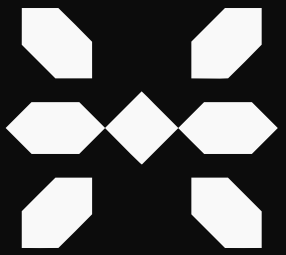
HOOX · AXIS · DOCUMENTATION



PLUGIN AUTHORS · INTEGRATORS

PLUGINS MANUAL

Contracts, registry, sources, streams, engines, storage

Open charting PWA — own the axes, swap the engine

2026-08-03 DIN A4 · HTTPS://HOOX.SH/AXIS/DOCS OPEN SOURCE

CONTENTS

OVERVIEW

title: "Plugins" description: "AXIS unified plugin system: source, stream, engine, storage — contracts, registry, catalogs, and dynamic ES-module loading."

PLUGINS

Abstract

AXIS is a **plugin-composed charting PWA**. Historical bars, live ticks, Pine evaluation, and the script library are not hard-coded product features — they are interchangeable modules that share one contract namespace:

```
pynescript.axis.plugins.v1
```

The shell (AXIS) never embeds a closed interpreter. Evaluation is always an **engine** plugin. Data arrives only through **source** and **stream**. Persistence goes through **storage**. Compose any lawful active set without a proprietary charting host.

Conceptual model

```
flowchart LR
  subgraph Active["Active set"]
    SRC[source]
    STR[stream]
    ENG[engine]
    STO[storage]
  end
  REG[(PluginRegistry)]
  SRC --> REG
  STR --> REG
  ENG --> REG
  STO --> REG
  REG --> UI[Chart · Editor · Results]
  DYN[loadPluginFromUrl] -->|ES module| REG
```

Kind	Role	Required surface
`source`	Historical OHLCV	`fetchHistorical(opts) → Bar[]`
`stream`	Live bar updates	`start(opts) → stop()`
`engine`	Pine / DSL evaluation	`isReady()`, `run(opts) → RunResult`
`storage`	Script library	`list` / `read` / `write` / `remove`
`component`	UI slots (phase 2, reserved)	`mount(slot, el, api)`

Built-ins register at bootstrap (`frontend/src/plugins/bootstrap.ts`). Dynamic plugins install from URL (`loader.ts`) and persist under `localStorage` key `pynescript.axis.plugins.v1`.

Interface surface

Module	Path
Contracts	`frontend/src/plugins/types.ts`
Registry	`frontend/src/plugins/registry.ts`
Bootstrap	`frontend/src/plugins/bootstrap.ts`
Active resolution	`frontend/src/plugins/active.ts`
Dynamic loader	`frontend/src/plugins/loader.ts`
Public barrel	`frontend/src/plugins/index.ts`
Source catalog	`frontend/src/sources/catalog.ts`
Stream catalog	`frontend/src/streams/catalog.ts`
Engine catalog	`frontend/src/engines/catalog.ts`
Storage catalog	`frontend/src/storage/catalog.ts`

Built-in inventory (quick)

Kind	Built-in IDs
Sources	`binance-rest`, `okx-rest`, `bybit-rest`, `coinbase-rest`, `mock-walk`, `csv-upload`
Streams	`binance-ws`, `okx-ws`, `bybit-ws`, `coinbase-ws`, `kraken-ws`, `mock-poll`
Engines	`server`, `pyodide`
Storage	`local`, `cloud`, `git`

Defaults when selection is missing (`active.ts`): source `binance-rest`, stream `binance-ws`, engine `server`, storage `local`.

Invariants

1. **AXIS ≠ engine** — the UI never ships a private Pine runtime; engines are plugins.
2. **Registry is the single source of truth** — catalogs write into `PluginRegistry`; UI lists come from `listSources()` / etc.
3. **Built-ins resist casual unregister** — `unregister*(id)` returns `false` for `builtIn: true` unless `allowBuiltIn: true`.
4. **Config keys** — prefer `pluginKey(kind, id)` → `` `${kind}:${id}` `` (e.g. `engine:server`).
5. **Storage via URL is rejected** — dynamic loader refuses `kind: 'storage'` until a hardened path exists.
6. **Install list persistence** — installed plugins are stored under `pynescript.axis.plugins.v1`.

Compose recipes (plugin view)

Recipe	Source	Stream	Engine	Storage
Offline lab	`mock-walk`	`mock-poll`	`pyodide`	`local`
Desk research	`binance-rest`	`binance-ws`	`server` → Flask	`local`
AXIS at the edge	venue REST	venue WS / DO	`server` → Worker	`cloud`
Repo library	any	any	any	`git`

Failure modes

Symptom	Likely cause
Empty source dropdown	`ensureBuiltins()` never ran at app entry
Dynamic plugin vanishes after reload	URL 404 or CORS; check System Logs (`plugins` channel)
Engine “not ready”	Flask/Worker down (`server`) or missing `/vendor/*.whl` / `/pyodide/` (`pyodide`)
Cloud library 401	Missing Bearer `pn_…` key or unbound `API_KEYS` KV in production

See also

- Contracts
- Registry

- Sources · Streams · Engines · Storage
- Dynamic loader
- Plugin examples
- Worker (cloud storage + stream relay)

CONTRACTS

title: "Plugin contracts" description: "Formal TypeScript interfaces for AXIS source, stream, engine, storage, and component plugins (pynescript.axis.plugins.v1)."

PLUGIN CONTRACTS

Abstract

All AXIS plugins share `PluginBase` and a discriminant `kind`. Contracts live in `frontend/src/plugins/types.ts`. Catalogs and the dynamic loader assert these shapes before registration; the UI never calls plugins that failed assertion.

Namespace (localStorage / config): **`pynescript.axis.plugins.v1`**.

Conceptual model

```
classDiagram
  class PluginBase {
    +id: string
    +name: string
    +kind: PluginKind
    +description?
    +version?
    +builtIn?
    +configSchema?
    +capabilities?
    +init(ctx)?
    +dispose()?
  }
  class SourcePlugin {
    +fetchHistorical(opts)
    +searchSymbols?(query)
  }
  class StreamPlugin {
    +start(opts) stop
  }
  class EnginePlugin {
    +isReady()
    +run(opts)
  }
  class StoragePlugin {
    +list/read/write/remove
    +saveDraft?/loadDraft?
    +sync?/getStatus?
  }
  PluginBase <|-- SourcePlugin
  PluginBase <|-- StreamPlugin
  PluginBase <|-- EnginePlugin
  PluginBase <|-- StoragePlugin
```

Interface surface

PluginKind

```
type PluginKind = 'source' | 'stream' | 'engine' | 'storage' | 'component';
```

PluginBase

Field	Type	Notes
`id`	`string`	Stable registry key within kind
`name`	`string`	UI label
`kind`	`PluginKind`	Discriminant
`description?`	`string`	Manager catalog
`version?`	`string`	Optional semver
`builtIn?`	`boolean`	Protects against unregister
`configSchema?`	`ConfigSchema`	Settings form
`capabilities?`	`PluginCapabilities`	Badges: offline / needsAuth / needsNetwork / needsProxy
`init?` / `dispose?`	lifecycle	Optional host hooks

ConfigSchema / FieldSchema

Each config field:

Field	Values
`type`	`'string' | 'number' | 'boolean' | 'select'`
`default?`	primitive
`label?`, `description?`, `placeholder?`	UI
`min?`, `max?`, `step?`	numbers
`options?`	select strings

PluginContext

Passed to `init` when used:

- `getConfig(): Record<string, unknown>`
- `setStatus(msg, level?)`
- `host.fetch?` — injectable fetch for tests / workers

pluginKey

```
pluginKey(kind, id) → `${kind}:${id}`
```

Used for `store.pluginsConfig` lookup (`active.ts`).

SourcePlugin

```
interface SourceOpts {
  symbol: string;
  interval: string;
  limit?: number;
  config?: Record<string, unknown>;
}

interface SourcePlugin extends PluginBase {
  kind: 'source';
  fetchHistorical(opts: SourceOpts): Promise<Bar[]>;
  searchSymbols?(query: string, config?: Record<string, unknown>): Promise<string[]>;
}
```

Bar shape (from `store/types`): `{ time, open, high, low, close, volume? }` with `time` in **unix seconds**.

Registry asserts: object, `id`, `name`, `kind === 'source'`, `fetchHistorical` is a function.

StreamPlugin

```
interface StreamOpts {
  symbol: string;
  interval: string;
  config?: Record<string, unknown>;
  lastBar?: Bar | null;
  onBar: (b: Bar) => void;
  onError: (e: Error) => void;
  onStatus: (s: {
    state: 'open' | 'closed' | 'reconnecting' | string;
    url?: string;
    detail?: string;
  }) => void;
}

interface StreamPlugin extends PluginBase {
  kind: 'stream';
  start(opts: StreamOpts): () => void; // stop
}
```

Invariant: `start` must return a **stop** function that closes sockets / clears timers. The AXIS calls stop on symbol change or unmount.

EnginePlugin

```
interface EngineOpts {
  script: string;
  bars: Bar[];
  config?: Record<string, unknown>;
  signal?: AbortSignal;
}

interface RunResult {
  status: 'success' | 'error';
  plots: (number | null)[];
  series?: Record<string, (number | null)[]>;
  events: Array<{ time: number; type: string; id?: string; price?: number; dir?: string; [k: string
  drawings?: Array<Record<string, unknown>>;
  error?: string;
  meta?: { mode?; script_id?; run_id?; ms?; overlay?; script_name?; [k: string]: unknown };
}

interface EnginePlugin extends PluginBase {
  kind: 'engine';
  isReady(): Promise<boolean>;
  run(opts: EngineOpts): Promise<RunResult>;
}
```

Notes:

- `plots` is the primary overlay series; multi-series engines also fill `series`.
- Strategy fills / orders appear in `events`.
- Pine line/label/box objects may appear in `drawings` from the interpret runtime.
- Prefer returning `status: 'error'` with `error` string over throwing for UI-friendly paths (built-in engines do both patterns carefully).

StoragePlugin

```
interface ScriptMeta {
  id: string;
  name: string;
  description?: string;
  path?: string;
  updatedAt: number;
  createdAt?: number;
  revision?: string;
  tags?: string[];
}

interface ScriptDocument extends ScriptMeta {
  content: string;
}

interface StoragePlugin extends PluginBase {
  kind: 'storage';
  list(opts?): Promise<ScriptMeta[]>;
  read(id, config?): Promise<ScriptDocument>;
  write(doc, config?): Promise<ScriptMeta>;
  remove(id, config?): Promise<void>;
  saveDraft?(doc, config?): Promise<void>;
  loadDraft?(config?): Promise<{ content: string; name?: string } | null>;
  sync?(direction: 'push' | 'pull' | 'both', config?): Promise<SyncResult>;
  getStatus?(config?): Promise<StorageStatus>;
}
```

Registry requires `list`, `read`, `write`, `remove`. Draft/sync/status are optional.

ComponentPlugin (reserved)

```
interface ComponentPlugin extends PluginBase {
  kind: 'component';
  slots: Array<'manager-tab' | 'results-tab' | 'topbar-action' | 'settings-section'>;
  mount(slot: string, el: HTMLElement, api: Record<string, unknown>): () => void;
}
```

Registered in the registry but not yet driven by the Manager UX. Do not rely on slots in production plugins.

Capabilities

```
interface PluginCapabilities {
  offline?: boolean;
  needsAuth?: boolean;
  needsNetwork?: boolean;
  needsProxy?: boolean;
}
```

Manager badges (`plugin-badges.tsx`) surface these for composition decisions (e.g. offline lab vs desk research).

Invariants & edge cases

1. **Export shape for dynamic modules** — default export, named `plugin`, or the module root object (`loader.asPlugin`).
2. **Config merge** — catalogs merge `configSchema` defaults with runtime `config` / `store.pluginsConfig`.
3. **AbortSignal** — `server` engine honors `signal` / adaptive timeouts; custom engines should too for long runs.
4. **Revision / If-Match** — cloud storage uses revision strings for optimistic concurrency (Worker `If-Match`).

Failure modes

Error	Contract violation
`source: fetchHistorical() required`	Missing method on register
`Unknown plugin kind`	Typo or future kind not supported by loader
`Custom storage plugins via URL are not supported yet`	Dynamic storage blocked by design

See also

- Registry
- Dynamic loader
- Plugin examples

REGISTRY

title: "Plugin registry" description: "PluginRegistry singleton: ordered maps per kind, listeners, built-in protection, and bootstrap wiring."

PLUGIN REGISTRY

Abstract

`PluginRegistry` (`frontend/src/plugins/registry.ts`) is the **single in-memory source of truth** for every installed plugin of every kind. Catalogs, the dynamic loader, active resolution, and Manager UI all read/write through this class (or thin wrappers that call it).

Conceptual model

```
flowchart TB
  BOOT[ensureBuiltins] --> SRC_C[sources/catalog]
  BOOT --> STR_C[streams/catalog]
  BOOT --> ENG_C[engines/catalog]
  BOOT --> STO_C[storage/catalog]
  SRC_C --> REG[registry singleton]
  STR_C --> REG
  ENG_C --> REG
  STO_C --> REG
  LOAD[loadPluginFromUrl] --> REG
  REG --> ACTIVE[getActiveSource/…]
  REG --> UI[PluginManager / topbar]
  REG --> LISTEN[on listeners]
```

Interface surface

Singleton

```

// or from './plugins' barrel
```

Per-kind API (pattern)

Method	Behavior
`registerSource(p)`	Assert contract; set map; append order if new; emit `registered`
`getSource(id)`	Map lookup
`listSources()`	Registration **order**, not insertion-hash order
`unregisterSource(id, { allowBuiltIn? })`	Refuses `builtIn` unless opted in
Same pattern for stream / engine / storage	
`registerComponent` / `listComponents`	Phase-2 components (unordered Map values)

Bulk

Method	Behavior
`register(plugin: AnyPlugin)`	Dispatch on `kind`
`unregister(kind, id, opts?)`	Kind switch
`clear()`	Wipe all maps and order arrays (tests)
`summary()`	`{ sources, streams, engines, storages }` lightweight summaries
`on(listener)`	Subscribe to `{ type, kind, id }`; returns unsubscribe

Order preservation

Private arrays `_sourceOrder`, `_streamOrder`, `_engineOrder`, `_storageOrder` ensure UI dropdowns stay stable when plugins re-register (idempotent `set` without reordering).

Built-in protection

```
if (p.builtIn && !opts?.allowBuiltIn) return false;
```

Dynamic uninstall paths must never silently delete `binance-rest` / `server` / `local`.

Internals (repo paths)

Concern	Path
Class + singleton	`frontend/src/plugins/registry.ts`
Idempotent builtins	`frontend/src/plugins/bootstrap.ts` → `ensureBuiltins()`
Active set	`frontend/src/plugins/active.ts`
Dynamic register helpers	`sources/catalog.ts` `registerDynamicSource`, etc.

Bootstrap

```
// bootstrap.ts — once per page lifetime
ensureSourcesRegistered();
ensureStreamsRegistered();
ensureEnginesRegistered();
ensureStoragesRegistered();
```

Safe to call repeatedly: catalogs use a local `registered` flag; bootstrap uses `done`.

Active resolution defaults

Slot	Default id
source	`binance-rest`
stream	`binance-ws`
engine	`server`
storage	`local`

Fallbacks chain: `store.activePlugins.*` → legacy store fields → default → registry get with hard fallback id → throw if still missing (source/stream/engine). Storage returns `undefined` if nothing registered (should not happen after builtins).

Engine endpoint surface

`getActiveEngineConfig()` injects `store.endpoint` when the engine has an `endpoint` config field or id is `server` — keeps the topbar Endpoint field and engine config aligned.

Invariants & edge cases

1. **Re-register same id** — overwrites the plugin object; does **not** duplicate order entry.
2. **Listener errors are swallowed** — one bad subscriber cannot break registration.
3. **`clear()` does not reset bootstrap/catalog flags** — tests use `_resetBootstrapFlag` / `_reset*RegistrationFlag` helpers.
4. **Components** — no order array; `listComponents` is Map iteration order.

Worked examples

Register a test source

```
registry.registerSource({
  id: 'fixture',
  name: 'Fixture',
  kind: 'source',
  async fetchHistorical() {
    return [{ time: 1, open: 1, high: 1, low: 1, close: 1 }];
  },
});
```

Listen for dynamic installs

```
const off = registry.on((ev) => {
  if (ev.type === 'registered' && ev.kind === 'engine') {
    console.log('engine ready', ev.id);
  }
});
// later: off()
```

Failure modes

Symptom	Cause
`source: kind must be 'source'`	Wrong discriminant on register
Dropdown empty after `clear()` in test	Forgot re-`ensureBuiltins` / reset flags
Unregister returns false	Built-in protection

See also

- Contracts
- Dynamic loader
- Plugins hub

SOURCES

title: "Sources" description: "Built-in historical data sources: venue REST, mock walk, CSV upload — contracts, config, and fallbacks."

SOURCES

Abstract

A **source** loads a finite window of OHLCV bars for the chart and engine. Sources live in `frontend/src/sources/catalog.ts` and register into the unified registry via `ensureSourcesRegistered()`.

Conceptual model

```
sequenceDiagram
  participant UI as AXIS
  participant ACT as getActiveSource
  participant SRC as SourcePlugin
  participant NET as Venue API / IDB file
  UI->>ACT: load symbol
  ACT->>SRC: fetchHistorical({symbol, interval, config})
  SRC->>NET: HTTP / memory
  NET-->>SRC: raw candles
  SRC-->>UI: Bar[]
```

Built-in catalog

id	Name	Network	Notes
`binance-rest`	Binance REST	yes	Public klines; **synthetic walk fallback** when `fallback: true` (default)
`okx-rest`	OKX REST	yes	Candles; symbol `BTCUSDT` → `BTC-USDT`; max 300 bars
`bybit-rest`	Bybit REST	yes	Spot v5 klines; newest-first reversed
`coinbase-rest`	Coinbase REST	yes	Exchange candles; `USDT` pair rewritten to `USD` product
`mock-walk`	Mock Walk	no	Offline random walk; optional Mulberry-like seed
`csv-upload`	CSV / JSON Upload	no	Reads last upload from `upload-store`

UI order is `BUILTIN_SOURCES` array order (Binance first).

Interface surface

Every source implements `SourcePlugin` (contracts):

```
fetchHistorical({ symbol, interval, limit?, config? }): Promise<Bar[]>
```

Config highlights

binance-rest

Key	Default	Meaning
`baseUrl`	`https://api.binance.com`	Override for mirrors/proxies
`limit`	`500`	50–1000
`fallback`	`true`	On network error, `synthesizeWalk`

mock-walk

Key	Default	Meaning
`seed`	`0`	`0` = non-deterministic; non-zero = deterministic PRNG
`startPrice`	`100`	Walk origin
`limit`	`500`	Bar count

csv-upload

- No schema fields. User must Upload a CSV (`time,open,high,low,close[,volume]`) or JSON array first; otherwise throws a clear error.

Internals

Path	Role
`frontend/src/sources/catalog.ts`	Definitions + register/list/get
`frontend/src/sources/upload-store.ts`	In-memory last upload for `csv-upload`
`frontend/src/data/load-symbol.ts`	App pipeline that calls active source
`frontend/src/data/parse-bars.ts`	CSV/JSON normalization

Interval → venue codes

Helpers map AXIS intervals (`1m`, `5m`, `15m`, `1h`, `4h`, `1d`, `1w`) to OKX `bar`, Bybit interval codes, Coinbase granularity seconds. Unmapped intervals fall back to daily-ish defaults — check venue docs if you add exotic TFs.

Dynamic registration

```
registerDynamicSource(plugin)  // from catalog / loader
unregisterDynamicSource(id)
listDynamicSourceIds()
```

Loader path: `kind === 'source'` + `fetchHistorical` required.

Invariants & edge cases

1. **Time unit** — always **seconds** unix (Binance ms ÷ 1000).
2. **Binance fallback** — offline demos “work” but are **not** real prices; disable `fallback` for strict desk research.
3. **CORS** — browser → public venue APIs must allow CORS; corporate proxies may require a Worker `/api/proxy` (not shipped as a general proxy today — plan carefully).
4. **Newest-first venues** (OKX, Bybit, Coinbase) — catalog reverses/sorts ascending by `time` for the chart.

Worked example

```
// Minimal custom source (ES module for loader)

  id: 'static-two',
  name: 'Two Bars',
  kind: 'source',
  async fetchHistorical() {
    return [
      { time: 1_700_000_000, open: 1, high: 2, low: 0.5, close: 1.5, volume: 10 },
      { time: 1_700_000_060, open: 1.5, high: 2, low: 1, close: 1.2, volume: 12 },
    ];
  },
};
```

Failure modes

Error / symptom	Fix
`No uploaded file…`	Use Upload before selecting `csv-upload`
Empty chart + Binance down	Expected synthetic walk if fallback on
OKX `code !== '0'`	Symbol format / region block
CORS blocked	Proxy or different source

See also

- Streams
- Contracts
- Plugin examples (CoinGecko)

STREAMS

title: "Streams" description: "Live bar plugins: venue WebSockets, mock poll, and Cloudflare Durable Object relay example."

STREAMS

Abstract

A **stream** advances the chart's present. It opens a connection (or timer), emits `onBar` updates, and returns a **stop** function. Built-ins live in `frontend/src/streams/catalog.ts`. An optional Cloudflare Durable Object relay lives as a loadable example (`example-cf-do-stream.js`) plus Worker `SessionDO`.

Conceptual model

```
flowchart LR
  STR[StreamPlugin.start]
  STR -->|WebSocket| VENUE[Venue WS]
  STR -->|setInterval| MOCK[mock-poll]
  STR -->|wss Worker| DO[SessionDO]
  VENUE --> onBar
  MOCK --> onBar
  DO --> onBar
  onBar --> CHART[Chart series]
```

Built-in catalog

id	Name	Offline	Upstream
`binance-ws`	Binance WebSocket	no	`wss://stream.binance.com:9443/ws/{symbol}@kline_{interval}`
`okx-ws`	OKX WebSocket	no	`wss://ws.okx.com:8443/ws/v5/business` candle channels
`bybit-ws`	Bybit WebSocket	no	Public linear/spot kline topics
`coinbase-ws`	Coinbase WebSocket	no	Exchange WS ticker/candles path
`kraken-ws`	Kraken WebSocket	no	Public OHLC channel
`mock-poll`	Mock Poll	**yes**	1s synthetic updates from `lastBar`

UI order: venues first, `mock-poll` last (`BUILTIN_STREAMS`).

Interface surface

```
start(opts: StreamOpts): () => void
```

Callbacks:

Callback	Use
`onBar`	OHLCV update (seconds)
`onError`	Hard failure (connect / parse)
`onStatus`	`{ state: 'open' | 'closed' | …, url?, detail? }`

Config is usually empty for venue streams; custom streams may declare `configSchema` (e.g. DO endpoint).

Internals

Path	Role
`frontend/src/streams/catalog.ts`	Built-ins + register helpers
`frontend/src/streams/binance.ts`	Shared Binance helpers (if split)
`frontend/src/streams/multiplex.ts`	Multi-symbol helpers
`frontend/worker/src/durable-objects/session.ts`	`SessionDO` fan-out
`frontend/public/plugins/example-cf-do-stream.js`	Loadable DO client stream

Binance kline mapping

```
onBar({
  time: Math.floor(k.t / 1000),
  open: +k.o, high: +k.h, low: +k.l, close: +k.c, volume: +k.v,
});
```

Open bars may update repeatedly for the same `time` until the kline closes — chart series should treat same-time updates as replace, not append-only.

mock-poll behavior

- Aligns bar `time` to interval slots.
- Updates high/low/close within the open slot; advances on slot change.
- Immediate first tick so live mode feels responsive offline.

Cloudflare DO stream (example plugin)

Not built-in. After load:

1. Config `endpoint` = Worker origin (`https://…workers.dev` or local `http://127.0.0.1:8787`).
2. Client opens `wss://…/api/stream?session=…&symbol=…&interval=…` .
3. Worker routes to `SessionDO` ; DO opens one Binance upstream and broadcasts raw kline JSON.

Requires `SESSIONS` Durable Object binding (often commented in `wrangler.toml` until provisioned).

Reconnect & status honesty

Built-in venue streams use `openReconnectableWs` (`frontend/src/streams/reconnect-ws.ts`):

- Exponential backoff (1s base, 30s cap, 8 attempts default).
- `onStatus({ state: 'reconnecting', detail })` between attempts.
- Hard `onError` only on construct failure or reconnect exhausted.
- Multiplex keeps stream green **only** after `state: 'open'` ; starts as `connecting` .

Optional `Bar.closed` (Binance `k.x` , OKX confirm, Bybit `confirm`) feeds live re-run mode `bar-close` (settings). Multiplex also treats **bar time advance** as a close for venues without a flag.

Invariants & edge cases

1. **Always return stop** — stop cancels reconnect timers and closes the socket.
2. **Symbol casing** — Binance lowercases stream symbols; OKX uses `BTC-USDT` inst ids.
3. **Browser WS limits** — many tabs × many symbols hit connection caps; DO relay amortizes upstream sockets.
4. **Reconnect is built-in for venue WS** — mock-poll does not reconnect (timer only).

Failure modes

Symptom	Cause
Stream never opens	CORS N/A for WS; check firewall / venue geo blocks
DO stream 503 `NO_DO`	`SESSIONS` binding missing
Silent no bars	Message shape mismatch (non-kline payloads)

See also

- Sources
- Worker durable objects
- Plugin examples

ENGINES

title: "Engines" description: "Calculation engines: server (Flask/Worker) and client Pyodide — run contracts, assets, and readiness."

ENGINES

Abstract

An **engine** evaluates a script against bars and returns plots, series, events, and optional drawings. AXIS ships two built-ins in `frontend/src/engines/catalog.ts`:

id	Name	Where Python runs
`server`	Server-Side	External Flask Pro API **or** Cloudflare Worker that proxies/evaluates
`pyodide`	Client-Side (Pyodide)	Browser WebAssembly via self-hosted Pyodide

There is **no** third built-in engine in AXIS itself. Worker in-process Pyodide is a **separate**, feature-gated path on the edge (`PYODIDE_IN_WORKER`) and is not production-ready — see Worker runtime.

Conceptual model

```
flowchart TB
  RUN[engine.run]
  RUN -->|server| HTTP["POST {endpoint}/run?mode="]
  HTTP --> FLASK[Flask :5002]
  HTTP --> WRK[Worker /api/run]
  WRK -->|EXTERNAL_BACKEND| FLASK
  WRK -.->|PYODIDE_IN_WORKER planned| EDGE[In-worker Pyodide]
  RUN -->|pyodide| WASM[Browser Pyodide + wheels]
```

Interface surface

```
isReady(): Promise<boolean>
run({ script, bars, config?, signal? }): Promise<RunResult>
```

`RunResult` is defined in contracts.

server engine

Config schema

Key	Default	Notes
`endpoint`	`http://localhost:5002`	Overridden by topbar / `store.endpoint`
`mode`	`interpret`	`interpret` | `compile`

HTTP contract

- `POST ${endpoint}/run?mode=…`
- Body: `{ script, data: bars }`
- Success JSON: `plots` , `series?` , `events?` , `drawings?` , `meta?` , `mode` , `script_id` , `run_id`
- Error: non-OK status or `status: 'error'` → mapped to `RunResult` with `error`

WebSocket contract (preferred when available)

- URL: `ws(s)://host/ws/run` (derived from `endpoint` ; requires `flask-sock` on the Pro API)
- Client → `{ type: "run", id, script, data, mode? }`
- Server → `{ type: "result", id, status: "success", plots, series, … }` or `{ type: "error", id, message }`
- AXIS server engine tries WS first (`preferWs` , default true), then REST. Health `GET /` reports `"websocket": true` when the route is mounted.
- `meta.transport` is `"ws"` or `"rest"` so the Connection HUD can show the path used.

Readiness: `GET ${endpoint}/` within 8s.

Timeouts: adaptive `min(180s, max(60s, 30s + bars×80ms))`.

Note on Worker path: the PWA's `server` engine posts to `/run` on whatever endpoint you set. The Worker exposes **`POST /api/run`**. When pointing the PWA at a Worker, either put a reverse-proxy path rewrite in front, or set the endpoint so the engine path matches your deployment. Worker `handleRun` validates `{ script, data, mode? }` and **today prefers proxying to `EXTERNAL_BACKEND`** (Flask). In-worker Pyodide only if `PYODIDE_IN_WORKER=enabled` and the wheel pipeline works.

pyodide engine

Config

Key	Default
`indexUrl`	`/pyodide/v0.26.2/` (self-hosted)

Boot pipeline (`_ensure`)

1. Prefetch assets (`prefetchPyodideAssets`) — wasm, stdlib zip, micropip wheels.
2. `loadPyodide({ indexURL })`.
3. `micropip.install` same-origin `/vendor/pynescript-0.2.0-py3-none-any.whl` and antlr runtime wheel.
4. Fetch `/pyodide/pynescript_runtime.py` and `runPythonAsync` it.

Refreshing the pyne wheel (after pyne compiler/runtime changes):

```
# from axis repo — builds sibling ../pynescript (or PYNE_ROOT) and vendors the wheel
./scripts/sync-pyne-wheel.sh
bun run build
```

Keep the hard-coded wheel filename in `src/engines/catalog.ts` / `index.js` in sync if the package version changes. The browser bridge is interpret-first; `mode=compile` / `auto` uses the wheel's `pynescript.compiler` when NumPy loads (object-mode works without Numba; pure-numeric compile still needs server/Numba). 5. `run` calls `run_script(script, bars)` in Python and `JSON.parse` s the result.

Guards: `assertZipAsset` rejects HTML SPA fallbacks (classic deploy footgun when `public/vendor` is missing from `dist/`).

Capabilities: `{ offline: true, needsNetwork: false }` after assets are cached same-origin.

Helpers: `preloadPyodide()` , `LOCAL_PYODIDE_VERSION = '0.26.2'` .

Internals

Path	Role
`frontend/src/engines/catalog.ts`	`serverEngine`, `pyodideEngine`, registration
`frontend/src/indicators/runner.ts`	UI run orchestration
`frontend/public/vendor/*.whl`	Shipped wheels
`frontend/public/pyodide/`	Self-hosted Pyodide + runtime py
`frontend/worker/src/runtime.ts`	Edge `/api/run`
`frontend/worker/RUNTIME.md`	In-worker Python plan

Invariants & edge cases

1. **AXIS never imports pynescript Python** except through engines.
2. **Abort** — pass `signal` from UI cancel; server engine respects it.
3. **Error as data** — both engines often return `status: 'error'` instead of throwing so the Results panel can show messages.
4. **Pyodide size** — ~14MB self-hosted index; first ready can take seconds; preload on idle.

Worked examples

Desk research (Flask)

- Active engine: `server`
- Endpoint: `http://127.0.0.1:5002`
- Backend: `make run` (Flask Pro API)

Offline lab

- Source `mock-walk`, stream `mock-poll`, engine `pyodide`, storage `local`
- Requires `dist/` (or Vite public) to serve `/pyodide/**` and `/vendor/**`

Tiny non-Python engine

See plugin examples — `example-tiny-pine-engine.js` implements a JS DSL with `sma` / `ema` / `rsi` for demos without PYNE.

Failure modes

Symptom	Cause
`pynescript wheel returned HTML`	SPA fallback; deploy vendor into dist; static server must not rewrite `.whl`
`loadPyodide not available`	Missing pyodide.js / blocked script
HTTP error from server	Flask down; wrong endpoint; CORS
Worker 503 `NO_BACKEND`	`EXTERNAL_BACKEND` empty and Pyodide path disabled/failed

See also

- Worker runtime
- Build and serve
- PYNE runtime

STORAGE

title: "Storage" description: "Script library backends: local (IndexedDB), cloud (Worker /api/scripts), and git (GitHub/GitLab)."

STORAGE

Abstract

Storage plugins persist the user's Pine library and drafts. Built-ins: `local`, `cloud`, `git` — registered from `frontend/src/storage/catalog.ts`. High-level UI/editor APIs go through `frontend/src/storage/service.ts`, which always dual-writes drafts to **local** for crash recovery.

Dynamic `kind: 'storage'` install via URL is **not supported** (`loader.ts` throws).

Conceptual model

```
flowchart TB
  SVC[storage/service]
  SVC --> ACTIVE[getActiveStorage]
  ACTIVE --> LOCAL[local IDB/LS]
  ACTIVE --> CLOUD[cloud Worker API]
  ACTIVE --> GIT[git GitHub/GitLab]
  SVC -->|draft always| LOCAL
```

Built-in plugins

local (`frontend/src/storage/local.ts`)

Concern	Detail
Primary	IndexedDB `pynescript.axis.storage` — stores `scripts` , `kv`
Fallback	`localStorage` keys `pynescript.axis.library.v1` , draft keys
Migration	Older library keys (`pynescript.axis.library.v1` , editor docs)
Memory	In-process maps when neither IDB nor LS exist (tests/SSR)
Capabilities	Offline-friendly

cloud (`frontend/src/storage/cloud.ts`)

Concern	Detail
Transport	`fetch` to Worker `/api/scripts`
Auth	`Authorization: Bearer <api_key>` (`pn_…`)
Config	`endpoint` (default `http://127.0.0.1:8787`), `apiKey`
Concurrency	`If-Match` revision headers on write
Partition	Worker hashes key → `userId` ; rows scoped per user

git (`frontend/src/storage/git.ts`)

Concern	Detail
Providers	GitHub Contents API / GitLab repository files
Config	`provider` , `token` , `owner` , `repo` , `branch` , `basePath` (default `pine-library`), commit template
Commit boundary	**Explicit Save only** — not every keystroke
Drafts	`saveDraft` / `loadDraft` are no-ops; local dual-write handles drafts
Capabilities	`needsNetwork` , `needsAuth`

Interface surface

Contract: StoragePlugin.

service.ts API (UI-facing)

Function	Behavior
`listScripts(prefix?)`	Active backend list
`readScript` / `writeScript` / `removeScript`	CRUD + log
`saveDraft` / `loadDraft`	Local first; also active if supported
`exportLibraryJson` / `importLibraryJson`	Portable backup
`getStorageStatus`	Connected / dirty / remote hints

Catalog helpers

```
ensureStoragesRegistered()
getStorage(id)
listStorages()
registerDynamicStorage(plugin)  // in-process only (tests / future)
unregisterDynamicStorage(id)
```

Internals

Path	Role
`storage/catalog.ts`	Registration
`storage/local.ts`	IDB + LS
`storage/idb.ts`	Promise wrappers for IDB
`storage/cloud.ts`	Worker client
`storage/git.ts`	Orchestration
`storage/git-github.ts` / `git-gitlab.ts`	Provider APIs
`storage/git-config.ts`	Config normalize
`storage/service.ts`	Facade
`worker/src/scripts.ts`	Cloud API implementation
`worker/schemas/scripts.sql`	D1 schema

D1 schema (cloud)

```
PRIMARY KEY (user_id, id)
-- scripts + script_drafts tables
```

Without D1, Worker keeps per-isolate in-memory maps (fine for `wrangler dev`).

Invariants & edge cases

1. **Draft recovery** — never rely solely on remote drafts; service always hits local.
2. **Git PAT scope** — GitHub `contents:write`; GitLab `api` / `write_repository`.

3. **Revisions** — local uses `local-${timestamp}`; cloud/git use remote revisions for conflict detection.
4. **Active default** — `getActiveStorageId()` → `local` when unset.

Worked examples

Export / import

```
const docs = await exportLibraryJson();
// …move machine…
await importLibraryJson(docs, { forceNewIds: true });
```

Point cloud at local Worker

1. `cd frontend/worker && npm run dev` → `:8787`
2. Manager → storage `cloud`
3. Config endpoint `http://127.0.0.1:8787`, key from `/api/keys` or `ALLOW_OPEN_KEYS=1` demo key

Failure modes

Error	Cause
`Cloud storage requires an API key`	Empty `apiKey`
401 `NO_KEY` / `INVALID_KEY`	Auth path; see Worker auth
409 on write	`If-Match` revision conflict
Git 401/404	Token, owner/repo, or basePath wrong
Quota errors on localStorage	Large libraries should prefer IDB (automatic when available)

See also

- Worker data plane
- Worker auth
- Contracts

DYNAMIC LOADER

title: "Dynamic plugin loader" description: "Install ES-module plugins from URL, persist installed list, restore on boot, and URL safety rules."

DYNAMIC PLUGIN LOADER

Abstract

The loader (`frontend/src/plugins/loader.ts`) lets users install **third-party or example** source/stream/engine plugins at runtime via dynamic `import()` . Installed URLs persist in `localStorage` under:

Key	Role
`pynescript.axis.plugins.v1`	Current
`pynescript.axis.plugins.v1`	Installed plugin list

On boot, `restoreInstalledPlugins()` re-imports every saved URL.

Conceptual model

```
sequenceDiagram
  participant M as Manager UI
  participant L as loadPluginFromUrl
  participant I as import(url)
  participant R as registry / catalogs
  participant LS as localStorage
  M->>L: URL
  L->>L: normalize + assertSafe
  L->>I: dynamic import
  I-->>L: module
  L->>R: registerDynamic*
  L->>LS: writeInstalled
  Note over L,LS: restoreInstalledPlugins on boot
```

Interface surface

```
loadPluginFromUrl(url: string): Promise<InstalledPlugin>
removePlugin(id: string, kind?: string): void
getInstalledPlugins(): InstalledPlugin[]
restoreInstalledPlugins(): Promise<void>
normalizePluginUrl(url: string): string
assertSafePluginUrl(href: string): void
PLUGINS_KEY // 'pynescript.axis.plugins.v1'
```

InstalledPlugin

```
{ url, id, name, kind, description? }
```

Module export shapes accepted

1. `export default plugin`
2. `export const plugin = …`
3. Module namespace object that *is* the plugin

Must include `id` and `kind`. Required methods:

kind	Method
`source`	`fetchHistorical`
`stream`	`start`
`engine`	`run`
`storage`	**Rejected** — “not supported yet”

Internals

URL normalization

```
// /src/plugins/foo.js → /plugins/foo.js
href.replace(/(^|\/)src\/plugins\//, '$1plugins/');
```

Dev-only Vite paths never ship in `dist/`; production examples live under `public/plugins/` → `/plugins/…`.

Safety

`assertSafePluginUrl` rejects:

- `javascript:`
- `vbscript:`
- `data:text/html…`

This is **not** a full sandbox. Dynamic plugins run with the page’s privileges (network, DOM). Treat plugin URLs like executable code supply chain.

Dedup on install

List filters out same URL **or** same `(kind, id)` before append — reloading an updated module replaces the entry.

Unregister

`removePlugin` calls `unregisterDynamic*` for the resolved kind (or all kinds if unknown) and rewrites the installed list.

Bootstrap coupling

`loadPluginFromUrl` / `restoreInstalledPlugins` call `ensureBuiltins()` first so catalogs exist before dynamic ids land.

Manager UX (product)

From `frontend/src/plugins/README.md`:

1. **Manager** → **Plugins** → **Load from URL**
2. Example: `http://localhost:8081/plugins/example-coingecko-source.js`
3. **Export installed** / **Import…** for machine migration
4. Auto-activate patterns for source/stream/engine after install (Manager catalog **Use**)

Shipped examples (also under `public/plugins/`):

File	Kind
`example-coingecko-source.js`	source
`example-tiny-pine-engine.js`	engine
`example-cf-do-stream.js`	stream

Invariants & edge cases

1. **CORS on module URL** — the JS file origin must allow module import (same-origin is easiest).
2. **CORS on plugin's own API** — separate problem; may need proxy (see plugins README).
3. **Partial restore** — one bad URL logs error and continues others.
4. **No code signing** — trust the URL owner.

Worked example

```
# After bun run build + axis_pwa_server.py
# Load:
http://127.0.0.1:8081/plugins/example-coingecko-source.js
```

Then select **CoinGecko** in the source picker (id `coingecko`).

Failure modes

Error	Meaning
`URL required`	Empty input
`Plugin URL scheme not allowed`	Dangerous scheme
`Module did not export a plugin object`	Wrong export shape
`Plugin needs id and kind`	Incomplete object
`Source plugin needs fetchHistorical()`	Incomplete source
`Unknown plugin kind`	Typo / component
Restore log errors	404, network, syntax error in remote JS

See also

- Plugin examples
- Registry
- CORS and origins