` in `packages/shared/src/types.ts`, sidebar items, `VIEWS` in `app.tsx`, and command palette entries:

ViewId	Component	Primary data source
`dashboard`	`DashboardView`	`fetchWorkers`, `monitorStatus`, `checkFix`, `agentHealthCheck`, kill-switch
`workers`	`WorkersOverview`	store workers + CLI deploy/logs/repair
`worker-detail`	`WorkerDetail`	`selectedWorkerId`, `configShow`, `workerLogs`
`trade-monitor`	`TradeMonitor`	`tradeStream` (SSE `streamTrades`)
`logs-viewer`	`LogsViewer`	`logs` (SSE `streamLogs` + CLI fetch)
`service-manager`	`ServiceManager`	deploy/repair/rebuild/kill-switch
`config-editor`	`ConfigEditor`	filesystem + `configValidate`
`setup-wizard`	`SetupWizard`	wizard steps + `deployAll`
`settings`	`SettingsView`	config store + `checkSetup` / `checkFix`
`queue-depth`	`QueueDepthView`	`monitorQueueDepth`
`kv-viewer`	`KvViewer`	`configKvList` / `configKvGet`
`secrets-viewer`	`SecretsViewer`	`configSecretsList` (names only)
`ai-chat`	`AiChatView`	`agentChatStream` SSE
`db-query`	`DbQueryView`	`validateReadOnlySql` + `dbQuery`
`edge-topology`	`EdgeTopology`	monorepo `graph-metadata.json`

Dual-channel + SSE

Startup sequence (`AppRoot`)

1. Restore session → set view / sidebar
2. `fetchWorkers()` (HTTP) → on failure, `cliBridge.monitorStatus()`
3. Fire-and-forget `streamTrades()` + `streamLogs()` (SSE; silent if API down)

CLI bridge (`src/services/cli-bridge`)

All mutating / diagnostic operator actions go through the `hoox` binary with timeout, abort tags, and structured `CliErrorDetails` sunk to the service store for the status bar expand panel.

Notable methods: `deployAll` , `deployWorker` , `repairWorker` , `rebuild` , `checkHealth` , `checkFix` , `checkSetup` , `monitorStatus` , `monitorKillSwitch` , `monitorQueueDepth` , `workerLogs` , `configShow` , `configValidate` , `configKvList` , `configKvGet` , `configSecretsList` , `dbQuery` , `agentHealthCheck` .

`checkHealthFix` exists on the bridge but is unused by views (dashboard/settings use `checkFix`).

Crash protection

Layer 1 — per-view `ErrorBoundary`

Every primary view is wrapped. Render failures show Retry without killing sidebar/status.

Layer 2 — `CrashRecoveryApp`

Process `uncaughtException` / `unhandledRejection` → CrashScreen:

- **Restart** — remount AppRoot
- **Safe Mode** — remount with `safeMode`
- **Report Bug** — write `$HOME/.hoox/.tui-state/crash.log`

Verification

```
cd packages/tui
bun run typecheck
bun test --preload ./src/test-setup.ts
# from monorepo root:
bun run test:tui
```

E2E smoke (`test/e2e/smoke.test.ts`) requires an interactive TTY; otherwise it skips with a clear reason.

Graph integration

- **Runtime topology view** reads `graph-metadata.json` (repo root), resolved from CWD walk-up or package-relative path.
- **Code graph** (`graph.json` / `graph.dot`) is produced by `bun run graph` (`scripts/extract-graph.ts`).
- All 15 view exports are present as function nodes under `packages/tui:…/views/…` .
- Regenerate after large refactors: `bun run graph` (~25s). Metadata should include every worker under `workers/*` (including `pine-worker`).

Known limitations (honest)

Area	Limitation
Worker Detail DOs	Names inferred from `durableObjectCount` + worker name; no DO listing API
Worker Detail config	Falls back to demo-ish keys when `configShow` empty
Trade/log live feed	Requires HTTP API + SSE endpoints; offline TUI uses CLI snapshots only
`usePolling` hook	Exported but unused — views poll via effects / intervals
CLI flags	`--fps` / `--no-mouse` set `TUI_FPS` / `TUI_MOUSE`; renderer reads them in `main.tsx`
Graph freshness	Full `graph.json` can lag submodules until `bun run graph` is re-run

OpenTUI `children must be strings (or text nodes) — never nest` inside `. Prefer sibling` inside a row ``.

Next steps

- **Setup & Operations**
- **Architecture overview**
- **End-user TUI guide**

37 · REFACTORING

WORKERS PATTERN CONSOLIDATION

Shared factories and standardized patterns for queues, crons, and exchange clients across workers.

Overview

This document describes the pattern consolidation refactorings applied to the hoox-setup workers monorepo. These changes eliminate code duplication and establish standardized patterns for common worker patterns including queue handling, cron scheduling, and exchange client implementations.

Status: Production-Ready
Last Updated: June 4, 2026
Version: 1.0

What Changed

1. Queue Handler Factory

Location: `packages/shared/src/queue-handler.ts`

What: A reusable factory for handling CloudFlare Queues with retry logic and exponential backoff.

Before: Each queue-based worker had inline retry/backoff logic (60+ lines per worker)

After: Single factory used across all workers (30 lines per worker)

Code Reduction:

- `trade-worker`: 61 lines → 30 lines (51% reduction)
- Consistent retry behavior across all queue-based workers

Usage Example:

```
  async queue(batch, env, ctx) {
    const handler = createQueueHandler({
      maxRetries: 5,
      backoffDelays: [0, 30, 60, 300, 900],
      logger,
      onMessage: async (message, attemptNumber) => {
        // Your message processing logic
        const result = await processMessage(message, env, ctx);
        if (!result.success) throw new Error(result.error);
      },
      onRetry: (_message, _attemptNumber, _error, _delaySeconds) => {
        // Logging is handled by createQueueHandler internally
      },
      onDLQ: async (message, attemptNumber, error) => {
        // Handle dead-lettering
        await logFailedMessage(message, error);
      },
    });

    return await handler(batch);
  },
};
```

Key Features:

- Generic message type support (`<T>`)
- Exponential backoff with configurable delays
- Automatic retry logic with attempt tracking
- Dead-letter queue (DLQ) handling
- Built-in logging with optional logger
- Type-safe with TypeScript

Benefits:

- Reduced code duplication (51% reduction in trade-worker)
- Consistent retry behavior across workers
- Type-safe with generic message types
- Built-in logging with `logger` option
- Easier to test and maintain

Updated Workers:

- `workers/trade-worker` - Uses factory for queue message handling

---

2. Cron Handler

Location: `packages/shared/src/cron-handler.ts`

What: Standardized handler for scheduled/cron-triggered workers with consistent logging and error handling.

Before: Custom logging in each scheduled worker, inconsistent patterns

After: Standardized logging with automatic duration measurement

Usage Example:

```
  async scheduled(event, env, ctx) {
    const handler = createCronHandler({
      name: "my-worker",
      logger,
      handler: async (event, env, ctx) => {
        // Your cron logic here
        await runCronTask(env, ctx);
      },
    });

    return await handler(event, env, ctx);
  },
};
```

Logging Output:

```
my-worker cron triggered { cron: "0 * * * *", scheduledTime: 1234567890 }
my-worker cron handler completed successfully { cron: "0 * * * *", durationMs: 145 }
```

Error Logging:

```
my-worker cron handler failed { cron: "0 * * * *", durationMs: 245, error: "Task failed" }
```

Key Features:

- Automatic execution time measurement
- Structured logging with event details
- Consistent error handling and logging
- Generic environment type support (`<Env>`)
- Optional logger instance

Benefits:

- Consistent logging across all scheduled workers
- Automatic duration measurement
- Structured error logging with context
- Cleaner handler code
- Easier debugging with standardized logs

Updated Workers:

- `workers/agent-worker` - Uses factory for 5-minute cron routine
- `workers/report-worker` - Uses factory for report generation cron

---

3. Exchange Client Base Class

Location: `packages/shared/src/exchanges/`

What: Abstract base class for exchange implementations (Binance, Bybit, MEXC) with common functionality.

Files:

- `base-exchange-client.ts` - Abstract class with 6 abstract methods
- `types.ts` - TypeScript types for exchanges
- `index.ts` - Barrel exports

Before: Base class duplicated in trade-worker, no shared interface

After: Single shared base class in packages/shared with consistent interface

Usage Example:

```
  TradeParams,
  OrderResponse,
} from "@jango-blockchained/hoox-shared/exchanges";

  async validateApiCredentials(): Promise<boolean> {
    // Implementation
    const response = await this.makeRequest("GET", "/api/v3/account");
    return !!response.balances;
  }

  async executeTrade(params: TradeParams): Promise<OrderResponse> {
    // Implementation
    const response = await this.makeRequest("POST", "/api/v3/order", {
      symbol: params.symbol,
      side: params.side,
      quantity: params.quantity,
      price: params.price,
    });
    return response;
  }

  async getMarkPrice(symbol: string): Promise<number> {
    // Implementation
  }

  async getOpenPositions(symbol?: string): Promise<Position[]> {
    // Implementation
  }

  async closePosition(
    symbol: string,
    positionId?: string
  ): Promise<OrderResponse> {
    // Implementation
  }

  async getWalletBalance(): Promise<number> {
    // Implementation
  }
}
```

Abstract Methods:

- `validateApiCredentials(): Promise<boolean>` - Verify API credentials are valid
- `executeTrade(params: TradeParams): Promise<OrderResponse>` - Execute a trade
- `getMarkPrice(symbol: string): Promise<number>` - Get current mark price
- `getOpenPositions(symbol?: string): Promise<Position[]>` - Get open positions

- `closePosition(symbol: string, positionId?: string): Promise<OrderResponse>` - Close a position
- `getWalletBalance(): Promise<number>` - Get wallet balance

Protected Helper Methods:

- `makeRequest(method: string, path: string, data?: any): Promise<any>` - Make HTTP request (subclass implements)
- `getErrorMessagePrefix(): string` - Get error message prefix for logging

Key Features:

- Type-safe configuration and responses
- Consistent interface across exchanges
- Helper methods for common operations
- Crypto signature support (HMAC-SHA256)
- Testnet/sandbox support

Benefits:

- Consistent interface for all exchange implementations
- Type-safe configuration and responses
- Reusable across multiple workers
- Helper methods for URL building, signing, etc.
- Easier to add new exchanges

Updated Workers:

- `workers/trade-worker` - Exchange clients (Binance, Bybit, MEXC) extend shared base class

---

Migration Guide

For New Workers with Queues

If you're building a new worker that needs queue handling:

```
interface MyQueueMessage {
  requestId: string;
  action: string;
  data: Record<string, any>;
}

const MAX_RETRIES = 5;
const BACKOFF_DELAYS = [0, 30, 60, 300, 900];

  async queue(batch, env, ctx) {
    const handler = createQueueHandler<MyQueueMessage>({
      maxRetries: MAX_RETRIES,
      backoffDelays: BACKOFF_DELAYS,
      logger,
      onMessage: async (message, attemptNumber) => {
        // Process message
        const result = await processMessage(message, env, ctx);
        if (!result.success) throw new Error(result.error);
      },
      onRetry: (_message, _attemptNumber, _error, _delaySeconds) => {
        // Logging is handled by createQueueHandler internally
      },
      onDLQ: async (message, attempt, error) => {
        // Log to dead-letter queue
        await logToDatabase(message, error);
      },
    });

    return await handler(batch);
  },
};
```

Key Points:

- Define your message type as a TypeScript interface
- Set `maxRetries` and `backoffDelays` as constants
- Implement `onMessage` with your processing logic
- Implement `onDLQ` for dead-letter handling
- The factory handles all retry logic and logging

For New Scheduled Workers

If you're building a new worker with scheduled/cron triggers:

```
interface Env {
  // Your environment bindings
  DB: D1Database;
  KV: KVNamespace;
}

  async scheduled(event, env, ctx) {
    const handler = createCronHandler<Env>({
      name: "my-scheduled-worker",
      logger,
      handler: async (event, env, ctx) => {
        // Your cron logic
        await runRoutine(env, ctx);
      },
    });

    return await handler(event, env, ctx);
  },
};
```

Key Points:

- Define your `Env` type with all bindings
- Set `name` to your worker name for logging
- Implement `handler` with your cron logic
- The factory handles logging and duration measurement
- Errors are logged with context and re-thrown

For New Exchange Clients

If you're adding a new exchange:

```
  TradeParams,
  OrderResponse,
  Position,
} from "@jango-blockchained/hoox-shared/exchanges";

  constructor(apiKey: string, apiSecret: string) {
    super(apiKey, apiSecret);
  }

  async validateApiCredentials(): Promise<boolean> {
    try {
      const response = await this.makeRequest("GET", "/api/account");
      return !!response.id;
    } catch {
      return false;
    }
  }

  async executeTrade(params: TradeParams): Promise<OrderResponse> {
    try {
      const response = await this.makeRequest("POST", "/api/orders", {
        symbol: params.symbol,
        side: params.side,
        quantity: params.quantity,
        price: params.price,
      });
      return {
        orderId: response.id,
        symbol: params.symbol,
        status: response.status,
      };
    } catch (e) {
      throw new Error(`Trade execution failed: ${String(e)}`);
    }
  }

  async getMarkPrice(symbol: string): Promise<number> {
    const response = await this.makeRequest("GET", `/api/ticker/${symbol}`);
    return parseFloat(response.price);
  }

  async getOpenPositions(symbol?: string): Promise<Position[]> {
    const response = await this.makeRequest("GET", "/api/positions");
    return response.positions.map((p: any) => ({
      symbol: p.symbol,
      side: p.side.toLowerCase() as "long" | "short",
      quantity: p.quantity,
      entryPrice: p.entryPrice,
      unrealizedPnl: p.pnl,
    }));
  }

  async closePosition(
    symbol: string,
    positionId?: string
  ): Promise<OrderResponse> {
    const response = await this.makeRequest("POST", "/api/positions/close", {
      symbol,
      positionId,
```

```
    });
    return {
      orderId: response.id,
      symbol,
      status: response.status,
    };
  }

  async getWalletBalance(): Promise<number> {
    const response = await this.makeRequest("GET", "/api/account/balance");
    return response.total;
  }

  protected async makeRequest(
    method: string,
    path: string,
    data?: any
  ): Promise<any> {
    // Implement your exchange-specific HTTP request logic
    // This is where you handle authentication, signing, etc.
    const url = `${this.baseUrl}${path}`;
    const response = await fetch(url, {
      method,
      headers: {
        "X-API-Key": this.apiKey,
        // Add other headers as needed
      },
      body: data ? JSON.stringify(data) : undefined,
    });

    if (!response.ok) {
      throw new Error(`API error: ${response.statusText}`);
    }

    return response.json();
  }
}
```

Key Points:

- Extend `BaseExchangeClient` with your exchange name
- Implement all 6 abstract methods
- Implement `makeRequest` with your exchange-specific logic
- Use the base class constructor to initialize credentials
- Return typed responses for type safety

Testing

All refactored patterns are fully tested:

Test Files

- `packages/shared/src/__tests__/queue-handler.test.ts` - 5 test cases
- `packages/shared/src/__tests__/cron-handler.test.ts` - 5 test cases
- `packages/shared/src/__tests__/exchanges.test.ts` - 3 test cases

Running Tests

```
# Test shared package patterns
bun test packages/shared/

# Test specific pattern
bun test packages/shared/src/__tests__/queue-handler.test.ts

# Test all workers
bun test workers/

# Test specific worker
bun test workers/trade-worker/
```

Test Coverage

All patterns have comprehensive test coverage:

Pattern	Test File	Cases	Coverage
Queue Handler	`queue-handler.test.ts`	5	100%
Cron Handler	`cron-handler.test.ts`	5	100%
Exchange Client	`exchanges.test.ts`	3	100%

Files Modified

Created Files

- `packages/shared/src/queue-handler.ts` - Queue handler factory (92 lines)
- `packages/shared/src/cron-handler.ts` - Cron handler factory (77 lines)
- `packages/shared/src/exchanges/base-exchange-client.ts` - Base class (209 lines)
- `packages/shared/src/exchanges/types.ts` - Exchange types (20 lines)
- `packages/shared/src/exchanges/index.ts` - Barrel exports
- `packages/shared/src/__tests__/queue-handler.test.ts` - Queue tests
- `packages/shared/src/__tests__/cron-handler.test.ts` - Cron tests
- `packages/shared/src/__tests__/exchanges.test.ts` - Exchange tests

Modified Files

- `packages/shared/src/index.ts` - Added exports for new utilities
- `workers/trade-worker/src/index.ts` - Updated to use `createQueueHandler`
- `workers/agent-worker/src/index.ts` - Updated to use `createCronHandler`
- `workers/report-worker/src/index.ts` - Updated to use `createCronHandler`
- `workers/trade-worker/src/mexc-client.ts` - Updated imports
- `workers/trade-worker/src/binance-client.ts` - Updated imports
- `workers/trade-worker/src/bybit-client.ts` - Updated imports

Impact Summary

Metric	Before	After	Change
Queue handler duplications	2 workers	1 shared factory	-50% duplication
Cron handler variations	Custom per worker	Standardized	100% consistency
Exchange client bases	Per-worker	Shared in packages/shared	Reusable
Lines of queue retry code	120 (2 workers)	30 (1 factory)	-75% duplication
Test coverage	80%	85%+	Improved
TypeScript errors	0	0	No regressions

Best Practices Going Forward

1. Use `createQueueHandler` for all queue-based workers

- No manual retry/backoff logic
- Consistent error handling and logging
- Type-safe with generic message types

```
const handler = createQueueHandler<YourMessageType>({
  maxRetries: 5,
  backoffDelays: [0, 30, 60, 300, 900],
  logger,
  onMessage: async (msg) => {
    /* ... */
  },
  onDLQ: async (msg, attempt, error) => {
    /* ... */
  },
});
```

2. Use `createCronHandler` for all scheduled workers

- Standardized logging format
- Automatic duration measurement
- Consistent error handling

```
const handler = createCronHandler<Env>({
  name: "my-worker",
  logger,
  handler: async (event, env, ctx) => {
    /* ... */
  },
});
```

3. Extend `BaseExchangeClient` for new exchanges

- Type-safe configuration

- Consistent interface across exchanges
- Reusable helper methods

```
  async validateApiCredentials(): Promise<boolean> {
    /* ... */
  }
  async executeTrade(params: TradeParams): Promise<OrderResponse> {
    /* ... */
  }
  // ... implement other abstract methods
}
```

4. Keep helper/utility functions organized

- **Factories** in `packages/shared/src/` for framework concerns (queues, cron)
- **Helpers** in `logic/` directories for domain logic (trade execution, notifications)
- **Types** in `types.ts` or `exchanges/types.ts` for shared types

5. Test all patterns before committing

- All pattern tests are in `packages/shared/__tests__/`
- Worker tests cover integration
- Run `bun test` before committing

Troubleshooting

Queue Handler Issues

Problem: Messages not being retried

- Check `maxRetries` is greater than 0
- Verify `backoffDelays` array has enough entries
- Ensure `onMessage` throws error on failure

Problem: DLQ not being called

- Verify `onDLQ` is implemented
- Check that `maxRetries` is being exceeded
- Ensure message attempts counter is working

Cron Handler Issues

Problem: Logs not appearing

- Verify `logger` is passed to handler
- Check logger implementation has `info` and `error` methods
- Ensure logger is properly initialized

Problem: Duration not measured

- Duration is always measured automatically

- Check logger output for `durationMs` field
- Verify handler is actually being called

Exchange Client Issues

Problem: API credentials validation failing

- Verify `apiKey` and `apiSecret` are correct
- Check exchange API endpoint is accessible
- Ensure `makeRequest` is properly implemented

Problem: Trade execution failing

- Verify `TradeParams` are correct for exchange
- Check exchange API documentation for parameter names
- Ensure proper error handling in `executeTrade`

Related Documentation

- Architecture Overview
- Service Bindings
- API Endpoint Directory
- Development Testing

Changelog

Version 1.0 (June 4, 2026)

- Queue handler factory implemented
- Cron handler factory implemented
- Exchange client base class extracted
- All workers refactored to use new patterns
- Comprehensive test coverage added
- Documentation completed

Status: Production-Ready
Last Updated: June 4, 2026
Maintainer: Hoox Development Team