TRADERS · OPERATORS · SELF-HOSTERS

END USER MANUAL

Install, configure, trade, and operate HOOX as a trader

CONTENTS

01 Documentation

02 End User Hub

03 Installation Guide

04 5-Minute Quick Start

05 Platform Configuration

06 How Hoox Works

07 Edge-First Architecture

08 Cloudflare Services Map

09 Signals & Trade Spec

10 Idempotency & Durable Objects

11 AI Risk Manager & AI Gateway

12 Local Development

13 Infrastructure Management

14 Deploying to Production

15 Database Operations

16 Secrets & Network Security

17 Monitoring Operations

18 Terminal UI (TUI)

19 Self-Healing & Repair

20 TradingView Webhook Integration

21 Telegram Bot Setup

22 Email Signal Routing

23 CLI Commands Reference

24 Configuration Dictionary

25 API Endpoint Specification

26 Glossary

27 Frequently Asked Questions

01 · END USER MANUAL

DOCUMENTATION

Edge-native algorithmic trading on the Cloudflare Edge — trader guides, operator manuals, and enterprise architecture.

HOOX is a free, open-source, edge-native algorithmic trading framework deployed on the Cloudflare Edge Network. A distributed microservice mesh ingests trade signals, evaluates risk, routes orders, and fires Telegram notifications — in milliseconds, from the edge nodes closest to the exchange.

```
graph TB
    TV[TradingView Alerts] -->|HTTPS Webhook| GW[hoox Gateway]
    GW -->|Service Binding| TW[trade-worker]
    GW -->|Service Binding| TGW[telegram-worker]
    TW -->|Signed API Request| CentralizedEx[Centralized Exchanges: Binance / Bybit / MEXC]
    TW -->|On-Chain Swap| Web3Wallet[DeFi Wallet Execution]
    AW[agent-worker: Cron Risk Monitor] -->|Scale/Kill Switch| TW
```

Pick a track. **End User** covers setup and trading operations. **DevOps** covers architecture and deployment. **Enterprise** describes the commercial layer.

Documentation Tracks

```
Getting started, core concepts, guides, tutorials, and reference for traders.


Architecture, workers, deployment, development, security, and API internals.


Multi-tenancy, compliance, observability, and the institutional extension layer.
```

Quick Links

- **Installation** — Prerequisites and workspace bootstrap
- **5-Minute Quick Start** — Deploy workers and fire a test trade
- **CLI Commands** — Full CLI reference
- **System Overview** — Edge topology and service mesh
- **GitHub Repository** — Source and contributions

END USER HUB

Trader-facing documentation — getting started, concepts, guides, tutorials, and reference.

> **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.

```
graph TD
    TV[TradingView Alerts] -->|HTTPS Webhook| GW[hoox Gateway]
    GW -->|Service Binding| TW[trade-worker]
    GW -->|Service Binding| TGW[telegram-worker]
    TW -->|Signed API Request| CentralizedEx[Centralized Exchanges: Binance / Bybit / MEXC]
    TW -->|On-Chain Swap| Web3Wallet[DeFi Wallet Execution]
    AW[agent-worker: Cron Risk Monitor] -->|Scale/Kill Switch| TW
```

Choose Your Path

Whether you are a retail algorithmic trader setting up your first automated TradingView strategies, a quantitative analyst exploring low-latency DeFi order routing, or a DevOps engineer maintaining multi-exchange infrastructure, our docs are split into highly focused tracks:

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** — Provision prerequisites (Bun runtime, Cloudflare credentials) and bootstrap a project.
- **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:

- **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** — Preventing catastrophic double-execution and race conditions during high-frequency events.
- **Signals & Trade Routing** — How parameters map from webhook JSON schemas to live exchange payloads.
- **AI Risk Manager** — The 5-minute autonomous risk scanner, 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 9-view operations cockpit (`./hoox-tui`).
- **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.
- **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.

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 v5 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!

INSTALLATION GUIDE

Prepare your local machine, install the hoox CLI tool, bootstrap workspaces, and verify system prerequisites.

This guide walks you through preparing your environment, installing the `@jango-blockchained/hoox-cli` tool, cloning the microservices monorepo recursively, and validating your local system prerequisites.

System Prerequisites

Before installing the Hoox command-line workspace, ensure your system meets the following standard software requirements:

1. Bun JavaScript Runtime (Version ≥ 1.2)

Hoox uses **Bun** as its primary package manager, script runner, and native testing engine. Bun's blazing-fast startup time and zero-config TypeScript compilation are critical to our local developer feedback loops.

- **macOS / Linux**:

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

- **Windows (via Powershell)**:

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

2. Cloudflare Account & Wrangler CLI

All Hoox workers are compiled to run on **Cloudflare's Edge V8 isolates**. You will need a free Cloudflare account:

- **Account**: Register a free account (the free tier provides 100,000 requests/day, D1 database, KV storage, Queues, R2, and vector search—costing $0/month).
- **Cloudflare CLI**: Ensure you have `wrangler` installed globally (though Hoox manages local wrangler operations automatically, having it indexed globally is recommended):

```
bun add -g wrangler
```

3. Docker & Docker Compose (Optional)

If you prefer running the entire Hoox local trading workspace in fully isolated container environments with one command, ensure you have **Docker Desktop** installed and running:

- Check status: `docker compose version`

Option A: Install via Package Manager (Recommended)

To install the global management CLI tool directly from npm/bun registry to manage new workspaces:

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

Once installed, verify that the `hoox` command is registered globally in your system path:

```
hoox --version
```

Option B: Build & Run from Source

If you plan to contribute to the Hoox CLI packages or prefer working directly inside a monolithic source clone:

```
# 1. Clone the parent repository with git submodules recursively
git clone --recursive https://github.com/jango-blockchained/hoox-setup.git hoox-trading
cd hoox-trading

# 2. Install all monorepo workspace dependencies via Bun
bun install

# 3. Verify the locally linked CLI runs from root
bun run build: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 deployments will fail.

Bootstrapping Your Trading Workspace

Now, initialize your new algorithmic trading workspace. This compiles directories, configures workspace settings, and sets up Git tracking.

```
# Bootstrap your trading folder
hoox clone my-trading-empire
cd my-trading-empire
```

This command automatically structures your directories as a clean monorepo:

```
my-trading-empire/
├── packages/
│   ├── cli/       # Workspace CLI management binaries
│   ├── tui/       # 9-view Terminal UI monitoring cockpit
│   └── shared/    # Reusable routers, rate-limiters, & errors
├── workers/
│   ├── hoox/      # Gateway, rate-limiter, DO idempotency lock
│   ├── trade-worker/    # Multi-exchange execution engine
│   └── ...        # Other analytical and web3 wallet workers
└── package.json
```

Running the Onboarding Wizard

With your folder structured, run the one-shot bootstrap wizard:

```
# Recommended: chains config + infrastructure in one command
hoox onboard
```

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 `hoox onboard` wizard chains both steps and will guide you through the critical setup phases:

1. **Cloudflare Authentication**: Prompts for your Cloudflare API token (with `Account.D1`, `Account.KV`,{" "} `Account.Workers` permissions) and Account ID.
2. **Microservice Profile Selection**: Lets you declaratively toggle which edge workers to enable (e.g., enable Gateway and Trade Worker, disable Web3 Wallet if you only trade centralized exchanges).
3. **Local Credentials Encryption**: Generates safe local{" "} `.dev.vars` matrices and sets up initial KV configuration structures.
4. **Infrastructure Provisioning**: Creates D1 databases, KV namespaces, and other resources on your Cloudflare account.

```
┌──────────────────────────────────────────────────────────┐
│               hoox Setup & Initialization                │
├──────────────────────────────────────────────────────────┤
│   bun (v1.2.1) found                                     │
│   wrangler (v3.50.0) 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            │
└──────────────────────────────────────────────────────────┘
```

Verifying Local Prerequisites

Verify that all dependencies and Cloudflare edge routes are accessible:

```
# Perform detailed pre-flight diagnostic checklist
hoox check prerequisites
```

If all diagnostic checks pass, you are ready to proceed with configuration!

Got installation issues? Run `hoox repair check` to automatically analyze common path resolution issues, missing environment variables, or node-gyp build failures, and recover your local workspace seamlessly.

Next Steps

- **Configuration Matrix** — Map out your 31-key environment variables and 16-key KV registries.
- **5-Minute Quick Start Guide** — Execute your first simulated edge trade in under 5 minutes.

5-MINUTE QUICK START

Deploy your first edge-native trading infrastructure on the Cloudflare Edge 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.

> **v0.8.0:** The recommended entry point is now `hoox onboard` (one-shot bootstrap) instead of running `hoox init` and `hoox setup` separately.

Step-by-Step Deployment Path

Step 1: Onboard Your Workspace

Run the one-shot bootstrap to collect your Cloudflare credentials, write `wrangler.jsonc`, and provision all infrastructure end-to-end:

```
hoox 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.

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
```

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:

```
# Inject Bybit Credentials (worker name is the first argument)
hoox secrets set trade-worker BYBIT_KEY_BINDING "your_bybit_key_here"
hoox secrets set trade-worker BYBIT_SECRET_BINDING "your_bybit_secret_here"

# Optional: Inject Binance Credentials
hoox secrets set trade-worker BINANCE_KEY_BINDING "your_binance_key_here"
hoox secrets set trade-worker BINANCE_SECRET_BINDING "your_binance_secret_here"
```

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
hoox deploy all --auto
```

This command automatically provisions:

1. **D1 Edge Database** (`hoox-db`)
2. **CONFIG_KV** configuration namespace
3. **Internal Workers** (`trade-worker` ,{" "} `d1-worker` , `telegram-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
```

Step 5: Fire a Simulated Trade Webhook

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

```
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
  }'
```

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**	Trading action direction: `LONG` (buy/open), `SHORT` (sell/open), or `CLOSE` (flatten position).
`symbol`	`string`	**Yes**	Standard market symbol.
`quantity`	`number`	**Yes**	Position size / quantity.
`leverage`	`number`	No	Leverage coefficient. Defaults to `1` (spot) if omitted.

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.

05 · 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.

---

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.

- **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.

06 · 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

```
sequenceDiagram
    autonumber
    actor Trigger as Signal Source (TradingView / Email / Telegram)
    participant GW as hoox Gateway (Public WAF + V8 Isolate)
    participant DO as Durable Object (Idempotency Lock)
    participant Q as Cloudflare Queue (Asynchronous Backup)
    participant TW as trade-worker (Execution Engine)
    participant CEX as Exchange API (Bybit / Binance / MEXC)
    participant D1 as D1 Database (Edge-SQLite)
    participant TG as telegram-worker (Notifications)

    Trigger->>GW: HTTP POST /webhook (JSON payload with Auth Passkey)
    GW->>DO: Lock Idempotency Key (Verify Trace ID)
    alt Already Processed (Duplicate Request)
        DO-->>GW: Fail: Duplicate Intercepted
        GW-->>Trigger: 409 Conflict (Duplicate Request)
    else Unique Request (Success Path)
        DO->>GW: Lock Acquired
        GW->>TW: Service Binding invoke (Fast Path)
        alt Direct Edge Call Succeeds
            TW->>CEX: Send Signed HMAC-SHA256 REST API Order
            CEX-->>TW: Order Executed (Fill Status + Details)
            TW->>D1: Write Transaction Record to SQLite Table
            TW->>TG: Service Binding invoke (Alert Notification)
            TG-->>Trigger: Send telegram alert to user mobile
            TW-->>GW: Return Executed JSON Payload
            GW-->>Trigger: 200 OK (Fill Result)
        else Direct Edge Call Fails (Exchange Offline / Rate Limited)
            GW->>Q: Enqueue Trade Payload (Guaranteed Delivery)
            GW-->>Trigger: 202 Accepted (Status: Enqueued)
            Note over Q, TW: Queue worker retries execution w/ exponential backoff
        end
    end
```

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 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**: Verifies that signal frequency does not exceed safe execution thresholds (default: 10 orders/minute).
5. **Idempotency Check**: Locks a unique transaction trace ID using a **SQLite-backed Durable Object**. If the DO detects that the exact same signal hash was already processed, it drops the request to prevent double-ordering.

---

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 SQLite Database**: Writes transaction logs, filled prices, execution fees, and order IDs to your D1 database.
- **R2 Log Offloading**: Offloads full JSON request-response packets to your secure R2 storage bucket to keep D1 database footprints 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.

07 · CONCEPTS

EDGE-FIRST ARCHITECTURE

Why V8 isolates and Cloudflare's global edge cut latency by 60% versus traditional VPS trading bots.

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

```
[Public Request] → hoox (gateway:8787)
                    |
        ┌───────────┼──────────────┐
        |           |              |
    [SB:trade]  [SB:d1-worker]  [SB:telegram]
        |           |              |
    trade-worker  d1-worker    telegram-worker
    (port:8789)  (port:8792)   (port:8791)
        |           |
   [Exchange API]  [D1 SQLite]
```

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.

08 · 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**: Drives the Gateway's **Idempotency Engine** inside `workers/hoox`.
- **Why it Matters**:
  - When TradingView fires multiple retries for the same alert (due to a transient network timeout), Durable Objects intercept the request, acquire an atomic mutex lock, and evaluate the trade's unique hash.
  - Prevents disastrous duplicate trades (e.g. accidentally buying the same spot position twice) during moments of high market congestion.

---

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.

09 · CONCEPTS

SIGNALS & TRADE SPEC

Webhook JSON schemas and the mapping from external trade signals to exchange order payloads.

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"; // 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)
  takeProfit?: number; // Optional take-profit price target
  stopLoss?: number; // Optional stop-loss price target
}
```

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` (buy/open), `SHORT` (sell/open), `CLOSE` (flatten position)
`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 from `SHA256(symbol + exchange + action + timestamp)`
`price`	`number`	No	Limit price; defaults to current market price if omitted
`takeProfit`	`number`	No	Target exit price for automatic partial close
`stopLoss`	`number`	No	Stop-loss trigger price; overrides agent-worker trailing stop if specified

2. Dynamic Payload Translation & Side Mapping

Exchanges do not understand actions like `LONG`, `SHORT`, or `CLOSE`. 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	Position State	Translated Exchange Side	Operation Type
`LONG`	Closed / No Position	**`BUY`**	Opens a long position (spot or margin)
`SHORT`	Closed / No Position	**`SELL`**	Opens a short position (margin/futures)
`CLOSE`	Long Position Active	**`SELL`**	Closes and flattens an existing long position
`CLOSE`	Short Position Active	**`BUY`**	Closes and flattens an existing short position

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:

```
Contract Quantity = (Order Size (USDT) × Leverage) / 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.

10 · CONCEPTS

IDEMPOTENCY & DURABLE OBJECTS

How Durable Objects guarantee exactly-once execution and prevent duplicate orders on webhook retries.

How Hoox utilizes Cloudflare Durable Objects to guarantee exactly-once execution and prevent catastrophic duplicate orders during network dropouts.

The Danger: How Webhook Retries Lead to Double-Ordering

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

```
[TradingView Webhook] ── (Signal Post) ──> [Gateway Node] ── (Submit Order) ──> [Exchange API
                                                                                        |
                                                                                (Order Fille
                                                                                        |
[TradingView Webhook] <── (TLS/TCP Dropout) ── [Gateway Node] <── (Send Success) ── (Connection
         |
(No response: Retries!)
         |
[TradingView Webhook] ── (Signal Post) ──> [Gateway Node] ── (Submit Order) ──> [Exchange API
                                                                                        |
                                                                               (DOUBLE-FILLE
```

The Hoox Solution: Durable Objects Mutex Locking

To enforce exactly-once execution, Hoox implements an **atomic dedup lock** inside `workers/hoox` 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

```
[Incoming Webhook Payload]
         │
         ▼
[Extract Trace ID / Mutex Key]
         │
         ▼
[Ping Dedicated Durable Object Isolate]
         │
  ┌──────┴──────────────────────────────────────────┐
  │ Single-Threaded Mutex Lock Acquired             │
  │                                                 │
  │ Check local SQLite dedup log:                   │
  │ "Has Trace ID '9b1deb4d...' been seen?"         │
  └─────────────────────────────────────────────────┘
         │
         │
         ├─► [YES: Duplicate Detected] ──────────────────┐
         │                                               │
         │                                               ▼
         │                                    [Silent Dropping of Request]
         │                                               │
         │                                               ▼
         │                                    [Return 409 Conflict Response]
         │
         └─► [NO: Unique Transaction] ───────────────────┐
                                                         │
                                                         ▼
                                                [Record ID in SQLite Log]
                                                         │
                                                         ▼
                                                [Process Pipeline Execution]
                                                         │
                                                         ▼
                                                [Set TTL-based DO Alarm]
```

The Dedup & Cleanup Algorithm

1. Trace ID Generation

When a trade signal is fired, it must include a unique transaction signature.

- For TradingView webhooks, this is automatically generated as a combination of alert parameters and timestamps.
- If a client does not supply a transaction ID, the Hoox Gateway dynamically hashes the symbol, exchange, action, and timestamp to create a unique **Idempotency Key**.

2. Atomic Evaluation

Before executing any order routing:

1. The gateway routes the request to the namespace-mapped Durable Object using the transaction ID as the binding key.
2. The Durable Object checks its internal state. Since DOs are single-threaded, there is **zero risk of race conditions**—if two identical HTTP requests hit Cloudflare simultaneously, they are processed sequentially

inside the DO.

3. If the ID exists in the DO's SQLite log, the DO immediately intercepts the request and throws a `409 Conflict` exception, stopping the pipeline before hitting exchange APIs.
4. If unique, it registers the ID, saves the current timestamp, and returns a lock approval.

3. Automatic TTL & Storage Alarms

To prevent the Durable Object's persistent storage from growing indefinitely and consuming unnecessary memory:

- The DO registers an **atomic alarm** scheduled for 24 hours in the future.
- When the alarm fires, the DO runs an automatic garbage collection script that purges old IDs from its local storage.
- This ensures that while you are 100% protected against duplicates during network dropouts, your storage footprint remains lightweight.

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.

AI RISK MANAGER & AI GATEWAY

How the agent-worker runs 5-minute cron checks 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 continuous **5-minute Cloudflare Cron trigger**, 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

Every 5 minutes, `agent-worker` executes a multi-point risk diagnostic loop across all active exchanges:

```
graph TD
    Cron[5-Min Cron Trigger] --> Query[Query D1 & Exchange Balances]
    Query --> Drawdown{Drawdown > Max Daily Limit?}
    Drawdown -->|YES | Kill[Flip Global Kill Switch & Flatten All Positions]
    Drawdown -->|NO| Stops[Evaluate Open Positions]
    Stops --> Trailing{Price Moved in Profit?}
    Trailing -->|YES | MoveStop[Move Stop-Loss Up dynamically]
    Trailing -->|NO| ScaleProfit{Price Hitted Take-Profit Target?}
    ScaleProfit -->|YES | ScaleOut[Market Sell partial position e.g., 25%]
    ScaleProfit -->|NO| Summary[Compile System Health Stats]
    Summary --> Send[Notify user via Telegram]
```

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**:

  ```
  Trailing Stop Price (Long) = Peak Price × (1 – 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:

```
Daily Drawdown % = ((Starting Daily Balance - Current Balance) / Starting Daily Balance) × 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

```
[User Chat Query]
       |
       ▼
[1. Cloudflare Workers AI] (Zero cost, native LLaMA 3) ──▶ Fail? ──┐
                                                                   |
                                                                   ▼
[2. Anthropic API] (Claude 3.5 Sonnet) ◀──────────────────────────┘
       |
     Fail? ──▶ [3. Google AI] (Gemini 1.5 Pro) ──▶ Fail? ──▶ [4. OpenAI] (GPT-4o)
```

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.

You can adjust all AI gateway defaults, Cron check intervals, and trailing deviations in real-time by writing to your KV configurations. No code deployments are ever required to tune your risk 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.

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.

Starting the Local Workspace

To spin up all enabled workers simultaneously:

```
hoox 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).
- **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 10 services)
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 |

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] (1,574 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.

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.

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.

DEPLOYING TO PRODUCTION

Deployment dependency chain, OpenNext builds, service bindings, and production rollout runbooks.

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.

This guide outlines our automated deployment sequence, 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:

```
[1. D1 Database Schema] ──▶ [2. d1-worker (SQL Hub)] ──▶ [3. trade-worker (Execution Engine)]
                                                                          │
    ┌─────────────────────────────────────────────────────────────────────┘
    ▼
[4. hoox Gateway (Public Router)] ──▶ [5. agent-worker & telegram-worker]
                                                          │
                                                          ▼
                                              [6. Next.js Dashboard (OpenNext)]
```

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	Background services — can deploy independently
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 tail dashboard`.
Telegram bot unresponsive	Webhook URL expired or token rotated.	Re-register: `hoox deploy telegram-webhook` and verify token: `hoox secrets check`.

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.

DATABASE OPERATIONS

D1 schema management, migrations, ledger queries, and backup runbooks for the edge SQL layer.

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.

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

```sql
-- 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.

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, and edge-level firewall protection runbooks.

---

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**:

```
[Public Internet] ──► [Cloudflare WAF / Firewall]
                                  │
                          (IP & Auth Checks)
                                  │
                                  ▼
                        [Gateway (hoox)] (Publicly Accessible Node)
                                  │
                  ┌───────────────┴────────────────┐
                  │ V8 Isolate Service Bindings    │
                  │ (Private, Encrypted, Zero-TCP) │
                  └────────────────┬───────────────┘
                                   │
                 ┌─────────────────┴─────────────────┐
                 ▼                 ▼                 ▼
          [trade-worker]      [d1-worker]      [agent-worker]
          (No Public URL)   (No Public URL)   (No Public URL)
                 │                 │                 │
        (Exchange Orders)   (SQLite Queries)   (AI Risk Audits)
```

- **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.

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

```
# 1. Inject a secure exchange credential (e.g. Bybit Secret)
hoox secrets set BYBIT_API_SECRET "your_private_signature_here"

# 2. Check the synchronization status of all required edge secrets
hoox secrets check
```

The Secrets Diagnostic Report

Running `hoox secrets check` queries Cloudflare's API to confirm that the key binding exists on the edge, without ever exposing or decrypting the actual values in your terminal:

```
┌──────────────────────────────────────────────────────────┐
│               Cloudflare Edge Secrets Audit              │
├──────────────────────────────────────────────────────────┤
│  BYBIT_API_KEY ...........  PRESENT (Active)            │
│  BYBIT_API_SECRET ........  PRESENT (Active)            │
│  TELEGRAM_BOT_TOKEN ......  PRESENT (Active)            │
│  OPENAI_API_KEY ..........  MISSING (Optional)         │
│                                                          │
│  Audit Result: SECURE (All required secrets bound)       │
└──────────────────────────────────────────────────────────┘
```

---

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`.

MONITORING OPERATIONS

Health probes, queue backlogs, live edge logs, and emergency kill-switch controls via the CLI.

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
```

> `hoox monitor status` was removed in v0.8.0. The single source of truth for health checks is now `hoox check health`.

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 tail <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 5-minute 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
`Filled`	Order fully executed on the exchange
`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

---

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 using Wrangler bindings:

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

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

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

# D. Filter logs by severity level
hoox logs tail 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 download hoox --output logs/gateway-backup.log

# Download trade-worker logs for the last 24 hours
hoox logs download 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.

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)** 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.

Launching the TUI

```
# 1. Recommended: via the hoox CLI
hoox tui

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

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

# 4. 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
`Ctrl+Q`	Quit (confirmation dialog)
`Esc`	Close palette / dismiss modal
`Tab` / `Shift+Tab`	Cycle focus
`Enter`	Select / confirm
`Space`	Toggle (e.g. pause trade feed)
`/`	Search (searchable views)

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`)

Theme / notifications / refresh interval / shortcuts reference, **Check Setup**, **Check Fix**, config path display.

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).

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

SELF-HEALING & REPAIR

Diagnose failures, rotate credentials, and recover workspaces with the hoox repair command group.

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 setting; 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 tail 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.

TRADINGVIEW WEBHOOK INTEGRATION

How to configure TradingView alerts, write copy-paste Pine Script v5 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's advanced charting engines with Hoox's low-latency edge execution, you can trigger orders instantly based on mathematical indicators, chart patterns, or custom Pine Script v5 logic.

This tutorial walks you through writing a Pine Script v5 script, setting up a TradingView webhook alert, and testing executions.

Step 1: Writing a Pine Script v5 Signal Indicator

To trigger clean trade alerts, use TradingView's **Pine Script v5**. Below is a copy-paste indicator script that evaluates a Moving Average Cross strategy and generates standardized trade alert signals:

```
//@version=5
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.aboveb

// 4. Alert Payloads Construction (JSON compliant)
longAlertJson = '{"apiKey": "your-hoox-webhook-passkey", "exchange": "bybit", "action": "LONG", "
shortAlertJson = '{"apiKey": "your-hoox-webhook-passkey", "exchange": "bybit", "action": "SHORT",

// 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.

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 tail hoox
```

Webhook Troubleshooting Runbook

Symptom / Error	Primary Cause	Resolution
Alert fires but no trade executes	Mismatched API keys or WAF block.	Run `hoox logs tail 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.

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.

---

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.

CLI COMMANDS REFERENCE

Comprehensive CLI directory detailing all 25 command groups, 60+ subcommands, global flags, and positional options for the hoox binary.

The `@jango-blockchained/hoox-cli` tool 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.

> **Heads up (v0.9.0):** The output framework was polished. New global flag `--no-color` (alongside `NO_COLOR` env var and `TERM=dumb` honored); every successful command prints a `Done in 1.2s` footer with an optional `→ next: hoox …` suggestion; typos like `hoox deplpy` produce `did you mean 'hoox deploy' ?`; `hoox --help` uses a new sectioned layout (Usage / Options / Examples / See also). See the changelog below. Also see the v0.8.0 refactor notes further down for the command-tree changes.

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

```
hoox                                  Launch TUI (when run with no args on an init'd workspace)
├── onboard                           One-shot full bootstrap (init + setup) [recommended]
├── init                              Interactive setup wizard (config only)
├── setup                             Auto-bootstrap infrastructure (keys, D1, secrets, dashboard)
├── clone [name]                      Bootstrap workspace from template
├── 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]      Get/Set dynamic KV runtime parameters
│   ├── 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]                 Sync local .dev.vars to Cloudflare
├── 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
├── 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                             Time-series log extraction
│   ├── tail <worker>                Stream live edge logging console
│   └── download <worker>            Download logs to a local file
├── schema                           Validate wrangler.jsonc schemas and workers
├── update                           Self-update the CLI or wrangler
├── test                             Run verification pipeline (CI)
├── waf                              Auto-configure Cloudflare WAF firewall rules
├── clone [name]                     Clone worker repositories as submodules
├── 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
├── trace                            Query Cloudflare Workers Observability (events, metrics)
├── 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
hoox secrets set trade-worker BINANCE_KEY_BINDING "your_key_here"
hoox secrets set trade-worker BINANCE_SECRET_BINDING "your_secret_here"

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

# Sync local .dev.vars to Cloudflare
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
```

🆕 v0.9.0 Output Polish

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 status` / `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** — `minimal` (was `legacy`). The banner's version is read dynamically from `package.json` (v0.9.0 also fixed a real version-drift bug where the banner was hardcoded to `v0.3.0` while the package was at `v0.8.0`).

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

🆕 v0.8.0 Refactor

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 (added in v0.8.0)

- **`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.

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.

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`). 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,
  "idempotencyKey": "uuid-9b1deb4d-3b7d"
}
```

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 `@jango-blockchained/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.

GLOSSARY

Vocabulary for trading, Cloudflare edge services, and Hoox platform concepts used throughout the docs.

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.
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.

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.
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.
Bun Workspace	A Bun feature enabling multiple packages within a monorepo to share dependencies and reference each other locally.
Wrangler	Cloudflare's official CLI tool for developing, deploying, and managing Cloudflare Workers and associated resources.
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.

FREQUENTLY ASKED QUESTIONS

Common questions about setup, operations, troubleshooting, and the Hoox trading platform.

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/jango-blockchained/hoox-setup.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
- **Git** ≥ 2.40
- **Wrangler CLI** (latest)
- 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: 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. They are never stored in plaintext on disk or in Git.

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 tail 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's the 5-minute cron doing exactly?

The `agent-worker` runs every 5 minutes via Cloudflare Cron 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

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.