HOOX TRADING PLATFORM

DEVOPS INFRASTRUCTURE, BINDINGS & SYSTEM RUNBOOKS

Generated: 2026-05-19

Classification: Technical Documentation Spec

Design Style: Monospace Terminal Console layout

TABLE OF CONTENTS

[SECTION: DEVOPS MANUAL]

DEVOPS MANUAL

> Welcome to the Hoox DevOps & System Operations Manual. This manual is the primary, production-grade reference for system administrators, security engineers, and platform operators responsible for provisioning edge databases, orchestrating secure V service bindings, deploying multi-exchange execution workers, auditing secrets, and monitoring system health.

OPERATOR'S SYSTEM DIRECTORY

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

. OPERATIONS & SETUP RUNBOOKS

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

- [Operations & Troubleshooting](setup_and_operations.md) - Core operations manual, -key environment variable matrices, and diagnostic codes.
- [Core Installation Flow](installation-flow.md) - Guided, step-by-step local machine and edge provision setup.
- [Terminal UI Cockpit Development](tui.md) - Code architectures, stores, and operations of the OpenTUI monitor.

. ARCHITECTURAL SPECIFICATIONS

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

- [System Topology Overview](architecture/overview.md) - Architectural layout, regional edge clustering, and scaling limits.
- [Isolate Communication](architecture/communication.md) - Deep dive into Service Bindings, zero-TCP routing, and V engines.
- [Data Flow Architecture](architecture/data-flow.md) - Flowcharts for trade execution, backup queues, and cron risk monitoring.
- [Bindings Matrix](architecture/bindings.md) - Exact mapping of D, KV, Queues, R, and Service Bindings.
- [Storage Engineering](architecture/storage.md) - Persistent storage boundaries, SQLite DDL parameters, and R buckets.
- [Internal Endpoints Map](architecture/endpoints.md) - Sub-millisecond binding paths and routing maps.
- [Visual Tokens & Design System](architecture/design-system.md) - Monochromatic token

mappings and visual SVGs catalog.

. EDGE WORKER MICROSERVICES (PROFILES)

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

- [hoox Gateway](workers/hoox.md) [trade-worker](workers/trade-worker.md) [agent-worker](workers/agent-worker.md) [telegram-worker](workers/telegram-worker.md)
- [d-worker](workers/d-worker.md) [web-wallet-worker](workers/web-wallet-worker.md) [email-worker](workers/email-worker.md) [analytics-worker](workers/analytics-worker.md)
- [report-worker](workers/report-worker.md) [dashboard (Next.js OpenNext)](workers/dashboard.md)

. DEPLOYMENT, WAF, & CI/CD PIPELINES

Rollout manuals, Access gates, and GitHub Actions telemetry:

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

. DEVELOPER & API REFERENCE

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

- [Wrangler Dev Setup](development/local-dev.md) [Testing Standards](development/testing.md) [Debugging Runbook](development/debugging.md)
- [Exposed API Routes](api/endpoints.md) [Request Payloads](api/payloads.md) [Standard Responses](api/responses.md)
- [CLI Commands Engine](cli_features.md) - Command-line argument parsing, binary execution, and JSON flags.

> Tip: First time deploying a Hoox workspace to production? Start with the [Production Deployment Manual](deployment/production.md) to verify your Cloudflare Account permissions and execute sequencial deployments seamlessly.

QUICK LINKS & OFFLINE REFERENCE

- [End-User Documentation Hub](../home.md) - Standard setup, cURL webhooks, and TradingView Pine Scripts.
- [Hoox Git Submodules](https://github.com/jango-blockchained/hoox-setup) - Central monorepo codebase.
- [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.

[SECTION: OPERATIONS & RUNBOOK]

OPERATIONS & RUNBOOK

This manual is the primary, production-grade operations runbook for Hoox administrators. It outlines the complete system topology, toolchain prerequisites, environment variables, sequential deployment protocols, and guided troubleshooting matrices.

. COMPLETE ENVIRONMENT VARIABLES MATRIX (KEYS)

These variables are defined in `.env.local` for local building/deploying or injected as encrypted Workers Secrets for runtime isolate compute.

A. CORE PLATFORM & INFRASTRUCTURE

- `CLOUDFLARE_API_TOKEN`: Cloudflare API Token with Workers, D, and KV read/write permissions.
- `CLOUDFLARE_ACCOUNT_ID`: Your unique -character Cloudflare dashboard account hash.
- `SUBDOMAIN_PREFIX`: Subdomain prefix under which your public gateway routes compile.
- `NODE_ENV`: Target environment profile: `development`, `staging`, or `production`.
- `HOOX_API_URL`: Local API target URL (automatically configured during dev runs).

B. EXCHANGE API CREDENTIALS (ENCRYPTED SECRETS)

- `BYBIT_API_KEY` & `BYBIT_API_SECRET`: Bybit order placement account key and HMAC-SHA signature key.
- `BINANCE_API_KEY` & `BINANCE_API_SECRET`: Binance trade permission key and private HMAC signature.
- `MEXC_API_KEY` & `MEXC_API_SECRET`: MEXC trade permission key and private HMAC signature.

C. TELEGRAM BOT ALERTS & TELEMETRY

- `TELEGRAM_BOT_TOKEN`: Telegram bot token from `@BotFather`.
- `TELEGRAM_CHAT_ID`: Authorized numeric Chat ID for pushed fills and commands.

D. MULTI-PROVIDER AI CREDENTIALS

- `OPENAI_API_KEY`: OpenAI API access key.
- `ANTHROPIC_API_KEY`: Anthropic Claude API access key.
- `GOOGLE_AI_API_KEY`: Google Gemini API access key.

E. DEFI & WEB WALLET SETTINGS

- `ETH_MNEMONIC`: Secure or -word seed phrase for EVM swaps.
- `RPC_PROVIDER_URL`: HTTP Ethereum / EVM JSON-RPC provider (e.g. Infura/Alchemy).

F. EMAIL PARSING INBOX CONNECTION

- `EMAIL_HOST`: POP/IMAP mailbox server address.
- `EMAIL_USER`: Target signal mailbox email address.
- `EMAIL_PASS`: Secure app password for mailbox access.

. WORKER DEPLOYMENT SEQUENCE

Because Hoox microservices communicate internally using fast-path Service Bindings, they have strict compile-time and deploy-time dependencies. Deploy workers in the following strict sequential hierarchy:

```
. analytics-worker    (No dependencies)
. report-worker       (Depends on: analytics-worker)
. d-worker           (Depends on: analytics-worker)
. telegram-worker     (Depends on: trade-worker, hoox, analytics-worker)
. web-wallet-worker  (Depends on: telegram-worker, analytics-worker)
. email-worker        (Depends on: trade-worker, analytics-worker)
. trade-worker        (Depends on: d-worker, telegram-worker, analytics-worker)
. agent-worker        (Depends on: d-worker, trade-worker, telegram-worker, analytics-worker)
. hoox Gateway        (Depends on: trade-worker, telegram-worker, analytics-worker)
. dashboard          (Depends on: all services being live)
```

```
 Automated Sequenced Deployment via CLI
hoox deploy all --auto

 Or deploy a single specific worker
hoox deploy worker trade-worker
```

. SECRET MANAGEMENT RUNBOOK

Secrets are securely uploaded to Cloudflare's hardware key vaults using the CLI:

```
 Inject Bybit Credentials
hoox secrets set BYBIT_API_KEY "your_bybit_key"
hoox secrets set BYBIT_API_SECRET "your_bybit_secret"

 Check active secret synchronization on Cloudflare
hoox secrets check
```

. GLOBAL KILL SWITCH & EMERGENCY OPERATIONS

The Global Kill Switch is your emergency brake. Stored inside the sub-millisecond `CONFIG_KV` namespace, flipping this parameter instantly blocks all incoming trade signals globally in under seconds:

```
 View active state of the Kill Switch
hoox monitor kill-switch show

 Emergency HALT - disable all trade execution immediately
hoox monitor kill-switch on

 Resume normal operations
hoox monitor kill-switch off
```

. TROUBLESHOOTING & DIAGNOSTICS

ALT 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
```

BAD GATEWAY

- Root Cause: Service binding target is missing or has crashed.
- Fix: Ensure the dependency worker (e.g. `trade-worker`) has been deployed successfully. Run `hoox deploy all` to rebuild all bonds.

SERVICE UNAVAILABLE

- Root Cause: The Global Kill Switch is active in KV.
- Fix: Verify switch state: `hoox monitor kill-switch show`. If safe, restore trading: `hoox monitor kill-switch off`.

NEXT STEPS

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

- [Architecture & Edge Topology](architecture/overview.md) - In-depth architectural outlines and Service Bindings routing maps.

[SECTION: CLOUDFLARE WORKERS SETUP FLOW]

CLOUDFLARE WORKERS SETUP FLOW

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

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

INTERACTIVE SETUP WIZARD (`HOOX INIT`)

To start the system bootstrap, run the init wizard from the monorepo root:

```
hoox init
```

The setup wizard guides you through critical onboarding phases:

PHASE : TOOLCHAIN DIAGNOSTICS

Probes your machine to verify that essential developer tools are accessible:

- Bun: Used for monorepo package management, CLI binaries execution, and fast unit testing.
- Git: Used to verify recursive cloning of worker submodules.
- Wrangler: Cloudflare's CLI used to login, tail logs, and upload V isolates.

PHASE : GLOBAL CONFIGURATION MAPPING

Configures your master workspace credentials stored in `.env.local` and `wrangler.jsonc`:

- Cloudflare API Token: Generates and checks permissions.
- Cloudflare Account ID: Uniquely identifies your hosting space.
- Subdomain Prefix: Defines the worker routing domain (e.g. `hoox` routes to `https://hoox.alpha-trading.workers.dev`).

PHASE : MICROSERVICE PROFILE SELECTION

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

PHASE : SQLITE DATABASE PROVISIONING

Checks if enabled workers require persistent D storage. If yes, it creates the database and initializes database schemas:

```
   Done automatically by the wizard
  wrangler d create trade-data-db
```

PHASE : MANIFEST COMPILATION

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

PHASE : WORKERS SECRETS INJECTION

Secures sensitive credentials (exchange API keys, Telegram tokens, AI API keys) by encrypting and uploading them to Cloudflare as encrypted Workers Secrets.

PHASE : INITIAL ROLLOUT

Compiles, lint-checks, type-checks, and deploys all enabled workers to Cloudflare's edge in the correct mathematical dependency sequence.

CONFIGURATION FILES SPEC

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

A. WRANGLER.JSONC (CENTRAL SETTINGS)

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

workers:

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

B. .INSTALL-WIZARD-STATE.JSON (ONBOARDING STATE)

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

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

SECRET BINDINGS ARCHITECTURE

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

LOCAL MOCKING (`.DEV.VARS`)

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

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

```
BYBIT_API_SECRET=mock_bybit_development_secret
```

PRODUCTION SECRET BINDINGS

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

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

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

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

NEXT STEPS

- [DevOps Setup & Operations Manual](setup_and_operations.md) - Dive into complete operations, variable matrices, and troubleshooting.
- [Terminal UI Cockpit](tui.md) - Run, hot-reload, and monitor your local workers via TUI.

[SECTION: TUI CODE ARCHITECTURE]

TUI CODE ARCHITECTURE

The Hoox Terminal UI (TUI) is a full-screen, keyboard-driven operations center built natively using OpenTUI, React , and Zustand state stores. This document outlines the TUI code architecture, directory structure, data flows, and error recovery mechanisms for senior engineers and system operators.

ARCHITECTURAL BLUEPRINT

The cockpit is designed as a modular Single Page Terminal Application (SPTA), structured around three decoupled layers:

STORE ARCHITECTURE & STATE FLOW

TUI state is managed using three specialized Zustand stores combined with Immer middleware to allow safe, mutating updates to deeply nested state trees:

. UI STORE (`SRC/STORES/UI-STORE.TS`)

- State Managed: Active view index, sidebar visibility toggles, active modal overlays (e.g. confirmation dialogs), and the Command Palette fuzzy-search string.
- Key Actions: `setView(index)`, `toggleSidebar()`, `pushModal(modal)`, `popModal()`.

. SERVICE STORE (`SRC/STORES/SERVICE-STORE.TS`)

- State Managed: Real-time worker health matrices, trade logs history, analytics telemetry, log buffers, and the connection status.
- Key Actions: `updateWorkerStatus(name, status)`, `addLogEntry(worker, log)`, `addTradeFill(trade)`, `setConnectionState(state)`.

. CONFIG STORE (`SRC/STORES/CONFIG-STORE.TS`)

- State Managed: UI themes (dark/light), data refresh intervals, Telegram alert routes, and custom keyboard binds.
- Persistence: Automatically serialized and saved to disk at `~/.hoox/config.json`.

CONNECTION RESILIENCE STATE MACHINE

The TUI maintains a strict connection lifecycle to track connectivity to the local API server or remote worker tunnels:

EXPONENTIAL BACKOFF INTERVALS

If the API connection drops out:

. The status bar immediately transitions to a yellow/orange pulsing RECONNECTING dot.
. The polling engine schedules reconnect attempts using exponential backoff:
  $$\text{Backoff Interval} = \min(^{\text{retry\_count}} \times \text{ms}, \text{ms})$$
. Active views display a subtle `"Stale Data: Last updated Xm ago"` warning badge, allowing the operator to read static stats without freezing the screen layout.
. After failed attempts, the state transitions to `OFFLINE` (red dot). The engine continues running minimal background checks and restores live statistics automatically as soon as the API recovers.

DUAL-CHANNEL DATA COMMUNICATION

To maintain high performance and low thread overhead:

. REST API (POLLING CHANNEL)

- Frequency: Configurable in `CONFIG_KV` (default: every seconds).
- Data Transferred: Core worker health status checks, database statistics, and configuration files.

. SERVER-SENT EVENTS (REAL-TIME INGESTION CHANNEL)

- Protocol: HTTP SSE streaming.
- Data Transferred: High-frequency real-time logs and exchange trade fills.
- Memory Buffer Constraints: To prevent memory leaks during extended sessions in the terminal, the TUI implements strict ring-buffer limits:
  - Trade Feed: entries max.
  - Log Stream: , lines max.
  - Alert Console: lines max.
  - _When a buffer limit is hit, the oldest entry is automatically dropped._

DOUBLE-LAYER CRASH PROTECTION

Since the TUI is designed to be an enterprise-grade command center, a rendering bug or syntax error in one pane must never crash the entire application.

LAYER : PER-VIEW ERROR BOUNDARIES

Every view (e.g. `WorkerDetail`, `ConfigEditor`) is wrapped inside a custom React `ErrorBoundary` component:

- If a rendering exception occurs within the `ConfigEditor` (e.g. due to a Tree-sitter WASM parsing exception), the error is caught, a clean error panel is displayed inside that specific tab, and a `[Retry]` button is mounted.
- No Side Effects: The rest of the terminal, sidebar, and status bar continue running normally, ensuring uninterrupted monitoring.

LAYER : PROCESS-LEVEL TRAP (`CRASHRECOVERYAPP`)

If an uncaught exception or unhandled promise rejection occurs at the root level of the Node/Bun process:

. The process trap intercepts the signal and prevents a hard exit to a broken shell.
. Displays a styled Terminal Recovery Screen detailing the error stack.
. Offers three instant recovery options:
   - `[Restart]`: Cleans Zustand stores and performs a fresh reboot.
   - `[Safe Mode]`: Disables heavy analytics feeds and WASM syntax highlighting to restore operations in minimal capacity.
   - `[Report Bug]`: Automatically exports the error trace and logs to a file at `logs/crash-report.log`.

> Tip: Adding custom components or views? Ensure all JSX elements comply with OpenTUI's native terminal components (using only lowercase intrinsic tags like `<box>`, `<text>`, and `<list>`). Absolute layout dimensions must be integers representing terminal characters or `"%"` values.

NEXT STEPS

- [DevOps Setup & Operations Manual](setup_and_operations.md) - Complete runbook, variable matrices, and troubleshooting.
- [Architecture & Edge Topology](architecture/overview.md) - In-depth architectural outlines and Service Bindings routing maps.

[SECTION: SYSTEM TOPOLOGY & OVERVIEW]

SYSTEM TOPOLOGY & OVERVIEW

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

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

HIGH-LEVEL SYSTEM ARCHITECTURE

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

COMPREHENSIVE MICRO-WORKER CATALOG

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

THE -LAYER SECURITY ARCHITECTURE

Security is designed as concentric protective corridors:

[WAF: IP Range Allow-list] -> [Gateway: Webhook Passkey] -> [Isolation: Service Bindings] -> [Worker Auth: INTERNAL_KEY] -> [Mutex: Durable Objects]

LAYER : EDGE-LEVEL FIREWALL & WAF

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

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

LAYER : WEBHOOK PASSKEY AUTHENTICATION

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

LAYER : SERVICE BINDING ENCRYPTED ISOLATION

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

LAYER : STANDARDIZED INTERNAL AUTHORIZATION

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

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

LAYER : DURABLE OBJECT IDEMPOTENCY LOCKS

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

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

NEXT STEPS

- [Worker Communication Specifications](communication.md) - Deep dive into service bindings, zero-TCP routing, and V engines.
- [Data Flow Maps](data-flow.md) - Step-by-step sequence charts of trade executions and cron risk evaluations.

[SECTION: ISOLATE COMMUNICATION SPEC]

ISOLATE COMMUNICATION SPEC

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

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

. SERVICE BINDINGS: ZERO-OVERHEAD V ROUTING

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

THE UNDER-THE-HOOD V MECHANICS

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

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

. COMPLETE SERVICE INVOCATIONS IMPLEMENTATION

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

```
import { requireInternalAuth } from "@jango-blockchained/hoox-shared/middleware";

export interface Env {
  TRADE_SERVICE: Fetcher; // Service Binding Fetcher
  INTERNAL_KEY_BINDING: string; // Authorized Internal Key secret
}

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

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

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

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

```
      }
    },
  };
```

. INTERNAL AUTHORIZATION MIDDLEWARE STANDARD

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

```
  import { requireInternalAuth } from "@jango-blockchained/hoox-shared/middleware";

  // Inside target worker's router or fetch handler:
  export async function handleRequest(
    request: Request,
    env: Env
  ): Promise<Response> {
    // Returns  Unauthorized Response if the header is invalid or missing
    const authError = requireInternalAuth(request, env, "INTERNAL_KEY_BINDING");
    if (authError) return authError;

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

> Standardization Alert: Every single internal worker (`hoox`, `trade-worker`, `d-worker`, `agent-worker`, `telegram-worker`, `email-worker`) binds the exact same secret name: `INTERNAL_KEY_BINDING`. This eliminates variable footprint drift and simplifies secret deployments across your workspace.

. TESTING & MOCKING SERVICE BINDINGS

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

```
  import { expect, test, mock } from "bun:test";

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

```
            return new Response(JSON.stringify({ success: false }), {
              status: ,
            });
          }

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

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

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

NEXT STEPS

- [Data Flow Mapping](data-flow.md) - Step-by-step sequence charts of trade executions, backups, and metrics.
- [Bindings Catalog](bindings.md) - Review complete environment namespaces and bound objects.

[SECTION: SYSTEM DATA ROUTING SPEC]

SYSTEM DATA ROUTING SPEC

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

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

. WEBHOOK TO TRADE EXECUTION FLOW (HIGH-SPEED PATH)

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

. AUTONOMOUS AI RISK MONITORING FLOW (CRON CYCLE)

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

. PDF PORTFOLIO REPORT RENDERING FLOW

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

. OBSERVABILITY & TIME-SERIES ANALYTICS FLOW

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

. GLOBAL DATA PERSISTENCE MAPPING

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

NEXT STEPS

- [Bindings Catalog](bindings.md) - Check wrangler settings and resource declarations.
- [Storage Engineering Manual](storage.md) - Dive into Drizzle database schemas and SQLite properties.

[SECTION: INFRASTRUCTURE BINDINGS INDEX]

INFRASTRUCTURE BINDINGS INDEX

In Cloudflare's serverless architecture, bindings represent the declarative bridges linking your isolate compute logic to other internal microservices and storage platforms. This document serves as the absolute, production-grade reference registry for all resource bindings configured in the Hoox monorepo.

. SECRETS & ENVIRONMENT VARIABLES MATRIX

These parameters represent encrypted variables injected directly into V execution isolates at runtime.

Variable Name	Type	Bound Workers	Operational Impact
`INTERNAL_KEY_BINDING`	Secret	`hoox`, `trade-worker`, `d-worker`, `telegram-worker`, `email-worker`	Cryptographic auth key used by the `requireInternalAuth` middleware to validate service-to-service calls.
`TG_BOT_TOKEN_BINDING`	Secret	`telegram-worker`	Secret bot token issued by `@BotFather` to authenticate Telegram API commands and alerts.
`TELEGRAM_SECRET_TOKEN`	Secret	`telegram-worker`	Webhook verification token to authorize Telegram push events.
`WEBHOOK_API_KEY`	Secret	`hoox`	General passkey validated during incoming webhook signal requests.
`BYBIT_API_KEY` / `_SECRET`	Secret	`trade-worker`	Credentials used to execute cryptographically signed order routes to Bybit's APIs.
`BINANCE_API_KEY` / `_SECRET`	Secret	`trade-worker`	Credentials used to execute trade routes to Binance's APIs.
`MEXC_API_KEY` / `_SECRET`	Secret	`trade-worker`	Credentials used to execute trade routes to MEXC's APIs.
`CF_API_TOKEN`	Secret	`report-worker`	Access token used to invoke the Cloudflare Puppeteer Browser Rendering APIs.

. SERVICE BINDINGS (COMPUTE CONNECTORS)

Service Bindings link workers' V runtimes together locally in memory, with microsecond latency and zero external internet hops.

Binding Name	Ingesting Worker	Target Service	Purpose
`TRADE_SERVICE`	`hoox`, `agent-worker`, `email-worker`	`trade-worker`	Handles leverage calculation, size scaling, and execution.
`TELEGRAM_SERVICE`	`hoox`, `trade-worker`, `agent-worker`, `report-worker`	`telegram-worker`	Dispatches real-time alerts and parses slash commands.
`D_SERVICE`	`trade-worker`, `agent-worker`, `report-worker`, `dashboard`	`d-worker`	Serves as the high-integrity proxy SQL data manager.
`AGENT_SERVICE`	`dashboard`	`agent-worker`	Triggers risk audits, chat streams, and telemetry updates.

. KV NAMESPACE CACHES

Key-Value caches store parameters that require sub-millisecond read access.

Binding Name	Bound Workers	Purpose
`CONFIG_KV`	All Workers + Dashboard	Primary cache for the -key runtime manifest, rate-limiter, and Kill Switch.
`SESSIONS_KV`	`hoox` Gateway	Stores active webhook session credentials and token cookie states.

. ASYNCHRONOUS QUEUES

Queues guarantee message delivery during times of heavy exchange network congestion or API rate limits.

Binding Name	Ingesting Worker	Queue Name	Type	Operational Action

:-- |
| `TRADE_QUEUE` | `hoox` Gateway | `trade-execution` | Producer | Serializes and enqueues signal payloads during failover. |
| `TRADE_QUEUE` | `trade-worker` | `trade-execution` | Consumer | Pulls enqueued signals and retries executions with backoff. |

. SQLITE-BACKED DURABLE OBJECTS

Durable Objects enforce exactly-once execution, preventing catastrophic double-trading.

Binding Name	Bound Worker	Target Class Name	Purpose
`IDEMPOTENCY_STORE`	`hoox` Gateway	`IdempotencyStore`	Mutex locking engine with local SQLite and auto-alarm garbage collection.

. R OBJECT STORAGE BUCKETS

R Buckets store heavy files with zero bandwidth egress retrieval fees.

Binding Name	Bound Workers	Bucket Name	Target Asset Payload
`REPORTS_BUCKET`	`trade-worker`, `report-worker`	`trade-reports`	Compiled PDF daily/weekly portfolio reports.
`SYSTEM_LOGS_BUCKET`	`trade-worker`	`hoox-system-logs`	Verbose exchange API request-response logs.
`UPLOADS_BUCKET`	`telegram-worker`	`user-uploads`	User chart screenshots and conversation images.

. D SQLITE DATABASES

Binding Name	Bound Workers	Database Instance Name	Purpose
`DB`	`d-worker`	`trade-data-db`	Primary transactional trade database storage.

. WORKERS AI & VECTORIZE INDEXES

Binding Name	Bound Workers	Target Asset Name	Operational Purpose
`AI`	`hoox`, `trade-worker`, `agent-worker`, `telegram-worker`	Edge LLM Models	Runs LLaMA- inference, risk analysis, and chat.
`VECTORIZE_INDEX`	`telegram-worker`	`rag-index`	Custom semantic search vector DB for RAG queries.

. PUPPETEER BROWSER RENDERING

The `report-worker` invokes Cloudflare's Browser Rendering Chrome isolates using a secure REST API (no binding required):

- Route: `POST https://api.cloudflare.com/client/v/accounts/{account_id}/browser-rendering/pdf`
- Headers:
  ```http
  Authorization: Bearer <CF_API_TOKEN_BINDING>
  Content-Type: application/json
  ```
- JSON Payload:
  ```json
  {
    "html": "<html>...</html>",
    "options": {
      "format": "A",
      "printBackground": true
    }
  }
  ```

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

NEXT STEPS

- [Storage & SQLite DDL](storage.md) - Dive into Drizzle schemas, R bucket configurations, and database rules.
- [Production Deployments](../deployment/production.md) - Learn how Wrangler compiles and maps these bindings to the live edge.

[SECTION: STORAGE & DATA ENGINEERING SPEC]

STORAGE & DATA ENGINEERING SPEC

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

. THE MULTI-TIER STORAGE ARCHITECTURE

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

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

. D RELATIONAL ENGINE & DRIZZLE SCHEMA SPECS

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

DRIZZLE ORM SCHEMA STANDARDS

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

```
import { sqliteTable, text, real, integer } from "drizzle-orm/sqlite-core";

// . Transactional Trades Ledger Schema
```

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

  // . Real-Time Open Position Matrix Schema
  export const positions = sqliteTable("positions", {
    symbol: text("symbol").primaryKey(),
    exchange: text("exchange").notNull(),
    side: text("side").notNull(), // "LONG" | "SHORT"
    size: real("size").notNull(), // Current contract size
    entryPrice: real("entry_price").notNull(), // Volume-weighted average entry price
    leverage: integer("leverage").default(),
    updatedAt: integer("updated_at", { mode: "timestamp" }).default(new Date()),
  });
```

. R LOG OFFLOADING: SQLITE CONSERVATION STRATEGY

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

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

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

. KV EVENTUAL CONSISTENCY & CACHE PERFORMANCE

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

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

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

NEXT STEPS

- [Internal Endpoints Mapping](endpoints.md) - Map out exposed ports, URLs, and service bindings.
- [Wrangler Setup & Tooling](../development/local-dev.md) - Configure Wrangler to bind local D and KV instances for dev testing.

[SECTION: INTERNAL ENDPOINTS MAP]

INTERNAL ENDPOINTS MAP

This document catalogs all internal REST, service-to-service, and queue endpoints exposed across the Hoox microservice monorepo. Because internal workers have zero public IP footprints and are completely isolated by Cloudflare's Zero Trust service bindings, this map serves as the primary integration blueprint for routing, debugging, and dashboard interactions.

INTERACTIVE COMPUTE & ROUTING FLOW

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

ENDPOINTS DIRECTORY BY WORKER

Every internal HTTP request between V isolates must transmit the standard bearer header: `X-Internal-Auth-Key: <INTERNAL_KEY_BINDING>`

. `HOOX` (GATEWAY ROUTER)

- Status: Public Ingress Node
- Bindings Mounts: `TRADE_SERVICE`, `TELEGRAM_SERVICE`, `ANALYTICS_SERVICE`, `CONFIG_KV`, `TRADE_QUEUE`

Route	Method	Description	Request Shape	Success Response
`/webhook`	`POST`	Primary webhook receiver for TradingView alerts.	`WebhookSignal` JSON	` OK` (Orderfilled metadata)
`/health`	`GET`	Probes gateway, D connectivity, and DO status.	N/A	`{"status": "ok"}`
`/telegram-webhook`	`POST`	Processes chat commands pushed from Telegram.	Telegram Updates	` OK`

. `TRADE-WORKER` (EXECUTION ENGINE)

- Status: Private Compute Node (No Public URL)
- Bindings Mounts: `D_SERVICE`, `TELEGRAM_SERVICE`, `ANALYTICS_SERVICE`, `CONFIG_KV`

Route	Method	Description	Request Shape	Success Response
`/webhook`	`POST`	Direct fast-path execution trigger.	`WebhookSignal` JSON	` OK` (Fill detail JSON)
`/dex`	`POST`	Dispatches EVM orders on-chain via web wallet.	DexTrade JSON	` OK` (Tx Hash metadata)
`/api/signals`	`GET`	Retrieves recent signal logs from D.	Query params filters	` OK` (Array of signals)
`/health`	`GET`	Probes CPU thread state and exchange connections.	N/A	`{"status": "ok"}`

. `D-WORKER` (SQLITE HUB)

- Status: Private Data Proxy (No Public URL)
- Bindings Mounts: `DB` (D SQLite database binding)

Route	Method	Description	Request Shape	Success Response
`/query`	`POST`	Executes a single SQL query against the SQLite database.	`{"query": "SELECT  FROM trades", "params": []}`	`{"success": true, "results": [...]}`
`/batch`	`POST`	Executes multiple transactional SQL operations.	`{"queries": [{"query": "...", "params": []}]}`	`{"success": true, "results": [...]}`
`/api/dashboard/stats`	`GET`	Computes aggregated Win Rate, drawdown, and daily totals.	N/A	`{"success": true, "stats": {...}}`
`/{tableName}`	`GET`	Lists rows inside a specific SQLite table (with filters).	Query params	`{"success": true, "rows": [...]}`

. `AGENT-WORKER` (AI RISK MANAGER)

- Status: Private Compute Node (Runs primarily on Cron schedule `/ `)
- Bindings Mounts: `TRADE_SERVICE`, `D_SERVICE`, `TELEGRAM_SERVICE`, `AI`

Route	Method	Description	Request Shape	Success Response
`/agent/chat`	`POST`	Starts a conversational market/risk query (SSE supported).	`{"prompt": "...", "stream": true}`	`text/event-stream` stream
`/agent/vision`	`POST`	Analyzes image bytes using multimodal AI models.	`{"image": "base...", "prompt": "..."}`	`{"analysis": "..."}`
`/agent/status`	`GET`	Returns active trailing stops and current drawdowns.	N/A	`{"status": "active", "stops": []}`
`/health`	`GET`	Probes AI model availability and Cron loop timers.	N/A	`{"status": "ok"}`

. `TELEGRAM-WORKER` (PUSH ALERTS)

- Status: Private Compute Node
- Bindings Mounts: `ANALYTICS_SERVICE`, `AI`, `VECTORIZE_INDEX`

Route	Method	Description	Request Shape	Success Response
`/alert`	`POST`	Sends a push trade fill notification or daily digest.	`{"chatId": "...", "message": "..."}`	`{"success": true}`
`/health`	`GET`	Probes Telegram API connection.	N/A	`{"status": "ok"}`

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

NEXT STEPS

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

- [System Storage Architecture](storage.md) - Deep dive into R logs offloading and Drizzle ORM schemas.

[SECTION: VISUAL TOKENS & DESIGN SYSTEM]

VISUAL TOKENS & DESIGN SYSTEM

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

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

. OKLCH COLOR PALETTE

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

DARK MODE (PRIMARY VISUAL STANDARD)

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

. LAYOUT, BORDERS & DENSITY

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

. TYPOGRAPHY & RUNTIMES

| Aesthetic Layer | Font Family | CSS Utility | Ideal

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

. SVG ICON MAPPING CATALOG

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

A. ROCKET ICON (`GETTING-STARTED`)

Used to represent system setup and installation paths.

```
<svg
  class="size-."
  viewBox="   "
  fill="none"
  stroke="currentColor"
  stroke-width=""
>
  <path
    d="M. .c-. .- - s.-. -c.-..-.-.-.a. .   -.-.z"
  />
  <path
    d="M l--a     -.A. .     c .-. .- a. .   - z"
  />
  <path d="M Hs.-. -c.-.    " />
  <path d="M vs.-. -c.-. - -" />
</svg>
```

B. BOOK ICON (`GUIDES`)

Used to represent operational manuals and setup instructions.

```
<svg
  class="size-."
  viewBox="   "
```

```
    fill="none"
    stroke="currentColor"
    stroke-width=""
  >
    <path d="M .A. .    . H" />
    <path d="M. HvH.A. .     .v-A. .    . z" />
  </svg>
```

C. LIGHTBULB ICON (`CONCEPTS`)

Used to represent structural theories and edge architectures.

```
  <svg
    class="size-."
    viewBox="   "
    fill="none"
    stroke="currentColor"
    stroke-width=""
  >
    <path d="M h" />
    <path d="M h" />
    <path
      d="M. c.-..-. .-.A. .            c  . . . .A. .    . "
    />
  </svg>
```

D. BOOK-OPEN ICON (`REFERENCE`)

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

```
  <svg
    class="size-."
    viewBox="   "
    fill="none"
    stroke="currentColor"
    stroke-width=""
  >
    <path d="M ha      va    --Hz" />
    <path d="M h-a    - va     -hz" />
  </svg>
```

E. TARGET ICON (`TUTORIALS`)

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

```
  <svg
    class="size-."
    viewBox="   "
    fill="none"
    stroke="currentColor"
    stroke-width=""
  >
    <circle cx="" cy="" r="" />
    <circle cx="" cy="" r="" />
    <circle cx="" cy="" r="" />
  </svg>
```

F. GEAR ICON (`DEVOPS` & `WORKERS`)

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

```
  <svg
    class="size-."
    viewBox="   "
    fill="none"
    stroke="currentColor"
    stroke-width=""
  >
    <circle cx="" cy="" r="" />
    <path
      d="M vM vM. .l. .M. .l. .M hM hM. .l.-.M. .l.-."
    />
  </svg>
```

G. HOME HOUSE ICON (`HOME`)

Used to return to the parent portal.

```
  <svg
    class="size-"
    viewBox="   "
    fill="none"
    stroke="currentColor"
    stroke-width=""
  >
    <path d="M l-  va    - Ha    --z" />
    <polyline points="       " />
  </svg>
```

NEXT STEPS

- [System Topology Overview](overview.md) - Analyze edge isolates and data layers integrations.
- [Isolate Communication Spec](communication.md) - Check service bindings and standard Bearer Auth middleware.

[SECTION: HOOX GATEWAY ISOLATE PROFILE]

HOOX GATEWAY ISOLATE PROFILE

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

ARCHITECTURAL TOPOLOGY

. DECLARED WRANGLER CONFIGURATIONS & BINDINGS

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


```
{
  "name": "hoox",
  "main": "src/index.ts",
  "compatibility_date": "--",
  "compatibility_flags": ["nodejs_compat"],
  "account_id": "debcebeabecbcdec",
  "placement": {
    "mode": "smart",
  },
  "vars": {
    "ENVIRONMENT": "production",
  },
  "kv_namespaces": [
    {
      "binding": "CONFIG_KV",
      "id": "caefffbd",
    },
    {
      "binding": "SESSIONS_KV",
      "id": "ffabedaacc",
    },
  ],
  "services": [
    { "binding": "TRADE_SERVICE", "service": "trade-worker" },
    { "binding": "TELEGRAM_SERVICE", "service": "telegram-worker" },
    { "binding": "ANALYTICS_SERVICE", "service": "analytics-worker" },
  ],
```

```
    "queues": {
      "producers": [{ "queue": "trade-execution", "binding": "TRADE_QUEUE" }],
    },
    "durable_objects": {
      "bindings": [
        { "name": "IDEMPOTENCY_STORE", "class_name": "IdempotencyStore" },
      ],
    },
    "migrations": [
      {
        "tag": "v",
        "new_sqlite_classes": ["IdempotencyStore"],
      },
    ],
  }
```

. ENVIRONMENTAL VARIABLES & ENCRYPTED SECRETS

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

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

LOCAL DEVELOPMENT MOCKING (`.DEV.VARS`)

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

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

. API ROUTE SPECIFICATIONS

The gateway exposes three public entryways:

A. INGEST SIGNAL WEBHOOK

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

  ```json
  {
    "apiKey": "dev_webhook_auth_passkey",
    "exchange": "bybit",
    "action": "LONG",
    "symbol": "BTCUSDT",
    "quantity": .,
    "leverage": ,
    "idempotencyKey": "uuid-bdebd-bd"
  }
  ```
- Success Response (OK):
  ```json
  {
    "success": true,
    "requestId": "bdebd-bd-bad-bdd-bdbdcbd",
    "exchange": "bybit",
    "symbol": "BTCUSDT",
    "action": "LONG",
    "result": { "orderId": "", "status": "Filled" }
  }
  ```

B. TELEGRAM BOT UPDATE WEBHOOK

- Endpoint: `/telegram-webhook`
- Method: `POST`
- JSON Payload: Standard encrypted update structure forwarded by Telegram's webhook servers.

C. GATEWAY HEALTH DIAGNOSTICS

- Endpoint: `/health`
- Method: `GET`
- Success Response (OK):
  ```json
  {
    "status": "ok",
    "timestamp": ,
    "bindings": {
      "d": "connected",
      "kv": "connected",
  ```

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

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

NEXT STEPS

- [trade-worker Spec](trade-worker.md) - Review how trade executions, order math, and margin settings compile on the edge.
- [D Database Operations](../guides/database-ops.md) - Manage schema migrations, query ledgers, and execute SQL scripts.

[SECTION: TRADE-WORKER ISOLATE PROFILE]

TRADE-WORKER ISOLATE PROFILE

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

. DECLARED WRANGLER CONFIGURATIONS & BINDINGS

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


```
{
  "name": "trade-worker",
  "main": "src/index.ts",
  "compatibility_date": "--",
  "compatibility_flags": ["nodejs_compat"],
  "account_id": "debcebeabecbcdec",
  "placement": {
    "mode": "smart",
  },
  "d_databases": [
    {
      "binding": "DB",
      "database_name": "trade-data-db",
      "database_id": "caefffba",
    },
  ],
  "r_buckets": [
    { "binding": "REPORTS_BUCKET", "bucket_name": "trade-reports" },
    { "binding": "SYSTEM_LOGS_BUCKET", "bucket_name": "hoox-system-logs" },
  ],
  "kv_namespaces": [
    {
      "binding": "CONFIG_KV",
      "id": "caefffbd",
    },
  ],
  "queues": {
    "consumers": [
      {
        "queue": "trade-execution",
        "max_batch_size": ,
        "max_batch_timeout": ,
```

```
        },
      ],
    },
    "secrets": [
      "INTERNAL_KEY_BINDING",
      "BYBIT_API_KEY",
      "BYBIT_API_SECRET",
      "BINANCE_API_KEY",
      "BINANCE_API_SECRET",
      "MEXC_API_KEY",
      "MEXC_SECRET_BINDING",
    ],
  }
```

. THE PROVIDER-BASED `EXCHANGEROUTER` PATTERN

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

A. GENERIC EXCHANGE PROVIDER INTERFACE

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

```
export interface IExchangeProvider<TClient, TEnv> {
  readonly name: string;
  createClient(env: TEnv): TClient;
  hasCredentials(env: TEnv): boolean;
}
```

B. DYNAMIC RUNTIME ROUTING

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

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

. INTERNAL REST API SPECIFICATION

A. PROCESS ORDER PIPELINE

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

- Endpoint: `/process`
- Method: `POST`
- Headers: `X-Internal-Auth-Key: <INTERNAL_KEY_BINDING>`
- JSON Payload:
  ```json
  {
    "requestId": "bdebd-bd-bad-bdd-bdbdcbd",
    "payload": {
      "exchange": "bybit",
      "action": "LONG",
      "symbol": "BTCUSDT",
      "quantity": .,
      "leverage":
    }
  }
  ```
- Success Response (OK):
  ```json
  {
    "success": true,
    "result": {
      "orderId": "",
      "status": "Filled",
      "price": .
    },
    "error": null
  }
  ```

. STANDARDIZED EXCEPTION HANDLING

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

```
import { Errors } from "@jango-blockchained/hoox-shared/errors";

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

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

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

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

NEXT STEPS

- [hoox Gateway Profile](hoox.md) - Review WAF rules and Durable Object idempotency locks.
- [D Database Operations](../guides/database-ops.md) - Manage Drizzle schemas, migrations, and query operations.

[SECTION: AGENT-WORKER ISOLATE PROFILE]

AGENT WORKER - HOOX AUTONOMOUS AI & RISK MANAGER

Last Updated: May (Post-Enhancement)

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

CORE CAPABILITIES

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

ARCHITECTURE & FLOW

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

ENDPOINTS & INTERACTIONS

MANAGEMENT ENDPOINTS

`GET /AGENT/CONFIG`

Returns current agent configuration including provider settings.


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


`POST /AGENT/CONFIG`

Update agent configuration at runtime.


```
{
  "defaultProvider": "openai",
  "fallbackChain": ["openai", "workers-ai", "anthropic", "google", "azure"],
  "modelMap": {
    "workers-ai": "@cf/meta/llama-.-b-instruct",
    "openai": "gpt-o-mini---",
    "anthropic": "claude--haiku-",
    "google": "gemini-.-flash-",
    "azure": "gpt-o-mini"
  },
  "timeoutMs": ,
  "retryCount":
}
```


`GET /AGENT/MODELS`

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

`POST /AGENT/TEST-MODEL`

Test a specific AI model.


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


`GET /AGENT/HEALTH`

Returns health status of all configured AI providers.

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

AI INTERACTION ENDPOINTS

`POST /AGENT/CHAT`

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

Request:

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

Streaming Response (SSE):

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

`POST /AGENT/VISION` (NEW!)

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

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

Or with base:

```
{
  "imageBase": "iVBORwKGgoAAAANSUhEUgAA...",
  "prompt": "What pattern do you see in this chart?"
```

```
  }
```

`POST /AGENT/REASONING` (NEW!)

Extended thinking queries with reasoning models like OpenAI o.

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

Response:

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

`GET /AGENT/USAGE` (NEW!)

Get AI API usage statistics across all providers.

```
  {
    "success": true,
    "usage": {
      "workers-ai": { "requests": , "tokens":  },
      "openai": { "requests": , "tokens":  }
    }
  }
```

`GET /AGENT/PROMPTS` (NEW!)

List available prompt templates.

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

`POST /AGENT/EMBEDDING`

Generate text embeddings using Workers AI embedding models.

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

```
  }
```

LEGACY ENDPOINTS

`POST /AGENT/RISK-OVERRIDE`

Manually enforce or release risk locks.

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

`GET /AGENT/STATUS`

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

CONFIGURATION

KV KEYS

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

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

DEFAULT AGENT CONFIG


```
{
  "defaultProvider": "workers-ai",
  "fallbackChain": ["workers-ai", "openai", "anthropic", "google", "azure"],
  "modelMap": {
    "workers-ai": "@cf/meta/llama-.-b-instruct",
    "openai": "gpt-o-mini---",
    "anthropic": "claude--haiku-",
    "google": "gemini-.-flash-",
    "azure": "gpt-o-mini"
  },
  "timeoutMs": ,
  "retryCount": ,
  "maxDailyDrawdownPercent": -,
  "trailingStopPercent": .,
  "takeProfitPercent": .
}
```


SUPPORTED MODELS

WORKERS AI MODELS

Task	Workers AI Model
Chat	`@cf/meta/llama-.-b-instruct`
Vision	`@cf/meta/llama-.-b-vision-instruct`
Reasoning	`@cf/deepseek-ai/deepseek-r-distill-qwen-b`
Code	`@cf/qwen/qwen.-coder-b-instruct`
Embeddings	`@cf/baai/bge-base-en-v.`
Summarization	`@cf/facebook/bart-large-cnn`

EXTERNAL PROVIDERS

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

AI GATEWAY FEATURES

The AI Gateway provides:

- Fallback Chain: Automatically tries providers in order on failure
- Health Checks: Providers self-report health status every minutes

- Retry Logic: Exponential backoff with configurable max retries
- Timeout Protection: Configurable per-request timeout (default s)
- Usage Tracking: Automatic token and request counting per provider

INTERNAL SERVICE BINDINGS

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

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

TESTING

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

```
 Run all tests
bun test

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

 Run with coverage
bun test --coverage
```

Test Coverage: >% across all modules

TEST STRUCTURE

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

[SECTION: TELEGRAM-WORKER ISOLATE PROFILE]

TELEGRAM-WORKER ISOLATE PROFILE

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

. DECLARED WRANGLER CONFIGURATIONS & BINDINGS

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


```
{
  "name": "telegram-worker",
  "main": "src/index.ts",
  "compatibility_date": "--",
  "compatibility_flags": ["nodejs_compat"],
  "account_id": "debcebeabecbcdec",
  "placement": {
    "mode": "smart",
  },
  "kv_namespaces": [
    {
      "binding": "CONFIG_KV",
      "id": "caefffbd",
    },
  ],
  "r_buckets": [
    {
      "binding": "UPLOADS_BUCKET",
      "bucket_name": "user-uploads",
    },
  ],
  "vectorize": [
    {
      "binding": "VECTORIZE_INDEX",
      "index_name": "rag-index",
    },
  ],
  "ai": {
    "binding": "AI",
  },
  "secrets": [
    "INTERNAL_KEY_BINDING",
```

```
      "TG_BOT_TOKEN_BINDING",
      "TELEGRAM_CHAT_ID_DEFAULT",
      "TELEGRAM_WEBHOOK_SECRET",
    ],
  }
```

. ENVIRONMENTAL VARIABLES & ENCRYPTED SECRETS

- `TG_BOT_TOKEN_BINDING`: Private HTTP bot token generated by `@BotFather`.
- `TELEGRAM_CHAT_ID_DEFAULT`: Your authorized Chat ID acting as a fallback receiver and admin lock.
- `TELEGRAM_WEBHOOK_SECRET`: A secure, random token appended to the webhook URL path to validate that the request originated from Telegram's servers.
- `INTERNAL_KEY_BINDING`: Shared key used to validate calls from `hoox` or `trade-worker`.

LOCAL DEVELOPMENT MOCKING (`.DEV.VARS`)

```
TG_BOT_TOKEN_BINDING=mock_telegram_bot_token
TELEGRAM_CHAT_ID_DEFAULT=
TELEGRAM_WEBHOOK_SECRET=local_secure_route_token
INTERNAL_KEY_BINDING=dev_shared_internal_security_key
```

. INTERNAL REST API SPECIFICATION

A. DISPATCH NOTIFICATION ENDPOINT

- Endpoint: `/alert`
- Method: `POST`
- Headers: `X-Internal-Auth-Key: <INTERNAL_KEY_BINDING>`
- JSON Payload:
  ```json
  {
    "requestId": "bdebd-bd-bad-bdd-bdbdcbd",
    "payload": {
      "chatId": "",
      "message": "<b> Bybit LONG Filled!</b>\nSymbol: <code>BTCUSDT</code>\nPrice: <code>$,.</code>\nQuantity: <code>.</code>",
      "parseMode": "HTML"
    }
  }
  ```
- Success Response (OK):

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

B. TELEGRAM INGRESS WEBHOOK ROUTE

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

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

. AI-POWERED CHATBOT & RAG MECHANICS

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

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

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

NEXT STEPS

- [hoox Gateway Profile](hoox.md) - Review how incoming commands route to the gateway.
- [Setting Up Telegram Bot Alerts](../../tutorials/telegram-bot.md) - Step-by-step

tutorial for BotFather configuration.

[SECTION: D-WORKER ISOLATE PROFILE]

D-WORKER ISOLATE PROFILE

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

. DECLARED WRANGLER CONFIGURATIONS & BINDINGS

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

```
{
  "name": "d-worker",
  "main": "src/index.ts",
  "compatibility_date": "--",
  "compatibility_flags": ["nodejs_compat"],
  "account_id": "debcebeabecbcdec",
  "placement": {
    "mode": "smart",
  },
  "d_databases": [
    {
      "binding": "DB",
      "database_name": "trade-data-db",
      "database_id": "caefffba",
    },
  ],
  "kv_namespaces": [
    {
      "binding": "CONFIG_KV",
      "id": "caefffbd",
    },
  ],
  "secrets": ["INTERNAL_KEY_BINDING"],
}
```

. INTERNAL REST API SPECIFICATION

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

header.

A. EXECUTE SINGLE SQL QUERY

- Endpoint: `/query`
- Method: `POST`
- JSON Payload:
  ```json
  {
    "requestId": "bdebd-bd-bad-bdd-bdbdcbd",
    "query": "SELECT created_at, symbol, action, price FROM trades WHERE symbol = ? ORDER BY created_at DESC LIMIT ?",
    "params": ["BTCUSDT", ]
  }
  ```
- Success Response (OK):
  ```json
  {
    "success": true,
    "results": [
      {
        "created_at": ,
        "symbol": "BTCUSDT",
        "action": "LONG",
        "price": .
      }
    ],
    "meta": { "rows_read": , "rows_written": , "duration": . }
  }
  ```

B. EXECUTE TRANSACTIONAL BATCH OPERATIONS

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

- Endpoint: `/batch`
- Method: `POST`
- JSON Payload:
  ```json
  {
    "requestId": "bdebd-bd-bad-bdd-bdbdcbd",
    "queries": [
      {
        "query": "INSERT INTO trades (id, symbol, price) VALUES (?, ?, ?)",
  ```

```
        "params": ["trade-", "BTCUSDT", ]
      },
      {
        "query": "UPDATE positions SET size = size + ? WHERE symbol = ?",
        "params": [., "BTCUSDT"]
      }
    ]
  }
  ```
- Success Response (OK):
  ```json
  {
    "success": true,
    "results": [{ "meta": { "changes":  } }, { "meta": { "changes":  } }]
  }
  ```

C. DASHBOARD TELEMETRY STATISTICS

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

- Endpoint: `/api/dashboard/stats`
- Method: `GET`
- Success Response (OK):
  ```json
  {
    "success": true,
    "stats": {
      "totalTrades": ,
      "winRate": .,
      "totalPnlUSDT": .,
      "activePositionsCount": ,
      "dailyTradesCount":
    }
  }
  ```

. SECURITY & SQL INJECTION PROTECTION

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

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

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

NEXT STEPS

- [System Storage Architecture](../architecture/storage.md) - Review SQLite properties and R bucketing pipelines.
- [Database Operations Manual](../guides/database-ops.md) - Learn commands to run query ledgers and restore backups.

[SECTION: WEB-WALLET-WORKER ISOLATE PROFILE]

WEB-WALLET-WORKER ISOLATE PROFILE

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

. DECLARED WRANGLER CONFIGURATIONS & BINDINGS

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

```
{
  "name": "web-wallet-worker",
  "main": "src/index.ts",
  "compatibility_date": "--",
  "compatibility_flags": ["nodejs_compat"],
  "account_id": "debcebeabecbcdec",
  "vars": {
    "DEFAULT_CHAIN": "ethereum",
  },
  "kv_namespaces": [
    {
      "binding": "CONFIG_KV",
      "id": "caefffbd",
    },
  ],
  "secrets": [
    "INTERNAL_KEY_BINDING",
    "WALLET_MNEMONIC_SECRET",
    "WALLET_PK_SECRET",
    "RPC_PROVIDER_URL",
  ],
}
```

. ENVIRONMENTAL VARIABLES & ENCRYPTED SECRETS

- `WALLET_PK_SECRET`: Encrypted private key used for single-account execution.
- `WALLET_MNEMONIC_SECRET`: Encrypted or -word HD wallet seed phrase used to derive multiple accounts.
- `RPC_PROVIDER_URL`: High-availability HTTP Ethereum / EVM RPC provider (e.g., Infura,

Alchemy, or QuickNode).
- `INTERNAL_KEY_BINDING`: Shared key used to validate calls from internal compute nodes.

LOCAL DEVELOPMENT MOCKING (`.DEV.VARS`)

```
WALLET_PK_SECRET=xabcdefabcdefabcdefabcdef
WALLET_MNEMONIC_SECRET="abandon abandon abandon abandon abandon abandon abandon abandon abandon abandon abandon about"
RPC_PROVIDER_URL=http://localhost:
INTERNAL_KEY_BINDING=dev_shared_internal_security_key
```

. INTERNAL REST API SPECIFICATION

A. EXECUTE ON-CHAIN TRANSACTION

- Endpoint: `/process`
- Method: `POST`
- Headers: `X-Internal-Auth-Key: <INTERNAL_KEY_BINDING>`
- JSON Payload:
  ```json
  {
    "requestId": "bdebd-bd-bad-bdd-bdbdcbd",
    "payload": {
      "action": "sendTransaction",
      "chain": "arbitrum",
      "to": "xbecdabeedeacdf",
      "value": ".",
      "data": "xacbb...",
      "gasLimit":
    }
  }
  ```
- Success Response (OK):
  ```json
  {
    "success": true,
    "result": {
      "txHash": "xaebfddacbcfddacbcfddacbcfdab",
      "nonce": ,
      "gasUsed": ,
      "effectiveGasPrice": ""
    },
    "error": null
  }
  ```

B. QUERY TOKEN BALANCE

- Endpoint: `/process`
- Method: `POST`
- JSON Payload:
  ```json
  {
    "requestId": "bdebd-bd-bad-bdd-bdbdcbd",
    "payload": {
      "action": "getBalance",
      "chain": "polygon",
      "address": "xbecdabeedeacdf",
      "tokenAddress": "xcddcaccaebbef"
    }
  }
  ```
- Success Response (OK):
  ```json
  {
    "success": true,
    "result": {
      "balance": ".",
      "symbol": "USDT",
      "decimals":
    },
    "error": null
  }
  ```

. ON-CHAIN SECURITY BEST PRACTICES

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

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

transaction payloads or try to brute-force auth codes.

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

NEXT STEPS

- [trade-worker Profile](trade-worker.md) - Review how execution orders route transactions to EVM wallet nodes.
- [D Database Operations](../guides/database-ops.md) - Manage your SQLite schemas and sync positions logs.

[SECTION: EMAIL-WORKER ISOLATE PROFILE]

EMAIL-WORKER ISOLATE PROFILE

The `email-worker` is an ancillary signal ingestion plugin. Deployed as a private compute isolate, this service is responsible for intercepting trading signals distributed via email lists, newsletters, or alert emails (e.g. from TradingView or custom notification pipelines). It validates domain security (SPF/DKIM), parses message subjects and bodies using dynamic Regular Expressions, and routes the extracted signals privately to `trade-worker` via V Service Bindings.

. DECLARED WRANGLER CONFIGURATIONS & BINDINGS

The `email-worker` does not expose any public endpoints, connecting internally to active executors and metrics collectors:


```
{
  "name": "email-worker",
  "main": "src/index.ts",
  "compatibility_date": "--",
  "compatibility_flags": ["nodejs_compat"],
  "account_id": "debcebeabecbcdec",
  "vars": {
    "USE_IMAP": "true",
  },
  "kv_namespaces": [
    {
      "binding": "CONFIG_KV",
      "id": "caefffbd",
    },
  ],
  "services": [
    { "binding": "TRADE_SERVICE", "service": "trade-worker" },
    { "binding": "ANALYTICS_SERVICE", "service": "analytics-worker" },
  ],
  "secrets": [
    "INTERNAL_KEY_BINDING",
    "EMAIL_HOST_BINDING",
    "EMAIL_USER_BINDING",
    "EMAIL_PASS_BINDING",
    "MAILGUN_API_KEY",
  ],
}
```

. ENVIRONMENTAL VARIABLES & ENCRYPTED SECRETS

- `EMAIL_HOST_BINDING`: POP/IMAP mailbox server address (e.g., `imap.gmail.com`).
- `EMAIL_USER_BINDING`: Dedicated signals email mailbox username.
- `EMAIL_PASS_BINDING`: Secure app-specific password.
- `MAILGUN_API_KEY`: Key used to validate incoming SMTP webhooks if using Mailgun routing.
- `INTERNAL_KEY_BINDING`: Shared key used to validate calls from internal compute nodes.

. REGEX PARSING MECHANICS IN V

When a new email is scanned or received via webhook, the worker evaluates the content against regular expression arrays stored in `CONFIG_KV`:

Configuration Key	Type	Default Pattern	Parser Purpose
`email:scan_subject`	`string`	`^TRADE SIGNAL:.`	Resolves whether the email should be parsed or ignored.
`email:coin_pattern`	`string`	`(BTC|ETH|SOL|LINK)`	Matches uppercase asset tokens in the email body.
`email:action_pattern`	`string`	`(BUY|SELL|LONG|SHORT)`	Resolves buy/sell execution parameters.
`email:quantity_multiplier`	`number`	`.`	Coefficient applied to parsed quantities to scale positions.

```
// Under the hood regex parsing logic
const subjectRegex = new RegExp(env.CONFIG_KV.get("email:scan_subject"));
const coinRegex = new RegExp(env.CONFIG_KV.get("email:coin_pattern"));
const actionRegex = new RegExp(env.CONFIG_KV.get("email:action_pattern"));

if (subjectRegex.test(email.subject)) {
  const asset = email.body.match(coinRegex)?.[]; // Extracts "BTC"
  const action = email.body.match(actionRegex)?.[]; // Extracts "LONG"
  // Routes to trade-worker...
}
```

. API INTERFACE SPECIFICATION

While the worker primarily runs background Cron scan loops, it exposes two ingestion endpoints for webhook integrations (e.g. Mailgun/SendGrid):

A. MAILGUN WEBHOOK ROUTER

- Endpoint: `/webhook/mailgun`
- Method: `POST`
- Headers: `Mailgun-Signature`, `Mailgun-Timestamp`, `Mailgun-Token`
- Security: The worker computes the HMAC-SHA signature of the timestamp and token using your `MAILGUN_API_KEY` secret. If the calculated signature doesn't match the header, the request is instantly rejected (Unauthorized).

B. DIRECT JSON PAYLOAD INGESTION

Used primarily by administrative automation tools or local testing scripts.

- Endpoint: `/process`
- Method: `POST`
- JSON Payload:
  ```json
  {
    "requestId": "bdebd-bd-bad-bdd-bdbdcbd",
    "payload": {
      "sender": "alerts@tradingview.com",
      "subject": "TRADE SIGNAL: Breakout SOL",
      "body": "Action: LONG, Asset: SOLUSDT, Size: ."
    }
  }
  ```
- Success Response (OK):
  ```json
  {
    "success": true,
    "result": { "symbol": "SOLUSDT", "action": "LONG", "quantity": . },
    "error": null
  }
  ```

. DOMAIN INGRESS PROTECTIONS: SPF & DKIM

To protect your wallet against email spoofing attacks:

. DKIM Check: Validates the cryptographic DKIM signature in the email header to prove the message body was not altered in transit.
. SPF Check: Matches the sending server's IP address against the DNS TXT SPF record of the authorized domain, dropping forged emails before parsing.

NEXT STEPS

- [trade-worker Profile](trade-worker.md) - Review how execution orders route transactions to EVM wallet nodes.
- [System Storage Architecture](../architecture/storage.md) - Review SQLite properties and R bucketing pipelines.

[SECTION: ANALYTICS-WORKER ISOLATE PROFILE]

ANALYTICS-WORKER ISOLATE PROFILE

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

. DECLARED WRANGLER CONFIGURATIONS & BINDINGS

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

```
{
  "name": "analytics-worker",
  "main": "src/index.ts",
  "compatibility_date": "--",
  "compatibility_flags": ["nodejs_compat"],
  "account_id": "debcebeabecbcdec",
  "analytics_engine_datasets": [
    {
      "binding": "ANALYTICS_ENGINE",
      "dataset": "hoox_telemetry",
    },
  ],
  "kv_namespaces": [
    {
      "binding": "CONFIG_KV",
      "id": "caefffbd",
    },
  ],
  "secrets": [
    "INTERNAL_KEY_BINDING",
    "CLOUDFLARE_API_TOKEN", // Required to run SQL queries against Analytics datasets
  ],
}
```

. ENVIRONMENTAL VARIABLES & ENCRYPTED SECRETS

- `CLOUDFLARE_API_TOKEN`: A secure token with `Account.Analytics` read permissions, allowing the worker to query stored datasets.
- `INTERNAL_KEY_BINDING`: Shared key used to validate calls from other V isolates.

. INTERNAL REST API SPECIFICATION

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

A. TRACK TELEMETRY EVENT

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

- Endpoint: `/track/api-call`
- Method: `POST`
- JSON Payload:
  ```json
  {
    "requestId": "bdebd-bd-bad-bdd-bdbdcbd",
    "payload": {
      "worker": "trade-worker",
      "endpoint": "/webhook",
      "latencyMs": .,
      "success": true
    }
  }
  ```

UNDER-THE-HOOD ANALYTICS ENGINE WRITE

When received, the worker formats and writes a time-series data point using standard Blobs (metadata strings) and Doubles (numeric values):

```
env.ANALYTICS_ENGINE.writeDataPoint({
  blobs: [
    payload.worker, // Blob : Worker Name
    payload.endpoint, // Blob : Endpoint Path
    payload.success ? "" : "", // Blob : Success state
  ],
  doubles: [
    payload.latencyMs, // Double : Latency in milliseconds
  ],
  indexes: [
    payload.requestId, // Custom index for distributed tracing
  ],
});
```

- Response (OK):
  ```json
  { "success": true }
  ```

B. QUERY METRICS (SQL INTERFACE)

Used primarily by the Next.js Dashboard to extract data points for chart rendering.

- Endpoint: `/query`
- Method: `POST`
- JSON Payload:
  ```json
  {
    "query": "SELECT sum(double) as total_latency, count() as total_calls FROM hoox_telemetry WHERE blob = ? AND timestamp >= now() - INTERVAL '' DAY",
    "params": ["trade-worker"]
  }
  ```
- Success Response (OK):
  ```json
  {
    "success": true,
    "results": [{ "total_latency": ., "total_calls":  }]
  }
  ```

. DASHBOARD VISUAL INTEGRATIONS

The metrics written to `hoox_telemetry` are queried by the dashboard to render:

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

> Tip: Testing analytics locally? Local Wrangler dev automatically intercepts `writeDataPoint` calls and logs the parsed Blobs and Doubles straight to your terminal

standard output, ensuring easy debugging!

NEXT STEPS

- [System Observability Guides](../deployment/monitoring.md) - Deepen your understanding of time-series logging and metrics.
- [trade-worker Profile](trade-worker.md) - Review how execution latency details are generated and offloaded.

[SECTION: REPORT-WORKER ISOLATE PROFILE]

REPORT-WORKER ISOLATE PROFILE

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

. DECLARED WRANGLER CONFIGURATIONS & BINDINGS

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

```
{
  "name": "report-worker",
  "main": "src/index.ts",
  "compatibility_date": "--",
  "compatibility_flags": ["nodejs_compat"],
  "account_id": "debcebeabecbcdec",
  "placement": {
    "mode": "smart",
  },
  "triggers": {
    "crons": ["    ", "    "], // Runs twice daily
  },
  "r_buckets": [
    {
      "binding": "REPORTS_BUCKET",
      "bucket_name": "trade-reports",
    },
  ],
  "services": [
    { "binding": "D_SERVICE", "service": "d-worker" },
    { "binding": "TELEGRAM_SERVICE", "service": "telegram-worker" },
  ],
  "secrets": [
    "INTERNAL_KEY_BINDING",
    "CF_API_TOKEN", // Required to call Browser Rendering REST API
  ],
}
```

. ENVIRONMENTAL VARIABLES & ENCRYPTED SECRETS

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

. BROWSER RENDERING & PDF PRINT PIPELINE

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

STEP : DATA AGGREGATION & HTML COMPILATION

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

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

STEP : CLOUDFLARE CHROME ISOLATE PRINT

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

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

STEP : R STORAGE & EXPIRATION

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

STEP : DISPATCH TELEGRAM ALERT

Calls `TELEGRAM_SERVICE` to send the link:

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

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

NEXT STEPS

- [telegram-worker Profile](telegram-worker.md) - Review push alert configurations and webhook tunnels.
- [D Database Operations](../guides/database-ops.md) - Manage Drizzle schemas and execute custom queries.

[SECTION: NEXT.JS DASHBOARD & OPENNEXT ISOLATE]

NEXT.JS DASHBOARD & OPENNEXT ISOLATE

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

NEXT.JS TO CLOUDFLARE EDGE ARCHITECTURE

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

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

. DECLARED WRANGLER CONFIGURATIONS & BINDINGS

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

```
{
  "name": "hoox-dashboard",
  "main": ".open-next/worker.js",
  "compatibility_date": "--",
  "compatibility_flags": ["nodejs_compat"],
  "account_id": "debcebeabecbcdec",
  "assets": {
    "directory": ".open-next/assets",
    "binding": "ASSETS",
  },
  "kv_namespaces": [
    {
      "binding": "CONFIG_KV",
      "id": "caefffbd",
    },
  ],
  "services": [
    { "binding": "D_SERVICE", "service": "d-worker" },
```

```
      { "binding": "AGENT_SERVICE", "service": "agent-worker" },
      { "binding": "TELEGRAM_SERVICE", "service": "telegram-worker" },
    ],
    "vars": {
      "D_SERVICE_URL": "https://d-worker.cryptolinx.workers.dev",
      "AGENT_SERVICE_URL": "https://agent-worker.cryptolinx.workers.dev",
      "TELEGRAM_SERVICE_URL": "https://telegram-worker.cryptolinx.workers.dev",
    },
    "secrets": [
      "DASHBOARD_USER",
      "DASHBOARD_PASS",
      "SESSION_SECRET",
      "INTERNAL_KEY_BINDING",
    ],
  }
```

. ENVIRONMENTAL VARIABLES & ENCRYPTED SECRETS

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

LOCAL DEVELOPMENT MOCKING (`.ENV.LOCAL`)

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

```
DASHBOARD_USER=admin
DASHBOARD_PASS=admin_dev_passkey_
SESSION_SECRET=abandon_abandon_abandon_abandon_
```

. SCHEMA-DRIVEN SETTINGS SYSTEM

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

```
{
  "sections": [
    {
      "id": "risk",
      "title": "Risk Management",
```

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

FORM SUBMISSION FLOW

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

. PRODUCTION BUILD & ROLLOUT RUNBOOK

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

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

   . Build the application using OpenNext
  bun run opennext:build

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

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

NEXT STEPS

- [agent-worker Profile](agent-worker.md) - Review AI chat streaming and risk evaluation structures.
- [d-worker Profile](d-worker.md) - Check REST schemas that feed database metrics to the dashboard.

[SECTION: PRODUCTION DEPLOYMENT RUNBOOK]

PRODUCTION DEPLOYMENT RUNBOOK

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

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

. PRODUCTION ROLLOUT PRE-FLIGHT CHECKLIST

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

- API Token Scoping: Confirm your active Cloudflare API Token inherits the strict minimal permission sets (`Account.D: Edit`, `Account.KV: Edit`, `Account.Queues: Edit`, `Account.Workers: Edit`, and `Zone.DNS: Edit` if mapping custom routing paths).
- Workspace Diagnostic Sweep: Run the automated pre-flight check to verify that all git submodules, local dependencies, and TypeScript variables compile without warnings:
  ```bash
  hoox check prerequisites
  hoox test
  ```
- Exchange Credentials Check: Ensure you have created dedicated exchange API keys with Withdrawal permissions turned OFF (disabled) to enforce least-privilege security.

. PRODUCTION ROLLOUT INVOCATIONS

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

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

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

DASHBOARD OPENNEXT COMPILATION

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

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

. CUSTOM DOMAIN & ROUTING MAPPING

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

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

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

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

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

. EDGE ISOLATE HARDENING STRATEGIES

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

A. ENABLE SMART PLACEMENT

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

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

```
}
```

B. RESTRICT CORS ORIGIN POLICIES

The public `/webhook` route inside the `hoox` gateway must only accept POST requests from authorized endpoints. Disable all CORS headers to prevent cross-origin script injections from browsers.

C. DEPLOY WAF RATE LIMIT TRAPS

Use the Hoox CLI to provisionZone-level rate limiters on Cloudflare's global edge network, dropping packet floods before they load the V isolates:

```
Provision a strict rate-limit rule (e.g., max  requests/minute to /webhook)
hoox waf configure --limit-requests
```

> Tip: Made an emergency change? You don't need a full rebuild. If the change was a configuration setting or a trade symbol routing change, simply update the KV store: `hoox config kv set <key> <value>`. The update will propagate globally in under seconds!

NEXT STEPS

- [CI/CD Pipelines Reference](cicd.md) - Automate edge deployments using GitHub Actions.
- [Observability & Time-Series Monitoring](monitoring.md) - Stream console logs and check analytics datasets.

[SECTION: CI/CD PIPELINE AUTOMATION]

CI/CD PIPELINE AUTOMATION

Automating your deployment pipeline ensures that every commit pushed to your repository is systematically vetted for code style, type safety, syntax regressions, and unit test coverage before being deployed to your live production trading nodes.

This guide provides the complete specification and production-grade YAML workflow for implementing GitHub Actions integration pipelines.

THE CONTINUOUS INTEGRATION & DEPLOYMENT PIPELINE FLOW

COMPLETE GITHUB ACTIONS WORKFLOW (`DEPLOY.YML`)

Create a YAML workflow file inside your repository at `.github/workflows/deploy.yml`:

```
name: "Hoox Production CI/CD Pipeline"

on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main

concurrency:
  group: ${{ github.workflow }}-${{ github.ref }}
  cancel-in-progress: true

jobs:
  verify-and-deploy:
    name: "Verify & Deploy Stack"
    runs-on: "ubuntu-latest"
    timeout-minutes:

    steps:
       . Checkout repository with all worker submodules recursively
      - name: "Checkout Codebase"
        uses: "actions/checkout@v"
        with:
          submodules: "recursive"
          fetch-depth:
```

```
         . Install Bun JavaScript runtime environment
        - name: "Setup Bun Runtime"
          uses: "oven-sh/setup-bun@v"
          with:
            bun-version: "latest"

         . Cache Bun dependency modules to optimize pipeline speeds
        - name: "Cache Workspace Dependencies"
          uses: "actions/cache@v"
          with:
            path: "~/.bun/install/cache"
            key: "${{ runner.os }}-bun-${{ hashFiles('/bun.lockb') }}"
            restore-keys: |
              ${{ runner.os }}-bun-

         . Install all monorepo dependencies
        - name: "Install Dependencies"
          run: "bun install"

         . Check formatting and ESLint rules
        - name: "Run Lint Checks"
          run: "bun run lint"

         . Verify TypeScript compile-time type-safety (tsc --noEmit)
        - name: "Run Type Checks"
          run: "bun run typecheck"

         . Execute all unit and integration test assertions (excluding live tests)
        - name: "Run Bun Test Suite"
          run: "bun test"

         . Deploy all database schemas and workers in correct sequence
        - name: "Deploy Entire Edge Stack"
          if: "github.ref == 'refs/heads/main' && github.event_name == 'push'"
          env:
            CLOUDFLARE_API_TOKEN: "${{ secrets.CLOUDFLARE_API_TOKEN }}"
            CLOUDFLARE_ACCOUNT_ID: "${{ secrets.CLOUDFLARE_ACCOUNT_ID }}"
            SUBDOMAIN_PREFIX: "${{ secrets.SUBDOMAIN_PREFIX }}"
          run: |
             Bootstrap local path binaries
            bun run build:cli

             Deploy D SQL schemas to production
            npx wrangler d execute trade-data-db --file=workers/trade-worker/schema.sql --remote
--yes

             Sequentially upload all enabled workers
            ./packages/cli/bin/hoox.js deploy all --auto --quiet

             Post-deployment: update service bindings URLs globally
            ./packages/cli/bin/hoox.js deploy update-internal-urls --quiet
```

```
            Post-deployment: apply KV manifest configurations
           ./packages/cli/bin/hoox.js deploy kv-config --quiet

            Post-deployment: update Telegram bot webhook routing path
           ./packages/cli/bin/hoox.js deploy telegram-webhook --quiet
```

MANAGING SECRETS & VARIABLE SCOPES

To authorize GitHub to interact with your Cloudflare account, navigate to your repository's Settings > Secrets and variables > Actions and register three Action Secrets:

. `CLOUDFLARE_API_TOKEN`: A secure, scoped token with Workers, D, KV, and DNS write permissions.
. `CLOUDFLARE_ACCOUNT_ID`: Your unique -character Cloudflare dashboard hash.
. `SUBDOMAIN_PREFIX`: The subdomain namespace chosen for your deployments.

> Note: You never need to store worker-specific secrets (like Bybit API keys or Telegram Bot tokens) in GitHub Secrets. These are stored directly in Cloudflare's secured key vaults. The CI pipeline only deploys the code logic; the running edge isolates pull their credentials locally from Cloudflare's hardware-level Secret Store at runtime.

PIPELINE CACHING & CONCURRENCY CONTROLS

- Concurrency Lock: The `concurrency` block configured in the YAML ensures that if you push a new commit while a previous deployment pipeline is running, GitHub Actions will automatically cancel the older, redundant run to avoid race conditions or double-deploying.
- Bun Cache: Caching dependency directories reduces build-time installation overhead from minutes to under seconds.

NEXT STEPS

- [Production Deployment Manual](production.md) - Audit DNS zones and custom domain routing options.
- [System Observability & Monitoring](monitoring.md) - Stream console logs and check analytics datasets.

[SECTION: OBSERVABILITY & TELEMETRY]

OBSERVABILITY & TELEMETRY

Operating a globally distributed algorithmic trading network demands extreme observability. A silent failure in an order loop or a transient API throttling event can lead to missed targets and unhedged exposures.

This guide outlines our telemetry architecture, covering Cloudflare Workers % request sampling, time-series SQL database queries, R logging, and automated alarm systems.

. % REQUEST HEAD SAMPLING

To monitor traffic at Cloudflare's edge compiler level, every worker in the Hoox monorepo enables native observability tracing inside its `wrangler.jsonc`:

```
{
  "observability": {
    "enabled": true,
    "head_sampling_rate": , // Captures and logs % of all incoming requests
  },
}
```

METRICS TRACKED AUTOMATICALLY

Once enabled, Cloudflare captures and graphs these metrics near-instantaneously:

- Requests velocity: Total hits and requests/second rates.
- CPU execution duration (ms): The exact CPU time consumed by the isolate thread.
- Uncaught Exceptions: Crashes or code errors that bypass middlewares.
- Subrequest Counts: Outbound HTTP calls made to exchange APIs or other workers.

. CLOUDFLARE ANALYTICS ENGINE (SQL TELEMETRY)

While Cloudflare collects basic HTTP metrics, Hoox uses Cloudflare Analytics Engine inside `analytics-worker` to track trading-specific variables (e.g. execution slippage, exchange response delays, fee sums).

UNDER-THE-HOOD SQL QUERIES

The `analytics-worker` exposes a SQL interface to query the metrics dataset directly from your Next.js Dashboard:

```
  / Query : Calculate the th percentile latency of Bybit API order fills /
  SELECT
    quantiles(.)(double) as p_latency_ms,
    count() as total_executions
  FROM hoox_telemetry
  WHERE blob = 'trade-worker'
    AND blob = 'bybit'
    AND timestamp >= now() - INTERVAL '' HOUR

  / Query : Aggregate error velocities across all edge isolates /
  SELECT
    blob as worker_name,
    count() as error_count
  FROM hoox_telemetry
  WHERE blob = '' -- Success = false
    AND timestamp >= now() - INTERVAL '' HOUR
  GROUP BY worker_name
```

. LOGGING & R STORAGE OFFLOADING

Because transactional databases like SQLite D have write limits, Hoox implements a dual-tier logging policy:

- High-Value Data: Transaction statuses and fills are written to your D `trades` table.
- Verbose Telemetry: Full, verbose JSON payload exchanges, network headers, and WebSocket stream records are serialized and saved to R Storage using date-based key paths:
  `logs/bybit/BTCUSDT/--/order-.json`

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

. STANDARDIZED ALARM SYSTEMS

If a critical error or execution failure occurs (e.g., an order is rejected due to insufficient margin, or the kill switch is flipped due to drawdown limits):

. The executing worker intercepts the exception using the shared error handler.
. Direct V Service Binding pings `telegram-worker` instantly.
. You receive an emergency alert on your phone.

```
  // Standard alarm notification trigger
  if (orderResponse.status === "Rejected") {
    await env.TELEGRAM_SERVICE.fetch("https://telegram-worker/alert", {
```

```
      method: "POST",
      headers: {
        "Content-Type": "application/json",
        "X-Internal-Auth-Key": env.INTERNAL_KEY_BINDING,
      },
      body: JSON.stringify({
        chatId: env.TELEGRAM_CHAT_ID_DEFAULT,
        message: ` <b>Emergency Order Reject!</b>\nExchange: Bybit\nSymbol: BTCUSDT\nReason:
Insufficient Margin`,
      }),
    });
  }
```

> Tip: Need to tail live worker logs to debug a custom strategy? Run `hoox logs tail trade-worker` in your terminal to open a real-time WebSocket connection to Cloudflare's global edge logging console!

NEXT STEPS

- [Debugging Runbook](../development/debugging.md) - Diagnose local and remote V exceptions using diagnostic profiles.
- [Self-Healing & Repair](../guides/repair.md) - Recover from degraded workers and provision missing bindings.

[SECTION: ZERO TRUST & SECURITY HARDENING]

ZERO TRUST & SECURITY HARDENING

While the Hoox dashboard features a highly secure, custom cookie-based authentication middleware, wrapping your dashboard and API endpoints inside Cloudflare Zero Trust (Access) provides an enterprise-grade security perimeter.

By placing your deployment behind Cloudflare Access, you can enforce Multi-Factor Authentication (MFA), restrict access to specific GitHub/Google SSO identities, evaluate device posture, and drop malicious scanner payloads at the DNS level before they ever hit your workers.

THE ZERO TRUST PROTECTIVE BOUNDARY

- Zero Public Exposure: The dashboard isolate does not evaluate public logins directly.
- MFA Gate: Users are intercepted by a secure Cloudflare authentication card at the nearest edge PoP.
- Zero Cost: Cloudflare's Zero Trust free tier includes up to users, which is more than enough for a personal algorithmic trading desk.

. STEP-BY-STEP DASHBOARD ACCESS SETUP

STEP : ENABLE ZERO TRUST ON YOUR ACCOUNT

. Log in to the Cloudflare Dashboard and click Zero Trust on the sidebar.
. If this is your first time, follow the onboarding prompts to register a unique Team Name (e.g. `alpha-trading.cloudflareaccess.com`).

STEP : CREATE A SELF-HOSTED APPLICATION

. In the Zero Trust dashboard, navigate to Access > Applications and click Add an application.
. Select Self-hosted.
. Application Name: `Hoox Dashboard Cockpit`.
. Session Duration: Select your preference (e.g. ` Hours` to prevent constant login prompts).
. Application Domain: Enter the custom domain mapped to your dashboard worker (e.g., `hoox.my-trading-empire.com`).

STEP : CONFIGURE AUTHORIZATION POLICIES

. Click Next to proceed to the Policies tab.
. Policy Name: `Allow Admin Only`.
. Action: `Allow`.
. Configure Rules:
   - Include: Select Emails and enter your personal email address (enables Email OTP).
   - Include (SSO): Alternatively, select GitHub Org/Teams or Google Workspace to enable SSO integrations.
. In the Require block, you can optionally require a valid security key (MFA) or device posture check (e.g. verifying that your laptop runs a specific OS version).

STEP : MAP IDENTITY PROVIDERS & SAVE

. In Settings > Authentication, link your desired login providers (Google Workspace, GitHub OAuth, or Email OTP).
. Save the application.
. Open your browser and navigate to your custom domain (`https://hoox.my-trading-empire.com`). You will be intercepted by your Cloudflare Access card. Once authorized, you are passed cleanly to your Next.js dashboard.

. STRICT WAF WEBHOOK IP ALLOW-LISTING

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

. Under your Cloudflare DNS zone dashboard, navigate to Security > WAF > Custom Rules.
. Click Create Rule.
. Rule Name: `Restrict /webhook to TradingView IPs`.
. Field: `URI Path` | Operator: `equals` | Value: `/webhook`.
. And: `IP Source Address` | Operator: `is not in` | Value: (Paste TradingView's official IP ranges here, which are automatically synced by running the `hoox waf configure --TradingView-only` command).
. Action: Block (or Challenge).
. Save. All unauthorized traffic hitting `/webhook` is dropped instantly at the DNS edge, preventing any V compute load.

Automated WAF setup via CLI
hoox waf configure --TradingView-only

. OPTIONAL: BYPASSING LOCAL DASHBOARD AUTH

Once your custom domain is wrapped inside Cloudflare Access, the dashboard's built-in login form (`DASHBOARD_USER`, `DASHBOARD_PASS`) becomes redundant.

To streamline access:

. Edit the Next.js `middleware.ts` file inside `workers/dashboard/src/`.
. Toggle the authentication checker to leverage Cloudflare's Access headers:
   ```typescript
   // Next.js Edge Middleware
   export function middleware(request: Request) {
     // Verify the JWT payload injected in headers by Cloudflare Access
     const cfAccessJwt = request.headers.get("Cf-Access-Jwt-Assertion");
     if (cfAccessJwt) {
       // Cloudflare has already authenticated the session. Bypass local login.
       return NextResponse.next();
     }
     // Fallback to local cookie checks...
   }
   ```

NEXT STEPS

- [Next.js Dashboard worker Profile](../workers/dashboard.md) - Review OpenNext compilation and asset bindings.
- [System Observability & Metrics](monitoring.md) - Setup time-series logging and Analytics Engine tables.

[SECTION: LOCAL DEVELOPMENT SETUP]

LOCAL DEVELOPMENT SETUP

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

. DEV RUNTIME SELECTION: NATIVE VS. DOCKER

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

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

THE GUIDED STARTUP FLOW

When you run `hoox dev start`:

. Wrangler Version Verification: Probes your global/local Wrangler package. If Wrangler is outdated, it prompts an advisory upgrade warning:

```
  Wrangler CLI is outdated (v.. < v..)
    Run `bunx wrangler update` to update, or press Enter to continue.
```

. Docker Environment Probing: Checks if the Docker daemon is active and `docker-compose.yml` is present in the workspace.
. Runtime Preference Lock: If both runtimes are available, the CLI prompts you to select your preference. This choice is written to your `wrangler.jsonc` file under the `"dev": { "runtime": "..." }` block, bypassing future prompts.

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

. DOCKER COMPOSE ORCHESTRATION & PROFILES

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

```
 Profile : Spin up all background workers only
docker compose --profile workers up

 Profile : Spin up the Next.js frontend only
docker compose --profile dashboard up

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

. LOCAL PORT MAPPING REGISTRY

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

Microservice	Default Port	Local Binding URL	Purpose
`hoox`	``	`http://localhost:`	Ingress Gateway
`trade-worker`	``	`http://localhost:`	Order Execution Engine
`telegram-worker`	``	`http://localhost:`	Notifications Hub
`d-worker`	``	`http://localhost:`	Relational SQLite Manager
`web-wallet`	``	`http://localhost:`	On-Chain DeFi Node
`dashboard`	``	`http://localhost:`	Next.js Dashboard SSR
`agent-worker`	``	`http://localhost:`	Cron Risk Monitor

. LOCAL ENVIRONMENTAL MOCKING (`.DEV.VARS`)

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

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

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

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

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

NEXT STEPS

- [Testing Standards](testing.md) - Run unit and integration tests using Bun's native test runner.
- [Debugging Runbook](debugging.md) - Debug local and remote isolates, tail logs, and trace queries.

[SECTION: TESTING FRAMEWORK & QA STANDARDS]

TESTING FRAMEWORK & QA STANDARDS

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

Our test suite is powered natively by Bun's high-speed test runner, comprising test files and , individual test assertions split into four distinct diagnostic layers.

THE QA TESTING LAYERS

RUNNING TESTS: CLI COMMANDS

A. CORE PLATFORM VERIFICATION (EXCLUDING LIVE)

```
  . Run all unit, integration, and EE smoke tests in parallel
bun test

  . Run the suite and output a detailed V coverage report
bun test --coverage
```

B. WORKSPACE-SPECIFIC TARGETED RUNS

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

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

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

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

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

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

C. ADVANCED INTEGRATION & LIVE RUNS

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

 Run EE CLI lifecycle smoke tests
bun run test:ee

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

TYPE-SAFE MOCKING SPECIFICATIONS (NO `AS ANY`)

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

```
import { describe, it, expect } from "bun:test";
import type { Env } from "../src/index";

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

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

    // . Execute assertions
    const res = await mockEnv.TELEGRAM_SERVICE.fetch(
      "https://telegram-worker/alert",
```

```
        {
          method: "POST",
          headers: {
            "X-Internal-Auth-Key": "local_secret_token_",
          },
        }
      );

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

CONTINUOUS INTEGRATION GATES & COVERAGE TARGETS

Our GitHub Actions workflow enforces two strict quality gates before any code is approved for production deployment:

. TypeScript Type Safety: All workspaces must compile without errors using `tsc --noEmit`.
. Coverage Thresholds: The monorepo enforces a minimum % coverage threshold across all core execution paths (`packages/cli`, `packages/shared`, `workers/hoox`, `workers/trade-worker`).

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

NEXT STEPS

- [Debugging Telemetry Runbook](debugging.md) - Learn how to trace active V memory, tail logs, and audit SQL execution.
- [Local Development Setup](local-dev.md) - Configure Wrangler and Docker compose to run testbeds.

[SECTION: EDGE DEBUGGING & TELEMETRY]

EDGE DEBUGGING & TELEMETRY

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

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

. REAL-TIME TELEMETRY: TAILING PRODUCTION LOGS

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

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

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

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

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

. DISTRIBUTED TRACING: THE `REQUESTID` STANDARD

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

THE REQUESTID PROTOCOL

. Generation: The `hoox` gateway generates a unique, cryptographically secure UUIDv immediately upon receiving a webhook payload. This is set as the `requestId` in the transaction context.
. Propagation: Every internal service binding call transmits this UUID as the `X-Request-Id` HTTP header.
. Structured Ingestion: When logging error telemetry or writing trade fills to R and D

databases, workers attach the `requestId` as a dedicated index column.
. Unified Auditing: You can audit an entire transaction's lifecycle across all workers by running a single database query filtering by the trace ID:

```
SELECT created_at, worker, message, severity
FROM system_logs
WHERE request_id = 'bdebd-bd-bad-bdd-bdbdcbd'
ORDER BY created_at ASC
```

. OPERATIONAL RUNBOOKS FOR COMMON EDGE FAILURES

A. SYMPTOM: ` UNAUTHORIZED` ON INTERNAL WORKER CALLS

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

B. SYMPTOM: ` BAD GATEWAY` ON WEBHOOK ROUTES

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

C. SYMPTOM: `D_ERROR: NO SUCH TABLE`

- Diagnostics: Worker database transactions fail with table missing errors.
- Primary Root Cause: Relational SQLite schemas were never initialized on your production

D instance.
- Resolution:
  . Initialize database tables and run pending drizzle migrations:
    ```bash
    hoox db apply --remote
    hoox db migrate --remote
    ```

D. SYMPTOM: KV CONFIGURATION PROPAGATION DELAYS

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

NEXT STEPS

- [Testing Framework & QA Standards](testing.md) - Run local unit and integration tests using Bun's native test runner.
- [Self-Healing & System Repair](../guides/repair.md) - Run diagnostics, targeted component repairs, and full system rebuilds.

[SECTION: API ENDPOINT DIRECTORY]

API ENDPOINT DIRECTORY

This directory provides the complete HTTP REST API specification for all public and internal endpoints exposed across the Hoox edge microservices stack.

SECURITY & AUTHORIZATION HEADERS

All internal calls between workers (via V Service Bindings) must provide the standard internal authentication header:

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

Public webhook and dashboard endpoints use cookie authorization or custom API keys:

- Webhook Ingestion: Authenticated using the `apiKey` property inside the JSON payload.
- Telegram webhook: Authenticated via the secure token path: `/telegram/<TELEGRAM_WEBHOOK_SECRET>`.

INGRESS WEBHOOK ENDPOINTS (`WORKERS/HOOX`)

The public gateway acts as the primary firewall and entry entryway for all external trade signals.

A. INGEST SIGNAL WEBHOOK

Receives TradingView alerts or automated cURL signals.

- Path: `/webhook`
- Method: `POST`
- JSON Payload:

```json
{
  "apiKey": "secure_webhook_key_",
  "exchange": "bybit",
  "action": "LONG",
  "symbol": "BTCUSDT",
  "quantity": .,
  "leverage": ,
  "idempotencyKey": "uuid-bdebd-bd"
}
```

  ```
- Success Response ( OK):
  ```json
  {
    "success": true,
    "requestId": "bdebd-bd-bad-bdd-bdbdcbd",
    "exchange": "bybit",
    "symbol": "BTCUSDT",
    "action": "LONG",
    "result": {
      "orderId": "",
      "status": "Filled",
      "price": .
    }
  }
  ```

B. PROACTIVE HEALTH DIAGNOSTICS

- Path: `/health`
- Method: `GET`
- Success Response (OK):
  ```json
  {
    "status": "ok",
    "timestamp": ,
    "bindings": {
      "d": "connected",
      "kv": "connected",
      "queue": "active"
    }
  }
  ```

DATABASE SERVICE ENDPOINTS (`WORKERS/D-WORKER`)

The `d-worker` serves as a private, internal SQL proxy database.

A. EXECUTE SINGLE SQL STATEMENT

- Path: `/query`
- Method: `POST`
  ```

- JSON Payload:
  ```json
  {
    "requestId": "bdebd-bd-bad-bdd-bdbdcbd",
    "query": "SELECT  FROM trades WHERE symbol = ? LIMIT ?",
    "params": ["BTCUSDT", ]
  }
  ```
- Success Response (OK):
  ```json
  {
    "success": true,
    "results": [{ "id": "trade-", "symbol": "BTCUSDT", "price":  }],
    "meta": { "rows_read": , "duration": . }
  }
  ```

B. EXECUTE TRANSACTIONAL BATCH STATEMENTS

- Path: `/batch`
- Method: `POST`
- JSON Payload:
  ```json
  {
    "requestId": "bdebd-bd-bad-bdd-bdbdcbd",
    "queries": [
      {
        "query": "INSERT INTO trades (id, symbol) VALUES (?, ?)",
        "params": ["t-", "ETHUSDT"]
      },
      {
        "query": "UPDATE positions SET size = size + . WHERE symbol = 'ETHUSDT'",
        "params": []
      }
    ]
  }
  ```
- Success Response (OK):
  ```json
  {
    "success": true,
    "results": [{ "meta": { "changes":  } }, { "meta": { "changes":  } }]
  }
  ```

AI RISK & CHAT ENDPOINTS (`WORKERS/AGENT-WORKER`)

Bridges background risk audits and multi-provider AI chat streams.

A. CONVERSATIONAL CHAT STREAM (SERVER-SENT EVENTS)

- Path: `/agent/chat`
- Method: `POST`
- Headers: `Accept: text/event-stream`
- JSON Payload:
  ```json
  {
    "prompt": "Summarize my trade history and average win rate today.",
    "stream": true,
    "provider": "anthropic"
  }
  ```
- Success Response: Emits standard text/event-stream updates, terminating with `data: [DONE]`.

B. MULTIMODAL AI VISION AUDIT

- Path: `/agent/vision`
- Method: `POST`
- JSON Payload:
  ```json
  {
    "imageBase": "iVBORwKGgoAAAANSUhEUgAA...",
    "prompt": "Evaluate this trading chart image for support and resistance levels."
  }
  ```
- Success Response (OK):
  ```json
  {
    "success": true,
    "analysis": "Based on the provided chart screenshot, there is strong horizontal support at $,..."
  }
  ```

NEXT STEPS

- [Request Payload Schemas](payloads.md) - Analyze the complete JSON request schemas and type rules.
- [Standard Response Schemas](responses.md) - Check error factories and normal JSON envelopes.

[SECTION: REQUEST PAYLOAD SCHEMAS]

REQUEST PAYLOAD SCHEMAS

To maintain robust data routing and prevent runtime failures, Hoox enforces a strict payload contract across all internal and public service boundaries. All incoming requests are validated by active JSON Schema or Zod middleware before entering V execution loops.

This document details the exact JSON schemas, TypeScript interfaces, and validation rules for all primary request payloads.

. STANDARD REQUEST ENVELOPE

Every service-to-service invocation (via Service Bindings) wraps its business parameters inside a standardized Request Envelope containing distributed tracing indices:

```
export interface RequestEnvelope<T> {
  requestId: string; // Unique UUIDv distributed trace ID
  payload: T; // Service-specific payload
}

{
  "requestId": "bdebd-bd-bad-bdd-bdbdcbd",
  "payload": {
    "symbol": "BTCUSDT",
    "quantity": .
  }
}
```

. TRADE EXECUTION PAYLOADS (`TRADE-WORKER`)

The execution engine processes two primary types of order payloads:

A. CENTRALIZED EXCHANGE ORDER PAYLOAD (`/WEBHOOK` & `/PROCESS`)

```
export interface CexOrderPayload {
  exchange: "binance" | "bybit" | "mexc"; // Target exchange
  action: "LONG" | "SHORT" | "CLOSE"; // Margin position action
  symbol: string; // Standardized asset ticker (e.g. "BTCUSDT")
  quantity: number; // Order size in contracts or tokens
  leverage?: number; // Optional margin leverage multiplier
  price?: number; // Optional execution limit price
  orderType?: "MARKET" | "LIMIT"; // Default: MARKET
}
```

```
  {
    "exchange": "bybit",
    "action": "LONG",
    "symbol": "BTCUSDT",
    "quantity": .,
    "leverage": ,
    "orderType": "MARKET"
  }
```

B. ON-CHAIN DEFI SWAP PAYLOAD (`/DEX`)

```
  export interface DexSwapPayload {
    chain: "ethereum" | "arbitrum" | "polygon"; // EVM target network
    to: string; // Recipient address or Uniswap Router
    value: string; // Transaction value in Wei (as string)
    data: string; // Encoded contract call data bytes
    gasLimit?: number; // Optional transaction gas limit
  }
```

```
  {
    "chain": "arbitrum",
    "to": "xbecdabeedeacdf",
    "value": "",
    "data": "xacbb..."
  }
```

. DATABASE SQL QUERY PAYLOADS (`D-WORKER`)

The SQLite database proxy accepts parameterized SQL statements to protect against SQL injections:

A. EXECUTE SINGLE SQL STATEMENT

```
  export interface SqlQueryPayload {
    query: string; // Parameterized SQL statement
    params: Array<string | number | boolean | null>; // Binding values
  }
```

```
  {
    "query": "SELECT  FROM trades WHERE symbol = ? AND status = ?",
    "params": ["BTCUSDT", "Filled"]
  }
```

. TELEGRAM NOTIFICATION PAYLOADS (`TELEGRAM-WORKER`)

Used to push structured alerts and charts to your mobile device:

```
export interface TelegramAlertPayload {
  chatId: string; // Target Telegram Chat ID
  message: string; // Formatted message content
  parseMode?: "HTML" | "MarkdownV"; // Default: HTML
}
```

```
{
  "chatId": "",
  "message": "<b>Alert:</b> Position filled at ,.",
  "parseMode": "HTML"
}
```

> Tip: Adding new fields to a payload schema? Remember to update the corresponding type interfaces in `@jango-blockchained/hoox-shared/types` to ensure complete type safety across all workers!

NEXT STEPS

- [API Endpoint Directory](endpoints.md) - Analyze REST routes, headers, and endpoints mappings.
- [Standard Response Schemas](responses.md) - Check success envelopes and error models.

[SECTION: STANDARD RESPONSE SCHEMAS]

STANDARD RESPONSE SCHEMAS

To ensure predictable error handling and parsing across all microservices, Hoox enforces a strict response contract. Every worker responds with a standardized JSON envelope containing tracing details, operation status, results payload, and detailed error models.

This document details the exact JSON templates, types, and error factories utilized in the monorepo.

. STANDARD RESPONSE ENVELOPE

Every HTTP response returned by an edge worker is encapsulated within a unified JSON envelope:

```
export interface ResponseEnvelope<T> {
  success: boolean; // Absolute state of the operation
  requestId: string; // UUIDv trace ID mapped from request
  result?: T; // Operation success metadata payload
  error?: {
    // Detail model present only if success = false
    code: string; // Unique machine-parseable error token
    message: string; // Human-readable error description
    details?: any; // Optional specific array of field issues
  };
}
```

. SUCCESS RESPONSE TEMPLATES

A. TRADE EXECUTION FILL SUCCESS (`TRADE-WORKER` - OK)

```
{
  "success": true,
  "requestId": "bdebd-bd-bad-bdd-bdbdcbd",
  "result": {
    "orderId": "",
    "status": "Filled",
    "executedQty": .,
    "price": .,
    "timestamp":
  },
  "error": null
}
```

B. TELEGRAM PUSH SUCCESS (`TELEGRAM-WORKER` - OK)


```
{
  "success": true,
  "requestId": "bdebd-bd-bad-bdd-bdbdcbd",
  "result": {
    "messageId": ,
    "delivered": true
  },
  "error": null
}
```

. ERROR MODELS & EDGE ERROR CODES

When a worker operation fails, the `success` flag is set to `false`, the `result` field is omitted or set to `null`, and the `error` model is fully populated:

A. BAD REQUEST (JSON VALIDATION ERROR)


```
{
  "success": false,
  "requestId": "bdebd-bd-bad-bdd-bdbdcbd",
  "error": {
    "code": "VALIDATION_ERROR",
    "message": "Invalid JSON payload structure.",
    "details": [{ "field": "quantity", "issue": "Must be greater than zero." }]
  }
}
```

B. CONFLICT (IDEMPOTENCY MUTEX INTERCEPT)


```
{
  "success": false,
  "requestId": "bdebd-bd-bad-bdd-bdbdcbd",
  "error": {
    "code": "DUPLICATE_REQUEST",
    "message": "Order rejected. Durable Object locked: trace ID already processed in the last
hours."
  }
}
```

. SHARED `ERRORS` FACTORY MIDDLEWARE

To enforce this response contract without writing redundant JSON wrappers in every worker, we use the standardized `Errors` factory from the shared monorepo package `@jango-blockchained/hoox-shared/errors`:

```
import { Errors } from "@jango-blockchained/hoox-shared/errors";

// . Validation Failures ( Bad Request)
if (!payload.symbol) {
  return Errors.badRequest("Symbol is a required parameter.");
}

// . Secret Key Mismatch ( Unauthorized)
if (requestHeaderKey !== env.INTERNAL_KEY_BINDING) {
  return Errors.unauthorized("Access Denied: Invalid X-Internal-Auth-Key.");
}

// . System Exceptions ( Internal Server Error)
try {
  await database.write(data);
} catch (err: any) {
  return Errors.internal(`Database write exception: ${err.message}`);
}
```

These factory methods automatically serialize the exception, attach the active `requestId` from the context, set the correct HTTP status code, and return a compliant `Response` object.

NEXT STEPS

- [API Endpoint Directory](endpoints.md) - Analyze REST routes, headers, and endpoints mappings.
- [Request Payload Schemas](payloads.md) - Check JSON request payload specifications and typescript interfaces.

[SECTION: CLI ARCHITECTURE & FEATURES]

CLI ARCHITECTURE & FEATURES

The `hoox` CLI (`packages/cli`) is the unified lifecycle engine of the trading platform monorepo. It governs local sandboxes, compiles TypeScript structures, coordinates Cloudflare infrastructure resources, deploys edge isolates, manages encrypted secrets, and executes self-healing diagnostics.

MONOREPO WORKSPACE DESIGN

The CLI is integrated into our monorepo using Bun Workspaces, which link local packages together. This design ensures that the CLI binary can resolve and load local shared types (`@jango-blockchained/hoox-shared`) and TUI components (`packages/tui`) instantly without network downloads or pre-compilation overhead:

```
hoox-setup/ (Monorepo Root)
|-- packages/
|   |-- cli/        CLI Source code (entry binary: bin/hoox.js)
|   |-- tui/        OpenTUI dashboard source code
|   |-- shared/     Common libraries (auth, router, error models)
|-- workers/
|   |-- ...         Edge V isolates
|-- package.json    Root workspace manager
```

. COMMAND-LINE CORE ARCHITECTURES

The CLI binary parses terminal instructions using the following architectural layers:

A. COMMAND DISPATCHER (`PACKAGES/CLI/SRC/INDEX.TS`)

Intercepts all incoming arguments (e.g. `hoox infra provision`), evaluates global flags (`--json`, `--quiet`), validates configuration integrity, and delegates execution to target command modules under `src/commands/`.

B. CLOUDFLARE API ADAPTERS (`SRC/ADAPTERS/`)

Translates command intentions (like `hoox infra d create`) into parameterized Cloudflare API REST payloads, bypassing wrangler wrappers when high-performance execution is needed.

C. STATE ENGINE (`SRC/CORE/`)

Tracks active project profiles, enabled worker matrices, and workspace configuration formats (`wrangler.jsonc` vs. `wrangler.toml`).

. DECLARATIVE CONFIG MAPPING & VALIDATION

To prevent configuration drift, the CLI enforces strict type validation on `wrangler.jsonc` files:

. Config Interfaces: Built-in parsers validate configuration files against type-safe TypeScript interfaces (`Config` and `WorkerConfig`) defined in `src/core/types.ts`.
. Setup Verification (`hoox check setup`): Compares active workspace profiles against example files, audits environment variables keys, and scans for missing bindings, outputting formatted terminal reports.

. SELF-HEALING & DIAGNOSTICS ENGINE

One of the CLI's most powerful features is its guided repair framework (`hoox repair` command groups):

- Diagnostic Probes: The `hoox repair check` command runs a -step checklist (verifying submodule checkouts, NPM/Bun package resolutions, TypeScript variables, Cloudflare zone links, and encrypted credentials).
- Automated Recovery: If a missing resource is identified (e.g. a D database ID is bound in `wrangler.jsonc` but the database doesn't exist on your Cloudflare account), the repair engine prompts you and provisions it automatically:

Provision missing Cloudflare bindings and repair system states
hoox repair infra

> Tip: Every single command supports the `--json` global flag. This outputs machine-parseable JSON payloads (e.g. `hoox monitor status --json`), allowing you to integrate the CLI with external telemetry dashboards or alert scripts effortlessly!

NEXT STEPS

- [CLI Reference Manual](../../enduser/reference/cli-commands.md) - Review the complete command tree, positional arguments, and flags.
- [Wrangler Setup & Tooling](development/local-dev.md) - Configure Wrangler to bind local D and KV instances for dev testing.