# HOOX TRADING PLATFORM

## END-USER WORKSPACE & CLIENT OPERATIONS MANUAL

Generated: 2026-05-19

Classification: Technical Documentation Spec

Design Style: Monospace Terminal Console layout

# TABLE OF CONTENTS

- HOOX DEVELOPER HUB
- INSTALLATION GUIDE
- PLATFORM CONFIGURATION
- -MINUTE QUICK START
- HOW HOOX WORKS
- EDGE-FIRST ARCHITECTURE
- CLOUDFLARE SERVICES MAP
- IDEMPOTENCY & DURABLE OBJECTS
- SIGNALS & TRADE SPEC
- AI RISK MANAGER & AI GATEWAY
- LOCAL DEVELOPMENT
- TERMINAL UI (TUI)
- DATABASE OPERATIONS
- SECRETS & NETWORK SECURITY
- INFRASTRUCTURE MANAGEMENT
- DEPLOYING TO PRODUCTION
- SELF-HEALING & REPAIR
- TRADINGVIEW WEBHOOK INTEGRATION
- TELEGRAM BOT SETUP
- EMAIL SIGNAL ROUTING
- CLI COMMANDS REFERENCE
- API ENDPOINT SPECIFICATION
- CONFIGURATION DICTIONARY

**[ SECTION: HOOX DEVELOPER HUB ]**

# WELCOME TO THE HOOX DEVELOPER HUB

> Hoox is a free, open-source, zero-latency 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.

---

## 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:

### . 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  minutes:

- [Core Installation](getting-started/installation.md) - Provision prerequisites (Bun runtime, Cloudflare credentials) and bootstrap a project.
- [Platform Configuration](getting-started/configuration.md) - Declaratively define environment variables, JSON profile templates, and KV keys.
- [-Minute Quick Start](getting-started/quick-start.md) - Launch local worker runners and execute a simulated webhook trade signal.

### . CORE CONCEPTS

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

- [How Hoox Works](concepts/how-hoox-works.md) - The end-to-end lifecycle of a trade signal from webhook alert to order confirmation.
- [Edge-First Architecture](concepts/edge-architecture.md) - Why V isolates and Cloudflare Workers outperform traditional VPS architectures by -%.
- [Cloudflare Services Map](concepts/cloudflare-services.md) - How D, KV, R, Queues, Vectorize, and Browser Rendering are integrated.
- [Idempotency & Durable Objects](concepts/idempotency.md) - Preventing catastrophic double-execution and race conditions during high-frequency events.
- [Signals & Trade Routing](concepts/signals-and-trades.md) - How parameters map from

webhook JSON schemas to live exchange payloads.
- [AI Risk Manager](concepts/ai-risk-manager.md) - The -minute autonomous risk scanner, trailing-stop mathematician, and account kill-switch.

## . OPERATIONAL GUIDES

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

- [Local Dev & Workspaces](guides/local-development.md) - Run, hot-reload, and test workers locally via Bun or Docker container stacks.
- [Terminal UI Operations](guides/tui.md) - Launch and navigate the full-screen -view operations cockpit (`./hoox-tui`).
- [Edge Database Operations](guides/database-ops.md) - Manage global SQLite schemas, track drizzle migrations, and run D queries.
- [Secrets & Network Security](guides/secrets-security.md) - Secure Cloudflare Zero Trust corridors and encrypt sensitive API credentials.
- [Infrastructure Management](guides/manage-infra.md) - Spin up and inspect KV, Queues, R, and Vectorize indexes in one command.
- [Deploying to Production](guides/deploy-workers.md) - Roll out code, bind V isolates, and deploy Next.js dashboard consoles.
- [Self-Healing & Repair](guides/repair.md) - Diagnose connection dropouts, rotate API keys, and run interactive system repair tools.

## . STEP-BY-STEP TUTORIALS

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

- [TradingView Webhook Integration](tutorials/tradingview-webhook.md) - Write customized Pine Script v indicators that ping the Hoox webhook receiver.
- [Telegram Bot Setup](tutorials/telegram-bot.md) - Configure real-time order alerts, P&L reports, and secure chat commands.
- [Email Signal Routing](tutorials/email-signals.md) - Configure email parsing services to convert raw inbox alerts into edge trade executions.

## . REFERENCE MATERIAL

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

- [CLI Commands Index](reference/cli-commands.md) - Complete options, positional arguments, and JSON flag trees for the `hoox` tool.
- [API Spec & REST Routes](reference/api-endpoints.md) - HTTP endpoints, request payloads, response templates, and edge errors list.
- [Configuration Properties](getting-started/configuration.md) - Comprehensive dictionary of all  env keys and  KV key-value items.

---

## OFFLINE REFERENCE & AI CONTEXT

Download our consolidated specifications or view complete text packages for feeding into LLMs:

- [Download Enduser Full PDF Manual](/hoox-setup/Enduser-Full-Documentation.pdf) - Complete concatenated offline guide.
- [Download DevOps Full PDF Manual](/hoox-setup/DevOps-Full-Documentation.pdf) - Complete concatenated offline DevOps spec.
- [View Consolidated LLM Context Text](/hoox-setup/llm.txt) - Giant single-file text format for AI/LLM models.

---

> Tip: Looking for deep engineering plans, DDL SQL schemas, or CI/CD deployment workflows? Check out the [DevOps & Architecture Manual](../devops/home.md) for developer-centric operator blueprints!

**[ SECTION: INSTALLATION GUIDE ]**

# INSTALLATION GUIDE

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:

### . BUN JAVASCRIPT RUNTIME (VERSION .)

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:
  ```bash
  curl -fsSL https://bun.sh | bash
  ```
- Windows (via Powershell):
  ```powershell
  powershell -c "irm https://bun.sh/install.ps | iex"
  ```

### . CLOUDFLARE ACCOUNT & WRANGLER CLI

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

- Account: [Register a free account](https://dash.cloudflare.com/) (the free tier provides , requests/day, D database, KV storage, Queues, R, and vector search-costing $/month).
- Cloudflare CLI: Ensure you have `wrangler` installed globally (though Hoox manages local wrangler operations automatically, having it indexed globally is recommended):
  ```bash
  bun add -g wrangler
  ```

### . 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:

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

  . Install all monorepo workspace dependencies via Bun
bun install

  . Verify the locally linked CLI runs from root
bun run build:cli
./packages/cli/bin/hoox.js --help
```

> Warning: 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/        -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 web wallet workers
|-- package.json
```

---

## RUNNING THE INTERACTIVE SETUP WIZARD

With your folder structured, execute the interactive bootstrap wizard:

```
hoox init
```

The CLI wizard will guide you through  critical setup phases:

. Toolchain Validation: Probes your machine for `bun`, `git`, `wrangler`, and `docker` configurations.
. Cloudflare Authentication: Prompts for your Cloudflare API token (with `Account.D`, `Account.KV`, `Account.Workers` permissions) and Account ID.
. Microservice Profile Selection: Lets you declaratively toggle which edge workers to enable (e.g., enable Gateway and Trade Worker, disable Web Wallet if you only trade centralized exchanges).
. Local Credentials Encryption: Generates safe local `.dev.vars` matrices and sets up initial KV configuration structures.

```
+-------------------------------------------------------+
|              hoox Setup & Initialization              |
|-------------------------------------------------------
|   bun (v..) found                                    |
|   wrangler (v..) 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!

---

> Tip: 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](configuration.md) - Map out your -key environment variables and -key KV registries.
- [-Minute Quick Start Guide](quick-start.md) - Execute your first simulated edge trade in under  minutes.

## [ SECTION: PLATFORM CONFIGURATION ]

# PLATFORM CONFIGURATION

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.

---

## . 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  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 `D`, `KV`, `Queue`, and `Workers` write permissions. | `cf_pat_abc...` |
| `CLOUDFLARE_ACCOUNT_ID` | Yes | Your unique -character Cloudflare dashboard account hash. | `debcebeabecbcdec` |
| `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-SHA 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. | `:ABCdefGh...` |
| `TELEGRAM_CHAT_ID` | No | The numeric chat ID of the user, group, or channel where alerts route. | `` |

### 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.

---

## . 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:

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

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

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

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

> Note: 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.

---

## . WORKER CONFIGURATIONS (`WRANGLER.JSONC`)

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

. Service Bindings: Connects gateway routes (`hoox`) to internal compute units (`trade-worker`, `d-worker`) directly via microsecond V isolates-no TCP/TLS overhead, no public routes.
. Resource Bindings: Declares which D databases, R storage buckets, and KV configurations are linked.
. 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.web-wallet-worker.enabled false
```

---

## . 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 -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` | `` | 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
```

---

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

## NEXT STEPS

- [-Minute Quick Start Guide](quick-start.md) - Launch local worker runners and execute a simulated webhook trade signal.
- [Deploying to Production](../guides/deploy-workers.md) - Deploy and bind all workers in the correct dependency sequence.

## [ SECTION: -MINUTE QUICK START ]

# -MINUTE QUICK START

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 minutes.

---

## STEP-BY-STEP DEPLOYMENT PATH

### STEP : INITIALIZE WORKSPACE CREDENTIALS

If you haven't already run the setup wizard during installation, execute the initial workspace configuration:

```
hoox init
```

_Follow the interactive prompts to paste your Cloudflare Account ID and API Token, and define your unique `SUBDOMAIN_PREFIX` (e.g., `alpha-trading`)._

---

### STEP : 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
hoox secrets set BYBIT_API_KEY "your_bybit_key_here"
hoox secrets set BYBIT_API_SECRET "your_bybit_secret_here"

 Optional: Inject Binance Credentials
hoox secrets set BINANCE_API_KEY "your_binance_key_here"
hoox secrets set BINANCE_API_SECRET "your_binance_secret_here"
```

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

---

### STEP : 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:_

. D Edge Database (`hoox-db`)
. CONFIG_KV configuration namespace
. Internal Workers (`trade-worker`, `d-worker`, `telegram-worker`)
. Public Gateway (`hoox` gateway router)
. 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 : 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": .,
    "leverage":
  }'
```

---

## 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 `` (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 D SQLite table.

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

```
{
  "success": true,
  "requestId": "bdebd-bd-bad-bdd-bdbdcbd",
  "exchange": "bybit",
  "symbol": "BTCUSDT",
  "action": "LONG",
  "result": {
    "orderId": "",
    "status": "Filled",
    "executedQty": .,
    "price": .,
    "timestamp":
  }
}
```

---

> Tip: 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](../concepts/how-hoox-works.md) - Take a deep dive into the edge processing pipelines.
- [Terminal UI Guide](../guides/tui.md) - Run, hot-reload, and inspect live metrics in a

full-screen console interface.
- [Set Up Telegram Alerts](../tutorials/telegram-bot.md) - Link your bot to get order fills pushed straight to your phone.

**[ SECTION: HOW HOOX WORKS ]**

# HOW HOOX WORKS

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

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

---

## THE EXECUTION PIPELINE

---

## DETAILED PIPELINE PHASE ANALYSIS

### . 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=. 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`.

---

### . EDGE FIREWALL & GATEWAY VALIDATION

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

. API Key Authorization: Authenticates the payload's `apiKey` property against your secure manifest stored in Cloudflare KV.
. 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).
. Global Kill Switch check: Ensures `trade:kill_switch` is `false`. If `true`, the pipeline aborts immediately to protect your capital.
. Rate Limiting: Verifies that signal frequency does not exceed safe execution thresholds

(default: orders/minute).
. 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.

---

## . SERVICE BINDING ORDER ROUTING

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

- Zero Latency: Communication occurs within a microsecond inside the V 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.

---

## . EXCHANGE EXECUTION & CRYPTOGRAPHIC SIGNING

The `trade-worker` loads your encrypted API credentials from Cloudflare Secrets, computes the HMAC-SHA 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 %.

---

## . PERSISTENT D LOGGING & TELEMETRY

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

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

---

## . 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.

---

> Tip: 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 (`s` `m` `m` `m`), keeping your strategy fully reliable even during periods of heavy market volatility.

## NEXT STEPS

- [Edge-First Architecture](edge-architecture.md) - Understand the absolute latency benefits of edge workers vs VPS.
- [Signals & Trade Specifications](signals-and-trades.md) - Analyze the complete JSON webhook and order formatting.

**[ SECTION: EDGE-FIRST ARCHITECTURE ]**

# EDGE-FIRST ARCHITECTURE

In algorithmic trading, latency is leverage. A delay of just  milliseconds between a signal firing and an order arriving at the exchange can mean the difference between a highly profitable fill and severe price slippage.

Hoox solves the latency problem by abandoning traditional server-centric architectures (like AWS, DigitalOcean, or standard VPS instances) and executing order logic natively on Cloudflare Edge Workers.

---

## 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 (MS+ SLIPPAGE)

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

**[London Alert] -- (ms) --> [Virginia VPS] -- (ms) --> [Frankfurt Bybit] = ms Latency**

### THE HOOX EDGE PATH (UNDER MS LATENCY)

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

**[London Alert] -- (.ms) --> [London PoP] -- (ms Service Binding) --> [Frankfurt Node] -- (ms) --> [Bybit] = .ms Latency**

---

## V 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 (V Isolate) |
| --- | --- | --- |
| Boot Architecture | Heavy guest OS, Node runtime, npm modules | Sandboxed JavaScript V Engine runtime |
| Cold-Start Delay | ,ms - ,ms (system stalls) | < ms (virtually non-existent) |
| Memory Footprint | MB - GB per instance | MB max (highly lightweight) |
| Global Failover | Requires complex DNS/Load Balancer setup | Natively distributed across + locations |
| Geographic Location | Fixed data center | Automatically routes to nearest user node |

---

## 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

. 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 `d-worker` SQLite database.
. 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).
. The V engine performs all signature calculations and database writes at the edge gateway node, reducing the final API transit time to under  milliseconds.

---

## HARDWARE-LEVEL SECURITY

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

- Isolate Protection: Memory leaks, side-channel attacks, and adjacent tenant exploits are prevented at the V compiler level.
- Microsecond Service Bindings: Communication between workers is processed entirely inside the local V runtime, meaning sensitive exchange payloads and API calls never travel over the public internet.

---

> Tip: 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](cloudflare-services.md) - Learn how D SQLite databases, KV, and Queues fit together on the edge.
- [AI Risk Manager](ai-risk-manager.md) - Understand how autonomous risk checks run on global cron schedules.

[ SECTION: CLOUDFLARE SERVICES MAP ]

# CLOUDFLARE SERVICES MAP

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 $ monthly costs on the free tier.

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

---

## . D 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  milliseconds because the database runs in the same V 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).

---

## . 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 -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  seconds. You can toggle settings in your terminal, and the change affects every worker globally in seconds, without code redeployment.

---

## . 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 (`s` `m` `m` `m` `m`). Your trades never get lost due to internet dropouts or API errors.

---

## . 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.

---

## . R OBJECT STORAGE (ZERO-EGRESS ASSET VAULT)

- Concept: A fully S-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 R saves massive write capacity on your D 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.

---

## . 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 , 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.

---

## . 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 D database P&L records, renders a beautiful, secure HTML dashboard report, captures it via Puppeteer, converts it to PDF, saves the file to R, and links it directly in your Telegram alert.
  - Completely automated, running entirely in the background near-instantaneously.

---

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

## NEXT STEPS

- [Idempotency Mechanics](idempotency.md) - Dive into the V Durable Objects coordination engine.
- [AI Risk Manager](ai-risk-manager.md) - Understand how autonomous edge cron jobs evaluate drawdowns and manage trailing stops.

**[ SECTION: IDEMPOTENCY & DURABLE OBJECTS ]**

# IDEMPOTENCY & DURABLE OBJECTS

In automated financial systems, execution integrity is everything. If a network dropout occurs at the exact millisecond after your gateway submits an order to an exchange but before the exchange sends back a confirmation, a standard system faces a dilemma:

- If it assumes the order failed and retries, it risks executing duplicate trades (e.g. accidentally buying the same spot position twice, doubling leverage, and exposing your account to high liquidation risks).
- If it assumes the order succeeded and does nothing, it risks missing critical trade entries.

Hoox solves this problem natively at the edge gateway layer using Cloudflare Durable Objects to enforce an absolute exactly-once execution policy (idempotency).

---

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

---

## 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 'bdebd...' been seen?"          |
   |-------+-----------------------------------------+
           |
           |- [YES: Duplicate Detected] ---------------------+
           |                                                 |
           |
           |                                      [Silent Dropping of Request]
           |                                                 |
           |
           |                                      [Return  Conflict Response]
           |
           |- [NO: Unique Transaction] ----------------------+
                                                             |

                                                  [Record ID in SQLite Log]
                                                             |

                                                  [Process Pipeline Execution]
                                                             |

                                                  [Set TTL-based DO Alarm]
```

---

## THE DEDUP & CLEANUP ALGORITHM

### . 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.

---

## . ATOMIC EVALUATION

Before executing any order routing:

. The gateway routes the request to the namespace-mapped Durable Object using the transaction ID as the binding key.
. 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.
. If the ID exists in the DO's SQLite log, the DO immediately intercepts the request and throws a ` Conflict` exception, stopping the pipeline before hitting exchange APIs.
. If unique, it registers the ID, saves the current timestamp, and returns a lock approval.

---

## . 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  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 % protected against duplicates during network dropouts, your storage footprint remains lightweight.

---

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

## NEXT STEPS

- [Signals & Trade Specifications](signals-and-trades.md) - Learn how to configure your Pine Script webhook payloads to transmit unique idempotency keys.
- [Platform Security Guides](../guides/secrets-security.md) - Deepen your understanding of Zero Trust headers and edge firewall configurations.

[ SECTION: SIGNALS & TRADE SPEC ]

# SIGNALS & TRADE SPEC

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

---

## . 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:

```
export interface WebhookSignal {
  apiKey: string; // Authentication token
  exchange: "binance" | "bybit" | "mexc"; // Target exchange
  action: "LONG" | "SHORT" | "CLOSE"; // Position intent
  symbol: string; // Asset symbol (e.g. "BTCUSDT")
  quantity: number; // Order size (amount of asset or contracts)
  leverage?: number; // Optional leverage multiplier (default: )
  idempotencyKey?: string; // Optional unique signature for dedup
}
```

### PARAMETER RULES & TYPE CONSTRAINTS

- `apiKey`: Must match the encrypted `webhooks:api_key` hash in your `CONFIG_KV` namespace.
- `symbol`: Parsed and standardized by Hoox to match exchange requirements. For example, Hoox automatically converts variations like `BTC-USDT`, `BTC_USDT`, and `btc/usdt` to the exchange-compliant flat uppercase format `BTCUSDT`.
- `quantity`: Verified to be greater than zero.
- `leverage`: Checked against exchange limits (e.g. `` to `` depending on exchange margin profiles).

---

## . 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. DYNAMIC POSITION RESOLUTION

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

. Queries your local D transaction ledger to resolve the active position direction for the symbol.
. If no position is tracked locally, it performs a real-time portfolio balance check against the exchange's private position endpoint.
. 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.

---

## . LEVERAGE SCALING & ORDER MATH

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

. Set Margin Mode: Configures the symbol's margin structure (Isolated vs. Cross) via the exchange API, matching your manifest defaults.
. Set Leverage Coefficient: Submits a leverage update payload to the exchange prior to routing the order.
. Calculate Collateral: If the order size is defined in USDT terms, the worker scales the execution quantity mathematically:
$$\text{Contract Quantity} = \frac{\text{Order Size (USDT)} \times \text{Leverage}}{\text{Current Market Price}}$$
. Precision Rounding: Automatically rounds the calculated quantity down to match the exchange's strict asset decimal precision requirements, preventing API rejects.

---

## . D 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 generated by trade-worker
  request_id TEXT NOT NULL,           -- Distributed trace ID mapping to 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"
  quantity REAL NOT NULL,             -- Filled asset quantity
  price REAL NOT NULL,                -- Execution entry price
  fee REAL NOT NULL,                  -- Exchange transaction fees
  order_id TEXT NOT NULL,             -- Exchange-provided order hash
  status TEXT NOT NULL,               -- "Filled", "Rejected", "Failed"
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

You can view this ledger at any time from your local terminal:

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

---

> Tip: By utilizing time-series logging via Cloudflare Analytics Engine, Hoox tracks the exact execution duration of these steps. You can audit the latency breakdown (e.g., Gateway auth took `.ms`, Trade-worker signing took `.ms`, Bybit API roundtrip took `.ms`) directly in your Next.js dashboard!

## NEXT STEPS

- [Autonomous AI Risk Management](ai-risk-manager.md) - Learn how agent-worker tracks these D entries to manage trailing stops and drawdowns.
- [CLI Reference Manual](../reference/cli-commands.md) - Review commands to manage D tables, perform dry runs, and tail logs.

**[ SECTION: AI RISK MANAGER & AI GATEWAY ]**

# AI RISK MANAGER & AI GATEWAY

The `agent-worker` is the autonomous central nervous system of the Hoox trading platform. Operating on a continuous -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.

---

## . AUTOMATED RISK PROTECTION ENGINE

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

### 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 V compiler level:

- The Formula:
  $$\text{Trailing Stop Price (Long)} = \text{Peak Price} \times ( - \text{Trailing Deviation \%})$$
- If a LONG trade entry is at $\$$ and has a $\%$ trailing deviation, the initial stop-loss is placed at $\$$.
- If the market price rallies to a peak of $\$$, the stop-loss is automatically adjusted up to $\$.$.
- If the price declines from $\$$ down to $\$.$, the stop is triggered at the exchange level, locking in a $\$.$ 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  ($\%$ Profit): Automatically flattens $\%$ of open position size.
- Target  ($\%$ Profit): Flattens another $\%$ and moves the stop-loss of the remaining $\%$ 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:

. Calculates cumulative daily loss:
$$\text{Daily Drawdown \%} = \frac{\text{Starting Daily Balance} - \text{Current Balance}}{\text{Starting Daily Balance}} \times $$
. If the drawdown exceeds your KV threshold (e.g. `trade:max_daily_drawdown_percent` is ``), the agent-worker immediately triggers the Global Kill Switch by writing `trade:kill_switch = true` to `CONFIG_KV`.
. 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
```

---

## . MULTI-PROVIDER AI GATEWAY & REASONING

Behind the scenes, `agent-worker` governs a Multi-Provider AI Gateway (supporting  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]
       |

[. Cloudflare Workers AI] (Zero cost, native LLaMA ) --> Fail? --+
                                                                 |

[. Anthropic API] (Claude . Sonnet) --------------------------+
       |
     Fail? --> [. Google AI] (Gemini . Pro) --> Fail? --> [. OpenAI] (GPT-o)
```

### 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 Base images.
- `POST /agent/reasoning` - Interfaces with advanced reasoning models (like OpenAI `o` or DeepSeek) to audit trading strategies and margin structures.
- `GET /agent/usage` - Provides detailed token counting and cost metrics across all providers.

---

> Tip: 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](../guides/secrets-security.md) - Lock down your AI endpoints and exchange API keys.
- [TUI Operations Cockpit](../guides/tui.md) - View live position drawdowns and trailing stops visually in your terminal.

[ SECTION: LOCAL DEVELOPMENT ]

# LOCAL DEVELOPMENT

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.

> Tip: 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 : Workers Only (No frontend dashboard)
docker compose --profile workers up

  Profile : Dashboard Only (Next.js server only)
docker compose --profile dashboard up

  Profile : Full Stack (All workers + Next.js dashboard)
docker compose --profile full up
```

---

## 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` | `` | `http://localhost:` | Public Gateway & Webhook Receiver |
| `trade-worker` | `` | `http://localhost:` | Trade Execution Engine |
| `telegram-worker` | `` | `http://localhost:` | Telegram Bot Alerts & Commands |
| `d-worker` | `` | `http://localhost:` | SQLite Database Operations |
| `web-wallet` | `` | `http://localhost:` | On-Chain DeFi Execution |
| `dashboard` | `` | `http://localhost:` | Next.js Dashboard Cockpit |
| `agent-worker` | `` | `http://localhost:` | AI Risk Manager & Cron 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:

. Lint Check (`bun run lint`): Validates ESLint styling rules across the monorepo.
. Type Check (`bun run typecheck`): Compiles code via `tsc --noEmit` to verify type safety.
. Unit Tests (`bun test`): Fires all unit and integration test assertions using Bun's native test runner.
. 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] (, assertions)
 [STARTED] workspace builds... [PASSED]
  Local CI Pipeline Succeeded!
```

---

> Note: 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

- [Terminal UI Operations](tui.md) - Launch the full-screen terminal cockpit (`./hoox-tui`) to monitor these local runs.
- [Database Migrations](database-ops.md) - Set up, query, and migrate your local SQLite D database.

**[ SECTION: TERMINAL UI (TUI) ]**

# TERMINAL UI (TUI)

The Hoox Terminal User Interface (TUI) is a full-screen, keyboard-driven operations cockpit built natively with OpenTUI, React , and Zustand state stores. Designed for command-line efficiency, the TUI provides real-time visibility into your entire distributed trading infrastructure-allowing you to inspect compute instances, tail logs, query D databases, manage configurations, and deploy edge workers in seconds.

---

## LAUNCHING THE TUI

Launch the cockpit directly from your terminal using any of the following methods:

```
  . Recommended: Shell script launcher (from project root)
 ./hoox-tui

  . Recommended: Via the hoox CLI tool
 hoox tui

  . Development: Run direct hot-reload development server
 cd packages/tui && bun run dev

  . Flags: Override frame rate and disable mouse interaction
 hoox tui --fps  --no-mouse
```

---

## GLOBAL KEYBOARD NAVIGATION CHEATSHEET

The TUI features a keyboard-first layout. All shortcuts are registered globally and can be triggered from any active view:

| Keyboard Shortcut | Operation | View Name |
|---|---|---|
| `Ctrl+` | Jump to View | Dashboard - System Health Overview |
| `Ctrl+` | Jump to View | Workers Overview - -Column Cards Grid |
| `Ctrl+` | Jump to View | Worker Detail - Deep-Dive Metrics |
| `Ctrl+` | Jump to View | Trade Monitor - Real-Time Transaction Feed |

| `Ctrl+` | Jump to View | Logs Viewer - Streaming Log Console |
| `Ctrl+` | Jump to View | Service Manager - Deploy & Rebuild Controls |
| `Ctrl+` | Jump to View | Config Editor - Syntax-Highlighted TOML/JSON Editor |
| `Ctrl+` | Jump to View | Setup Wizard - Onboarding Flow |
| `Ctrl+` | Jump to View | Settings - UI Preferences & theme toggle |
| `Ctrl+P` | Action | Command Palette - Fuzzy-search search overlay |
| `Ctrl+B` | Action | Toggle Sidebar - Show/Hide menu |
| `Ctrl+R` | Action | Force Refresh - Recalculate metrics |
| `Ctrl+Q` | Action | Quit - Displays exit confirmation dialog |
| `Esc` | Action | Close overlay / Dismiss modal / Go back |
| `Tab` / `Shift+Tab` | Navigation | Cycle active focus between elements |
| `Enter` | Navigation | Select / Execute / Toggle button |

---

## COCKPIT TOUR: THE CORE VIEWS

### . THE OPERATIONS DASHBOARD (`CTRL+`)

Your high-signal, single pane of glass.

- Service Status Grid: Operational, Degraded, or Offline indicators for all edge workers.
- Telemetry Statistics: Session P&L, cumulative win rates, Sharpe ratio, and total API counts.
- Alert Box: Color-coded, scrolling logs of warnings, drawdowns, and security events.

---

### . WORKERS OVERVIEW (`CTRL+`)

A cards-based visual grid representing every running isolate.

- Displays worker status, uptime metrics, CPU cycles (ms), and active RAM footprint (MB).

- Shows active Durable Object instances (locks) and the number of processed signals.
- Selecting a worker and hitting `Enter` opens the Worker Detail deep-dive.

---

### . WORKER DETAIL (`CTRL+`)

A -quadrant analytical layout focused on a single worker:

- Top Left: Metrics: Graphs representing memory allocation and request velocities over time.
- Top Right: Live Logs: Scrollable, real-time log terminal (press `p` to pause logging).
- Bottom Left: Durable Objects: SQLite state indexes and active DO alarm TTL counts.
- Bottom Right: Config Preview: Snapshot of active `wrangler.jsonc` binds.

---

### . TRADE MONITOR (`CTRL+`)

The financial ledger center.

- Live Feed: Scrolling record of recent fills (Symbol, Side, Execution Price, Contracts, P&L).
- Open Positions: Grouped active exposure showing current size, entry price, liquidation boundaries, and real-time unrealized P&L.
- Win Metrics: Graphical gauges for win/loss ratio, average trade duration, and daily performance metrics.

---

### . LOGS VIEWER (`CTRL+`)

A centralized console debugger:

- Filter Panel: Multi-select checkboxes for severity filters (`INFO`, `WARN`, `ERROR`), target worker select, and a fuzzy text search box.
- Log Stream: Scrolling terminal window showing formatted, color-coded lines.

---

### . SERVICE MANAGER (`CTRL+`)

Your deployment pipeline:

- Worker Controls: Trigger direct deployments (`hoox deploy`) or soft restarts (`wrangler dev` resets) on individual workers.

- Interactive Edge Map: Displays locations of Cloudflare Points of Presence (PoPs) globally with status checks.
- Bulk Executions: Re-deploy the entire ecosystem in one click, secured by confirmation modals.

---

### . CONFIGURATION EDITOR (`CTRL+`)

An IDE-class terminal file editor:

- Syntax Highlighting: Supports full color-coded rendering for TOML and JSON structures.
- Live Linter: Evaluates brackets, balancing quotes, and JSON structures on the fly, showing error locations.
- Prettifier: Automatically formats code blocks (spacing, indentation) on save (`Ctrl+S`).

---

### . GUIDED SETUP WIZARD (`CTRL+`)

Onboards new machines in  steps:

. API Keys: Authenticate Cloudflare credentials.
. Exchanges: Inject Bybit, Binance, or MEXC trade API keys.
. AI Credentials: Set up multi-provider keys (OpenAI, Gemini, Anthropic).
. Strategies: Configure default margins and symbols.
. Telegram Bot: Link tokens and Chat IDs.
. Kickoff: Initiate deployments.

---

### . COCKPIT SETTINGS (`CTRL+`)

Configure cockpit parameters:

- Theme: Swap color themes in real-time.
- Telemetry: Set metrics refresh rates (default: every  seconds).
- Reset: Clean local TUI caching (`~/.hoox/session.json`).

---

## OFFLINE CONNECTION RESILIENCE STORE

The TUI utilizes an advanced Zustand state machine to govern connectivity to your local API dev server or remote workers:

```
  [Connected (OK)] ----> [Connection Dropped]
                                |
        |
  [State Restored] --- [Reconnecting (Exponential Backoff)]
```

- States: `Connected` (Green dot), `Reconnecting` (Yellow/Orange pulsing dot), `Offline` (Red/Empty dot).
- Resilience: In the event of network dropouts, the TUI activates an exponential backoff engine (starting at `ms`, doubling up to `s`), trying to reconnect in the background without freezing or crashing the terminal display.

---

## TROUBLESHOOTING & TERMINAL COMPATIBILITY

### ALTERNATE SCREEN BUFFER CLEANUP

If you force-close the terminal and find that your prompt remains garbled, execute a terminal reset:

```
  reset
   or
  tput reset
```

### "CTRL+Q" IS INTERCEPTED BY FLOW CONTROL

If `Ctrl+Q` (Quit) fails to trigger, your terminal emulator is likely intercepting flow control commands (XON/XOFF). Disable software flow control by running this command before launching the TUI:

```
  stty -ixon
```

### GARBLED LAYOUT OR TEXT OVERLAPS

The TUI requires a minimum screen resolution of  columns  rows. If your terminal is too small, resize the window. Additionally, ensure your system `TERM` variable is configured to support  colors:

```
  export TERM=xterm-color
```

## NEXT STEPS

- [Local Development Guides](local-development.md) - Spin up the API dev server that feeds the TUI.
- [CLI Reference Manual](../reference/cli-commands.md) - Read the commands running under

the TUI hood.

[ SECTION: DATABASE OPERATIONS ]

# DATABASE OPERATIONS

Hoox utilizes Cloudflare D-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:

. `trades`: The primary ledger. Stores execution prices, contract quantities, timestamps, transaction fees, and exchange order IDs.
. `positions`: Tracks open margin/futures exposure (average entry price, size, leverage, direction).
. `balances`: Periodic snapshots of account equity, margin balance, and available free collateral.
. `trade_signals`: Historical record of every raw incoming webhook alert before execution processing.
. `system_logs`: Crucial debug and error messages offloaded from compute nodes.

---

## APPLYING SCHEMAS & TABLES

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

```
  . Apply schema and seed initial tables to local dev SQLite
hoox db apply

  . Deploy schema and seed tables directly to live production Cloudflare D
hoox db apply --remote
```

> Warning: 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 D 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 D
hoox db migrate --remote
```

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

---

## INSPECTING & 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  most recent trade fills with formatting
hoox db query "SELECT created_at, symbol, action, price, quantity FROM trades ORDER BY created_at
DESC LIMIT " --remote

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

---

## BACKUP & EXPORT WORKFLOWS

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

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

### EXPORT OUTPUT & STRUCTURE

The export command creates a timestamped SQL dump inside your workspace directory: `backups/db-backup----.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;
```

---

## 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 D database (USE WITH EXTREME CAUTION)
hoox db reset --remote --confirm
```

> Danger: Wiping your production D 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!

## NEXT STEPS

- [Local Development & Testing](local-development.md) - Run your local V wrangler isolates with local D SQLite bindings.
- [Infrastructure Management](manage-infra.md) - Manage KV config namespaces, Queue parameters, and R storage buckets.

**[ SECTION: SECRETS & NETWORK SECURITY ]**

# SECRETS & NETWORK SECURITY

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.

---

## . 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)
                                  |
                 +----------------+----------------+
                 | V Isolate Service Bindings     |
                 | (Private, Encrypted, Zero-TCP)  |
                 |----------------+----------------+
                                  |
                +-----------------+-----------------+

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

---

## . 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:

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

  . 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)    |
|-------------------------------------------------------+
```

---

## . WEBHOOK FIREWALL & TRADINGVIEW IP ALLOW-LISTING

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

. Passkey Verification: The gateway checks the `apiKey` property inside the JSON payload against your secure manifest in `CONFIG_KV`.
. 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
```

---

## . 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  days. Deleting old keys and injecting new ones takes less than  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:
  ```bash
  hoox monitor kill-switch on
  ```

## NEXT STEPS

- [Astro Docs Site Config](../getting-started/configuration.md) - Map out your build-time environment configurations.
- [Local Development & Testing](local-development.md) - Run local wrangler sandboxes with securely encrypted `.dev.vars`.

[ SECTION: INFRASTRUCTURE MANAGEMENT ]

# INFRASTRUCTURE MANAGEMENT

Hoox is fully serverless, using Cloudflare's distributed services as its compute and storage engine. Managing these resources-such as provisioning D databases, registering KV config buckets, setting up S-compatible R 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.

---

## . 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, D, R, Vectorize, and Queue resources
hoox infra provision
```

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

. D SQLite Database (`hoox-db`).
. CONFIG_KV Namespace bucket.
. `trade-execution` messaging Queue.
. `trade-reports` and `system-logs` R object storage buckets.
. `rag-index` Vectorize vector search database.

---

## . D DATABASE ADMINISTRATION

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

```
  A. List all active D databases in your Cloudflare account
hoox infra d list

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

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

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

---

## . 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 -key configuration manifest to your active namespace**
```
hoox config kv apply-manifest
```

---

## . 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:  seconds initial delay, scaling exponentially up to  minutes.
- Message Retention:  days (ensuring messages are preserved even during prolonged exchange maintenance events).

---

## . R OBJECT STORAGE ADMINISTRATION

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

**A. List all R buckets**

```
hoox infra r list
```

**B. Create the trade-reports bucket**

```
hoox infra r create trade-reports
```

---

## . 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.  for OpenAI embeddings)**

```
hoox infra vectorize create rag-index --dimensions  --metric cosine
```

---

> Warning: Destroying resources via `hoox infra <service> delete` is highly destructive and irreversible. Deleting a D 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](database-ops.md) - Run queries, manage drizzle schemas, and restore backups.
- [Take-to-Production Deployments](deploy-workers.md) - Deploy your compiled workers and bind V isolates globally.

**[ SECTION: DEPLOYING TO PRODUCTION ]**

# DEPLOYING TO PRODUCTION

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

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:

```
[. D Database Schema] --> [. d-worker (SQL Hub)] --> [. trade-worker (Execution Engine)]
                                                                                |
    +-------------------------------------------------------------+

[. hoox Gateway (Public Router)] --> [. agent-worker & telegram-worker]
                                                        |

                                     [. Next.js Dashboard (OpenNext)]
```

---

## DEPLOYMENT CLI COMMANDS

To execute a complete production rollout:

```
  . Compile and deploy all enabled workers in the correct dependency order
hoox deploy all --auto

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

### CRITICAL POST-DEPLOYMENT AUTOMATION

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
```

---

## NEXT.JS DASHBOARD DEPLOYMENT (OPENNEXT)

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

When you run `hoox deploy dashboard`:

. The CLI compiles the Next.js app using the Turbopack build engine.
. The OpenNext compiler bundles the application logic into a single Edge-compliant V worker file: `.open-next/worker.js`.
. All static files (images, JS chunks, CSS) are isolated under `.open-next/assets/`.
. 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
```

---

## ROUTINE UPDATE & UPGRADE WORKFLOW

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

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

  . Install updated dependencies
bun install

  . Execute the full local CI verification pipeline
hoox test

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

---

## POST-DEPLOYMENT TROUBLESHOOTING MATRIX

| Error Code / Symptom | Primary Root Cause | Guided Resolution |
| --- | --- | --- |
| `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. |
| `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`. |
| `Unauthorized` | Webhook request failed passkey check. | Audit KV authorization key: `hoox config kv get webhooks:api_key`. Verify the payload `apiKey` matches this value. |
| `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. |

---

> Tip: 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](monitor-trading.md) - Audit live fills, tail console logs, and run health diagnostics.
- [System Self-Healing & Repair](repair.md) - Rebuild broken bindings and recover from deployment anomalies.

**[ SECTION: SELF-HEALING & REPAIR ]**

# SELF-HEALING & REPAIR

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

---

## . RUNNING THE -STEP SYSTEM DIAGNOSTICS

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

```
 Execute the automated -step repair check
hoox repair check
```

The diagnostics engine runs a strict, sequential analysis of your entire monorepo environment:

```
+-------------------------------------------------------+
|              hoox Repair Diagnostic Checklist         |
|-------------------------------------------------------
|  [Step ] Submodule Integrity .........  COMPLETED   |
|  [Step ] Dependency Resolutions .......  COMPLETED   |
|  [Step ] TypeScript Type Safety ......  COMPLETED   |
|  [Step ] Cloudflare Edge Bindings ....  COMPLETED   |
|  [Step ] Hardware Secrets Validation .  COMPLETED   |
|                                                       |
|  Diagnostic Result:  errors detected                 |
|-------------------------------------------------------+
```

. Submodule Check: Verifies that all  worker folders under `workers/` are fully populated and registered in Git.
. Dependency Check: Audits `node_modules` across Bun workspaces for missing or duplicate libraries.
. Type Safety Check: Compiles the CLI and shared packages to catch syntax or interface errors.
. Edge Bindings Check: Connects to Cloudflare's APIs to verify that your D, KV, and Queue namespaces physically exist on your account.
. Secrets Check: Audits your Workers Secrets to ensure exchange keys and Telegram tokens are securely bound.

---

## . TARGETED COMPONENT REPAIRS

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

```
  A. Re-verify, provision, and repair missing Cloudflare bindings
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 D database migrations
hoox repair db

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

---

## . FULL SYSTEM INTERACTIVE REBUILD

In the event of a catastrophic system failure (e.g. lost Cloudflare credentials, corrupted SQLite files, or major monorepo drift), you can execute a full, interactive self-healing rebuild:

```
  Execute an interactive, guided workspace rebuild
hoox repair rebuild
```

### THE REBUILD SEQUENCE

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

. D Database Backup: Automatically exports your current D ledger as a secure `.sql` file in `backups/recovery-pre-rebuild.sql`.
. Infrastructure Tear-Down: Deletes corrupted D, KV, and Queue instances.
. Fresh Provisioning: Recreates all database tables and config buckets on Cloudflare.
. Sequenced Redeployment: Re-compiles and deploys all  edge workers in correct dependency sequence.
. Database Seeding: Restores your backup SQL data and re-applies schema migrations.
. KV Sync: Applies the -key configuration manifest defaults.

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

---

## . COMMON TROUBLESHOOTING SCENARIOS

| Symptom / Failure | Primary Root Cause | CLI Healing Action |
|---|---|---|
| `bun install` or build crashes | Git submodules are empty or out of sync. | `git submodule update --init --recursive` |
| `D_ERROR: no such table` | SQLite D tables were not initialized. | `hoox db apply --remote` |
| ` Internal Server Error` | Missing or rotated exchange API secrets. | `hoox secrets set BYBIT_API_KEY "key"` followed by `hoox deploy update-internal-urls`. |
| Telegram Bot is mute | BotFather token is wrong, or webhook is unregistered. | `hoox secrets set TELEGRAM_BOT_TOKEN "token"` followed by `hoox deploy telegram-webhook`. |

## NEXT STEPS

- [Monitoring Operations Guide](monitor-trading.md) - Audit system status, tail console logs, and verify fills.
- [CLI Reference Manual](../reference/cli-commands.md) - Review all CLI commands, options, and JSON flags.

[ SECTION: TRADINGVIEW WEBHOOK INTEGRATION ]

# TRADINGVIEW WEBHOOK INTEGRATION

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 v logic.

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

---

## STEP : WRITING A PINE SCRIPT V SIGNAL INDICATOR

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

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

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

// . 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")

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

plotshape(longCondition, title="Long Cross", style=shape.triangleup, location=location.belowbar,
color=color.green, size=size.small)
plotshape(shortCondition, title="Short Cross", style=shape.triangledown,
location=location.abovebar, color=color.red, size=size.small)

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

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

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

> Tip: 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 ` Unauthorized` error.

---

## STEP : CREATING THE TRADINGVIEW WEBHOOK ALERT

Once your script is saved and added to your chart:

. Click the Alarm Clock (Alerts) icon on the top right panel in TradingView.
. 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).
. Under Expiration: Select your alert lifespan (Premium accounts support Open-Ended alerts).
. Navigate to the Notifications Tab:
   - Check the Webhook URL box.
   - Paste your public Hoox Gateway endpoint URL:
     `https://hoox.alpha-trading.workers.dev/webhook`
. 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).
. Click Create.

---

## STEP : VERIFYING EXECUTION IN PRODUCTION

When the Moving Average crossover occurs on your chart:

. TradingView compiles the JSON payload and POSTs it to your edge gateway.
. The `hoox` gateway validates the signature and routes the order to Bybit.
. Check the transaction directly in your terminal:
   ```bash
   hoox monitor trades
   ```
. Stream live gateway telemetry logs to verify execution latency:

```bash
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. |
| ` 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. |
| ` 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](telegram-bot.md) - Integrate Telegram notifications to get fills pushed directly to your phone.
- [API Endpoint Reference](../reference/api-endpoints.md) - Review full request schemas and status codes.

**[ SECTION: TELEGRAM BOT SETUP ]**

# TELEGRAM BOT SETUP

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

. Push Notifications: Sends instant real-time order fill alerts, daily P&L audits, and system health status updates directly to your mobile phone.
. Interactive Command Console: Provides a secure chat terminal allowing you to query open positions, check D 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 : PROVISION A BOT VIA BOTFATHER

. Open the Telegram app and search for the verified `@BotFather` account.
. Click Start and send the command:
   ```telegram
   /newbot
   ```
. Enter a friendly name for your bot (e.g. `My Hoox Edge Bot`).
. Choose a unique username ending in "bot" (e.g. `AlphaHooxTradeBot`).
. BotFather will output your HTTP API Token (save this securely):
   `:AAFkLjL-kLl...`

---

### STEP : 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:

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

---

### STEP : 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 ":AAFkLjL-kLl..."

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

---

### STEP : 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 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 D SQL database, aggregates context using Vectorize, and responds with a LLaMA- 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 . Sonnet or Gemini . Pro) to analyze the chart and return an automated risk audit.

---

> Warning: 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 % secure.

## NEXT STEPS

- [Astro Docs Site Config](../getting-started/configuration.md) - Review your system environment configurations.
- [Real-Time Observability & Monitoring](../guides/monitor-trading.md) - Stream console logs and check queue depths.

**[ SECTION: EMAIL SIGNAL ROUTING ]**

# EMAIL SIGNAL ROUTING

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.

---

## . 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:

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

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

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

> Warning: 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.

---

## . 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:

. The worker matches the subject. If the subject is `"TRADE SIGNAL: Breakout Detected"`, the check passes.
. 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`).
. Automatically triggers an order via the `trade-worker` service binding.

---

## . 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.

---

## . TESTING YOUR EMAIL PIPELINE

To verify that the email signal ingestion loop works perfectly:

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

### NEXT STEPS

- [Real-Time Observability & Monitoring](../guides/monitor-trading.md) - Audit executed

fills, check logs, and inspect queues.
- [API Endpoint Reference](../reference/api-endpoints.md) - Review full REST payloads and HTTP error returns.

**[ SECTION: CLI COMMANDS REFERENCE ]**

# CLI COMMANDS REFERENCE

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.

---

## 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 monitor status --json` |
| `--quiet` | Suppresses headers, banners, and interactive prompts. | `hoox deploy kv-config --quiet` |
| `--help` | Prints complete parameter and option listings. | `hoox infra d --help` |
| `--version` | Outputs active CLI package build version. | `hoox --version` |

---

## COMMAND GROUPS DIRECTORY

```
hoox                                Launch TUI (when run with no args)
|-- init                            Interactive multi-phase setup wizard
|-- 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
|   |-- 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
|-- infra                           Cloudflare resource provisioning IaC
|   |-- provision                   Auto-provision all required services
```

```
|   |-- d [list/create/delete]      Manage D databases
|   |-- kv [list/create/delete]      Manage KV namespaces
|   |-- r [list/create/delete]      Manage R object storage buckets
|   |-- queues [list/create/delete]  Manage Cloudflare Queues
|   |-- vectorize [create/delete]    Manage vector search indexes
|-- config                           Workspace and environmental variables
|   |-- env [init/show/validate]     Manage .env.local build variables
|   |-- kv [set/get/list/delete]     Get/Set dynamic KV runtime parameters
|   |-- show/set                     View or set wrangler.jsonc values
|-- check                            Toolchain and route diagnostic suite
|   |-- prerequisites                Validate local machine tool installs
|   |-- setup                        Audit .env configurations
|   |-- health                       Probe worker /health endpoints
|-- db                               SQLite D Database administration
|   |-- apply                    Apply DDL schemas to D
|   |-- 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
|   |-- status                       Probe active worker health routes
|   |-- 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
|-- repair                           System diagnostics & self-healing
|   |-- check                        Run -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
|-- test                             Run verification pipeline (CI)
|-- waf                              Auto-configure Cloudflare WAF firewall rules
```

---

## KEY COMMANDS DETAIL & EXAMPLES

### A. INITIALIZATION & BOOTSTRAP

Onboard a new machine, configure accounts and select worker profile

```
hoox init
```

```
  Verify that git, bun, wrangler, and docker are correctly configured
hoox check prerequisites
```

### B. LOCAL DEVELOPMENT

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

  Start only the Next.js frontend developer server
hoox dev dashboard
```

### 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
```

### D. OPERATIONAL MONITORING & EMERGENCY HALTING

```
  Emergency Halt: halt all trade signals globally in under  seconds
hoox config kv set trade:kill_switch true

  Audit active trade history table
hoox monitor trades
```

### E. 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
```

### F. SELF-HEALING & DIAGNOSTICS

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

---

> Tip: 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

- [Astro Docs Site Config](../getting-started/configuration.md) - Map out your build-time environment configurations.

- [API Endpoint Reference](api-endpoints.md) - Analyze REST routes, schemas, and gateway error structures.

## [ SECTION: API ENDPOINT SPECIFICATION ]

# API ENDPOINT SPECIFICATION

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 V 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 V engine context.

---

## . 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_",
  "exchange": "bybit",
  "action": "LONG",
  "symbol": "BTCUSDT",
  "quantity": .,
  "leverage": ,
  "idempotencyKey": "uuid-bdebd-bd"
}
```

## RESPONSE MODELS

### A. SUCCESS PATH (ORDER FILLED INSTANTLY - OK)

```
{
  "success": true,
  "requestId": "bdebd-bd-bad-bdd-bdbdcbd",
  "exchange": "bybit",
  "symbol": "BTCUSDT",
  "action": "LONG",
  "result": {
    "orderId": "",
    "status": "Filled",
    "executedQty": .,
    "price": .,
    "timestamp":
  }
}
```

### B. QUEUE FAILOVER PATH (EXCHANGE OFFLINE / RATE-LIMITED - ACCEPTED)

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

```
{
  "success": true,
  "requestId": "bdebd-bd-bad-bdd-bdbdcbd",
  "status": "Enqueued",
  "message": "Signal enqueued in trade-execution Queue for background execution retry."
}
```

---

## . SYSTEM HEALTH CHECK

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

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

### SUCCESS RESPONSE ( OK)

```
{
  "status": "ok",
  "timestamp": ,
  "bindings": {
    "d": "connected",
    "kv": "connected",
```

```
      "queue": "active"
    }
  }
```

---

## . 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  hours.",
    "stream": true,
    "provider": "anthropic"
  }
```

---

### B. AI GATEWAY TELEMETRY

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

### RESPONSE PAYLOAD ( OK)

```
  {
    "success": true,
    "timestamp": ,
    "usage": {
      "openai": { "inputTokens": , "outputTokens": , "costUSD": . },
      "anthropic": {
        "inputTokens": ,
        "outputTokens": ,
        "costUSD": .
      },
      "workersAi": { "neuronsUsed": , "costUSD": . }
    }
  }
```

---

## . 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. 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. UNAUTHORIZED (INVALID API KEY / IP ADDRESS)

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

### C. CONFLICT (DUPLICATE REQUEST INTERCEPTED)

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

### D. 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 in CONFIG_KV."
    }
  }
```

---

> Tip: 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

- [Astro Docs Site Config](../getting-started/configuration.md) - Map out your build-time environment configurations.
- [CLI Commands Reference](cli-commands.md) - Review all CLI commands, options, and JSON flags.

[ SECTION: CONFIGURATION DICTIONARY ]

# CONFIGURATION DICTIONARY

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.

---

## . ENVIRONMENT VARIABLES & SECRETS MATRIX ( 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, D, and KV read/write permissions. |
| `CLOUDFLARE_ACCOUNT_ID` | Yes | `string` | - | Your -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-SHA 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 & WEB WALLET SETTINGS

| Variable | Required | Type | Scope | Description |
| --- | --- | --- | --- | --- |
| `ETH_MNEMONIC` | No | `string` | `web-wallet` | Secure  or -word seed phrase for on-chain contract swaps. |
| `RPC_PROVIDER_URL` | No | `string` | `web-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` | POP/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. |

---

## . DYNAMIC KV MANIFEST SETTINGS ( 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` | `.` | 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\|ETH)` | Regex expression used to extract asset tokens from email bodies. |
| `email:action_pattern` | `string` | `(BUY\|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. |

---

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

## NEXT STEPS

- [-Minute Quick Start Guide](quick-start.md) - Deploy all workers and fire your first simulated webhook.
- [API Endpoint Reference](api-endpoints.md) - Review full REST schemas, parameters, and error models.