TRADERS · OPERATORS · PLATFORM ENGINEERS

HOOX DOCUMENTATION

Complete manuals — End User, DevOps, and Enterprise

2026-08-27 DIN A4 · HTTPS://HOOX.SH/DOCS OPEN SOURCE

CONTENTS

32 glossary

33 ⚙ DevOps Manual

34 Cloudflare® Workers Setup Flow

35 Hoox Trading System — Complete Setup & Operations Guide

36 Cloudflare® Workers Bindings & Environment Variables

37 Local Development Setup

38 Testing Framework & QA Standards

39 Edge Debugging & Telemetry

40 Production Deployment Runbook

41 CI/CD Pipeline Automation

42 Observability & Telemetry

43 Zero Trust & Security Hardening

44 Private ingress (Access + Tunnel)

45 Security Testing & Hardening

46 CLI Architecture & Features

47 TUI Code Architecture

48 Workers Pattern Consolidation

49 HOOX Enterprise

50 HOOX Enterprise Architecture

51 Multi-Tenancy with Workers for Platforms

52 Enterprise Security & Compliance

53 Enterprise Observability & Audit

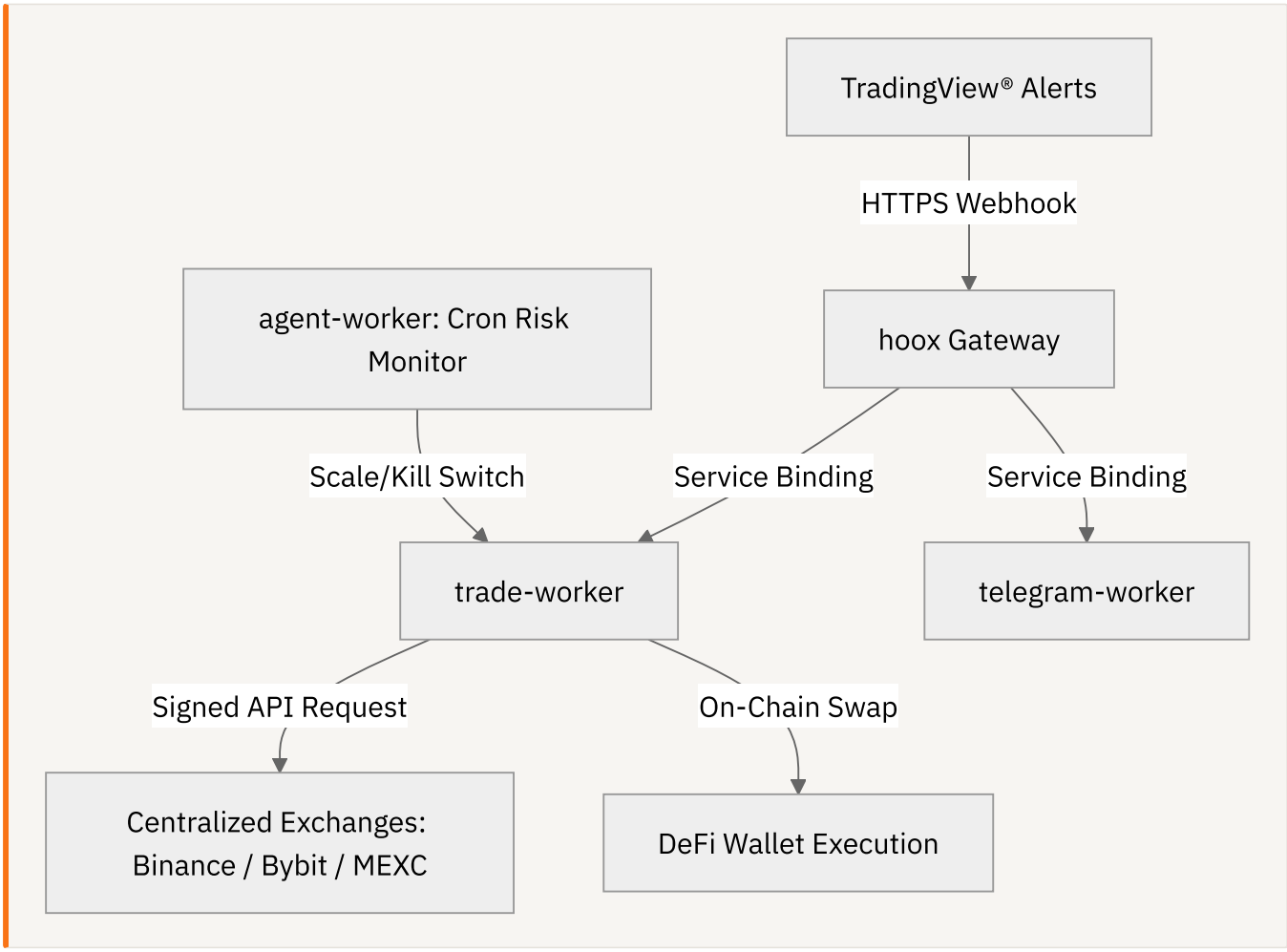
01 · END USER MANUAL

HOOX DOCUMENTATION

Edge-Native Algorithmic Trading on the Cloudflare® Edge

Hoox is a free, open-source, edge-native algorithmic trading framework and automation engine deployed natively to the Cloudflare® Edge Network. By utilizing a distributed microservice architecture, Hoox processes trade signals, evaluates risk parameters, executes order routing, and fires Telegram notifications—all within milliseconds and directly from the edge nodes closest to exchange servers.

Install: `bun add -g @hoox-sh/hoox-cli` → `hoox onboard` (alias `hx`).



Docs are split into **main tabs** (same pattern as AXIS / PYNE): trader-facing **End User**, platform **Architecture** and **Workers**, operators **DevOps**, contracts **API**, and commercial **Enterprise**.

Documentation categories

```
Getting started, concepts, guides, tutorials, CLI and FAQ for traders and
operators of a live mesh.


HOOX vs PYNE vs AXIS vs PyneTS vs pyne-worker — what each surface does.


Strategy events from pyne-worker to trade-worker. Alerts are a separate path.


The PWA evaluates. The mesh executes. They do not share an order router.


Mesh topology, data flow, bindings, storage, and ADRs.


Per-isolate profiles — gateway, trade, D1, agent, telegram, dashboard, and
more.


Install, local development, deployment, CI/CD, security, CLI and TUI.


HTTP endpoints, request payloads (including `test: true`), and response
envelopes.


Multi-tenancy, compliance, and institutional architecture notes (commercial
layer).
```

Popular paths

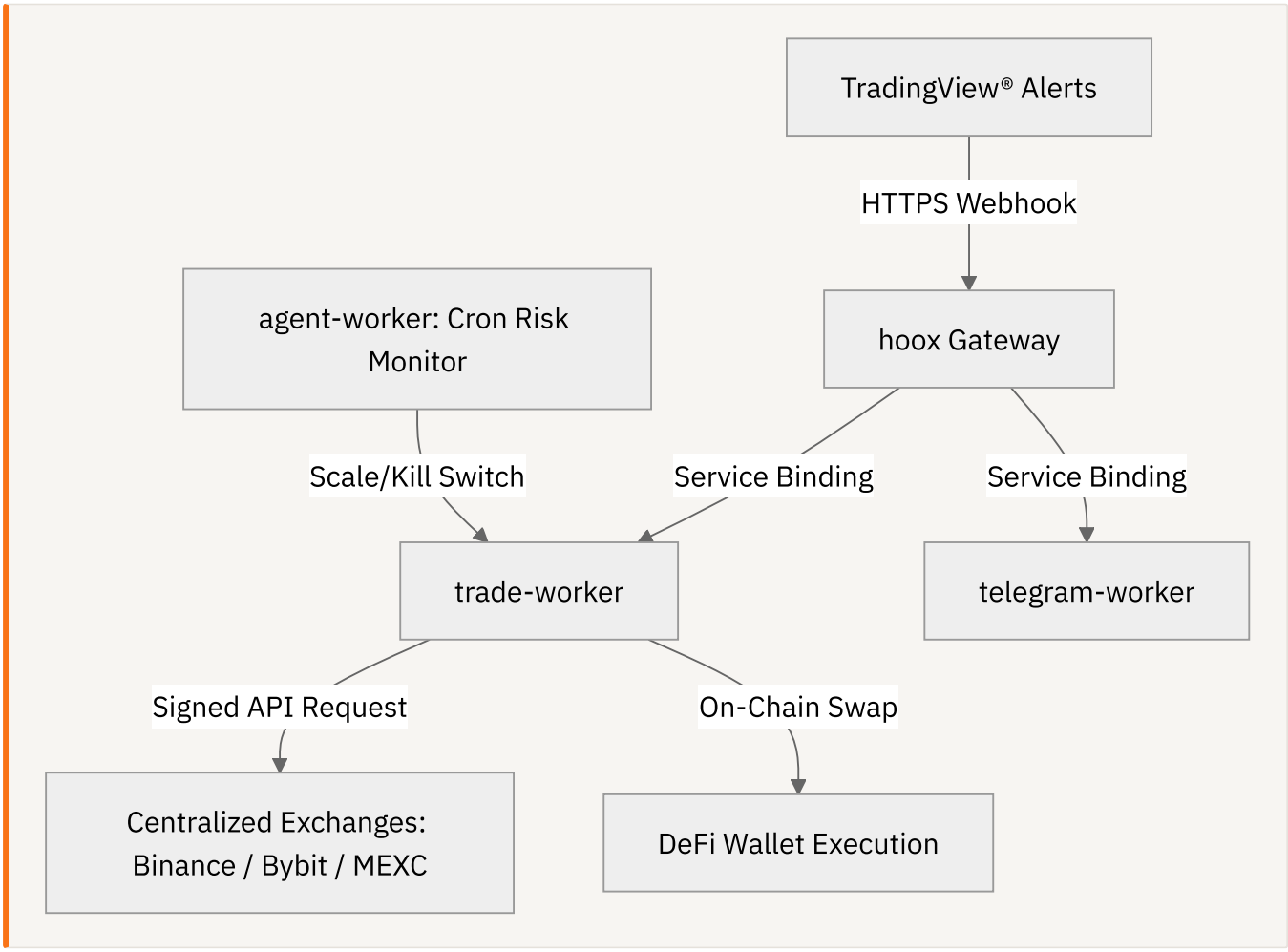
Goal	Start here
Install CLI + first deploy	Installation · Quick Start
Sandbox / testnet orders	Test Trading
How a signal becomes a fill	How Hoox Works
Pine strategy → live order	PYNE live trading
Chart without trading	AXIS and HOOX
Exchange execution isolate	trade-worker
Production rollout	Production Deployment

HOOX DEVELOPER HUB

Edge-Native Algorithmic Trading on the Cloudflare® Edge

> **Hoox** is a free, open-source, edge-native algorithmic trading framework and automation engine deployed natively to the **Cloudflare® Edge Network**. By utilizing a distributed microservice architecture, Hoox processes trade signals, evaluates risk parameters, executes order routing, and fires Telegram notifications —all within milliseconds and directly from the edge nodes closest to exchange servers.

Install with `bun add -g @hoox-sh/hoox-cli`, then `hoox onboard` (alias `hx`).



Choose Your Path

Whether you are a retail algorithmic trader setting up your first automated TradingView® strategies or a quantitative analyst wiring webhooks, this **End User** tab covers trader-facing setup and operations.

Platform design lives under **Architecture** and **Workers**; install/deploy under **DevOps**; HTTP contracts under **API**.

1. 🚀 Getting Started

If you are brand new to the Hoox ecosystem, start here to prepare your machine, provision resources, and deploy your first live microservice in under 5 minutes:

- **Core Installation** — Install `@hoox-sh/hoox-cli`, clone the monorepo, and run `hoox onboard`.
- **Platform Configuration** — Declaratively define environment variables, JSON profile templates, and KV keys.
- **5-Minute Quick Start** — Launch local worker runners and execute a simulated webhook trade signal.

2. 🧠 Core Concepts

Understand the underlying technology, low-latency edge architecture, and security layers that protect your api keys:

- **Product ecosystem** — HOOX vs PYNE vs AXIS vs PyneTS vs pyne-worker.
- **How Hoox Works** — The end-to-end lifecycle of a trade signal from webhook alert to order confirmation.
- **Edge-First Architecture** — Why V8 isolates and Cloudflare® Workers outperform traditional VPS architectures by 30-60%.
- **Cloudflare® Services Map** — How D1, KV, R2, Queues, Vectorize, and Browser Rendering are integrated.
- **Idempotency & Durable Objects** — Two-phase reserve/commit/release, fail-closed missing DO, and rate-limit Durable Objects.
- **Signals & Trade Routing** — How parameters map from webhook JSON schemas to live exchange payloads.
- **AI Risk Manager** — The configurable-cron autonomous risk scanner (1–1440 min), trailing-stop mathematician, and account kill-switch.

3. 🛠 Operational Guides

Practical runbooks and blueprints for daily operations, local development, and system health maintenance:

- **Local Dev & Workspaces** — Run, hot-reload, and test workers locally via Bun or Docker container stacks.
- **Terminal UI Operations** — Launch and navigate the full-screen operations cockpit (`hoox tui` / `@hoox-sh/hoox-tui`).
- **Interface gallery** — Screenshots and GIFs of the CLI, TUI, and dashboard.
- **Edge Database Operations** — Manage global SQLite schemas, track drizzle migrations, and run D1 queries.
- **Secrets & Network Security** — Secure Cloudflare® Zero Trust corridors and encrypt sensitive API credentials.
- **Test Trading** — Sandbox orders with `"test": true`, dedicated testnet secrets, and D1/dashboard isolation.
- **Infrastructure Management** — Spin up and inspect KV, Queues, R2, and Vectorize indexes in one command.
- **Deploying to Production** — Roll out code, bind V8 isolates, and deploy Next.js dashboard consoles.
- **Self-Healing & Repair** — Diagnose connection dropouts, rotate API keys, and run interactive system repair tools.
- **PYNE live trading** — Bar-close cron: strategy events → trade-worker. Alerts are a separate webhook.
- **AXIS and HOOX** — The chart evaluates. The mesh executes. They do not share an order router.

4. 📚 Step-by-Step Tutorials

Follow guided, end-to-end tutorials to integrate your trade sources and notification handlers:

- **TradingView® Webhook Integration** — Write customized Pine Script™ v6 indicators that ping the Hoox webhook receiver.

- **Telegram Bot Setup** — Configure real-time order alerts, P&L reports, and secure chat commands.
- **Email Signal Routing** — Configure email parsing services to convert raw inbox alerts into edge trade executions.

5. □ Reference Material

Deep specifications, schema registries, and command-line manual trees:

- **CLI Commands Index** — Complete options, positional arguments, and JSON flag trees for the `hoox` tool.
- **API Spec & REST Routes** — HTTP endpoints, request payloads, response templates, and edge errors list.
- **Configuration Properties** — Comprehensive dictionary of all 31 env keys and 16 KV key-value items.

□ Offline Reference & AI Context

Consolidated specifications and LLM context packages are available in the GitHub repository.

Looking for deep engineering plans, DDL SQL schemas, or CI/CD deployment workflows? Check out the **DevOps & Architecture Manual** for developer-centric operator blueprints!

03 · GETTING STARTED

INSTALLATION GUIDE

Install the hoox CLI, bootstrap the monorepo with hoox onboard, and verify system prerequisites.

This guide walks you through installing the `@hoox-sh/hoox-cli` tool, cloning the microservices monorepo, running `hoox onboard`, and validating local prerequisites.

System Prerequisites

1. Bun JavaScript Runtime (Version ≥ 1.2)

Hoox uses **Bun** as its package manager, script runner, and test engine. The CLI is a Bun bundle and will not run under Node.

- **macOS / Linux**:

```
curl -fsSL https://bun.sh/install | bash
```

- **Windows (via PowerShell)**:

```
powershell -c "irm https://bun.sh/install.ps1 | iex"
```

2. ⚡ Cloudflare® Account

All Hoox workers run on **Cloudflare® Workers** (V8 isolates). A free account is enough for typical retail volume:

- **Account**: Register free (Workers, D1, KV, Queues, R2, Vectorize — free-tier limits apply).
- **API token**: Created during `hoox onboard` (or pass `--token` / `--account` non-interactively). Permissions: Workers Scripts Edit, D1 Edit, KV Edit, R2 Edit, Queues Edit, AI Read.

The CLI ships with **Wrangler** as a dependency — you do not need a separate global `wrangler` install for normal operator workflows. Prefer `hoox` / `hx` commands over raw Wrangler for provision, secrets, and deploy.

3. Docker & Docker Compose (Optional)

For isolated local mesh / self-host profiles:

- Check status: `docker compose version`

Install the CLI (Recommended)

```
# Install globally using Bun
bun add -g @hoox-sh/hoox-cli
```

Verify the binary (alias `hx` is identical):

```
hoox --version
# e.g. 0.13.x
hx --version
```

Clone the Workspace

Workers are git submodules. Clone recursively:

```
git clone --recursive https://github.com/hoox-sh/hoox.git
cd hoox
```

If you already cloned without submodules:

```
git submodule update --init --recursive
# or: hoox clone --all
```

From source (contributors) — install workspace deps and build the local CLI if needed:

```
bun install
bun run build:cli   # optional; monorepo scripts can invoke packages/cli
./packages/cli/bin/hoox.js --help
```

You MUST use `git clone --recursive` or run `git submodule update --init --recursive` after cloning. If you omit submodules, the worker directories under `workers/` will be empty and setup gates will abort deployment.

Bootstrap with `hoox onboard`

`hoox onboard` is the recommended entry point. It writes `wrangler.jsonc`, collects secrets, generates keys, applies the D1 schema, pushes secrets, and deploys the dashboard (init + setup in one shot).

```
# From the monorepo (first run discovers & remembers the path)
hoox onboard

# Non-interactive
hoox onboard --token cfut_xxx --account xxx --preset full
```

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

```
hoox init    # Step 1: write wrangler.jsonc, collect integration secrets
hoox setup   # Step 2: generate keys, apply D1 schema, push secrets, deploy dashboard
```

The wizard walks through:

1. **Cloudflare® authentication** — API token and Account ID.
2. **Microservice profile** — enable/disable edge workers (gateway, trade, agent, pyne, web3, ...).
3. **Local credentials** — generates `.dev.vars` (mode{" "} `0600`) and initial KV structures.

4. **Infrastructure provisioning** — D1, KV, and related resources.
5. **Setup gates** — aborts if worker trees are still empty after submodule clone, or if init did not produce a valid `wrangler.jsonc`.

```
┌──────────────────────────────────────────────────────┐
│               hoox Setup & Initialization            │
├──────────────────────────────────────────────────────┤
│  ✔ bun found                                         │
│  ✔ git found                                         │
│  ✔ Cloudflare® credentials verified                  │
│                                                      │
│  Enable central Gateway Worker? [Y/n]: y             │
│  Enable Multi-Exchange trade-worker? [Y/n]: y        │
│  Enable agent-worker AI Risk Manager? [Y/n]: y       │
└──────────────────────────────────────────────────────┘
```

Run from any directory (monorepo remember)

After you have used the CLI **once inside the monorepo**, the path is saved to `~/.hoox/config/monorepo.json`. Later you can call `hoox` / `hx` from any folder (e.g. `~/Videos`) and the CLI will `chdir` into the monorepo automatically.

```
# First time: inside the clone (remembers the path)
cd ~/Git/hoox
hoox doctor

# Later: any working directory
cd ~/Videos
hx check setup
hx doctor          # Source: remembered · Runtime root: /…/hoox
```

Resolution order: `HOOX_REPO` env → walk up from cwd → remembered monorepo file → `~/.hoox/repo` (managed global clone via `hoox doctor --fix-runtime`).

Override	Purpose
`HOOX_REPO=/path/to/hoox`	Force monorepo root
`HOOX_HOME=~/.hoox`	Home for config / data / global repo (default)
`HOOX_CLI_SILENT=1`	Hide the one-line “using monorepo at …” notice

Verifying Local Prerequisites

```
# Pre-flight tools on the machine
hoox check prerequisites

# Full bootstrap validation (config, infra, secrets presence, db)
hoox check setup

# Live worker /health probes
hoox check health
```

`hoox check setup` only fails secrets when declared names are **missing on Cloudflare®** (with `hoox secrets sync` hints). Healthy local `.dev.vars` and complete remote name lists stay quiet.

Got installation issues? Run `hoox repair check` to analyze path resolution, missing env vars, or empty worker submodules, and recover the workspace.

Next Steps

- **Configuration Matrix** — Environment variables and KV registries.
- **5-Minute Quick Start Guide** — Deploy and fire a simulated trade webhook.

04 · GETTING STARTED

PLATFORM CONFIGURATION

Master reference for environment variables, Cloudflare® secrets, wrangler V8 profiles, and dynamic KV runtime manifests.

Hoox uses a dual-layer configuration topology: a **declarative build-time environment layer** (managed via local files and Cloudflare® Secrets) and an **instantaneous runtime configuration layer** (managed via Cloudflare® KV key-value databases). This ensures maximum agility: deploy secure code once, and adjust trading parameters or flip kill switches in real-time without redeploying code. Prefer `hoox config` / `hoox secrets` over manual Wrangler edits for day-to-day ops.

---

1. Declarative Environment Variables (`.env.local`)

For local operations, build-time variables, and workspace linking, Hoox loads a `.env.local` file at your project root.

Copy the secure template during initialization:

```
cp .env.example .env.local
```

The Hoox environment is split into 5 core logical sections. Below is the comprehensive dictionary of key configuration parameters:

A. ⚡ Cloudflare® Core Infrastructure

Parameter	Required	Description	Example
`CLOUDFLARE_API_TOKEN`	**Yes**	API token with `D1`, `KV`, `Queue`, and `Workers` write permissions.	`cf_pat_abc123...`
`CLOUDFLARE_ACCOUNT_ID`	**Yes**	Your unique 32-character Cloudflare® dashboard account hash.	`debc6545e63bea36be059cbc82d80ec8`
`SUBDOMAIN_PREFIX`	**Yes**	Subdomain prefix under which your public gateway routes compile.	`hoox-edge`

B. □ Exchange API Keys (Securely Redacted)

These credentials must be injected as secure encrypted environment variables into your Cloudflare® Worker isolates. The Hoox CLI automates this injection (`hoox secrets set`).

- **Binance**:
  - `BINANCE_API_KEY` — Your read/write exchange account trade permission key.
  - `BINANCE_API_SECRET` — Private secret key used to HMAC-SHA256 sign order payloads.
- **Bybit**:
  - `BYBIT_API_KEY` — Order routing account key.
  - `BYBIT_API_SECRET` — Order signing private key.
- **MEXC**:

- `MEXC_API_KEY` — Order routing account key.
- `MEXC_API_SECRET` — Order signing private key.

C. □ Telegram Bot Alerts

Parameter	Required	Description	Example
`TELEGRAM_BOT_TOKEN`	No	HTTP API authentication token provided by Telegram's BotFather.	`123456789:ABCdefGh...`
`TELEGRAM_CHAT_ID`	No	The numeric chat ID of the user, group, or channel where alerts route.	`987654321`

D. □ Multi-Provider AI Credentials

Used to power autonomous risk assessments, Telegram conversation loops, and time-series summaries in `agent-worker`.

- `OPENAI_API_KEY` — Access key for OpenAI GPT models.
- `ANTHROPIC_API_KEY` — Access key for Anthropic Claude models.
- `GOOGLE_AI_API_KEY` — Access key for Google Gemini models.

---

2. Managing Environment & Secrets via CLI

To prevent plain-text exposure and ensure proper edge variable binding, **never** manually paste sensitive keys into `wrangler.jsonc` files. Instead, utilize the high-integrity `hoox config env` command groups:

```
# 1. Start the guided terminal config wizard
hoox config env init

# 2. View active workspace configuration (sensitive credentials automatically redacted)
hoox config env show

# 3. Validate env file structure against required platform variables
hoox config env validate

# 4. Generate local decrypted .dev.vars files for every enabled worker
hoox config env generate-dev-vars
```

Decrypted `.dev.vars` files contain local environment credentials and are excluded from git history via `.gitignore` automatically. Running `hoox config env generate-dev-vars` ensures that local worker testing via `bun run dev` has access to simulated credentials safely.

---

3. Worker Configurations (`wrangler.jsonc`)

Each worker in the monorepo has a standard `wrangler.jsonc` file at its directory root. These configurations declare:

1. **Service Bindings**: Connects gateway routes (`hoox`) to internal compute units (`trade-worker`, `d1-worker`) directly via microsecond V8 isolates—no TCP/TLS overhead, no public routes.

2. **Resource Bindings**: Declares which D1 databases, R2 storage buckets, and KV configurations are linked.
3. **Execution Mode**: Toggles latency-saving features like **Smart Placement** (`"placement": { "mode": "smart" }`).

You can inspect, check, or change worker toggle status directly:

```
# Print complete microservice enabled/disabled status tree
hoox config show

# Declaratively enable/disable specific workers in the manifest
hoox config set workers.web3-wallet-worker.enabled false
```

---

4. Runtime KV Configuration (Sub-millisecond Settings)

One of Hoox's core architectural features is **instant runtime parameter updates**. By using Cloudflare® KV, certain settings can be modified globally in sub-milliseconds and take effect on the very next signal execution—without rebuilding or redeploying any worker.

Hoox manages a standard **16-key runtime manifest** inside the `CONFIG_KV` namespace. Below are the most critical runtime parameters:

KV Key	Type	Default	Description
`trade:kill_switch`	`boolean`	`false`	**Global Kill Switch**. If set to `true`, all incoming trade execution loops immediately halt.
`trade:max_daily_drawdown_percent`	`number`	`5`	Maximum daily drawdown before the AI Risk Manager flips the kill switch.
`webhooks:queue_mode`	`string`	`queue_failover`	Trade delivery behavior: `direct_only`, `queue_failover`, or `queue_everywhere`.
`exchanges:default_routing`	`string`	`bybit`	Default centralized exchange router. Options: `binance`, `bybit`, `mexc`.

Managing KV Settings via CLI

```
# Get the current value of the Global Kill Switch
hoox config kv get trade:kill_switch

# Manually trigger a global trading halt
hoox config kv set trade:kill_switch true

# Restore trading by flipping the switch back
hoox config kv set trade:kill_switch false

# Declaratively apply default manifest variables to Cloudflare® KV
hoox config kv apply-manifest
```

Changed exchange configurations or toggled the kill switch? There is **no need** to redeploy. The changes are distributed across Cloudflare®'s 330+ global edge locations near-instantaneously!

Next Steps

- **5-Minute Quick Start Guide** — Launch local worker runners and execute a simulated webhook trade signal.
- **Deploying to Production** — Deploy and bind all workers in the correct dependency sequence.

05 · GETTING STARTED

5-MINUTE QUICK START

Install the CLI, run hoox onboard, deploy the edge mesh, and fire a simulated trade webhook in under 5 minutes.

This guide gets you from a blank console to a **fully active, edge-deployed algorithmic trading ecosystem** on the Cloudflare® network, processing simulated signals in under 5 minutes.

The recommended path is `bun add -g @hoox-sh/hoox-cli` → clone → `hoox onboard` → `hoox deploy all --auto`. Prefer the CLI over manual Wrangler for provision, secrets, and deploy.

Step-by-Step Deployment Path

Step 0: Install CLI & Clone

```
# Bun ≥ 1.2 required (CLI is Bun-only)
curl -fsSL https://bun.sh/install | bash

bun add -g @hoox-sh/hoox-cli

git clone --recursive https://github.com/hoox-sh/hoox.git
cd hoox
```

Step 1: Onboard Your Workspace

Run the one-shot bootstrap from the monorepo root (or after the CLI has remembered that path — see Installation → Run from any directory):

```
hoox onboard
# alias: hx onboard
```

Interactive prompts will ask for your Cloudflare® Account ID, API Token, and your unique `SUBDOMAIN_PREFIX` (e.g., `alpha-trading`). For non-interactive use, pass `--token` and `--account` flags.

After the first successful discovery, `hx check setup` and other commands work from any cwd.

If you want fine-grained control over each step, run them separately:

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

Setup **gates** stop the flow when init is incomplete (no `wrangler.jsonc`) or worker submodule trees are still empty — fix with `git submodule update --init --recursive` (or `hoox clone --all`) and re-run.

Step 2: Inject Encrypted Exchange API Keys

For your safety, exchange credentials (API keys and private signatures) are never stored in plain text. Inject them as encrypted **Cloudflare® Workers Secrets** bound securely to your compute instances:

```
# One key pair for all venues (Binance / Bybit / MEXC).
# Which exchange is used comes from the signal / routing config — not the secret name.
hoox secrets set trade-worker EXCHANGE_KEY_BINDING "your_exchange_api_key"
hoox secrets set trade-worker EXCHANGE_SECRET_BINDING "your_exchange_api_secret"

# Optional dedicated testnet pair (when signals use "test": true)
# hoox secrets set trade-worker EXCHANGE_TESTNET_KEY_BINDING "..."
# hoox secrets set trade-worker EXCHANGE_TESTNET_SECRET_BINDING "..."
```

Cloudflare® Secrets are encrypted at rest using hardware-level keys and are injected straight into your V8 execution isolates at runtime. They can never be decrypted or read back via the API, ensuring top-tier security for your capital.

Step 3: Deploy All Workers in Sequence

Hoox microservices communicate internally via Service Bindings. The CLI automatically manages the deployment sequence, ensuring databases, queues, and configuration stores compile first, followed by gateway routers and background compute tasks:

```
# Compile and deploy all enabled workers (includes pyne-worker when enabled)
hoox deploy all --auto
```

This command automatically provisions:

1. **D1 Edge Database** (`hoox-db` / trade-data-db)
2. **CONFIG_KV** configuration namespace
3. **Internal Workers** (`trade-worker` ,{" "} `d1-worker` , `telegram-worker` ,{" "} `pyne-worker` , ...)
4. **Public Gateway** (`hoox` gateway router)
5. **Next.js Dashboard Command Center** (`workers/dashboard`)

Once completed, the CLI will output your public Gateway endpoint URL: `https://hoox.alpha-trading.workers.dev`

Step 4: Verify the Deployment

Run the health check to confirm everything is online:

```
hoox check health
```

You should see all enabled workers reporting `healthy` status. To fix any issues automatically:

```
hoox check health --fix
```

Also useful after onboard:

```
hoox check setup   # config / infra / secrets presence (quiet when healthy)
```

Step 5: Fire a Simulated Trade Webhook

Now, fire a test webhook trade signal to your live gateway using `curl`.

```
# Live trade (default)
curl -X POST https://hoox.alpha-trading.workers.dev/webhook \
  -H "Content-Type: application/json" \
  -d '{
    "apiKey": "your-hoox-webhook-passkey",
    "exchange": "bybit",
    "action": "LONG",
    "symbol": "BTCUSDT",
    "quantity": 0.001,
    "leverage": 10
  }'

# Testnet trade (Binance / Bybit only)
# Prefer dedicated secrets: BYBIT_TESTNET_KEY_BINDING / BYBIT_TESTNET_SECRET_BINDING
curl -X POST https://hoox.alpha-trading.workers.dev/webhook \
  -H "Content-Type: application/json" \
  -d '{
    "apiKey": "your-hoox-webhook-passkey",
    "exchange": "bybit",
    "action": "LONG",
    "symbol": "BTCUSDT",
    "quantity": 0.001,
    "leverage": 10,
    "test": true
  }'
```

See **Test Trading** for credential setup, D1 isolation, and dashboard filters.

Step 6 (Optional): Measure Fast-Path Latency

Once live, you can probe the deployed system to measure end-to-end latency:

```
# Send 50 synthetic probes and report p50/p95/p99 per-hop latency
hoox perf fastpath run --n 50
```

This reports per-hop latency for `hoox`, `trade-worker`, and `analytics-worker` so you can identify bottlenecks.

Webhook Payload Parameters Spec

Every webhook payload fired to your Gateway must match the following JSON Schema:

Parameter	Type	Required	Description
`apiKey`	`string`	**Yes**	Your custom webhook authorization passkey (defined in `CONFIG_KV`).
`exchange`	`string`	**Yes**	Target exchange router: `binance`, `bybit`, or `mexc`.
`action`	`string`	**Yes**	`LONG`, `SHORT`, `CLOSE_LONG`, or `CLOSE_SHORT`.
`symbol`	`string`	**Yes**	Standard market symbol.
`quantity`	`number`	**Yes**	Position size / quantity.
`leverage`	`number`	No	Leverage coefficient. Defaults to `1` (spot) if omitted.
`test`	`boolean`	No	When `true`, use exchange testnet (Binance/Bybit). Rejected for MEXC. Prefer `*_TESTNET_*` secrets. Default: live.

Expected Success Response

When a signal arrives, the Hoox Gateway authorizes the request, locks execution via Durable Objects, routes order calculations to the edge node nearest to Bybit's servers, executes the order, and registers the transaction in your D1 SQLite table.

You will receive an instantaneous, low-latency JSON response:

```
{
  "success": true,
  "requestId": "9b1deb4d-3b7d-4bad-9bdd-2b0d7b3dcb6d",
  "exchange": "bybit",
  "symbol": "BTCUSDT",
  "action": "LONG",
  "result": {
    "orderId": "18049284739",
    "status": "Filled",
    "executedQty": 0.001,
    "price": 68425.5,
    "timestamp": 1779261050000
  }
}
```

If the exchange is temporarily undergoing system maintenance or experiences high network congestion, Hoox will automatically intercept the failure, enqueue the trade in **Cloudflare® Queues** with exponential backoff retry policies, and return a `"status": "Enqueued"` response to guarantee delivery!

Next Steps

- **How Hoox Works** — Take a deep dive into the edge processing pipelines.
- **Terminal UI Guide** — Run, hot-reload, and inspect live metrics in a full-screen console interface.
- **Set Up Telegram Alerts** — Link your bot to get order fills pushed straight to your phone.

06 · CONCEPTS

PRODUCT ECOSYSTEM

HOOX, PYNE, PyneTS, AXIS, pyne-worker, and pyne-agent-worker — what each surface does.

PRODUCT ECOSYSTEM

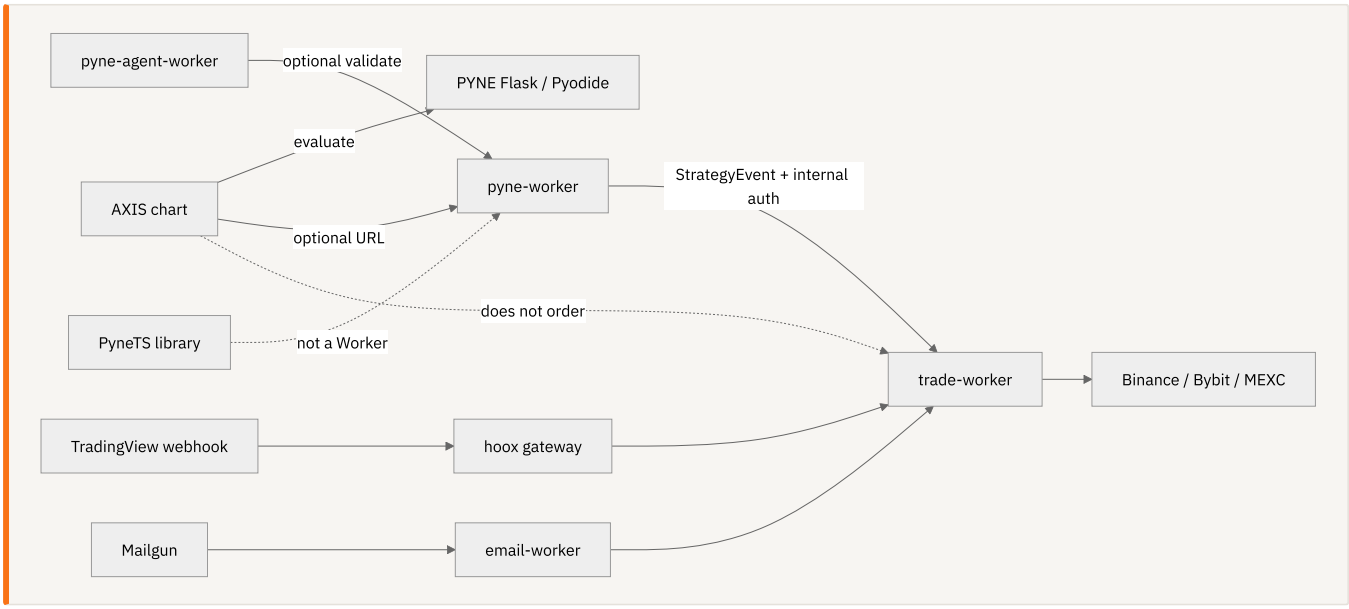
HOOX is the **execution mesh**. It is not the Pine language, and it is not the chart. The family splits those jobs on purpose.

Use this page when a name with "pyne" or "pine" appears in a worker list, a CLI, or a dashboard tile. The wrong hop is usually "I charted a strategy, so it should have traded."

Abstract

Product	Job	Docs
HOOX	Signal in → risk → CEX/DEX hop → notify	this site
PYNE	Pine parse / evaluate (`hoox-pyne` , `Runtime.run`)	hoox.sh/pyne/docs
PyneTS	TypeScript library `@hoox-sh/pynets`	PyneTS
pyne-worker	Python Cloudflare isolate, `POST /run`	isolate · PYNE docs
pyne-agent-worker	Natural language → Pine	agent
AXIS	Charting PWA	hoox.sh/axis/docs

Conceptual model



Interface surface

What executes a live order

Only **trade-worker** holds exchange keys. Paths that reach it:

- HOOX gateway `POST /webhook` (TradingView / HTTP), after `WEBHOOK_API_KEY_BINDING`
- email-worker `POST /email-signal` (Mailgun HMAC or internal JSON)
- telegram-worker commands (allowlisted)
- **pyne-worker** `TRADE_SERVICE` on `StrategyEvent` (`X-Internal-Auth-Key`)

AXIS `engine.run` is **not** on that list.

What evaluates Pine

- PYNE library / CLI / Flask Pro API `:5002`
- pyne-worker `POST /run`
- AXIS Pyodide wheel (offline, no Numba)
- PyneTS `Runtime.run` in **your** TypeScript process

See PYNE modes.

Internals

Repo	Role
`hoox-sh/hoox`	Mesh, CLI, TUI, dashboard
`hoox-sh/pyne`	Language SoT
`hoox-sh/pynets`	TS library
`hoox-sh/pyne-worker`	Edge evaluate isolate (submodule under `workers/pyne-worker`)
`hoox-sh/axis`	PWA
`hoox-sh/pyne-agent-worker`	Authoring agent

Invariants

1. **Alerts are not orders.** `alert()` → `ALERT_WEBHOOK_URL` unless you deliberately point that URL at the gateway.
2. **11 compute surfaces**, not 9: 10 workers + dashboard.
3. Email is **Mailgun**, not IMAP.
4. `apiKey` on a webhook is stripped at the gateway; it is not a trade-worker field.

Worked examples

You want	Do this
Chart a script	AXIS + Flask or Pyodide
Evaluate at the edge	`hoox pyne deploy`
Trade from strategy events	PYNE live trading
Embed TS evaluate	`bun add @hoox-sh/pynets`
Chat → script	pyne-agent-worker plugin in AXIS

See also

- How Hoox works
- PYNE live trading
- AXIS and HOOX
- pyne-worker isolate
- PYNE ecosystem

07 · CONCEPTS

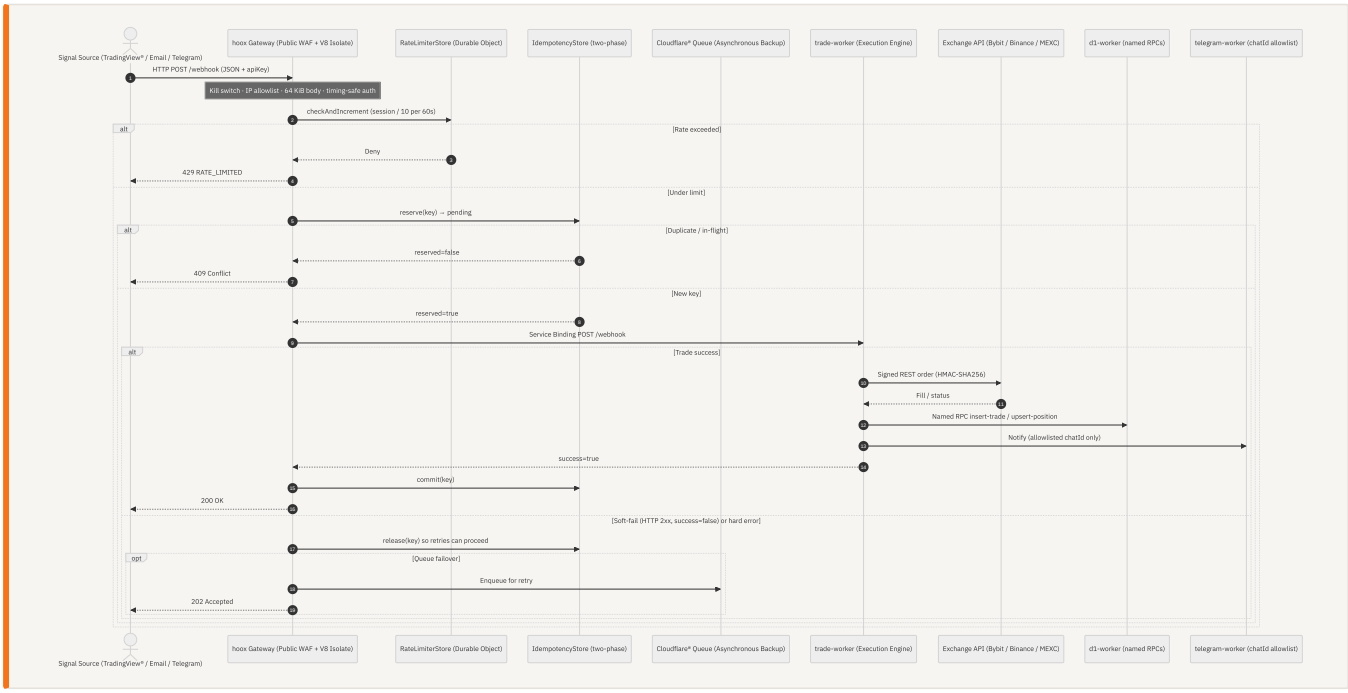
HOW HOOX WORKS

High-level overview of the pipeline turning a trade signal into an executed edge order.

At its core, **Hoox** functions as a highly resilient, low-latency pipeline that intercepts trading signals from external sources, validates and decodes the payloads, and converts them into cryptographically signed order placements on global exchanges in milliseconds.

Here is the exact lifecycle of an edge trade execution, from alert trigger to brokerage fill.

The Execution Pipeline



Detailed Pipeline Phase Analysis

1. Signal Arrival & Ingestion

Signals represent specific instructions to buy, sell, or close positions. Hoox ingests these instructions through three primary channels:

- **TradingView® Webhooks**: High-integrity trade alerts configured via Pine Script™ (`//@version=6`) that issue JSON POST payloads to your gateway's public `/webhook` route.
- **Telegram Commands**: Direct user commands sent to your private Telegram Bot (e.g. `/trade buy btc quantity=0.01 exchange=bybit`), parsed via your webhook router.
- **Email Signals**: Secure email triggers forwarded from specialized alerts (e.g. TradingView® email alerts), parsed natively by `email-worker` .

2. Edge Firewall & Gateway Validation

When a signal hits your public endpoint, the **`hoox` gateway** interceptor immediately evaluates the request inside a sandboxed V8 isolate:

1. **API Key Authorization**: Authenticates the payload's{" "}`apiKey` property against your secure manifest stored in Cloudflare® KV.
2. **IP Allow-listing**: Verifies that the client IP matches a authorized IP range in KV (essential for restricting access solely to TradingView®'s known webhook IPs).
3. **Global Kill Switch check**: Ensures{" "}`trade:kill\_switch` is `false`. If `true`, the pipeline aborts immediately to protect your capital.
4. **Rate Limiting**: Atomic{" "}`RateLimiterStore` Durable Object when bound (else KV / in-memory). Default: **10 requests / 60s** per session. Exceeded → `429`.
5. **Two-phase idempotency**:{" "}`IdempotencyStore` **reserves** the key as{" "}`pending` before dispatch, **commits** only on true success, and **releases** on soft-fail or error so retries can proceed. Missing DO is **fail-closed** (`503`). Duplicates → `409`. Prefer a client{" "}`idempotencyKey` / `Idempotency-Key` header; auto fingerprints include a per-minute time bucket. See{" "}Idempotency.
6. **Notify chat allowlist**: Telegram{" "}`chatId` must be listed when allowlists are configured (fail-closed for unlisted IDs).

---

3. Service Binding Order Routing

Once validated, the gateway bypasses public internet routing and calls the internal **`trade-worker`** directly via **Service Bindings**:

- **Low Latency**: Communication occurs within a microsecond inside the V8 engine, with zero TCP handshakes or TLS decryption overhead.
- **Private Isolation**: The `trade-worker` does not expose any public URL, remaining completely isolated from external attacks.
- **Dynamic Exchange Routing**: The worker parses the symbol (e.g. `BTCUSDT`) and evaluates your KV config. If Bybit is undergoing maintenance, you can instantly redirect trades to MEXC or Binance with one command.

---

4. Exchange Execution & Cryptographic Signing

The `trade-worker` loads your encrypted API credentials from Cloudflare Secrets, computes the HMAC-SHA256 signature for the order parameters, and pushes the signed request to the exchange's nearest API gateway:

- **Smart Placement**: Because Smart Placement is enabled, Cloudflare automatically runs your worker on the physical edge node located geographically closest to the exchange's servers (e.g. Frankfurt for Bybit, Tokyo for Binance), cutting network transit time by up to 60%.

---

5. Persistent D1 Logging & Telemetry

Upon receipt of the order response (e.g. `Filled`, `Partial`, or `Rejected`), Hoox updates your ledger:

- **D1 via named RPCs**: Hot-path writes use fixed templates (`/rpc/insert-trade`, `/rpc/upsert-position`, `/rpc/insert-system-log`) and reads use `/rpc/list-signals`, `/rpc/list-system-logs`, `/rpc/list-open-positions` — not free-form `/query` for trading paths.

- **Background work**: Persistence and analytics ride `safeWaitUntil` so the isolate does not drop work when the response returns.
- **R2 Log Offloading**: Offloads full JSON request-response packets to R2 to keep D1 compact.

---

6. User Notifications & Alerts

Finally, the `trade-worker` pings **`telegram-worker`** via an internal binding:

- You receive an immediate Telegram notification on your phone detailing the symbol, filled price, order status, and transaction ID.
- Twice-daily, `report-worker` uses Browser Rendering to compile beautiful HTML reports of your trading metrics, generating PDFs and dropping the link directly in your chat.

If the exchange API experiences transient network dropouts or severe rate limits, the Hoox Gateway automatically transfers the payload to **Cloudflare Queues** with an exponential retry schedule (`30s` → `1m` → `5m` → `15m`), keeping your strategy fully reliable even during periods of heavy market volatility.

□ Next Steps

- **Edge-First Architecture** — Understand the absolute latency benefits of edge workers vs VPS.
- **Signals & Trade Specifications** — Analyze the complete JSON webhook and order formatting.

EDGE ARCHITECTURE

⚡ EDGE-FIRST ARCHITECTURE

Why running algorithmic trading bots on V8 isolates and Cloudflare's 330+ global data centers cuts latency by 60% and eliminates slippage.

The Physics of Latency: Why Traditional VPS Bots Fail

Traditional trading bots are deployed on a single virtual private server (VPS) in a fixed geographical data center (e.g., Virginia, USA).

The VPS Bottleneck (200ms+ slippage)

1. **Signal Generation**: A TradingView® alert fires in London.
2. **Network Transit**: The alert travels across the Atlantic to your Virginia VPS (~85ms).
3. **Execution Delay**: The VPS boots a heavy Node/Docker runtime to process the signal (~10ms).
4. **Exchange Transit**: The order travels from Virginia to Bybit's API servers in Tokyo or Frankfurt (~110ms).
5. **Total Transit Latency**: **205ms** before the exchange matches your order.

```
[London Alert] ── (85ms) ──> [Virginia VPS] ── (110ms) ──> [Frankfurt Bybit] = 205ms Latency
```

The Hoox Edge Path (Under 15ms latency)

1. **Signal Generation**: A TradingView® alert fires in London.
2. **Edge Ingestion**: The alert hits Cloudflare's nearest London Point of Presence (PoP) in **1.8ms**.
3. **V8 Isolate Processing**: A sandboxed V8 isolate processes and validates the order instantly (**<3ms**).
4. **Smart Placement Routing**: The internal service binding transfers compute to the edge node geographically closest to Bybit's servers (Frankfurt) within **8ms**.
5. **Total Edge Latency**: **Under 15ms** total network transit time.

```
[London Alert] ── (1.8ms) ──> [London PoP] ── (8ms Service Binding) ──> [Frankfurt Node] ── (1ms)
```

V8 Isolates vs. Traditional VMs / Containers

Traditional architectures run on Virtual Machines or Docker containers. These runtimes carry significant memory overhead and introduce **cold starts**—the time required to spin up a container after inactivity.

Architectural Dimension	Traditional VM / Container (VPS)	Cloudflare Edge Worker (V8 Isolate)
Boot Architecture	Heavy guest OS, Node runtime, npm modules	Sandboxed JavaScript V8 Engine runtime
Cold-Start Delay	**1,000ms – 15,000ms** (system stalls)	**< 3ms** (virtually non-existent)
Memory Footprint	256MB – 2GB per instance	**128MB max** (highly lightweight)
Global Failover	Requires complex DNS/Load Balancer setup	Natively distributed across 330+ locations
Geographic Location	Fixed data center	Automatically routes to nearest user node
Scaling Model	Manual provisioning or autoscaling groups	Automatic — each request may hit a different PoP
DDoS Protection	Requires separate service (e.g. Cloudflare)	Built-in at the network edge
Cost Model	Pay for idle capacity 24/7	Pay-per-request; zero cost when idle

---

Smart Placement: Zero-Config Latency Optimizer

To achieve sub-millisecond edge execution, Hoox enables Cloudflare's **Smart Placement** engine natively on all critical workers:

```
"placement": {
  "mode": "smart"
}
```

How Smart Placement Works

1. When you deploy `trade-worker`, Cloudflare monitors its network dependencies. It notices that `trade-worker` makes outbound signed HTTPS REST requests to Bybit's Frankfurt server (`api.bybit.com`) and writes transaction rows to the `d1-worker` SQLite database.
2. Instead of running the worker's CPU compute at the location where the user's browser or alert hits (e.g. California), Smart Placement **automatically shifts the compute isolate** to the Cloudflare node physically closest to the Bybit servers (Frankfurt).
3. The V8 engine performs all signature calculations and database writes at the edge gateway node, reducing the final API transit time to **under 2 milliseconds**.

Latency Comparison: Real Numbers

Scenario	VPS (Virginia)	Edge (Smart Placement)	Improvement
London → Exchange (Bybit/DE)	~205ms	~10.8ms	**19x faster**
New York → Exchange (Binance/US)	~45ms	~8ms	**5.6x faster**
Tokyo → Exchange (Binance/JP)	~120ms	~3ms	**40x faster**
Singapore → Exchange (MEXC/SG)	~85ms	~5ms	**17x faster**

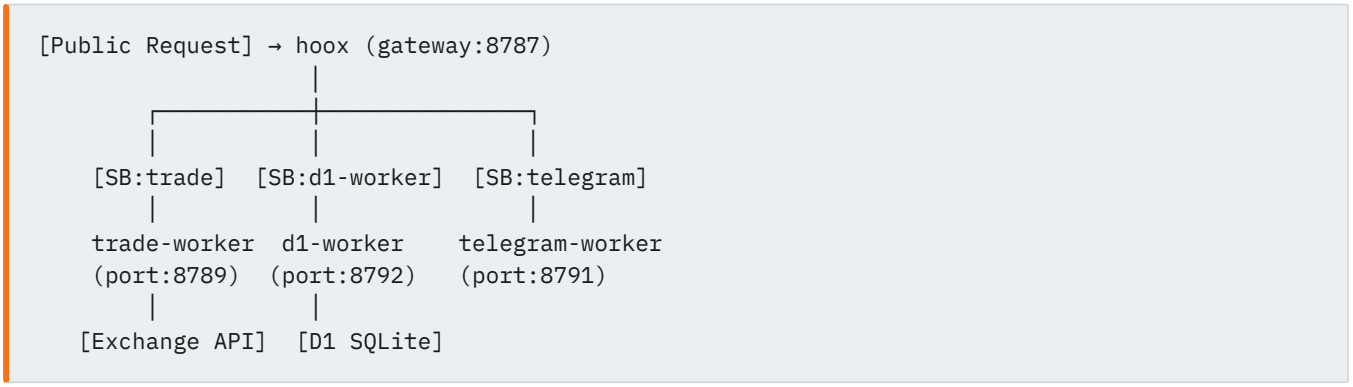
Actual latency depends on exchange API server location, network conditions, and Cloudflare PoP availability. Measurements represent typical round-trip times observed in production.

⚙ Hardware-Level Security

Because V8 isolates run in strictly isolated memory contexts managed directly by Google's Chromium V8 engine, your code is secure:

- **Isolate Protection**: Memory leaks, side-channel attacks, and adjacent tenant exploits are prevented at the V8 compiler level. Each tenant's code runs in a completely separate memory space with zero shared state.
- **Microsecond Service Bindings**: Communication between workers is processed entirely inside the local V8 runtime, meaning sensitive exchange payloads and API calls never travel over the public internet. This eliminates TLS decryption overhead and packet-sniffing risks.
- **No Shared Filesystem**: Unlike container-based approaches, V8 isolates have no persistent filesystem access — preventing supply-chain attacks via compromised dependencies writing to disk.

Service Binding Architecture



All `SB:` prefixed connections occur **inside the V8 engine** with zero TCP handshakes. External connections (Exchange APIs, Telegram Bot API) use standard HTTPS egress.

By removing VPS management, you do not just save latency—you also save operational overhead. Cloudflare natively manages automatic scaling, load balancing, SSL negotiation, and route failovers for free.

Next Steps

- **Cloudflare Services Explained** — Learn how D1 SQLite databases, KV, and Queues fit together on the edge.
- **AI Risk Manager** — Understand how autonomous risk checks run on global cron schedules.

09 · CONCEPTS

CLOUDFLARE® SERVICES MAP

How D1 edge-SQLite, KV, R2, Queues, Durable Objects, Vectorize, and Browser Rendering power our distributed edge monorepo.

Hoox is built natively on a **serverless microservice architecture** powered by a suite of fully integrated Cloudflare® services. By offloading database scaling, messaging retry loops, file storage, and AI inference to Cloudflare®'s global infrastructure, Hoox runs with **zero server maintenance, ultra-low latency, and $0 monthly costs** on the free tier.

Here is an architectural map of how each service functions in the Hoox monorepo.

---

1. D1 Database (Edge SQL Engine)

- **Concept**: A serverless, globally distributed SQLite database engine natively hosted at Cloudflare®'s edge locations.
- **Hoox Implementation**: Used to store critical transactional data, including executed trades ledger, historical win rates, daily asset balances, and position tracking tables.
- **Why it Matters**:
  - Writes are atomic and ACID-compliant.
  - Queries complete in under 5 milliseconds because the database runs in the same V8 isolate context as your worker logic.
  - No need to host, secure, patch, or configure connection pools for a separate database server (like PostgreSQL or MySQL).

---

2. KV Store (Sub-Millisecond Key-Value Configuration)

- **Concept**: A highly available, globally distributed key-value store optimized for high-read/low-write operations.
- **Hoox Implementation**: Houses the global 16-key runtime manifest, including the **Global Kill Switch** (`trade:kill_switch`), exchange routing paths, rate-limiter profiles, and authorized client IP ranges.
- **Why it Matters**:
  - Read operations are cached directly at Cloudflare®'s edge nodes, delivering sub-millisecond lookups.
  - Updates propagate worldwide in under 10 seconds. You can toggle settings in your terminal, and the change affects every worker globally in seconds, without code redeployment.

---

3. Cloudflare® Queues (Asynchronous Messaging Guard)

- **Concept**: A high-integrity, serverless message broker guaranteeing at-least-once delivery with built-in retry schedules.
- **Hoox Implementation**: Acts as an asynchronous buffer and retry buffer for signal executions (`queue_failover` mode).
- **Why it Matters**:

- If a centralized exchange (like Bybit or Binance) undergoes system maintenance or experiences rate limits, the Hoox Gateway intercepts the API error, serializes the trade payload, and enqueues it.
- The queue automatically retries execution with an exponential backoff schedule (`30s` → `1m` → `5m` → `15m` → `30m`). Your trades never get lost due to internet dropouts or API errors.

---

4. Durable Objects (Distributed Coordination & Idempotency)

- **Concept**: Single-threaded, in-memory compute isolates containing persistent local storage, ensuring that exactly one instance of an object runs globally.
- **Hoox Implementation** (gateway `workers/hoox-worker`):
  - **`IdempotencyStore`** (`IDEMPOTENCY_STORE`, migration `v1`) — two-phase **reserve / commit / release** so in-flight retries cannot double-dispatch and failed attempts can retry. Fail-closed if the binding is missing.
  - **`RateLimiterStore`** (`RATE_LIMITER`, migration `v2`) — atomic multi-isolate trade rate limits (default 10 / 60s) with KV/memory fallback when unbound.
  - **`ExchangeConnectionManager`** (trade-worker) — connection coordination; order placement stays **REST-only**.
- **Why it Matters**:
  - TradingView® retries after timeouts hit a single-threaded SQLite mutex, not an eventually consistent KV flag.
  - Rate limits stay correct across concurrent isolates without a shared in-memory counter.

---

5. R2 Object Storage (Zero-Egress Asset Vault)

- **Concept**: A fully S3-compatible, serverless object storage bucket featuring **zero bandwidth egress fees**.
- **Hoox Implementation**: Stores large data payloads, verbose exchange API request/response packets, and compiled HTML/PDF reports.
- **Why it Matters**:
  - Offloading heavy system logging to R2 saves massive write capacity on your D1 SQL database, keeping your database compact and responsive.
  - Zero-egress pricing means you can retrieve, export, and audit your historic trade logs as many times as you like without incurring network charges.

---

6. Vectorize & Workers AI (Embedded RAG & LLMs)

- **Concept**: A serverless vector search database and an on-demand edge AI inference engine running specialized open-source models (like LLaMA 3, Qwen, and Mistral).
- **Hoox Implementation**: Integrates with the Telegram Bot (`telegram-worker`) to provide semantic trade analysis and context-aware natural language conversations.
- **Why it Matters**:
  - User chat queries are converted into high-dimensional vector embeddings, searched against historical trades inside `Vectorize`, and fed into `Workers AI` as context.

- Deliver highly intelligent, context-aware risk analysis directly on your mobile chat without calling expensive external APIs.

---

7. Browser Rendering (Edge Puppeteer Renderer)

- **Concept**: Serverless Chrome instances running at Cloudflare®'s edge, allowing developers to automate browser actions, interact with websites, and convert web elements to documents.
- **Hoox Implementation**: Deployed in `report-worker` to generate twice-daily, styled PDF portfolio reports.
- **Why it Matters**:
  - Reads D1 database P&L records, renders a beautiful, secure HTML5 dashboard report, captures it via Puppeteer, converts it to PDF, saves the file to R2, and links it directly in your Telegram alert.
  - Completely automated, running entirely in the background near-instantaneously.

Every single one of these services is supported by the Hoox CLI! You can check database tables using `hoox db query`, provision R2 buckets with `hoox infra r2 create`, and sync your KV manifest using `hoox config kv apply-manifest` instantly.

Next Steps

- **Idempotency Mechanics** — Dive into the V8 Durable Objects coordination engine.
- **AI Risk Manager** — Understand how autonomous edge cron jobs evaluate drawdowns and manage trailing stops.

IDEMPOTENCY

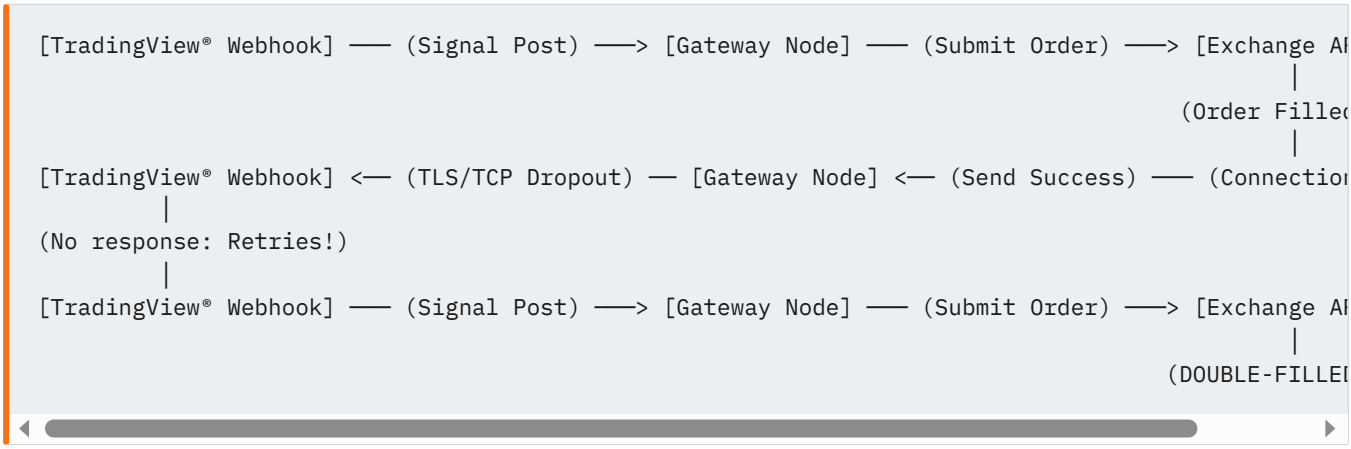
IDEMPOTENCY & DURABLE OBJECTS

How Hoox uses Cloudflare Durable Objects for **at-most-once acceptance** at the gateway (two-phase reserve/commit/release) so webhook retries cannot double-dispatch orders during network dropouts.

⚠ The Danger: How Webhook Retries Lead to Double-Ordering

Gateway **two-phase** flow (v0.13+): **reserve** the key as `pending` before queue/service work, **commit** only when the trade truly succeeds (including soft-fail HTTP 2xx with `success: false` → **release**), so failed attempts can retry. Auto fingerprints include a **per-minute time bucket** so intentional same-size trades later are not blocked for the full TTL. Prefer a client `idempotencyKey` / `Idempotency-Key` header when you need cross-minute uniqueness.

Without idempotency, a typical signal failure sequence looks like this:

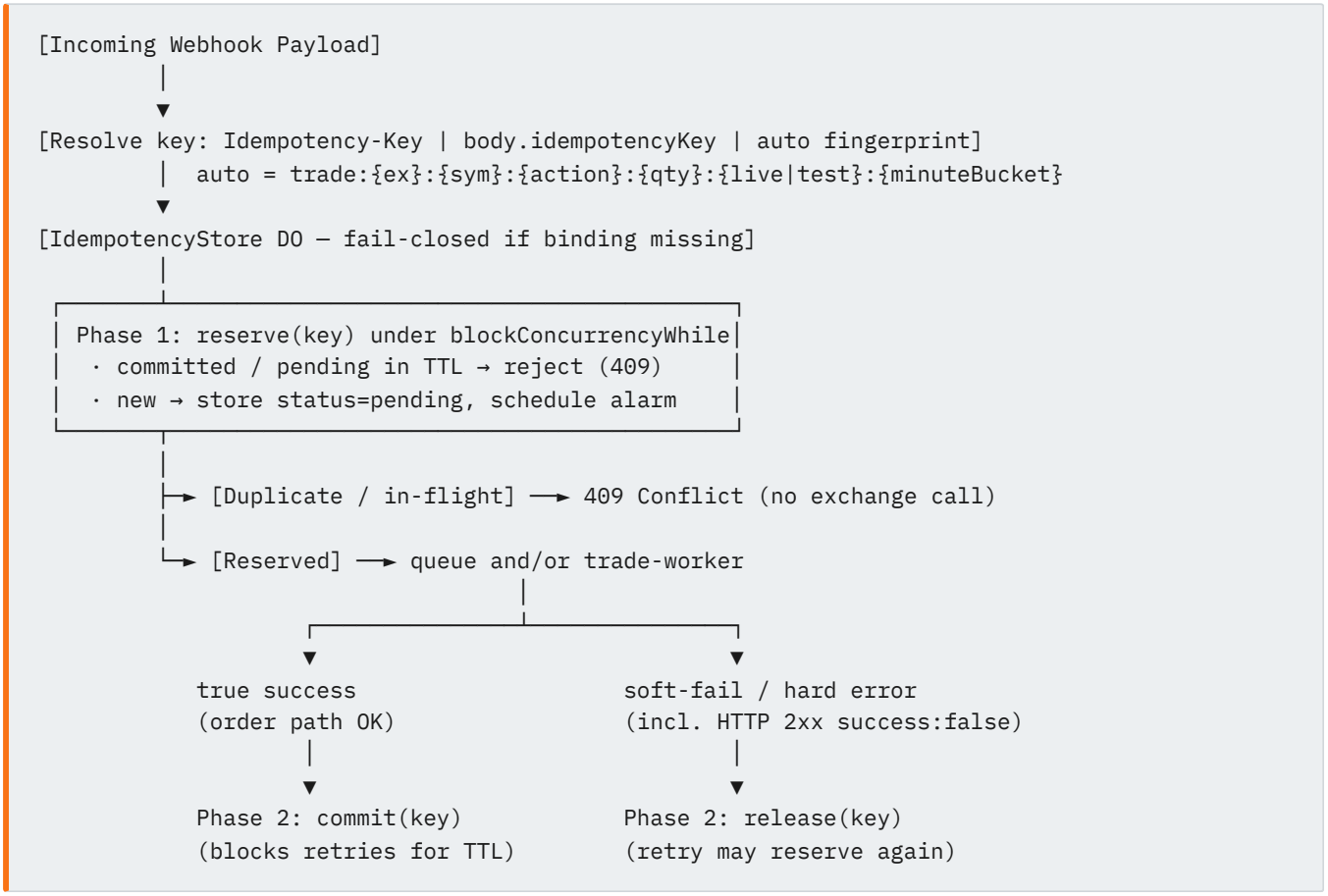


The Hoox Solution: Durable Objects Mutex Locking

To enforce at-most-once acceptance within the TTL, Hoox implements an **atomic two-phase dedup** inside `workers/hoox-worker` utilizing **Cloudflare Durable Objects**.

A Durable Object is a unique, single-threaded compute isolate managed by Cloudflare that maintains its own highly optimized, in-memory state and persistent on-disk SQLite storage. Because access to a specific Durable Object instance is single-threaded, it acts as an absolute **distributed lock** (mutex).

The Idempotency Workflow (two-phase)



The Dedup & Cleanup Algorithm

1. Key resolution

Preference order:

1. Body `idempotencyKey` or HTTP `Idempotency-Key` header (max 256 chars) — use this for cross-minute uniqueness.
2. Auto fingerprint: `trade:{exchange}:{symbol}:{action}:{quantity}:{live|test}:{minuteBucket}` so intentional same-size trades in a later minute are not blocked for the full TTL. Live and test modes never share a key.

2. Atomic two-phase evaluation

Before any queue or service dispatch:

1. The gateway shards to an `IdempotencyStore` instance from the key. Missing `IDEMPOTENCY_STORE` binding → **fail-closed**{" "} (`503` / `IDEMPOTENCY_UNAVAILABLE`).
2. **reserve**: under single-threaded DO execution, pending or committed keys in TTL return duplicate; new keys become{" "} `pending` so concurrent retries cannot double-dispatch.
3. On pipeline **success**: **commit** so further retries stay blocked for the TTL.
4. On soft-fail (`success: false` even with HTTP 2xx) or hard error: **release** so a later retry can reserve again.

3. Automatic TTL & Storage Alarms

- Default window is on the order of **minutes** (configured TTL); alarms reclaim expired keys so storage does not grow without bound.
- Sharded object names keep hot keys from serializing the entire gateway behind one DO instance.

4. Cold-Start Resilience

When a DO instance has been idle (no requests for the TTL period), Cloudflare may evict it from memory. Upon the next request:

1. The DO is **reconstructed** from its persisted SQLite state on disk.
2. All previously recorded trace IDs are loaded back into memory.
3. The dedup check continues seamlessly — no data loss, no duplicate risk.

This means idempotency protection **survives worker restarts, cold starts, and infrastructure failovers** without any manual intervention.

□ Performance Impact

Metric	Value	Notes
DO ping overhead	**< 2ms**	Per-request latency added
Lock acquisition	**< 1ms**	Single-threaded, no contention
Dedup check (cache hit)	**< 0.5ms**	In-memory SQLite query
Storage alarm cleanup	**< 5ms**	Runs in background, non-blocking

Never disable idempotency check bindings in your `wrangler.jsonc` file in production. The performance cost of pinging the DO is less than **2 milliseconds**, while the cost of a duplicate order could be catastrophic.

□ Next Steps

- **Signals & Trade Specifications** — Learn how to configure your Pine Script™ webhook payloads to transmit unique idempotency keys.
- **Platform Security Guides** — Deepen your understanding of Zero Trust headers and edge firewall configurations.

SIGNALS AND TRADES

SIGNALS & TRADE SPEC

This document details the exact specifications, JSON validation schemas, and internal translation logic that occurs when an external trade signal (such as a TradingView® alert or Email parser) is ingested by the Hoox Gateway and mapped to an active exchange order.

1. Webhook Signal Ingestion Schema

Every signal received by the public gateway at `/webhook` must contain a valid, authenticated JSON payload. Below is the strict TypeScript interface and parameter schema validated by the gateway's validation middleware:

```
  apiKey: string; // Authentication token matching CONFIG_KV webhooks:api_key
  exchange: "binance" | "bybit" | "mexc"; // Target exchange router
  action: "LONG" | "SHORT" | "CLOSE_LONG" | "CLOSE_SHORT"; // Position intent
  symbol: string; // Asset symbol (e.g. "BTCUSDT", "ETHUSDT")
  quantity: number; // Order size (amount of asset or contracts)
  leverage?: number; // Optional leverage multiplier (default: 1, max: 125)
  idempotencyKey?: string; // Optional unique signature for dedup (auto-generated if omitted)
  price?: number; // Optional limit price (default: market price)
  /**
   * When true, execute on the exchange testnet/sandbox (Binance Futures
   * testnet, Bybit testnet). MEXC has no public REST sandbox and rejects
   * the request. Prefer dedicated BINANCE_TESTNET_* / BYBIT_TESTNET_* secrets.
   */
  test?: boolean;
}
```

Parameter Rules & Type Constraints

Parameter	Type	Required	Constraints
`apiKey`	`string`	**Yes**	Must match the encrypted `webhooks:api_key` hash in your `CONFIG_KV` namespace
`exchange`	`string`	**Yes**	One of: `binance`, `bybit`, `mexc` — must match a configured exchange router
`action`	`string`	**Yes**	One of: `LONG`, `SHORT`, `CLOSE_LONG`, `CLOSE_SHORT`
`symbol`	`string`	**Yes**	Parsed and standardized — `BTC-USDT`, `BTC_USDT`, and `btc/usdt` all become `BTCUSDT`
`quantity`	`number`	**Yes**	Must be greater than zero; checked against exchange minimum order size
`leverage`	`number`	No	Range: `1` to `125` depending on exchange margin profiles; defaults to `1` (spot)
`idempotencyKey`	`string`	No	If omitted, auto-generated including live/test mode so modes do not dedupe each other
`price`	`number`	No	Limit price; defaults to current market price if omitted
`test`	`boolean`	No	`true` → testnet for Binance/Bybit; MEXC returns `TEST_TRADING_UNSUPPORTED`

Test trading

Exchange	Supports `test: true`	API host when enabled
Binance	Yes	`testnet.binancefuture.com`
Bybit	Yes	`api-testnet.bybit.com`
MEXC	No	Rejected (no public REST sandbox)

Side effect	Live	Test (`test: true`)
D1 trade status	`EXECUTED`	`TEST_EXECUTED`
D1 position id	`{ex}-{sym}-{side}`	`{ex}-testnet-{sym}-{side}`
Telegram label	exchange name	`{exchange} [TEST]`
Analytics exchange blob	`bybit`	`bybit:test`
WebSocket DO	optional	always REST testnet
Agent risk loop	manages OPEN	skips `*-testnet-*` ids
Dashboard stats	counted	excluded from headlines

Credentials: prefer `BINANCE_TESTNET_KEY_BINDING` / `BINANCE_TESTNET_SECRET_BINDING` and `BYBIT_TESTNET_*`. If unset, live bindings are used (warning logged).

Full operator guide: **Test Trading**.

---

2. Dynamic Payload Translation & Side Mapping

Exchanges do not understand actions like `LONG` or `SHORT`. They process order instructions in terms of **`Side`** (`BUY` or `SELL`) and **`PositionSide`** (`LONG` or `SHORT` for multi-margin hedge modes).

The `trade-worker` parses the incoming signal and translates the business logic into exchange-specific API calls:

A. Translation Table (One-Way Margin / Spot)

Action	Translated Exchange Side	Operation Type
`LONG`	**`BUY`**	Opens a long position
`SHORT`	**`SELL`**	Opens a short position
`CLOSE_LONG`	**`SELL`**	Closes an existing long (`reduceOnly`)
`CLOSE_SHORT`	**`BUY`**	Closes an existing short (`reduceOnly`)

B. Hedge-Mode PositionSide Mapping

When running in hedge mode (supported on Bybit and Binance futures), both LONG and SHORT positions can coexist:

```
Hedge Mode:
  LONG PositionSide = LONG  → BUY opens LONG, SELL closes LONG
  SHORT PositionSide = SHORT → SELL opens SHORT, BUY closes SHORT

One-Way Mode:
  BUY always opens/increases, SELL always closes/reduces
```

C. Dynamic Position Resolution

When the action is **`CLOSE`**, the `trade-worker` automatically performs a sub-millisecond edge check:

1. Queries your local D1 transaction ledger to resolve the active position direction for the symbol.
2. If no position is tracked locally, it performs a real-time portfolio balance check against the exchange's private position endpoint.
3. Automatically sets the order quantity to match your current open exposure, ensuring a perfect, slippage-free execution that flattens the position without leaving residual micro-contracts.

---

3. Leverage Scaling & Order Math

If the signal includes a `leverage` parameter greater than `1` (e.g., `"leverage": 10`), the `trade-worker` automatically executes a secure margin transition sequence:

1. **Set Margin Mode**: Configures the symbol's margin structure (Isolated vs. Cross) via the exchange API, matching your manifest defaults.
2. **Set Leverage Coefficient**: Submits a leverage update payload to the exchange prior to routing the order.
3. **Calculate Collateral**: If the order size is defined in USDT terms, the worker scales the execution quantity mathematically: $$\text{Contract Quantity} = \frac{\text{Order Size (USDT)}} \times \text{Leverage}}{\text{Current Market Price}}$$

4. **Precision Rounding**: Automatically rounds the calculated quantity down to match the exchange's strict asset decimal precision requirements, preventing API rejects.

Precision Table by Exchange

Exchange	Price Precision	Quantity Precision	Min Order Size
Binance	Varies by symbol	Varies by symbol	~$10 equivalent
Bybit	0.1 – 0.001	3–6 decimal places	Varies by tier
MEXC	0.1 – 0.00000001	2–8 decimal places	~$5 equivalent

---

4. D1 Database Transaction Ledger

Every filled order is persistently written to your globally distributed edge SQLite table. The schema ensures a clean audit log:

```
CREATE TABLE trades (
  id TEXT PRIMARY KEY,          -- UUID v4 generated by trade-worker
  request_id TEXT NOT NULL,     -- Distributed trace ID from gateway
  exchange TEXT NOT NULL,       -- 'bybit', 'binance', or 'mexc'
  symbol TEXT NOT NULL,         -- e.g. 'BTCUSDT'
  action TEXT NOT NULL,         -- 'LONG', 'SHORT', or 'CLOSE'
  side TEXT NOT NULL,           -- 'BUY' or 'SELL' (exchange-side)
  quantity REAL NOT NULL,       -- Filled asset quantity
  price REAL NOT NULL,          -- Execution entry price (USDT)
  leverage INTEGER DEFAULT 1,   -- Applied leverage multiplier
  fee REAL NOT NULL,            -- Exchange transaction fees (USDT)
  order_id TEXT NOT NULL,       -- Exchange-provided order hash
  status TEXT NOT NULL,         -- 'Filled', 'Rejected', 'Failed', 'Partial'
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE positions (
  id TEXT PRIMARY KEY,
  symbol TEXT NOT NULL UNIQUE,
  side TEXT NOT NULL,           -- 'LONG' or 'SHORT'
  quantity REAL NOT NULL,
  entry_price REAL NOT NULL,
  mark_price REAL,              -- Last known market price
  leverage INTEGER DEFAULT 1,
  unrealized_pnl REAL DEFAULT 0,
  updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

You can query the ledger at any time from your local terminal:

```
# Query the 10 most recent trades in your D1 database
hoox db query "SELECT created_at, exchange, symbol, action, price, quantity FROM trades ORDER BY

# Check open positions
hoox db query "SELECT symbol, side, quantity, entry_price FROM positions WHERE quantity > 0"
```

5. Queue Failover & Guaranteed Delivery

When the exchange API returns an error (timeout, rate limit, HTTP 5xx), the gateway automatically enqueues the trade payload into **Cloudflare® Queues** for guaranteed delivery:

Retry Attempt	Delay	Trigger Condition
1	Immediate	First failure
2	30 seconds	Network timeout
3	1 minute	Rate limit (429)
4	5 minutes	Exchange error (502/503)
5	15 minutes	Extended maintenance
6	30 minutes	Final retry before logging as failed

After max retries, the trade is logged to D1 with `status = 'Failed'` — not lost, but visibly flagged for manual review.

By utilizing time-series logging via Cloudflare® Analytics Engine, Hoox tracks the exact execution duration of each pipeline step. You can audit the latency breakdown (e.g., Gateway auth took `1.2ms`, Trade-worker signing took `0.8ms`, Bybit API roundtrip took `18.5ms`) directly in your Next.js dashboard!

Next Steps

- **Autonomous AI Risk Management** — Learn how agent-worker tracks D1 entries to manage trailing stops and drawdowns.
- **CLI Reference Manual** — Review commands to manage D1 tables, perform dry runs, and tail logs.
- **Error Models & Status Codes** — Understand all gateway error responses.

12 · CONCEPTS

AI RISK MANAGER & AI GATEWAY

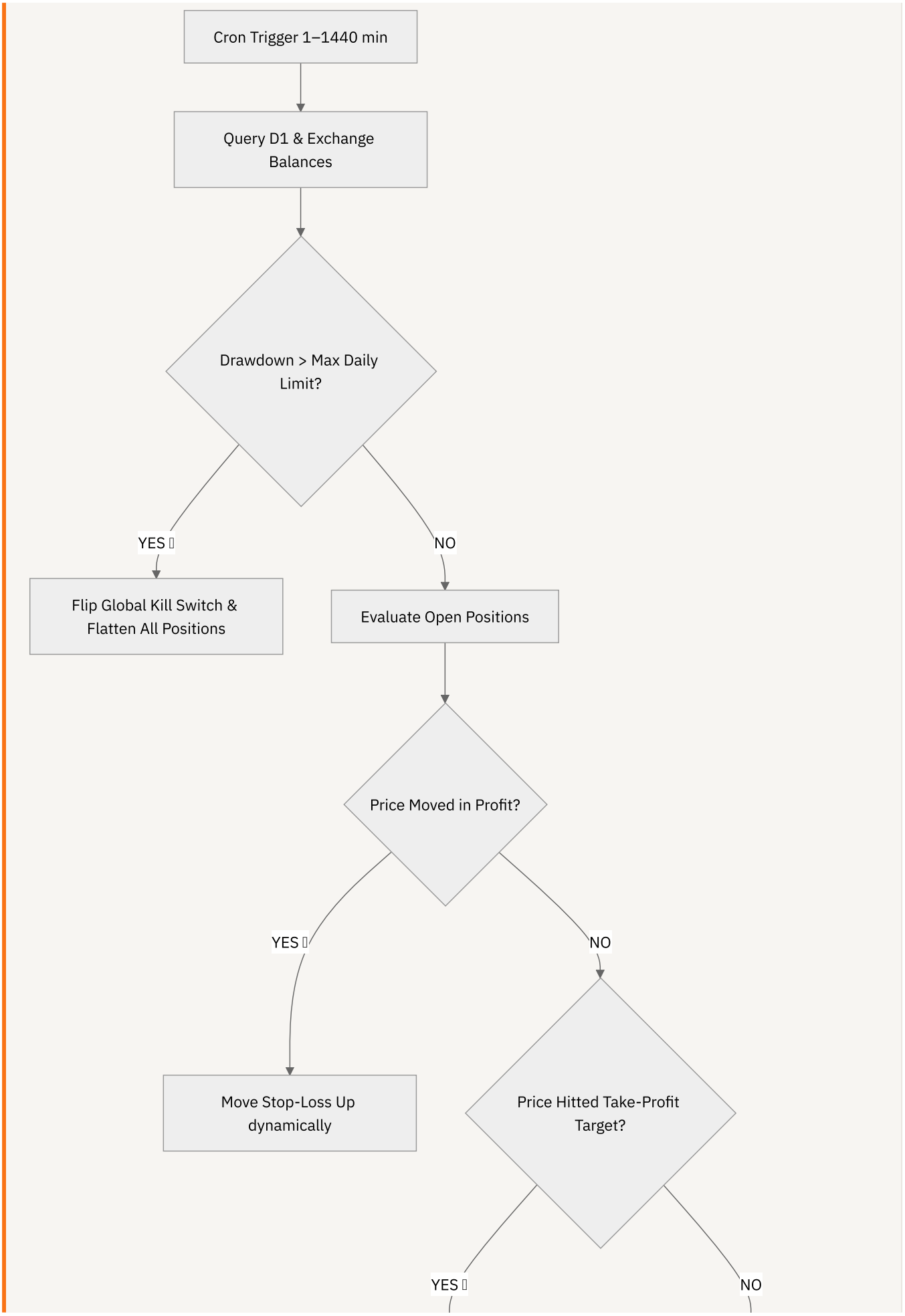
How the agent-worker runs configurable cron checks (1–1440 minutes) to calculate trailing stops, evaluate max daily drawdowns, and govern a multi-provider fallback AI gateway.

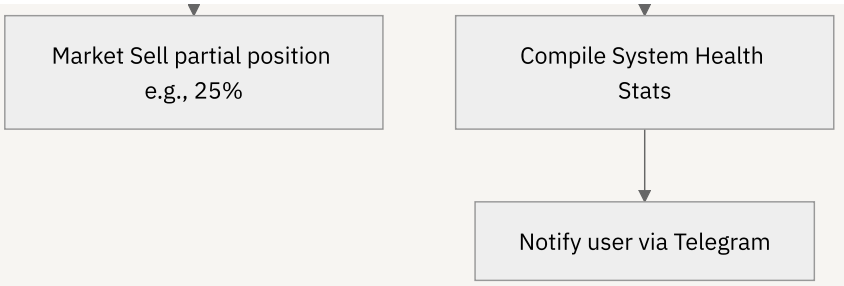
The **`agent-worker`** is the autonomous central nervous system of the Hoox trading platform. Operating on a **configurable Cloudflare® Cron trigger (1–1440 minutes; default every 15 minutes)**, it acts as an intelligent sentinel: it monitors open exchange positions, mathematically scales trailing stop-losses, and deploys a multi-provider fallback AI engine to analyze market conditions, audit risk, and send real-time summaries to your phone.

---

1. Automated Risk Protection Engine

On each cron tick, `agent-worker` executes a multi-point risk diagnostic loop across all active exchanges:


Cron Trigger 1–1440 min
Query D1 & Exchange Balances
Drawdown > Max Daily Limit?
YES
NO
Flip Global Kill Switch & Flatten All Positions
Evaluate Open Positions
Price Moved in Profit?
YES
NO
Move Stop-Loss Up dynamically
Price Hitted Take-Profit Target?
YES
NO



A. Trailing Stop-Loss Mathematics

Rather than using static stops that leave profits exposed, the risk engine calculates a **dynamic trailing stop-loss** at the V8 compiler level:

- **The Formula**: $$\text{Trailing Stop Price (Long)} = \text{Peak Price} \times (1 - \text{Trailing Deviation %})$$
- If a LONG trade entry is at $$100$ and has a $2%$ trailing deviation, the initial stop-loss is placed at $$98$.
- If the market price rallies to a peak of $$120$, the stop-loss is automatically adjusted up to $$117.60$.
- If the price declines from $$120$ down to $$117.60$, the stop is triggered at the exchange level, locking in a $$17.60$ profit. The stop never moves downward.

B. Scaled Take-Profits (Partial Fills)

To manage extreme market swings, the agent-worker supports multi-target scaling:

- **Target 1 ($3%$ Profit)**: Automatically flattens $25%$ of open position size.
- **Target 2 ($5%$ Profit)**: Flattens another $25%$ and moves the stop-loss of the remaining $50%$ to break-even, eliminating downside risk.

C. Max Daily Drawdown & Global Kill Switch

If high volatility causes unexpected stop-outs, the agent-worker aggregates your real-time realized P&L:

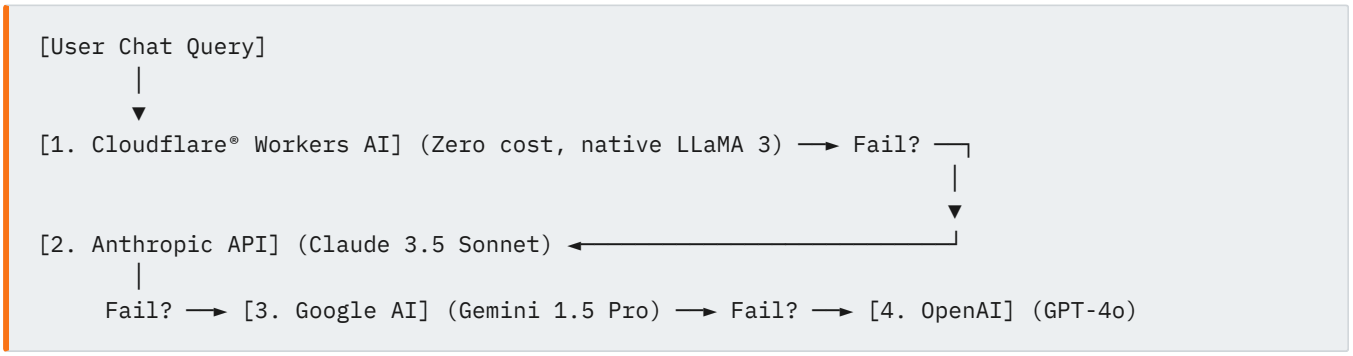
1. Calculates cumulative daily loss: $$\text{Daily Drawdown %} = \frac{\text{Starting Daily Balance} - \text{Current Balance}}{\text{Starting Daily Balance}} \times 100$$
2. If the drawdown exceeds your KV threshold (e.g. `trade:max_daily_drawdown_percent` is `5`), the agent-worker immediately triggers the **Global Kill Switch** by writing `trade:kill_switch = true` to `CONFIG_KV`.
3. Calls the `trade-worker` service binding to submit immediate market close orders, **flattening all active positions across all exchanges** near-instantaneously.

```
# Check current Kill Switch status via CLI
hoox monitor kill-switch show

# Manually override to halt all trade processes during extreme macro events
hoox monitor kill-switch on

# Resume operations once conditions stabilize
hoox monitor kill-switch off
```

🤖 2. Multi-Provider AI Gateway & Reasoning

Behind the scenes, `agent-worker` governs a **Multi-Provider AI Gateway** (supporting 5 major providers) with built-in resilience. If you query the bot for trade advice, market summaries, or risk audits, the gateway processes the request through an automatic fallback chain:

A. The Resilient Provider Fallback Chain



B. Custom Telemetry & Chat Endpoints

The gateway exposes professional endpoints for secure inter-worker and external integrations:

- `POST /agent/chat` — Accepts user prompts and handles SSE (Server-Sent Events) streaming responses.
- `POST /agent/vision` — Analyzes charts, balance screenshots, or trading signals from Base64 images.
- `POST /agent/reasoning` — Extended thinking queries with reasoning models (o1, DeepSeek). Respects `thinking_effort` levels.
- `GET /agent/usage` — Detailed token usage and cost metrics across all 5 providers, including per-provider input/output tokens and cumulative USD cost.
- `GET /agent/health` — Health check for all AI providers — returns latency and status for each provider in the fallback chain.

AI gateway defaults and trailing deviations update in real-time via KV — no redeploy needed for risk profile tweaks. The **cron interval** is set in `wrangler.jsonc` → `triggers.crons` (1–1440 minutes); change it and redeploy the agent-worker. See the agent-worker profile.

🚀 Next Steps

- **Zero Trust & Secret Security** — Lock down your AI endpoints and exchange API keys.
- **TUI Operations Cockpit** — View live position drawdowns and trailing stops visually in your terminal.

13 · GUIDES

LOCAL DEVELOPMENT

How to run, hot-reload, and test Hoox workers locally using native wrangler runtimes or Docker Compose containers.

Hoox provides a comprehensive local development workspace designed to match your production Cloudflare® edge environment. You can run all microservices with hot-reload enabled, monitor them visually via the Terminal UI (TUI), and execute the full test suite using native Bun tools or isolated Docker containers. Prefer `hoox dev` / `hx dev` over calling Wrangler by hand.

Starting the Local Workspace

Run from the monorepo (or any directory after the CLI has remembered that path — see Installation).

To spin up all enabled workers simultaneously:

```
hoox dev start
# alias:
hx dev start
```

On execution, the CLI automatically detects your environment and prompts you to select a runtime:

Option A: Native Runtime (Recommended for speed)

- Runs each worker in a separate background thread using `wrangler dev` (the official Cloudflare® local server; invoked by the Hoox CLI).
- **Speed**: Instant startup and sub-millisecond hot-reloading.
- **Requirements**: Local node/bun installation.

Option B: Docker Runtime (Recommended for isolation)

- Launches a multi-container stack using **Docker Compose**.
- **Isolation**: All environment variables, SQLite databases, and queue handlers run in isolated Linux containers, ensuring zero conflicts with local packages.
- **Requirements**: Docker Desktop installed.

  The CLI saves your runtime preference inside `wrangler.jsonc.dev.runtime`. Subsequent launches skip the prompt. You can override your preference at any time using flags:

```
# Force native execution
hoox dev start --runtime native

# Force Docker Compose execution
hoox dev start --runtime docker
```

Docker Compose Profiles

If you choose the Docker runtime, Hoox manages orchestration using three specialized Docker Compose profiles defined in `docker-compose.yml`:

```
# Profile 1: Workers Only (9 services — all background workers, no UI)
docker compose --profile workers up

# Profile 2: Dashboard + transitive worker deps (5 services: hoox,
#           d1-worker, trade-worker, agent-worker, dashboard).
#           trade-worker is pulled in because agent-worker depends on it
#           for executing risk-approved trades.
docker compose --profile dashboard up

# Profile 3: Full Stack (all workers including pyne-worker + dashboard)
docker compose --profile full up
```

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

Local Port Mapping & Endpoint Access

During local development, all enabled workers are assigned dedicated local ports, simulating service boundaries locally:

| Worker | Local Port | Endpoint URL | Purpose | || :---------- | :--------: | :---------- | :---------- | || **hoox** | 8787 | http://localhost:8787 | Public Gateway & Webhook Receiver | || **trade-worker** | 8789 | http://localhost:8789 | Trade Execution Engine | || **telegram-worker** | 8791 | http://localhost:8791 | Telegram Bot Alerts & Commands | || **d1-worker** | 8792 | http://localhost:8792 | SQLite Database Operations | || **web3-wallet** | 8793 | http://localhost:8793 | On-Chain DeFi Execution | || **dashboard** | 8794 | http://localhost:8794 | Next.js Dashboard Cockpit | || **agent-worker** | 8795 | http://localhost:8795 | AI Risk Manager & Cron Engine | || **email-worker** | 8796 | http://localhost:8796 | Email Signal Parsing | || **report-worker** | 8797 | http://localhost:8797 | PDF Portfolio Report Generator | || **analytics-worker** | 8798 | http://localhost:8798 | Analytics & Reporting Engine | || **pyne-worker** | 8799 | http://localhost:8799 | Pine Script™ edge evaluate (PYNE) |

```
# Probe / evaluate via CLI (when pyne-worker is enabled)
hoox pyne health
hoox pyne run path/to/script.pine
```

Operating Individual Services

If you only want to work on a single microservice rather than running the full stack, you can spin up individual modules:

```
# Dev run a single worker (gateway)
hoox dev worker hoox

# Dev run trade-worker with hot-reloading
hoox dev worker trade-worker

# Dev run dashboard separately (Next.js dev server with Turbopack)
hoox dev dashboard
```

Running the Verification CI Pipeline

Hoox features a rigorous local test pipeline to ensure that all TypeScript types, formatting, and unit tests pass perfectly before pushing to git:

```
# Run the complete CI verification pipeline locally
hoox test
```

The pipeline executes four verification steps in a strict dependency sequence:

1. **Lint Check** (`bun run lint`): Validates ESLint styling rules across the monorepo.
2. **Type Check** (`bun run typecheck`): Compiles code via `tsc --noEmit` to verify type safety.
3. **Unit Tests** (`bun test`): Fires all unit and integration test assertions using Bun's native test runner.
4. **Build Check** (`bun run build`): Compiles all workspaces (`cli`, `tui`, `shared`, `dashboard`) to verify production packaging.

```
 Running local CI Pipeline...
[STARTED] lint check... [PASSED]
[STARTED] TypeScript typecheck... [PASSED]
[STARTED] bun test runner... [PASSED] (~4,600+ unit tests · ~9k assertions)
[STARTED] workspace builds... [PASSED]
✔ Local CI Pipeline Succeeded!
```

Local unit tests utilize Bun's native test runner for instantaneous execution. You can target specific workspace folders or run files individually: `bun test workers/trade-worker/src/index.test.ts`.

Next Steps

- **Testing Standards** — Run unit and integration tests using Bun's native test runner.
- **Terminal UI Operations** — Launch the full-screen terminal cockpit (`./hoox-tui`) to monitor these local runs.
- **Database Migrations** — Set up, query, and migrate your local SQLite D1 database.

INTERFACE GALLERY

Screenshots and short GIFs of the HOOX CLI, TUI, and dashboard — every primary view, command group, and page.

The same stack has three operator surfaces. Stills live under `docs/images/`. GIFs are looping, cropped tours — not recordings of a live exchange.

TUI stills were captured against a local mesh (`localhost:8787`). Dashboard stills use a local Next.js dev server without worker bindings, so metric cards may show empty or skeleton states. Setup, chrome, and navigation are complete.

CLI

```
hx --help
hx onboard --help
hx doctor
```

Still	Command
help-onboard.png	`hoox onboard --help`
help-deploy.png	`hoox deploy --help`
help-tui.png	`hoox tui --help`
doctor.png	`hoox doctor`
help-pyne.png	`hoox pyne --help`
help-secrets.png	`hoox secrets --help`
help-infra.png	`hoox infra --help`

Every top-level group has a `help-<group>.png` in the cli stills folder. Full command tree: CLI commands.

TUI

Launch: `hoox tui` (or `hx tui`). Shortcuts: `Ctrl+1` – `Ctrl+9`, `Ctrl+0`, `Ctrl+Alt+K/S/Q/C/E/W`. See Terminal UI.

View	File	Shortcut
Dashboard	dashboard.png	`Ctrl+1`
Workers	workers.png	`Ctrl+2`
Worker detail (empty selection)	worker-detail.png	`Ctrl+3`
Trade monitor	trade-monitor.png	`Ctrl+4`
Logs	logs-viewer.png	`Ctrl+5`
Service manager	service-manager.png	`Ctrl+6`
Config editor	config-editor.png	`Ctrl+7`
Setup wizard	setup-wizard.png	`Ctrl+8`
Settings	settings.png	`Ctrl+9`
Queue depth	queue-depth.png	`Ctrl+0`
KV (list error / offline)	kv-viewer.png	`Ctrl+Alt+K`
Secrets	secrets-viewer.png	`Ctrl+Alt+S`
DB query	db-query.png	`Ctrl+Alt+Q`
AI chat	ai-chat.png	`Ctrl+Alt+C`
Topology	edge-topology.png	`Ctrl+Alt+E`
Worker settings	worker-settings.png	`Ctrl+Alt+W`

Wizard steps: `setup-wizard-step-01-prereqs.png` … `setup-wizard-step-07-deploy.png`.

Dashboard

```
hoox dev dashboard    # http://localhost:3000
```

Page	File
Setup	setup.png
Setup — welcome / workers / secrets / webhook / done	setup-welcome · workers · secrets · webhook · done
Overview	overview.png
Positions	positions.png
Signal Flow	signal-flow.png
Analytics	analytics.png
Logs	logs.png
Signals	signals.png
Notifications	notifications.png
Reports	reports.png
Database	database.png
Agent	agent.png
Agent chat	agent-chat.png
Vision	agent-vision.png
Reasoning	agent-reasoning.png
Models	agent-models.png
Risk	agent-risk.png
Usage	agent-usage.png
Settings	settings.png

Mobile crops: overview-mobile.png. The `/login` gateway uses a full-viewport shader that does not rasterize reliably in headless capture — use the setup page for the first visual.

Operator isolate notes: Dashboard worker.

15 · GUIDES

TERMINAL UI (TUI)

Master reference for the full-screen terminal operations cockpit, keyboard shortcuts, view registries, and resilience engines.

The **Hoox Terminal User Interface (TUI)** (`@hoox-sh/hoox-tui` **0.3.x**) is a full-screen, keyboard-driven operations cockpit built with **OpenTUI**, React 19, and Zustand. It provides real-time visibility into the edge trading mesh—workers, logs, trades, config, queues, KV, secrets, D1, and AI chat—without leaving the terminal.

Still of every view and wizard step: Interface gallery.

Recent quality pass: **reconnect controller** (status bar `connected` → `reconnecting` → `offline`), **overlay-safe keyboard** handling, Service Manager / Dashboard key ops, fail-closed prefs path, and auth / Cloudflare® Access parity for remote mode.

Launching the TUI

```
# 1. Recommended: via the hoox CLI (install TUI if missing)
bun add -g @hoox-sh/hoox-cli @hoox-sh/hoox-tui
hoox tui
# alias: hx tui

# 2. Optional flags (forwarded as env to the renderer)
hoox tui --fps 60
hoox tui --no-mouse

# 3. Remote management plane (fail-closed auth)

hoox tui --remote --api-url https://mgmt.example.com

# 4. Development: package entry
cd packages/tui && bun run dev

# 5. Built bundle
cd packages/tui && bun run build && bun run start
```

Requirements: Bun ≥ 1.2, 256-color terminal, minimum **80×24**, OpenTUI installed via `bun install`.

Persistent UI state lives under `$HOME/.hoox/.tui-state/` (session, crash log, chat history, DB query history).

⌨ Global Keyboard Navigation

Shortcut	Action
`Ctrl+1`	**Dashboard** — system health overview
`Ctrl+2`	**Workers Overview** — 2-column worker cards
`Ctrl+3`	**Worker Detail** — metrics / logs / DOs / config (select a worker first)
`Ctrl+4`	**Trade Monitor** — live fills + positions + performance
`Ctrl+5`	**Logs Viewer** — filtered log stream
`Ctrl+6`	**Service Manager** — deploy / repair / kill-switch / edge map
`Ctrl+7`	**Config Editor** — TOML/JSON editor + validate
`Ctrl+8`	**Setup Wizard** — onboarding
`Ctrl+9`	**Settings** — preferences, check setup, fix
`Ctrl+0`	**Queue Depth** — backlog visualization
`Ctrl+Alt+K`	**KV Viewer** — read-only KV browser
`Ctrl+Alt+S`	**Secrets Viewer** — secret names only (never values)
`Ctrl+Alt+C`	**AI Chat** — streaming agent chat
`Ctrl+Alt+Q`	**DB Query** — read-only D1 SQL
`Ctrl+Alt+E`	**Edge Topology** — graph-metadata mesh map
`Ctrl+P`	Command palette (fuzzy)
`Ctrl+B`	Toggle sidebar
`Ctrl+R`	Refresh worker data (**force reconnect** if offline)
`Ctrl+Q`	Quit (confirmation dialog)
`Ctrl+Shift+D`	Toggle status-bar error diagnostics
`Esc`	Close palette / dismiss modal / go back
`Tab` / `Shift+Tab`	Cycle focus
`Enter`	Select / confirm
`Space`	Toggle (e.g. pause trade feed)
`/`	Search (searchable views)

Service Manager keys (`Ctrl+6`)

Key	Action
`↑` `↓`	Select worker
`d` / `r`	Deploy / restart selected
`D` / `R` (Shift)	Deploy all / restart all
`k` / `e` / `u`	Kill-switch refresh / engage / release

Dashboard keys (`Ctrl+1`)

Key	Action
`←` `→`	Focus worker health card
`Enter`	Open focused worker detail
`↑` `↓` / `x`	Alerts: navigate / dismiss

If `Ctrl+Q` is swallowed by XON/XOFF flow control: `stty -ixon`, or use **Ctrl+P → QUIT HOOX**.

All 15 Views

1. Dashboard (`Ctrl+1`)

Service health grid, kill-switch status, auto-repair (`hoox check fix`), AI model health, alerts, quick stats (P&L, trades, AI calls).

2. Workers Overview (`Ctrl+2`)

2-column cards: status, uptime, CPU, memory, requests, DO count, edges. **Enter** opens Worker Detail. Actions: logs / deploy / repair via CLI bridge.

3. Worker Detail (`Ctrl+3`)

Four panes for the **selected** worker (select from Workers first): metrics, live logs, Durable Objects (names inferred from count), config preview (`hoox config show` with fallbacks).

4. Trade Monitor (`Ctrl+4`)

Live trade feed (SSE `/trades/stream` when API is up), open positions, today/7d/30d P&L, win rate, Sharpe. **Space** pauses the feed.

5. Logs Viewer (`Ctrl+5`)

Level + worker filters, text search, pause, fetch via `hoox` logs, export to `~/.hoox/logs-export-*.json`. SSE `/logs/stream` when API is up.

6. Service Manager (`Ctrl+6`)

Per-worker deploy/repair, deploy-all, rebuild, kill-switch show/engage/disengage, interactive PoP-style edge map.

7. Config Editor (`Ctrl+7`)

File tree + TOML/JSON syntax highlighting, live validate (`hoox config validate`), format on demand.

8. Setup Wizard (`Ctrl+8`)

Guided onboarding (Cloudflare®, exchanges, AI, strategies, Telegram) ending in deploy.

9. Settings (`Ctrl+9`)

Dark-only theme note, refresh interval, default view, notifications, keyboard shortcut reference, **Check Setup** / **Check Fix**, and a **read-only connection readout** (mode, API host, transport, auth presence — no secrets). Change transport via `hx config transport` / env; probe with `hx doctor --security`.

10. Queue Depth (`Ctrl+0`)

Queue backlog snapshot via `hoox monitor queue-depth` (auto-refresh while focused).

11. KV Viewer (`Ctrl+Alt+K`)

Read-only list + on-demand value reveal for non-secret keys (`config kv list` / `get`). Writes stay CLI-only.

12. Secrets Viewer (`Ctrl+Alt+S`)

Read-only **names and metadata only** — values are never fetched or shown.

13. AI Chat (`Ctrl+Alt+C`)

Streaming chat with agent worker models. History persisted under `.tui-state/chat-history.json` (Bun has no `localStorage`).

14. DB Query (`Ctrl+Alt+Q`)

Read-only SQL (`SELECT` / `WITH` / `EXPLAIN` only). Client-side validation + CLI enforcement. History under `.tui-state/db-query-history.json` .

15. Edge Topology (`Ctrl+Alt+E`)

Interactive map from monorepo `graph-metadata.json` (workers, infrastructure, communities, data flows). Resolves the file from CWD or monorepo root.

Data channels

Channel	Role
HTTP	`fetchWorkers` + API when `HOOX_API_URL` is reachable
CLI bridge	Spawns `hoox …` for deploy, logs, config, queues, KV, secrets, D1, health
SSE	Trade + log streams (`streamTrades` / `streamLogs`) started after session restore when the API is available

Connection state machine: `connected` → `reconnecting` → `offline` (status bar pills + expandable CLI error panel). The reconnect controller retries automatically; use `Ctrl+R` or the command palette **reconnect** action to force a cycle.

Crash protection

1. **Per-view ErrorBoundary** — a broken view shows Retry; sidebar/status keep running.
2. **CrashRecoveryApp** — process-level trap → Restart / Safe Mode / Report Bug → `$HOME/.hoox/.tui-state/crash.log` .

Troubleshooting

Symptom	Fix
Garbled prompt after exit	`reset` or `tput reset`
Ctrl+Q ignored	`stty -ixon` or command palette Quit
Layout broken	Resize to ≥ 80×24; `export TERM=xterm-256color`
OFFLINE forever	Start mesh / set `HOOX_API_URL`, ensure `hoox` on PATH
Topology empty	Run from monorepo root or `bun run graph` to refresh metadata

Next steps

- **Local Development** — API / wrangler mesh that feeds the TUI
- **CLI Reference** — commands the TUI invokes
- **TUI architecture (DevOps)** — stores, backoff, crash model

DATABASE-OPS

DATABASE OPERATIONS

Hoox utilizes **Cloudflare® D1**—a fully serverless, highly optimized SQLite database engine distributed globally across Cloudflare®'s edge network. This document serves as your operational runbook for executing database schemas, running schema migrations, querying transaction ledgers, and performing secure database backup and recovery. Prefer `hoox db` over raw Wrangler D1 commands.

The Core Database Tables

The database schema defines five fundamental tables designed for trading operations, plus supporting indices for fast query performance:

Table	Purpose	Row Estimate (Active User)
`trades`	The primary ledger. Execution prices, contract quantities, timestamps, fees, and exchange order IDs.	~10,000/mo
`positions`	Tracks open margin/futures exposure (average entry price, size, leverage, direction).	~50–200
`balances`	Periodic snapshots of account equity, margin balance, and available free collateral.	~30/day
`trade_signals`	Historical record of every raw incoming webhook alert before execution processing.	~10,000/mo
`system_logs`	Crucial debug and error messages offloaded from compute nodes.	~5,000/mo

Schema Details

```
-- Core trades ledger
CREATE TABLE trades (
  id TEXT PRIMARY KEY,
  request_id TEXT NOT NULL,
  exchange TEXT NOT NULL,           -- 'bybit', 'binance', 'mexc'
  symbol TEXT NOT NULL,
  action TEXT NOT NULL,             -- 'LONG', 'SHORT', 'CLOSE'
  side TEXT NOT NULL,               -- 'BUY' or 'SELL'
  quantity REAL NOT NULL,
  price REAL NOT NULL,
  leverage INTEGER DEFAULT 1,
  fee REAL NOT NULL DEFAULT 0,
  order_id TEXT NOT NULL,
  status TEXT NOT NULL,             -- 'Filled', 'Rejected', 'Failed', 'Partial'
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Active positions tracking
CREATE TABLE positions (
  id TEXT PRIMARY KEY,
  symbol TEXT NOT NULL UNIQUE,
  side TEXT NOT NULL,
  quantity REAL NOT NULL,
  entry_price REAL NOT NULL,
  mark_price REAL DEFAULT 0,
  leverage INTEGER DEFAULT 1,
  unrealized_pnl REAL DEFAULT 0,
  updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Account balance snapshots
CREATE TABLE balances (
  id TEXT PRIMARY KEY,
  total_balance REAL NOT NULL,
  available_balance REAL NOT NULL,
  unrealized_pnl REAL DEFAULT 0,
  snapshot_time TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Incoming signal audit trail
CREATE TABLE trade_signals (
  id TEXT PRIMARY KEY,
  raw_payload TEXT NOT NULL,
  parsed_symbol TEXT,
  parsed_action TEXT,
  received_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- System diagnostic logs
CREATE TABLE system_logs (
  id TEXT PRIMARY KEY,
  level TEXT NOT NULL,              -- 'info', 'warn', 'error'
  source TEXT NOT NULL,             -- Worker name
  message TEXT NOT NULL,
  metadata TEXT,                    -- JSON blob of additional context
```

```
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

⚡ Applying Schemas & Tables

When launching a new workspace, you must initialize the required tables. The Hoox CLI handles this via declarative migration scripts:

```
hoox db apply

# 2. Deploy schema and seed tables directly to live production Cloudflare® D1
hoox db apply --remote
```

Migration Files Location

Migrations are stored in `workers/d1-worker/src/drizzle/` and are automatically versioned:

```
workers/d1-worker/src/drizzle/
├── 0001_initial_schema.sql
├── 0002_add_leverage_column.sql
├── 0003_balances_snapshot_table.sql
└── meta/
    └── snapshot.json
```

Running `hoox db apply` compiles migrations and seeds the database locally. For production deployment, you **must** append the `--remote` flag to execute operations directly on your active Cloudflare® D1 instance.

Managing Database Migrations

When new features are added that require changes to the database structure, you must run migrations:

```
# List all applied and pending database migrations in production
hoox db migrate status --remote

# Apply all pending schema migrations sequentially to production D1
hoox db migrate --remote

# Rollback the most recent migration (use with caution)
hoox db migrate rollback --remote
```

The migration engine tracks history inside a special `d1_migrations` table, ensuring that migrations are never executed twice or applied in the wrong sequence.

Migration Best Practices

1. **Always write reversible migrations** — include both{" "} `UP` and `DOWN` scripts.
2. **Never modify existing migrations** — create new ones for schema changes.
3. **Test locally first** — run `hoox db apply` without{" "} `--remote` to verify.

4. **Backup before migrating** — export your production database first.

Querying Data from the CLI

The Hoox CLI features a built-in SQL interface allowing you to run arbitrary queries directly against your local or remote database:

```
# A. List all active tables in your production database
hoox db list --remote

# B. Count the total number of executed trades
hoox db query "SELECT COUNT(*) FROM trades" --remote

# C. Inspect the 5 most recent trade fills with formatting
hoox db query "SELECT created_at, symbol, action, price, quantity FROM trades ORDER BY created_at

# D. Check currently open positions on Bybit
hoox db query "SELECT symbol, side, size, entry_price FROM positions WHERE size > 0" --remote

# E. Calculate daily P&L
hoox db query "SELECT DATE(created_at) as day, SUM(CASE WHEN side='BUY' THEN -price*quantity ELSE

# F. Export trades as CSV
hoox db query "SELECT * FROM trades ORDER BY created_at DESC" --remote --format csv > trades_expo
```

Useful Query Templates

```
-- Win rate calculation
SELECT
  COUNT(CASE WHEN realized_pnl > 0 THEN 1 END) * 100.0 / COUNT(*) AS win_rate_pct
FROM trades WHERE status = 'Filled';

-- Average trade size by exchange
SELECT exchange, AVG(quantity * price) as avg_trade_size
FROM trades WHERE status = 'Filled'
GROUP BY exchange;

-- Hourly trading distribution (for optimal timing)
SELECT strftime('%H', created_at) as hour, COUNT(*) as trade_count
FROM trades WHERE status = 'Filled'
GROUP BY hour ORDER BY hour;
```

Backup & Export Workflows

To secure your historical P&L records, transaction ledger, and bot performance telemetry, execute regular exports:

```
# Export the entire D1 database as a clean SQL script
hoox db export --remote

# Export to a specific file path
hoox db export --remote --output backups/my-backup.sql
```

Export Output & Structure

The export command creates a timestamped SQL dump inside your workspace directory: `backups/db-backup-2026-05-19-174000.sql`

This file contains standard DDL and DML commands:

```
PRAGMA foreign_keys=OFF;
BEGIN TRANSACTION;
CREATE TABLE IF NOT EXISTS trades (...);
INSERT INTO trades VALUES(...);
COMMIT;
```

Restoring from Backup

```
# 1. Create a fresh database first
hoox infra d1 create hoox-db-backup

# 2. Apply the backup SQL
bunx wrangler d1 execute hoox-db-backup --file=backups/db-backup-2026-05-19.sql
```

⚠ Database Reset (Destructive Operations)

If you are running simulated paper trading and wish to wipe all ledger history to start fresh, you can reset the tables:

```
# Wipe all tables in the local development database
hoox db reset --confirm

# Wipe all tables in the live production D1 database (USE WITH EXTREME CAUTION)
hoox db reset --remote --confirm
```

Wiping your production D1 database is an irreversible operation. It will permanently delete all trade logs, position records, and asset histories. **Always** execute `hoox db export --remote` before running a reset!

Soft Reset Alternative

If you want to clear data but preserve schema structure:

```
# Truncate all data tables while keeping schema intact
hoox db query "DELETE FROM trades; DELETE FROM positions; DELETE FROM balances; DELETE FROM syste
```

D1 Free Tier Limits & Monitoring

Resource	Free Tier Limit	Monitoring Command
Rows Read/Day	5,000,000	`hoox db query "SELECT COUNT(*) FROM trades WHERE created_at >= date('now', '-1 day')"`
Rows Written/Day	100,000	`hoox db query "SELECT COUNT(*) FROM trades WHERE created_at >= date('now', '-1 day')"`
Storage	5 GB	`hoox db query "SELECT page_count * page_size FROM pragma_page_count(), pragma_page_size()"`
Max Connections	Concurrent	N/A — serverless, no connection pool

---

Next Steps

- **Local Development & Testing** — Run your local V8 wrangler isolates with local D1 SQLite bindings.
- **Infrastructure Management** — Manage KV config namespaces, Queue parameters, and R2 storage buckets.
- **Schema Reference** — Detailed DDL documentation and storage engineering details.

17 · GUIDES

SECRETS & NETWORK SECURITY

How to manage encrypted Cloudflare® Worker Secrets, secure Zero Trust service boundaries, and configure edge firewalls and IP allowlists.

Security is the most critical component of the Hoox trading platform. When deploying automated execution scripts, your capital and API credentials must be protected against malicious exploits, unauthorized webhook payloads, and network interceptions.

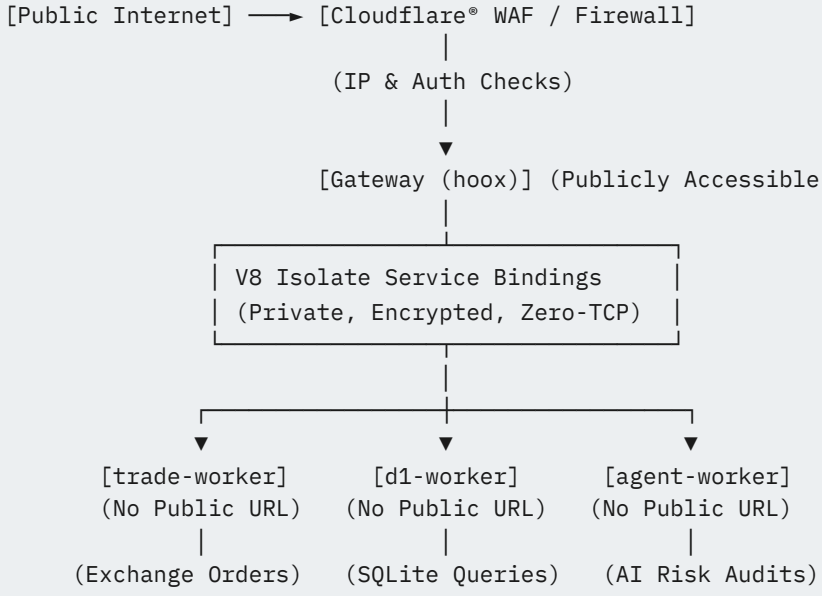
This guide outlines our **Zero Trust security architecture**, encrypted secret management procedures (including **sync modes**), and edge-level firewall protection runbooks. Prefer `hoox secrets` over manual `wrangler secret put` for day-to-day ops.

---

1. Zero Trust Network Isolation

Traditional trading systems expose database and exchange execution APIs to the public internet (secured by simple HTTP headers or ports). This creates an active attack surface.

Hoox implements a strict **Zero Trust microservice isolation topology**:



- **No Public Endpoints**: The `trade-worker`, `d1-worker`, and `agent-worker` literally **do not exist** on the public internet. They have no public IP addresses or URLs.
- **V8 Service Bindings**: Communication between the public gateway (`hoox`) and internal compute nodes is routed entirely inside Cloudflare®'s secure V8 engine isolates. Your trade routing data, database queries, and private logs never travel over the public internet, eliminating TLS decryption and packet-sniffing risks.

🔐 2. Encrypted Secret Management via CLI

API keys, exchange secrets, and Telegram bot tokens are never committed to git repositories or written in plain-text configuration files. Instead, they are stored directly on Cloudflare®'s hardware-secured key vaults. Local writers chmod secret files `0600`.

The Hoox CLI features deep encryption integrations to automate secret provisioning:

```
# 1. Inject a secure exchange credential (writes .dev.vars + puts to Cloudflare®)
hoox secrets set trade-worker BYBIT_SECRET_BINDING

# 2. Optional: dedicated testnet keys (preferred when webhook has "test": true)
hoox secrets set trade-worker BYBIT_TESTNET_KEY_BINDING
hoox secrets set trade-worker BYBIT_TESTNET_SECRET_BINDING
hoox secrets set trade-worker BINANCE_TESTNET_KEY_BINDING
hoox secrets set trade-worker BINANCE_TESTNET_SECRET_BINDING

# 3. List edge secret names for a worker (values are never printed)
hoox secrets list trade-worker

# 4. After key rotation: push mesh/system secrets only (INTERNAL_KEY_BINDING, WEBHOOK_*, …)
hoox secrets sync --system
# Alias: hoox secrets sync --required

# 5. Push all declared secrets from .dev.vars
# Partial results report synced / skipped (placeholders) / failed
hoox secrets sync
```

Secret sync modes

Mode	Flag	Behavior
Full	`hoox secrets sync`	Sync every declared secret from `.dev.vars` (skips empty/placeholder values)
System / required	`--system` or `--required`	Mesh keys only (`INTERNAL_KEY_BINDING`, webhook passkeys, ...) — safer after rotation
Skip during setup	`hoox setup --skip-secrets`	Generate keys / infra without pushing secrets (load later from `.keys/setup.env` with `--skip-keys` when appropriate)

Live vs testnet exchange secrets

One unified key pair is used for all venues. The exchange (Binance / Bybit / MEXC) is chosen by the signal payload and routing config — not by secret name.

Binding	Used when
`EXCHANGE_KEY_BINDING` / `EXCHANGE_SECRET_BINDING`	Live orders on the routed exchange
`EXCHANGE_TESTNET_KEY_BINDING` / `EXCHANGE_TESTNET_SECRET_BINDING`	Preferred for `"test": true` (Binance / Bybit)

Legacy per-venue bindings (`BINANCE_*`, `BYBIT_*`, `MEXC_*`) still resolve as fallbacks if the unified pair is unset. If a testnet pair is missing, `trade-worker` falls back to the live pair and logs a warning.

See Test Trading.

Listing Edge Secrets

Running `hoox secrets list` queries Cloudflare® for secret **names** bound on the worker, without ever exposing or decrypting the actual values in your terminal. Use `hoox check setup` for a broader secrets-presence audit during system validation — it stays **quiet** when local secrets are healthy and remote names are complete, and only fails when declared secrets are **missing** on Cloudflare® (with `hoox secrets sync` hints).

🛡️ 3. Webhook Firewall & TradingView® IP Allow-listing

To ensure that only TradingView®'s official servers can fire signals to your `/webhook` gateway:

1. **Passkey Verification**: The gateway checks the{" "} `apiKey` property inside the JSON payload against your secure manifest in `CONFIG\_KV`.
2. **Cloudflare® WAF (Web Application Firewall)**: Since TradingView® publishes their official IP ranges, you can configure Cloudflare®'s edge firewall to block all webhook traffic that does not originate from these verified IPs.

```
# Auto-configure WAF rules to lock the /webhook route to TradingView® IPs
hoox waf configure --TradingView-only
```

🔒 4. Security Best Practices Checklist

- **Least Privilege API Keys**: When creating API keys on Bybit, Binance, or MEXC, **never** enable "Withdrawal" permissions. Only check "Trade" and "Account Read" permissions.
- **Credential Rotation**: Automatically rotate your exchange API keys every 90 days. Deleting old keys and injecting new ones takes less than 60 seconds with `hoox secrets set`.
- **Zero-Commit Rule**: Verify that your `.env.local` and `.dev.vars` files are registered in your workspace's `.gitignore` file to prevent accidental pushes to public repos.
- **Emergency Response**: If you suspect a strategy error or exchange anomaly, immediately halt all execution via the CLI:

  ```
  hoox monitor kill-switch on
  ```

🔗 Next Steps

- **Platform Configuration Guide** — Map environment variables, secrets, and KV runtime settings.
- **Local Development & Testing** — Run local wrangler sandboxes with securely encrypted `.dev.vars`.

TEST TRADING (EXCHANGE TESTNET)

Run sandbox orders with test: true — per-exchange support, dedicated secrets, D1 isolation, dashboard, and agent safety.

Use **test trading** to place orders on exchange testnet/sandbox APIs without touching live capital. Enable it with a single boolean on any trade payload:

```
{
  "apiKey": "your-hoox-webhook-passkey",
  "exchange": "bybit",
  "action": "LONG",
  "symbol": "BTCUSDT",
  "quantity": 0.001,
  "leverage": 10,
  "test": true
}
```

Omit `test` or set `"test": false` for **live** trading (default).

Exchange support

Exchange	Supports `test: true`	REST host when enabled
Binance	Yes	`testnet.binancefuture.com`
Bybit	Yes	`api-testnet.bybit.com`
MEXC	No	Rejected (`TEST_TRADING_UNSUPPORTED`)

MEXC has no public REST sandbox for automated trading (UI demo only). Requests with `test: true` and `exchange: "mexc"` fail with a clear 400 error.

Credentials (recommended setup)

Prefer **dedicated testnet secrets** so live keys never touch sandbox hosts:

```
hoox secrets set trade-worker BINANCE_TESTNET_KEY_BINDING
hoox secrets set trade-worker BINANCE_TESTNET_SECRET_BINDING
hoox secrets set trade-worker BYBIT_TESTNET_KEY_BINDING
hoox secrets set trade-worker BYBIT_TESTNET_SECRET_BINDING
```

Mode	Secret bindings used
Live (`test` omitted/false)	`BINANCE_KEY_BINDING` / `BINANCE_SECRET_BINDING` , `BYBIT_*` , `MEXC_*`
Test (`test: true`)	`BINANCE_TESTNET_*` / `BYBIT_TESTNET_*` if both key+secret are set; otherwise **fall back** to live bindings (worker logs a warning)

Create API keys on the exchange testnet portal (not the production account) and inject them with the CLI above.

End-to-end behavior

When `test: true` is accepted:

1. **Gateway (`hoox`)** validates the payload, splits idempotency keys by live vs test mode, and forwards `test` on the service binding or queue message.
2. **trade-worker** builds a REST client against the testnet host (never the live WebSocket Durable Object).
3. **D1 ledger**
   - Trade status: `TEST_EXECUTED` (not `EXECUTED`)
   - Position id: `{exchange}-testnet-{symbol}-{side}` (e.g. `bybit-testnet-BTCUSDT-LONG`)
4. **Telegram** confirmations include `[TEST]` in the exchange label.
5. **Analytics** records the fill with exchange blob `bybit:test` (or `binance:test`) so test volume can be filtered.
6. **Dashboard stats** exclude test fills and testnet position ids from headline counts.
7. **Agent risk loop** ignores `*-testnet-*` positions (no live closes, no drawdown contamination).

Closing a testnet position

Send a close signal with the **same** `test: true` flag:

```
{
  "apiKey": "your-hoox-webhook-passkey",
  "exchange": "bybit",
  "action": "CLOSE_LONG",
  "symbol": "BTCUSDT",
  "quantity": 0.001,
  "test": true
}
```

In the **Dashboard → Positions** view:

- Default filter is **Live only**.
- Switch the mode filter to **Testnet** or **Live + Test** to see sandbox rows.
- Rows show a **TEST** badge next to the exchange.
- **Close** automatically sends `test: true` for testnet position ids.

Email and other ingress paths

Ingress	How to enable test mode
Public `/webhook`	`"test": true` in JSON body
Queue (`trade-execution`)	`"test": true` on the queue message (gateway sets this when queuing)
Email JSON signal	Include `"test": true` in the structured JSON body
Email plaintext	Not parsed as test (always live) — use JSON email bodies for sandbox

---

Safety notes

- Kill switch, max position size, and default leverage **still apply** to test trades.
- Do not rely on the agent to flatten testnet exposure; use an explicit close with `test: true`.
- Shared live-key fallback is convenient for local experiments but is an operator footgun in production — prefer dedicated `*_TESTNET_*` secrets.
- Testnet and live fills never share D1 position ids, so open exposure cannot clobber each other.

Quick curl examples

```
# Live
curl -X POST "https://hoox.example.workers.dev/webhook" \
  -H "Content-Type: application/json" \
  -d '{"apiKey":"…","exchange":"bybit","action":"LONG","symbol":"BTCUSDT","quantity":0.001}'

# Testnet
curl -X POST "https://hoox.example.workers.dev/webhook" \
  -H "Content-Type: application/json" \
  -d '{"apiKey":"…","exchange":"bybit","action":"LONG","symbol":"BTCUSDT","quantity":0.001,"test'
```

Related

- Signals & Trade Spec — full webhook field table
- Secrets & Network Security — secret injection
- API endpoints — request examples

INFRASTRUCTURE MANAGEMENT

How to provision and manage Cloudflare® D1 databases, KV namespaces, R2 buckets, Queues, and Vectorize indexes using hoox infra.

Hoox is fully serverless, using Cloudflare®'s distributed services as its compute and storage engine. Managing these resources—such as provisioning D1 databases, registering KV config buckets, setting up S3-compatible R2 storage, and mounting asynchronous Queues—is done directly via the **`hoox infra`** command group (prefer this over manual dashboard / Wrangler calls).

This guide is your operations manual for provisioning, inspecting, and managing Hoox edge infrastructure.

⚡ 1. One-Click Quick Provisioning

When setting up your trading workspace for the first time, you do not need to manually configure resources in Cloudflare®'s dashboard. The Hoox CLI automatically parses your enabled workers' configurations and spins up the entire infrastructure stack in one command:

```
# Auto-provision all required KV, D1, R2, Vectorize, and Queue resources
hoox infra provision
```

This command performs a dependency audit and calls Cloudflare®'s APIs to provision:

1. **D1 SQLite Database** (`hoox-db`).
2. **CONFIG_KV** Namespace bucket.
3. **`trade-execution`** {" "} messaging Queue.
4. **`trade-reports`** {" "} and{" "} **`system-logs`** {" "} R2 object storage buckets.
5. **`rag-index`** {" "} Vectorize vector search database.

🗄️ 2. D1 Database Administration

D1 databases store your critical transactional data (ledger, positions, balance history).

```
# A. List all active D1 databases in your Cloudflare® account
hoox infra d1 list

# B. Create a new custom database instance
hoox infra d1 create my-custom-db

# C. Delete a database instance (destructive)
hoox infra d1 delete my-custom-db
```

Once a D1 database is created, the CLI will output its unique `database_id` (a 36-character UUID). You must ensure this ID is bound in your worker's `wrangler.jsonc` file so the worker V8 isolate can establish a connection at runtime.

---

3. KV Configuration Namespace Management

KV stores are used to manage fast-path configurations, global parameters, and the kill switch.

```
# A. List all KV namespaces in your account
hoox infra kv list

# B. Create a new KV namespace (e.g. for staging or production)
hoox infra kv create CONFIG_KV

# C. Apply the default 16-key configuration manifest to your active namespace
hoox config kv apply-manifest
```

---

4. Asynchronous Queues Setup

Queues guarantee order delivery during periods of extreme exchange rate limits or network congestion.

```
# A. List all active Cloudflare® Queues
hoox infra queues list

# B. Create the trade execution queue
hoox infra queues create trade-execution
```

Queue Parameters & Throttling

During provisioning, Hoox configures the queue with:

- **Retry Backoff**: 30 seconds initial delay, scaling exponentially up to 15 minutes.
- **Message Retention**: 4 days (ensuring messages are preserved even during prolonged exchange maintenance events).

---

5. R2 Object Storage Administration

R2 is an S3-compatible, zero-egress fee object storage service used to offload heavy JSON payloads and PDF portfolio reports.

```
# A. List all R2 buckets
hoox infra r2 list

# B. Create the trade-reports bucket
hoox infra r2 create trade-reports
```

---

6. Vectorize RAG Index Management

Vectorize indexes house high-dimensional vector embeddings that power semantic search and Telegram AI bot memory.

```
# A. List all Vectorize indexes
hoox infra vectorize list

# B. Create a vector index with specific dimensions (e.g. 1536 for OpenAI embeddings)
hoox infra vectorize create rag-index --dimensions 1536 --metric cosine
```

Destroying resources via `hoox infra delete` is highly destructive and irreversible. Deleting a D1 database instantly wipes your trading records; deleting a KV bucket drops all configurations and triggers immediate gateway errors. Always backup ledgers before running deletions!

Next Steps

- **Database Migrations & SQL** — Run queries, manage drizzle schemas, and restore backups.
- **Take-to-Production Deployments** — Deploy your compiled workers and bind V8 isolates globally.

DEPLOY-WORKERS

DEPLOYING TO PRODUCTION

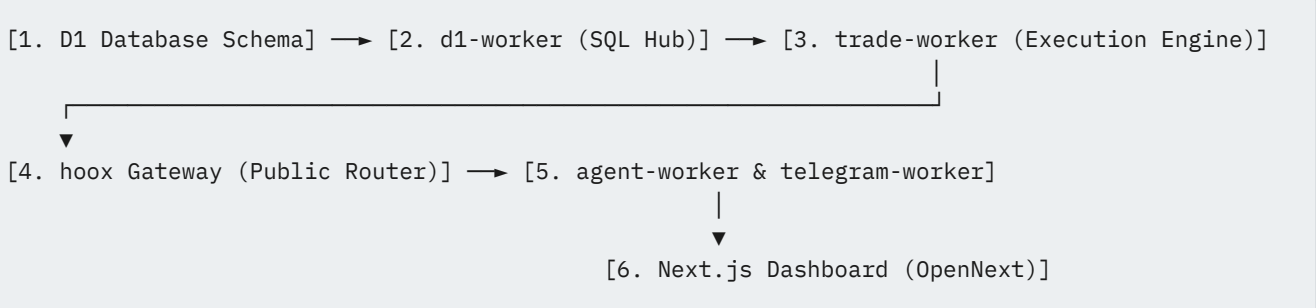
Deploying your algorithmic trading ecosystem to Cloudflare®'s production edge requires careful orchestration. Because workers communicate internally using fast-path **Service Bindings**, they have strict compile-time and deploy-time dependencies.

Prefer `hoox deploy all --auto` over manual Wrangler deploys. This guide outlines the automated deployment sequence (including **pyne-worker** when enabled), Next.js OpenNext compilation steps, post-deployment URL bindings, and troubleshooting runbooks.

The Strict Deployment Dependency Chain

When you deploy, workers must be uploaded in a specific sequence. If you attempt to deploy the public gateway (`hoox`) before the database or execution engines are live, the deployment will fail because the gateway's compile-time service bindings cannot resolve their targets.

The Hoox CLI automates this dependency hierarchy:



Deployment Order Explained

Step	Worker	Why First?
1	D1 Schema	Database tables must exist before workers try to read/write
2	d1-worker	SQL hub must be available for other workers' DB operations
3	trade-worker	Core execution engine — other workers depend on it
4	hoox Gateway	Public-facing router — needs all internal workers live
5	agent-worker, telegram-worker, pyne-worker	Background / tooling isolates — deploy after core path
6	Dashboard (OpenNext)	Frontend — depends on gateway and all APIs

---

⚡ Deployment CLI Commands

To execute a complete production rollout:

```
hoox deploy all --auto

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

# 3. Redeploy only the dashboard
hoox deploy dashboard
```

Deploy All Flags

Flag	Description	Example
`--auto`	Automatically resolves dependency order	`hoox deploy all --auto`
`--skip-tests`	Skip pre-deploy test suite	`hoox deploy all --auto --skip-tests`
`--dry-run`	Preview deployment plan without uploading	`hoox deploy all --auto --dry-run`
`--workers-only`	Skip dashboard deployment	`hoox deploy all --auto --workers-only`

Critical Post-Deployment Steps

Once `hoox deploy all` finishes uploading the workers, you must run three final scripts to link webhooks and sync environment databases:

```
# A. Register your private Telegram Bot webhook with Cloudflare
hoox deploy telegram-webhook

# B. Auto-update and bind internal Service URLs across all workers
hoox deploy update-internal-urls

# C. Sync and apply the default CONFIG_KV manifest variables
hoox deploy kv-config
```

Why These Steps Are Required

1. **Telegram Webhook Registration** — Tells Telegram's Bot API where to send user messages. Without this, your bot will be unreachable.
2. **Internal URL Binding** — After deployment, each worker gets a new URL. This command updates all Service Binding references so workers can find each other.
3. **KV Manifest Sync** — Ensures your runtime configuration (kill switch, rate limits, routing rules) matches the freshly deployed workers.

Next.js Dashboard Deployment (OpenNext)

The Hoox Command Center is a modern Next.js 16 web application that runs directly on Cloudflare® Workers using the **OpenNext adapter**.

When you run `hoox deploy dashboard`:

1. The CLI compiles the Next.js app using the **Turbopack** build engine.

2. The **OpenNext** compiler bundles the application logic into a single Edge-compliant V8 worker file: `.open-next/worker.js`.
3. All static files (images, JS chunks, CSS) are isolated under{" "} `.open-next/assets/`.
4. The CLI uses `wrangler` to deploy the worker and binds the static assets to Cloudflare®'s **ASSETS binding**, serving pages at sub-millisecond speeds.

```
# Build and deploy the Next.js Dashboard to Cloudflare® Workers
hoox deploy dashboard

# Rebuild dashboard only (after UI changes)
hoox deploy dashboard --force-rebuild
```

Dashboard Features

Once deployed, the Command Center provides:

- Real-time portfolio value and P&L tracking
- Win rate statistics and trade history charts
- Live position monitoring with one-click close
- Risk settings adjustment (no redeploy needed)
- AI-powered trade analysis chat

Routine Update & Upgrade Workflow

To pull the latest platform releases, run local tests, and update your active production edge:

```
# 1. Fetch latest changes and update git submodules recursively
git pull --recurse-submodules

# 2. Install updated dependencies
bun install

# 3. Execute the full local CI verification pipeline
hoox test

# 4. Run pre-deploy health check
hoox repair check

# 5. Roll out updates globally to the Cloudflare® Edge
hoox deploy all --auto
```

Rollback Procedure

If the new deployment causes issues:

```
# 1. Revert git to previous release
git checkout v2.0.0  # or your previous tag

# 2. Redeploy the stable version
hoox deploy all --auto
```

Post-Deployment Troubleshooting Matrix

Error Code / Symptom	Primary Root Cause	Guided Resolution
`502 Bad Gateway`	Service binding target is missing or has crashed.	Ensure the dependency worker (e.g., `trade-worker`) has been deployed successfully. Run `hoox deploy all` to rebuild all bonds.
`503 Service Unavailable`	The Global Kill Switch is active in KV.	Verify switch state: `hoox monitor kill-switch show`. If safe, restore trading: `hoox monitor kill-switch off`.
`401 Unauthorized`	Webhook request failed passkey check.	Audit KV authorization key: `hoox config kv get webhooks:api_key`. Verify the payload `apiKey` matches this value.
`409 Conflict`	Durable Object intercepted a duplicate trace ID.	Verify TV alert settings. Do not configure rapid-fire duplicate alerts within the same minute unless idempotency keys are unique.
Dashboard shows 500	Next.js build failed or D1 schema mismatch.	Re-run `hoox deploy dashboard` and check logs: `hoox logs worker dashboard`.
Telegram bot unresponsive	Webhook URL expired or token rotated.	Re-register: `hoox deploy telegram-webhook` and verify token: `hoox secrets list telegram-worker`.

By utilizing **Smart Placement** in your production builds, Cloudflare® will automatically route execution to the edge nodes located geographically closest to your exchanges (Frankfurt, Tokyo), ensuring high-speed fills!

Next Steps

- **Real-Time Observability & Monitoring** — Audit live fills, tail console logs, and run health diagnostics.
- **System Self-Healing & Repair** — Rebuild broken bindings and recover from deployment anomalies.
- **Infrastructure Management** — Provision and inspect KV, Queues, R2, and Vectorize indexes.

REPAIR

SELF-HEALING & REPAIR

Algorithmic trading environments must be resilient and self-healing. If you experience deployment failures, database routing discrepancies, expired authentication tokens, or missing git submodules, the Hoox CLI features a dedicated `hoox repair` command group designed to diagnose issues, check infrastructure status, and recover your workspace automatically.

---

1. Running the 5-Step System Diagnostics

If your trading gateway throws errors or the dashboard reports connection issues, run a comprehensive diagnostic sweep:

```
hoox repair check
```

The diagnostics engine runs a strict, sequential analysis of your entire monorepo environment. Each of the 5 steps completes and reports back:

- Submodule Integrity (verify all 9 workers are cloned)
- Dependency Resolution (verify bun workspaces)
- TypeScript Type Safety (run tsc --noEmit)
- Cloudflare® Edge Bindings (verify D1, KV, R2, Queues)
- Hardware Secrets Validation (verify API keys are bound)

---

2. Targeted Component Repairs

If a diagnostic check fails, you do not need to redeploy the entire stack. You can run highly targeted repairs:

```
hoox repair infra

# B. Re-audit and sync encrypted Workers Secrets to Cloudflare
hoox repair secrets

# C. Reset the CONFIG_KV configuration namespace to default variables
hoox repair kv

# D. Re-apply Drizzle DQL schemas and run pending D1 database migrations
hoox repair db

# E. Rebuild and redeploy a single, degraded worker
hoox repair worker trade-worker

# F. Full interactive workspace rebuild (last resort)
hoox repair rebuild
```

Repair Command Quick Reference

Command	Use Case	Safe?	Downtime
`hoox repair infra`	Missing D1/KV/Queue bindings	Yes	None
`hoox repair secrets`	Rotated API keys need re-binding	Yes	None
`hoox repair kv`	Configuration drift, need defaults	Caution	under 10s
`hoox repair db`	Schema mismatch or failed migrations	Caution	Seconds
`hoox repair worker NAME`	Single worker crash or build failure	Yes	Seconds
`hoox repair rebuild`	Catastrophic failure, full system reset	Destructive	Minutes

---

3. Full System Interactive Rebuild

In the event of a catastrophic system failure, execute a full interactive self-healing rebuild:

```
hoox repair rebuild
```

The Rebuild Sequence

When triggered, the CLI walks you through a structured recovery protocol:

- **Phase 1** — D1 Database Backup: Exports current ledger to `backups/recovery-pre-rebuild.sql`.
- **Phase 2** — Infrastructure Tear-Down: Deletes corrupted D1, KV, and Queue instances.
- **Phase 3** — Fresh Provisioning: Recreates all database tables and config buckets on Cloudflare®.
- **Phase 4** — Sequenced Redeployment: Re-compiles and deploys all 9 edge workers in correct dependency order.
- **Phase 5** — Database Seeding: Restores backup SQL data and re-applies schema migrations.
- **Phase 6** — KV Sync: Applies the 16-key configuration manifest defaults.

  The `hoox repair rebuild` command performs destructive resets on D1 and KV instances. Ensure you carefully read the interactive prompts and confirm database backup completion before letting the script proceed!

---

4. Common Troubleshooting Scenarios

Symptom	Cause	Action
`bun install` crashes	Git submodules are empty	`git submodule update --init --recursive` then `hoox repair check`
`D1_ERROR: no such table`	SQLite D1 tables not initialized	`hoox db apply --remote`
`500 Internal Server Error`	Missing or rotated exchange API secrets	`hoox secrets set trade-worker BYBIT_KEY_BINDING "key"`
`500 Internal Server Error` (2)	Or webhooks not registered	`hoox deploy update-internal-urls`
Telegram Bot is mute	BotFather token is wrong	`hoox secrets set telegram-worker TELEGRAM_BOT_TOKEN "token"`
Telegram Bot is mute (2)	Or webhook is unregistered	`hoox deploy telegram-webhook`
Queue depth growing	Exchange API returning errors	Check API key validity; `hoox monitor queue-depth`
Dashboard at 500 errors	Next.js build failed or D1 schema outdated	`hoox repair db` then `hoox dashboard deploy`
All trades returning 409	Idempotency keys colliding from rapid duplicate webhooks	Verify TradingView® alert frequency; wait 24h for DO TTL cleanup
Workers unreachable / unhealthy	One or more workers not responding	`hoox check health` (single source of truth)
Interrupted `setup` / `deploy` (Ctrl+C, network drop)	Workers in partially-deployed state	`hoox check health` → `hoox repair worker <name>` → `hoox deploy all --auto` (idempotent)
CLI crashes with "Bun is not defined"	Ran under Node instead of Bun	Re-run with `bunx hoox` or install Bun (`curl -fsSL https://bun.sh | bash`)
`check prerequisites` shows "token lacks required scopes"	API token missing D1/KV/R2/Queues:Edit	Recreate token with: Workers Scripts:Edit, D1:Edit, KV:Edit, R2:Edit, Queues:Edit, AI:Read

---

5. Emergency Decision Tree

```
Trading not working?

  Is the kill switch ON? -> hoox monitor kill-switch show
    YES -> hoox monitor kill-switch off
    NO  (continue)

  Are workers healthy? -> hoox check health
    Any FAILED? -> hoox repair check
    All OK  (continue)

  Are secrets bound? -> hoox secrets list <worker>
    Any missing? -> hoox secrets set <worker> NAME VALUE
    All present  (continue)

  Is queue depth > 0? -> hoox monitor queue-depth
    YES -> Exchange API issue; check exchange status page
    NO  (continue)

  Still stuck? -> Open GitHub Issue with hoox logs worker output
```

Pro Tips

- Scheduled Health Checks: Set up a cron job to run `hoox repair check` daily and email yourself the results.
- Pre-Deploy Checklist: Always run `hoox repair check` and `hoox test` before deploying to production.
- Secrets Rotation: Rotate exchange API keys quarterly. Use `hoox secrets set` to update — no redeployment needed.

Next Steps

- **Monitoring Operations Guide** — Audit system status, tail console logs, and verify fills.
- **CLI Reference Manual** — Review all CLI commands, options, and JSON flags.
- **Configuration Dictionary** — Map all 31 environment variables and 16 KV manifest entries.

MONITOR-TRADING

MONITORING OPERATIONS

Algorithmic trading demands high-integrity, real-time observability. Because Hoox microservices are distributed across Cloudflare®'s global edge network, tracking health, logs, and message queues requires a consolidated management plane.

This guide outlines how to use the Hoox CLI to audit system status, monitor queue backlogs, stream live edge logs, and manage the emergency kill switch.

---

1. Probing Worker Health Status

Every Hoox worker is configured with a secure `/health` endpoint that validates local CPU capacity, binding access, and D1 database connection integrity.

To probe and audit all active endpoints simultaneously:

```
hoox check health
```

> Use `hoox check health` as the single source of truth for worker health probes (not a monitor subcommand).

Expected Output

The CLI runs asynchronous concurrent health probes, displaying green indicators and public route locations:

```
┌──────────────────────────────────────────────────────────┐
│              Hoox Microservice Health Probes             │
├──────────────────────────────────────────────────────────┤
│  hoox (Gateway)     ... ✅ OK  https://hoox.alpha.workers.dev│
│  trade-worker       ... ✅ OK  (Internal Binding Verified)   │
│  d1-worker          ... ✅ OK  (Internal Binding Verified)   │
│  telegram-worker    ... ✅ OK  (Internal Binding Verified)   │
│  agent-worker       ... ✅ OK  (Internal Binding Verified)   │
│  web3-wallet-worker ... ✅ OK  (Internal Binding Verified)   │
│  email-worker       ... ✅ OK  (Internal Binding Verified)   │
│  report-worker      ... ✅ OK  (Internal Binding Verified)   │
│  analytics-worker   ... ✅ OK  (Internal Binding Verified)   │
│                                                          │
│  System Status: HEALTHY (0 warnings, 0 errors)           │
└──────────────────────────────────────────────────────────┘
```

Interpreting Health States

Indicator	Meaning	Action Required
✅ OK	All bindings active, D1 responsive	None — operating normally
⚠️ DEGRADED	Some bindings unavailable	Check `hoox logs worker <worker>` for errors
❌ FAILED	Worker unreachable or D1 disconnected	Run `hoox repair check` immediately
⏳ STARTING	Worker is still booting after deploy	Wait 30–60 seconds and retry

---

⚙️ 2. Governing the Global Kill Switch

The **Global Kill Switch** is your emergency brake. Stored inside the sub-millisecond `CONFIG_KV` namespace, flipping this parameter instantly blocks all incoming trade signals globally in under 10 seconds, without requiring code updates or worker redeployments.

```
hoox monitor kill-switch show

# B. Emergency HALT - disable all edge trade execution immediately
hoox monitor kill-switch on

# C. Resume normal operations
hoox monitor kill-switch off
```

Kill Switch Behavior

When activated, the following sequence occurs automatically:

1. `hoox` gateway rejects all incoming `/webhook` POST requests with `503 Service Unavailable`.
2. A `KILL\_SWITCH\_ACTIVE` event is logged to D1 for audit trail.
3. `agent-worker` halts its cron execution loop.
4. `trade-worker` queues any in-flight payloads for retry (not abandoned).
5. Telegram notification is sent to alert you.

When the Kill Switch is turned `ON`, the `hoox` gateway router immediately rejects all incoming TradingView® alerts and API webhook requests with a `503 Service Unavailable` error and logs a `KILL_SWITCH_ACTIVE` warning. Active positions must be flattened manually or via `hoox monitor trades close-all`.

📜 3. Auditing Recent Transactions

To review execution history directly from your terminal:

```
# Print the 10 most recent trades executed across all exchanges
hoox monitor trades

# Print the 50 most recent trades
hoox monitor trades 50

# Filter by specific exchange
hoox monitor trades --exchange bybit

# Filter by date range (ISO format)
hoox monitor trades --from 2026-05-15 --to 2026-05-20
```

This aggregates entries from your production D1 database and prints a formatted terminal table outlining: `TIMESTAMP | REQUEST_ID | EXCHANGE | SYMBOL | ACTION | QUANTITY | PRICE | STATUS`

Trade Status Codes

Status	Meaning
`EXECUTED`	Live order filled; written by trade-worker to D1
`TEST_EXECUTED`	Testnet (`"test": true`) fill; excluded from dashboard headline stats
`Filled`	Exchange-side full fill (report/CLI display)
`Partial`	Only portion filled; remainder queued
`Rejected`	Exchange returned a business logic error
`Failed`	Max queue retries exhausted; logged for review
`Enqueued`	Submitted to Cloudflare® Queues for async retry

Sandbox fills use position ids like `bybit-testnet-BTCUSDT-LONG`. Filter them in the Dashboard Positions **mode** control (Live only / Testnet / Live + Test). See Test Trading.

□ 4. Inspecting Queue Depth (Guaranteed Delivery Backlog)

If network volatility causes exchange API dropouts, Hoox offloads trade execution payloads to Cloudflare® Queues. To check if there is an active execution backlog:

```
hoox monitor queue-depth
```

This displays the number of pending messages in the queue:

- `0`: System is running normally (all trades executing on the fast-path direct service binding).
- `> 0`: Exchange API is experiencing rate-limits or downtime. Trade-worker is retrying transactions asynchronously in the background.

Queue Health Indicators

Queue Depth	Risk Level	Recommended Action
0	□ Normal	No action needed
1–5	□ Watchful	Monitor exchange health pages
6–20	□ Elevated	Check exchange maintenance schedule; consider `hoox config kv set webhooks:queue_mode direct_only`
20+	□ Critical	Verify exchange API keys are valid; check rate limit headers; scale manually if needed

---

□ 5. Real-Time Log Streaming & Telemetry

When debugging custom strategies, you can stream verbose console logs directly from Cloudflare®'s global edge nodes to your local terminal (`hoox logs worker`):

```
# A. Stream live console logs for a specific worker
hoox logs worker trade-worker

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

# C. Stream all workers simultaneously (diagnostic mode)
hoox logs all

# D. Filter logs by severity level
hoox logs worker trade-worker --level error
```

Downloading Logs for Off-Line Analysis

To download historical logs offloaded to your R2 storage bucket for compliance audits or tax reports:

```
hoox logs worker hoox --output logs/gateway-backup.log

# Download trade-worker logs for the last 24 hours
hoox logs worker trade-worker --output logs/trade-execution.log --since 24h
```

□ 6. System Metrics Overview

Hoox tracks key performance indicators via Cloudflare® Analytics Engine. Access summary metrics via:

```
# View current system metrics snapshot
hoox monitor metrics

# Or use the Dashboard Command Center for visual charts
hoox dashboard
```

Key Metrics Tracked

Metric	Source	Description
Orders/Min	analytics-worker	Trade execution throughput
Avg Latency (ms)	analytics-worker	Signal-to-fill round-trip time
Error Rate (%)	analytics-worker	Failed orders / total attempts
Queue Depth	Cloudflare® Queues	Pending execution backlog
Daily P&L	D1 Database	Realized profit/loss calculation
Drawdown %	agent-worker	Current daily asset drawdown

If the system status check returns a red error indicator (e.g. `d1-worker ... ✖ FAILED`), immediately proceed to the **Self-Healing & Repair Guide** to run automated diagnostics and restore service bindings!

Next Steps

- **Self-Healing & Repair** — Diagnose connection drops, recreate broken bindings, and rebuild environments.
- **Terminal UI Operations** — Monitor health status, tail logs, and edit configurations interactively in a full-screen GUI cockpit.
- **Infrastructure Management** — Manage KV config namespaces, Queue parameters, and R2 storage buckets.

PYNE LIVE TRADING

Bar-close cron: PYNE strategy events become HOOX WebhookPayloads on trade-worker. Alerts are a separate webhook path.

PYNE LIVE TRADING

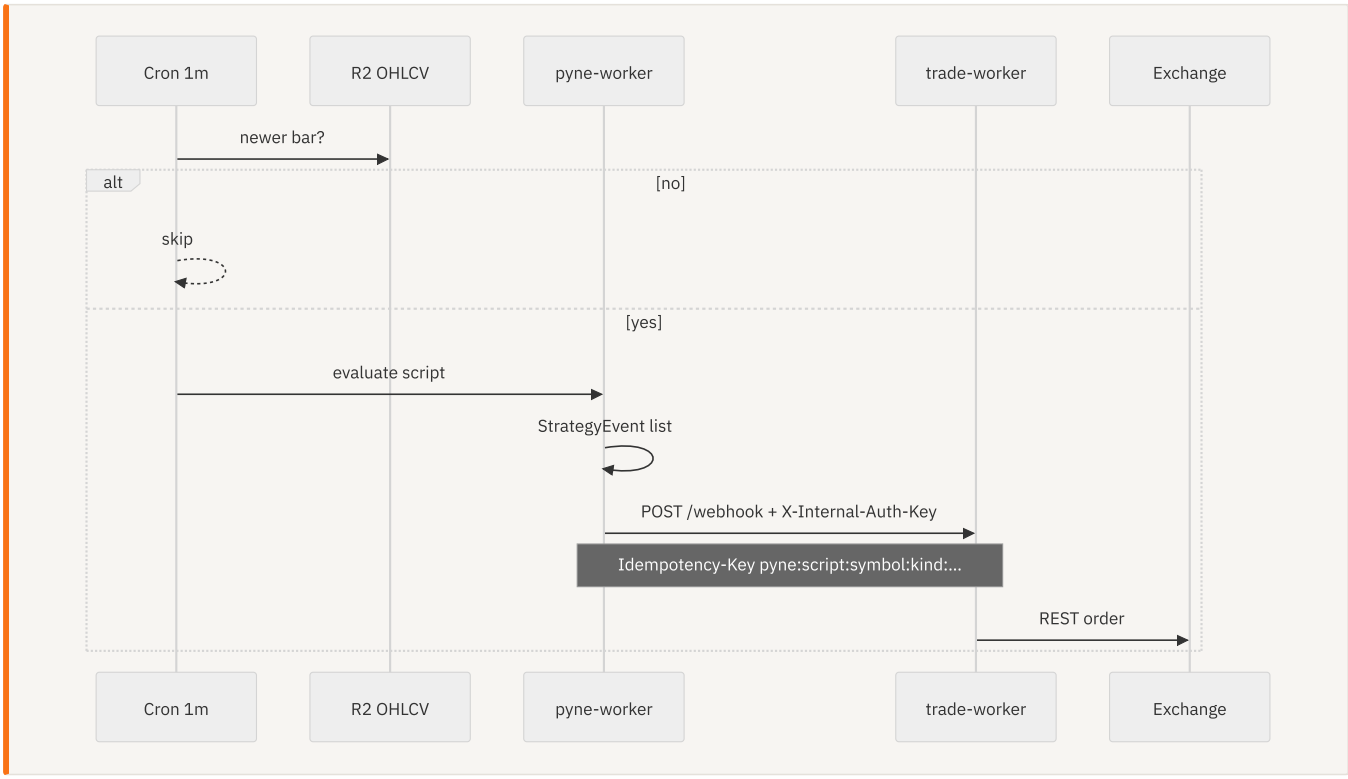
Deploy **pyne-worker**, register a strategy, give it R2 bars, and let the 1-minute cron re-run on new closes. `StrategyEvent` rows map to HOOX actions and hop to **trade-worker** over `TRADE_SERVICE`. This is the supported Pine → order path.

Test with `test: true` / testnet keys first. `close_all` becomes **two** payloads (`CLOSE_LONG` and `CLOSE_SHORT`).

Abstract

pyne-worker vendors `pynescript` (`hoox-pyne`) and speaks the same evaluate contract as Flask `POST /run` (`mode` interpret | compile | auto). It is a **tooling isolate**: public `/run` uses `X-API-Key` (`API_KEY`), not mesh `INTERNAL_KEY_BINDING`. The mesh key is sent **only** on the trade hop.

Conceptual model



Event map (`trade_forwarder.py`)

`StrategyEvent.kind`	HOOX `action`
`entry` + long	`LONG`
`entry` + short	`SHORT`
`close` / `exit` + long	`CLOSE_LONG`
`close` / `exit` + short	`CLOSE_SHORT`
`close_all`	`CLOSE_LONG` **and** `CLOSE_SHORT`
`order`	mapped from direction

Idempotency: `pyne:{script}:{symbol}:{kind}:{direction}:{bar}:{time}:{action}`.

Alert path (not orders)

`alert()` / `alertcondition()` → JSON `type: pine_alert`, `source: pyne-worker` to `ALERT_WEBHOOK_URL` or per-job `webhook_url`. HTTPS + public host only (SSRF fail-closed). Discord-style `content` is supported. **Does not** hit trade-worker unless you point the webhook at the gateway yourself.

Interface surface

Secrets

```
hoox secrets set pyne-worker API_KEY
hoox secrets set pyne-worker INTERNAL_KEY_BINDING   # live TRADE_SERVICE
hoox secrets set pyne-worker ALERT_WEBHOOK_URL      # optional
hoox secrets set pyne-worker DEFAULT_EXCHANGE       # optional
hoox secrets set dashboard PYNE_API_KEY             # match API_KEY
```

Deploy and health

```
hoox pyne deploy
hoox pyne health
# or: hoox deploy worker pyne-worker
```

Operator CLI

`hoox pyne {health,run,scripts,cron,feed,ingest,sync-vendor,deploy}`

Caps (isolate): 5 MB body, 100 KB script, 100 K bars, `/run` wall 30 s. Rate limit 100 req / 60 s per key.

Internals

Piece	Path / binding
Worker	`workers/pyne-worker` (submodule)
R2	`OHLCV_DATA`
Trade hop	`TRADE_SERVICE` → trade-worker `POST /webhook`
Cron	`* * * * *` — skip if R2 has no newer bar
Language	vendored `pynescript.runtime`

KV: pyne `dashboard.jsonc` is **secrets only** (`API_KEY` , `ALERT_WEBHOOK_URL`). Cron and `TRADE_SERVICE` are wrangler, not CONFIG_KV.

Invariants

1. Same-symbol `request.security` may work; **foreign / missing feed →** `na` . The isolate will not invent bars to force a trade.
2. Unset `API_KEY` is open **dev** mode — do not ship that.
3. Without `INTERNAL_KEY_BINDING` , live forwards are rejected by `requireInternalAuth` .
4. Paid Workers is recommended for a 1-minute cron; free-tier limits apply.

Worked examples

1. `hoox pyne deploy` and `hoox pyne health` .
2. `hoox pyne ingest` (or `/ingest`) a testnet symbol.
3. `hoox pyne scripts` — register a **strategy** with tiny size.
4. `hoox pyne cron` — enable the job.
5. Confirm trade-worker test fills, then remove `test: true` .

Language-level event fields: PYNE strategy events. Alert frequency: PYNE alerts.

Failure modes

Symptom	Cause	Fix
`/run` 401	Bad `X-API-Key`	Match `API_KEY`
Events in `/run` JSON, no order	Missing `TRADE_SERVICE` / internal key	Set `INTERNAL_KEY_BINDING`
Cron idle	R2 has no new bar	`hoox pyne feed` / ingest
Alert fired, no fill	Alert path ≠ trade path	Expected
AXIS showed the same script	AXIS does not execute	This page, not the PWA

See also

- pyne-worker isolate
- trade-worker

- Test trading
- Ecosystem
- AXIS and HOOX
- Evaluate scripts

AXIS AND HOOX

AXIS evaluates Pine on a chart. HOOX executes orders. Connecting them is optional and goes through pyne-worker, not the PWA.

AXIS AND HOOX

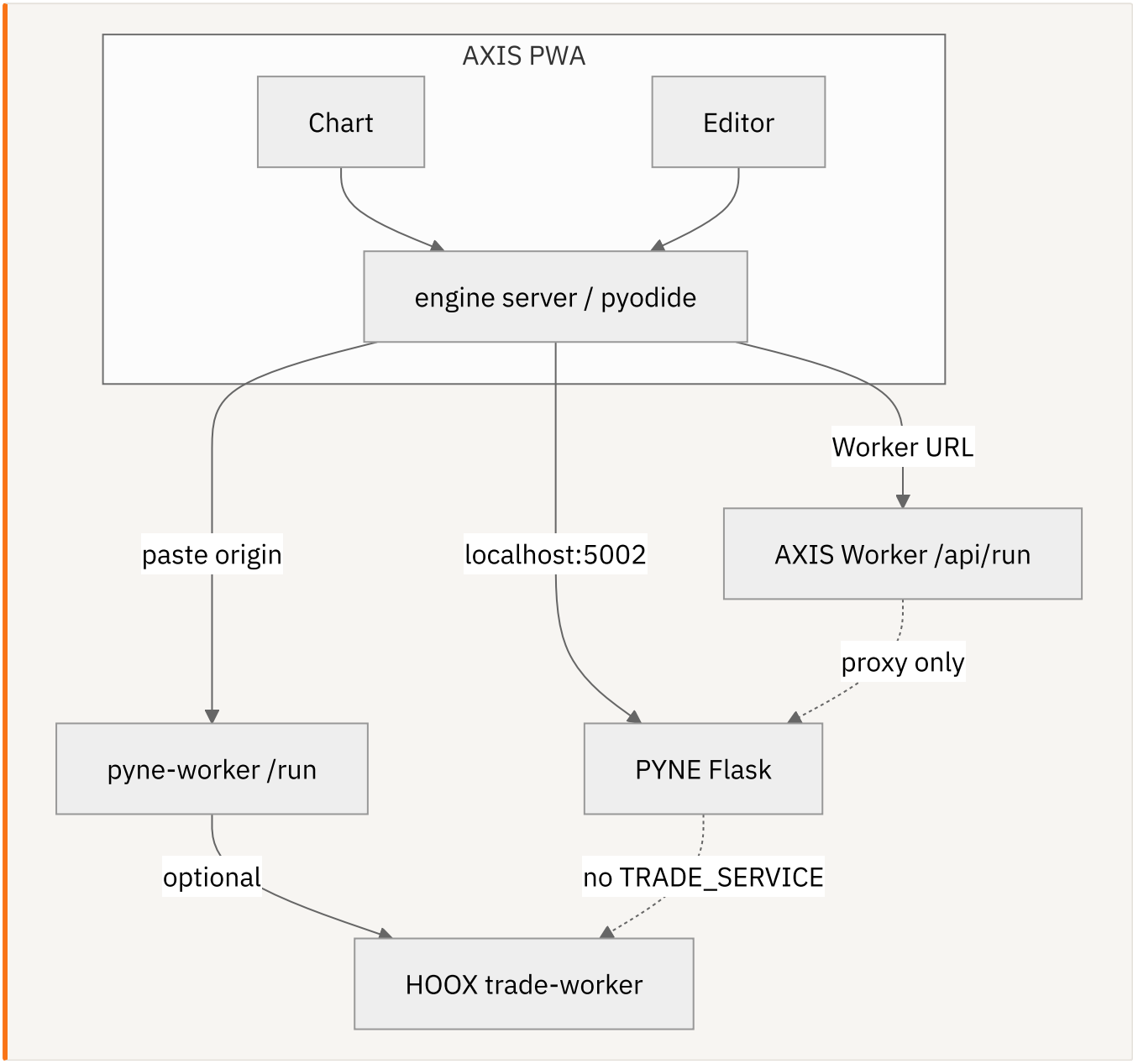
AXIS is the **chart**. HOOX is the **mesh**. They share a language (PYNE) and a website tab pattern. They do not share an order router.

If a strategy looks correct on axis.hoox.sh and nothing hits the exchange, that is working as designed.

Abstract

AXIS engines (`server` , `pyodide`) call PYNE. The HOOX **pyne-worker** isolate can be that `server` Backend URL. Only if **pyne-worker** has `TRADE_SERVICE` + `INTERNAL_KEY_BINDING` do strategy events become live orders. The PWA never holds exchange keys.

Conceptual model



Interface surface

AXIS action	HOOX effect
Engine `pyodide`	None
Engine `server` → Flask `:5002`	None (unless you built a custom forwarder)
Engine `server` → AXIS Worker	Proxy / 503 — still no trade-worker
Engine `server` → pyne-worker origin	Evaluate. Orders **only** if isolate forwarding is on
Local Alerts panel webhook	Your URL. Not certified as a trade path
PYNE Agent plugin	Writes scripts; optional validate via `/run`

AXIS Worker (`pynescript-axis` , `pynescript-axis.cryptolinx.workers.dev`) is **not** the HOOX `pyne-worker` isolate.

Full engine table: AXIS evaluation map.

Internals

Product	Default evaluate host
AXIS desk	Flask `http://localhost:5002`
AXIS offline	Pyodide + vendored `pynescript` wheel
HOOX live Pine	`workers/pyne-worker`

PyneTS (`@hoox-sh/pynets`) is a TypeScript **library**. AXIS does not load it. You can build your own host with it; that host is still not HOOX until you implement the trade hop.

Invariants

1. **AXIS ≠ execution.**
2. **AXIS ≠ engine** — evaluation is always a plugin.
3. Pasting a HOOX gateway webhook into AXIS alerts is **not** the supported live-trading path. Use PYNE live trading.
4. Runtimes Hub entries for "pyne-worker" vs "AXIS Worker" are different origins.

Worked examples

Research only

AXIS + Pyodide or local Flask. No HOOX secrets.

Research + same evaluate host as prod

Set AXIS Backend URL to the pyne-worker origin. Run with tiny size / `test` first. Confirm you understand that **the Worker**, not the PWA, is what may forward events.

Authoring

Install the PYNE Agent component plugin, generate a script, evaluate on Flask, then (separately) register that source on pyne-worker cron.

Failure modes

Symptom	Cause	Fix
Great backtest, flat account	No trade hop	Deploy pyne-worker forwarding
503 `NO_BACKEND` in AXIS	AXIS Worker has no `EXTERNAL_BACKEND`	Set it, or use Pyodide / Flask
Confused two Worker URLs	`pynescript-axis` vs `pyne-worker`	Check the hostname
Discord alert, no fill	Alert webhook ≠ `TRADE_SERVICE`	Expected

See also

- Ecosystem
- PYNE live trading
- AXIS docs
- AXIS evaluation
- PYNE

TRADINGVIEW® WEBHOOK INTEGRATION

How to configure TradingView® alerts, write copy-paste Pine Script™ v6 indicators, and link webhook signals to your edge gateway.

Connecting **TradingView® Alerts** directly to your Hoox edge gateway is the most common way to automate your trading strategies. By combining TradingView® charting with Hoox's low-latency edge execution, you can trigger orders instantly based on mathematical indicators, chart patterns, or custom Pine Script™ v6 logic.

This tutorial walks you through writing a Pine Script™ v6 script, setting up a TradingView® webhook alert, and testing executions.

Step 1: Writing a Pine Script™ v6 Signal Indicator

To trigger clean trade alerts, use TradingView®'s **Pine Script™** (`//@version=6`). Below is a copy-paste indicator script that evaluates a Moving Average Cross strategy and generates standardized trade alert signals:

```
//@version=6
indicator("Hoox EMA Cross Signals", overlay=true)

// 1. Inputs & Parameters
fastLength = input.int(9, title="Fast EMA Length")
slowLength = input.int(21, title="Slow EMA Length")

// 2. Indicators Calculation
fastEMA = ta.ema(close, fastLength)
slowEMA = ta.ema(close, slowLength)

plot(fastEMA, color=color.blue, title="Fast EMA")
plot(slowEMA, color=color.orange, title="Slow EMA")

// 3. Trade Conditions
longCondition = ta.crossover(fastEMA, slowEMA)
shortCondition = ta.crossunder(fastEMA, slowEMA)

plotshape(longCondition, title="Long Cross", style=shape.triangleup, location=location.belowbar,
plotshape(shortCondition, title="Short Cross", style=shape.triangledown, location=location.abovel

// 4. Alert Payloads Construction (JSON compliant)
// Live:
longAlertJson = '{"apiKey": "your-hoox-webhook-passkey", "exchange": "bybit", "action": "LONG", '
shortAlertJson = '{"apiKey": "your-hoox-webhook-passkey", "exchange": "bybit", "action": "SHORT",
// Testnet rehearsal (optional — add "test": true; prefer BYBIT_TESTNET_* secrets):
// longAlertJson = '{"apiKey": "your-hoox-webhook-passkey", "exchange": "bybit", "action": "LONG'

// 5. Trigger Alerts
if (longCondition)
    alert(longAlertJson, alert.freq_once_per_bar_close)

if (shortCondition)
    alert(shortAlertJson, alert.freq_once_per_bar_close)
```

Replace `"your-hoox-webhook-passkey"` in your Pine Script™ with the actual passkey configured in your `CONFIG_KV` store (`webhooks:api_key`). This is a critical security step—the gateway will reject any alerts with mismatched keys with a `401 Unauthorized` error. For exchange testnet rehearsal, add `"test": true` to the JSON and configure dedicated testnet secrets — see Test Trading.

Step 2: Creating the TradingView® Webhook Alert

Once your script is saved and added to your chart:

1. Click the **Alarm Clock (Alerts)** icon on the top right panel in TradingView®.
2. Select **Condition**: Choose your indicator `"Hoox EMA Cross Signals"` and select `"Any Alert Function Call"` (this directs TradingView® to parse the custom JSON payloads from the `alert()` functions inside your script).
3. Under **Expiration**: Select your alert lifespan (Premium accounts support Open-Ended alerts).
4. Navigate to the **Notifications** Tab:
   - Check the **Webhook URL** box.
   - Paste your public Hoox Gateway endpoint URL: `https://hoox.alpha-trading.workers.dev/webhook`
5. Navigate to the **Settings** Tab:
   - In the **Message** box, type: `{{strategy.order.alert_message}}` (if using a Backtesting Strategy) or leave it blank (if using an Indicator, as the JSON payload is already defined inside our `alert()` function).
6. Click **Create**.

Step 3: Verifying Execution in Production

When the Moving Average crossover occurs on your chart:

1. TradingView® compiles the JSON payload and POSTs it to your edge gateway.
2. The `hoox` gateway validates the signature and routes the order to Bybit.
3. Check the transaction directly in your terminal:

```
hoox monitor trades
```

4. Stream live gateway telemetry logs to verify execution latency:

```
hoox logs worker hoox
```

Webhook Troubleshooting Runbook

Symptom / Error	Primary Cause	Resolution
Alert fires but no trade executes	Mismatched API keys or WAF block.	Run `hoox logs worker hoox` to stream traffic. Check if the client IP was dropped by Cloudflare® WAF.
`401 Unauthorized`	Incorrect `apiKey` in Pine Script™.	Run `hoox config kv get webhooks:api_key` to check your key. Paste the exact string into your Pine Script™ alert payload.
`409 Conflict`	Double-alert fired within the same bar.	Adjust the alert frequency in Pine Script™: use `alert.freq_once_per_bar_close` instead of `alert.freq_all` to prevent rapid-fire retries.
`Invalid Symbol` API reject	Symbol format mismatch.	Hoox automatically converts variations (like `BTC-USDT` or `btc/usdt`) to `BTCUSDT`, but ensure your script sends standard uppercase symbols.

Next Steps

- **Setting Up Telegram Alerts** — Integrate Telegram notifications to get fills pushed directly to your phone.
- **API Endpoint Reference** — Review full request schemas and status codes.

TELEGRAM BOT SETUP

How to provision a secure Telegram Bot via BotFather, bind credentials, deploy webhooks, and execute commands.

The **`telegram-worker`** is a central operational pillar of the Hoox trading ecosystem. It serves two critical functions:

1. **Push Notifications**: Sends instant real-time order fill alerts, daily P&L audits, and system health status updates directly to your mobile phone.
2. **Interactive Command Console**: Provides a secure chat terminal allowing you to query open positions, check D1 balance logs, and trigger the Global Kill Switch directly via Telegram chat.

This tutorial guides you through creating a bot, binding secrets, deploying the secure Cloudflare® webhook, and using commands.

□ Step-by-Step Configuration Path

Step 1: Provision a Bot via BotFather

1. Open the Telegram app and search for the verified **`@BotFather`** account.
2. Click **Start** and send the command:

```
/newbot
```

3. Enter a friendly name for your bot (e.g. `My Hoox Edge Bot`).
4. Choose a unique username ending in "bot" (e.g. `AlphaHooxTradeBot`).
5. BotFather will output your **HTTP API Token** (save this securely): `5829104829:AAFkL92jL-492kLl...`

Step 2: Retrieve Your Private Chat ID

To ensure that only **you** can command your bot (and to prevent unauthorized users from viewing your trading balances), you must capture your private Telegram Chat ID:

1. Search for the username **`@userinfobot`** in Telegram and start a chat.
2. The bot will instantly return your numeric **Id** (e.g., `987654321`).

Step 3: Inject Credentials via Hoox CLI

Bind your bot token and authorized Chat ID as encrypted Workers Secrets in your workspace:

```
# Inject the secure Telegram Bot token
hoox secrets set TELEGRAM_BOT_TOKEN "5829104829:AAFkL92jL-492kLl..."

# Inject your authorized Telegram Chat ID
hoox secrets set TELEGRAM_CHAT_ID "987654321"
```

Step 4: Register the Webhook with Telegram

To allow Telegram's servers to route incoming chat commands straight to your edge worker, you must register your deployed gateway URL as the bot's official webhook handler:

```
# Automatically register the webhook endpoint with Telegram's APIs
hoox deploy telegram-webhook
```

The CLI calls Telegram's API to configure the route: `https://hoox.alpha-trading.workers.dev/telegram-webhook`

Interactive Chat Commands

Open your private bot chat in Telegram, click **Start**, and send the following commands to manage your edge:

Command	Action	Description
`/start`	Onboarding	Verifies connection and returns system welcome message.
`/status`	System Audit	Probes all workers and returns online/offline health status.
`/trades`	Transaction Log	Retrieves a formatted table of the 5 most recent executed fills.
`/positions`	Exposure Audit	Lists all currently active long/short positions with unrealized P&L.
`/kill_on`	**Emergency Halt**	Instantly activates the Global Kill Switch in KV, halting all trading.
`/kill_off`	Resume	Deactivates the Global Kill Switch, restoring signal processing.

Conversational AI Chat & Chart Audits

Beyond static commands, `telegram-worker` is integrated with **Cloudflare® Workers AI** and the **Multi-Provider AI Gateway**:

- **Natural Language Queries**: You can ask the bot conversational questions like *"Should I close my BTC position?"* or *"Analyze my trade win rate today"*. The bot queries your D1 SQL database, aggregates context using Vectorize, and responds with a LLaMA-3 analytical summary.
- **Chart Image Audits**: If you upload a screenshot of a trading chart or a portfolio page, the AI Gateway uses vision models (like Claude 3.5 Sonnet or Gemini 1.5 Pro) to analyze the chart and return an automated risk audit.

The `telegram-worker` strictly filters all incoming messages. If a message originates from a `Chat ID` that does not match your authorized `TELEGRAM_CHAT_ID` secret, it is silently dropped and logged as a security alert, ensuring your account remains 100% secure.

Next Steps

|- **Alert Troubleshooting** — Common TradingView® webhook errors and resolutions. |- **Platform Configuration Guide** — Comprehensive dictionary of all 31 env keys and 16 KV key-value items.

27 · TUTORIALS

EMAIL SIGNAL ROUTING

How to configure the email-worker parsing microservice, set up regex scanning patterns, and securely route SMTP/IMAP signals.

The **`email-worker`** is an ancillary input plugin that allows you to trigger automated trades via raw email alerts. While webhooks are the standard path, many legacy systems, charting platforms, or custom newsletters only distribute trade signals via email.

This tutorial walks you through setting up `email-worker` credentials, configuring regex subject scanning patterns, and securely routing messages.

---

1. Injecting Email Connection Credentials

To allow `email-worker` to log into your dedicated signals inbox, inject your SMTP/IMAP mailbox credentials as encrypted Workers Secrets:

```
# 1. Inject the IMAP/POP3 host address
hoox secrets set EMAIL_HOST "imap.securemail.com"

# 2. Inject the mailbox username/email address
hoox secrets set EMAIL_USER "signals@my-trading-app.com"

# 3. Inject the secure mailbox app password
hoox secrets set EMAIL_PASS "your_app_password_here"
```

Always use a **dedicated** email address solely for trade signals. Never use your personal inbox. Additionally, configure your email provider (e.g., Gmail, Fastmail) to require an **App Password** rather than your master account credentials to ensure tight access control.

---

2. Configuring Regex Parsing Rules in KV

Once `email-worker` establishes connection, it scans the inbox on a background schedule, evaluating incoming subjects and message bodies against **Regular Expression (regex)** patterns defined in `CONFIG_KV`:

```
# A. Define the prefix pattern required in the email subject line
hoox config kv set email:scan_subject "^TRADE SIGNAL:.*"

# B. Define the regex pattern used to extract the target asset token
hoox config kv set email:coin_pattern "(BTC|ETH|SOL|LINK|AVAX)"

# C. Define the regex pattern used to resolve trade action
hoox config kv set email:action_pattern "(LONG|SHORT|CLOSE|BUY|SELL)"
```

The Parsing Mechanism

When an email is scanned:

1. The worker matches the subject. If the subject is `"TRADE SIGNAL: Breakout Detected"`, the check passes.
2. The worker scans the email body using the regex patterns:
   - Body: `"...we are entering a LONG position on SOL/USDT..."`
   - Hitting `email:coin_pattern` resolves: **`SOL`**
   - Hitting `email:action_pattern` resolves: **`LONG`** (translated internally to `BUY`).
3. Automatically triggers an order via the `trade-worker` service binding.

JSON body with testnet mode

If the email body is structured JSON (instead of free-text), optional fields are forwarded to trade-worker:

```
{
  "exchange": "bybit",
  "action": "LONG",
  "symbol": "BTCUSDT",
  "quantity": 0.001,
  "test": true
}
```

`"test": true` requests exchange testnet execution (Binance/Bybit). Plaintext extraction never sets `test` — always live. See Test Trading.

🔒 3. Security Audits: SPF & DKIM Validation

To prevent malicious "spoofing" (e.g., an unauthorized sender trying to forge a trade signal to your inbox):

- **Sender Validation**: The `email-worker` parses the email's headers and matches the sender's address against a safe allowlist in KV: `email:authorized_senders = ["alerts@tradingview.com"]`.
- **DNS Verification**: The worker validates the **SPF (Sender Policy Framework)** and **DKIM (DomainKeys Identified Mail)** headers to verify that the message physically originated from the authorized domain and was not intercepted.

🧪 4. Testing Your Email Pipeline

To verify that the email signal ingestion loop works perfectly:

1. Compose a mock email from your authorized sender address.
2. Subject: `TRADE SIGNAL: Cross Over`
3. Body: `Action: SHORT, Symbol: ETHUSDT, Quantity: 0.05`
4. Send to your dedicated inbox.
5. In your terminal, tail the email worker's runtime logs to confirm the parse and order submission:

```
hoox monitor logs email-worker
```

Next Steps

|- **Alert Troubleshooting** — Common TradingView® webhook errors and resolutions. |- **Platform Configuration Guide** — Comprehensive dictionary of all 31 env keys and 16 KV key-value items. |- **API Endpoint Reference** — Review full REST payloads and HTTP error returns.

28 · REFERENCE

CLI COMMANDS REFERENCE

Comprehensive CLI directory detailing ~30 command groups, 60+ subcommands, global flags, and positional options for the hoox binary.

The `@hoox-sh/hoox-cli` tool (**0.13.x**) manages the entire monorepo development, provisioning, deployment, monitoring, and self-healing pipelines. This reference provides the complete command tree, positional arguments, optional flags, and concrete examples for all command groups. The binary is also available as **`hx`**.

Per-group stills: Interface gallery.

Install: `bun add -g @hoox-sh/hoox-cli`. Recommended bootstrap: `hoox onboard`. Prefer CLI commands over raw Wrangler for day-to-day ops.

Current defaults (0.13.x): ships with mesh security hardening (`@hoox-sh/hoox-shared@1.4.0` + worker submodules) — two-phase gateway idempotency, `RateLimiterStore`, chatId allowlists, named D1 list RPCs, mesh-wide `safeWaitUntil`. CLI surface still includes monorepo auto-detect + remember (run `hx` from any cwd), Linear Rail banner (`◆ H · O · O · X`), quieter `check setup` secrets UX, setup gates, secret sync modes (`--system` / `--required`), `hoox pyne` for **pyne-worker**, operator security plane (`doctor --security`, `tunnel check`). Global UX: `--no-color` / `NO_COLOR` / `TERM=dumb`, completion footers, “did you mean …”, sectioned help. Historical release notes: v0.9.0 · v0.8.0.

Workspace resolution (any cwd)

On every invocation the CLI resolves a monorepo root, remembers it, and `chdir`s into it so relative paths (`wrangler.jsonc`, `workers/`, …) work even when you launch `hx` from `~/Videos` or another project.

Order:

1. **`HOOX_REPO`** — explicit absolute path
2. **Walk up from cwd** — local checkout markers
3. **Remembered path** — `~/.hoox/config/monorepo.json`
4. **Global clone** — `~/.hoox/repo` (`hoox doctor --fix-runtime`)

Monorepo markers: `packages/cli/package.json` plus one of `wrangler.jsonc`, `wrangler.jsonc.example`, `workers/`, or `.gitmodules`.

```
# Discover once from the monorepo
cd ~/Git/hoox && hoox doctor

# Use from anywhere
cd ~/Videos && hx check health
```

Env / file	Role
`HOOX_REPO`	Force monorepo root
`HOOX_HOME`	Base for `~/.hoox` layout (config, data, global repo)
`HOOX_CLI_SILENT=1`	Suppress “using monorepo at …” stderr line
`~/.hoox/config/monorepo.json`	Persisted `{ "root", "updatedAt" }`

`hoox doctor` prints **Remembered**, **Runtime root**, and **Source** (`env` | `cwd` | `remembered` | `global` | `none`).

Global Flags

These options are registered globally and can be appended to any command:

Flag	Description	Example
`--json`	Outputs machine-parseable JSON format (ideal for script pipelines).	`hoox check health --json`
`--quiet`	Suppresses headers, banners, and interactive prompts.	`hoox deploy kv-config --quiet`
`--no-color`	Disable ANSI color output (also: `NO_COLOR=1` env var, `TERM=dumb`).	`hoox --no-color check health`
`--help`	Prints complete parameter and option listings.	`hoox infra d1 --help`
`--version`	Outputs active CLI package build version.	`hoox --version`
`-y, --yes`	Skip confirmation prompts (where supported).	`hoox repair rebuild --yes`

Command Groups Directory

~30 top-level command groups (including `doctor` , `tunnel` , `pyne` , and the rest below).

```
hoox                                Startup: resolve monorepo + chdir. No args: auto-onboard if
├── onboard                         One-shot full bootstrap (init + setup) [recommended]
├── init                            Interactive setup wizard (config only)
├── setup                           Auto-bootstrap infrastructure (keys, D1, secrets, dashboard)
├── clone [name]                    Clone worker repositories as submodules
├── dev                             Local development execution engine
│   ├── start                       Start all workers (native or Docker)
│   ├── worker <name>               Start a single worker
│   └── dashboard                   Start Next.js dashboard dev server
├── deploy                          Edge deployment pipelines
│   ├── all                         Roll out all workers + dashboard in sequence
│   ├── workers                     Deploy all workers only (skip dashboard)
│   ├── worker <name>               Deploy a single edge worker
│   ├── dashboard                   Deploy Next.js dashboard via OpenNext
│   ├── telegram-webhook            Register Telegram bot webhook API
│   ├── update-internal-urls        Bind inter-worker Service Binding URLs
│   ├── kv-config                   Synchronize KV manifest variables
│   ├── history <worker>            Show deployment history
│   └── rollback <worker>           Rollback to a previous version
├── infra                           Cloudflare resource provisioning IaC
│   ├── provision                   Auto-provision all required services
│   ├── d1 [list/create/delete]     Manage D1 databases
│   ├── kv [list/create/delete]     Manage KV namespaces
│   ├── r2 [list/create/delete]     Manage R2 object storage buckets
│   ├── queues [list/create/delete]  Manage Cloudflare Queues
│   ├── vectorize [create/delete]   Manage vector search indexes
│   └── analytics [list]            Manage Analytics Engine datasets
├── config                          Manage wrangler.jsonc configuration
│   ├── show                        View current configuration
│   ├── set <key> <value>           Update a config value
│   ├── env [init/show/validate/generate-dev-vars]  Manage .env.local build variables
│   ├── kv [set/get/list/delete/apply-manifest/manifest]  CONFIG_KV runtime keys
│   ├── transport [show|set]        Operator transport (public|access|mtls|tunnel)
│   ├── secrets [list/set/delete/sync]  Cloudflare Worker secrets (legacy location)
│   └── keys [generate/list]        Internal auth keys (legacy location)
├── secrets                         Manage Cloudflare Worker secrets [top-level]
│   ├── list [worker]               List secrets for a worker
│   ├── set <worker> <name>         Set a secret
│   ├── delete <worker> <name>      Delete a secret
│   └── sync [worker] [--system|--required]  Sync .dev.vars → CF (mesh-only with --system)
├── keys                            Manage internal auth keys [top-level]
│   ├── generate                    Generate new keys
│   └── list                        List existing keys
├── check                           Toolchain and route diagnostic suite
│   ├── prerequisites               Validate local machine tool installs
│   ├── setup                       Full system validation (config, infra, secrets, db)
│   ├── health [--fix]              Probe worker /health endpoints
│   ├── fix [--dry-run]             Auto-repair common issues
│   └── submodule-gitignore (sg)    Validate/fix worker submodule .gitignore
├── doctor                          Runtime paths, remembered monorepo, TUI entry, security hyg
├── tunnel                          Private ingress helpers (cloudflared / Access)
│   └── check                       Detect cloudflared; optional /v1/health probe
├── db                              SQLite D1 Database administration
│   ├── apply                       Apply DDL schemas to D1
│   ├── migrate                     Run schema migrations
│   ├── list                        Display active SQLite tables
│   ├── query <sql>                 Execute custom SQL reads
│   ├── export                      Dump database as a secure SQL file
```

```
│   └── reset                          Destructive truncate of all tables
├── monitor                            Observability and emergency controls
│   ├── trades [N]                     Aggregate recent filled transactions
│   ├── logs [worker]                  Tail and inspect time-series logs
│   ├── queue-depth                    Audit message counts in queues
│   ├── backup                         Backup SQL ledger to backups/
│   ├── kill-switch [show/on/off]      Emergency global trading halt
│   └── analytics                      Query analytics data from D1
├── workers                            Per-worker operations
│   ├── list                           List all workers with status/path/secrets count
│   ├── dev <name>                     Start a worker for local development
│   └── logs <name>                    Tail logs for a specific worker
├── repair                             System diagnostics & self-healing
│   ├── check                          Run 5-step checklist
│   ├── worker <name>                  Rebuild a degraded worker
│   ├── infra                          Verify and repair missing edge bindings
│   ├── secrets                        Re-sync hardware secrets
│   ├── kv                             Restore KV manifest defaults
│   ├── db                             Re-apply Drizzle migrations
│   └── rebuild                        Full destructive interactive rebuild
├── logs                               Real-time Cloudflare Worker log tailing
│   ├── worker <name>                  Tail logs for a specific worker
│   └── all                            Tail logs from all enabled workers
├── schema                             Validate/generate worker manifests
│   ├── validate [worker]              Validate worker(s) against canonical manifest
│   ├── list                           List workers with binding counts
│   └── generate <worker>              Generate wrangler.jsonc + .dev.vars from manifest
├── update                             Self-update the CLI or wrangler
├── test                               Run verification pipeline (CI)
├── waf                                Auto-configure Cloudflare WAF firewall rules
├── dashboard                          Unified dashboard operations
│   ├── dev                            Start the dashboard dev server
│   └── deploy [--rebuild]             Build and deploy the dashboard
├── tui                                Launch the OpenTUI terminal dashboard
├── agent                              AI agent operations and health checks
│   └── health [--probe]               Provider config check; optional live probes
├── pyne                               PYNE edge evaluate (pyne-worker / Pine Script™)
│   ├── health                         Probe pyne-worker GET /health
│   ├── run <script>                   POST /run with a .pine file
│   ├── scripts [list|get|deploy|delete]  Managed scripts
│   ├── cron [jobs|run]                Bar-close jobs list / manual trigger
│   ├── feed refresh                   Pull latest klines into R2
│   ├── ingest                         Fetch OHLCV into storage
│   ├── sync-vendor                    Sync pynescript vendor modules
│   └── deploy                         sync-vendor + wrangler deploy
├── trace                              Cloudflare Workers Observability (beta)
│   ├── events                         Query trace events (spans) with filters
│   ├── metrics                        Aggregate metrics (count, p99, …)
│   ├── live                           Live tail traces
│   ├── keys / values <key>            Filter key discovery
│   ├── destinations [add|remove]      OTLP export destinations
│   └── usage                          Trace event counts
├── perf                               Performance measurement tools
│   └── fastpath                       Probe-based latency measurement
│       ├── run [options]              Send N probes, report p50/p95/p99
│       ├── tail [options]             Continuous probing for a duration
│       └── report [options]           Query past probe events
```

```
├── disclaimer                          Display legal disclaimer
└── completion [shell]                  Generate shell completion script (bash, zsh, fish)
```

Key Commands Detail & Examples

A. Onboarding (Start Here)

```
# Recommended: one-shot full bootstrap (collects credentials, provisions, deploys)
hoox onboard

# Non-interactive one-shot
hoox onboard --token cfut_xxx --account xxx --preset full

# Step-by-step (more control)
hoox init        # Write wrangler.jsonc, collect integration secrets
hoox setup       # Generate keys, apply D1 schema, push secrets, deploy dashboard

# Verify the system
hoox check setup
hoox check prerequisites
```

B. Local Development

```
# Start all workers in a Docker container stack
hoox dev start --runtime docker

# Start all workers natively via wrangler dev
hoox dev start --runtime native

# Start a single worker
hoox dev worker trade-worker

# Start the Next.js dashboard dev server
hoox dev dashboard
# OR (equivalent):
hoox dashboard dev
```

C. Edge Provisioning & Deployment

```
# Provision all databases, buckets, and queues on your Cloudflare® account
hoox infra provision

# Deploy the entire microservices stack in sequence, automatically updating URLs
hoox deploy all --auto

# Deploy a single worker
hoox deploy worker trade-worker

# Build and deploy the Next.js dashboard
hoox dashboard deploy --rebuild
# OR (equivalent):
hoox deploy dashboard --rebuild
```

D. Health Checks & Monitoring

```
# Check all worker health endpoints (the single source of truth)
hoox check health

# Try to auto-fix any issues found
hoox check health --fix

# Audit active trade history table
hoox monitor trades 25

# Emergency Halt: halt all trade signals globally in under 10 seconds
hoox monitor kill-switch on
```

E. Secrets & Keys Management

```
# Set a Cloudflare® Worker secret (live exchange keys)
hoox secrets set trade-worker BINANCE_KEY_BINDING "your_key_here"
hoox secrets set trade-worker BINANCE_SECRET_BINDING "your_secret_here"

# Optional: dedicated testnet keys for "test": true webhooks
hoox secrets set trade-worker BINANCE_TESTNET_KEY_BINDING "your_testnet_key"
hoox secrets set trade-worker BINANCE_TESTNET_SECRET_BINDING "your_testnet_secret"
hoox secrets set trade-worker BYBIT_TESTNET_KEY_BINDING "your_testnet_key"
hoox secrets set trade-worker BYBIT_TESTNET_SECRET_BINDING "your_testnet_secret"

# List secrets for a worker
hoox secrets list trade-worker

# Sync mesh/system secrets only (INTERNAL_KEY_BINDING, WEBHOOK_*, …)
hoox secrets sync --system
# Alias: hoox secrets sync --required

# Sync all declared secrets from .dev.vars
# Reports partial results: synced / skipped (placeholders) / failed
hoox secrets sync

# Generate internal auth keys
hoox keys generate

# List existing keys
hoox keys list
```

F. Database Administration

```
# Query the total sum of fees paid today across Bybit
hoox db query "SELECT SUM(fee) FROM trades WHERE created_at >= date('now')" --remote
```

G. Performance Measurement

```
# Send 50 synthetic probes and report p50/p95/p99 per-hop latency
hoox perf fastpath run --n 50

# Continuous probing for 60 seconds
hoox perf fastpath tail --duration 60

# Query past probe events
hoox perf fastpath report --from 1h
```

H. Self-Healing & Diagnostics

```
# Run a full workspace system audit
hoox repair check

# Verify worker health (the single source of truth)
hoox check health
```

Historical: v0.9.0 Output Polish

> **Historical notes** from the v0.9.0 release (retained for upgrade context). Current CLI is **0.13.x** — Linear Rail is the default banner style; monorepo remember and quieter secrets checks landed later; mesh security hardening ships with 0.13.

The v0.9.0 release polished the output framework. **No new commands and no breaking changes** — every command picks up the improvements automatically through the shared formatters. The new primitives are also available to plugin authors via the package's public exports.

New output primitives

- `--no-color` **global flag** — suppresses all ANSI color output. Also honored: `NO_COLOR=1` env var (https://no-color.org standard) and `TERM=dumb`.
- `formatCompletion(message, { durationMs, suggestion })` — prints a `✓ Done in 1.2s` footer after every successful command, with an optional `→ next: hoox …` suggestion. Wired into the global `program.hook("postAction", ...)`.
- **"Did you mean" suggestions** — typos like `hoox deplpy` produce `did you mean 'hoox deploy' ?`. Levenshtein distance ≤ 2 against all registered command names.
- **Custom help formatter** — `hoox --help` and per-command `--help` use a sectioned layout (Usage / Options / Examples / See also) with refined colors.
- `formatNumber(n)` — compact notation (`1.2K`, `1.5M`, `2.5B`) used by `formatTable` number auto-alignment and by `hoox perf fastpath` / `hoox monitor trades` / `hoox trace metrics`.
- `formatBytes(n, { binary? })` — SI (`KB`, `MB`) or binary (`KiB`, `MiB`) units.

Visual changes (visible in every command)

- **Theme palette** — refined to a modern-minimal aesthetic: zinc/slate text, single indigo-400 accent, de-saturated status colors (emerald/rose/amber/sky). The visual change ripples through every command; information content is unchanged.
- **Badge style** — `formatBadge()` no longer uses high-contrast background chips; it now renders colored glyph + colored text (Vercel / Linear style).
- **Spinner** — uses braille dots (`⠋⠙⠹⠸⠼⠴⠦⠧⠇⠏`) instead of plain ASCII (`-\|/`).
- `formatTable` **options** — supports `zebra`, `alignNumbers`, `colorizeStatus`, `compact` (all default to on, except `compact` which defaults to off). Backward compatible.
- `formatError` — card layout with `[code]` badge and `suggestions` support. JSON output now includes a new `suggestions` field.
- **Banner default** — later superseded by **Linear Rail** (`◆ H · O · O · X`) as the default compact banner in 0.11.x; version is read dynamically from `package.json`.

See `packages/cli/CHANGELOG.md` for the full release notes.

Historical: v0.8.0 Refactor

> **Historical notes** from the v0.8.0 release. Commands listed here remain current; this section documents when they were introduced.

The v0.8.0 release consolidated and cleaned up the CLI surface. **Pre-1.0 breaking changes were made without deprecation warnings** since this is still active development.

New commands (introduced in v0.8.0; still current)

- **`hoox onboard`** (aliases: `bootstrap`, `quickstart`) — One-shot full bootstrap that chains `init` + `setup`. The recommended entry point for new users. When you run `hoox` with no arguments and no `wrangler.jsonc` exists, the CLI now auto-launches this.
- **`hoox secrets`** (top-level) — Promoted from `hoox config secrets` for discoverability.
- **`hoox keys`** (top-level) — Promoted from `hoox config keys` for discoverability.
- **`hoox perf fastpath`** — Probe-based latency measurement (active synthetic probes with p50/p95/p99 per-hop breakdown).
- **`hoox dashboard dev`** / **`hoox dashboard deploy`** — Unified dashboard operations that were previously split across `hoox dev dashboard` and `hoox deploy dashboard`.

Removed commands (use the replacement)

- `hoox monitor status` → use `hoox check health` (single source of truth for health checks)
- `hoox workers status` → use `hoox check health`
- `hoox dashboard update-urls` → use `hoox deploy update-internal-urls`
- `hoox config secrets` → use `hoox secrets` (top-level now)
- `hoox config keys` → use `hoox keys` (top-level now)

Every single subcommand is fully documented locally! Append `--help` to any command (e.g. `hoox config env --help`) to view advanced positional arguments and specific flag options instantly.

Next Steps

- **Configuration Dictionary** — Comprehensive dictionary of all 31 env keys and 16 KV key-value items.
- **API Endpoint Reference** — Analyze REST routes, schemas, and gateway error structures.

29 · REFERENCE

API ENDPOINT SPECIFICATION

Complete REST API reference for the Hoox gateway, webhooks, health checks, AI chat streams, and edge error models.

This document details the public REST API endpoints exposed by the Hoox gateway (`workers/hoox-worker`). These endpoints are used to ingest automated trade signals, register Telegram bots, query AI metrics, and check serverless V8 isolate health status.

Request Headers & Security Policy

Every public HTTP request submitted to the Hoox gateway must include the following headers:

```
Content-Type: application/json
Accept: application/json
```

CORS & Origin Policies

For security, **Cross-Origin Resource Sharing (CORS)** is disabled by default on the `/webhook` route—it only accepts direct server-to-server POST requests (e.g. from TradingView® or custom server scripts).

To interact with the dashboard, the gateway verifies active authorization cookies and tokens inside the V8 engine context.

---

1. Ingest Trade Signal Webhook

Receives, validates, and routes trade orders to exchange APIs.

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

Request Payload (JSON Schema)

```
{
  "apiKey": "alpha_secure_key_18305",
  "exchange": "bybit",
  "action": "LONG",
  "symbol": "BTCUSDT",
  "quantity": 0.005,
  "leverage": 10,
  "test": true,
  "idempotencyKey": "uuid-9b1deb4d-3b7d"
}
```

Optional field **`test`** (`boolean`): when `true` , the trade-worker routes the order to the exchange testnet (Binance Futures / Bybit). MEXC does not support API test trading and returns `TEST_TRADING_UNSUPPORTED` .

	Live	Test (`"test": true`)
Host	Production REST / optional WS DO	Testnet REST only
Secrets	`BINANCE_*` / `BYBIT_*` / `MEXC_*`	Prefer `*_TESTNET_*`, else live fallback
D1	`EXECUTED` · normal position id	`TEST_EXECUTED` · `*-testnet-*` id

Full setup: Test Trading.

Response Models

A. Success Path (Order Filled Instantly - 200 OK)

```
{
  "success": true,
  "requestId": "9b1deb4d-3b7d-4bad-9bdd-2b0d7b3dcb6d",
  "exchange": "bybit",
  "symbol": "BTCUSDT",
  "action": "LONG",
  "result": {
    "orderId": "18049284739",
    "status": "Filled",
    "executedQty": 0.005,
    "price": 68425.5,
    "timestamp": 1779261050000
  }
}
```

B. Queue Failover Path (Exchange Offline / Rate-limited - 202 Accepted)

If the exchange is down, the signal is enqueued for guaranteed asynchronous delivery.

```
{
  "success": true,
  "requestId": "9b1deb4d-3b7d-4bad-9bdd-2b0d7b3dcb6d",
  "status": "Enqueued",
  "message": "Signal enqueued in trade-execution Queue for background execution retry."
}
```

---

2. System Health Check

Probes the gateway isolate, validating connections to D1 databases and CONFIG_KV caches.

- **Endpoint**: `/health`
- **Method**: `GET`

Success Response (200 OK)

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

🤖 3. Conversational AI Chat & Telemetry

Integrates with the `agent-worker` Multi-Provider AI Gateway.

A. Conversational Chat Stream

- **Endpoint**: `/agent/chat`
- **Method**: `POST`
- **Headers**: `Accept: text/event-stream` (Supports Server-Sent Events)

Request Payload

```
{
  "prompt": "Evaluate my trade performance over the last 24 hours.",
  "stream": true,
  "provider": "anthropic"
}
```

B. AI Gateway Telemetry

- **Endpoint**: `/agent/usage`
- **Method**: `GET`

Response Payload (200 OK)

```
{
  "success": true,
  "timestamp": 1779261050000,
  "usage": {
    "openai": { "inputTokens": 45180, "outputTokens": 8920, "costUSD": 0.183 },
    "anthropic": {
      "inputTokens": 18240,
      "outputTokens": 4010,
      "costUSD": 0.081
    },
    "workersAi": { "neuronsUsed": 842000, "costUSD": 0.0 }
  }
}
```

4. Standard Platform Error Models

When an API check fails, the gateway uses the shared `@hoox-sh/hoox-shared` package to return a standardized JSON error format:

A. 400 Bad Request (Payload Validation Failure)

```
{
  "success": false,
  "error": {
    "code": "VALIDATION_ERROR",
    "message": "Invalid JSON payload structure.",
    "details": [
      { "field": "quantity", "issue": "Must be greater than zero." },
      {
        "field": "exchange",
        "issue": "Invalid exchange router. Options: binance, bybit, mexc."
      }
    ]
  }
}
```

B. 401 Unauthorized (Invalid API Key / IP Address)

```
{
  "success": false,
  "error": {
    "code": "UNAUTHORIZED",
    "message": "Authentication failed. Mismatched apiKey or unauthorized origin IP."
  }
}
```

C. 409 Conflict (Duplicate Request Intercepted)

```
{
  "success": false,
  "error": {
    "code": "DUPLICATE_REQUEST",
    "message": "Order rejected. Durable Object locked: trace ID already processed in the last 24
  }
}
```

D. 503 Service Unavailable (Kill Switch Engaged)


```
{
  "success": false,
  "error": {
    "code": "KILL_SWITCH_ACTIVE",
    "message": "Trading halted globally. The emergency Global Kill Switch is currently turned ON
  }
}
```

Every error code is designed to map directly to a readable prompt in the Telegram bot and Next.js dashboard UI, allowing you to instantly diagnose execution problems.

Next Steps

- **Request Payload Schemas** — Analyze the complete JSON request schemas and TypeScript interfaces.
- **CLI Commands Reference** — Review all CLI commands, options, and JSON flags.

30 · REFERENCE

CONFIGURATION DICTIONARY

Exhaustive matrix of all 31 environment variable keys and 16 Cloudflare® KV dynamic manifest parameters.

This document serves as the absolute, exhaustive reference dictionary for every environment variable, encrypted Workers Secret, and dynamic KV runtime setting utilized in the Hoox trading ecosystem.

1. Environment Variables & Secrets Matrix (31 Keys)

These parameters are configured in your local `.env.local` file for building/deploying or injected as encrypted **Workers Secrets** for runtime isolate compute.

A. Core Platform & Infrastructure

Variable	Required	Type	Default	Description
`CLOUDFLARE_API_TOKEN`	**Yes**	`string`	—	Cloudflare® API Token with Workers, D1, and KV read/write permissions.
`CLOUDFLARE_ACCOUNT_ID`	**Yes**	`string`	—	Your 32-character Cloudflare® Dashboard Account hash.
`SUBDOMAIN_PREFIX`	**Yes**	`string`	—	Subdomain prefix under which public worker gateways deploy.
`NODE_ENV`	No	`string`	`production`	Environment profile: `development`, `staging`, or `production`.
`HOOX_API_URL`	No	`string`	—	Local API target URL (automatically configured during dev runs).

B. Exchange API Credentials (Encrypted Secrets)

Variable	Required	Type	Scope	Description
`BYBIT_API_KEY`	No	`string`	`trade-worker`	Bybit order placement account credential.
`BYBIT_API_SECRET`	No	`string`	`trade-worker`	Bybit HMAC-SHA256 private order signature.
`BINANCE_API_KEY`	No	`string`	`trade-worker`	Binance trade permission credential.
`BINANCE_API_SECRET`	No	`string`	`trade-worker`	Binance private HMAC order signature.
`MEXC_API_KEY`	No	`string`	`trade-worker`	MEXC trade permission credential.
`MEXC_API_SECRET`	No	`string`	`trade-worker`	MEXC private HMAC order signature.

C. Telegram Bot Alerts & Telemetry

Variable	Required	Type	Scope	Description
`TELEGRAM_BOT_TOKEN`	No	`string`	`telegram-worker`	Telegram HTTP API bot auth token from `@BotFather`.
`TELEGRAM_CHAT_ID`	No	`string`	`telegram-worker`	Authorized numeric Chat ID for pushed fills and commands.

D. Multi-Provider AI Credentials

Variable	Required	Type	Scope	Description
`OPENAI_API_KEY`	No	`string`	`agent-worker`	OpenAI API access key.
`ANTHROPIC_API_KEY`	No	`string`	`agent-worker`	Anthropic Claude API access key.
`GOOGLE_AI_API_KEY`	No	`string`	`agent-worker`	Google Gemini API access key.

E. DeFi & Web3 Wallet Settings

Variable	Required	Type	Scope	Description
`ETH_MNEMONIC`	No	`string`	`web3-wallet`	Secure 12 or 24-word seed phrase for on-chain contract swaps.
`RPC_PROVIDER_URL`	No	`string`	`web3-wallet`	HTTP Ethereum / EVM JSON-RPC provider (e.g. Infura/Alchemy).

F. Email Parsing Inbox Connection

Variable	Required	Type	Scope	Description
`EMAIL_HOST`	No	`string`	`email-worker`	POP3/IMAP mailbox server address.
`EMAIL_USER`	No	`string`	`email-worker`	Target signal mailbox email address.
`EMAIL_PASS`	No	`string`	`email-worker`	Secure app password for signal mailbox access.

---

2. Dynamic KV Manifest Settings (16 Keys)

These runtime variables exist inside the `CONFIG_KV` namespace. They are accessed globally at sub-millisecond speeds and take effect instantly upon being modified.

KV Key	Type	Default	Operational Impact
`trade:kill_switch`	`boolean`	`false`	**Global emergency halt**. If `true`, all executions halt.
`trade:max_daily_drawdown_percent`	`number`	`5.0`	Maximum daily loss before the AI Risk Manager triggers the kill switch.
`webhooks:api_key`	`string`	`your_key`	Public webhook passkey checked during signal ingestion.
`webhooks:queue_mode`	`string`	`queue_failover`	Routing behavior: `direct_only`, `queue_failover`, `queue_everywhere`.
`exchanges:default_routing`	`string`	`bybit`	Default CEX fallback router: `binance`, `bybit`, or `mexc`.
`routing:dynamic:enabled`	`boolean`	`false`	Enables/disables dynamically shifting routes based on symbols.
`email:scan_subject`	`string`	`TRADE`	Prefix required in signal email subjects.
`email:coin_pattern`	`string`	`BTC or ETH`	Regex expression used to extract asset tokens from email bodies.
`email:action_pattern`	`string`	`BUY or SELL`	Regex expression used to resolve buy/sell actions in email bodies.
`email:authorized_senders`	`array`	`[]`	List of email addresses authorized to submit trade signals.
`webhook:tradingview:ip_check`	`boolean`	`false`	Toggles dropping payloads that don't originate from TradingView® IPs.

You can sync, check, or reset this entire KV manifest in one command: `hoox config kv apply-manifest`!

Next Steps

- **5-Minute Quick Start Guide** — Deploy all workers and fire your first simulated webhook.
- **API Endpoint Reference** — Review full REST schemas, parameters, and error models.

FAQ

? FREQUENTLY ASKED QUESTIONS (FAQ)

Answers to the most common questions about setting up, operating, and troubleshooting the Hoox trading platform.

Getting Started

Q: How do I clone the repository correctly?

```
git clone --recursive https://github.com/hoox-sh/hoox.git
```

You **must** use `--recursive` because the worker directories are Git submodules. If you already cloned without this flag, run:

```
git submodule update --init --recursive
```

Q: What are the minimum system requirements?

- **Bun** ≥ 1.2 (required — CLI is Bun-only)
- **Git** ≥ 2.40
- **`@hoox-sh/hoox-cli`** via `bun add -g @hoox-sh/hoox-cli` (ships Wrangler as a dependency; prefer `hoox` / `hx` over a separate global Wrangler install)
- Optionally **Docker Desktop** for containerized local development
- A free **Cloudflare® account** with API token (Worker Edit, D1 Edit, KV Edit, R2 Edit, Queues Edit, AI Read)

Q: How do I install and bootstrap?

```
bun add -g @hoox-sh/hoox-cli
git clone --recursive https://github.com/hoox-sh/hoox.git && cd hoox
hoox onboard
```

After the first run inside the monorepo, `hx` works from any directory (path remembered in `~/.hoox/config/monorepo.json`).

Q: Can I use Hoox for free?

Yes! All components — Workers, D1, KV, R2, Queues, Vectorize, Workers AI, Browser Rendering, WAF, and Smart Placement — are within the **Cloudflare® Free Plan** limits. The only potential cost is if you exceed free tier resource limits (100k requests/day, 5M D1 reads/day, etc.).

Q: Do I need to know Solidity or blockchain development?

No. Hoox is designed for **algorithmic traders**, not blockchain developers. On-chain execution is handled by the `web3-wallet-worker` with pre-built EIP-1559 transaction signing. Centralized exchange execution requires only API credentials.

Credentials & Secrets

Q: Where do I get my Cloudflare® API token?

1. Go to dash.cloudflare.com → **My Profile** → **API Tokens**
2. Click **Create Token**
3. Use the **"Edit Cloudflare Workers"** template as a starting point
4. Ensure these permissions are included: `Account > Workers Scripts > Edit`, `Account > D1 > Edit`, `Account > KV > Edit`, `Account > R2 > Edit`, `Account > Queues > Edit`, `Account > AI > Read`

Q: How do I create exchange API keys?

- **Binance**: API Management — Enable "Spot & Margin Trading" permission only. **Never enable "Withdrawals."**
- **Bybit**: API Keys — Select "Read-Only" + "Trade" permissions.
- **MEXC**: API Management — Select "Spot Trading" only.

Q: Are my API keys safe?

Yes. API keys are stored as **Cloudflare® Workers Secrets** — encrypted at rest with hardware-level key management. They are injected directly into the V8 execution isolate at runtime and can never be read back via any API. Prefer `hoox secrets set` / `hoox secrets sync --system` for mesh keys.

Q: How do I paper-trade / use exchange testnets?

Add `"test": true` to the webhook (or queue) JSON. Binance and Bybit route to their public Futures testnet REST hosts; MEXC rejects the request (no public REST sandbox). Prefer dedicated secrets:

```
hoox secrets set trade-worker BYBIT_TESTNET_KEY_BINDING
hoox secrets set trade-worker BYBIT_TESTNET_SECRET_BINDING
```

Test fills are stored as `TEST_EXECUTED` with `*-testnet-*` position ids so they do not mix with live exposure. See **Test Trading**.

Q: Can I rotate API keys without downtime?

Yes. Simply update the secret (worker name is the first argument):

```
hoox secrets set trade-worker BYBIT_KEY_BINDING "new_key_here"
hoox secrets set trade-worker BYBIT_SECRET_BINDING "new_secret_here"
```

The change takes effect on the next worker request — no redeployment needed.

⚡ Deployment & Configuration

Q: What's the difference between `hoox deploy all` and deploying individual workers?

`hoox deploy all --auto` compiles and deploys all enabled workers in their **correct dependency order** (D1 → trade-worker → gateway → agent/telegram → dashboard). Individual worker deploys are useful for hot-fixes but skip dependency validation.

Q: Why does my first deploy take longer than subsequent ones?

The initial deployment provisions Cloudflare resources (D1 databases, KV namespaces, Queue topics, R2 buckets). Subsequent deploys only update the worker code, which is much faster.

Q: How do I change the exchange routing?

Dynamically — no code changes required:

```
# Change default exchange from Bybit to Binance
hoox config kv set exchanges:default_routing binance

# Enable dynamic symbol-based routing
hoox config kv set routing:dynamic:enabled true
```

Q: What happens when I toggle the kill switch on?

The `trade:kill_switch` flag is set to `true` in KV. On the next incoming webhook request, the `hoox` gateway checks this flag and immediately returns a `503 Service Unavailable` response. The agent-worker also runs this check before executing any trailing-stop operations. **Active existing positions are NOT automatically closed** — you must manually flatten them.

🔧 Troubleshooting

Q: I get "409 Conflict" errors. What's wrong?

This means the Durable Object idempotency engine detected a duplicate request. This typically happens when:

- TradingView fires multiple alerts for the same signal (configure `alert.freq_once_per_bar_close`)
- You accidentally reused the same `idempotencyKey` in a manual test

This is **by design** — it protects you from accidentally executing a trade twice. Wait for the TTL expiry (24 hours) or use a unique ID for subsequent signals.

Q: My webhooks work locally but fail in production. Why?

Common causes:

1. **WAF IP blocking** — Enable TradingView IP allow-listing: `hoox waf configure --TradingView-only`
2. **Missing secrets** — Run `hoox secrets list <worker>` to verify all required secrets are bound
3. **Service binding not deployed** — Run `hoox deploy all --auto` to ensure all internal workers are live

Q: How do I check what's happening inside my workers?

```
# Live logs from a specific worker
hoox logs worker trade-worker

# Health status of all workers (single source of truth)
hoox check health

# Recent trade execution history
hoox monitor trades

# Or use the full-screen Terminal UI
hoox tui
```

Q: What should I do if `hoox repair check` fails?

Follow the guided repair prompts:

1. Submodule failure → `git submodule update --init --recursive`
2. Dependency failure → `bun install`
3. Type error → `bun run typecheck`
4. Missing Cloudflare binding → `hoox repair infra`
5. Missing secret → `hoox secret set <NAME> <VALUE>`

AI & Advanced Features

Q: Do I need AI providers for basic trading?

No. The `agent-worker` can run trailing-stop and kill-switch logic **without any AI provider**. AI providers are only needed for:

- Conversational chat (`/agent/chat`)
- Chart image analysis (`/agent/vision`)
- Complex strategy reasoning (`/agent/reasoning`)
- Natural language Telegram bot interactions

Q: Which AI provider is best?

All five are supported with automatic fallback ordering:

1. **Workers AI** (Cloudflare) — Free with generous limits, runs at the edge
2. **Anthropic** (Claude) — Strong reasoning, complex analysis
3. **Google AI** (Gemini) — Large context windows, multimodal
4. **OpenAI** (GPT-4o) — Most capable general-purpose model
5. **Azure OpenAI** — Enterprise SLA, private networking

Providers are tried sequentially on failure. You only pay for the one that responds.

Q: What does the agent-worker cron do exactly?

The `agent-worker` runs on a **configurable Cloudflare Cron** (any interval from **1 to 1440 minutes**; default **15 minutes** via `*/15 * * * *` in `wrangler.jsonc`) and performs:

1. Reads all open positions from D1
2. Fetches current market prices from exchange APIs
3. Calculates trailing stop-loss prices
4. Checks if daily drawdown limit is exceeded → kills switch if so
5. Partially closes positions that hit take-profit targets
6. Sends a Telegram health summary

To change the interval, update `triggers.crons` and redeploy. See agent-worker Cron setup.

Infrastructure

Q: What data is stored in D1 vs KV vs R2?

Store	What Lives Here
D1 (SQLite)	Trade fills, open positions, balance history, system logs, trade signals
KV	Runtime config (kill switch, exchange routing, rate limiter state, AI keys, email patterns)
R2	Raw API request/response logs, generated PDF reports

Q: How does Hoox handle exchange downtime?

When a direct exchange API call fails (timeout, rate limit, HTTP 5xx), the gateway automatically enqueues the trade payload into **Cloudflare Queues** with exponential backoff:

- Retry 1: 30 seconds
- Retry 2: 1 minute
- Retry 3: 5 minutes
- Retry 4: 15 minutes
- Retry 5: 30 minutes After max retries, the trade is logged as failed (not lost).

Q: Can I use a custom domain?

Yes. Cloudflare Workers supports custom domains. After deploying, configure your domain in the Cloudflare dashboard under **Workers & Pages → Custom Domains**.

Comparisons

Q: Why edge workers instead of a VPS?

Factor	VPS	Cloudflare Edge
Latency to Binance (Frankfurt)	80-150ms	2-8ms
Cold-start time	1,000-15,000ms	<3ms
Monthly cost	$5-50+	$0 (free tier)
Maintenance	Manual updates	Fully managed
DDoS protection	Self-managed	Built-in WAF
Global failover	Complex setup	Automatic

Q: Why Cloudflare over AWS Lambda@Edge?

- **No cold starts** — V8 isolates stay warm between requests
- **True global distribution** — 330+ PoPs vs. Lambda@Edge's 6 regions
- **Native SQLite** — D1 is first-class, not bolted-on
- **Built-in queuing** — Cloudflare Queues vs. SQS
- **Zero egress fees** — R2 is S3-compatible with no bandwidth charges

---

Open an issue on GitHub or join our community discussions.

GLOSSARY

GLOSSARY

A curated vocabulary of terms used throughout the Hoox documentation. Understanding these terms is essential for effective navigation of the platform.

General Trading

Term	Definition
Webhook	An automated HTTP POST callback triggered by an external system (e.g., TradingView®) when a specific event occurs, such as a technical indicator crossover.
Signal	A structured JSON payload sent to the Hoox Gateway instructing it to execute a trade. Signals originate from TradingView® alerts, email parsing, or Telegram commands.
Pine Script™	TradingView®'s domain-specific language for indicators and strategies. Hoox docs use `//@version=6`. Edge evaluate also runs via **pyne-worker** / `hoox pyne`.
Leverage	
LONG	A trade direction indicating the purchase (buy) of an asset with the expectation that its price will rise.
SHORT	A trade direction indicating the sale (sell) of an asset with the expectation that its price will fall.
CLOSE	An action to flatten (exit) an existing open position, returning the account to a neutral state.
Side (BUY/SELL)	The translated exchange-level direction. `LONG` maps to `BUY`, `SHORT` maps to `SELL`.
PositionSide	The margin mode direction (`LONG` or `SHORT`) used in hedge-mode futures trading.
Trailing Stop-Loss	A dynamic stop-loss order that automatically adjusts as the market moves in the trader's favor, locking in profits while limiting downside.
Take-Profit (TP)	A target price at which a position is automatically closed to lock in profits.
Drawdown	The percentage decline in account equity from its peak to its current level.
Liquidation	The forced closure of a leveraged position when margin requirements are no longer met.
Fill	A successfully executed trade order on the exchange.
Slippage	The difference between the expected price of a trade and the actual price at which it is executed, often caused by market volatility or latency.
Test trading	Sandbox execution mode enabled by `"test": true` on a webhook/queue payload. Routes to exchange testnet hosts (Binance/Bybit); ledger status `TEST_EXECUTED`.
TEST_EXECUTED	D1 trade status written for testnet fills so live reports and stats can exclude them.
Testnet position id	Position primary key pattern `{exchange}-testnet-{symbol}-{side}` isolating sandbox exposure from live OPEN rows.

Architecture & Infrastructure

Term	Definition
Edge Worker	A lightweight JavaScript function running on Cloudflare®'s globally distributed edge servers, executing code as close to end users (or data sources) as possible.
V8 Isolate	A sandboxed JavaScript runtime instance within Cloudflare® Workers, providing strong isolation between tenants with zero shared memory.
pyne-worker	Python PYNE tooling isolate for Pine Script™ edge evaluate (`/run` , bar-close cron, R2 OHLCV); managed via `hoox pyne` . Mesh-auth when forwarding to trade-worker.
Microservice	A small, independently deployable service that performs a single business function within the Hoox architecture.
Service Binding	A Cloudflare feature enabling direct, low-latency communication between edge workers inside the V8 engine — no public internet routing required.
Durable Object (DO)	A Cloudflare primitive providing single-threaded, persistent server-side state. Used by Hoox for idempotency locks and distributed coordination.
D1 Database	Cloudflare's serverless, edge-native SQLite database offering ACID-compliant transactions with sub-5ms query latency.
KV (Key-Value Store)	A globally distributed, highly available key-value store optimized for high-read/low-write operations at sub-millisecond latency.
R2 Object Storage	An S3-compatible, zero-egress-fee object storage service for storing arbitrary data (logs, reports, backups).
Cloudflare Queues	A serverless message broker providing at-least-once delivery with built-in exponential backoff retry for asynchronous processing.
Vectorize	A serverless vector search database enabling semantic matching and retrieval-augmented generation (RAG).
Browser Rendering	A Cloudflare service providing headless Chrome instances at the edge for automated web interaction and PDF generation.
Analytics Engine	A time-series metrics platform for tracking API call latency, error rates, and custom events across all workers.
Smart Placement	A Cloudflare optimization that automatically deploys worker code to the edge node geographically closest to its external API dependencies.
Zero Trust Architecture	A security model where no service trusts any other by default. All access is authenticated, authorized, and encrypted — even internal communications.
Kill Switch	An emergency mechanism (stored in KV) that instantly halts all trade execution globally when activated, requiring no code redeployment.

Security

Term	Definition
Idempotency	The property ensuring that a given operation (e.g., a trade signal) produces the same result whether executed once or multiple times — preventing duplicate trades.
HMAC-SHA256	A cryptographic signing algorithm used to authenticate order payloads sent to exchange APIs, ensuring message integrity and origin verification.
Secret Binding	An encrypted credential injected directly into a V8 isolate at runtime. Secrets are never stored in code, config files, or logs.
WAF (Web Application Firewall)	A network-layer security service that filters, monitors, and blocks HTTP traffic based on customizable rules (e.g., IP allow-listing).
CORS (Cross-Origin Resource Sharing)	A browser security mechanism. Hoox disables CORS on the gateway to prevent unauthorized cross-domain requests.
IP Allow-listing	Restricting access to the webhook endpoint to a predefined list of trusted IP addresses (e.g., TradingView®'s known ranges).

Development & Tooling

Term	Definition
Bun	A fast JavaScript runtime, package manager, and test runner used as Hoox's primary development toolchain.
monorepo	A repository structure containing multiple packages/workers managed together with shared dependencies and tooling.
Remembered monorepo	Path stored in `~/.hoox/config/monorepo.json` after the CLI discovers a local checkout, so `hx` works from any working directory.
`HOOX_REPO`	Environment variable forcing the monorepo root; highest priority in CLI workspace resolution.
Bun Workspace	A Bun feature enabling multiple packages within a monorepo to share dependencies and reference each other locally.
Wrangler	Cloudflare®'s official CLI for Workers. Hoox embeds it; prefer `hoox` / `hx` for operator workflows.
Linear Rail banner	Default CLI banner style (`◆ H · O · O · X`) with tagline/version; compact TTY animation.
Docker Compose	A container orchestration tool used by Hoox for running isolated local development environments with all services.
TUI (Terminal User Interface)	A full-screen, keyboard-driven terminal dashboard (`./hoox-tui`) for monitoring and controlling the Hoox platform.
OpenTUI	A React-based framework for building terminal user interfaces, used to construct Hoox's TUI.
OpenNext	An adapter enabling Next.js applications to run on Cloudflare Workers. Hoox uses it for the dashboard Command Center.
Drizzle	A lightweight TypeScript ORM used with D1 for database schema management and migrations.
CI/CD	Continuous Integration / Continuous Deployment. Hoox's pipeline runs linting, type checks, unit tests, and build verification.

Exchange-Specific

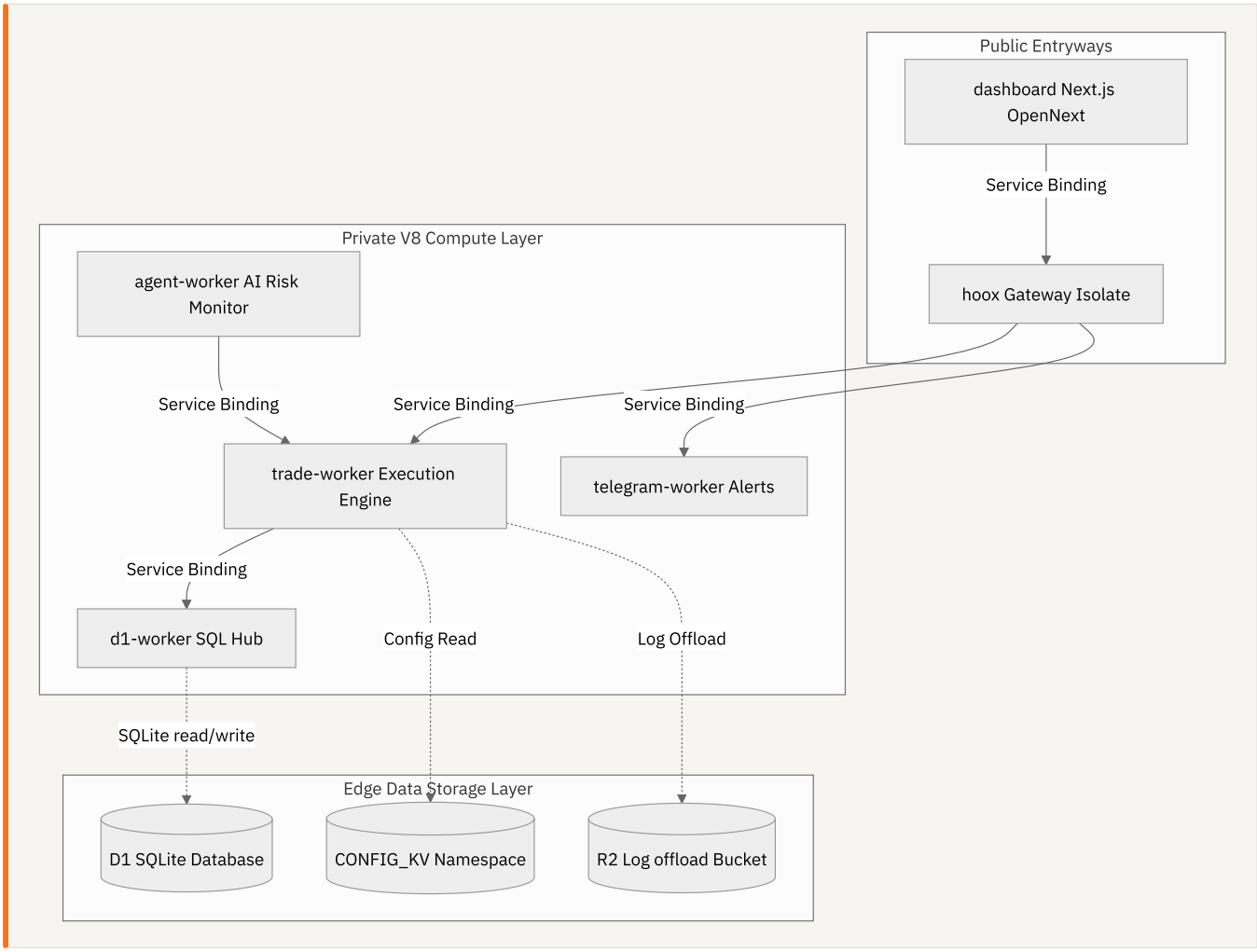
Term	Definition
MEXC	A global cryptocurrency exchange supporting spot and futures trading. One of three supported exchanges.
Bybit	A cryptocurrency derivatives exchange known for its high-performance API. One of three supported exchanges.
Binance	The world's largest cryptocurrency exchange by volume. One of three supported exchanges.

Cross-referenced from nearly every section of the documentation.

⚙ DEVOPS MANUAL

Infrastructure provisioning, deployment sequences, secret management, and edge operations reference 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. Day-to-day ops are **CLI-first**: `hoox` / `hx` for `onboard`, `setup`, `deploy`, `secrets`, `check health`, and `doctor`.



Docs navigation (tabs)

HOOX docs use the same **multi-tab** layout as AXIS / PYNE. This **DevOps** tab is for install, local development, deploy, security, and tooling. Sibling tabs hold the rest of the platform:

Tab	Contents
End User	Getting started, concepts, guides, tutorials
Architecture	Topology, data flow, storage, ADRs
Workers	Per-isolate profiles
DevOps (this page)	Setup, development, deployment, security, CLI/TUI
API	HTTP endpoints, payloads, responses
Enterprise	Multi-tenancy, compliance (commercial notes)

Operator's System Directory

1. Setup runbooks

- **Operations & Troubleshooting** — Deploy order, secrets modes, health probes, repair.
- **Core Installation Flow** — `hoox onboard` / monorepo remember path / setup gates.
- **Bindings registry** — Cross-worker binding inventory.

2. Development

- **Wrangler Dev Setup** · **Testing** · **Debugging**

3. Deployment, WAF & CI/CD

- **Production Deployment** — Wrangler, account provisioning, production vars.
- **CI/CD** — GitHub Actions and edge uploads.
- **Monitoring** — Logs and Analytics Engine.
- **Zero Trust** · **Private Ingress**

4. Security & tooling

- **Security overview**
- **CLI features** (0.13.x: mesh hardening notes, monorepo remember, secrets modes, Linear Rail, setup gates) · **TUI** · **Pattern consolidation** · **Security**

5. Related tracks (other tabs)

- **Architecture hub** · **Workers hub** · **API hub**
- **Test Trading** — sandbox mode across the mesh

First production deploy? Start with **Production Deployment** to verify Cloudflare permissions and sequential worker rollout.

Quick links

- **Documentation home** — All category hubs
- **Enterprise** — Multi-tenancy and compliance notes

- **OPEN_CORE.md** — Open-core vs commercial split

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.

Workspace `wrangler.jsonc` is validated by the CLI (show/set/check setup and schema commands). Prefer the documented shape and presets rather than bypassing validation when extending worker parameters.

Onboarding Wizard (`hoox onboard`)

The recommended entry point is **`hoox onboard`** (alias: `hx onboard`). It chains `init` (configuration) and `setup` (infrastructure) into a single command. Running `hoox` / `hx` with no arguments on an uninitialized workspace auto-launches `onboard` (or `onboard --resume` when `.wizard-state.json` exists).

Setup gates (0.11+ / current 0.13.x): onboard will **not** run setup if `init` was cancelled or never wrote `wrangler.jsonc`. Setup also aborts when worker trees are still missing after the submodule clone attempt, and requires Cloudflare® auth (`wrangler login` or `CLOUDFLARE_API_TOKEN` + `CLOUDFLARE_ACCOUNT_ID`).

To start the system bootstrap, run the one-shot wizard from the monorepo root (the CLI remembers that path so later `hx` calls work from any directory):

```
cd /path/to/hoox   # first discovery writes ~/.hoox/config/monorepo.json
hoox onboard
# later, from any directory:
hx doctor          # shows Remembered + Runtime root
hx check health
```

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: Worker Preset Selection

Chooses a **preset** rather than free-form per-worker toggles in the wizard:

Preset	Focus
`minimal`	Gateway + D1 + analytics — webhook processing only
`standard`	Trading + analytics + Telegram notifications
`full`	All workers (AI agent, DeFi wallet, email, pyne-worker / Pine Script™, ...) + integrations

Pass `--preset minimal|standard|full` for non-interactive runs (`hoox onboard` / `hoox init`). You can still edit `wrangler.jsonc` afterward to refine enabled workers.

Phase 3: Integration Secrets

Collects any integration secrets required by the selected preset (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 (via `hoox setup` / onboard chain).

Phase 6: Verification (suggested next step)

Does **not** auto-run validation. When setup finishes successfully, the CLI **suggests** `hoox check setup` as the next step so you can verify config, infra, secrets, and database health.

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. `.wizard-state.json` (Onboarding State)

During the interactive setup, the CLI caches your current step and intermediate inputs inside `.wizard-state.json` at your project root (constant `WIZARD_STATE_PATH` in `@hoox-sh/hoox-shared`).

- **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. Running `hoox` with no args and no `wrangler.jsonc` also resumes via `onboard --resume` when this file exists.
- **Auto-Cleanup**: When the wizard completes successfully (or when the workspace is already initialized), the state file is cleaned up so the project root stays tidy.

Secret Bindings Architecture

Hoox utilizes Cloudflare® 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? Re-run `hoox check setup` at any time for high-integrity validation of config, infra, secrets, and bindings. Workspace `wrangler.jsonc` is also checked by `hoox schema validate` and related CLI commands.

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.
- **CLI features** — Monorepo remember path, secrets modes, Linear Rail banner.

35 · DEVOPS MANUAL

HOOX TRADING SYSTEM — COMPLETE SETUP & OPERATIONS GUIDE

CLI-first setup, deploy, secrets, health, and repair runbook for the HOOX Cloudflare® Workers mesh.

> **CLI:** `@hoox-sh/hoox-cli` 0.13.x (`hoox` / `hx`)
> **Last Updated:** 2026-08-13
> **Classification:** Internal Operations Manual
> **Audience:** DevOps Engineers, Senior TypeScript Developers, System Administrators

Prefer the **CLI** for day-to-day ops: `hoox onboard` → `hoox setup` → `hoox deploy all` → `hoox check health`. After the first monorepo discovery, `hx` works from any directory (remembered path under `~/.hoox/config/monorepo.json`). Diagnose layout with `hoox doctor`; operator plane with `hoox doctor --security`.

Table of Contents

---

1. System Overview

1.1 Architecture

Hoox is an edge-deployed cryptocurrency trading system built on Cloudflare® Workers. It consists of **11 compute surfaces** (10 workers + dashboard):

Layer	Components
Gateway	`hoox` (webhook entrypoint, DO idempotency + DO rate limiter, notify allowlist)
Execution	`trade-worker` (multi-exchange), `web3-wallet-worker` (DeFi)
Intelligence	`agent-worker` (AI risk manager, configurable 1–1440 min 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)
Tooling	`pyne-worker` (Python Pine Script™ edge evaluate — `API_KEY` tooling auth)
Dashboard	`workers/dashboard` (Next.js 16 + OpenNext on Cloudflare® Workers)
CLI	`packages/cli` (`hoox` / `hx` lifecycle 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 → trade-worker
pyne-worker (cron /run) → trade-worker (mesh internal auth on live forward)
```

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, migration `v1`)
Durable Objects	`RateLimiterStore`	`RATE_LIMITER`	hoox (atomic trade rate limits, migration `v2`)
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/hoox-sh/hoox.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 sync`. 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

Secret Name	Worker(s)	Set Via	Required For	Description
`AUTHORIZED_CHAT_IDS`	telegram-worker	`wrangler secret put`	Required (webhook); recommended (alert allowlist)	Comma-separated chat IDs; webhook fail-closed when unset; `/alert` restricts to list or `TG_CHAT_ID_BINDING`
`TELEGRAM_ALLOWED_CHAT_IDS`	hoox	`wrangler secret put`	Required for notify	Comma-separated chat IDs for public webhook notify (fail-closed when unset); alias `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
`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
`API_KEY`	pyne-worker	`hoox secrets set`	Optional*	PYNE evaluate / management (`X-API-Key`); *required for production

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-worker",
      "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/hoox-sh/hoox.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 (submodules + dashboard):
# workers/hoox-worker
# 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
# workers/pyne-worker
# workers/dashboard
```

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 ≥1.2 and `@hoox-sh/hoox-cli` (`bun add -g @hoox-sh/hoox-cli`)
4. Authenticate: `wrangler login` (or set `CLOUDFLARE_API_TOKEN` + `CLOUDFLARE_ACCOUNT_ID`)

5.2 Phase 2: Repository Setup

```
# Preferred one-shot (from monorepo root — path is remembered for later hx)
git clone --recursive https://github.com/hoox-sh/hoox.git
cd hoox
bun install
hoox doctor                   # Remembered + Runtime root
hoox onboard                  # init + setup (auth, keys, D1, secrets, dashboard)

# Or manual clone / verify
bun run check:worker-submodules
```

5.3 Phase 3: Infrastructure Provisioning

`hoox onboard` / `hoox setup` and `hoox infra` cover D1/KV/R2/queues for typical installs. See Section 6 for raw wrangler equivalents.

5.4 Phase 4: Configuration

```
# Interactive config (writes wrangler.jsonc)
hoox init
# Or edit wrangler.jsonc / .env.local after onboard:
# - cloudflare_account_id, subdomain_prefix, enabled workers
```

5.5 Phase 5: Secret Deployment

```
# Push mesh/system secrets only (recommended after key rotation)
hoox secrets sync --system
# Alias: hoox secrets sync --required

# Or push all declared secrets from .dev.vars
# (reports partial sync: synced / skipped placeholders / failed)
hoox secrets sync

# Or set individually per worker:
hoox secrets set hoox WEBHOOK_API_KEY_BINDING
wrangler secret put WEBHOOK_API_KEY_BINDING --config workers/hoox-worker/wrangler.jsonc
```

5.6 Phase 6: Database Setup

```
# Preferred
hoox setup --skip-keys --skip-secrets --skip-dashboard
# or: hoox db apply (when using db command group)

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

5.7 Phase 7: KV Configuration

```
# Manifest-driven (when configured)
hoox deploy kv-config

# Or set required KV keys manually
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

```
hoox deploy all --auto
# Order: see Section 7 (includes pyne-worker)
```

5.9 Phase 9: Dashboard Deployment

```
# Preferred (CLI)
hoox deploy dashboard
# or: hoox dashboard deploy

# Manual OpenNext path
cd workers/dashboard
bun run opennext:build
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 Durable Object migrations (`v1` + `v2`):

```
// Already defined in workers/hoox-worker/wrangler.jsonc(.example)
"durable_objects": {
  "bindings": [
    {
      "name": "IDEMPOTENCY_STORE",
      "class_name": "IdempotencyStore"
    },
    {
      "name": "RATE_LIMITER",
      "class_name": "RateLimiterStore"
    }
  ]
},
"migrations": [
  {
    "tag": "v1",
    "new_sqlite_classes": ["IdempotencyStore"]
  },
  {
    "tag": "v2",
    "new_sqlite_classes": ["RateLimiterStore"]
  }
]
```

Applied automatically on `wrangler deploy` / `hoox deploy worker hoox`. New tags run once per account/script. `RATE_LIMITER` is required for atomic multi-isolate rate limits; without the binding, the gateway falls back to `CONFIG_KV` / in-memory.

---

7. Worker Deployment Sequence

Deploy order matters due to service bindings. The CLI sequence below matches `DEPLOY_ORDER` in `packages/cli` (`hoox deploy all` / `hoox deploy workers`).

7.1 Deployment Order

Order used by the CLI when deploying all enabled workers:

```
1.  analytics-worker    (observability fan-in; few dependencies)
2.  d1-worker           (SQL hub)
3.  telegram-worker     (notifications)
4.  web3-wallet-worker  (DeFi identity)
5.  email-worker        (email signal ingress)
6.  trade-worker        (execution — needs d1 / telegram bindings)
7.  pyne-worker         (Pine Script™ tooling; live forward → trade-worker)
8.  report-worker       (PDF / Browser Rendering)
9.  agent-worker        (AI risk — needs trade / d1 / telegram)
10. hoox                (gateway — last worker so bindings resolve)
11. dashboard           (OpenNext UI — after services are live)
```

Unknown workers (not in the list) are appended before `dashboard`.

7.2 Deployment Commands

```
# Preferred: deploy all enabled workers + dashboard (CLI applies order)
hoox deploy all
hoox deploy all --auto          # non-interactive

# Workers only (skip dashboard)
hoox deploy workers

# One isolate
hoox deploy worker trade-worker
hoox deploy worker pyne-worker
# pyne also has vendor+deploy helper:
hoox pyne deploy                # sync-vendor then wrangler deploy
```

7.3 Post-Deployment: Telegram Webhook

```
# Preferred CLI helper
hoox deploy telegram-webhook
# or with explicit token:
hoox deploy telegram-webhook --token <TELEGRAM_BOT_TOKEN>

# Manual fallback
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 deploy 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 one secret via hoox CLI (writes .dev.vars + puts to Cloudflare)
hoox secrets set <WORKER_NAME> <SECRET_NAME>

# Sync mesh/system secrets only (INTERNAL_KEY_BINDING, WEBHOOK_*, …)
hoox secrets sync --system
# Alias: hoox secrets sync --required

# Sync all declared secrets from .dev.vars
# Partial results report synced / skipped (placeholders) / failed
hoox secrets sync
hoox secrets sync trade-worker
```

9.2 Local Development Secrets

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

```
# workers/hoox-worker/.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

```
# Workspace / toolchain
hoox doctor                   # Remembered monorepo + runtime root
hoox check prerequisites      # bun, git, wrangler, CF auth
hoox check setup              # Config, infra, secrets, database

# Secrets inventory (names only)
hoox secrets list

# Worker health — single source of truth (GET /health per enabled worker)
hoox check health
hoox check health --json

# Specialist probes
hoox agent health
hoox pyne health              # pyne-worker GET /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

Public liveness is **`GET /health`** on each isolate (unauthenticated). Prefer `hoox check health` over ad-hoc curls; the snippets below are for debugging.

10.2.1 Gateway Health

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

# Expected: {"status":"ok"} (or equivalent healthy JSON)
```

10.2.2 Trade Worker Health

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

# Protected signals endpoint (mesh / service key — not public)
curl https://trade-worker.<SUBDOMAIN_PREFIX>.workers.dev/api/signals \
  -H "X-Internal-Auth-Key: <INTERNAL_KEY_BINDING>"
```

10.2.3 Agent Worker Health

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

# Or via CLI
hoox agent health
```

10.2.4 D1 Worker Health

```
curl https://d1-worker.<SUBDOMAIN_PREFIX>.workers.dev/health

# Authenticated SQL (mesh)
curl -X POST https://d1-worker.<SUBDOMAIN_PREFIX>.workers.dev/query \
  -H "Content-Type: application/json" \
  -H "X-Internal-Auth-Key: <INTERNAL_KEY_BINDING>" \
  --data '{"query":"SELECT 1"}'
```

10.2.5 Telegram Worker Health

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

10.2.6 Analytics Worker Health

```
curl https://analytics-worker.<SUBDOMAIN_PREFIX>.workers.dev/health
```

10.2.7 PYNE Worker Health

```
# Tooling isolate — public /health; evaluate routes need X-API-Key
curl https://pyne-worker.<SUBDOMAIN_PREFIX>.workers.dev/health
hoox pyne health
# or: hoox pyne health --url https://pyne-worker.<SUBDOMAIN_PREFIX>.workers.dev
```

10.2.8 Dashboard Health

```
# Check dashboard (OpenNext worker URL depends on wrangler name)
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 enabled workers return healthy from `hoox check health` (`GET /health`)
- [ ] D1 database has all required tables
- [ ] KV namespace has required configuration keys
- [ ] Secrets are set for all enabled workers (`hoox secrets list` / `hoox check setup`)
- [ ] Service bindings resolve correctly (no 502/503 errors)
- [ ] Queue is configured and consumer is registered
- [ ] Telegram webhook is set and responding (`hoox deploy telegram-webhook`)
- [ ] Dashboard accessible and authenticated
- [ ] Analytics Engine receiving data points
- [ ] If enabled: `hoox pyne health` OK; `API_KEY` set for production evaluate routes
- [ ] Cron triggers scheduled (agent-worker: 1–1440 min via `triggers.crons`, default 15 min; pyne-worker: `* * * * *` bar-close)

---

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 deploy worker <worker-name>

# Check worker logs
hoox workers logs <worker-name>
# or: hoox logs worker <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 mesh keys are missing after rotation, re-upload system secrets only
hoox secrets sync --system

# Or re-upload all declared secrets from .dev.vars (partial sync reported)
hoox secrets sync

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

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_checl
wrangler kv:key put --namespace-id=c5917667a21745e390ff969f32b1847d "trade:max_daily_drawdown_pe
```

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 + dashboard (CLI order)
hoox deploy all --auto

# 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) or: hoox deploy kv-config

# 7. Reconfigure Telegram webhook
hoox deploy telegram-webhook

# 8. Redeploy dashboard alone (if needed)
hoox deploy dashboard
```

---

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 internal/mesh keys (writes to .keys/ and .dev.vars)
hoox keys generate

# Push mesh/system secrets only to Cloudflare
hoox secrets sync --system

# Or rotate a single integration secret
hoox secrets set <WORKER_NAME> <SECRET_NAME>

# 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-worker/wrangler.jsonc`	`workers/hoox-worker/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 sync --system` (mesh) or `hoox secrets sync`
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 --system    # Push mesh/system secrets only
hoox secrets sync             # Push all .dev.vars (partial reporting)

# 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 all               # Workers + dashboard (ordered)
hoox deploy workers           # Deploy enabled workers only
hoox deploy worker <name>     # Deploy one worker
hoox deploy dashboard         # Deploy dashboard
# OR: hoox dashboard deploy
hoox pyne deploy              # Vendor pynescript + deploy pyne-worker

# Operations
hoox doctor                   # Paths / remembered monorepo / TUI entry
hoox check health             # Worker health (GET /health; single source of truth)
hoox workers logs <name>      # View worker logs
hoox check setup              # Validate setup
hoox secrets list <worker>    # Check secrets
hoox secrets sync --system    # Mesh keys only
hoox secrets sync             # All .dev.vars (partial reporting)

# 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 sync --system        # mesh keys only
hoox secrets sync                 # all from .dev.vars (partial reporting)

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

Appendix B: Directory Structure

```
hoox/
├── .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/                      # @hoox-sh/hoox-cli (hoox / hx)
│   │   ├── bin/hoox.js           # CLI entry
│   │   └── src/
│   │       ├── index.ts          # Command dispatcher
│   │       ├── commands/         # CLI commands
│   │       ├── services/         # setup, secrets, workspace, CF, …
│   │       ├── ui/               # Interactive menu + Linear Rail banner
│   │       └── utils/            # formatters, theme, help
│   ├── tui/                      # @hoox-sh/hoox-tui (OpenTUI)
│   └── shared/                   # Shared types/utilities
│       └── src/
│           ├── types.ts          # Core types
│           ├── router.ts         # Custom router
│           ├── middleware/       # Auth, rate-limit, logger
│           └── errors.ts         # Error factories
│
├── workers/
│   ├── hoox-worker/              # Gateway worker (published name: hoox)
│   ├── 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
│   ├── pyne-worker/              # Pine Script™ edge evaluate (Python)
│   └── 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: inbound webhook fail-closed when unset; outbound `/alert` allowlist (else only `TG_CHAT_ID_BINDING`)
`TELEGRAM_ALLOWED_CHAT_IDS`	Secret	hoox	Comma-separated chat IDs for webhook notify (fail-closed when unset); alias `AUTHORIZED_CHAT_IDS`; optional `CONFIG_KV` `telegram:allowed_chat_ids`
`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

Durable Object Bindings

Binding	Worker	Class	Migration	Description
`IDEMPOTENCY_STORE`	hoox	`IdempotencyStore`	`v1`	Exactly-once trade fingerprint store
`RATE_LIMITER`	hoox	`RateLimiterStore`	`v2`	Atomic multi-isolate trade rate limits (`MAX_TRADES_PER_MINUTE`)

Both classes are exported from `workers/hoox-worker/src/index.ts`. `wrangler deploy` applies new migration tags once. Without `RATE_LIMITER`, the gateway falls back to `CONFIG_KV` / in-memory counters (best-effort).

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.

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-worker/.dev.vars.example workers/hoox-worker/.dev.vars

# 2. Inject local mock keys for testing
echo 'INTERNAL_KEY_BINDING="local_test_key"' >> workers/hoox-worker/.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-metadata.json`	Human-authored worker/infra semantics (tracked in git)
`graph.json`	Full machine-readable graph — **generated**, not committed
`graph.dot`	Graphviz DOT — **generated**, not committed (`bun run graph`)

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.

38 · DEVELOPMENT

TESTING FRAMEWORK & QA STANDARDS

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

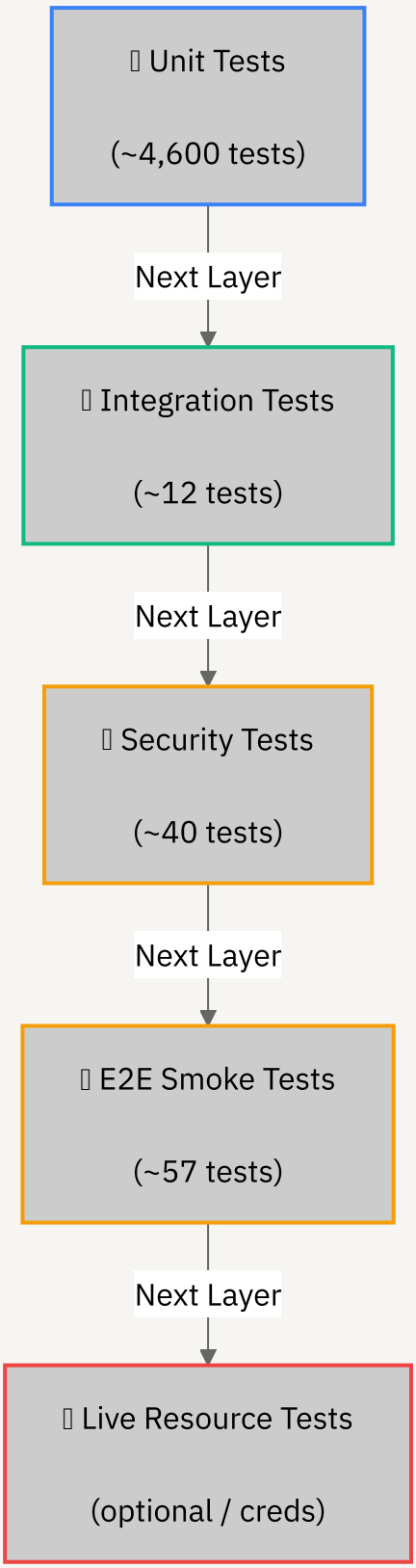
To protect live capital and ensure robust order routing, Hoox mandates a rigorous testing pipeline. With money on the line, we verify every contract calculation, rate-limiting gate, and database query.

Our test suite is powered natively by **Bun's high-speed test runner**, comprising **~250 test files** and **~9,300 `expect()` assertions** (~4,600+ unit/package tests) split across five diagnostic layers.

As of 2026-08 (line coverage, package-scoped `bun test --coverage`):

Workspace	Line coverage (approx.)	Tests (pass)
`packages/cli`	~80% overall (~95% on `src/` command/services)	~1,260
`packages/shared`	~98%	~780
`packages/tui`	~58% (views still light; pure helpers high)	~900
Edge workers (aggregate)	~70–94% per isolate	~1,300
Dashboard unit	~25% (component-heavy)	~330

The 5 QA Testing Layers



⚡ 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**: CI and contributor standards target **≥80% line coverage** on core execution paths (`packages/cli`, `packages/shared`, gateway/trade workers), with aspirational **90% line / 95% function** floors in `.opencode/context/core/standards/test-coverage.md`. Shared is already ~98%; CLI `src/` averages ~95%. Per-file CLI floor: `bun run coverage:check` (50% hard floor on `packages/cli/src/**`, ratcheting upward).
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/cli/ --coverage
bun test packages/shared/ --coverage
bun run coverage:check   # CLI per-file floor (needs lcov from a coverage run)
```

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.

39 · 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 worker trade-worker

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

# C. Alternatively, invoke Wrangler directly for custom configurations
npx wrangler tail hoox --config workers/hoox-worker/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 secret names on both workers: `hoox secrets list hoox` / `hoox secrets list trade-worker`.
  2. Re-generate mesh keys if needed, then push system secrets only:

     ```
     hoox keys generate
     hoox secrets sync --system
     ```

     Or set per worker: `hoox secrets set <worker> INTERNAL_KEY_BINDING`

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.

40 · 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® 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.
- **Gateway DO migrations**: `hoox` wrangler must declare `IDEMPOTENCY_STORE` (`v1`) and `RATE_LIMITER` / `RateLimiterStore` (`v2`). Deploy applies migrations; no fake resource IDs.
- **Notify chat allowlists** (names only — never commit values):
  - hoox: `TELEGRAM_ALLOWED_CHAT_IDS` (or alias `AUTHORIZED_CHAT_IDS`)
  - telegram-worker: `AUTHORIZED_CHAT_IDS`, `TG_CHAT_ID_BINDING`, `TELEGRAM_SECRET_TOKEN`, `TG_BOT_TOKEN_BINDING`
- **Post-hardening gateway checklist**: `workers/hoox-worker/DEPLOY.md`

---

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 (includes pyne-worker when enabled)
hoox deploy all --auto

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

# 3. PYNE tooling isolate (vendor + deploy)
hoox pyne deploy

# 4. Post-deploy health
hoox check health
```

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-worker/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.

41 · DEPLOYMENT

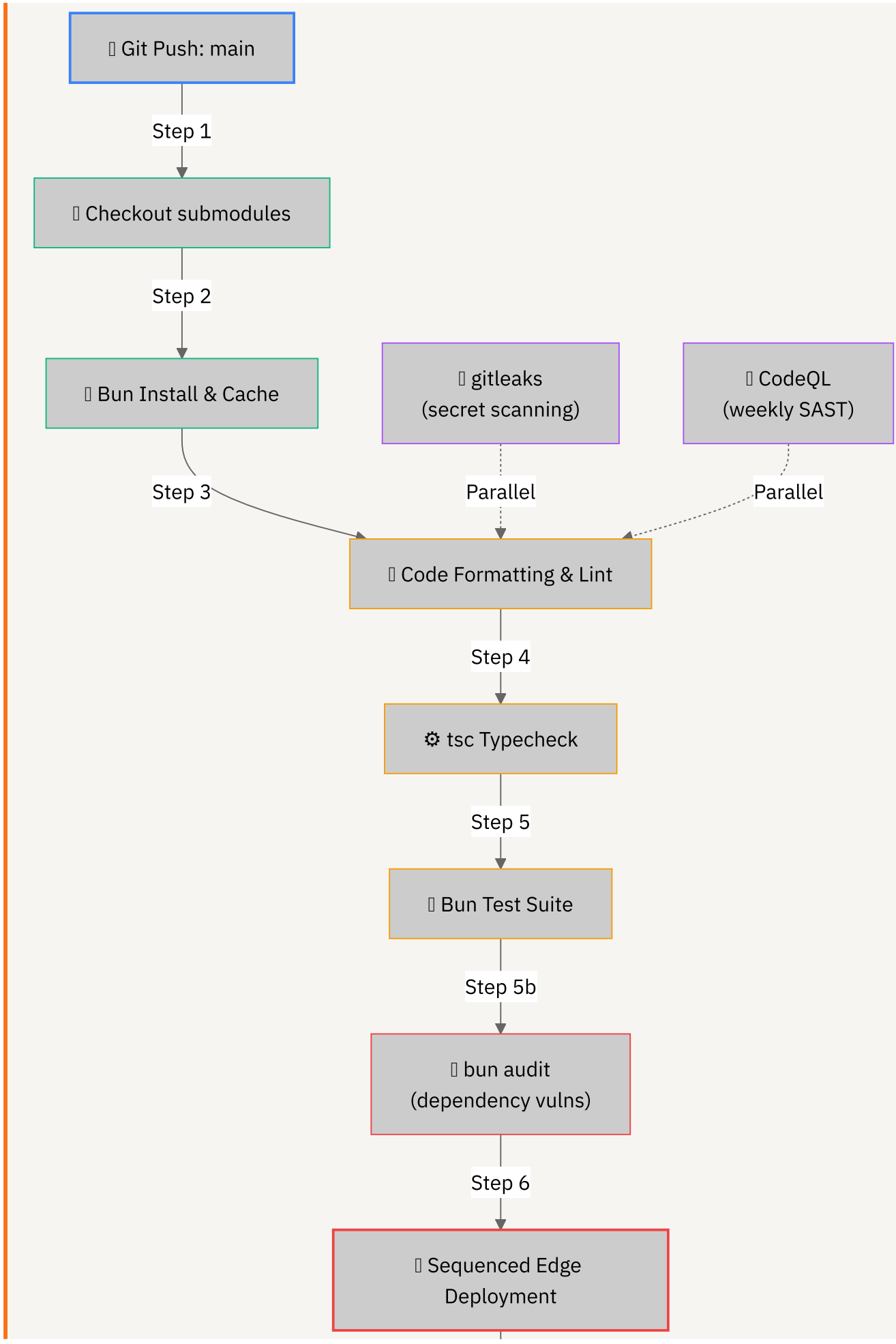
CI/CD PIPELINE AUTOMATION

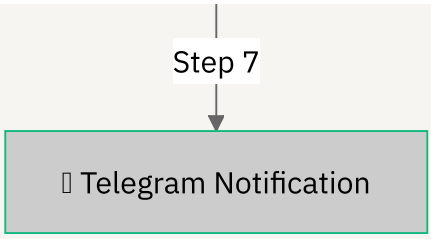
How to automate the Hoox deployment pipeline using GitHub Actions, including code style checks, TypeScript type-checking, Bun unit testing, and Cloudflare edge uploads.

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


Git Push: main
Step 1
Checkout submodules
Step 2
Bun Install & Cache
gitleaks
(secret scanning)
CodeQL
(weekly SAST)
Step 3
Parallel
Parallel
Code Formatting & Lint
Step 4
tsc Typecheck
Step 5
Bun Test Suite
Step 5b
bun audit
(dependency vulns)
Step 6
Sequenced Edge
Deployment



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.

42 · 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: Insu
    }),
  });
}
```

Need to tail live worker logs to debug a custom strategy? Run `hoox logs worker 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.

43 · 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 **operator management APIs** 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.

CLI / TUI remote operators

`hoox tui --remote` is **fail-closed**: it requires `HOOX_API_TOKEN` / `--token`, **or** Cloudflare Access service-token env vars:

```
# Server (Worker secret) — must match client token
wrangler secret put OPERATOR_API_KEY

# Client

hoox tui --remote --api-url https://mgmt.example.com
```

Management routes (all require Bearer):

Method	Path	Purpose
`GET`	`/v1/health`	Operator auth + plane probe
`GET`	`/v1/workers`	Minimal worker list (JSON array)
`GET`	`/v1/trades/stream`	SSE trades (polls trade-worker)
`GET`	`/v1/logs/stream`	SSE logs (polls trade-worker)

`GET /health` stays public for liveness. Do **not** put operator Bearer on TradingView® `/webhook`.

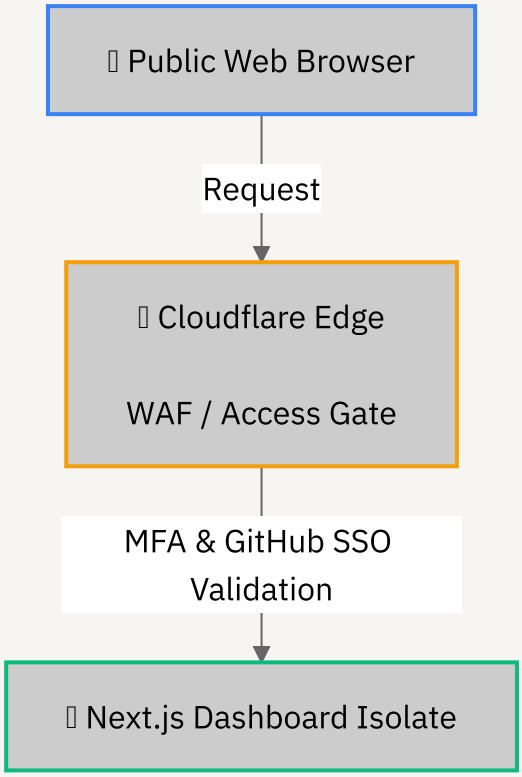
Recommended layout:

1. **Separate hostnames** — `gateway.example.com` for TradingView® `/webhook` (IP allowlist + body apiKey); `mgmt.example.com` for `/v1/*` only.
2. **Access application** on `mgmt.example.com` (SSO for browsers; **service token** for CLI/TUI).
3. **Application Bearer** (`HOOX_API_TOKEN` ↔ `OPERATOR_API_KEY`) enforced by `requireOperatorAuth`.
4. **Optional** — Cloudflare Tunnel so the origin is not on the public internet; **Enterprise** — API Shield mTLS client certificates for device-bound identity.

Do **not** put long-lived full-account `CLOUDFLARE_API_TOKEN` values into TUI debug logs or world-readable config files. Prefer scoped API tokens for deploy vs read-only monitor; `~/.hoox/config.json` is written with mode `0600`.

For tunnel vs Access decision tables, probe expectations, and `hoox tunnel check`, see Private ingress.

The Zero Trust Protective Boundary



- **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® 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® 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.

PRIVATE INGRESS (ACCESS + TUNNEL)

How to protect the Hoox operator management plane with Cloudflare Access and optional Cloudflare Tunnel.

PRIVATE INGRESS FOR OPERATORS

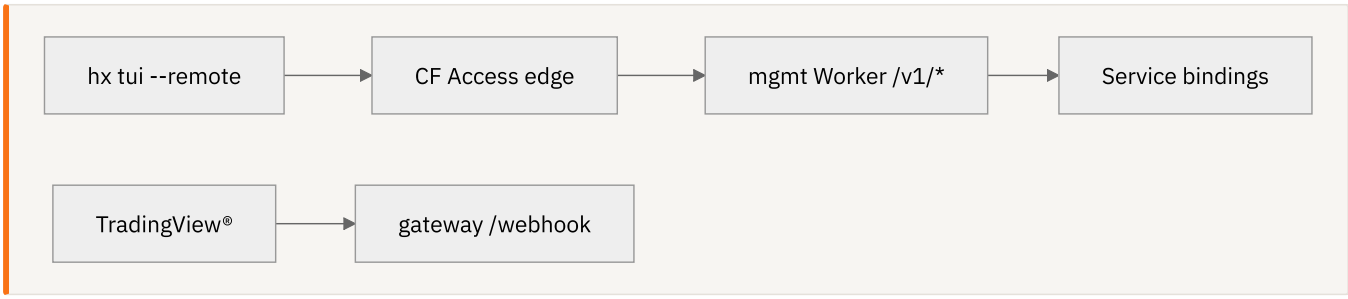
The **operator management plane** (`/v1/*` — workers list, SSE streams, health probe) should not sit open on the public internet the way TradingView® webhooks do.

Surface	Hostname (recommended)	Auth
Trading signals	`gateway.example.com`	Body `apiKey` + IP allowlist
Operator TUI / CLI	`mgmt.example.com`	Access (+ Bearer `OPERATOR_API_KEY`)
Dashboard UI	`app.example.com`	Access / SSO (existing)

When to use Access vs Tunnel vs mTLS

Approach	Use when	Open core
Cloudflare Access (service token)	Default private operator path; free tier friendly	Yes
Cloudflare Tunnel (`cloudflared`)	Origin must not be public; self-hosted; strict network isolation	Docs + `hoox tunnel check`
API Shield mTLS	Device-bound client certs / compliance	Enterprise

Recommended default: Access on `mgmt.example.com` + Bearer `HOOX_API_TOKEN` ↔ Worker `OPERATOR_API_KEY` .
Tunnel is optional defense-in-depth (or required for private origins).
mTLS stacks on top for Enterprise.



Quick setup (Access only)

1. Map `mgmt.example.com` to the Worker that serves `/v1/*` (often the same script as the gateway **only if** path policies differ — prefer a separate hostname).
2. Zero Trust → Access → Applications → Self-hosted → `mgmt.example.com` .
3. Create a **service token** for automation; put values in the laptop env:

```
```

4. Set the application Bearer on the Worker and client:

```
wrangler secret put OPERATOR_API_KEY
```

5. Verify:

```
hoox doctor --security --api-url https://mgmt.example.com
hoox tunnel check --api-url https://mgmt.example.com --probe
hoox tui --remote --api-url https://mgmt.example.com
```

Expected probe results

Caller	Healthy outcome
Anonymous `GET /v1/health`	`302` Access redirect or `401` (not `200`)
Authed (Bearer + Access headers)	`200` JSON `{ plane: "operator", … }`

Optional: Cloudflare Tunnel

Use a tunnel when the management origin must not be reachable on a public Worker URL, or when self-hosting behind a firewall.

1. Install CLI: cloudflared downloads
2. `cloudflared tunnel create hoox-mgmt`
3. Route DNS `mgmt.example.com` → tunnel
4. Run the connector with a config that points at your origin (Worker via private path, or local `server.js`)
5. Put **Access** on the tunnel hostname
6. `hoox tunnel check` confirms the binary; `--probe` exercises `/v1/health`

```
hoox tunnel check
hoox tunnel check --api-url https://mgmt.example.com --probe
```

Tunnel does **not** replace `CLOUDFLARE_API_TOKEN` for deploy/secrets (control plane). It only protects **your** management API.

CLI surface

Command	Purpose
`hoox doctor --security`	Hygiene + optional probes
`hoox tunnel check`	cloudflared detect + optional probes
`hoox config transport`	Show resolved transport profile
`hoox config transport set access`	Persist preference to `~/.hoox/config.json`
`hoox tui --remote`	Fail-closed launch (Bearer or Access env)

Env reference:

Variable	Role
`HOOX_API_URL`	Management base URL
`HOOX_API_TOKEN`	Bearer → `OPERATOR_API_KEY`
`HOOX_TRANSPORT`	`public` | `access` | `mtls` | `tunnel`
`CF_ACCESS_CLIENT_ID` / `SECRET`	Access service token

Align dashboard + management Access

Prefer one Access **policy group** (emails / GitHub team) for:

- Dashboard hostname (human SSO)
- Management hostname (service token for `hx`, SSO for browsers if any)

Device posture and SCIM remain Enterprise policy layers.

Related

- Zero Trust
- TUI architecture
- Security overview
- Enterprise mTLS: `docs/enterprise/security-compliance.mdx`

45 · 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** — Fail-closed middleware, timing-safe comparisons, mesh controls
2. **Security Testing** — Auth bypass tests, fuzz tests, header verification
3. **CI/CD Scanning** — Dependency auditing, secret detection, CodeQL analysis

Gateway mesh hardening (v0.13+)

Applied on the public webhook path **before** exchange dispatch:

Control	Behavior
Two-phase idempotency	`IdempotencyStore`: **reserve** (pending) → dispatch → **commit** on true success / **release** on soft-fail or error. Missing DO → **fail-closed** `503`.
Atomic rate limit	`RateLimiterStore` DO (`RATE_LIMITER`, migration `v2`); KV/memory fallback. Default 10 req / 60s → `429`.
Body cap	**64 KiB** hard limit on webhook bodies.
Timing-safe webhook auth	`timingSafeEqual` / async variant for `apiKey`.
Notify chat allowlist	`TELEGRAM_ALLOWED_CHAT_IDS` / `AUTHORIZED_CHAT_IDS` — unlisted `chatId` rejected (fail-closed when configured).
Named D1 RPCs	Hot paths use fixed SQL templates (`list-signals`, `list-system-logs`, `list-open-positions`, insert/upsert RPCs) instead of free-form `/query`.
safeWaitUntil	Mesh-wide helper so background D1/analytics/notify work is not dropped on isolate exit (including DO lifecycle on trade-worker).
REST-only orders	Exchange order placement stays REST; WS DO is not the order path.

Deploy notes (DO migrations `v1` / `v2`, allowlist secrets): `workers/hoox-worker/DEPLOY.md`. Concepts: Idempotency · worker profile: hoox gateway.

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-worker/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 + kill switch + DO rate limit + notify chat allowlist (`TELEGRAM_ALLOWED_CHAT_IDS`)	`POST /` , `POST /webhook`	`GET /health`
hoox operator plane	Bearer `OPERATOR_API_KEY` (`requireOperatorAuth`)	`GET /v1/health` , `/v1/workers` , SSE `/v1/trades	logs/stream`
CLI/TUI operator	`HOOX_API_TOKEN` + optional Access service-token headers	Client transport profile (`HOOX_TRANSPORT`)	Fail-closed remote launch without credentials
d1-worker	`X-Internal-Auth-Key` (scoped read/write)	`POST /query` , `POST /batch` , `GET /api/*`	`GET /health`
trade-worker	`X-Internal-Auth-Key`	`POST /webhook` , `POST /process` , `GET /api/signals` , `GET /report`	`GET /health`
agent-worker	`X-Internal-Auth-Key`	All `/agent/*` (including chat, vision, risk-override)	`GET /` , `GET /health`
analytics-worker	`X-Internal-Auth-Key`	All `POST /track/*`	`GET /health`
telegram-worker	`X-Internal-Auth-Key` + outbound chat allowlist (`AUTHORIZED_CHAT_IDS` / `TG_CHAT_ID_BINDING`)	`POST /alert` , `POST /process`	`GET /health` , `POST /webhook` (Telegram secret + inbound chat allowlist)
email-worker	Mailgun HMAC + timestamp window	—	`GET /health` , `POST /webhook` (HMAC + ±15m replay window)
report-worker	`X-Internal-Auth-Key`	`GET /report` , generate/delivery paths	`GET /health`
web3-wallet-worker	`X-Internal-Auth-Key`	Transfer / balance / approve / swap / config	`GET /health`
pyne-worker	`X-API-Key` (`API_KEY` , not mesh internal key)	`/run` , `/ingest` , scripts, cron, feed when key set	`GET /health` ; open if `API_KEY` unset (dev only)
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.

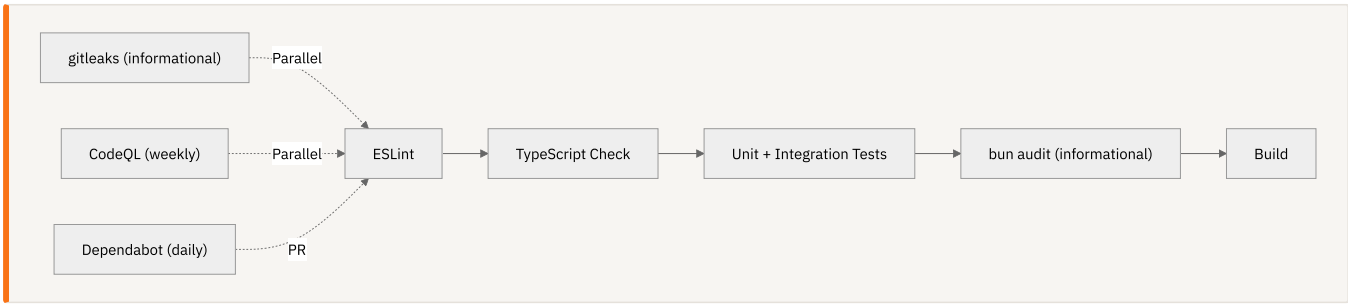
Dependabot

File: `.github/dependabot.yml`

Daily npm and GitHub Actions dependency checks with automatic PR creation.

---

6. CI Pipeline Security Flow



---

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	Agent cron cycle simulation (1–1440 min schedule)
`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

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 (`@hoox-sh/hoox-shared`) and TUI components (`packages/tui`) instantly without network downloads or pre-compilation overhead:

```
hoox/ (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 services (`src/services/cloudflare/`, `src/services/*`)

Translates command intentions (like `hoox infra d1 create`) into wrangler subprocess calls or Cloudflare REST API requests. Domain services (setup, secrets, DB, repair, perf) live under `src/services/`.

C. Shared config & path resolution (`@hoox-sh/hoox-shared`)

Workspace configuration (`wrangler.jsonc`), path layout (`$HOME/.hoox`, monorepo detection), and operator transport live in the shared package rather than a CLI-local `src/core/`.

D. Workspace context (startup)

Before commands run, `ensureWorkspaceContext()` (`packages/cli/src/services/workspace/`) resolves the monorepo and switches the process into it:

Priority	Source	Notes
1	`HOOX_REPO`	Explicit override
2	Walk up from `cwd`	Local clone markers
3	`~/.hoox/config/monorepo.json`	Last discovered checkout
4	`~/.hoox/repo`	Managed global clone (`doctor --fix-runtime`)

Markers (`isHooxSetupRoot`): `packages/cli/package.json` **and** one of `wrangler.jsonc` / `wrangler.jsonc.example` / `workers/` / `.gitmodules` .

On success the CLI:

1. writes/refreshes `monorepo.json` when source is `cwd` , `env` , or `remembered`
2. sets session `HOOX_REPO` for child processes
3. `process.chdir(root)` when not already there (notice on stderr unless silent/json/quiet)

This is why `hx check setup` works after you leave the monorepo directory.

□ 2. Declarative Config Mapping & Validation

To prevent configuration drift, the CLI enforces strict type validation on `wrangler.jsonc` files:

1. **Config validation**: Built-in parsers validate workspace `wrangler.jsonc` (worker enablement, paths, secrets, vars) via the CLI config/schema services — there is no `src/core/types.ts` path; operator config also lives in `@hoox-sh/hoox-shared` .
2. **Setup Verification (`hoox check setup`)**: Compares active workspace profiles against expected shape, audits environment variable 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

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 — **modern-minimal** aesthetic (zinc/slate text, single indigo-400 accent, de-saturated status colors). 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`) — Linear Rail

`renderBanner()` / `animateBanner()` run when the CLI opens the interactive menu (no arguments, initialized workspace). The version is read at module init by walking up from `import.meta.url` looking for the `@hoox-sh/hoox-cli` `package.json` — this works in source, bundle, and global install layouts.

Default is Linear Rail — compact `◆ H · O · O · X` with tagline `Cloudflare Workers Platform · vX.Y.Z` (Cloudflare® Workers Platform in prose). On a TTY (and not `NO_COLOR` / CI / `TERM=dumb`), the banner assembles → pulses → settles; otherwise a single static frame is printed. Variants: `logo` / `minimal` (aliases of Linear Rail), `legacy` (block letters), `horizon`, and `signal`. Compact one-liner: `◆ Hoox · v…` via `renderCompactBanner()`.

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.

---

5. Secrets modes (`hoox secrets`)

Top-level `hoox secrets` (alias of `hoox config secrets`) manages Worker secrets declared in workspace `wrangler.jsonc`:

Mode	Command	Behavior
System / mesh only	`hoox secrets sync --system`	Syncs internal auth + gateway keys only (`INTERNAL_KEY_BINDING`, `WEBHOOK_API_KEY_BINDING`, session keys, …). Skips exchange keys, bot tokens, and other integration secrets. Alias: `--required`. Prefer after `hoox keys generate` / key rotation.
Full sync	`hoox secrets sync` / `hoox secrets sync <worker>`	Puts all declared secrets present in local `.dev.vars`. Reports **synced** / **skipped** (placeholders or missing) / **failed**.
Single secret	`hoox secrets set <worker> <name>`	Writes `.dev.vars` and `wrangler secret put` for one name.
List / delete	`hoox secrets list [worker]`, `hoox secrets delete …`	Inventory or remove.

`hoox check setup` only fails remote secret checks when declared names are **missing on Cloudflare®** (with `hoox secrets sync` hints) — healthy local values no longer produce noisy warnings.

---

6. Setup gates (`hoox onboard` / `hoox setup`)

Gate	Behavior
Init incomplete	Onboard does **not** run setup unless `wrangler.jsonc` exists after step 1 (cancel / risk-decline stay fail-closed).
Cloudflare® auth	Setup and onboard step 2 require `wrangler whoami` / token env; fail early with login hints.
Workers gate	After submodule clone, setup aborts if worker trees are still missing.
`--skip-keys`	Loads mesh keys from `.keys/setup.env` so secrets can still push without regenerating keys.

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.
- **Setup & Operations** — Deploy order, health probes, runbooks.

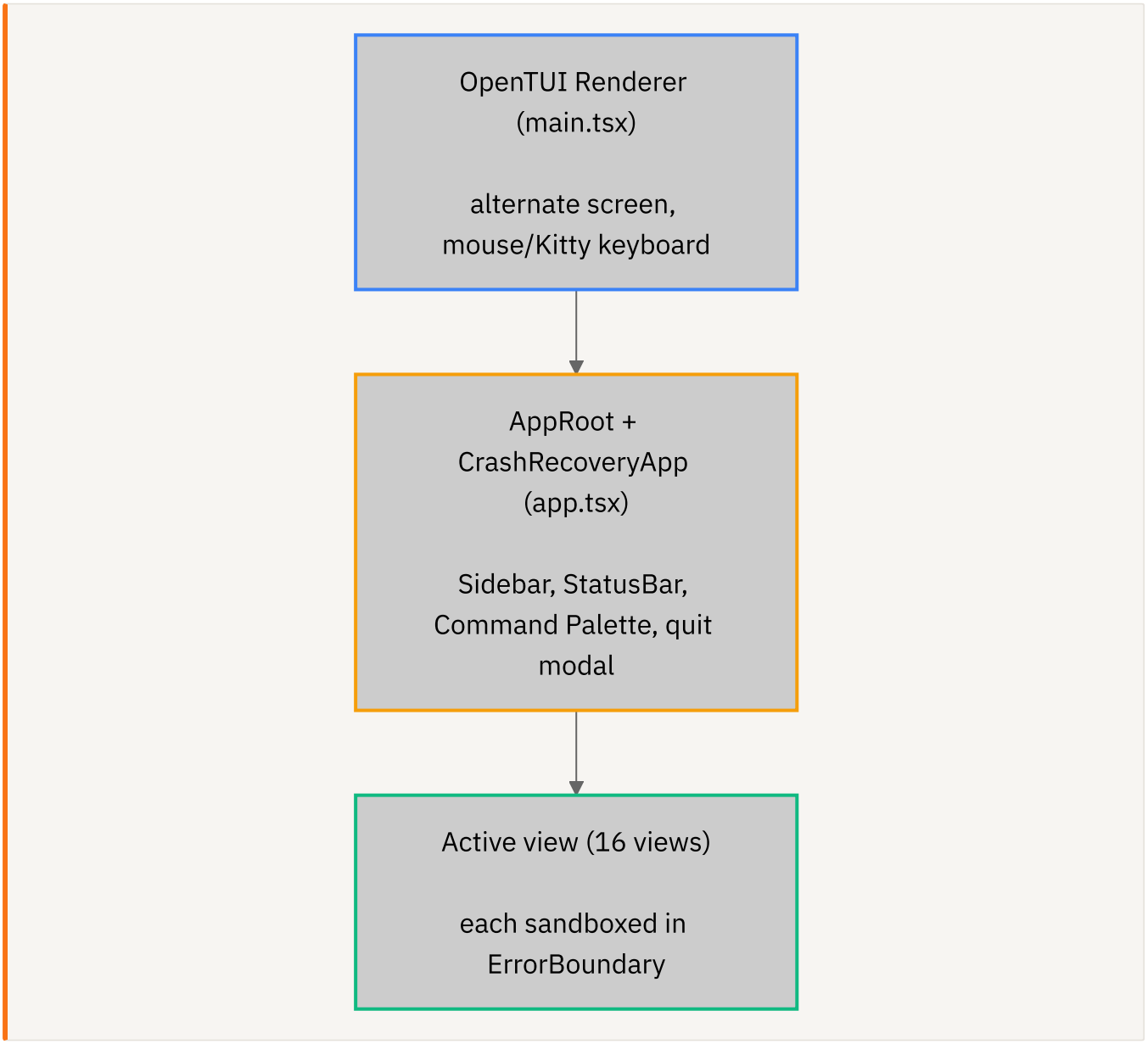
TUI CODE ARCHITECTURE

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` (`@hoox-sh/hoox-tui`)
Entry: `src/main.tsx` · **Root:** `src/app.tsx` · **CLI launcher:** `hoox tui`

Architectural Blueprint



Directory map

```
packages/tui/src/
├── main.tsx                # createCliRenderer (TUI_FPS / TUI_MOUSE env)
├── app.tsx                 # shell, session, SSE kickoff (views via view-registry)
├── components/
│   ├── views/              # 16 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 `@hoox-sh/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 (dark-only in TUI), refresh interval, notifications, shortcuts

Session persistence: `$HOME/.hoox/.tui-state/session.json` via shared `restoreSession` / `saveSession`. All **16** `ViewId` values are valid for restore; unknown IDs fall back to `dashboard`.

Navigation registry & colors

- View factories, sidebar labels, keyboard shortcuts, and palette **view** commands are defined in `packages/tui/src/view-registry.tsx`.
- Semantic status colors (`ConnectionStatusColor`, `WorkerStatusColor`, `LogLevelColor`, `AlertSeverityColor`) live in `@hoox-sh/hoox-shared` (`packages/shared/src/colors.ts`). Do not invent local status→hex maps in views.
- Shared chrome: `ViewHeader` + `Panel` (cool focus border). Settings uses dark-only theme (light is not implemented) and shows mode/host/transport/auth via `getSettingsConnectionSnapshot()` — never secret values.

Test doubles (no polluting `mock.module`)

Installed once from `packages/tui/src/test-setup.ts` (preload):

Double	Module	Control
CLI	`cli-bridge-test-double.ts`	`cliBridgeDouble` / `resetCliBridgeDouble()` (includes `pyneHealthCheck`)
HTTP + SSE	`network-test-double.ts`	`setMockApiData` / `setMockApiFailure` / `emitSseEvent`

View tests must **not** call `mock.module` for `cli-bridge`, shared stores, `api-client`, or `sse`. Override methods on the doubles instead. Edge topology fixtures use `HOOX_GRAPH_METADATA_PATH` (temp file) rather than mocking `fs`.

View registry (16)

Must stay aligned with `ViewId` in `packages/shared/src/types.ts`, sidebar items, `view-registry.tsx`, and command palette entries:

ViewId	Component	Primary data source
`dashboard`	`DashboardView`	`fetchWorkers`, `monitorStatus`, `checkFix`, `agentHealthCheck`, `pyneHealthCheck`, 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 + connection snapshot + `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`
`worker-settings`	`WorkerSettingsView`	dashboard.jsonc / worker settings via CLI

Local vs remote mode

`hoox tui` resolves an API base URL and forwards it to the child process:

Launch	Mode	`HOOX_API_URL`	Status bar
`hoox tui`	`local`	`HOOX_API_URL` env or `http://localhost:8787`	`[LOCAL] localhost:8787`
`hoox tui --remote`	`remote`	`resolveGatewayUrl()` (`HOOX_GATEWAY_URL` / CF account + wrangler subdomain)	`[REMOTE] <gateway-host>`
`hoox tui --api-url <url>`	`remote`	explicit URL (strips trailing `/`; overrides `--remote`)	`[REMOTE] <host>`

Env vars set on the TUI process: `HOOX_API_URL`, `HOOX_TUI_MODE` (`local` | `remote`), optional `HOOX_API_TOKEN`. Shared `api-client` / `sse` read `HOOX_API_URL` + `HOOX_API_TOKEN` at import time.

Security posture (operator plane)

Layer	Status
Client Bearer (`HOOX_API_TOKEN`)	Sent as `Authorization: Bearer …` by TUI HTTP/SSE via operator transport profile
Access service token headers	Auto-attached when `CF_ACCESS_CLIENT_ID` + `CF_ACCESS_CLIENT_SECRET` are set
Transport profile	`HOOX_TRANSPORT=public|access|mtls|tunnel` (mtls/tunnel reserved; headers like public until certs land)
Fail-closed remote launch	**Yes** — remote requires Bearer, Access env, or `--allow-insecure`
Server management routes	**`/v1/*` on gateway** with `requireOperatorAuth` (`OPERATOR_API_KEY` preferred, else `INTERNAL_API_KEY`)
Public liveness	`GET /health` remains unauthenticated (no secrets)
Cloudflare Access / mTLS / tunnel edge	Put **Access** on a mgmt hostname; mTLS + tunnel are Enterprise / private-ingress follow-ups

Server secret: `wrangler secret put OPERATOR_API_KEY` (value must match client `HOOX_API_TOKEN`).

Client paths (TUI): `GET /v1/workers`, SSE `/v1/trades/stream`, `/v1/logs/stream`, auth probe `GET /v1/health`.

Prefer a **dedicated management hostname** (e.g. `mgmt.example.com`) separate from TradingView® `/webhook` ingress — see Zero Trust.

Auth token UX (remote)

Surface	Behavior
CLI launch	**Fails closed** without Bearer or Access env (unless `--allow-insecure`)
CLI flags	`--token <value>` sets `HOOX_API_TOKEN` for the child only (never logged)
Access env	`CF_ACCESS_CLIENT_ID` + `CF_ACCESS_CLIENT_SECRET` satisfy the launch gate
Escape hatch	`--allow-insecure` remote without credentials (not recommended)
Status bar	Remote without token → compact `AUTH?`; 401/403 → `AUTH!`
Toasts	Mode-aware connect / lost / reconnect; auth missing & auth failed messages
Startup	**REMOTE never uses CLI fallback** (gateway HTTP only); LOCAL may fall back to `cliBridge.monitorStatus()`
Config file	`~/.hoox/config.json` written `0600` (may hold `apiToken`)

Dev logging

File-backed only (never stdout — would corrupt the alternate screen):

```
hoox tui --debug
# or: HOOX_DEBUG=1 / TUI_DEBUG=1
# → $HOME/.hoox/.tui-state/debug.log  (JSON lines; secret-like keys + Bearer/env patterns redacte
```

Wired at startup (`main.tsx`), connection path (`app.tsx`), and CLI bridge execs.

□ Dual-channel + SSE

Startup sequence (`AppRoot`)

1. Restore session → set view / sidebar
2. `fetchWorkers()` (HTTP) → on failure only, `cliBridge.monitorStatus()` (CLI fallback)
3. Fire-and-forget `streamTrades()` + `streamLogs()` (SSE; silent if API down). Store retains abort handles; `stopStreams()` tears them down (re-subscribe replaces prior subscriptions).

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`, `pyneHealthCheck`.

Dashboard surfaces **agent** health (`agentHealthCheck`) and **PYNE** edge health (`pyneHealthCheck` → `PyneHealthSection`).
`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 **16** view exports are present as function nodes under `packages/tui:…/views/…` (plus dashboard subpanels).
- Regenerate after large refactors: `bun run graph` (~25s). Metadata should include every worker under `workers/*` .

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**

WORKERS PATTERN CONSOLIDATION

WORKERS PATTERN CONSOLIDATION - COMPLETE REFERENCE

Overview

This document describes the pattern consolidation refactorings applied to the hoox 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 configurable cron routine (1–1440 min)
- `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 "@hoox-sh/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 "@hoox-sh/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%	80%+ core (shared ~98%)	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

HOOX ENTERPRISE

Documentation hub for the institutional / Enterprise version of HOOX on Cloudflare.

HOOX ENTERPRISE

> **This is the commercial / Enterprise extension layer.**
>
> HOOX follows an **Open Core** model. The core architecture, most code, and documentation are open source. This section describes the **commercial Enterprise layer** built on top of the open core.
>
> See:

- OPEN_CORE.md
- OPEN_CORE_FEATURE_SPLIT.md

This section contains **public architecture documentation** for the institutional / paid Enterprise version of HOOX (the commercial extension layer built on the open core). It is designed to run on **Cloudflare Enterprise**.

Note: No proprietary Enterprise source code is included in this public repository.

Documents

- Architecture — Overall topology, bleeding-edge primitives, and evolution from retail HOOX.
- Multi-Tenancy — Workers for Platforms, Dispatch + User Workers, isolation model.
- Observability & Audit — Logpush, Traces, Tail Workers, immutable R2 audit.
- Security & Compliance — Extended 5-layer model with Bot Management, API Shield, Zero Trust, AI Guardrails.

Key Differentiators (vs Retail HOOX)

- Workers for Platforms for true multi-tenancy / platform use
- Cloudflare Workflows for durable, long-running trade & compliance processes
- Full Logpush + R2 for regulatory-grade audit
- Enterprise security features (Bot Management, API Shield, etc.)
- Real-time persistent connections via hibernatable WebSockets in DOs
- Higher limits, custom SLAs, AI Gateway at scale
- Data residency and advanced compliance controls

See the approved architecture plan (internal `plan.md`) for phasing and rationale.

All designs build directly on the existing high-quality retail codebase (Service Bindings, DOs, Smart Placement, shared middleware, etc.).

HOOX ENTERPRISE ARCHITECTURE

Bleeding-edge architecture for HOOX on Cloudflare Enterprise, leveraging Workers for Platforms, Workflows, Logpush, AI Gateway, advanced security, and more for institutional-grade algorithmic trading.

HOOX ENTERPRISE ARCHITECTURE

> **Important – Open Core Model**
>
> This document describes the **commercial Enterprise extension layer**.
>
> HOOX follows an **Open Core** model:
>
> - The core architecture, most workers, `packages/shared`, CLI, TUI, and the majority of documentation are **open source**.
> - Advanced multi-tenancy, proprietary features, full compliance stacks, hosted SaaS platform, and certain advanced implementations live in a **separate closed-source commercial repository**.
>
> See:
>
> - OPEN_CORE.md
> - OPEN_CORE_FEATURE_SPLIT.md
>
> The documents in this folder describe **how the commercial layer is built on top of the open core**. They are published publicly for transparency and to support the academic paper.

Status: Proposed / In Design
Target: Cloudflare Enterprise accounts (or hosted SaaS on Enterprise)
Goal: Institutional-grade, multi-tenant, highly observable, durable, secure, real-time edge-native trading platform built as an extension on the open core.

This document expands the approved plan into concrete architecture for the commercial layer, leveraging the full power of Cloudflare Enterprise and 2025-2026 bleeding-edge primitives.

Vision & Principles

- **Everything at the Edge, Amplified**: Preserve the current zero-server, Service Binding, Smart Placement, DO-based model. Enterprise unlocks scale, isolation, compliance, and durability without introducing traditional infra.
- **Multi-Tenancy First**: Support hosted SaaS (funds, prop shops, platforms) or dedicated high-volume Enterprise deployments with strong isolation.
- **Durable + Event-Driven**: Move from cron-heavy to Workflows + real-time WS-driven.
- **Audit & Compliance by Default**: Every action logged immutably.
- **Bleeding Edge but Practical**: Use features that are GA or recently stabilized (WfP dashboard improvements, synchronous User Worker deploys, Workflows enhancements, Logpush one-click + custom fields, etc.).
- **Evolutionary**: Existing single-tenant HOOX remains for retail/self-hosted. Enterprise is additive/parallel.

Current HOOX (retail PoC) already uses advanced primitives (Service Bindings ~0.4ms, SQLite DOs for idempotency, Smart Placement, Queues, D1/R2/Vectorize, Browser Rendering). Enterprise takes this to the next level for 10x-100x volume, regulatory needs, and platform offerings.

Core Levers (Enterprise / Paid Unlocks)

1. **Workers for Platforms (WfP)**
   - Dynamic Dispatch Worker routes to isolated "User Workers" (per tenant, per strategy, per fund).
   - Namespaces, tags for metering/billing/isolation.
   - User Workers can have their own bindings, DOs, Queues (isolated).
   - Recent improvements: Dashboard support for namespaces/dispatch/templates/tags (Dec 2025), static assets support, synchronous first-time uploads (Feb 2025) so deploys are immediately routable.
   - Perfect for "bring your own strategy" or multi-fund platforms.
   - Enterprise: higher scale, custom limits, better support for customer code execution.
2. **Workflows (Durable Execution)**
   - Long-running (hours/days), stateful, automatically retrying, pause-for-external-events.
   - Step-level persistence, input/output gates.
   - Integrates with Agents (real-time) and existing DOs.
   - Use cases:
     - Full trade lifecycle (validate → risk → execute → reconcile → report).
     - Kill switch + position flattening sequences that must complete atomically.
     - Daily compliance/report generation with human approval gates.
     - Post-trade reconciliation across exchanges/D1/R2.
   - Recent enhancements: step context (retry count), dynamic workflows, AgentWorkflow integration.
   - Enterprise: higher execution volume, longer durations, more steps.
3. **Observability & Audit (Logpush + Traces + Observability suite)**
   - Full Workers Trace Events, HTTP, WAF, Queue, DO, etc. pushed to R2 (or external SIEM like SentinelOne).
   - One-click R2 setup, custom fields (raw/transformed headers), daily partitions.
   - Workers Logs (dashboard), Real-time logs, Tail Workers for sampling/filtering/enrichment.
   - End-to-end traces across Service Bindings, DOs, Queues, Workflows.
   - Analytics Engine remains for high-cardinality metrics; Logpush for compliance/audit.
   - Enterprise: higher log volume, longer retention, dedicated support, integration options.
4. **Enterprise Security Stack**
   - **Bot Management** (Enterprise): bot scores, verified bots, corporate proxy detection, JA4/JA3, JS detection. Protect signal sources (TradingView webhooks, Telegram, email).
   - **API Shield**: schema validation, JWT validation, mTLS for clients (dashboard or partner APIs), sequence mitigation.
   - **Advanced WAF + Rulesets**: 100+ rate limit rules, custom high-cardinality rules, managed rules for trading threats.
   - **Zero Trust / Access**: Enforce identity for dashboard, CLI mgmt, internal tools. SCIM, device posture, etc.

- **mTLS / Mutual TLS**: Between services or for outbound.
- **Data Loss Prevention / Gateway** if needed for sensitive configs.
- Existing 5-layer model (WAF → Gateway auth → Service Bindings → Internal key → DO mutex) is extended with Ent controls.

5. **Real-time & Persistent Connections**
   - Hibernatable WebSockets in Durable Objects (keep connections alive cheaply while DO sleeps).
   - Evolve `ExchangeConnectionManager` DO to maintain persistent WS to Binance/Bybit/etc. for live fills, partials, order books.
   - Drive risk management, notifications, and analytics from real events instead of (or alongside) cron-only polling.
   - Combined with Workflows for durable handling of streams.
6. **Scale, Limits & Performance (Enterprise Unlocks)**
   - Significantly higher CPU time, memory, subrequests, concurrent connections, Worker size, # of Workers/crons per account.
   - Custom limit increases via account team.
   - Higher D1 storage/reads/writes, R2 operations, Queue throughput, Vectorize dimensions/index size.
   - DO: higher concurrency per namespace, more DOs, better SQLite performance at scale.
   - Smart Placement + Enterprise routing hints for even lower jitter to exchange regions.
   - Hyperdrive for accelerating any external data sources (if hybrid Postgres/etc. used for large historical data).
7. **AI at Institutional Scale**
   - **AI Gateway**: logging, caching, rate limiting, fallbacks, cost control, prompt sanitization in front of Workers AI or external providers (OpenAI, etc.).
   - Workers AI + Vectorize at Enterprise volumes (large strategy corpus, market embeddings, RAG for compliance docs or historical trades).
   - Integration with Workflows/Agents for sophisticated risk agents, anomaly detection, natural language strategy config.
   - Existing simple RAG in telegram-worker becomes production-grade with Gateway + fallbacks.
8. **Storage & Data Sovereignty**
   - R2: Event Notifications (trigger Workflows on new trade logs/reports), jurisdiction controls (data residency), larger scale, lifecycle policies for immutable audit logs.
   - D1: Enterprise storage/throughput, point-in-time recovery, read replicas (if/when available), larger DBs per tenant.
   - KV + DO SQLite for low-latency config/state.
   - Immutable audit: Logpush + R2 WORM-like + cryptographic hashes/signatures in D1.
9. **Other Bleeding Edge**
   - Browser Rendering at scale (more concurrent renders for compliance reports).
   - Builds (CI/CD for User Workers in WfP).
   - Email Workers / advanced routing for signal ingestion.
   - Magic Transit / Spectrum if private connectivity to exchanges or on-prem required (rare for CEX but useful for some prop shops).

- Queues + Workflows for reliable backpressure and exactly-once-ish semantics at high volume.

High-Level Target Topology (Enterprise)

```
Public Ingress (WAF + Bot Mgmt + API Shield)
          |
          ▼
Dispatch Worker (Workers for Platforms)  ← Tenant/Strategy routing + auth (Zero Trust)
          |
     ┌────┴─────┐
     ▼          ▼
User Workers / System Workers (isolated per tenant/strategy)
     |
     ├──► Service Bindings (still <1µs, now tenant-scoped)
     |
     ├──► Durable Objects (per-tenant DO namespaces)
     |      - Idempotency (SQLite)
     |      - ExchangeConnectionManager (Hibernatable WS for live data)
     |      - Risk state / position managers
     |
     ├──► Workflows (durable lifecycles)
     |      - Trade execution + reconciliation
     |      - Kill-switch sequences
     |      - Daily compliance reports
     |
     ├──► Queues (higher throughput, with Workflows consumers)
     |
     └──► Storage (tenant-partitioned)
            - D1 (per-tenant or sharded)
            - R2 (immutable logs + jurisdiction)
            - Vectorize (filtered by tenant)
            - Analytics Engine (tagged)

Observability Layer (always on)
- Workers Traces (cross everything)
- Logpush → R2 (full audit, custom fields)
- Tail Workers (sampling, enrichment)
- AI Gateway (for any AI calls)
- Dashboard (Zero Trust protected, multi-tenant views)
```

Existing workers (trade-worker, agent-worker, hoox, etc.) become templates or are refactored into dispatchable components + system services.

Key New / Evolved Components

1. Dispatch + User Workers (WfP)

- Central Dispatch Worker inspects request (tenant id from subdomain, header, or JWT), looks up namespace, dispatches to User Worker.
- Each User Worker can contain strategy-specific logic or a full lightweight HOOX instance.
- Static assets support for per-tenant dashboards/UI if needed.
- Tagging for billing/quotas.

2. Workflows Definitions

Examples:

- `TradeLifecycleWorkflow`: validate → pre-risk → execute (with retries) → persist → post-risk → notify → reconcile.
- `ReconciliationWorkflow`: runs daily, compares D1 vs exchange reports (via Browser Rendering or API), flags discrepancies.
- `ComplianceReportWorkflow`: gathers logs (from R2), generates PDF (Browser), signs, uploads, notifies.

Workflows can call existing services via Service Bindings and pause for external events (e.g., manual approval).

3. Enhanced DOs

- `IdempotencyStore` generalized with tenant/strategy prefix.
- `ExchangeConnectionManager` uses `hibernatableWebSockets` + alarms for cleanup.
- New DOs for per-tenant risk engines, session state, etc.
- RPC methods + SQLite for best DX/performance.

4. Observability Pipeline

- Every Worker/Workflow/DO emits structured events.
- Logpush jobs (one-click or API) to R2 with `output_options` for relevant fields.
- Tail Worker example for redacting secrets or adding tenant context.
- Full traces for root-causing a missed fill across 5+ Workers.

5. Security Hardening

- Bot score checks early in gateway/dispatch.
- API Shield schemas for all public + internal-ish endpoints.
- Zero Trust policies for dashboard + mgmt APIs.
- Per-tenant secrets (using Workers secrets or Secret Store at scale).
- mTLS where Services talk to trusted partners.

Tenant Isolation Model

- **Namespace per tenant** (or per "fund" / "book").
- DOs, Queues, KV, R2 paths, D1 databases prefixed or bound per namespace.
- User Workers run in isolated isolates with their own limits.
- Billing/quotas via tags + custom dashboards (Enterprise).
- Cross-tenant leakage prevented at Dispatch + binding auth layers (evolved `requireInternalAuth` with tenant claims).

Migration & Coexistence

- Retail HOOX stays as-is (free/paid friendly).
- Enterprise customers get:
  - Dedicated Enterprise CF account + templates, or

    - Onboarded into hosted WfP platform.
- Existing code in `packages/shared`, workers, etc. is reused as libraries/templates.
- Gradual: start with WfP + Logpush + one Workflow, then add real-time WS, Bot Mgmt, etc.

Open Decisions (from plan review)

- Primary target: Hosted SaaS (heavy WfP) vs Power-user self-hosted on Ent account vs both?
- Migration strategy for existing users?
- Specific compliance jurisdictions / external SIEM requirements?
- Depth of "user code" execution (full arbitrary strategies vs safe DSL / config-driven)?

Next Steps / Implementation Order

1. Create `docs/devops/enterprise/` docs (this + security, multi-tenancy, observability).
2. Prototype Dispatch Worker + minimal User Worker + one Workflow (trade-lifecycle).
3. Extend `@hoox-sh/hoox-shared` with tenant context, Workflow client, AI Gateway wrapper.
4. Add Logpush config examples + a Tail Worker.
5. Update root architecture docs, ADRs, and papers (new section or companion Enterprise paper).
6. wrangler templates in `examples/enterprise/`.
7. (Later) Full implementation, tests, `bun run graph` validation, academic paper updates.

This architecture keeps HOOX's unique edge-native soul while making it suitable for serious institutional use with the full power of Cloudflare Enterprise.

References (use latest docs):

- Workers for Platforms
- Workflows
- Logpush / Observability
- AI Gateway
- Durable Objects (SQLite, WS, Alarms, RPC)
- Enterprise limits & features (higher everything + dedicated support)
- Zero Trust, Bot Management, API Shield

See the approved plan.md for the detailed "why" and phasing.

MULTI-TENANCY WITH WORKERS FOR PLATFORMS

How HOOX Enterprise achieves strong multi-tenant isolation using Cloudflare Workers for Platforms, namespaces, dispatch, and per-tenant Durable Objects / bindings.

MULTI-TENANCY WITH WORKERS FOR PLATFORMS

> **This document describes commercial Enterprise capabilities.** Basic Workers for Platforms concepts and examples are part of the open core. Full production multi-tenant SaaS platform, billing, and advanced isolation live in the closed-source Enterprise layer.
>
> See OPEN_CORE_FEATURE_SPLIT.md.

HOOX Enterprise uses **Workers for Platforms (WfP)** as the foundation for multi-tenancy in the commercial offering.

Why WfP?

- Isolated "User Workers" per tenant / fund / strategy book.
- Centralized **Dispatch Worker** performs routing + auth, then dispatches.
- Each User Worker gets its own (or shared-but-namespaced) bindings: DO namespaces, Queues, D1 (per-tenant or sharded), R2 paths, etc.
- Native support for tags (for metering/billing), namespaces, and (as of late 2025) dashboard management.
- Synchronous first deploy + static assets support.
- Perfect for "platform" or "hosted" HOOX where customers can even bring limited custom strategy logic.

Architecture

```
Incoming Request (subdomain, header, or JWT tenant claim)
          |
          ▼
Dispatch Worker (Enterprise account)
  - Resolve tenant from claim / subdomain / tag
  - Enforce Zero Trust / Bot Mgmt / API Shield
  - Lookup namespace + script
  - env.TENANT_DISPATCHER.dispatch(..., { tenantId, ... })
          |
          ▼
Tenant / Strategy "User Worker"
  - Runs in strong isolation
  - Own (or prefixed) DOs, Queues, secrets
  - Can still call system services via Service Bindings (with tenant-scoped auth)
          |
    Tenant-specific logic or full lightweight HOOX mesh
```

Dispatch Worker Pattern

```
// workers/dispatch/src/index.ts (Enterprise only)

  async fetch(request: Request, env: Env) {
    const tenantId = extractTenant(request); // subdomain, header, or JWT
    if (!tenantId) return new Response("Missing tenant", { status: 400 });

    // Optional: per-tenant rate limiting, bot score check, etc.
    const namespace = env.NAMESPACES.get(env.NAMESPACES.idFromName(tenantId));

    return namespace.dispatch(request, {
      // pass tenant context
      tenantId,
      // forward internal auth with tenant claim
    });
  }
};
```

User Workers are uploaded under the tenant's namespace/script tag.

Per-Tenant Isolation

- **Durable Objects**: Use `idFromName(\` tenant:${'{'}tenantId{'}'}:idempotency`)` or dedicated namespaces per tenant.
- **Queues**: One queue per tenant or heavily tagged.
- **D1**: One database per tenant (recommended for isolation + billing) or row-level with tenant_id + RLS-like checks.
- **R2**: Keys prefixed `tenants/${tenantId}/...` + separate buckets for strict residency.
- **KV**: Per-tenant or namespaced.
- **Secrets**: Per-script or using Secret Store with tenant scoping.
- **Vectorize**: Metadata filtering by tenant or separate indexes.

Existing `requireInternalAuth` is extended with tenant claims:

```
// in @hoox-shared

  const err = requireInternalAuth(req, env);
  if (err) return err;
  const claim = req.headers.get("X-Tenant-Id");
  if (claim !== tenantId) return new Response("Tenant mismatch", { status: 403 });
}
```

Workers for Platforms Specifics (2025-2026)

- Dashboard support for creating namespaces, dispatch templates, tags.
- Static assets can be attached directly to User Workers.
- First-time User Worker uploads are synchronous (200 means ready to dispatch).
- Tags for cost attribution and quota enforcement (Enterprise advantage).

Hosted SaaS vs Dedicated Enterprise

- **Hosted (SaaS)**: One Enterprise account runs the Dispatch + system workers. Customers get User Workers in isolated namespaces. Billing via tags + external system.
- **Dedicated (Self-hosted on Ent account)**: Customer gets their own Enterprise account + our templates + CLI that sets up WfP (or just higher limits + Logpush + Workflows without full multi-tenancy).

Both modes share the same core architecture.

Limits & Scaling Notes

Enterprise customers get:

- Much higher numbers of scripts / namespaces.
- Higher request volume and concurrency.
- Custom limit increases.
- Better support for customer-provided code (security review processes).

Related

- See architecture.mdx for overall topology.
- observability-audit.mdx for per-tenant log isolation.
- Existing `wrangler.jsonc` patterns will be extended with `dispatch` and namespace bindings in Enterprise templates.

ENTERPRISE SECURITY & COMPLIANCE

Extending the 5-layer model with Bot Management, API Shield, Zero Trust, mTLS, AI Guardrails, and immutable audit for HOOX Enterprise.

ENTERPRISE SECURITY & COMPLIANCE

> **Open Core vs Enterprise** The base 5-layer security model and many patterns are part of the open core. Advanced Enterprise-only security features (full Bot Management, API Shield at scale, SCIM, dedicated mTLS setups, etc.) are described in this commercial layer document.

HOOX's existing 5-layer security model is excellent. Enterprise adds several powerful layers that are either only available or significantly stronger on paid/Enterprise plans.

Extended Layers

Layer 0: Bot Management (Enterprise)

- Scores every request (`cf.botManagement.score`).
- Detects verified bots, corporate proxies, automated tooling.
- JA3/JA4 fingerprinting, JS challenge results.
- Early drop of suspicious TradingView/Telegram/email signal sources.
- Combined with WAF custom rules and rate limiting.

Example in Dispatch / Gateway:

```
const bot = request.cf?.botManagement;
if (bot && !bot.verifiedBot && bot.score < 30) {
  return new Response("Blocked", { status: 403 });
}
```

Layer 1: Advanced WAF + API Shield

- Enterprise WAF supports 100 rate limiting rules, complex expressions, high cardinality.
- **API Shield**:
  - JSON Schema validation on webhook and internal-ish endpoints.
  - JWT validation.
  - mTLS (client certificates) for dashboard or partner integrations.
  - Sequence / anomaly detection.

Layer 2: Zero Trust / Access

- Dashboard and management APIs protected by Cloudflare Access (SSO, device posture, WARP, etc.).
- Replaces or augments custom passkeys for human operators.
- SCIM for team provisioning in larger organizations.

- Gateway policies for outbound control if needed.

Layer 3: Internal + mTLS

- Existing `INTERNAL_KEY_BINDING` + `requireInternalAuth` remains.
- Add mTLS between Workers where possible (or use Service Binding + strong claims).
- Per-tenant claims propagated in auth headers.

Layer 4: Durable Object + Cryptographic Controls

- SQLite DOs for tamper-evident logs (append-only tables with hashes).
- Signatures on important events (fills, kill switches, config changes).
- Stored in D1 + R2 (dual write for durability).

Layer 5: AI Guardrails (new)

- **AI Gateway** in front of all LLM calls (risk agents, summaries, strategy parsing).
  - Prompt injection detection / sanitization.
  - Logging + caching + rate limits + fallbacks.
  - Cost control and audit of every AI decision.
- Existing prompt-sanitizer in agent-worker is promoted and expanded.

Immutable Audit (Compliance Core)

Every security-relevant event flows to Logpush → R2 (see observability-audit.mdx).

Key events:

- All webhook / signal ingress (with bot score, WAF decision)
- Auth decisions (success + failure)
- Idempotency DO outcomes
- Order placement + exchange responses (full)
- Risk actions (trailing stop, kill switch)
- Config / secret changes (via audit hooks)
- Workflow step outcomes

Retention: typically 7 years for financial services (configurable via R2 lifecycle).

Data Residency & Sovereignty

- R2 jurisdiction controls (e.g., EU, US, APAC only buckets).
- DO location hints + Worker placement for sensitive tenants.
- D1 database location choices.
- Optional: separate Enterprise accounts per major jurisdiction.

Kill Switch & Fail-Closed Enhancements

- Multiple kill switch sources (manual via Zero Trust dashboard, API with mTLS, automated via Workflow or AI anomaly).
- Every activation produces a signed, immutable record.
- Position flattening Workflow must run to completion (durable).

Secrets & Key Management

- Use Cloudflare Secret Store / per-script secrets.
- Rotate via CLI + audit event.
- For very high security: integrate with external KMS via Hyperdrive or mTLS if needed (rare).

Threat Model Updates (Enterprise)

See original threat model (papers/appendices/C-threat-model.tex and docs).

New / amplified threats addressed:

- Malicious or compromised tenant code (mitigated by WfP isolation + review gates).
- Supply-chain attacks on customer strategies (AI Gateway + sandboxing patterns).
- Regulatory "prove you didn't front-run" → full immutable trace + signatures.
- Nation-state / sophisticated bot attacks on signal sources → Bot Management + WAF.

Implementation Notes

- Bot Management and advanced WAF rules are configured at the zone/account level (Enterprise).
- API Shield schemas live with the Dispatch Worker and public ingress points.
- Zero Trust policies defined for `dashboard`, `cli` (via API tokens with context), and internal tools.
- All new Enterprise components must emit audit events using the shared `trackAnalytics` / audit path.

Related Documents

- Root security overview: `docs/devops/security/overview.mdx`
- Architecture: `docs/devops/enterprise/architecture.mdx`
- Multi-tenancy isolation guarantees: `multi-tenancy.mdx`
- Original 5-layer model in the academic paper.

ENTERPRISE OBSERVABILITY & AUDIT

Full auditability using Logpush, Workers Traces, Tail Workers, R2 immutable storage, and integration with Workflows for compliance in HOOX Enterprise.

ENTERPRISE OBSERVABILITY & AUDIT

> **Note:** Core observability (Analytics Engine, basic traces, Logpush examples) is available in the open core. The full production-grade compliance, SIEM integration, long-term immutable audit, and tenant-aware pipelines described here are part of the commercial Enterprise layer.

One of the biggest jumps from retail HOOX to Enterprise is moving from "some analytics" to **regulator-grade, immutable, queryable audit trail**.

Pillars

1. **Workers Traces** (automatic)
   - End-to-end visibility across Service Bindings, DO RPC, Queues, Workflows, external fetches.
   - No code changes needed for basic tracing.
2. **Logpush** (Enterprise strength)
   - Push Workers Trace Events, HTTP requests, WAF events, Queue events, etc.
   - Destination: R2 (primary), or external SIEM (SentinelOne, Splunk, etc.).
   - One-click R2 setup + advanced options (custom fields, raw vs transformed headers, daily partitioning).
   - High volume support on Enterprise.
3. **Tail Workers**
   - Sample, filter, enrich, or redact logs in real time before they go to Logpush or storage.
   - Example: add tenant context, hash sensitive fields, drop health checks.
4. **R2 as Immutable Audit Store**
   - Combined with R2 Event Notifications → trigger Workflows for post-processing (indexing, compliance checks, retention policies).
   - Jurisdiction controls for data residency.
   - Lifecycle + Object Lock for immutability.
5. **Analytics Engine + Dashboard**
   - Keep high-cardinality real-time metrics (latency, fills, risk events).
   - Tenant-tagged datasets.
6. **Workflows + Observability**
   - Every Workflow step can emit structured events.
   - Workflow execution history is durable and auditable.

Example Logpush Job (R2)

```
{
  "name": "hoox-enterprise-audit",
  "dataset": "workers_trace_events",
  "destination_conf": "r2://hoox-audit-logs/{DATE}?account-id=...&access-key-id=...&secret-access
  "output_options": {
    "field_names": ["EventTimestamp", "EventType", "RayID", "Worker", "Outcome", "TenantId", ...]
    "timestamp_format": "rfc3339"
  },
  "enabled": true
}
```

Use the one-click flow or the API for automation in CLI/deploy.

Tail Worker Example (Tenant Enrichment + Redaction)

```
// workers/tail-audit/src/index.ts

  async tail(events: TraceItem[], env: Env) {
    for (const event of events) {
      // enrich
      const tenant = event.scriptTags?.find(t => t.startsWith("tenant:"))?.slice(7) || "unknown"

      // redact secrets
      const logs = (event.logs || []).map(l => ({
        ...l,
        message: l.message?.replace(/secret:[a-z0-9]+/gi, "secret:[REDACTED]")
      }));

      await env.AUDIT_QUEUE.send({
        tenant,
        ray: event.rayID,
        logs,
        // ...
      });
    }
  }
};
```

Compliance Use Cases

- **Trade reconstruction**: Pull all events for a `clientOrderId` or `traceId` across Workers, DOs, and Queues from R2.
- **Kill switch audit**: Every activation + reason + affected positions logged immutably.
- **Regulatory reports**: Scheduled Workflow reads from R2 + D1, produces signed PDF via Browser Rendering, stores result + manifest in R2.
- **Anomaly detection**: Feed Logpush stream (or sampled Tail) into AI Gateway + Vectorize for real-time risk signals.

Retention & Cost

- R2 is dramatically cheaper than traditional logging at scale.
- Use lifecycle rules: hot (30d) → cold → delete or archive after N years.
- Enterprise customers often keep 7+ years for compliance.

Integration Points

- Existing `trackAnalytics` in shared can be extended to also emit to the audit path.
- All new Workflows and DOs should emit structured `auditEvent` payloads.
- Dashboard (Enterprise) should have "Audit Explorer" views querying R2 (via Worker or R2 SQL if available) + Analytics Engine.

Related

- architecture.mdx
- multi-tenancy.mdx — tenant tagging in logs
- Root `wrangler.jsonc` Enterprise profiles will include Logpush + Tail bindings.