

PYNE: An Open Dual-Engine Toolchain for Offline Pine Script Evaluation

jango_blockchained*

HOOX Project contributors

Independent Open Source Research

<https://hoox.sh> · <https://github.com/hoox-sh/pyne>

2026

Abstract

Domain-specific languages (DSLs) for technical analysis are often tightly coupled to proprietary charting platforms, which limits reproducible research, continuous integration, and editor-centric development workflows. We present PYNE (PyPI package `hoox-pyne`, `import name pynescrypt`), an independent, unofficial open toolchain that models Pine Script™ (v5–v6) as an inspectable of-line pipeline: source text is lexed and parsed with ANTLR4, lowered to an ASDL abstract syntax tree (AST), and evaluated under a deterministic bar-loop runtime with a dual execution engine—an AST interpreter and a NUMBA-backed compiler with nopython kernels and object-mode fallback. The same AST underpins a desk CLI, a Language Server Protocol (LSP) implementation with hundreds of builtins, a Flask Pro API, editor integrations, and edge/browser workers. We formalize series semantics, `na` propagation, and strategy fill behaviour as implemented by the runtime; describe warm-compile caching and security policy for `request.security`; and report internal performance measurements (2026) showing order-of-magnitude gains for numeric scripts under the compile path relative to interpretation. PYNE is licensed under AGPL-3.0-or-later and is part of the HOOX open trading stack. It is not affiliated with TradingView.¹

Keywords: domain-specific languages, financial time series, dual-engine runtimes, language toolchains, Pine Script, open source compilers

Contents

1	Introduction	4
1.1	Motivation	4
1.2	Contributions	4
1.3	Paper roadmap	4
2	Background	5
2.1	Bar model and execution cadence	5
2.2	Series semantics	5
2.3	The <code>na</code> value and propagation	5
2.4	Persistence: <code>var</code> and <code>varip</code>	5
2.5	Indicators versus strategies	6
2.6	Illustrative program	6

*Correspondence: contact@hoox.sh

¹Pine Script™ and TradingView® are trademarks of TradingView, Inc. PYNE is an independent, unofficial implementation and is not affiliated with, authorized by, sponsored by, or endorsed by TradingView, Inc.

3	System Architecture	6
3.1	Design goals	6
3.2	Layered overview	6
3.3	Pipeline notation	7
3.4	Hand-edited versus generated code	7
3.5	Shared AST between editor and Pro API	8
4	Language Front-End	8
4.1	Grammar	8
4.2	ASDL AST	8
4.3	Round-trip fidelity	9
4.4	Linters and type system	9
4.5	Builtin metadata	9
5	Dual-Engine Runtime	9
5.1	Execution modes	9
5.2	Bar-loop data flow	9
5.3	Warm compile and IR cache	10
5.4	Incremental technical analysis kernels	10
5.5	Strategy engine	11
5.6	Security policy for <code>request.security</code>	11
5.7	Response envelope and parity harness	12
6	Product Surfaces	12
6.1	CLI	12
6.2	Language server	12
6.3	Pro API	13
6.4	Editors and edge	13
6.5	Surface comparison	13
7	Evaluation and Deployment Experience	13
7.1	Microbenchmark methodology	14
7.2	Front-end performance	14
7.3	Interpret versus compile	14
7.4	Memory: series cap and ring buffer	14
7.5	Compatibility posture	15
7.6	Test suite scale	15
7.7	Deployment lessons	15
8	Related Work	16
8.1	DSL toolchains and compiler construction	16
8.2	Financial and statistical time-series languages	16
8.3	Dual interpreter + JIT systems	16
8.4	Language servers and editor ecosystems	16
8.5	Edge and browser runtimes	16
9	Discussion and Limitations	17
9.1	No full platform parity	17
9.2	Foreign <code>request.security</code> policy	17
9.3	Compile surface incompleteness	17
9.4	AGPL implications	17
9.5	Trademark and independence	17

9.6 Threats to validity for benchmarks	17
10 Conclusion	17

1 Introduction

1.1 Motivation

Pine Script™ is a widely used domain-specific language for indicators and trading strategies [TradingView, Inc., 2024, Murphy, 1999]. In common practice, scripts are authored and executed inside a browser-locked host environment: the editor, chart surface, historical data plane, and execution engine are provided as a single product surface. While this coupling optimizes interactive charting, it creates friction for several engineering activities that are standard elsewhere in software:

- **Reproducible offline evaluation.** Batch backtests, property-based tests, and continuous integration require deterministic execution without a live UI session.
- **Editor-native tooling.** Language servers [Microsoft, 2023, Microsoft DevBlogs, 2016] expect parse trees, diagnostics, and symbol tables in-process or over a local protocol, not only inside a remote web IDE.
- **Program analysis and transformation.** Static linting, formatting-as-unparsing, and compile-time lowering benefit from a first-class AST rather than opaque host evaluation.
- **Heterogeneous deployment.** The same evaluation contract is useful at the desk (CLI), on a server (API), at the edge (workers), and in the browser (PYODIDE/AXIS).

PYNE addresses these needs by treating Pine Script™ as a language with a classical compiler pipeline [Aho et al., 2006, Cooper and Torczon, 2011, Nystrom, 2021] rather than solely as a feature of a charting application. The project is an *independent, unofficial* implementation: it aims for interoperability with public language documentation and community scripts, not for binary compatibility with proprietary platform internals [HOOX Project, 2026b,a].

1.2 Contributions

This paper makes the following contributions:

1. **System architecture.** A layered architecture in which editors, LSP, core package, dual-engine runtime, Pro API, and edge workers share one AST and one evaluate contract (Section 3).
2. **Language front-end.** An ANTLR4 grammar approximating Pine Script™ v5–v6, ASDL-generated nodes, visitor/transformer patterns, and lossless parse↔unparse round-trip (Section 4; grammar and series details are expanded in sister papers of this series).
3. **Dual-engine runtime.** A bar-loop host with modes $\in \{\text{auto}, \text{compile}, \text{interpret}\}$, warm IR cache, object-mode fallback, and a uniform response envelope for plots, drawings, strategy events, and alerts (Section 5).
4. **Product surfaces.** CLI, LSP (~800+ builtins), VS Code extension, Flask Pro API, and editor/edge clients built on the shared core (Section 6).
5. **Evaluation experience.** Internal benchmarks (2026) for parse, unparse, interpret, and compile paths; compatibility posture; and deployment lessons (Section 7).

1.3 Paper roadmap

Section 2 reviews the Pine Script™ bar model and series semantics. Section 3 presents the end-to-end system. Sections 4–6 detail the front-end, runtime, and product surfaces. Section 7 reports measurements and operational experience. Section 8 situates PYNE among DSL toolchains and financial languages. Section 9 discusses limitations and licensing implications. Section 10 concludes.

2 Background

2.1 Bar model and execution cadence

Pine Script™ programs are not general-purpose stream processors with arbitrary event loops. They are evaluated against a finite sequence of market bars. Let a bar series of length n be

$$B = (b_0, b_1, \dots, b_{n-1}), \quad b_i = (o_i, h_i, l_i, c_i, v_i, t_i), \quad (1)$$

where o, h, l, c, v, t denote open, high, low, close, volume, and timestamp (OHLCV plus time). The integer *bar index* $i \in \{0, \dots, n-1\}$ is exposed to scripts as `bar_index`.

Definition 2.1 (Bar-loop evaluation). Given a script P and bar sequence B , bar-loop evaluation produces a run trace $\mathcal{T}$ by invoking the script body once for each $i = 0, \dots, n-1$ in increasing order, with built-in series (open, high, low, close, volume, ...) bound to the corresponding columns of B at the current index, and with mutable series storage updated after each bar.

This model matches classical technical analysis practice, in which indicators are defined as causal functions of historical prices [Wilder, 1978, Box and Jenkins, 1970, Tsay, 2010].

2.2 Series semantics

Many values in Pine Script™ are *series*: sequences indexed by bar history.

Definition 2.2 (Series). A series of type τ is a partial function $s : \mathbb{N}_0 \rightarrow \tau \cup \{\text{na}\}$ where $s(k)$ denotes the value k bars ago relative to the current bar index ($k = 0$ is the current bar). Writing $s[k]$ in source notation denotes historical indexing.

On each bar, the runtime *pushes* a current value onto each live series and retains a finite history window. PYNE optionally caps history length (series cap) and can store history in a ring buffer to bound memory on long runs (Section 7).

2.3 The na value and propagation

Pine Script™ uses a dedicated missingness token `na` (“not available”), distinct from IEEE floating-point NaN though related in spirit [IEEE Computer Society, 2019, Higham, 2002].

Definition 2.3 (na-propagation). For arithmetic operators $\oplus \in \{+, -, \times, /\}$ and comparisons $\bowtie$, if either operand is `na` then the result is `na` (with documented exceptions for short-circuit Boolean operators). Built-ins that require a full lookback window of valid samples typically return `na` until sufficient history exists.

Short-circuit `and/or` evaluate only as many operands as needed; this interacts with side-effecting expressions and with `na` in Boolean contexts. PYNE implements these rules in both engines and validates them with a parity harness (Section 5).

2.4 Persistence: `var` and `varip`

Two declaration forms control cross-bar state:

- `var` initializes on the first bar (conceptually when $i = 0$) and retains the value across subsequent bars unless reassigned.
- `varip` retains values across realtime tick updates within a forming bar in host environments that distinguish tick vs. bar-close evaluation; offline historical runs treat `varip` analogously to retained state with host-specific tick semantics when ticks are supplied.

These constructs are essential for accumulators, state machines, and strategy position bookkeeping.

2.5 Indicators versus strategies

An *indicator* script primarily produces visual and analytic series: `plot`, `plotshape`, `fill`, `hline`, drawings, and alerts. A *strategy* script additionally emits order intents (`strategy.entry`, `strategy.close`, ...) that a broker simulation fills subject to commission, slippage, pyramiding, and pending-fill rules. PYNE implements both surfaces under one runtime host, with strategy events exported in the response envelope (Section 5).

2.6 Illustrative program

Listing 1 shows a minimal indicator. Under bar-loop evaluation, `ta.rsi` maintains incremental Wilder-style state [Wilder, 1978], and `plot` records one output sample per bar.

Listing 1: Minimal RSI indicator (Pine).

```
1 //@version=6
2 indicator("RSI Demo", overlay=false)
3 len = input.int(14, "Length")
4 r = ta.rsi(close, len)
5 plot(r, "RSI")
6 hline(70)
7 hline(30)
```

3 System Architecture

3.1 Design goals

The architecture is driven by four goals:

1. **Single semantic core.** Parsing, static checks, interpretation, and compilation share one AST schema so that editor diagnostics and Pro evaluation cannot diverge in structure.
2. **Engine substitutability.** Clients select `interpret`, `compile`, or `auto` without changing the external response schema.
3. **Small hand-edited surface.** Only grammar resources (`*.g4`) and the ASDL schema are hand-maintained; lexer/parser and node classes are generated [Parr, 2013, Wang et al., 1997].
4. **Deployability.** The same package installs from PyPI, runs as CLI/LSP binaries, and powers HTTP and edge deployments [HOOX Project, 2026a].

3.2 Layered overview

Figure 1 summarizes the system. Editors communicate with `pynescrypt-lsp` (a `pygls`-based server [Open Law Library, 2020, Microsoft, 2023]) over STDIO. The server parses source into the shared AST, runs the linter, and services completion, hover, formatting (via `unparse`), definition/references, and document symbols. The core package (`pynescrypt`) owns the grammar, AST builder, evaluators, and compiler. The dual-engine runtime executes scripts against OHLCV data and emits plots, drawings, strategy events, and alerts. The Pro API (Flask) exposes `POST /run`, chart preview, and quick backtest endpoints on the same evaluation path. Edge workers and browser runtimes (Pyodide/AXIS) reuse the evaluate contract with thinner hosts.

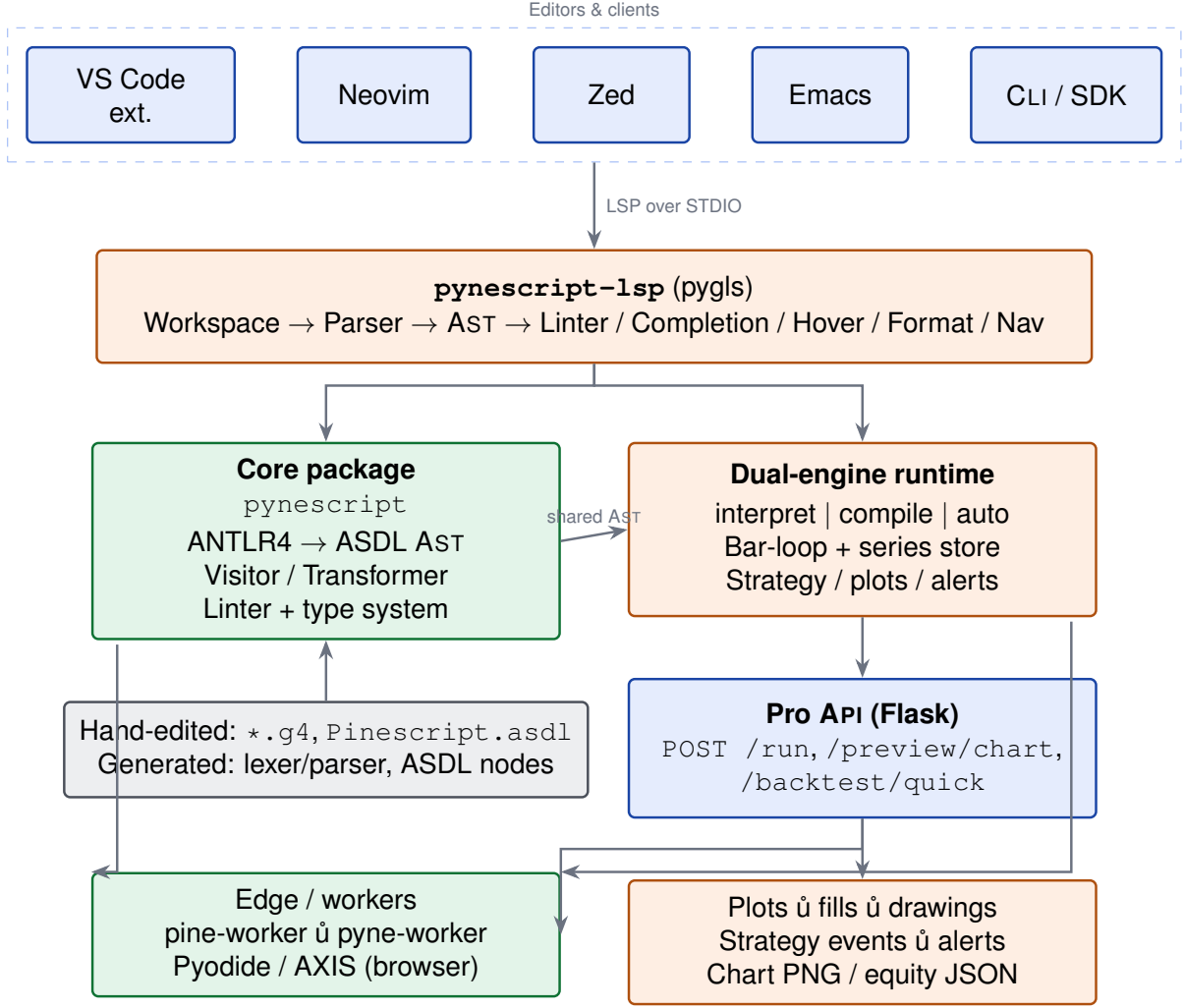


Figure 1: End-to-end PYNE architecture: editors and clients, LSP, core package and dual-engine runtime, Pro API, and edge/browser workers. Hand-edited artifacts are the ANTLR grammars and ASDDL schema; generated code includes the lexer/parser and ASDDL node classes.

3.3 Pipeline notation

The canonical offline pipeline is

$$\text{source} \xrightarrow{\text{lex/parse}} \text{parse tree} \xrightarrow{\text{builder}} \text{AST} \xrightarrow{\text{bar-loop}} \text{artifacts}, \quad (2)$$

where *artifacts* denote plots, fills, drawings, strategy events, and alerts. Source suffixes `.pine` and `.pyne` are accepted equivalently. Optionally, clients post-process artifacts (PNG charts, equity curves, webhook alerts).

3.4 Hand-edited versus generated code

Following classic compiler engineering practice [Aho et al., 2006, Appel, 1998], PYNE minimizes the trusted hand-written front-end surface:

- **Hand-edited:** `antlr4/resource/*.g4`, `asdl/resource/Pinescript.asdl`, `builder/visitor/evaluator/compiler` logic.
- **Generated:** Python lexer and parser under `antlr4/generated/`, ASDDL node classes under `asdl/generated/`.

Committed generated artifacts allow end users to install from PyPI without a Java toolchain, while maintainers regenerate from resources when the language surface evolves (e.g., v6 multiline strings).

3.5 Shared AST between editor and Pro API

A recurring failure mode in dual-product systems is semantic drift between “IDE parsing” and “server execution.” PYNE avoids this by construction: the Pro API and the LSP both call into the same `pynescrypt` package. The Pro path adds authentication, data binding, chart rendering, and operational controls (timeouts, prewarm), not a second language implementation [Ronacher, 2024].

4 Language Front-End

This section gives a high-level account of the front-end. Sister papers in the series expand grammar engineering, series storage, and parity methodology; here we emphasize interfaces required by the architecture.

4.1 Grammar

The lexer and parser are specified in ANTLR4 4 [Parr, 2013, Parr et al., 2014, Parr, 2022]. The grammar approximates the public Pine Script™ v5–v6 surface, including:

- Script kinds: `indicator`, `strategy`, `library`.
- Declarations: functions, type user-defined types (UDT), `enum`, `method`, `imports/exports`.
- Control flow: `if/else`, `for/for...to`, `while`, `switch`, `break/continue`.
- Literals: numbers, strings (including v6 triple-quoted multiline), `colours`, `na`, `booleans`.
- Significant indentation interactions handled via a hand-written lexer base class cooperating with generated rules.

Practical grammar evolution (e.g., triple-quoted strings) requires careful fragment design, selective regeneration of lexer artifacts, and round-trip fixtures [HOOX Project, 2026b].

4.2 ASDL AST

Node types are defined in an ASDL schema [Wang et al., 1997, Python Software Foundation, 2024] and generated into Python classes. The root `Script` node carries a statement list and annotation strings (e.g., `//@version=6`) promoted from comments so that `parse`→`unparse` preserves version and description metadata.

Listing 2: Parse and unparse round-trip (Python API).

```
1 from pynescrypt.ast.helper import parse, unparse
2
3 src = '''//@version=6
4 indicator("Demo")
5 plot(close)
6 '''
7 tree = parse(src)
8 assert "plot" in unparse(tree)
```

Visitors and transformers [Gamma et al., 1994a,b] provide the extension points used by the linter, compiler emitter, and evaluators.

4.3 Round-trip fidelity

Lossless round-trip is a project invariant for supported syntax:

$$\text{unparse}(\text{parse}(s)) \equiv_{\text{syn}} s', \quad (3)$$

where $\equiv_{\text{syn}}$ denotes syntactic equivalence up to documented normalization (whitespace policies, annotation attachment). Round-trip underpins formatting-as-unparsing in the LSP and guards against silent AST incompleteness when new syntax is added.

4.4 Linter and type system

The linter implements a set of structural and style rules (undeclared names, suspicious assignments, version issues, etc.) on the AST. A type system layer tracks primitive types, series wrappers, and UDT field shapes for completion and diagnostics [Pierce, 2002, Cardelli, 1996]. Full formalization of the type rules is out of scope for this architecture paper; the operational interface is: parse once, lint and complete from the same tree, evaluate from the same tree.

4.5 Builtin metadata

Builtin completion and hover draw from structured metadata covering namespaces such as `ta`, `math`, `str`, `array`, `matrix`, `map`, `strategy`, `request`, and drawing objects. The LSP exposes on the order of 800+ completion items, exceeding early inventory counts as v6 surface area grew [HOOX Project, 2026b].

5 Dual-Engine Runtime

5.1 Execution modes

The runtime accepts an explicit mode parameter

$$\text{mode} \in \{\text{auto}, \text{compile}, \text{interpret}\}. \quad (4)$$

- **Interpret.** A classical AST walk [Nystrom, 2021]: each bar re-enters the script body; builtins dispatch through Python mixins; series are updated in the host.
- **Compile.** A compiler visitor lowers supported numeric subgraphs to NUMBA [Lam et al., 2015, Latner and Adve, 2004] nopython kernels (and related helpers), with object-mode regions for features that are not yet nopython-safe (strategy bookkeeping, some drawings, dynamic Python objects).
- **Auto.** Prefer compile when feasible; on hard failure or unsupported lowering, fall back to interpret while preserving the response envelope.

Figure 2 shows the decision flow for warm cache, nopython success, object-mode fallback, and interpret recovery.

5.2 Bar-loop data flow

Figure 3 details data movement inside a run. OHLCV columns are bound as builtins; the host iterates i ; script evaluation updates the series store, invokes incremental or full `ta` kernels, and drives the strategy engine. Outputs are packed into plots, drawings, and events.

Algorithm 1 captures the interpret host in pseudocode. The compile path replaces the body of the inner evaluation with kernel calls but preserves the same outer envelope construction.

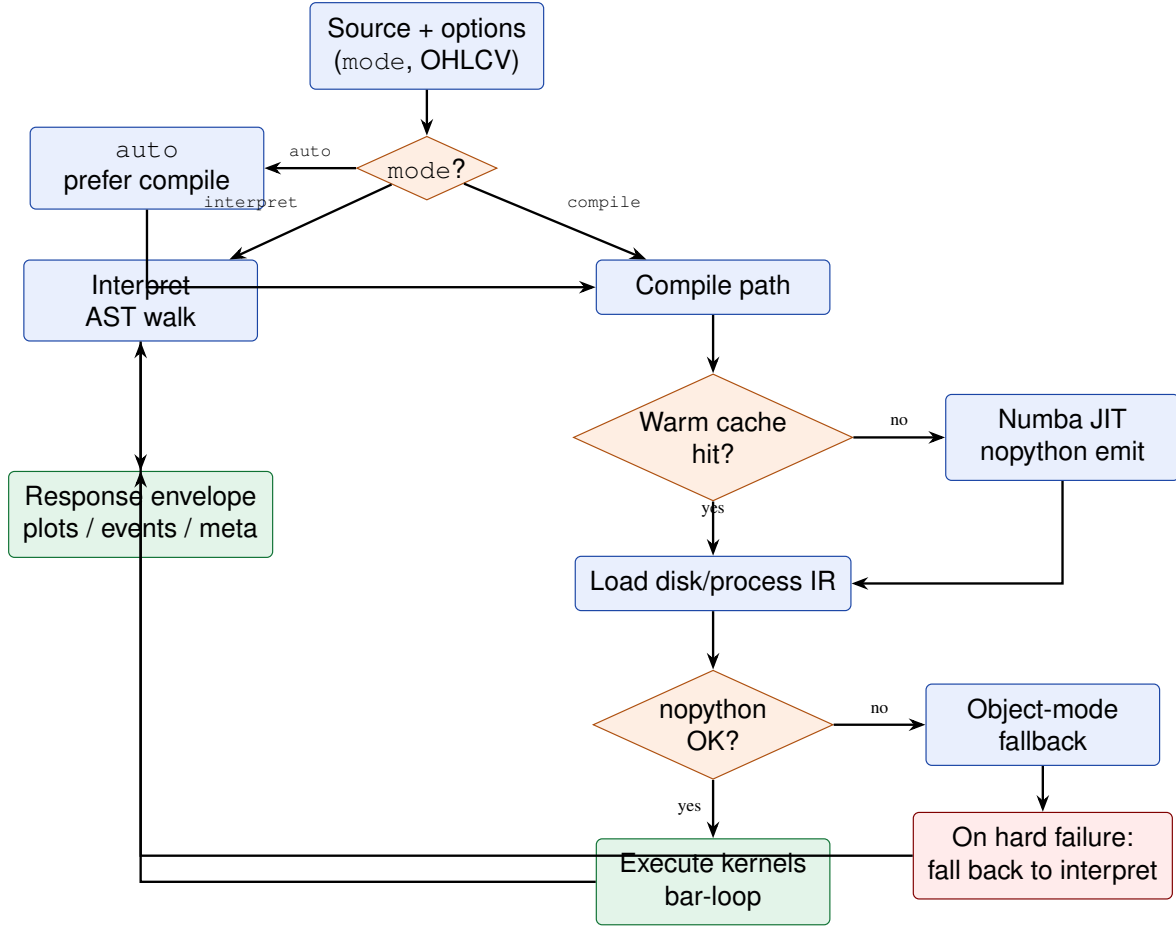


Figure 2: Dual-engine decision flowchart. Mode `auto` prefers the compile path; warm disk/process IR avoids repeated JIT cost; nopython kernels execute when available, otherwise object-mode; hard failures fall back to interpretation.

5.3 Warm compile and IR cache

Cold NUMBA compilation can dominate latency for short interactive runs. PYNE therefore implements:

- **Disk IR cache** keyed by compilation inputs (source digest, option flags, library versions as applicable).
- **Process prewarm** via CLI/API hooks that force compilation of common kernels at worker start.
- **Corrupt cache recovery** that discards invalid entries and recomputes rather than failing the request path.

Internal Round 7 measurements report warm compile on the order of ~ 0.01 ms for a representative `ta_combo` artifact, with run latency ~ 1.06 ms at 5,000 bars (Section 7).

5.4 Incremental technical analysis kernels

Many indicators admit $O(1)$ or amortized updates per bar given prior state [Acar, 2008, Wilder, 1978, Bollinger, 2001, Kaufman, 2013]. Both engines exploit incremental forms for families such as `sma/ema/rma`, `rsi`, `macd`, `bb`, `kama`, `stoch`, and others. Kernel microbenchmarks in Round 7 show large per-kernel speedups (e.g., `kama` $\sim 121\times$, `bb` $\sim 217\times$ versus naive recompute styles in the measured configuration). Numerical agreement is checked against full recompute within floating-point tolerance [Higham, 2002, Press et al., 2007].

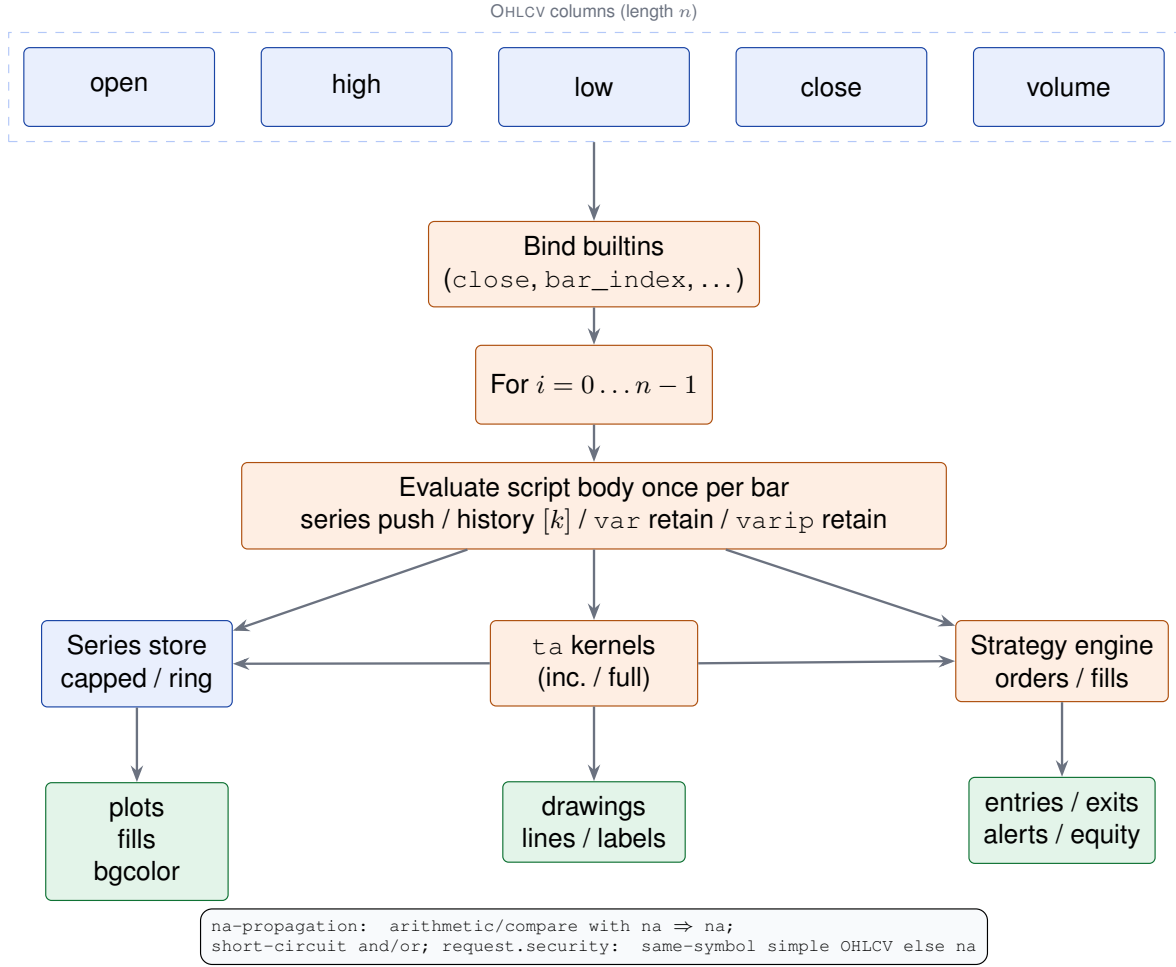


Figure 3: Bar-loop data flow from OHLCV inputs through series storage and `ta`/strategy engines to plots, drawings, and events. The bottom annotation summarizes `na` and `request.security` policies.

5.5 Strategy engine

Strategy support includes entries and exits, commission and slippage models, one-cancels-other style constraints where implemented, and pending-fill behaviour under pyramiding limits. A representative semantic choice: when pyramiding is disabled or capped, pending fills may average according to documented VWAP-style rules rather than silently dropping intents. Risk helpers and order-fill tests form part of the regression suite.

5.6 Security policy for `request.security`

Multi-symbol and multi-timeframe requests are a major complexity axis of the language [TradingView, Inc., 2024]. PYNE adopts a conservative default:

Definition 5.1 (Security resolution policy). For `request.security` (and related forms), *same-symbol simple* OHLCV expressions may resolve against the bound data feed. Foreign symbols or complex expressions that would require inventing unavailable market state resolve to `na` rather than fabricating closes.

This policy prioritizes honesty for offline and CI contexts over incomplete emulation of a full multi-asset platform data graph.

Algorithm 1 Interpret bar-loop host (simplified)

Require: script AST P , bars $B[0..n-1]$, options Ω

Ensure: response envelope $\mathcal{E}$

```
1:  $store \leftarrow \text{InitSeriesStore}(\Omega)$ 
2:  $strat \leftarrow \text{InitStrategy}(\Omega)$ 
3:  $plots \leftarrow \emptyset$ ;  $events \leftarrow \emptyset$ 
4: for  $i \leftarrow 0$  to  $n - 1$  do
5:    $\text{BindBuiltin}(store, B[i], i)$ 
6:    $\text{BeginBar}(strat, B[i])$ 
7:    $\text{EvalBody}(P, store, strat, plots)$ 
8:    $\text{ProcessPendingFills}(strat, B[i])$ 
9:    $\text{EndBar}(strat, events)$ 
10:   $\text{CapSeries}(store)$  ▷ optional memory bound
11: end for
12:  $\mathcal{E} \leftarrow \text{PackEnvelope}(plots, events, meta)$ 
13: return  $\mathcal{E}$ 
```

5.7 Response envelope and parity harness

Regardless of engine, a successful run returns a structured envelope:

- **Plots / fills / bgcolor** series aligned to bar indices.
- **Drawings** with garbage-collection limits (`max_lines_count`, `max_labels_count`, ...).
- **Strategy events** (entries, exits, equity samples).
- **Alerts** from `alert/alertcondition` with frequency semantics (`once_per_bar`, `once_per_bar_close`, `all`).
- **Metadata** (engine used, timings, warnings, mode fallback flags).

The *parity harness* compares interpret vs. compile outputs on shared fixtures, asserting series alignment within absolute/relative tolerances (ε_{abs} , ε_{rel}) and matching event shapes. Parity here means *internal engine consistency*, not certification against a proprietary platform.

6 Product Surfaces

6.1 CLI

The `pynescrypt` CLI supports check, format, lint, compile, run, data fetch, and prewarm subcommands. It is the primary interface for local scripts and automation:

Listing 3: Programmatic evaluation sketch.

```
1 # Conceptual host usage (simplified)
2 # runtime.run(source, ohlcv, mode="auto") -> envelope
```

Container images publish multi-arch CLI builds for reproducible desks and CI.

6.2 Language server

`pynescrypt-lsp` implements LSP 3.x features [Microsoft, 2023]: diagnostics (parse + lint), completion, hover signatures, go-to-definition, references, document symbols, semantic tokens, and formatting via unparse. Distribution options include pip extras (`hoox-pyne[lsp]`) and Nuitka onefile binaries [Hayen, 2024] bundled by the VS Code extension.

Table 1: Product surface comparison for PYNE and related hosts.

Surface	Package / binary	Role
Core library	hoox-pyne / pynescrypt	Parse, AST, evaluate, compile
CLI	pynescrypt	Desk automation, CI, pre-warm
LSP	pynescrypt-lsp	Editor intelligence (~800+ builtins)
VS Code ext.	VSIX	Bundled LSP, language config
Pro API	Flask / containers	HTTP run, chart, backtest
pine-worker	Bun / TS	Edge evaluate (partial builtins)
pyne-worker	Python worker	Thin edge host over core semantics
Pyodide/AXIS	browser	In-page demos and docs

6.3 Pro API

The Flask Pro backend [Ronacher, 2024] exposes authenticated HTTP endpoints, notably:

- `POST /run` — evaluate script with data and options;
- `POST /preview/chart` — render chart artifacts;
- `POST /backtest/quick` — strategy summary paths.

Middleware enforces API keys and tier limits. Workers may prewarm compile caches to meet interactive latency SLOs.

6.4 Editors and edge

- **VS Code extension:** first-class `.pyne/` `.pine` associations and bundled LSP binary.
- **Neovim / Zed / Emacs:** client configs under `clients/`.
- **pine-worker / pyne-worker:** TypeScript and Python edge hosts sharing evaluate contracts [Cloudflare, Inc., 2024].
- **Pyodide / AXIS:** browser-side evaluation for documentation and lightweight apps [Pyodide Contributors, 2024].

6.5 Surface comparison

Table 1 compares major product surfaces along packaging, primary users, and evaluation capability.

7 Evaluation and Deployment Experience

Unless noted, performance figures are *internal measurements* collected on development hardware during 2026 performance campaigns (Rounds 1–7 and related agent runs). They are representative engineering benchmarks, not standardized public leaderboards.

Table 2: Representative median latencies (internal measurements, 2026). Interpret figures are bar-loop style at $\sim 2k$ bars for combo; compile run figures are warm kernels at $\sim 5k$ bars where noted.

Workload	Interpret (ms)	Compile run (ms)	Approx. ratio	Notes
minimal @2k	15.2	—	—	Round 7 interpret
ta_sma @2k	23.6	0.04	$\sim 590\times$	compile @5k SMA
ta_rsi	~ 22	0.08	large	warm compile
ta_macd	~ 35	0.12	large	warm compile
MULTI	~ 48	0.21	$\sim 230\times$	multi-indicator
ta_combo	152.9	1.06	$\sim 144\times$	interp@2k / comp@5k
strategy_ish @2k	68.9	(object mix)	—	fills + events
warm compile only	—	~ 0.01	—	cache hit, combo

7.1 Microbenchmark methodology

Scripts include minimal empty indicators, single-indicator programs (`ta.sma`, `ta.rsi`, `ta.macd`), multi-indicator combinations, and strategy-shaped fixtures. Bar counts of 2,000 and 5,000 are common. We report median wall times over repeated runs after optional warm-up. Compile paths distinguish cold JIT vs. warm cache hits.

7.2 Front-end performance

Early agent campaigns measured:

- **Parse:** approximately $5.4\times$ improvement on a complex ~ 16 KB script via SLL-first parsing with LL fallback, cheaper annotation handling, and faster location attachment.
- **Unparse:** approximately $1.95\times$ on a multi-script corpus via thread-local unparser reuse and type-keyed dispatch.

An AST LRU cache keyed by source digest further reduces multi-parse latency in LSP and repeated API submissions (multi-parse speedups up to three orders of magnitude on cache hits in Round 7 notes).

7.3 Interpret versus compile

For numeric `ta` scripts, compile+execute can be ~ 49 – $110\times$ faster than interpret in measured MACD/-multi configurations after incremental kernels and plot packing improvements. Interpret-side incremental `ta` yields roughly 1.9 – $6.5\times$ gains on relevant Runtime paths versus non-incremental baselines.

Table 2 and Figure 4 summarize representative median latencies. Interpret rows reflect bar-loop host costs at a few thousand bars; compile rows reflect warm nopython execution style latencies (run phase). Exact values vary by CPU, Numba version, and plot export settings; the table is intended to communicate magnitude and ranking.

7.4 Memory: series cap and ring buffer

Long historical runs accumulate series history. Enabling a series cap (default on in recent builds) reduced long-run list memory by approximately $78\times$ in Round 7 experiments relative to uncapped growth. An optional chronological ring buffer further bounds residency; default rollout is gated on corpus greenness.

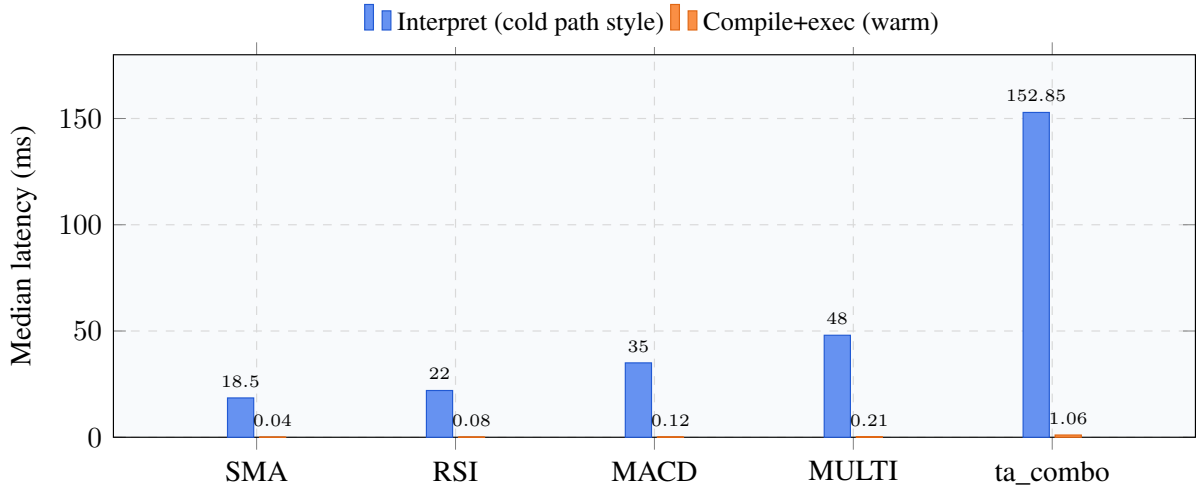


Figure 4: Interpret vs. warm compile+exec median latency for representative numeric workloads (internal measurements, 2026). Axes are not normalized to identical bar counts across all series; the chart emphasizes order-of-magnitude gaps for kernel-heavy scripts.

Table 3: Compatibility matrix summary (qualitative). Full feature tables live in project documentation [[HOOX Project, 2026b](#)].

Area	pynescrypt	pine-worker	pyne-worker
Parser (v5/v6)	Full (ANTLR)	Full (generated TS)	Delegated
AST coverage	Full ASDL	Partial (Zod)	Delegated
na / series	Full	Full (subset builtins)	Delegated
Strategy fills	Full	Full (evolving)	Full (via core)
Compile engine	Numba dual path	N/A	N/A
<code>request.security</code>	Same-symbol simple; else na	Policy-dependent	Delegated
Parity vs core	Reference	Partial fixtures	Fixture-oriented

7.5 Compatibility posture

Table 3 summarizes compatibility themes across the Python core and edge hosts. The core `pynescrypt` implementation is the reference for parity tests; workers may delegate or reimplement subsets.

Corpus parse rates on sanitized public-style script sets have been measured in the high nineties of percent for supported syntax after lexer/sanitize hardening; residual failures concentrate on exotic scrapes and unsupported platform-only constructs.

7.6 Test suite scale

The repository maintains a substantial `pytest` suite spanning parser round-trips, linter cases, `ta` goldens, strategy fills, compiler kernels, interpret $\leftrightarrow$ compile parity, LSP features, and backend routes. Recent verification notes cite on the order of 600+ passing core tests after Round 7 merges, with additional edge-worker suites elsewhere. Tests prefer deterministic synthetic OHLCV over live network data.

7.7 Deployment lessons

Operational experience suggests:

1. Prefer **warm compile workers** on the Pro path; interpret remains valuable for debugging and for scripts outside the `nopython` surface.

2. Treat **plot packing** as a first-class cost center: profiles show export/registry work dominating some multi-plot interpret runs even when kernels are cheap.
3. Keep **parse caches** coherent with call-site metadata scrubbing so that cache hits cannot poison empty plot registries.
4. Document **security na policy** prominently: users expecting full multi-symbol platform behaviour must supply data explicitly.

8 Related Work

8.1 DSL toolchains and compiler construction

Domain-specific languages are a long-standing software engineering topic [Hudak, 1996, Fowler, 2010, Erwig and Walkingshaw, 2011]. PYNE follows the classical pipeline of lexing, parsing, AST construction, and multi-backend execution [Aho et al., 2006, Cooper and Torczon, 2011, Muchnick, 1997]. Using ANTLR4 [Parr, 2013] and ASDL [Wang et al., 1997] mirrors choices in other language implementations that separate declarative syntax from semantic passes. Visitor-based lowering resembles patterns in production compilers and teaching interpreters [Gamma et al., 1994a, Nystrom, 2021].

8.2 Financial and statistical time-series languages

Technical analysis packages in general-purpose languages (e.g., NumPy-centric stacks [Harris et al., 2020, McKinney, 2017]) provide library APIs rather than a charting DSL. Pine Script™ [TradingView, Inc., 2024] is distinctive in coupling series indexing, plot effects, and strategy simulation in one language. PYNE reimplements that *language-shaped* interface offline without claiming platform equivalence. Classical indicator definitions remain the numerical foundation [Wilder, 1978, Bollinger, 2001, Murphy, 1999, Kaufman, 2013].

8.3 Dual interpreter + JIT systems

Many production language VMs combine interpretation with selective compilation [Chambers and Ungar, 1990, Peyton Jones and Marlow, 2002, Lattner and Adve, 2004]. PYNE’s dual engine is narrower: a domain bar-loop host with NUMBA kernels [Lam et al., 2015] for numeric subgraphs and an AST interpreter for full language coverage. The design goal is pragmatic performance for indicators, not a general tracing JIT for arbitrary Python.

8.4 Language servers and editor ecosystems

The Language Server Protocol [Microsoft, 2023, Microsoft DevBlogs, 2016] decouples editors from language implementations. PYNE’s pygls server [Open Law Library, 2020] follows this model, enabling VS Code, Neovim, Zed, and Emacs from one codebase—an explicit alternative to browser-only authoring.

8.5 Edge and browser runtimes

Deploying evaluation to Cloudflare Workers [Cloudflare, Inc., 2024] and Pyodide [Pyodide Contributors, 2024] continues a trend of moving compute closer to clients. PYNE’s contribution is a shared evaluate contract across these hosts rather than a single-runtime monoculture.

9 Discussion and Limitations

9.1 No full platform parity

PYNE does not reproduce the entire TradingView® product: proprietary charting UI, closed data services, social features, and some platform-only builtins are out of scope. Numerical results on community scripts can differ when hosts implement different fill models, session calendars, or security resolution. Users should treat PYNE as an open evaluation and tooling substrate, not as a drop-in legal or functional substitute for the commercial platform.

9.2 Foreign request . security policy

Resolving foreign or complex `request.security` calls to `na` (Definition 5.1) avoids silent wrong answers but surprises users who expect multi-asset charts without providing data. Future work includes richer multi-feed binders under explicit user control.

9.3 Compile surface incompleteness

Not every language feature lowers to `nopython`. Object-mode and interpret fallbacks preserve correctness at the cost of latency cliffs when a script leaves the fast path. Continuing kernel coverage and strategy lowering remains active engineering.

9.4 AGPL implications

PYNE is licensed under AGPL-3.0-or-later [Free Software Foundation, 2007]. Network-facing services that modify and provide the software to users inherit corresponding source-offer obligations. Organizations embedding PYNE in SaaS should review compliance with counsel; the license choice intentionally favours open reciprocity for a community toolchain [HOOX Project, 2026a].

9.5 Trademark and independence

Pine Script™ and TradingView® are trademarks of TradingView, Inc. PYNE is independent and unofficial. Documentation references public language materials for interoperability only and does not redistribute proprietary platform software.

9.6 Threats to validity for benchmarks

Reported speedups depend on hardware, library versions, plot export settings, and script mix. Median times from internal campaigns should be read as engineering evidence of relative engine behaviour, not as guaranteed SLAs.

10 Conclusion

We presented PYNE, an open dual-engine toolchain for offline Pine Script™ evaluation. By structuring the system as a classical language pipeline—ANTLR4 front-end, ASDL AST, bar-loop host with interpret and NUMBA compile modes—and by sharing that core across CLI, LSP, Pro API, and edge clients, the project enables reproducible research and editor-native workflows outside browser-locked environments. Internal measurements show substantial gains from incremental kernels and warm compilation for numeric scripts, while a conservative security policy and parity harness emphasize honest offline semantics. Limitations remain around full platform parity and compile coverage; we view these as a roadmap rather than a claim of completion. PYNE is available as `hoox-pyne` on PyPI under AGPL-3.0-or-later as part of the HOOX open trading stack [HOOX Project, 2026b,a].

Acknowledgments

We thank contributors to the HOOX/PYNE ecosystem and users who filed compatibility reports. Performance campaigns (Rounds 1–7) benefited from automated multi-agent benchmarking harnesses and regression corpora maintained in-tree.

References

- Umut A. Acar. Incremental computation. In *Encyclopedia of Algorithms*. Springer, 2008.
- Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson, Boston, 2nd edition, 2006.
- Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.
- John Bollinger. Bollinger on bollinger bands. *McGraw-Hill*, 2001.
- George E. P. Box and Gwilym M. Jenkins. Time series analysis: Forecasting and control. *Holden-Day*, 1970.
- Luca Cardelli. Type systems. *ACM Computing Surveys*, 28(1):263–264, 1996.
- Craig Chambers and David Ungar. Iterative type analysis and extended message splitting. In *PLDI*, 1990.
- Cloudflare, Inc. Cloudflare Workers documentation. <https://developers.cloudflare.com/workers/>, 2024.
- Keith Cooper and Linda Torczon. *Engineering a Compiler*. Morgan Kaufmann, 2nd edition, 2011.
- Martin Erwig and Eric Walkingshaw. Semantics first! Rethinking the language design process. In *Proceedings of the 10th International Conference on Generative Programming*, 2011.
- Martin Fowler. Domain-specific languages. Addison-Wesley, 2010.
- Free Software Foundation. GNU Affero General Public License version 3. <https://www.gnu.org/licenses/agpl-3.0.html>, 2007.
- Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: Elements of reusable object-oriented software. *Addison-Wesley*, 1994a.
- Erich Gamma et al. *Design Patterns*. Addison-Wesley, 1994b. Visitor pattern.
- Charles R. Harris et al. Array programming with NumPy. *Nature*, 585:357–362, 2020. doi: 10.1038/s41586-020-2649-2.
- Kay Hayen. Nuitka: Python compiler. <https://nuitka.net/>, 2024.
- Nicholas J. Higham. Accuracy and stability of numerical algorithms. *SIAM*, 2002.
- HOOX Project. HOOX: Open trading stack. <https://hoox.sh>, 2026a.
- HOOX Project. PYNE: Independent open toolchain for the Pine Script language. <https://github.com/hoox-sh/pyne>, 2026b. Version 0.3.0; PyPI package `hoox-pyne`.
- Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4es):196, 1996.

- IEEE Computer Society. IEEE standard for floating-point arithmetic. *IEEE Std 754-2019*, 2019. doi: 10.1109/IEEESTD.2019.8766229.
- Perry J. Kaufman. Trading systems and methods. Wiley, 2013.
- Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A LLVM-based python JIT compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, pages 1–6. ACM, 2015. doi: 10.1145/2833157.2833162.
- Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 75–86. IEEE, 2004.
- Wes McKinney. Python for data analysis. O’Reilly, 2017.
- Microsoft. Language server protocol specification. <https://microsoft.github.io/language-server-protocol/>, 2023. Version 3.17.
- Microsoft DevBlogs. Language server protocol and the future of language tooling. 2016.
- Steven S. Muchnick. Advanced compiler design and implementation. *Morgan Kaufmann*, 1997.
- John J. Murphy. *Technical Analysis of the Financial Markets*. New York Institute of Finance, 1999.
- Robert Nystrom. *Crafting Interpreters*. Genever Benning, 2021.
- Open Law Library. Building language servers with pygls. 2020.
- Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, Raleigh, NC, 2013.
- Terence Parr. ANTLR 4 documentation. <https://www.antlr.org/>, 2022.
- Terence Parr, Sam Harwell, and Kathleen Fisher. Adaptive LL(*) parsing: The power of dynamic analysis. In *OOPSLA*. ACM, 2014.
- Simon Peyton Jones and Simon Marlow. Secrets of the Glasgow Haskell Compiler inliner. *Journal of Functional Programming*, 12(4):393–434, 2002.
- Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
- William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, 3rd edition, 2007.
- Pyodide Contributors. Pyodide: Python scientific stack for the browser. <https://pyodide.org/>, 2024.
- Python Software Foundation. Python ASDL and the ast module. <https://docs.python.org/3/library/ast.html>, 2024.
- Armin Ronacher. Flask web framework. <https://flask.palletsprojects.com/>, 2024.
- TradingView, Inc. Pine Script language reference manual. <https://www.tradingview.com/pine-script-docs/>, 2024. Accessed 2026.
- Ruey S. Tsay. *Analysis of Financial Time Series*. Wiley, 3rd edition, 2010.
- Daniel C. Wang, Andrew W. Appel, Jeff L. Korn, and Christopher S. Serra. The Zephyr abstract syntax description language. In *Proceedings of the Conference on Domain-Specific Languages*, pages 213–228. USENIX, 1997.
- J. Welles Wilder. New concepts in technical trading systems. *Trend Research*, 1978.