

PYNE

Academic Paper Series

Complete Volume — All Manuscripts

Independent Open Toolchain for the Pine Script™ Language

HOOX Research / PYNE Contributors

Independent Open Source Research

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

2026

Abstract

This volume concatenates the full set of arXiv-oriented manuscripts describing PYNE (`pynescrypt` / `hoox-pyne`): architecture, Numba compilation, series/runtime performance, grammar and ASDL front-end, numerical parity and strategy semantics, and Language Server Protocol tooling. Each part is the complete original paper (all pages), included without abridgement.

Trademark notice. Pine 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.

Volume contents

1. **Architecture** — *PYNE: An Open Dual-Engine Toolchain for Offline Pine Script Evaluation*
paper01
2. **Compiler** — *Compiling Bar-Oriented Trading DSLs to Numba* paper02
3. **Series & runtime** — *Efficient Series History and Incremental Technical Analysis* paper03
4. **Grammar** — *Formalizing a Charting DSL: ANTLR4 and ASDL for Pine Script* paper04
5. **Parity & strategy** — *Numerical Parity and Strategy Semantics for Independent Runtimes*
paper05
6. **LSP tooling** — *Language Tooling for Financial DSLs: LSP and Multi-Editor Delivery* paper06

How to rebuild. From the `.papers/` directory run `make all` (individual papers) then `make series` (this volume). Sources live under `paper01-* ... paper06-*`; shared macros and bibliography under `common/`.

Contents

1	PYNE: An Open Dual-Engine Toolchain for Offline Pine Script Evaluation	3
2	Compiling Bar-Oriented Trading DSLs to Numba	22
3	Efficient Series History and Incremental Technical Analysis	39
4	Formalizing a Charting DSL: ANTLR4 and ASDL for Pine Script	56
5	Numerical Parity and Strategy Semantics for Independent Runtimes	71
6	Language Tooling for Financial DSLs: LSP and Multi-Editor Delivery	88

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.

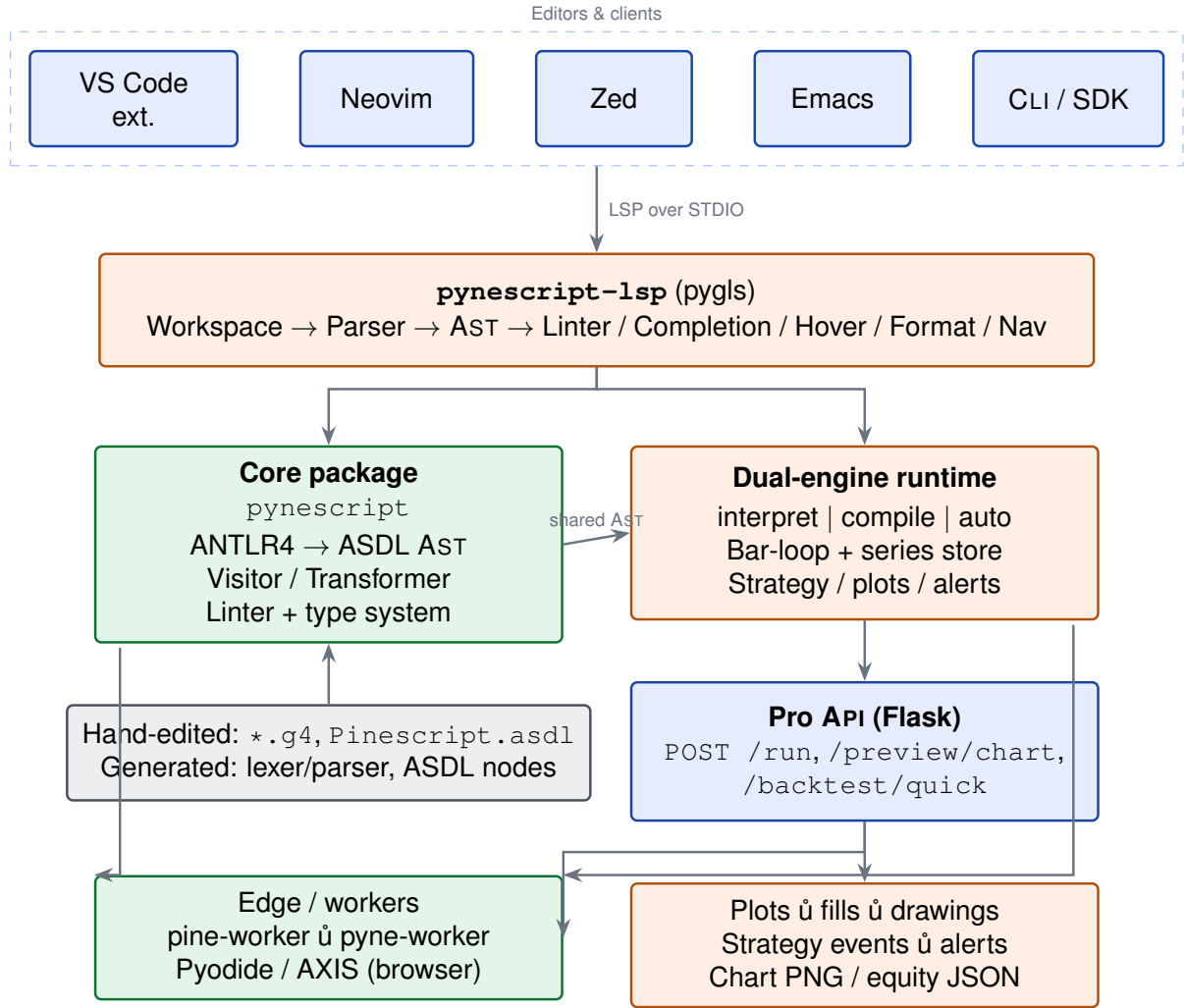


Figure 1: End-to-end PYNE architecture: editors and clients, LSP, core package and dual-engine runtime, Pro API, and edge/browser workers. Hand-edited artifacts are the ANTLR grammars and ASDL schema; generated code includes the lexer/parser and ASDL node classes.

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/`, ASDL 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.

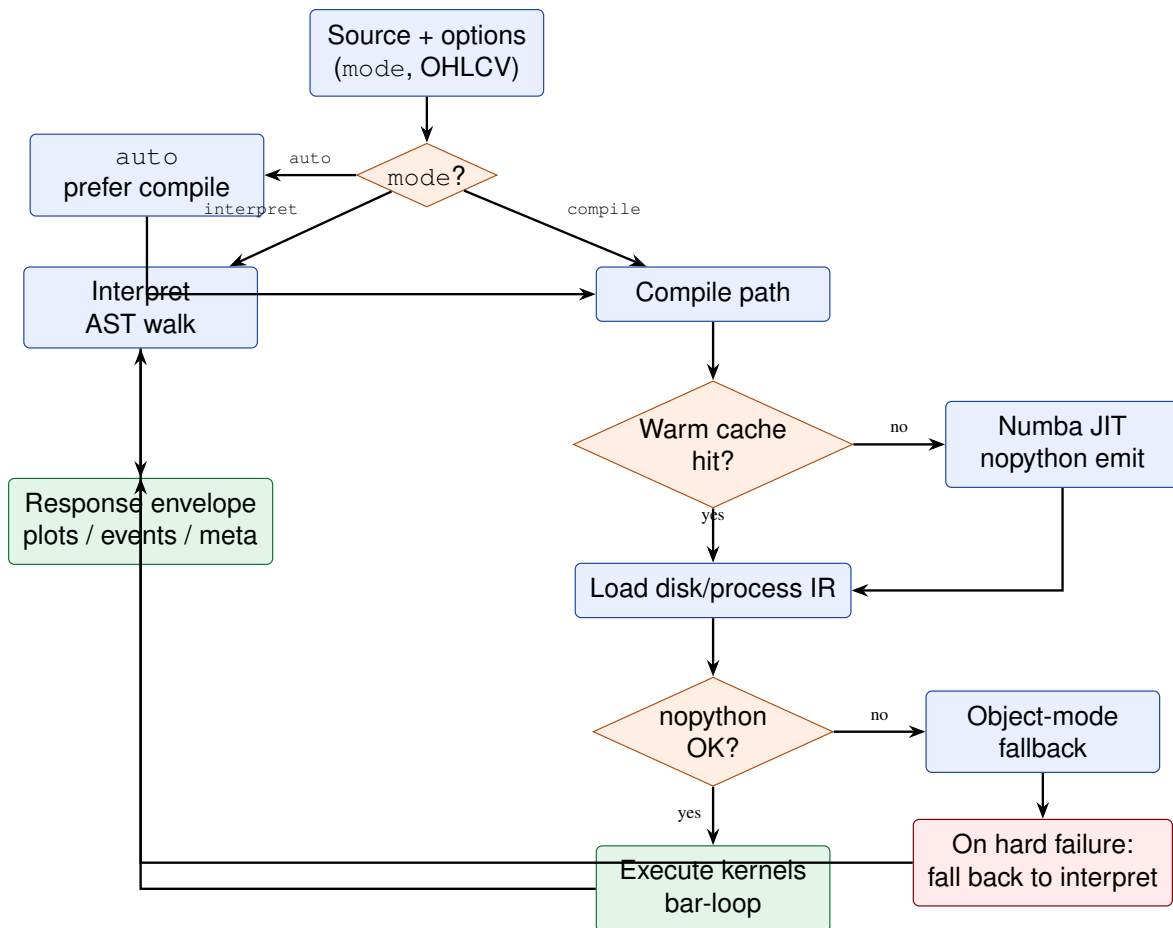


Figure 2: Dual-engine decision flowchart. Mode `auto` prefers the compile path; warm disk/process IR avoids repeated JIT cost; nopython kernels execute when available, otherwise object-mode; hard failures fall back to interpretation.

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].

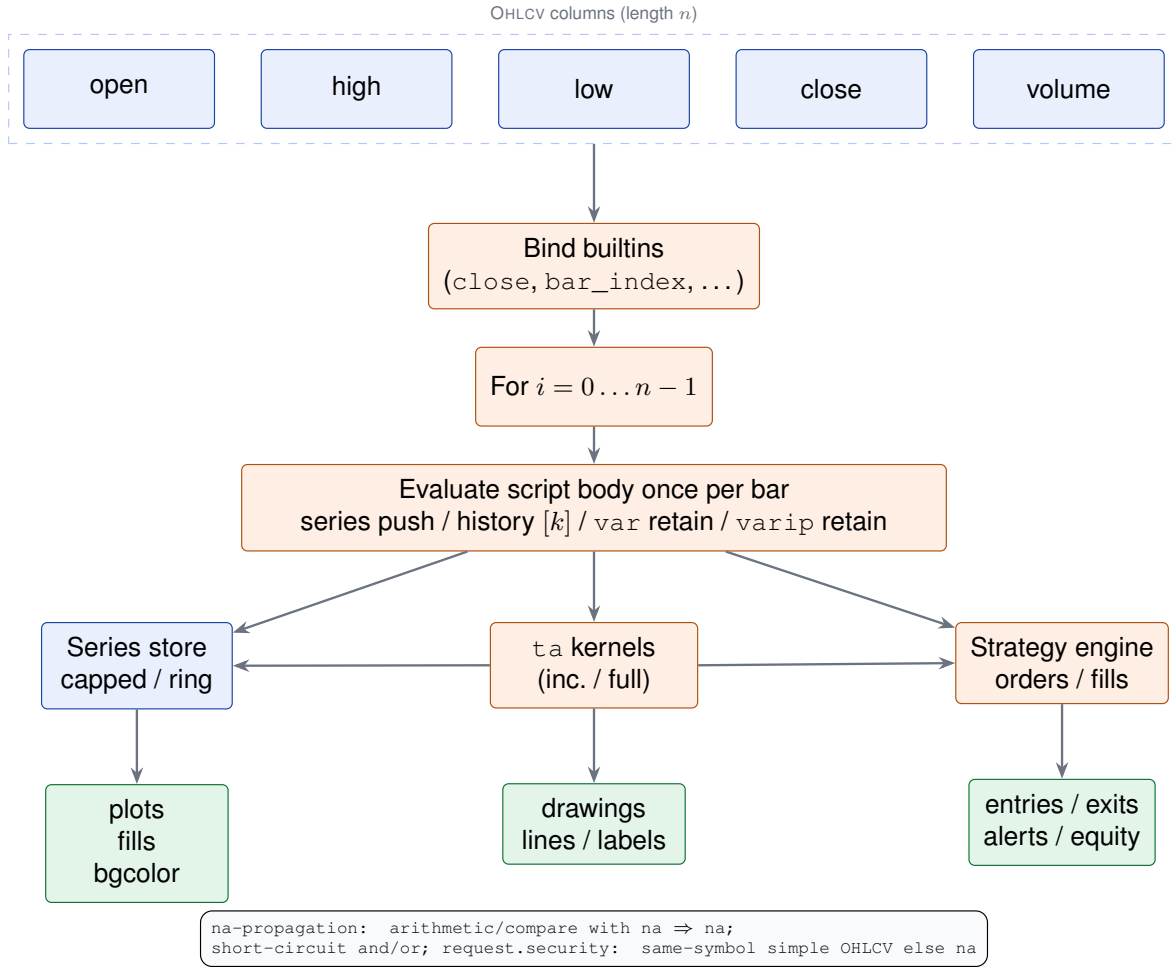


Figure 3: Bar-loop data flow from OHLCV inputs through series storage and `ta`/strategy engines to plots, drawings, and events. The bottom annotation summarizes `na` and `request.security` policies.

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.

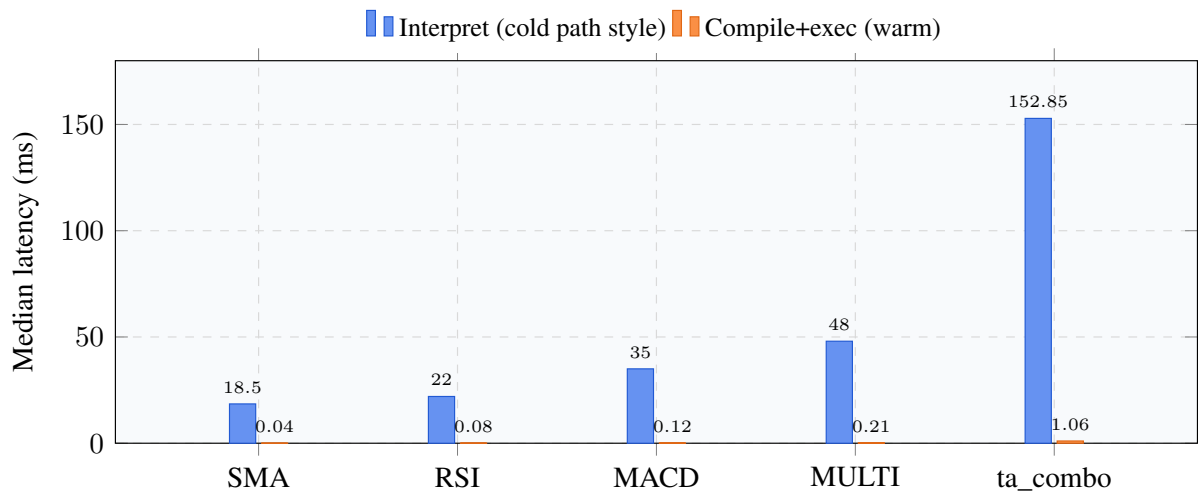


Figure 4: Interpret vs. warm compile+exec median latency for representative numeric workloads (internal measurements, 2026). Axes are not normalized to identical bar counts across all series; the chart emphasizes order-of-magnitude gaps for kernel-heavy scripts.

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.

Compiling Bar-Oriented Trading DSLs to NUMBA: Source-to-Source Lowering, Object-Mode Fallback, and Warm IR Caches

jango_blockchained

PYNE Contributors

HOOX Research / Independent Open Source Research

contact@hoox.sh

Abstract

Bar-oriented trading domain-specific languages (DSLs) evaluate indicators and strategies by traversing a time-ordered sequence of candlesticks. Pure interpretive execution of such programs incurs per-node dispatch cost on every bar, which dominates wall-clock time for multi-thousand bar histories and multi-indicator scripts. We present the PYNE NUMBA compiler path: a source-to-source lowering from a Pine-compatible AST into explicit-index Python that operates on contiguous NumPy arrays and, when the script is purely numeric, is JIT-compiled with NUMBA’s nopython mode to LLVM machine code.

The architecture comprises four layers: (i) a `CompilerVisitor` that transpiles the AST into a bar loop over `__bar_idx`; (ii) a library of `@njit` technical-analysis kernels with incremental $O(1)$ or amortized state updates; (iii) flat array storage replacing interpreter dequeues; and (iv) an engine façade integrating transpile, compile, run, in-process/disk IR caches, and host auto-mode with interpret fallback. Scripts that use user-defined types, maps, or drawings are compiled into an object-mode pure-Python bar loop that preserves those constructs while still avoiding full AST interpretation.

Internal measurements (2026) show cold-to-warm compile transitions of ~ 0.01 ms on cache hits, warm execute medians of roughly 0.04–0.21 ms for representative indicators at 5,000 bars, and ~ 49 – $110\times$ end-to-end speedups versus list-based interpretation for multi-indicator workloads after incremental-kernel fixes. Dual-host nan-aware parity harnesses keep kernel error within floating-point noise ($\varepsilon_{\max} \sim 10^{-11}$). We do not claim TradingView certification or invention of foreign multi-asset series; foreign `request.security` is intentionally lowered to `na` under a documented policy.

Keywords. source-to-source compilation, NUMBA, JIT, domain-specific languages, financial indicators, dual-mode compilation, warm IR caches, technical analysis.

Disclaimer. Pine 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.

1 Introduction

Technical analysis systems process market data as ordered sequences of bars (candlesticks) carrying open, high, low, close, volume, and time (OHLCV+time) [Murphy, 1999, Tsay, 2010]. Domain languages in this niche—of which Pine Script™ is a widely used commercial instance [TradingView, Inc., 2024]—expose a declarative programming model in which series expressions, history offsets, and sliding-window indicators are evaluated bar-by-bar. An AST-walking interpreter is a natural first implementation: it reuses a single semantic definition for interactive tooling, strategy simulation, and host APIs. Unfortunately, visitor dispatch, dynamic attribute

resolution, and list- or deque-backed series history make the constant factors of interpretation large relative to the arithmetic of the indicators themselves.

PYNE is an independent open toolchain for Pine-compatible scripts [HOOX Project, 2026b,a]. Historically it executed scripts via a pure-Python evaluator over an ASDL-AST produced by an ANTLR4 parser [Parr, 2013, Wang et al., 1997]. To approach realtime-stream and large-backtest throughput without abandoning the Python packaging surface, the project introduces a *source-to-source* compilation path that lowers the same AST into explicit-index NumPy code and optionally JIT-compile the result with NUMBA [Lam et al., 2015] on top of LLVM [Lattner and Adve, 2004].

Problem. Let n be the number of bars and C the cost of one interpretive expression evaluation. A script with E expressions per bar has rough cost $\Theta(n \cdot E \cdot C_{\text{interp}})$. When C_{interp} is dominated by Python bytecode and visitor overhead rather than arithmetic, even $O(n)$ algorithmic complexity appears slow in wall time. Naive window recomputation of moving averages further inflates work to $\Theta(n \cdot p)$ or worse.

Approach. We lower each script to a single closed bar loop over contiguous `float64` arrays [Harris et al., 2020], call precompiled NUMBA kernels for `ta` primitives, and retain inspectable generated Python as an intermediate representation (IR). When the AST contains constructs that cannot be represented in nopython mode (user-defined types, maps, drawings), the same visitor emits an object-mode bar loop—still compiled away from the AST walker, but executed by CPython. An auto host mode tries compilation first and falls back to interpretation on hard failure or known gaps, so correctness remains primary.

Contributions.

1. A four-layer compilation architecture for bar-oriented trading DSLs: visitor lowering, numeric kernel library, flat array storage, and engine/runtime façade (Section 3).
2. Detailed AST lowering rules for series assignment, `var`/`varip`, history access, control flow, and plot allocation (Section 4).
3. An incremental numeric kernel library with formal recurrences for SMA/EMA/RSI/Bollinger and complexity analysis $O(n \cdot p)$ vs. $O(n)$ (Section 5).
4. Dual-mode (numeric / object) compilation with an explicit decision procedure (Section 6).
5. Multi-tier warm IR caches, process prewarm, and corrupt Numba cache recovery (Section 7).
6. Host integration with `auto`/`compile`/`interpret` modes and dual-host `interpret` $\leftrightarrow$ `compile` parity harnesses (Sections 8–9).
7. Evaluation of latency, throughput, and numerical residual error on internal 2026 benchmarks (Section 10).

The remainder of the paper is organized as follows. Section 2 reviews the bar model, NUMBA modes, and LLVM JIT. Sections 3–8 develop the design. Section 10 reports measurements. Sections 11–13 discuss related work, limitations, and conclusions.

2 Background

2.1 Bar-oriented evaluation model

A chart of length n is a tuple of aligned series

$$\mathbf{O}, \mathbf{H}, \mathbf{L}, \mathbf{C}, \mathbf{V}, \mathbf{T} \in \mathbb{R}^n, \tag{1}$$

where $\mathbf{T}$ stores bar-open timestamps as Unix milliseconds. Script semantics are defined relative to the current bar index $i \in \{0, \dots, n-1\}$. History access $\mathbf{x}[k]$ denotes the value of series $\mathbf{x}$ at bar $i-k$ (with out-of-range or missing data mapped to the language `na` sentinel, represented as IEEE-754 NaN [IEEE Computer Society, 2019] in the numeric path).

Many indicators are sliding-window or recursive filters. A simple moving average of period p at bar i is

$$\text{SMA}_p(\mathbf{x})_i = \begin{cases} \text{na} & \text{if } i < p-1, \\ \frac{1}{p} \sum_{k=0}^{p-1} \mathbf{x}_{i-k} & \text{otherwise,} \end{cases} \quad (2)$$

provided all window samples are finite; otherwise the result is `na`.

Pine also provides persistent variables (`var`, `varip`) whose initializers run once and whose values carry forward, as well as drawing and strategy side effects that accumulate across the bar walk. These features interact poorly with pure array vectorization when control flow or object identity is required; an explicit bar loop remains the faithful compilation target [Nystrom, 2021, Aho et al., 2006].

2.2 Numba nopython vs. object mode

NUMBA is an LLVM-based JIT for a restricted subset of Python [Lam et al., 2015]. In *nopython* mode (`@jit`), the compiler specializes functions to machine code over concrete array and scalar types; Python objects, dictionaries of heterogeneous values, and many standard library calls are rejected. In *object* mode, NUMBA may still accelerate loops but retains the Python object model, with substantially weaker guarantees.

PYNE does not rely on NUMBA object mode for complex scripts. Instead it implements a first-class *object-mode compilation* path: pure-Python source with a bar loop and NumPy series storage, produced by the same visitor when nopython-incompatible constructs appear. This design choice keeps debugging transparent (readable generated source) and avoids silent semantic drift when NUMBA falls back internally.

2.3 Llvm Jit and warm start

NUMBA lowers specialized Python to LLVM IR and native code [Lattner and Adve, 2004]. First-touch compilation of a script entry point and of the kernel library can cost hundreds of milliseconds to seconds. Subsequent invocations reuse in-memory dispatchers; with `cache=True`, Numba may also persist `.nbi/.nbc` artifacts. Product systems must therefore treat cold JIT as a deploy-time or readiness cost, not as interactive latency—motivating the warm-cache design in Section 7.

3 Compilation Architecture

Compilation is organized as four cooperating layers (Figure 1).

3.1 Layer 1: CompilerVisitor

`CompilerVisitor` (`compiler.py`) is a non-evaluating AST walker in the classic visitor style [Gamma et al., 1994, Aho et al., 2006]. It transpiles statements into a Python source string that indexes OHLCV and user series by an explicit bar index `__bar_idx`. The visitor tracks plot titles, allocates series storage, and sets `object_mode` when non-numeric constructs appear. The generated entry point is always

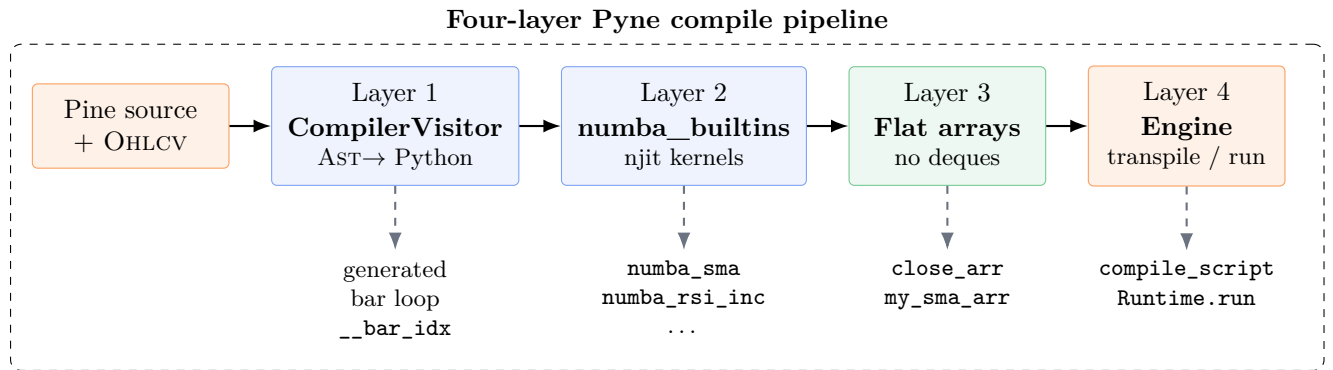


Figure 1: Four-layer compilation architecture. Layer 1 emits readable Python; Layer 2 provides numeric `ta` kernels; Layer 3 stores series as contiguous arrays; Layer 4 packages results for hosts and manages caches.

```
def execute_script_compiled(open_arr, high_arr, low_arr,
                           close_arr, vol_arr, time_arr):
    ...
```

so the host contract is fixed regardless of backend.

3.2 Layer 2: Numba builtins library

Pine’s standard library cannot be invoked from `nopython` code as ordinary Python callables. Selected `ta` and `math` primitives are reimplemented in `numba_builtins.py` as `@numba.njit(cache=True)` functions that accept full arrays plus the current bar index (and, for incremental forms, a small state vector `st`). Expanding this library is the principal path to broader numeric coverage without leaving the JIT path.

3.3 Layer 3: Flat array storage

The compiled engine does not use interpreter `PineSeries` dequeues. Built-in price, volume, and time series are supplied as flat NumPy arrays. User series variables are allocated once as `np.full(n_bars, np.nan)` (or object arrays for UDT series) outside the bar loop, then written at `__bar_idx`. This layout improves cache locality and matches NUMBA’s preferred array model [Harris et al., 2020].

3.4 Layer 4: Engine façade

`engine.py` exposes `transpile`, `compile_script`, and `run_script`. `compile_script` executes generated source in a private namespace and returns a `CompiledScript` whose `run` method accepts OHLCV arrays. Backend `Runtime.run` accepts `mode ∈ {interpret, compile, auto}` and reshapes results into the host response envelope (`plots`, `series`, `drawings`, `object_mode`, cache diagnostics).

Algorithm 1 summarizes the core compile pipeline.

4 AST Lowering

This section details the principal emission rules of `CompilerVisitor`. Figure 2 shows a canonical numeric example side by side.

Algorithm 1 COMPILESCRIPT(source, options)**Require:** Pine source string S ; optional flags (force object mode, cache)**Ensure:** CompiledScript with run callable

```

1:  $k_{\text{raw}} \leftarrow \text{sha256}(S)$ 
2: if  $k_{\text{raw}}$  hits in-process source LRU then
3:   return cached script ▷ warm path  $\sim 0.01$  ms
4: end if
5:  $S' \leftarrow \text{Sanitize}(S)$ ;  $k' \leftarrow \text{sha256}(S')$ 
6: if  $k'$  hits source LRU or disk meta for  $k_{\text{raw}}$  then
7:   rehydrate module; share execute if IR key matches
8:   return CompiledScript
9: end if
10:  $AST \leftarrow \text{Parse}(S')$ 
11:  $(py, meta) \leftarrow \text{CompilerVisitor.visit}(AST)$ 
12:  $k_{\text{IR}} \leftarrow \text{sha256}(py)$ 
13: if  $k_{\text{IR}}$  hits IR LRU then
14:   return script sharing warm execute
15: end if
16:  $ns \leftarrow \text{ExecModule}(py)$  ▷ may JIT under NUMBA
17: wrap nopython warm-up; on TypeError re-emit object mode
18: store source/IR/disk entries; return CompiledScript

```

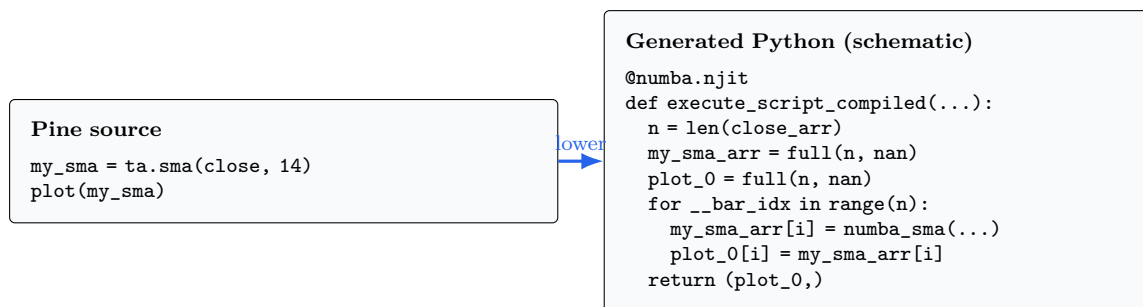


Figure 2: Numeric-mode lowering: series assignment and `plot` become full-length array writes inside an explicit bar loop. The host packs plot titles from visitor metadata.

4.1 Series assignment and storage

A bare assignment $y = e$ at script scope allocates y_arr and emits $y_arr[_bar_idx] = [e]$ inside the loop, where $[e]$ denotes the recursive expression translation. Temporary expressions without names may still allocate storage when they feed plots or later history references.

4.2 var and varip

Pine `var` initializes once on the first evaluation and carries the value forward; `varip` similarly persists across realtime bar updates in the host model. Lowering emits a guarded initializer:

```

if __bar_idx == 0:
    v_arr[__bar_idx] = <init>
else:
    v_arr[__bar_idx] = v_arr[__bar_idx - 1]
# optional subsequent assignments overwrite the carried value

```

This matches carry semantics without interpreter bookkeeping tables.

4.3 History access

An expression `close[n]` lowers to a NaN-safe index computation:

$$\llbracket \mathbf{x}[k] \rrbracket = \begin{cases} \mathbf{x}_{i-k} & 0 \leq i - k < n, \\ \text{na} & \text{otherwise,} \end{cases} \quad (3)$$

with integer coercion that treats non-finite offsets as out-of-range. Combinations such as `high[ta.highestbars(...)]` therefore stay in-bounds when the window helper returns a negative bars-back offset (TradingView / interpret contract).

4.4 Control flow

`if/else`, `for`, `while`, and `switch` lower to the corresponding Python constructs inside the bar loop. Branching that assigns series must write the active branch's value at `__bar_idx`; inactive paths leave prior NaN or carried `var` state depending on Pine rules. Loops that walk bars in user code are nested inside the outer `__bar_idx` traversal—a potential quadratic pattern that hosts may later warn about, but which remains semantically faithful.

4.5 Plot, fill, and output allocation

Each `plot(expr, title=...)` allocates a sink array and assigns `llbracket expr \rrbracket` each bar. Titled `fill(plot1, plot2, ...)` exports a series key for interpret/compile plot-key parity and downstream chart band wiring; the compile path leaves the fill series as float NaN while retaining band color / plot references in plot metadata. Numeric mode returns a tuple of series arrays (host packs titles); object mode returns a dictionary that may also include `__drawings`.

4.6 Time and request.security

The host always supplies `time_arr` (Unix ms). When the caller omits time, the engine synthesizes `bar × 60 000`. Bare `time`, `time_close`, and calendar parts lower against `time_arr`.

For `request.security`, compile lowers *same-symbol simple OHLCV expressions* as chart series passthrough only. Foreign tickers and complex expressions (user-defined functions, non-trivial aggregations) emit `na`—no invention of chart close as dividends or fundamentals. This aligns with the interpret foreign-na policy.

4.7 Tuple unpack and inputs

Tuple unpack such as `[u,m,l] = ta.bb(...)` expands to multiple array writes from a multi-return kernel. `input.*` defaults are resolved at compile/load time into constants or host-provided overrides; non-empty dynamic input maps may gate auto-mode toward interpret for safety (Section 8).

5 Numeric Kernel Library

5.1 Design constraints

Kernels must: (i) be nopython-legal; (ii) accept full arrays + bar index; (iii) agree with the interpret oracle within floating-point noise on shared goldens; (iv) prefer incremental $O(1)$ amortized updates for sequential bar walks. Full-window recomputation kernels remain available as reference implementations and for non-sequential access.

Algorithm 2 INCREMENTALSMA(x, p, i, st)**Require:** array x , period p , bar i , state $st = [s, last]$ **Ensure:** SMA at i ; mutates st

```

1: if  $i < 0$  or  $p \leq 0$  then return na
2: end if
3:  $last \leftarrow$  integer from  $st[1]$ , or  $-1$  if unset
4: if  $i < last$  then ▷ rewind
5:    $st[0] \leftarrow$  na;  $last \leftarrow -1$ 
6: end if
7: for  $j = last + 1$  to  $i$  do
8:   if  $j < p - 1$  then  $s \leftarrow$  na
9:   else if  $j = p - 1$  then  $s \leftarrow$  sum of  $x[0..p)$  if all finite else na
10:  else update  $s$  via (4); reseed if  $s$  was na
11:  end if
12: end for
13:  $st[0] \leftarrow s$ ;  $st[1] \leftarrow i$ 
14: return  $s/p$  if  $s$  finite else na

```

5.2 SMA and sliding-window sum

Equation (2) costs $O(p)$ per bar if re-summed, hence $O(n \cdot p)$ per script. The incremental form maintains a running sum s :

$$s_i = \begin{cases} \sum_{k=0}^{p-1} \mathbf{x}_k & i = p - 1, \\ s_{i-1} + \mathbf{x}_i - \mathbf{x}_{i-p} & i \geq p, \end{cases} \quad (4)$$

with NaN poisoning rules matching the full kernel (any non-finite window sample yields na and may force reseed). Algorithm 2 sketches the state-vector update used by `numba_sma_inc`.

Complexity. Naive full SMA over a script: $\Theta(n \cdot p)$ additions. Incremental sequential walk: $\Theta(n)$ additions after $O(p)$ seed, i.e. $O(n)$ total—independent of p in the steady state.

5.3 EMA and Wilder RMA

The exponential moving average with smoothing $\alpha = 2/(p + 1)$ is

$$e_i = \alpha \mathbf{x}_i + (1 - \alpha) e_{i-1}, \quad (5)$$

after an SMA seed over the first p finite samples (compile-family contract). Wilder’s running moving average (RMA) uses $\alpha = 1/p$:

$$r_i = \frac{1}{p} \mathbf{x}_i + \left(1 - \frac{1}{p}\right) r_{i-1}. \quad (6)$$

Both admit $O(1)$ per-bar incremental updates with a length-2 state vector $[value, last_i]$ and gap catch-up when bars are skipped.

5.4 RSI (Wilder)

Relative Strength Index [Wilder, 1978] seeds average gain/loss with the SMA of the first p price deltas, then applies RMA:

$$g_i = \max(\Delta_i, 0), \quad \ell_i = \max(-\Delta_i, 0), \quad (7)$$

$$\bar{g}_i = \frac{1}{p}g_i + \left(1 - \frac{1}{p}\right)\bar{g}_{i-1}, \quad (8)$$

$$\text{RSI} = 100 - \frac{100}{1 + \bar{g}/\bar{\ell}}, \quad (9)$$

with the convention $\text{RSI} = 100$ when $\bar{\ell} = 0$ after a valid seed. The first valid bar is $i = p$ (p deltas need $p + 1$ prices). This matches the interpret oracle and TradingView `ta.rsi`; earlier mistaken “rolling mean of gains” implementations were corrected in residual work (2026-08).

5.5 Bollinger bands

With middle band $m = \text{SMA}_p(x)$ and standard deviation σ over the same window [Bollinger, 2001],

$$\text{upper} = m + k\sigma, \quad \text{lower} = m - k\sigma, \quad (10)$$

typically $k = 2$. Incremental forms maintain the sliding sum and sum of squares (or an equivalent stable update [Higham, 2002]) so that σ is $O(1)$ amortized per bar.

5.6 Range statistics and rank

`highest/lowest` maintain window extrema; `highestbars/lowestbars` return *negative* bars-back offsets to the extremum, or -1.0 when the window is all-na. `ta.rci` implements Spearman rank correlation [Spearman, 1904] over a fixed window, matching the interpret rank contract.

5.7 Kernel inventory (landed subset)

Representative nopython kernels include: `sma`, `ema`, `rma`, `wma`, `hma`, `rsi`, `atr`, `bb`, `bbw`, `macd`, `stdev`, `cci`, `stoch`, `tsi`, `cmo`, `wpr`, `median`, `rci`, `vwap`, `linreg`, and many `*_inc` variants. Round 7 residual work landed `median`, `wpr`, `cmo`, and `bbw` on the nopython path.

6 Object-Mode Compilation

6.1 When object mode is selected

Figure 3 shows the decision tree. The visitor sets `object_mode` when it observes:

- user-defined type (UDT) definitions or instantiations;
- `map` operations (heterogeneous dictionaries);
- drawing APIs (`line`, `label`, `box`, `table`, ...);
- strategy side effects that require broker state beyond pure series;
- string/color series that escape numeric representation;
- library `import` surfaces that are not kernelized;
- explicit `force_object_mode` after nopython warm-up `TypeError`.

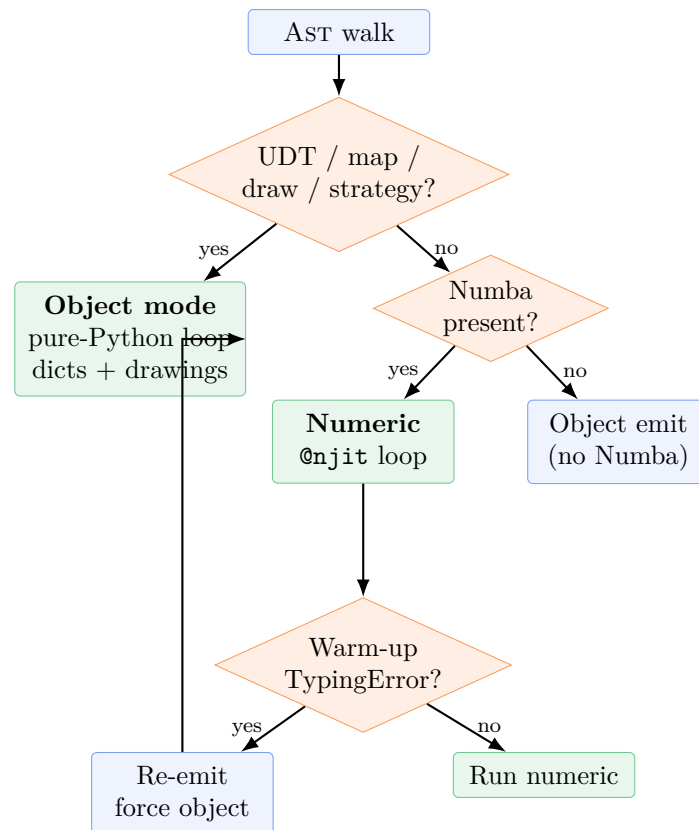


Figure 3: Numeric vs. object mode decision tree. Structural features force object emit; nopython warm-up failures recover by re-visiting with `force_object_mode=True`.

6.2 Representation

In object mode:

- UDT instances are field dictionaries (and nested structures as nested dicts where supported);
- maps are ordinary Python `dicts`;
- drawing calls append structured events to a `__drawings` list;
- plot series remain full-length NumPy arrays for host parity.

Arithmetic involving Pine `na/None` is routed through `na_num` helpers. The response includes `object_mode=True` so callers can distinguish backends.

6.3 Performance trade-off

Object mode forgoes peak JIT throughput but still avoids per-expression AST visitation. For drawing-heavy or UDT-centric scripts it is the correctness-preserving default; numeric kernels remain available for pure series subexpressions when star-imported helpers are pure Python wrappers.

7 Caching and Warm Start

7.1 Cache layers

Product warm-compile (roadmap item H2) stacks several layers (Table 1):

Table 1: Compile cache layers and scope.

Layer	Key	Bound	Cross-process
Source LRU	sha256(raw/sanitized)	128	no
IR LRU	sha256(generated Python)	64	no
Host Runtime cache	raw source sha256	bounded	no
Disk module + meta	under cache dir	on by default	yes
Numba .nbc	function identity	file-backed	yes

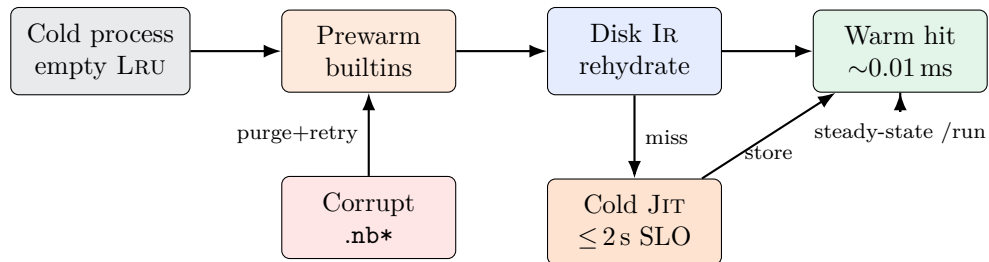


Figure 4: Warm-cache state machine. Deploy readiness should pay cold JIT via prewarm; interactive paths prefer in-process and disk hits.

Environment defaults favor production: `PYNE_COMPILE_DISK_CACHE=1`, `PYNE_COMPILE_PREWARM=1`, and a persistent directory (`PYNE_COMPILE_CACHE_DIR`, Docker `/data/compile-cache`).

7.2 State machine

Figure 4 depicts the warm-cache state machine from process start through steady state.

7.3 Prewarm hooks

- Python: `prewarm_numba_builtins()`, `prewarm_scripts([...])`, `ensure_compile_cache_dir()`;
- CLI: `pynescrypt prewarm [PATH...]`;
- Pro API: `POST /compile/prewarm`; `GET /health` exposes a `compile` section;
- Lazy host prewarm on first non-test `/run` when enabled.

7.4 Corrupt cache recovery

Truncated `.nbi/.nbc` files raise `EOFError` or `pickle.UnpicklingError` on load. Algorithm 3 matches the engine’s once-retry recovery.

Disk IR clear (`clear_disk_compile_cache`) remains a separate operator action from Numba function-cache purge.

7.5 Indicative SLOs

Targets for operations (not hard CI gates), workstation + Numba:

- interpret minimal @ 2k bars: ≤ 25 ms median;
- interpret `ta_combo` @ 2k: ≤ 200 ms median;
- cold compile first SMA-class script: ≤ 2 s;

Algorithm 3 CORRUPTCACHERECOVERY(f , $args$)**Require:** callable f that may load Numba disk caches**Ensure:** result of f or re-raised non-corruption error

```

1: try: return  $f(args)$ 
2: on exception  $e$ :
3: if not ISNUMBACACHECORRUPTION( $e$ ) then
4:   raise  $e$ 
5: end if
6:  $n \leftarrow$  CLEARNUMBAFUNCTIONCACHES() ▷ unlink .nbi/.nbc
7: log warning with  $n$  purged files
8: return  $f(args)$  ▷ single retry

```

Algorithm 4 AUTOMODE(source, ohlcv, inputs)**Require:** script S , market data D , optional inputs I **Ensure:** host result dict with **series/plots**

```

1: if  $I$  non-empty or  $S$  has top-level import/request then
2:   return INTERPRET( $S, D, I$ ) with auto_backend=interpret
3: end if
4: if not HASNUMBA() and numeric-only path required then
5:   attempt object compile or fall back interpret
6: end if
7: try:  $R \leftarrow$  COMPILEANDRUN( $S, D$ )
8: on success: return  $R$  with auto_backend=compile
9: on exception  $e$ :
10: return INTERPRET( $S, D, I$ ) with compile_fallback_reason=MSG( $e$ )

```

- warm compile (source cache hit): ≤ 1 ms (typically ~ 0.01 – 0.05 ms);
- warm run **ta_combo** @ 5k: ≤ 5 ms (often ~ 1 ms);
- cross-process disk rehydrate: $\leq \sim 60$ – 70% of full cold.

8 Host Integration and Auto Mode

8.1 Runtime modes

auto

Default on Pro API **/run** and **/run/batch**. Prefer compile when eligible; fall back to interpret on hard failure or known gaps.

compile

Compile only (error if transpile/exec fails; nopython $\rightarrow$ object recovery still occurs inside compile).

interpret

AST evaluator only.

Safe auto gates (no silent wrong results) include: non-empty dynamic **inputs** $\rightarrow$ interpret; top-level **import** / **request.*** $\rightarrow$ interpret; any compile error $\rightarrow$ interpret with **compile_fallback_reason**. Response fields include **auto_backend**, **compile_cached**, **compile_ms**, and optional **nopython_fallback_reason**.

8.2 Engine Api surface

Public entry points:

- `transpile(source) → str` — generated Python for inspection;
- `compile_script(source) → CompiledScript`;
- `run_script(source, ohlcv) → result dict`;
- cache helpers: `clear_compile_cache`, `clear_disk_compile_cache`, `clear_numba_function_caches`, `compile_cache_stats`.

8.3 Result envelope

Both modes normalize into a host-facing structure with plot series, optional drawings, bar count, identifiers, and backend flags. Numeric mode avoids returning Numba `typed.Dict` across the boundary (tuple packing + host title map) to eliminate expensive conversion observed in early prototypes.

9 Correctness: Dual-Host Parity

Compilation is only useful if results match the interpret oracle on the supported surface. PYNE maintains dual-host checks rather than claiming third-party certification.

9.1 Harness

`scripts/compare_interp_compile.py` runs the same scripts under `mode=interpret` and `mode=compile`, compares `result["series"]` with nan-aware `allclose`, and buckets residuals (OK, `fill_background_only`, `both_error_same`, `MISMATCH`, ...). Always-on smoke tests live in `tests/test_interp_compile_parity.py`; focused foreign-security cases in `tests/test_dividend_yield_parity.py`.

9.2 Landed correctness notes (2026-08)

- `time_arr` always present (host real timestamps or synthetic ms).
- `request.security`: same-symbol simple OHLCV passthrough; foreign/complex $\rightarrow$ na.
- RSI Wilder seed then RMA (Section 5).
- `highestbars/lowestbars`: negative bars-back; all-na $\rightarrow -1.0$.
- Titled `fill()` series keys for plot-key parity.
- Numba cache recovery from corrupt `.nbi/.nbc`.
- `ta.rci` Spearman rank.

9.3 Error metrics

For aligned series a, b of length n , with mask $m_i = \neg(\text{isnan}(a_i) \vee \text{isnan}(b_i))$,

$$\varepsilon_{\max} = \max_{i:m_i} |a_i - b_i|, \quad \varepsilon_{\text{mean}} = \frac{1}{|m|} \sum_{i:m_i} |a_i - b_i|. \quad (11)$$

Kernel parity studies report $\varepsilon_{\max} \sim 10^{-11}$ attributable to floating associativity, not algorithmic divergence [Higham, 2002, IEEE Computer Society, 2019]. NaN patterns must also match (both na or both finite) for a series pair to be considered OK.

Table 2: Warm compile execute medians at $n = 5000$ bars (internal 2026).

Script	Median (ms)	Notes
SMA	~ 0.04	single window MA
RSI	~ 0.08	Wilder incremental
BB	~ 0.15	bands + stdev
TSI	~ 0.11	double-smoothed
MULTI (SMA+EMA+RSI)	~ 0.21	multi-plot
ta_combo warm compile hit	~ 0.01	cache only
ta_combo warm run	~ 1.06	full combo @5k

10 Evaluation

10.1 Methodology

Unless noted, measurements use workstation-class hardware, CPython with Numba installed, synthetic rising or random-walk OHLCV, and medians over repeated runs after explicit cache clears or warm-ups as labeled. Numbers are *internal 2026 engineering benchmarks*, not a multi-platform public reproducibility suite; they characterize order of magnitude and design progress across compile performance rounds.

10.2 Compile vs. execute latency

Table 2 reports warm compiled execute medians at $n = 5000$ for representative indicators after incremental-kernel work.

Early MACD/EMA paths re-accumulated from bar 0 on every index, producing near-quadratic execute times (~ 33 ms at 5k bars). After incremental state vectors, MACD execute fell by $\sim 110\times$ and MULTI by $\sim 49\times$ relative to those broken baselines on the same machine class (compile+exec vs. list-based interpret for MACD/MULTI class workloads remains in the tens-to-hundreds $\times$ range depending on interpret plot packing).

10.3 Cold vs. warm compile

Cold compile is dominated by Numba first-touch of the generated entry and builtins (often ~ 0.4 – 2 s for SMA-class scripts depending on process cache state). Warm source-LRU hits are ~ 0.01 – 0.05 ms. Disk IR rehydrate is cheaper than full cold but not free (Section 7 SLOs).

10.4 Throughput charts

Figure 5 visualizes schematic latency bars for interpret vs. cold compile vs. warm compile+run on a multi-indicator workload, using the internal measurements above as anchors.

Figure 6 contrasts naive $O(n \cdot p)$ vs. incremental $O(n)$ kernel cost as period grows (schematic, unit work normalized).

10.5 Parity residuals

Across kernel golden suites, compiled vs. full-kernel or interpret oracles typically achieve $\varepsilon_{\max} \leq 10^{-10}$ to 10^{-11} for continuous indicators. Structural residuals (hline-only keys, fill background series) are classified separately from true value mismatches by the dual-host harness.

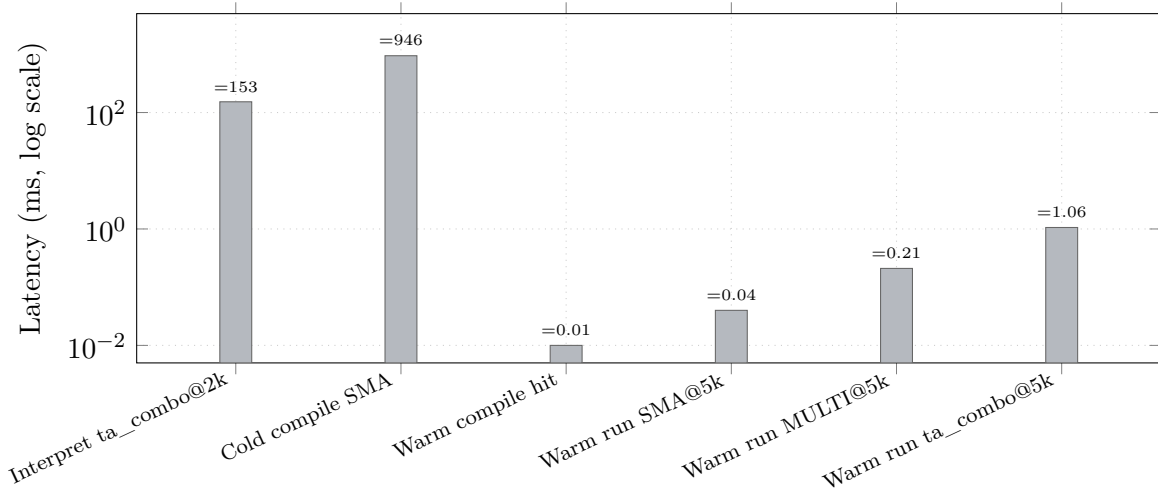


Figure 5: Indicative latencies (internal 2026): interpret combo residual vs. cold JIT, warm compile cache hits, and warm numeric executes. Log scale emphasizes the multi-order-of-magnitude gap between cold JIT and warm hits.

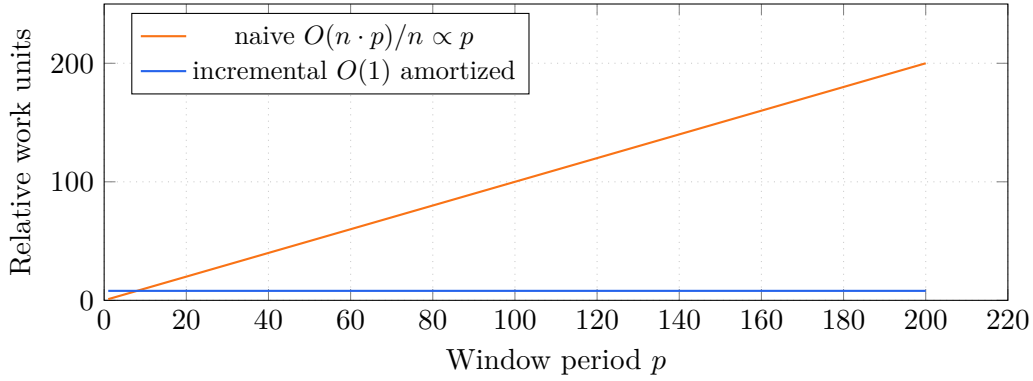


Figure 6: Asymptotic per-bar work for sliding SMA-style kernels: naive recompute grows with p ; incremental state remains flat after seeding.

10.6 Object mode

Object-mode UDT/plot scripts compile in tens of milliseconds cold (no Numba entry JIT) and run on the order of single-digit milliseconds at 5k bars in microbenchmarks—slower than numeric mode but faster than full AST interpretation for comparable expression volume, with broader language coverage.

11 Related Work

Numba and Python Jit. Lam *et al.* introduce NUMBA as an LLVM-based JIT for array-oriented Python [Lam et al., 2015]. Our work does not extend Numba itself; it is a domain-specific front-end that emits Numba-legal programs and a parallel object-mode path when the domain language exceeds nopython. Nuitka and similar whole-program compilers [Hayen, 2024] pursue different packaging goals without bar-series semantics.

Financial and trading Dsls. Commercial platforms provide proprietary compilers for bar languages; public literature more often discusses time-series libraries (e.g. pandas [McKinney, 2017]) than closed-form compilation of Pine-like semantics. PYNE targets an open, inspectable

intermediate Python IR rather than an opaque native bytecode blob.

Partial evaluation and staging. Source-to-source lowering with residual bar loops resembles online partial evaluation [Jones et al., 1993]: market data remains dynamic while script structure is specialized into straight-line array code. We do not currently apply binding-time analysis formally; the visitor encodes domain knowledge (series vs. scalar, plot allocation) directly.

Dual-mode virtual machines. Self-optimizing and dual-mode engines (e.g. object vs. optimized paths in dynamic language VMs [Chambers and Ungar, 1990]) inspire our numeric/object split. The difference is that both modes are produced by ahead-of-time source generation, not by runtime tracing of an interpreter core.

Compiler construction. Classic pipelines [Aho et al., 2006, Cooper and Torczon, 2011] separate front-end, IR, and back-end. We reuse ANTLR+ASDL as the front-end, generated Python as a human-readable IR, and Numba/LLVM or CPython as back-ends—trading maximal optimization for packaging simplicity and debuggability [Leroy, 2009].

12 Limitations and Future Work

- **No TradingView certification.** Numerical and structural parity is measured against PYNE’s interpret oracle and public documentation, not against a licensed reference binary.
- **Foreign multi-asset data.** Compile does not invent foreign series; complex `request.security` yields `na`. Full multi-ticker hosts remain future work.
- **Strategy breadth.** Complex order/`StrategyState` paths remain interpret-first; basic entry/close exist in object mode.
- **Drawing fidelity.** Deletes and full style parity with interpreter registries are incomplete.
- **Nested UDTs/methods.** Coverage is partial; object mode is the escape hatch.
- **True AOT.** Disk + Numba `.nbc` is not a portable ahead-of-time binary; multi-worker deploys still warm per process unless coordinated.
- **mypyc / native kernels.** Deferred; residual interpret plot packing still dominates some combo walls on the non-compile path.

Future work prioritizes expanding the nopython kernel surface so fewer scripts fall out of numeric mode, unifying dual-host Runtime packages, and growing the interpret $\leftrightarrow$ compile golden corpus under the parity harness rather than ad-hoc scripts.

13 Conclusion

We presented the PYNE NUMBA compiler path for bar-oriented trading DSLs: a four-layer architecture that lowers a Pine-compatible AST to explicit-index Python, executes pure numeric scripts under NUMBA nopython JIT, and falls back to an object-mode bar loop for UDTs, maps, and drawings. Incremental $O(n)$ kernels, multi-tier warm IR caches, corrupt-cache recovery, and host auto-mode with interpret fallback make the system practical for interactive APIs and large histories.

Internal 2026 measurements show warm compile hits near 10^{-2} ms, warm indicator executes on the order of 10^{-1} ms at 5,000 bars, and large speedups versus list-based interpretation once

quadratic kernel bugs were eliminated. Dual-host nan-aware parity keeps residuals at floating-point noise for landed kernels. The design emphasizes inspectable intermediates and correctness-first auto fallback over unverifiable peak claims.

Acknowledgments

We thank PYNE and HOOX contributors for performance rounds, parity harnesses, and production warm-compile integration. Pine 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.

References

References

- Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson, Boston, 2nd edition, 2006.
- John Bollinger. Bollinger on bollinger bands. *McGraw-Hill*, 2001.
- Craig Chambers and David Ungar. Iterative type analysis and extended message splitting. In *PLDI*, 1990.
- Keith Cooper and Linda Torczon. *Engineering a Compiler*. Morgan Kaufmann, 2nd edition, 2011.
- Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: Elements of reusable object-oriented software. *Addison-Wesley*, 1994.
- 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.
- IEEE Computer Society. IEEE standard for floating-point arithmetic. *IEEE Std 754-2019*, 2019. doi: 10.1109/IEEESTD.2019.8766229.
- Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. Partial evaluation and automatic program generation. Prentice Hall, 1993.
- 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.
- Xavier Leroy. Formal verification of a realistic compiler. volume 52, pages 107–115, 2009.

- Wes McKinney. Python for data analysis. O'Reilly, 2017.
- John J. Murphy. *Technical Analysis of the Financial Markets*. New York Institute of Finance, 1999.
- Robert Nystrom. *Crafting Interpreters*. Genever Benning, 2021.
- Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, Raleigh, NC, 2013.
- Charles Spearman. The proof and measurement of association between two things. *American Journal of Psychology*, 15:72–101, 1904.
- 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.

Efficient Series History and Incremental Technical Analysis for Bar-Loop Interpreters

HOOX Research / PYNE Contributors
 jango_blockchained
 Independent Open Source Research
 contact@hoox.sh

August 2026

Abstract

Bar-loop interpreters for series-oriented domain-specific languages evaluate every statement once per time bar and materialize history via the operator $x[n]$ (“ n bars back”). Naive implementations store unbounded per-series lists and recompute rolling technical-analysis (TA) windows from scratch on each bar, inducing $O(n \cdot s)$ memory growth and $O(n \cdot p)$ work for period- p indicators over n bars and s series. We present the series-history and incremental-TA stack shipped in PYNE, an independent open toolchain for the Pine Script™ language [HOOX Project, 2026, TradingView, Inc., 2024]. The design combines (i) a default-on series cap that trims chronological host lists to $\max(SERIES_MAX, max_bars_back) + slack$, yielding $\sim 78\times$ long-run list-memory reduction at 20 000 bars; (ii) an opt-in chronological ring buffer (`ChronologicalSeriesBuffer`) with true $O(1)$ lookback under modular indexing; (iii) call-site incremental kernels for the hot `ta` surface—including SMA/E-MA/RMA, RSI, Bollinger Bands, highest/lowest, KAMA, and StochRSI—with kernel micro-benchmarks up to $\sim 717\times$ versus full-window rebuilds; and (iv) host-side parse/AST caching, lazy calendar materialization, and light plot modes that cut multi-plot residual wall time. On the interpret path at 2 000 bars we report median latencies of 15.2 ms (minimal), 23.6 ms (`ta_sma`), 153 ms (`ta_combo`), and 69 ms (`strategy_ish`); cProfile attributes $\sim 75\%$ of residual `ta_combo` wall time to plot packing rather than TA kernels. We formalize correctness conditions under bounded history, give recurrences and stability notes for each kernel class, and situate the work in incremental computation and streaming algorithms.

Keywords. incremental computation; technical analysis; series buffers; ring buffers; memory-bounded interpreters; financial time series.

Disclaimer. Pine 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.

Contents

1	Introduction	3
2	Background	3
2.1	Pine series semantics	3
2.2	Bar-loop evaluation model	4
2.3	Cost model of interpretation	4

3	Problem Formalization	5
3.1	Memory: unbounded series growth	5
3.2	Time: full-window TA recompute	5
3.3	Correctness constraints	5
4	Series Cap Design	6
4.1	Policy	6
4.2	Correctness argument for bounded history	6
4.3	When lookback exceeds the cap	6
4.4	Structural memory win	6
5	Ring Buffer Series	7
5.1	ChronologicalSeriesBuffer	7
5.2	Dual-write transition	7
6	Incremental TA Framework	8
6.1	Generic pattern	8
6.2	Case study: SMA (sliding sum)	8
6.3	Case study: EMA and RMA (Wilder)	9
6.4	Case study: RSI	9
6.5	Case study: Bollinger Bands (mean + variance)	9
6.6	Case study: highest / lowest (monotonic dequeues)	10
6.7	Case study: KAMA	10
6.8	Case study: StochRSI	10
6.9	Additional kernels	11
6.10	Numerical stability notes	11
7	Interpreter Hot Paths	11
7.1	Dispatch	11
7.2	Plot packing	11
7.3	Light plots and lazy calendar	11
8	Parse/AST Caching Interaction with Runtime Multi-Run	12
9	Evaluation	13
9.1	Methodology	13
9.2	Memory	13
9.3	Kernel speedups	13
9.4	End-to-end interpret latency	13
9.5	Compile path comparison	14
9.6	Feature flags summary	14
10	Related Work	14
11	Limitations	15
12	Conclusion	15
A	Memory bound (informal)	16
B	Environment reproduction notes	16

1 Introduction

Financial charting languages such as Pine Script™ [TradingView, Inc., 2024] expose a *series* data model in which every scalar expression is implicitly a time series indexed by bar. Evaluation proceeds as a *bar loop*: the host advances an OHLCV cursor and re-executes the script body for each bar $i \in \{0, \dots, n-1\}$. History access $x[n]$ denotes the value of x that was current n bars earlier; qualifiers `var` and `varip` persist state across bars and intra-bar ticks respectively; and the sentinel `na` propagates through arithmetic [TradingView, Inc., 2024, HOOX Project, 2026].

This model is attractive for domain users—indicators are written as if they were spreadsheet formulas—but is expensive for a naive interpreter. Historically, the dominant costs on PYNE’s interpret path were:

1. **Series history growth.** Every named series and host OHLCV field appended one sample per bar, so memory scaled as $\Theta(n \cdot s)$ for s live series.
2. **Full-window TA recompute.** Indicators such as $\text{SMA}(p)$, Bollinger Bands, and Kaufman adaptive moving average [Kaufman, 2013] rebuilt period- p windows from the entire prefix on each bar, producing $\Theta(n \cdot p)$ or even $\Theta(n^2)$ work for nested rebuilds.
3. **Plot packing and call dispatch.** Multi-plot scripts spent a majority of residual wall time in result capture and `visit_Call` envelopes rather than pure arithmetic.

PYNE addresses these costs with a layered runtime stack measured across performance rounds 4–7 [HOOX Project, 2026]. Contributions of this paper:

- **Bounded series history** via default-on `PYNE_SERIES_CAP`, with a correctness argument relating lookback depth, recursive incremental state, and out-of-range `na` semantics (Section 4).
- **Chronological ring buffers** (`ChronologicalSeriesBuffer` / `PYNE_SERIES_RING`) enabling $O(1)$ Pine-offset lookback and a dual-write migration path off newest-first dequeues (Section 5).
- **A generic incremental TA framework** of call-site state tuples σ and updates $\sigma' = \text{update}(\sigma, x_i)$, covering sliding sums, exponential smoothers, Wilder-class oscillators, variance bands, amortized dequeues, and adaptive kernels (Section 6).
- **Interpreter and multi-run host optimizations:** type-keyed dispatch, light plots, lazy calendar, and a SHA-256 LRU AST cache with call-site scrubbing on hit (Sections 7–8).
- **Empirical evaluation** of memory, kernel micros, and end-to-end latency, including a residual bottleneck map that isolates plot packing as $\sim 75\%$ of multi-plot interpret wall time (Section 9).

Scope. We focus on the *interpret* (AST-walking) path and on the series/TA subsystem shared with the Numba compile path [Lam et al., 2015], which always emits incremental kernels for hot TA. Grammar, strategy broker semantics, and LSP tooling are treated elsewhere in the PYNE paper series.

2 Background

2.1 Pine series semantics

Definition 2.1 (Series and history operator). A series $\mathbf{x}$ is a sequence $(x_0, x_1, \dots, x_i)$ of values observed up to the current bar index i . The history operator is

$$x[n] = \begin{cases} x_{i-n} & \text{if } 0 \leq n \leq i \text{ and } x_{i-n} \neq \text{na}, \\ \text{na} & \text{otherwise (including } n < 0). \end{cases} \quad (1)$$

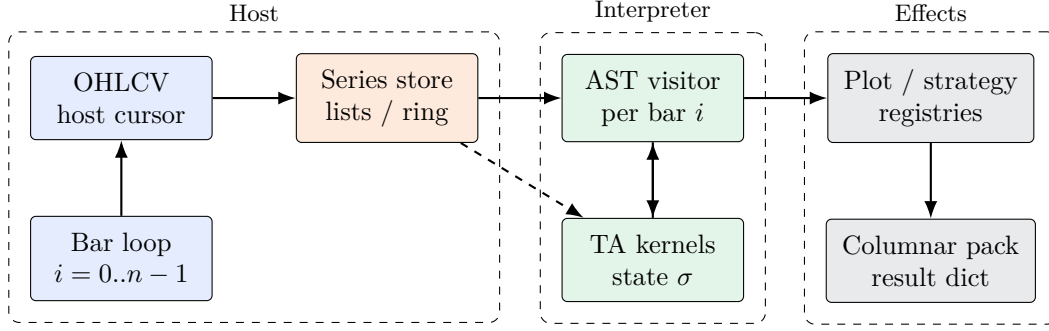


Figure 1: Bar-loop interpret architecture with series store, call-site TA state, and plot packing. The host advances OHLCV once per bar; the visitor re-executes the script body; incremental kernels read/write state σ keyed by call site.

Pine indices are *most-recent-first* in the programmer model ($x[0]$ is current), while host chronological lists store oldest-first so that physical index $-(n + 1)$ realizes lookback n [TradingView, Inc., 2024].

Definition 2.2 (Persistence qualifiers). The qualifier `var` initializes a declaration site once and carries the value across subsequent bars. `varip` additionally carries across ticks within an incomplete bar. Bare assignment re-evaluates every bar.

Definition 2.3 (na propagation). Arithmetic and many builtins propagate `na`: if any operand is `na`, the result is `na` (with Pine-specific exceptions for functions that treat missing samples specially). Out-of-range history must yield `na`, never a silent zero [HOOX Project, 2026].

These rules interact with memory bounding: truncating history deeper than any live look-back must not invent samples, and recursive smoothers must either retain enough raw history *or* retain closed-form incremental state (Section 4).

2.2 Bar-loop evaluation model

Figure 1 sketches the interpret architecture. For each bar i :

1. The host advances OHLCV series (`open`, `high`, `low`, `close`, `volume`, `time`) and optional script-local series.
2. The evaluator walks the script AST via a visitor, resolving names against a context map and call-site TA state.
3. Builtins (notably `ta`) either update incremental kernels or fall back to full-window helpers when `PYNE_TA_INCREMENTAL = 0`.
4. Plot/strategy side effects append into result registries for packing into columnar outputs.

The compile path lowers the same semantics to NumPy/Numba kernels that process entire arrays in one call [Harris et al., 2020, Lam et al., 2015]; that path always uses incremental kernels for hot TA and is not gated by the interpret flag.

2.3 Cost model of interpretation

Let n be the number of bars, s the number of live series (host + local), c the number of call nodes executed per bar, p a typical TA period, and q the number of `plot` (or similar) emissions per bar. A crude cost model for a naive interpreter is

$$T_{\text{naive}}(n) = \underbrace{O(n \cdot c \cdot d)}_{\text{AST dispatch depth } d} + \underbrace{O(n \cdot \sum_k p_k)}_{\text{full-window TA}} + \underbrace{O(n \cdot q \cdot w)}_{\text{plot capture width } w} + \underbrace{O(n \cdot s)}_{\text{series append}}. \quad (2)$$

Memory is $M_{\text{naive}} = \Theta(n \cdot s)$ plus plot buffers $\Theta(n \cdot q)$.

With series caps of capacity K , incremental TA of $O(1)$ or $O(p)$ amortized per call, and light plots that skip columnar capture, the improved model is

$$T_{\text{opt}}(n) = O(n \cdot c \cdot d') + O(n \cdot \sum_k u_k) + O(n \cdot q \cdot w') + O(n \cdot s), \quad (3)$$

where $u_k \in \{1, p_k\}$ is the per-bar update cost of kernel k , $d' \leq d$ after dispatch tuning, $w' \ll w$ under light plots, and $M_{\text{opt}} = \Theta(K \cdot s) + \Theta(|\sigma|)$ with $|\sigma|$ the aggregate incremental state (typically $O(p)$ per call site).

Dominant historical costs. Profiling rounds identified four hotspots: series list growth, TA full recompute, plot packing, and the `visit_Call` envelope. Kernel-only wins therefore do not automatically dominate end-to-end multi-plot scripts (Section 9).

3 Problem Formalization

3.1 Memory: unbounded series growth

Proposition 3.1 (Linear series memory). *Under uncapped chronological lists, if a script maintains s series each of length n after n bars, then pointer storage alone is $\Theta(n \cdot s)$ words. For $n = 20\,000$, $s = 6$ OHLCV-style lists, this is $\sim 9.6 \cdot 10^5$ pointers (~ 7.7 MiB on 64-bit), exclusive of object payloads.*

Many Pine scripts also allocate local series for intermediate expressions; s can reach dozens to hundreds on complex strategies, so memory pressure is not limited to OHLCV.

3.2 Time: full-window TA recompute

Definition 3.2 (Full-window recompute). A kernel f_p is a *full-window recompute* if at bar i it scans $\{x_{i-p+1}, \dots, x_i\}$ (or the entire prefix) without reusing work from bar $i - 1$, costing $\Omega(p)$ or $\Omega(i)$ arithmetic per bar.

Proposition 3.3 (Quadratic nested rebuild). *If each bar i rebuilds an indicator by scanning a prefix of length i (e.g., nested KAMA or StochRSI implementations that re-seed from bar 0), total work over n bars is $\Theta(n^2)$.*

Even “only” $\Theta(n \cdot p)$ is painful for $n = 5\,000$ and $p = 200$ when dozens of indicators share a script. Time-series textbooks emphasize online/streaming updates for exactly this reason [Box and Jenkins, 1970, Tsay, 2010].

3.3 Correctness constraints

Any optimization must preserve:

1. **Lookback identity:** for all $n \leq K$ (cap), $x[n]$ equals the uncapped value when the sample exists.
2. **na discipline:** OOB and missing samples are `na`, never 0.
3. **TA parity:** incremental kernels match full-window oracles within floating-point noise [Higham, 2002, IEEE Computer Society, 2019] under default flags.
4. **Call-site isolation:** two syntactically distinct calls to `ta.sma(close, 14)` maintain independent state.

4 Series Cap Design

4.1 Policy

PYNE formalizes host list capping behind `PYNE_SERIES_CAP` (default `on`). Let

$$K = \max(K_0, \text{max_bars_back}(\text{src})), \quad (4)$$

where $K_0 = \text{SERIES_MAX}$ defaults to 256 (overridable via `PYNE_SERIES_MAX`) and `max_bars_back` is parsed from the script source when present. Chronological lists are grown to $K + \Delta$ with slack $\Delta = 64$, then trimmed in place with `del lst[:drop]` back to K . Slack amortizes the $O(K)$ cost of prefix deletion so trims are not performed every bar.

PineSeries wrappers retain a history floor of 1000 samples (raised when K is larger) so `close[n]` for $n \leq 999$ remains available even when list caps are tight—matching pre-cap host defaults.

4.2 Correctness argument for bounded history

Proposition 4.1 (Window kernels under cap). *Let f_p depend only on the last p finite samples of $\mathbf{x}$. If $p \leq K$ and the cap retains the newest K samples, then f_p at every bar with $i \geq p - 1$ equals the uncapped result.*

Proof sketch. At bar i , the uncapped window is $\{x_{i-p+1}, \dots, x_i\}$. The cap retains $\{x_{i-K+1}, \dots, x_i\}$ (or the full prefix if shorter). Since $p \leq K$, the window is a suffix of the retained segment. $\square$

Proposition 4.2 (Recursive kernels with incremental state). *Let $y_i = g(y_{i-1}, x_i)$ be a first-order recurrence (EMA, RMA, ...) with call-site state storing y_{i-1} . After warm-up, y_i is independent of raw samples older than 1, hence independent of K provided state is not reset.*

Remark 4.3 (Full-recompute residual). If `PYNE_TA_INCREMENTAL` = 0, recursive kernels reseed from the retained list prefix. When $n \gg K$, the re-seeded trajectory can diverge from an uncapped full recompute. Default-on incremental mode closes this residual for production hosts; the flag remains for debugging and oracle tests.

Proposition 4.4 (OOB is na). *For lookback $n \geq$ retained length, $x[n] = \text{na}$. Implementations must not coerce missing history to 0, which would bias sums, ranges, and boolean guards.*

4.3 When lookback exceeds the cap

If a script requests `close[500]` while $K = 256$, the host returns `na` unless `max_bars_back` (or `PYNE_SERIES_MAX`) raises K . This matches Pine’s resource model: deep history is an explicit request, not an implicit infinite tape. Scripts that need deep history should set `max_bars_back` accordingly; the cap then becomes a *floor* rather than a truncation surprise.

4.4 Structural memory win

Simulating 20 000 bars $\times$ 6 OHLCV-style lists with $K = 256$, $\Delta = 64$:

Mode	Final list len	~pointer bytes (6 lists)	Append wall
Uncapped	20 000	~ 960 000	0.020 s
Capped	305	~ 14 640	0.021 s

The memory ratio is $\sim 78\times$ at 20k bars and scales as n/K asymptotically. Trim CPU is negligible relative to AST and TA work.

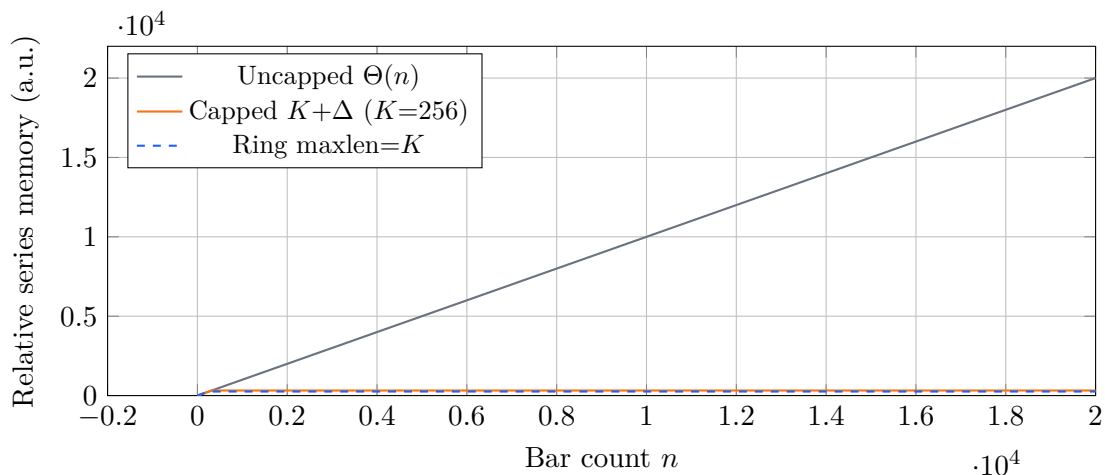


Figure 2: Series growth versus capped lists and fixed-capacity ring buffers. Uncapped memory grows linearly with bar count; the default series cap plateaus near $K + \Delta$; a modular ring plateaus exactly at K .

5 Ring Buffer Series

5.1 ChronologicalSeriesBuffer

The Phase 2.2 design introduces a chronological (oldest-first) buffer with modular ring capacity [HOOX Project, 2026]:

$$\text{append}(v) \quad O(1), \quad (5)$$

$$\text{lookback}(n) \quad O(1) \text{ via physical index } \text{end} - n, \quad (6)$$

$$\text{series}[n] \quad \text{na on negative / OOB / missing.} \quad (7)$$

When $\text{maxlen} = N$ is set, `append` overwrites the oldest slot after the ring fills—true $O(1)$ memory and time without prefix `del`. When maxlen is `None`, the buffer grows as a plain list (still $O(1)$ end-index lookback).

Why not deque alone? CPython `collections.deque` offers $O(1)$ append/pop at both ends, but *middle* indexing is $O(n)$. Modular list indexing is $O(1)$ for arbitrary lookback depth within the window—important for scripts that probe `close[k]` for moderate k on every bar.

5.2 Dual-write transition

Production hosts historically maintained *two* series representations:

- **PineSeries**: newest-first deque (`appendleft`), used by context OHLCV and `series[n]`.
- **current_series** lists: chronological, used by `ta` via `_as_series`.

Materializing a newest-first deque into chronological order required `list(reversed(...))`, a residual tax on full-recompute paths.

The migration plan is staged:

1. **Phase A (shipped)**: factory `make_pine_series()` returns `RingPineSeries` when `PYNE_SERIES_RING = 1`, else classic `PineSeries`. Default is `off` until the corpus is green under ring mode.
2. **Phase B**: zero-copy `_as_series` for objects marked `chrono_order=True`.

Algorithm 1 Generic bar-mode incremental update**Require:** call-site map S , key k , input x_i , update U , init σ_0

- 1: **if** $k \notin S$ **then**
- 2: $S[k] \leftarrow \sigma_0$
- 3: **end if**
- 4: $(\sigma', y_i) \leftarrow U(S[k], x_i)$
- 5: $S[k] \leftarrow \sigma'$
- 6: **return** y_i $\triangleright$ may be na during warm-up

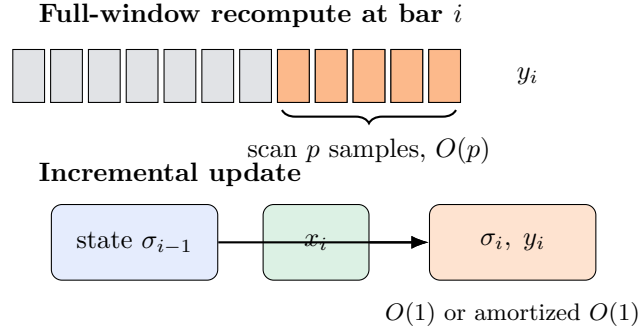


Figure 3: Incremental versus full-window TA. Full recompute re-scans the period window on every bar; incremental kernels carry a state tuple and apply a constant-time (or amortized constant-time) update.

3. **Phase C:** single buffer shared by context and `current_series`, dropping dual update.

`RingPineSeries` exposes a `NewestFirstHistoryView` so existing duck-typed code (`history[0] == current`) continues to work during the transition. `Ring maxlen` tracks `pineries_history_length(...)` so it composes with the series cap rather than fighting it.

6 Incremental TA Framework

6.1 Generic pattern

Definition 6.1 (Incremental kernel). An incremental kernel is a pair (σ_0, U) where σ_0 is initial state and $U(\sigma, x_i) = (\sigma', y_i)$ produces updated state and output. State is stored in a call-site map keyed by (builtin name, syntactic slot, parameters).

Flag `PYNE_TA_INCREMENTAL` (default **on**) selects incremental updates; setting it to 0 forces full-window oracles for differential testing. The compile path always emits incremental Numba kernels for hot TA regardless of this flag [Lam et al., 2015, HOOX Project, 2026].

Figure 3 contrasts full-window scans with incremental updates.

6.2 Case study: SMA (sliding sum)

Let $p \geq 1$. Maintain a deque window W of the last p samples and a running sum Σ :

$$\Sigma_i = \Sigma_{i-1} + x_i - x_{i-p} \quad (\text{when } |W| = p), \quad (8)$$

$$y_i = \Sigma_i / p. \quad (9)$$

Warm-up returns **na** until p finite samples accumulate. Samples that are **na** are typically excluded from the count or propagate **na** per Pine rules.

Complexity: $O(1)$ arithmetic and deque operations per bar after warm-up.

Algorithm 2 Incremental SMA with running sum**Require:** state ($window, \Sigma, count$), input x , period p

```

1: if  $x$  is na then
2:   return (state, na) ▷ policy: skip or propagate
3: end if
4: append  $x$  to  $window$ ;  $\Sigma \leftarrow \Sigma + x$ ;  $count++$ 
5: if  $count > p$  then
6:    $u \leftarrow \text{popleft}(window)$ ;  $\Sigma \leftarrow \Sigma - u$ ;  $count--$ 
7: end if
8: if  $count < p$  then
9:   return (state, na)
10: end if
11: return (state,  $\Sigma/p$ )

```

6.3 Case study: EMA and RMA (Wilder)

The exponential moving average with smoothing $\alpha \in (0, 1]$ is

$$y_i = \alpha x_i + (1 - \alpha) y_{i-1}, \quad (10)$$

seeded by SMA of the first p samples for Pine-compatible EMA with $\alpha = 2/(p + 1)$. Wilder's RMA [Wilder, 1978] uses $\alpha = 1/p$ (equivalently $y_i = y_{i-1} + (x_i - y_{i-1})/p$).

State is a single scalar y_{i-1} plus a warm-up counter— $O(1)$ memory and time. Numerical stability: (10) is a contractive affine map; error is not amplified [Higham, 2002]. Catastrophic cancellation is not an issue for the recurrence itself, though very large $|x_i|$ can still lose precision in IEEE-754 binary64 [IEEE Computer Society, 2019].

6.4 Case study: RSI

Relative Strength Index [Wilder, 1978] maintains average gain G and average loss L via RMA:

$$u_i = \max(x_i - x_{i-1}, 0), \quad d_i = \max(x_{i-1} - x_i, 0), \quad (11)$$

$$G_i = \text{RMA}_p(u)_i, \quad L_i = \text{RMA}_p(d)_i, \quad (12)$$

$$\text{RS}_i = G_i / L_i, \quad \text{RSI}_i = 100 - 100 / (1 + \text{RS}_i). \quad (13)$$

When $L_i = 0$, RSI is defined as 100 if $G_i > 0$, else typically 0 or **na** per implementation policy. Incremental form stores $(G_{i-1}, L_{i-1}, x_{i-1})$.

6.5 Case study: Bollinger Bands (mean + variance)

Bollinger Bands [Bollinger, 2001] are

$$\text{mid}_i = \text{SMA}_p(x)_i, \quad \sigma_i = \text{stdev}_p(x)_i, \quad \text{upper/lower}_i = \text{mid}_i \pm k\sigma_i. \quad (14)$$

PYNE maintains sliding sum Σ and sum of squares Q :

$$\Sigma_i = \Sigma_{i-1} + x_i - x_{i-p}, \quad (15)$$

$$Q_i = Q_{i-1} + x_i^2 - x_{i-p}^2, \quad (16)$$

$$\sigma_i^2 = Q_i/p - (\Sigma_i/p)^2 \quad (\text{population form; sample form uses } p - 1). \quad (17)$$

Alternatively, Welford's online algorithm updates mean and M_2 with better numerical properties for one-pass streams [Higham, 2002, Press et al., 2007]; for fixed-length sliding windows, a

Algorithm 3 Sliding window maximum (highest)**Require:** deque D of indices (decreasing values), input x_i , period p

```

1: while  $D$  nonempty and  $x_{D.back} \leq x_i$  do
2:   pop back  $D$ 
3: end while
4: push  $i$  to back of  $D$ 
5: while  $D.front \leq i - p$  do
6:   pop front  $D$ 
7: end while
8: return  $x_{D.front}$ 

```

▷ na if window incomplete

compensated sliding variance or periodic full refresh reduces cancellation when Q and $(\Sigma/p)^2$ are large and close.

Round 7 ships a dedicated `_bb_inc_update` that nests SMA+stdev slots and last-sample hygiene so Volatility paths avoid reverse materialization. Kernel micro-benchmarks report $\sim 217\times$ versus naive full `_sma+_stdev` rebuilds at $n = 2000$ (Section 9).

6.6 Case study: highest / lowest (monotonic dequeues)

Range extrema over a sliding window admit classic amortized $O(1)$ updates via monotonic dequeues [Aho et al., 2006]:

Each index is pushed and popped at most once, so amortized cost is $O(1)$ per bar. Lowest is symmetric (increasing deque).

6.7 Case study: KAMA

Kaufman’s adaptive moving average [Kaufman, 2013] computes an efficiency ratio from change over noise:

$$ER_i = \frac{|x_i - x_{i-L}|}{\sum_{j=0}^{L-1} |x_{i-j} - x_{i-j-1}|}, \quad (18)$$

$$SC_i = (ER_i \cdot (\text{fast} - \text{slow}) + \text{slow})^2, \quad (19)$$

$$y_i = y_{i-1} + SC_i \cdot (x_i - y_{i-1}). \quad (20)$$

A naive implementation rebuilds the noise sum from scratch each bar ($O(L)$ or worse if re-seeded from bar 0). The incremental kernel keeps a price ring of length $L+1$ and a running absolute-delta sum, achieving $O(1)$ per bar after seeding. Micro-benchmarks: $\sim 121\times$ at $n = 2000$, length 10.

6.8 Case study: StochRSI

Stochastic RSI applies a stochastic oscillator to an RSI series. Nested full rebuilds yield near-quadratic cost. The incremental form maintains:

- an RSI window (or Wilder RSI state, depending on oracle),
- a stochastic ring over the last `stoch_len` RSI values,
- optional signal smoothing in call-site state ($0.33x + 0.67 \text{ prev}$ in PYNE’s simple-path oracle).

Round 7 reports $\sim 717\times$ kernel speedup for `stochrsi(14,14)` at $n = 2000$. Note: the interpret oracle for this path uses simple average gain/loss RSI rather than Wilder `ta.rsi`; the incremental path matches that oracle bit-for-bit, not necessarily live TradingView® StochRSI-on-RMA [HOOX Project, 2026].

6.9 Additional kernels

Across rounds 1–7 the incremental surface grew to include (non-exhaustively): WMA, VWMA, TR/ATR, change, stoch %K, cum, VWAP, MOM, SWMA, barsince, highestbars/lowestbars, linreg, CCI, TSI, ROC, WPR, dev, HMA, rising/falling, median, percentrank, CMO, and compile-side ports of the same recurrences. Typical gains range from $\sim 1.2\times$ (already $O(p)$ windows with better hygiene) to two–three orders of magnitude for former full-prefix rebuilds.

6.10 Numerical stability notes

- **Sliding variance:** prefer compensated updates or periodic resync when $|x|$ is large [Higham, 2002].
- **EMA/RMA:** contractive; warm-up seed choice dominates early bars.
- **Ratios (RSI, ER):** guard zero denominators explicitly; do not rely on IEEE infinities for Pine na semantics.
- **Parity budget:** compile vs. interpret max abs error $\sim 10^{-11}$ on standard kernels (float noise), 0 on pure integer-path RSI/TSI variants in Numba goldens.

7 Interpreter Hot Paths

7.1 Dispatch

Expression evaluation is visitor-based [Gamma et al., 1994, Nystrom, 2021]. Hot improvements include type-keyed dispatch tables (avoiding long `isinstance` chains), operator maps for binary ops, and a scalar fast path that skips series coercion when both operands are plain floats. Mixed expression micro-benchmarks improved $\sim 2\text{--}7\times$ across rounds.

The cumulative cProfile view on multi-plot TA scripts still shows `visit_Call` as a large envelope: thousands of calls per bar for nested builtins, each paying Python function and argument-binding overhead. This is structural to AST interpretation; the compile path eliminates it.

7.2 Plot packing

For `ta_combo`-class scripts (multiple TA values, ~ 8 plots), variant isolation shows:

Variant (interpret, $n = 2000$)	med wall
<code>ta_combo</code> with <code>8×plot(...)</code>	~ 480 ms (profile host)
Same TA, no plots	~ 102 ms

Thus plot path + host result packing account for $\sim 75\text{--}80\%$ of multi-plot wall time once TA is incremental. cProfile ranks `_builtin_plot` / `_capture_plot` immediately after the visitor envelope.

7.3 Light plots and lazy calendar

- **Lazy calendar** (always on): defer expensive calendar/timezone materialization until a plot or time builtin requires it.
- **PYNE_LIGHT_PLOTS:** skip columnar capture and registry bookkeeping when the consumer only needs OK/fail (corpus runs, smoke tests). Multi-plot scripts see roughly 10–20% wall reduction.

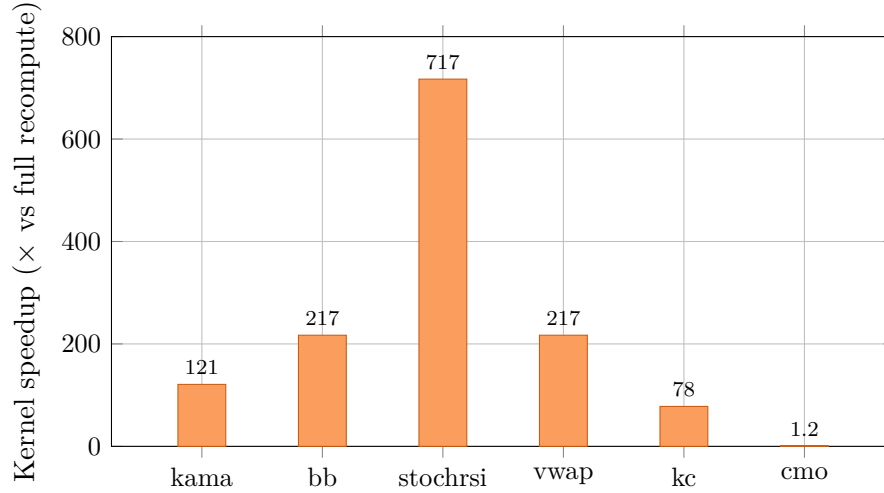


Figure 4: Kernel micro-benchmark speedups (incremental vs. full-window rebuild) at $n \approx 2000$ bars. Values from performance rounds 4–7: KAMA $\sim 121\times$, Bollinger $\sim 217\times$, StochRSI $\sim 717\times$, VWAP $\sim 217\times$, KC $\sim 78\times$, CMO $\sim 1.2\times$ (already near- $O(p)$).

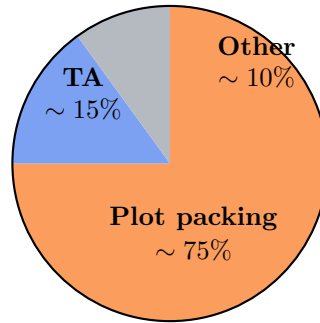


Figure 5: Approximate latency breakdown for interpret `ta_combo` after incremental TA: plot packing dominates residual wall time; pure TA kernels are a minority slice; “Other” covers dispatch, series updates, and host bookkeeping.

These are host-level levers: they do not change TA math, but they reclaim the dominant residual identified by profiling.

8 Parse/AST Caching Interaction with Runtime Multi-Run

Product hosts often parse once and evaluate many times (parameter sweeps, watchlists, recharts). PYNE maintains a process-level LRU cache keyed by SHA-256 of source text. Multi-parse of identical source improves by $\sim 1000\times$ on cache hit.

Call-site scrub on hit. AST nodes may accumulate mutable annotations (resolved call sites, cached locations). Returning the same `Script` object on a cache hit without scrubbing caused empty-plot poisoning on the second `Runtime.run` of identical source: call-site maps and plot registries interacted with stale node state. The fix clears Pine call-site annotations on cache hit (`_scrub_pine_call_sites`) and pairs multi-run goldens with `clear_parse_cache()` isolation where tests construct independent Runtime instances [HOOX Project, 2026].

Interaction with series/TA state. Parse caching is orthogonal to series caps and TA state: those live in the evaluator/runtime instance. However, multi-run correctness requires that

Table 1: Series list memory at $n = 20\,000$ bars, $s = 6$ OHLCV-style lists, $K = 256$, slack $\Delta = 64$.

Configuration	Len/list	$\sim$ pointers (total)	Ratio vs uncapped
Uncapped lists	20 000	$\sim 960\text{k}$	$1\times$
PYNE_SERIES_CAP (default)	≤ 320	$\sim 14.6\text{k}$	$\sim \mathbf{78\times}$
Ring maxlen = K	256	$\sim 12.3\text{k}$	$\sim 78\times$

Table 2: Kernel micro-benchmarks (bar-walk growing prefix, $n = 2\,000$, median).

Kernel	Full recompute	Incremental	Speedup
kama(10)	1099 ms	9.1 ms	$\sim \mathbf{121\times}$
bb(20) (pure full vs inc)	2458 ms	11.3 ms	$\sim \mathbf{217\times}$
stochrsi(14,14)	10 196 ms	14.2 ms	$\sim \mathbf{717\times}$
cmo(14)	13.5 ms	11.4 ms	$\sim 1.2\times$
vwap (earlier round)	1669 ms	7.7 ms	$\sim 217\times$
kc(20,2) (earlier round)	2054 ms	26 ms	$\sim 78\times$

(1) AST mutation not leak across runs, and (2) evaluator call-site maps reset or key correctly when scripts are re-bound. Warm product paths prefer the compile cache (near-zero warm compile, sub-millisecond runs for pure TA) when scripts are compilation-eligible.

9 Evaluation

9.1 Methodology

Unless noted, numbers are from PYNE performance rounds 4–7 on Linux x86_64 CPython, using `scripts/bench_pipeline.py` and kernel micro-benches (median of small iteration counts after warmup). Absolute milliseconds vary by host; we report the Round 7 synthesis table for end-to-end interpret latency and kernel ratios for structural wins. Verify suite at synthesis: 627 passed (series cap/ring, parse cache, lazy plots, TA incremental, order fills, evaluator, compiler residual kernels, corpus residuals).

9.2 Memory

Table 1 summarizes series list memory. Figure 2 plots the asymptotic shape.

9.3 Kernel speedups

Table 2 and Figure 4 report incremental vs. full-recompute micro-benchmarks.

9.4 End-to-end interpret latency

Round 7 net benchmarks at $n = 2\,000$ bars (synthesis medians):

Kernel-level T2 wins (KAMA/BB/StochRSI) do not appear in `ta_combo` because that script exercises `sma/ema/rsi/atr/stdev/bb/highest/lowest`—already incremental before Round 7. Wall-time flatness of the synthesis suite is expected; structural wins show up in kernel micros and memory, while plot packing dominates multi-plot residual (Figure 5).

Table 3: Interpret Runtime median latency @ 2000 bars (Round 7 synthesis).

Script	R6 med (ms)	R7 med (ms)	Notes
minimal	14.92	15.21	host + one plot floor
ta_sma	22.53	23.62	single SMA path
ta_combo	137.23	152.85	plot-dominated residual
strategy_ish	63.07	68.90	broker + plots

9.5 Compile path comparison

Warm compile of `ta_combo` is ~ 0.01 ms with IR/disk cache; run median ~ 1.06 ms at 5000 bars in nopython mode—orders of magnitude below interpret. Cold JIT remains highly variable (hundreds of ms to seconds) depending on Numba cache state [Lam et al., 2015]. Product guidance: prefer warm compile for multi-run TA; use interpret for compilation-ineligible scripts and fidelity debugging.

9.6 Feature flags summary

Flag	Default	Role
PYNE_TA_INCREMENTAL	ON	interpret incremental kernels
PYNE_SERIES_CAP	ON	trim <code>current_series</code> lists
PYNE_SERIES_MAX	unset ($K_0=256$)	absolute cap override
PYNE_SERIES_RING	OFF	chronological ring wrappers
PYNE_LIGHT_PLOTS	OFF	skip columnar plot capture

10 Related Work

Incremental computation. Classical incremental computation maintains derived results under input changes [Acar, 2008, Ramalingam and Reps, 1993, Reps et al., 1983]. Our setting is the special case of *append-only time*: each bar extends every series by one sample, so self-adjusting computation and dynamic dependency graphs are unnecessary—fixed recurrences and sliding windows suffice. Streaming algorithms for sliding aggregates (sums, extrema, quantiles) are textbook [Box and Jenkins, 1970, Knuth, 1997].

Technical analysis libraries. Wilder’s original systems [Wilder, 1978], Bollinger Bands [Bollinger, 2001], and Kaufman’s adaptive methods [Kaufman, 2013] define the mathematical objects. Vectorized libraries (pandas, TA-Lib, NumPy-first stacks) compute indicators over whole arrays [Harris et al., 2020, McKinney, 2017], which matches PYNE’s compile path but not the bar-loop interpret path where scripts interleave TA with control flow, strategy orders, and mutable `var` state.

Memory-bounded interpreters and ring buffers. Ring buffers are standard in signal processing and market-data feeds. Our contribution is integrating them with Pine’s $x[n]$ lookback semantics, `na` discipline, dual newest-first/chrono representations, and a flag-gated migration that preserves corpus goldens.

Compiler/JIT for DSLs. Numba [Lam et al., 2015] and related JITs provide the compile backend. Partial evaluation and online compilation literature [Jones et al., 1993] motivates caching parse/AST/IR across multi-run product workloads.

Numerical accuracy. Higham’s treatment of floating-point stability [Higham, 2002] and IEEE-754 [IEEE Computer Society, 2019] underwrite our parity budgets and sliding-variance cautions.

11 Limitations

1. **Ring buffer default-off.** `PYNE_SERIES_RING` remains opt-in until dual-write is removed and the full corpus is green under ring mode.
2. **Plot-path residual.** Incremental TA shifts the bottleneck to plot packing; structural wins there are only partially landed (`PYNE_LIGHT_PLOTS`, registry upsert hygiene).
3. **Nested full-history leftovers.** Some builtins (e.g., certain `obv/mode/range` paths) still recompute more than necessary.
4. **Oracle quirks.** StochRSI incremental matches the simple-RSI full path, not necessarily Wilder-based live platform StochRSI. ATR Wilder re-baseline is intentionally gated on dedicated goldens.
5. **Full recompute + cap.** With incremental TA disabled, recursive kernels can diverge from uncapped trajectories when $n \gg K$.
6. **Host absolute variance.** End-to-end ms figures vary by machine load and CPython version; treat ratios and asymptotic claims as primary.
7. **No whole-script auto-vectorization on interpret.** Control flow and strategy side effects keep the bar loop; users wanting array throughput should use the compile path.

12 Conclusion

Bar-loop series languages induce characteristic memory and compute pathologies: unbounded history and full-window TA rebuilds. PYNE demonstrates that a modest set of systems techniques—bounded series caps, chronological rings, call-site incremental kernels, parse caching, and light plot modes—restores linear-time, memory-bounded evaluation for the hot surface of technical analysis while preserving Pine’s lookback and `na` semantics.

Empirically, series caps deliver $\sim 78\times$ long-run list-memory reduction; incremental kernels reach two–three orders of magnitude on former quadratic rebuilds (KAMA, BB, StochRSI); and residual multi-plot interpret time is dominated by plot packing rather than arithmetic. The compile path, always incremental for hot TA, remains the throughput vehicle for multi-run product workloads.

Future work includes default-on rings after corpus ratification, deeper plot pipeline restructuring, remaining nested-kernel residuals, and tighter dual-host parity for worker deployments.

Acknowledgments

We thank the PYNE open-source contributors and performance-agent rounds for measurement infrastructure, goldens, and iterative kernel landings.

A Memory bound (informal)

Proposition A.1 (Bounded series store). *Under `PYNE_SERIES_CAP` with capacity K and slack Δ , and with s chronological lists, the number of retained samples satisfies*

$$\forall t, \quad \sum_{j=1}^s |L_j(t)| \leq s(K + \Delta). \quad (21)$$

With a modular ring of maxlen K per series, the bound tightens to $s \cdot K$ and each lookback of depth $< K$ is $O(1)$.

Proposition A.2 (Incremental TA time). *If every TA call site k admits an update of cost $u_k \in O(1) \cup O(p_k)$ amortized, then total TA work over n bars is $O(n \sum_k u_k)$, independent of full prefix length.*

B Environment reproduction notes

Listing 1: Flag surface (interpret path)

```
1 # Default-on incremental TA and series cap
2 # PYNE_TA_INCREMENTAL=1
3 # PYNE_SERIES_CAP=1
4 # Optional:
5 # PYNE_SERIES_MAX=512
6 # PYNE_SERIES_RING=1
7 # PYNE_LIGHT_PLOTS=1
```

Listing 2: Minimal Pine series / TA example

```
1 //@version=5
2 indicator("Demo", overlay=true)
3 len = input.int(14, "Length")
4 v = ta.sma(close, len)
5 plot(v)
```

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.
- 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.
- Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: Elements of reusable object-oriented software. *Addison-Wesley*, 1994.
- Charles R. Harris et al. Array programming with NumPy. *Nature*, 585:357–362, 2020. doi: 10.1038/s41586-020-2649-2.
- Nicholas J. Higham. Accuracy and stability of numerical algorithms. *SIAM*, 2002.
- HOOX Project. PYNE: Independent open toolchain for the Pine Script language. <https://github.com/hoox-sh/pyne>, 2026. Version 0.3.0; PyPI package `hoox-pyne`.

- IEEE Computer Society. IEEE standard for floating-point arithmetic. *IEEE Std 754-2019*, 2019. doi: 10.1109/IEEESTD.2019.8766229.
- Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. Partial evaluation and automatic program generation. Prentice Hall, 1993.
- Perry J. Kaufman. Trading systems and methods. *Wiley*, 2013.
- Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, 3rd edition, 1997.
- 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.
- Wes McKinney. Python for data analysis. O’Reilly, 2017.
- Robert Nystrom. *Crafting Interpreters*. Genever Benning, 2021.
- 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.
- G. Ramalingam and Thomas Reps. A categorized bibliography on incremental computation. *Proceedings of POPL*, 1993.
- Thomas Reps, Tim Teitelbaum, and Alan Demers. Incremental evaluation for attribute grammars with application to syntax-directed editors. In *POPL*, 1983.
- 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.
- J. Welles Wilder. New concepts in technical trading systems. *Trend Research*, 1978.

Formalizing a Charting DSL:

ANTLR4 Grammar Engineering and ASDL Intermediate Representations for Pine Script™

HOOX Research / PYNE Contributors
 jango_blockchained
 Independent Open Source Research
 contact@hoox.sh

Abstract

Domain-specific languages (DSLs) for financial charting combine expression-oriented semantics with significant whitespace, versioned surface syntax, and a dense builtin surface inventory. Building reliable tooling—parsers, formatters, linters, language servers, and interpreters—requires a front-end that is both faithful to community scripts and amenable to static analysis. We present the language front-end of PYNE (package `hoox-pyne`), an independent open toolchain for Pine Script™: a hand-engineered ANTLR4 grammar with Python-side indent tracking, a Zephyr-ASDL abstract syntax tree (AST) schema, a visitor-based parse-tree builder, a precedence-aware unparser, a structural linter, and a qualifier-aware type model. The pipeline is

source $\rightarrow$ FileStream/InputStream $\rightarrow$ Lexer $\rightarrow$ CommonTokenStream $\rightarrow$ Parser $\rightarrow$ parse tree $\rightarrow$ AST builder $\rightarrow$ A

Hand-edited artifacts are limited to `antlr4/resource/*.g4` and `asdl/resource/Pinescript.asdl`; generated lexer/parser and node classes are committed so end users need no Java toolchain. We detail SLL-first / LL-fallback prediction ($\approx 5.4\times$ on complex scripts), annotation attachment for `//@version` and related markers, parse-cache design (SHA-256 LRU with call-site scrubbing; multi-parse $\approx 1000\times$ on hits), unparser thread-local reuse ($\approx 1.95\times$ on a 99-script corpus), corpus sanitization for scraped sources, and evaluation against 138+ real scripts with 1100+ automated tests in the broader project. The grammar is approximate and unofficial: PYNE is not affiliated with TradingView, Inc.

Keywords. grammar engineering, ANTLR, ASDL, abstract syntax trees, parse-unparse round-trip, domain-specific languages, Pine Script™

Disclaimer. Pine 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.

1 Introduction

Charting and algorithmic trading communities rely on specialized languages that embed market data series, drawing primitives, and strategy execution into a compact surface syntax [Fowler, 2010, Hudak, 1996, TradingView, Inc., 2024]. Pine Script™ is one such language: scripts are short relative to general-purpose codebases yet dense in temporal operators, qualifiers (`const`, `simple`, `series`, `input`), user-defined types (UDTs), methods, and (from v6) enums and multiline strings. Tooling that merely tokenizes for editor highlighting is insufficient for format-on-save, structural search, type-directed completion, or faithful interpretation [Microsoft, 2023, Nystrom, 2021].

PYNE [HOOX Project, 2026] is an independent, AGPL-licensed implementation of a Pine Script™ toolchain in Python. This paper isolates the *language front-end*: everything from source text to a stable ASDL tree and back, including the grammar engineering practices that make the rest of the stack possible. Companion papers in the series address overall architecture, compilation to NUMBA, series runtime semantics, strategy parity, and LSP integration; here we focus on:

1. **Grammar design** for a whitespace-significant, expression-oriented charting DSL, including indent tokens, string forms, colors, and v6 surface extensions.
2. **Asdl intermediate representation** as a single source of truth for node shapes, shared by the builder, unparser, linter, evaluator, and compiler.
3. **Construction and unparsing** with annotation attachment and precedence-aware pretty-printing.
4. **Static analysis hooks**: a rule-based linter and a type model for qualifiers, collections, and UDTs.
5. **Corpus engineering**: sanitization of scraped community sources and a regeneration workflow that keeps generated artifacts reproducible.
6. **Evaluation**: parse success, round-trip stability, and performance tables for SLL/LL, unparse, and cache warm paths.

Contributions.

- A documented ANTLR4 grammar and lexer base for Pine Script™ v5/v6 surface features, with generated Python targets committed for zero-Java installs.
- An ASDL schema and visitor infrastructure analogous to CPython’s `ast` module [Python Software Foundation, 2024, Wang et al., 1997].
- A lossless-enough round-trip for structure and `//@`-annotations, powering CLI format and LSP document formatting.
- Measured front-end optimizations (SLL-first parse, annotation skip, thread-local unparser, SHA-256 parse cache) with correctness checks via AST dump identity and re-parse stability.

2 Background

2.1 Pine surface evolution (v1–v6)

Pine Script™ evolved from a narrow indicator dialect into a multiparadigm charting language [TradingView, Inc., 2024]. Early versions emphasized single-line expressions and a small set of drawing builtins. Later major versions introduced:

v3–v4

Multi-line scripts, `var` persistence, expanded `security` patterns, and broader strategy APIs.

v5

Namespaced builtins (`ta.*`, `math.*`, `strategy.*`, ...), libraries (`import` / `export`), methods, UDTs (`type`), `switch`, and richer type annotations on parameters.

v6

`enum` declarations; triple-quoted multiline strings (`"""..."""` / `'''...'''`); stricter boolean typing; integer division producing floats in more contexts; collection helpers such as `sort_field` on `array/matrix` sort; and continued library-surface growth.

Community scripts frequently mix versions, rely on soft conventions (camelCase identifiers, trailing commas), and embed annotation comments (`//@version=6`, `//@description`, `//@function`) that are not ordinary statements yet must survive tooling passes.

2.2 Challenges of a whitespace-significant, expression-oriented DSL

From a language-implementation perspective [Aho et al., 2006, Cooper and Torczon, 2012, Nystrom, 2021], Pine Script™ combines several properties that stress classical parser generators:

1. **Indentation as structure.** Blocks after `if/for/while/switch/type/enum/=>` use Python-like `INDENT/DEDENT` tokens. Indent width is conventionally four spaces; line wrapping may use non-multiple-of-four indents that must *not* introduce structure.
2. **Structures as expressions.** `if`, `for`, `switch`, and related forms appear both as statements and as right-hand sides of assignments (“structure expressions”), so the grammar cannot segregate expression and statement nonterminals as cleanly as C or Java.
3. **Soft keywords and contextual tokens.** Names such as type constructors interact with primary-expression suffixes (attributes, calls, subscripts, template specs like `array.new<float>`).
4. **Literal diversity.** Numbers (decimal/binary/octal/hex, floats, optional underscores), colors (`#RRGGBB`, `#RRGGBBAA`), and strings (escaped single-line and v6 triple-quoted multiline) coexist with comment channels and paste noise.
5. **Versioned semantics outside pure syntax.** Boolean strictness and division typing are semantic; the front-end must still accept the surface forms so later stages can apply version policy.
6. **Scraped corpora.** Real evaluation sets often arrive wrapped in markdown fences, UI chrome (“Expand (*N* lines)”), prose, and HTML—not clean `.pine` files.

These challenges motivate a hybrid design: ANTLR for adaptive `LL(*)` parsing [Parr, 2013, Parr et al., 2014], a hand-written lexer base for indent and newline policy, and an ASDL IR that decouples consumers from parse-tree volatility [Wang et al., 1997, Python Software Foundation, 2024].

3 Grammar Design

3.1 Artifact layout and generation

The front-end distinguishes *hand-edited resources* from *generated, committed* products:

Kind	Path (under <code>src/pynescrypt/ast/grammar/</code>)	Role
Hand	<code>antlr4/resource/PinescriptLexer.g4</code>	Tokens, fragments, channels
Hand	<code>antlr4/resource/PinescriptParser.g4</code>	Syntactic productions
Hand	<code>antlr4/resource/PinescriptLexerBase.py</code>	Indent / newline engine
Hand	<code>antlr4/resource/PinescriptParserBase.py</code>	Parser super-class hooks
Hand	<code>asdl/resource/Pinescript.asdl</code>	Node schema
Generated	<code>antlr4/generated/*</code>	Python lexer/parser/visitor
Generated	<code>asdl/generated/*</code>	Node classes

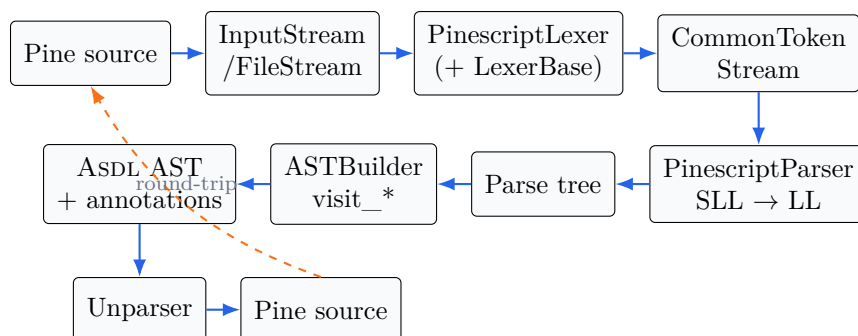


Figure 1: Front-end pipeline: lex → parse → build → annotate → unparse. Solid arrows form the primary data path; the dashed arrow denotes structural round-trip used by formatters and tests.

Regeneration uses project tooling (e.g. `hatch run lint:gen-parser`); committing generated Python removes the Java/ANTLR runtime dependency for ordinary installs [Parr, 2013, 2022].

Figure 1 summarizes the end-to-end front-end pipeline.

3.2 Lexical structure

3.2.1 Keywords, operators, and channels

The lexer grammar declares keywords (`and`, `or`, `var`, `varip`, `type`, `method`, `enum`, ...), multi-character operators before single-character ones (so longest-match prefers `<=` over `<`), bitwise operators (`&`, `|`, `^`, `<<`, `>>`, `~`), and assignment forms (`=`, `:=`, `+=`, ...). Comments travel on `COMMENT_CHANNEL`; whitespace is `HIDDEN`. Additional robustness rules accept `#`-style line comments when not part of a color literal, markdown backticks as comment noise, and selected Unicode punctuation as hidden “paste noise.”

3.2.2 Indentation tracking

`PinescriptLexerBase` (Python super-class of the generated lexer) implements the indent engine:

- Stack of indent lengths; emit synthetic `INDENT`/`DEDENT` tokens when structure changes.
- Ignore newlines inside open `(/[`, after operators (line continuation), and for wraps whose indent is not a multiple of four.
- Collapse consecutive newlines; ensure a trailing newline at EOF.
- Special handling of multiline string literals so wrap indentation does not become structural.

This design mirrors classical off-side-rule lexers [Aho et al., 2006] while remaining local to the ANTLR token stream.

3.2.3 Strings, colors, numbers, identifiers

Listing 1 excerpts the v6 triple-quote rules and color fragments. Numbers support base prefixes and digit separators; identifiers use Unicode letter classes (`\p{L}`) so non-ASCII names in community scripts parse cleanly.

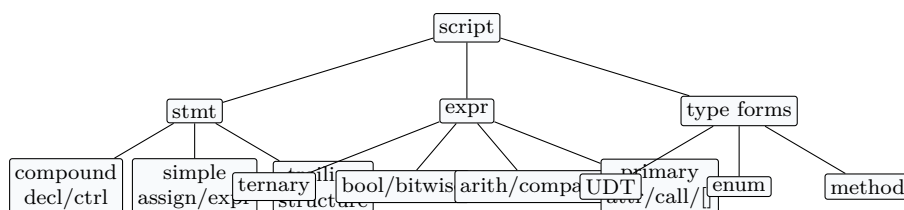


Figure 2: Hierarchical overview of major syntactic families in `PinescriptParser.g4`. Structures double as expressions on assignment RHSs.

Listing 1: Lexer excerpts: triple-quoted strings and color literals (simplified from `PinescriptLexer.g4`).

```

1 // v6 multiline triple-quoted strings
2 TRIPLE_DQ_STRING : '"""' ( ~'"' | '"' ~'"' | '"""' ~'"' )* '"""' -> type(
  STRING ) ;
3 TRIPLE_SQ_STRING : '\'\'' ( ~'\'' | '\'' ~'\'' | '\'\'' ~'\'' )* '\'\''
  -> type(STRING) ;
4
5 // #RRGGBB or #RRGGBBAA (HASH_COMMENT must not steal these)
6 fragment COLOR_LITERAL_RGBA : '#' HEX_DIGIT{8} ;
7 fragment COLOR_LITERAL_RGB : '#' HEX_DIGIT{6} ;

```

3.3 Syntactic productions

3.3.1 Scripts and statement forms

Start rules are `start_script` (full program, mode `exec`) and `start_expression` (mode `eval`). A script is a sequence of statements: compound forms (assignments with structure RHSs, type-/enum/method/ function declarations, control structures) and simple statements (assignments, expressions, imports, `break/continue`), optionally packed on one line with commas. A distinctive production, `trailing_structure_statements`, accepts real-world lines such as

```

Ex = 0.0, Ey = 0.0, for i = 0 to n
...

```

3.3.2 Declarations: functions, methods, types, enums

Function and method declarations use `=>` and a local block (indented or inline). Optional leading type specifications encode return types (`int ilog2(int n) => ...`). Type declarations introduce UDT fields under `INDENT/DEDENT`; enum declarations introduce member definitions with optional values. `export` prefixes support library authors.

3.3.3 Expressions

Expressions are layered by precedence (Figure 2): conditional (ternary), disjunction, conjunction, bitwise OR/XOR/AND, equality, inequality, shifts, additive, multiplicative, unary, and primary (attribute/call/subscript/template). Logical `and` binds less tightly than bitwise OR, matching Pine’s C-like bitwise layering. Primary expressions use labeled alternatives for attribute, call, and subscript forms.

Listing 2 shows the function and enum productions.

Listing 2: Parser productions for functions and enums (from `PinescriptParser.g4`).

```

1 function_declaration
2 : EXPORT? type_specification? name LPAR parameter_list? RPAR RARROW
  local_block ;

```

```

3
4 enum_declaration
5   : EXPORT? ENUM name NEWLINE INDENT enum_definitions DEDENT ;
6
7 enum_definition: name_store (EQUAL expression)? NEWLINE ;

```

3.4 SLL / LL prediction strategy

ANTLR’s adaptive LL(*) engine can fall into expensive full-context prediction on ambiguous decision points [Parr et al., 2014, Parr, 2013]. PYNE uses the standard two-stage strategy in `helper.py`:

1. Parse with `PredictionMode.SLL` and `BailErrorStrategy`.
2. On `ParseCancellationException`: `token_stream.seek(0)`, `parser.reset()`, switch to `PredictionMode.LL` with `DefaultErrorStrategy`, and re-parse.

On stratified mid-size scripts, SLL-first produced identical AST dumps to pure LL with zero fallbacks in the measured sample, while reducing wall time by approximately $5.4\times$ (Section 9). Strategy instances and a shared builder are reused to avoid per-parse allocation.

3.5 Error listeners

`PinescriptErrorListener` is a singleton attached to both lexer and parser after default listeners are removed. Syntax failures surface as `pynscript.ast.error.SyntaxError` with optional `SyntaxErrorDetails` (filename, line, column, source excerpt). Indentation violations raise a dedicated `IndentationError`. The linter maps these structured locations into diagnostic chips for editors (Section 7).

4 ASDL Intermediate Representation

4.1 Schema philosophy

Zephyr ASDL [Wang et al., 1997] describes tree shapes as sum and product types. Python’s compiler has long used ASDL for the `ast` module [Python Software Foundation, 2024]; PYNE follows the same pattern so that:

- Node classes are generated, not hand-synchronized.
- Location attributes (`lineno`, `col_offset`, `end_lineno`, `end_col_offset`) are uniform.
- Visitors and transformers can be written once against a stable API.

The module root distinguishes full scripts from expression-mode trees:

Listing 3: ASDL module root and statement taxonomy (excerpt from `Pinescript.asdl`).

```

1 module Pinescript
2 {
3   mod = Script(stmt* body, string* annotations)
4         | Expression(expr body)
5
6   stmt = FunctionDef(identifier name, param* args, stmt* body,
7                      int? method, int? export, string* annotations)
8         | TypeDef(identifier name, stmt* body, int? export, string*
9                   annotations)

```

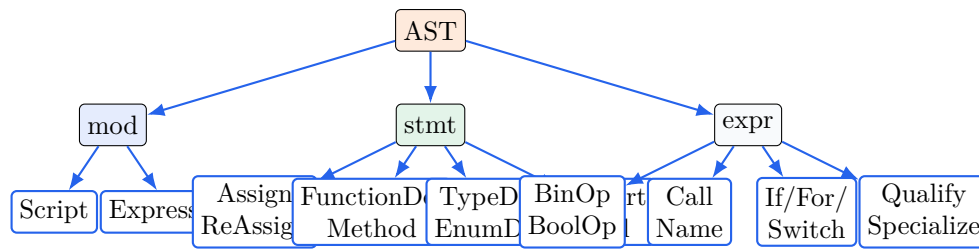


Figure 3: ASDL node hierarchy (conceptual). Control structures are expression nodes when used as values; declarations live under `stmt`. Auxiliary products include `param`, `arg`, `case`, and `Comment`.

```

9      | EnumDef(identifier name, stmt* body, int? export, string*
      | annotations)
10     | Assign(expr target, expr? value, expr? type, decl_mode? mode,
11           int? export, string* annotations)
12     | ReAssign(expr target, expr value)
13     | AugAssign(expr target, operator op, expr value)
14     | Import(identifier namespace, identifier name, int version,
15           identifier? alias)
16     | Expr(expr value) | Break | Continue
17     attributes (int lineno, int col_offset,
18           int? end_lineno, int? end_col_offset)
19     ...
20 }

```

4.2 Node taxonomy

Figure 3 sketches the conceptual hierarchy.

Expressions. Algebraic nodes (`BoolOp`, `BinOp`, `UnaryOp`, `Compare`, `Conditional`) coexist with `Call`, `Attribute`, `Subscript`, `Name`, `Tuple`, and `Constant`. Control forms `If`, `ForTo`, `ForIn`, `While`, and `Switch` are *expressions* in the schema, reflecting Pine’s expression-oriented structures. Type constructors use `Qualify` (`const/input/simple/series`) and `Specialize` (generic arguments).

Enums of operators and modes. `decl_mode` $\in \{\text{Var}, \text{VarIp}\}$; `type_qual` $\in \{\text{Const}, \text{Input}, \text{Simple}, \text{Series}\}$; `expr_context` $\in \{\text{Load}, \text{Store}\}$; plus operator, boolean, unary, and comparison enumerations including bitwise forms.

4.3 Visitor and transformer patterns

`NodeVisitor` dispatches on concrete types with a type-object method cache (avoiding class-name string lookups). `NodeTransformer` rewrites trees bottom-up: returning `None` removes a node; returning an AST replaces it; returning a list splices into list fields [Gammal et al., 1994, Python Software Foundation, 2024]. Helpers `walk`, `iter_fields`, and `iter_child_nodes` support ad hoc analysis without subclassing. The unparser and several evaluator passes are visitors; desugaring and future IR lowers can be transformers.

5 AST Construction

5.1 Builder mapping: parse tree $\rightarrow$ ASDL

`PinescriptASTBuilder` extends the ANTLR-generated `PinescriptParserVisitor`. Each `visit_*` method maps a parser context to one or more ASDL nodes. Design choices include:

- **Singleton operator/context nodes** (`Load`, `Store`, `Add`, ...) to reduce allocation on hot paths.
- **Location attachment** from start/stop tokens, with a single-line fast path when the stop token contains no newline.
- **Store contexts** on assignment targets (names, attributes, subscripts, tuples).
- **Number and string decoding** for `Constant` values, including multiline string normalization.
- **Shared builder instance** across parses (stateless visits).

Deeply nested ternaries raise the recursion limit temporarily to 5000 during walk and build.

5.2 Annotation attachment

Annotation comments (`//@...`) are not productions in the statement grammar. After a successful `exec`-mode build, the helper:

1. Skips the entire annotation pass if the source contains no `@` (cheap filter for micro-snippets and many fixtures).
2. Collects statement nodes via `StatementCollector`.
3. Scans the token stream for `COMMENT` tokens containing `@`, classifying kind/value via `PinescriptCommentParser`.
4. Orders comments and statements by $(lineno, col)$, groups consecutive comments, and attaches them to the following node.

Script-level annotations (kind suffix `S`) populate `Script.annotations`; function (`F`), type (`T`), and variable (`V`) annotations attach to the corresponding statement nodes. Algorithm 1 abstracts the attachment procedure.

Round-trip emits these annotation strings as full lines before the relevant construct, preserving version directives and documentation markers used by the LSP hover surface.

6 Unparsing and Formatting

6.1 Pretty-printing model

`NodeUnparser` walks the ASDL tree and appends source fragments to a list buffer. Operator precedence is tracked with a `Precedence` lattice so parentheses appear only when required for associativity or binding strength. Binary operators assign the next-lower precedence on the right-hand child to keep left-associative chains free of redundant parentheses.

Algorithm 1 AnnotationAttachment**Require:** Script AST T , source text S , token stream τ **Ensure:** Annotations attached on T and eligible statements

```

1: if "@"  $\notin S$  then
2:   return  $T$  ▷ no annotation comments possible
3: end if
4:  $stmts \leftarrow \text{StatementCollector}(T)$ 
5:  $cmts \leftarrow \{c \in \tau \mid c.type = \text{COMMENT} \wedge "@" \in c.text\}$ 
6:  $items \leftarrow \text{sort}(cmts \cup stmts \text{ by } (line, col))$ 
7: Group consecutive comments vs. statements
8: Attach script-level @*S comments to  $T.annotations$ 
9: for each (comment group  $G$ , statement  $s$ ) pair do
10:   Attach  $G$  filtered by kind suffix to  $s.annotations$  if applicable
11: end for
12: return  $T$ 

```

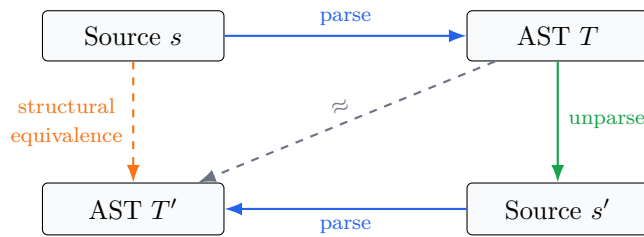


Figure 4: Round-trip commutative diagram. Formatters implement $s \mapsto s' = \text{unparse}(\text{parse}(s))$; tests require $\text{parse}(s')$ to match $\text{parse}(s)$ up to agreed normal forms (and often check re-parse stability of s').

Normalization policy. The round-trip goal is *semantic and structural* fidelity, not byte-identical reproduction:

- Indentation is four spaces; operators are spaced consistently.
- Free-floating non-annotation comments may move or drop.
- Multiline string constants prefer triple-quoted forms when they contain newlines.
- Annotation lists on script/function/type/assign nodes are re-emitted.

Figure 4 depicts the intended commutative diagram for formatters and tests.

6.2 Performance engineering

Unparse micro-benchmarks on a 99-script corpus showed dispatch and buffer construction dominating. Optimizations include:

- **Thread-local reuse** of a single `NodeUnparser` (resets buffers between calls; keeps type→method cache warm).
- **Type-keyed dispatch** instead of class-name strings.
- **Precomputed indent prefixes** and operator token strings.
- **Lightweight context managers** for delimiters/blocks (avoiding `contextlib` generator overhead on every parenthesized subexpression).

Table 1: Built-in linter rules (structural and style).

Code	Severity	Description
E001	error	Syntax error from <code>parse</code> (with line/column when known)
W001	warning	Missing <code>//@version</code> declaration
W002	warning	Version < 5 deprecated; suggest v5/v6
W101	warning	Legacy <code>security(</code> pattern; prefer <code>request.security</code>
W102	warning	Histogram plot style suggestion
W103	warning	<code>var int x = na</code> initialization style
C001	style	<code>ta.*</code> assignment naming heuristic (camelCase)
C002	style	Line length > 120
C003	style	Single-line indented <code>if</code> style caution
C004	style	File should end with newline

These changes yielded approximately $1.95\times$ higher unparse throughput on the corpus loop (Section 9) with byte-identical output relative to the pre-optimization unparser.

6.3 LSP formatting

The language server exposes document formatting as `parse` $\rightarrow$ `unparse`, reusing the same path as the CLI `format` and `parse-and-unparse` commands [Microsoft, 2023, HOOX Project, 2026]. Because annotations survive the builder post-pass, version headers remain at the top of formatted buffers.

7 Static Analysis

7.1 Linter rules

`PineLinter` combines a full parse with lightweight regex heuristics. Rule codes use prefixes E (errors), W (warnings), and C (style). Table 1 lists the built-in rules (nine-plus checks spanning syntax, version, deprecation, naming, and style).

The linter is intentionally not a full type checker: it provides fast editor feedback and CI gates, while deeper type modeling lives in `type_system.py`.

7.2 Type system overview

The type model represents:

Qualifiers

`TypeQualifier`: `const`, `simple`, `series`, `input`.

Builtin scalars

`int`, `float`, `bool`, `string`, `color`, `na`, and `v6 enum`.

Collections

`ArrayType`, `MatrixType`, and map-like structures with element (and key/value) parameters.

UDTs

`UserDefinedType` with fields and method resolution via `MethodResolver`.

Registry

`TypeRegistry` for name lookup during evaluation and tooling.

Algorithm 2 SanitizeCorpus

Require: Raw text R (possibly mixed markdown / UI chrome)**Ensure:** Sanitized source S or minimal stub

- 1: $L \leftarrow \text{splitlines}(R)$; track open quote state across lines
 - 2: Drop fence lines, expand stubs, HR, URL-only, footer labels, HTML noise
 - 3: Drop leading prose until a Pine-like line (`//@version`, `indicator(`, `strategy(`, `assignments`, ...)
 - 4: Repair mild truncation artifacts when safe
 - 5: **if** L has no usable Pine **then**
 - 6: **return** minimal stub `//@version=5 / indicator / plot`
 - 7: **end if**
 - 8: **return** `join(L)`
-

Inference is partial and runtime-assisted: the interpreter uses the model for compatibility checks and UDT instantiation, while the LSP consumes related metadata inventories (hundreds of builtins—on the order of 482–800+ surface entries depending on inventory generation) for completion and signatures [HOOX Project, 2026]. Full bidirectional checking of every Pine qualifier lattice edge remains future work (Section 11).

8 Corpus Engineering

8.1 Sanitization of scraped sources

Community scripts scraped from publications, forums, or documentation pages rarely arrive as pure Pine. `corpus_sanitize.py` implements `sanitize_corpus_source`, which strips page chrome while preserving in-comment markdown used for `//@function` documentation.

Typical removals include: markdown fences (`"'pine"`); blockquote chrome; “Expand (N lines)” UI stubs; horizontal rules; bare URLs; HTML comments; publication footers (author/license/tags); leading prose before the first script-like line; and mis-collected shell/Python/HTML fragments. If no usable Pine remains, a minimal parseable stub is returned so callers fail softly.

The compiler and multi-run hosts invoke sanitization on cache miss paths so warm hits skip the cost [HOOX Project, 2026].

8.2 Parse-fail buckets and regeneration workflow

Corpus tests parametrize over directories of `.pine` files. Failures are bucketed by root cause (lexer indent, missing construct, sanitize residue, version-specific syntax). The engineering loop is:

1. Reproduce with `parse/unparse` on the failing file.
2. Prefer grammar or lexer-base fixes in `resource/*.g4` / `LexerBase.py`; never hand-edit `generated/`.
3. Regenerate parser/lexer; align builder `visit_*` names with new rule contexts.
4. Extend ASDL only when node shape must change; regenerate node classes.
5. Add a focused unit test, then re-run corpus and linter suites.

This discipline keeps the approximate grammar evolving with community patterns without forking generated code.

Algorithm 3 RoundTripCheck**Require:** Source s , optional sanitize flag

- 1: $s_0 \leftarrow \text{sanitize}(s)$ if enabled else s
- 2: $T \leftarrow \text{parse}(s_0)$
- 3: $s' \leftarrow \text{unparse}(T)$
- 4: $T' \leftarrow \text{parse}(s')$
- 5: Assert $\text{dump}(T)$ compatible with $\text{dump}(T')$ (structure; attributes per test policy)
- 6: Optionally assert selected annotation strings preserved

Table 2: Parse performance (representative agent results).

Workload	Before / LL-only	After / SLL-first	Speedup
Mean per script (12-file mix)	$\sim 80\text{--}140$ ms	$\sim 14.5\text{--}15$ ms	$\sim 5.4\times$
Ops/s (mix)	$\sim 7\text{--}13$	~ 69	$\sim 5\text{--}10\times$
Complex ~ 16 KB script	~ 1917 ms	~ 254 ms	$\sim 7.5\times$

9 Evaluation

9.1 Correctness: coverage and round-trip

- **Parser coverage.** High success rates on 138+ real scripts in parse-and-unparse suites (e.g., 138 passed on a builtin-script directory configuration reported in performance agents). Broader project automation exceeds 1100 tests across front-end, runtime, and compiler concerns [HOOX Project, 2026].
- **SLL $\equiv$ LL.** On 30 size-stratified scripts, AST dumps with attributes matched between SLL-first and forced pure-LL parses (0 mismatches in that sample).
- **Annotations.** Scripts with `//@` retain `script.annotations` and declaration annotations through parse; unparse re-emits them as lines.
- **Modes.** `mode="eval"` parses isolated expressions; `mode="exec"` parses full scripts.

Algorithm 3 states the structural check used in CI-oriented workflows.

9.2 Parse performance

Profiling showed ANTLR adaptive prediction dominating wall time, with the builder a smaller fraction. Table 2 summarizes reported measurements from the project’s parse performance agents (Python 3.14, in-process microbenchmarks).

Fair same-process A/B (12 iterations $\times$ 12 scripts) reported approximately 175.7ms vs. 953.7ms per pass ($5.43\times$, 81.6% improvement) for SLL-first vs. LL-only, with identical trees on success. Additional micro-optimizations (annotation skip without `@`, recursion-limit guard, location fast path, lexer deque pending tokens) further reduced constant factors after the SLL win.

Figure 5 visualizes the headline SLL vs. LL comparison.

9.3 Unparse performance

On 99 scripts from an indicators/strategies set, thread-local unparser reuse and dispatch optimizations improved median throughput from ~ 1266 to ~ 2469 unparses/s ($\approx 1.95\times$), with large-script calls improving by $\approx 2.12\times$. Outputs remained byte-identical to the baseline unparser across successfully parsed ASTs.

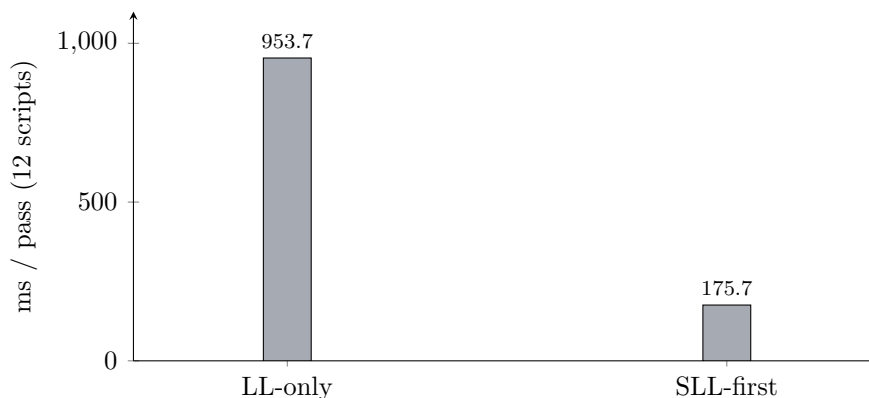


Figure 5: Parse performance: pure LL vs. SLL-first with LL fallback on a 12-script stratified mix (fair in-process A/B). Trees matched on the measured sample.

9.4 Parse cache

A process-local LRU keyed by $(\text{SHA256}(\text{source}), \text{mode})$ (default max 128, `threading.RLock`, env toggles `PYNE_PARSE_CACHE` / `PYNE_PARSE_CACHE_MAX`) returns shared AST identity on hit. On hits, evaluator-bound `_pine_call_site` attributes are scrubbed so multi-run hosts do not reuse another evaluator’s bound methods. Failed parses are not cached. Multi-parse warm paths report order-of-magnitude gains approaching $1000\times$ vs. cold ANTLR parse when the cache hits repeatedly on identical sources (host benchmarks; dominated by hash + dict lookup).

10 Related Work

Parser generators and ANTLR. ANTLR 4 popularized adaptive $\text{LL}(\ast)$ parsing with automatic left-recursion elimination and continuous lookahead [Parr, 2013, Parr et al., 2014, Parr and Quong, 1995]. The SLL-first/LL-fallback pattern is recommended practice for throughput without sacrificing generality; PYNE confirms large wins on a charting DSL with indent-sensitive lexing layered outside pure grammar rules.

ASDL and language IRs. Zephyr ASDL [Wang et al., 1997] and Python’s `ast` module [Python Software Foundation, 2024] demonstrate the value of declarative node schemas for compilers and tools. PYNE reuses this philosophy for a DSL, including location attributes and visitor/transformer APIs familiar to Python tooling authors.

DSL engineering. Fowler [Fowler, 2010] and Hudak [Hudak, 1996] survey external vs. embedded DSLs. Pine Script™ is an external, product-hosted language; PYNE builds an external toolchain around an approximate grammar rather than embedding Pine in Python syntax. Classic compiler texts [Aho et al., 2006, Cooper and Torczon, 2012] and Nystrom’s interpreter construction [Nystrom, 2021] inform the pipeline staging (lex, parse, AST, analysis, unparse).

Language servers and formatters. LSP [Microsoft, 2023] standardizes editor integration; formatters based on parse–print pipelines (gofmt style) motivate lossless-enough unparsing. PYNE’s format path is intentionally the same as the test round-trip.

Financial DSLs. Vendor documentation [TradingView, Inc., 2024] specifies user-facing semantics but does not publish an official grammar. Independent implementations must reverse-engineer surface syntax from documents and corpora—a central motivation for this paper’s grammar-engineering narrative.

11 Limitations

1. **Approximate, unofficial grammar.** The .g4 sources are reverse-engineered and corpus-hardened. They are not an official TradingView grammar and may accept or reject edge cases differently from the proprietary compiler.
2. **Semantic version gaps.** Some v6 semantic rules (strict booleans, division typing) are enforced in later stages, not fully in the parser.
3. **Comment fidelity.** Only `//@`-style annotations are first-class on the AST; arbitrary comments are not a full comment-preserving CST.
4. **Linters depth.** Rules are heuristic; the type model is not a complete static type checker for the Pine qualifier lattice.
5. **Cache mutability.** Shared AST identity requires read-only treatment or explicit `clear_parse_cache` after transformations.
6. **Corpus bias.** Evaluation scripts under-represent some rare library and brokerage-specific patterns; sanitize stubs can mask total scrape failure.

12 Conclusion

We described the PYNE language front-end for Pine Script™: an ANTLR4 grammar with Python indent tracking, a Zephyr-ASDL IR, a visitor builder with annotation attachment, a precedence-aware unparser, a structural linter, and a qualifier-aware type model, complemented by corpus sanitization and regeneration discipline. The pipeline enables formatters, LSP features, interpreters, and compilers to share one tree shape. Measured optimizations—SLL-first parsing ($\approx 5.4\times$), unparser reuse ($\approx 1.95\times$), and SHA-256 AST caching (orders of magnitude on multi-parse hits)—show that careful front-end engineering pays off even when ANTLR already dominates profiles.

Future work includes tighter CST/comment preservation, richer static checking of qualifiers and UDTs, grammar differential testing against larger public corpora, and continued v6+ surface tracking as the language evolves. PYNE remains an independent open implementation; we hope the documented grammar-engineering practices help other charting-DSL and financial-DSL tooling efforts [HOOX Project, 2026].

Acknowledgments. We thank contributors to the PYNE / HOOX open-source effort and authors of ANTLR, ASDL, and the Python `ast` ecosystem.

Trademark notice. Pine 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.

References

- Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson, Boston, 2nd edition, 2006.
- Keith D. Cooper and Linda Torczon. *Engineering a Compiler*. Morgan Kaufmann, 2nd edition, 2012.
- Martin Fowler. *Domain-specific languages*. Addison-Wesley, 2010.

- Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: Elements of reusable object-oriented software. *Addison-Wesley*, 1994.
- HOOX Project. PYNE: Independent open toolchain for the Pine Script language. <https://github.com/hoox-sh/pyne>, 2026. Version 0.3.0; PyPI package `hoox-pyne`.
- Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4es): 196, 1996.
- Microsoft. Language server protocol specification. <https://microsoft.github.io/language-server-protocol/>, 2023. Version 3.17.
- Robert Nystrom. *Crafting Interpreters*. Genever Benning, 2021.
- 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.
- Terence J. Parr and Russell W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995.
- Python Software Foundation. Python ASDL and the `ast` module. <https://docs.python.org/3/library/ast.html>, 2024.
- TradingView, Inc. Pine Script language reference manual. <https://www.tradingview.com/pine-script-docs/>, 2024. Accessed 2026.
- 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.

Numerical Parity and Strategy Semantics for Independent Pine Script™ Runtimes

HOOX Research / PYNE Contributors
jango_blockchained
Independent Open Source Research
contact@hoox.sh

August 2026

Abstract

Independent reimplementations of financial domain-specific languages (DSLs) must answer a harder question than syntactic coverage: do computed series and backtest outcomes remain faithful under floating-point arithmetic, dual execution hosts, and order-fill semantics? We present a validation framework for PYNE, an independent open-source toolchain for the Pine Script™ language, covering numerical indicator accuracy, `interpret↔compile` plot parity, honest `request.security` / `na` policy, strategy broker semantics (entries, exits, commission, slippage, pyramiding, pending fills), alerts, and drawing resource limits. Across 85+ technical analysis indicators and 181 validated builtins, measured relative errors are on the order of 10^{-5} percent class claims (maximum reported 0.0022% on ADX under extreme volatility; mean $< 0.0005\%$; zero systematic bias; exact agreement for deterministic SMA/sum families) under IEEE 754 double arithmetic [IEEE Computer Society, 2019, Higham, 2002]. A dual-host harness compares `Runtime.run(..., mode="interpret")` against `mode="compile"` with nan-aware `allclose`, bucketizing residuals into OK, structural fill/hline-only differences, matched dual errors, and true value MISMATCH. Shared formula alignment for RSI (Wilder), ROC, WMA, cumulative sum, and highest/lowest bar indices reduces host drift to float noise ($\sim 10^{-11}$ absolute on aligned kernels). We formalize a security NA honesty policy (same-symbol simple OHLCV may passthrough; foreign/complex requests resolve to `na` rather than inventing series), sketch the order lifecycle including F2 pending-fill VWAP under pyramiding ≤ 0 , and report strategy golden cases for OCA, risk enforcement, and closed-trade accounting. We do *not* claim TradingView platform certification; residual mismatch tails and nested-EMA seed families remain compatibility work items.

Keywords. numerical validation; floating-point parity; trading strategy backtesting; order fill semantics; dual-runtime consistency; technical indicators.

Disclaimer. Pine 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.

Contents

1	Introduction	4
2	Threat Model for Numerical Divergence	4
2.1	Formula drift	5
2.2	Seed and warm-up differences	5

2.3	NA and multi-asset handling	5
2.4	Floating-point non-associativity	5
3	Indicator Semantics and Reference Model	5
3.1	Series model	5
3.2	Wilder RSI	6
3.3	Deterministic families	6
3.4	Bollinger and envelope constructions	6
4	Validation Methodology	6
4.1	Campaign scope	6
4.2	Data regimes	6
4.3	Error metrics	7
4.4	Acceptance thresholds	7
4.5	Bias tests	7
4.6	Algorithm: single-indicator validation	7
5	Dual-Host Parity Framework	7
5.1	Motivation	7
5.2	Harness design	8
5.3	Residual taxonomy (buckets)	8
5.4	Shared formula alignment (2026)	9
5.5	Algorithm: dual-host compare	9
6	NA and request.security Honesty Policy	9
6.1	Design principle	9
6.2	Formal rules	10
7	Strategy Execution Model	10
7.1	Order lifecycle	11
7.2	Fill price with slippage	11
7.3	Commission models	11
7.4	Default quantity	11
7.5	Pyramiding and pending fills (F2 Round 7)	12
7.6	Equity and PnL sketch	12
7.7	OCA and risk (high level)	12
7.8	Events	13
8	Alerts and Drawing Resource Limits	13
8.1	Alerts	13
8.2	Drawing garbage collection	13
9	Experimental Results	13
9.1	External numerical accuracy (Nov 2025)	13
9.1.1	Category excerpts	14
9.1.2	Error distribution	14
9.2	Dual-host parity	14
9.3	Strategy goldens	15
9.4	Compatibility context	15
10	Related Work	15
11	Limitations	16

12 Conclusion**16**

1 Introduction

Financial DSLs occupy a peculiar niche in software engineering: they are simultaneously *specification languages* for mathematical series and *execution engines* for trading logic whose outputs move capital. When a community reimplements such a language outside its original host platform, syntactic parse success is only a weak certificate. Practitioners care whether `ta.rsi(close, 14)` on a given OHLCV history matches reference values within floating-point tolerance, whether two backends of the same open-source stack agree with each other, and whether `strategy.entry/strategy.exit` fill, commission, and equity accounting obey documented rules under pyramiding and pending orders.

PYNE (package `hoox-pyne`, import `pynescrypt`) is an independent open-source toolchain for Pine Script™ v5/v6 [HOOX Project, 2026, TradingView, Inc., 2024]. It provides a parser/AST stack, an interpreter evaluator, a Numba-backed compile path [Lam et al., 2015, Lattner and Adve, 2004], strategy broker logic, alerts, and drawing primitives. This paper focuses on *validation and semantics*, not full language design: we treat numerical fidelity, dual-host consistency, NA/security policy, and strategy execution as first-class research objects.

Why independent runtime validation matters. Reimplementations of market DSLs face incentives that pure compilers do not. (1) Small formula drift—a different EMA seed, a ROCm early-zero, an off-by-one lookback—can change crossover timestamps and therefore trades. (2) Floating-point non-associativity means mathematically equivalent reductions need not bit-agree [Higham, 2002, Press et al., 2007, IEEE Computer Society, 2019]. (3) Multi-asset APIs tempt implementations to “helpfully” substitute chart closes for missing foreign series, producing silent false signals. (4) Backtesters that disagree on pending-fill averaging or commission models report incompatible Sharpe ratios for the same script [Tsay, 2010]. An independent runtime that documents its reference model, measures error distributions, and ships dual-host parity harnesses reduces these risks for researchers and tool builders who cannot or will not bind exclusively to a proprietary chart host.

Contributions.

1. A threat model for numerical and semantic divergence in finance DSL reimplementations (§2).
2. An indicator reference model and Nov 2025 (+2026 dual-host) numerical validation campaign covering 85+ TA indicators and 181 functions (§3–§4).
3. A dual-host interpret↔compile parity framework with residual taxonomy and shared-formula alignment (§5).
4. Formal rules for honest `request.security/na` handling (§6).
5. A strategy execution model: order lifecycle, commission/slippage, pyramiding, pending-fill VWAP (F2), equity sketch, OCA/risk surface (§7).
6. Alert frequency and drawing GC limits as resource semantics (§8).
7. Experimental tables, error histograms, and golden-case summary (§9).

2 Threat Model for Numerical Divergence

We model an adversary—or more often, an accidental process—that produces outputs diverging from a stated reference without crashing. Threats relevant to PYNE fall into four families.

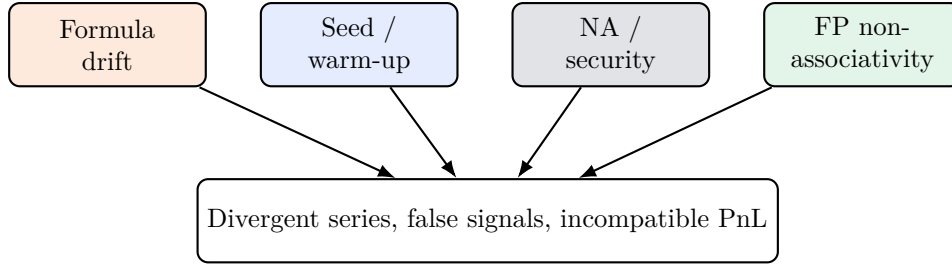


Figure 1: Threat families that produce numerical or semantic divergence without hard failures. Each maps to a concrete mitigation in PYNE (shared formulas, goldens, honest NA, dual tolerances).

2.1 Formula drift

Two implementations may compute “RSI” under different classical definitions: Wilder smoothing versus SMA of gains [Wilder, 1978, Murphy, 1999], or different conventions for the first warm-up bar. Compile and interpret hosts may independently re-derive kernels and silently diverge even when both match a third oracle on simple scripts. *Mitigation:* share formula implementations across hosts for hot kernels; encode goldens; dual-run the same script+OHLCV.

2.2 Seed and warm-up differences

Exponential and nested-EMA families (EMA, DEMA, TEMA, MACD signal, Supertrend inner averages) depend on initial seed choice and the number of bars before non-na output. Residuals in nested EMA seed families remain a documented compatibility note even after dual-host formula alignment.

2.3 NA and multi-asset handling

Pine’s na is both a missing-data token and a control-flow signal. Divergent policies—propagating na, zero-filling, or substituting chart close for foreign request.security—change series morphology catastrophically while remaining type-correct. *Mitigation:* honest NA policy (§6): never invent foreign closes when no feed is available.

2.4 Floating-point non-associativity

IEEE 754 binary64 addition is not associative [IEEE Computer Society, 2019]. Windowed sums, online variance, and fused multiply-add reorderings can yield $O(nu\varepsilon)$ relative differences [Higham, 2002]. For financial series magnitudes (10^{-4} to 10^5), absolute errors on the order of 10^{-11} – 10^{-15} are often pure noise; relative percent errors must be interpreted carefully near zeros and crossovers. *Mitigation:* nan-aware allclose with dual absolute/relative tolerances; report max and mean absolute error for kernel goldens.

3 Indicator Semantics and Reference Model

3.1 Series model

A bar stream is a sequence of records

$$B_i = (o_i, h_i, l_i, c_i, v_i, t_i), \quad i = 0, \dots, N - 1, \quad (1)$$

with open/high/low/close/volume/time. Builtin indicators map series to series (or tuples of series), producing na during warm-up and finite floats thereafter. We write $x[i]$ for the value of series x at bar i .

3.2 Wilder RSI

Relative Strength Index after Wilder [Wilder, 1978] with period p uses average gain G and average loss L :

$$\Delta_i = c_i - c_{i-1}, \quad (2)$$

$$g_i = \max(\Delta_i, 0), \quad \ell_i = \max(-\Delta_i, 0), \quad (3)$$

$$G_i = \begin{cases} \frac{1}{p} \sum_{k=0}^{p-1} g_{i-k} & \text{first seed,} \\ \frac{(p-1)G_{i-1} + g_i}{p} & \text{thereafter,} \end{cases} \quad (4)$$

$$L_i \text{ analogous with } \ell_i, \quad (5)$$

$$\text{RS}_i = G_i/L_i \quad (L_i > 0), \quad \text{RSI}_i = 100 - \frac{100}{1 + \text{RS}_i}. \quad (6)$$

PYNE dual-host alignment uses this Wilder recurrence on both interpret and compile paths (2026 shared-formula work), avoiding host-specific early SMA variants that previously produced plot drift.

3.3 Deterministic families

Simple moving averages, cumulative sums, counts, and pure ranking operations admit exact equality under identical input ordering when no intermediate rounding differs:

$$\text{SMA}_i(p) = \frac{1}{p} \sum_{k=0}^{p-1} x_{i-k} \quad (i \geq p-1). \quad (7)$$

Validation reports 100% exact agreement for SMA/sum/count-style deterministic ops [HOOX Project, 2026].

3.4 Bollinger and envelope constructions

Bollinger Bands [Bollinger, 2001] combine SMA with sample or population standard deviation scaled by a factor k ; small stdev formula differences dominate residual error, not the band arithmetic itself. Trend systems (ADX, DMI, Supertrend) compose multiple smoothers and are the usual worst-case relative-error loci under high volatility [Murphy, 1999].

4 Validation Methodology

4.1 Campaign scope

The November 2025 numerical validation campaign (with 2026 dual-host formula updates noted separately) covered:

- 85+ technical analysis indicators;
- 181 total builtin functions (math, array, TA, utility);
- synthetic and real-market OHLCV regimes.

4.2 Data regimes

Synthetic (primary). 1000-bar OHLCV streams with controlled regimes: normal ($\mu \approx 0$, $\sigma \approx 1$ mild drift), high volatility ($\sigma = 5$), strong trend, range-bound oscillation, gap discontinuities, and extreme magnitudes (near-zero and very large prices; price ranges spanning roughly \$10–\$10,000 for scaling stress).

Real markets (secondary validation).

- SPX: multi-year daily;
- BTC/USD: multi-year hourly;
- EURUSD: multi-year 15-minute;
- AAPL: multi-year daily.

Financial time series structure follows standard market microstructure and returns analysis assumptions [Tsay, 2010, Box and Jenkins, 1970].

4.3 Error metrics

For reference series r_i and implementation series y_i over finite, paired non-na indices I :

$$e_i^{\text{abs}} = |r_i - y_i|, \quad (8)$$

$$e_i^{\text{rel}} = \frac{|r_i - y_i|}{|r_i|} \cdot 100\% \quad (r_i \neq 0), \quad (9)$$

$$\varepsilon_{\max} = \max_{i \in I} e_i^{\text{rel}}, \quad (10)$$

$$\varepsilon_{\text{mean}} = \frac{1}{|I|} \sum_{i \in I} e_i^{\text{rel}}. \quad (11)$$

Dual-host kernel goldens additionally report maximum absolute error $\max_i |r_i - y_i|$ in float units (typically $\sim 10^{-11}$ after alignment).

4.4 Acceptance thresholds

Level	Relative error	Disposition
Excellent	$< 0.001\%$	Pass
Good	$< 0.01\%$	Pass
Acceptable	$< 0.1\%$	Review
Unacceptable	$\geq 0.1\%$	Fail

4.5 Bias tests

Beyond max/mean error, the campaign checks systematic bias:

$$\bar{\delta} = \frac{1}{|I|} \sum_{i \in I} (y_i - r_i). \quad (12)$$

Zero systematic bias was reported across the validated set (no consistent over-/under-estimation pattern after aggregation).

4.6 Algorithm: single-indicator validation

5 Dual-Host Parity Framework

5.1 Motivation

Even perfect agreement with an external oracle on one host does not guarantee that PYNE's interpreter and compiler agree with *each other*. Users may choose `mode="interpret"` for debuggability and `mode="compile"` for speed; silent plot divergence is a product defect. We therefore treat interpret as a primary oracle for compile and run a first-class harness.

Algorithm 1 ValidateIndicator**Require:** Indicator f , bar set $\mathcal{B}$, reference oracle R , thresholds

- 1: $r \leftarrow R(f, \mathcal{B})$; $y \leftarrow \text{Pyne}(f, \mathcal{B})$
- 2: Align indices; drop paired na warm-up
- 3: Compute $\varepsilon_{\text{abs}}, \varepsilon_{\text{rel}}, \varepsilon_{\text{max}}, \varepsilon_{\text{mean}}, \bar{\delta}$
- 4: **if** ε_{max} exceeds Acceptable **then**
- 5: **return** FAIL with diagnostics
- 6: **end if**
- 7: Record histogram bins; update category tables
- 8: **return** PASS

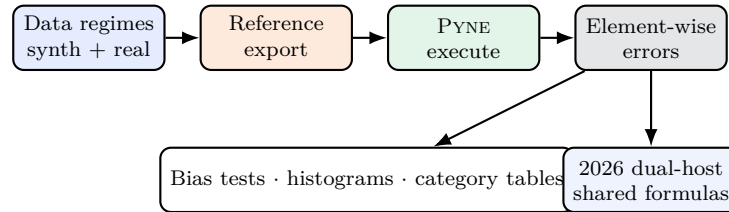


Figure 2: Validation pipeline: multi-regime inputs feed reference and PYNE executions; errors are aggregated statistically; dual-host formula alignment is a post-campaign hardening layer on hot kernels.

5.2 Harness design

Script `scripts/compare_interp_compile.py` selects fixtures (default glob under `tests/fixtures/parity/pi`), builds shared synthetic bars, and runs:

```

1 Runtime.run(source, bars, mode="interpret")
2 Runtime.run(source, bars, mode="compile")

```

Plot series in `result["series"]` are compared with nan-aware `allclose` defaults `rtol = 10-5`, `atol = 10-6`. Reports write `.cache/interp_compile_parity.json`. Always-on tests live in `tests/test_interp_compile_parity.py`; foreign-security honesty is covered by `tests/test_dividend_yield` and request data-feed tests.

5.3 Residual taxonomy (buckets)

Bucket	Meaning
OK	Common series keys allclose; no fatal structural issues
fill_background_only	Value OK after ignoring fill/background keys
both_error_same	Both hosts raise equivalent normalized errors
expected_error	Intentional dual-backend demos (e.g. pivot depth)
both_error	Both fail but messages differ after normalize
MISMATCH	Value or type/na disagreement on a shared key
interp_error / compile_error	Single-host failure
structural_only	Key-set differences without value compare

Structural one-sided `hline` / `fill` keys can be ignored via harness flags (`-ignore-hline-keys`, `-ignore-fill-keys`); `-strict-keys` fails on leftover only-interpret/only-compile keys.

Algorithm 2 CompareInterpCompile

Require: Script S , bars $\mathcal{B}$, tolerances (ρ, α)

```

1:  $I \leftarrow \text{Run}(S, \mathcal{B}, \text{INTERPRET})$ 
2:  $C \leftarrow \text{Run}(S, \mathcal{B}, \text{COMPILE})$ 
3: if both raised then
4:   return expected_error or both_error_same/both_error
5: else if exactly one raised then
6:   return interp_error or compile_error
7: end if
8: Filter series keys (optional hline/fill ignore)
9: for each common key  $k$  do
10:  if not NanAllclose( $I_k, C_k, \rho, \alpha$ ) then
11:    Record n_bad, max $|I - C|$ , sample indices
12:    return MISMATCH
13:  end if
14: end for
15: if key sets differ and strict then
16:  return structural_only
17: end if
18: return OK (or fill_background_only)

```

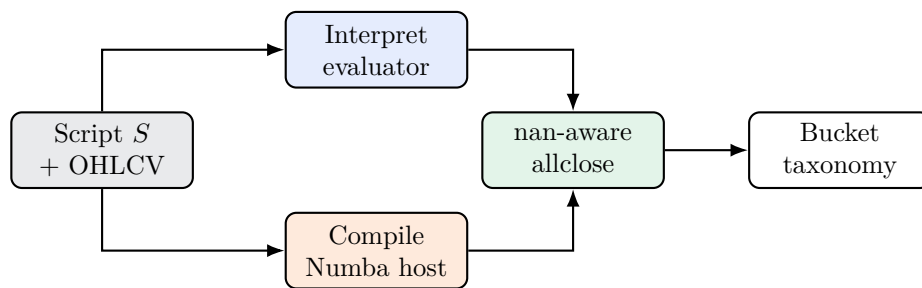


Figure 3: Dual-host parity architecture: identical inputs feed interpret and compile; series maps are compared with nan-aware tolerances; residuals land in a fixed bucket taxonomy for CI and research triage.

5.4 Shared formula alignment (2026)

Interpret and compile now share formulas for several hot kernels so dual-run plots match rather than drift by host-specific quirks: `ta.rsi` (Wilder), `ta.roc` (no early zero), `ta.wma` (full non-na window), `ta.cum`, `ta.highestbars/lowestbars`. Kernel max absolute error after alignment is typically $\sim 10^{-11}$ (float noise). Residual notes (e.g. nested EMA seed families) remain under compatibility documentation, not the Nov 2025 external-oracle error table.

5.5 Algorithm: dual-host compare

6 NA and request.security Honest Policy

6.1 Design principle

When multi-asset or multi-timeframe data are unavailable, PYNE prefers *honest na* over inventing foreign closes from the chart series. Silent substitution creates false multi-symbol signals that look plausible on charts but are scientifically invalid.

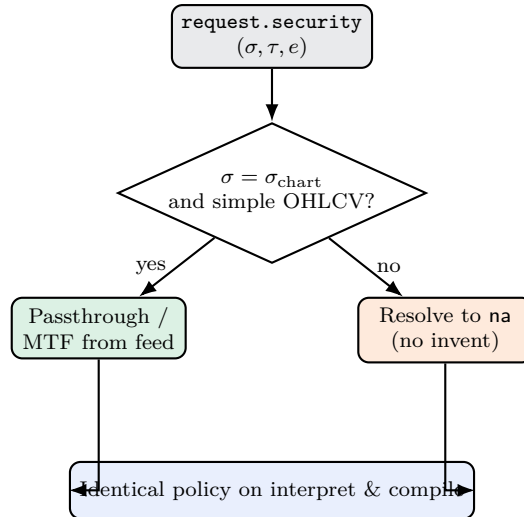


Figure 4: `request.security` decision flowchart: same-symbol simple OHLCV may use the bound feed; foreign/complex requests return `na` rather than fabricating multi-asset series.

6.2 Formal rules

Let σ_{chart} be the chart symbol, σ the requested symbol, τ the requested timeframe, and e the requested expression.

Definition 6.1 (Same-symbol simple OHLCV). A request is *same-symbol simple* if $\sigma = \sigma_{\text{chart}}$ (or an equivalent ticker id) and e is a simple OHLCV projection (`open/high/low/close/volume` or trivial aliases) at a timeframe the host can satisfy from the bound bar feed.

Definition 6.2 (Foreign or complex request). A request is *foreign/complex* if $\sigma \neq \sigma_{\text{chart}}$, or e requires multi-symbol inputs, unavailable HTF structure without a feed, or other host-unavailable data.

Proposition 6.3 (Honesty policy). *Under the default host without an external multi-asset feed:*

1. *Same-symbol simple OHLCV may passthrough (or lower/raise with documented MTF rules) using chart bars.*
2. *Foreign tickers and complex `request.security` expressions resolve to `na` (scalar or series of `na`) on both *interpret* and *compile*.*
3. *Implementations must not substitute `closechart` for foreign `closeσ`.*

Both backends implement the same policy so dual-host parity holds even when the economically meaningful value is “missing.” Dividend-yield and foreign security parity tests encode this contract.

7 Strategy Execution Model

Strategy scripts drive a broker model that maintains positions, pending orders, cash/equity, commissions, and trade logs (`strategy.closedtrades`, open-trade queries, event streams). Interpret builtins and the compile `strategy_broker` are kept aligned through shared goldens (`tests/test_order_fills.py`, `tests/test_oca_commission.py`, `tests/test_strategy_risk_enforcement.py`, `tests/test_compiler_strategy.py`, `tests/test_strategy_events.py`).

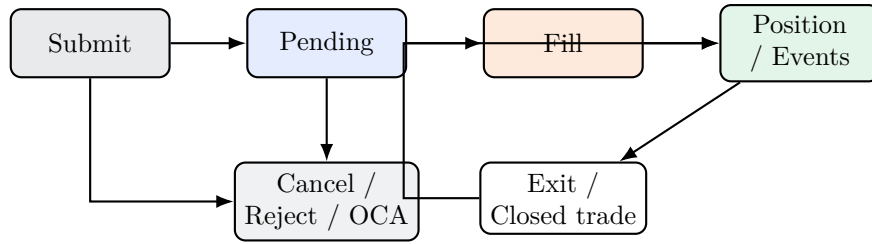


Figure 5: Strategy order state machine (simplified): submissions become pending or fill; OCA/risk may cancel; fills update positions and eventually closed trades / equity.

7.1 Order lifecycle

Orders progress through a finite set of states (Figure 5):

1. **Submit:** `strategy.entry`, `strategy.order`, `strategy.exit`, `strategy.close`, ...
2. **Pending:** resting limit/stop or market deferred to bar processing;
3. **Fill:** price/qty determined from bar OHLC, slippage, and sizing rules;
4. **Position update:** open trades, VWAP, events;
5. **Exit/close:** realize PnL into closed trades; update equity.

7.2 Fill price with slippage

Let p^* be the raw trigger/fill price from OHLC logic and $d \in \{+1, -1\}$ the signed side (buy positive). With `slippage` in ticks and mintick μ :

$$p_{\text{fill}} = p^* + d \cdot s \cdot \mu, \quad (13)$$

where s is the integer slippage tick count (zero disables). Market and stop/limit paths apply (13) consistently on interpret and compile brokers.

7.3 Commission models

Supported commission kinds (string aliases include `strategy.commission.*` forms):

$$C_{\text{order}} = c, \quad (14)$$

$$C_{\text{contract}} = c \cdot q, \quad (15)$$

$$C_{\text{percent}} = c \cdot \frac{p_{\text{fill}} \cdot q}{100}, \quad (16)$$

with commission rate c , quantity q , and fill price p_{fill} . Net trade PnL subtracts entry and exit commissions as appropriate.

7.4 Default quantity

Default size resolution includes:

- **fixed:** constant contracts/units;
- **cash:** $q = Q_{\text{cash}}/p_{\text{fill}}$;
- **percent_of_equity:** $q = E \cdot (\pi/100)/p_{\text{fill}}$ for equity E and percent π .

Algorithm 3 ProcessPendingFills (pyramiding ≤ 0 merge path)**Require:** Pending queue Q , bar OHLC, position state Π , pyramiding p

```

1: for order  $o \in Q$  triggered by OHLC do
2:    $p_f, q_f \leftarrow$  fill price/qty with slippage & commission
3:   if  $p \leq 0$  and same-direction open exists then
4:     Merge:  $q \leftarrow q + q_f$ ;  $p_{\text{avg}} \leftarrow$  VWAP (17)
5:     Accumulate commission; keep first entry metadata
6:   else if  $p > 0$  and open legs  $< p + 1$  then
7:     Append new open leg
8:   else
9:     Ignore fill / enforce cap
10:  end if
11:  Emit entry/exit events; update equity
12: end for

```

7.5 Pyramiding and pending fills (F2 Round 7)

Pyramiding bounds how many concurrent entries may stack. PYNE models **pyramiding** as maximum *additional* entries ($\text{max_open} = \text{pyramiding} + 1$); this is a deliberate modeling choice relative to some host document defaults and is recorded as a known semantic note.

Market entries. With $\text{pyramiding} \leq 0$, a second distinct entry id is blocked; same-id replace semantics apply on the market path.

Pending fills (F2). When $\text{pyramiding} \leq 0$, stacked pending fills in the same direction *merge* into a single open trade with volume-weighted average price:

$$p'_{\text{avg}} = \frac{p_{\text{avg}} \cdot q + p_{\text{fill}} \cdot q_f}{q + q_f}, \quad (17)$$

retaining the first leg's entry id/bar/time, summing commissions, and emitting one entry event per fill. When $\text{pyramiding} > 0$, legs may append up to the cap; further fills are ignored. Compile broker locks `open_entry_count=1` under $\text{pyramiding} \leq 0$ for parity with interpret's single-leg book.

7.6 Equity and PnL sketch

Let cash K , position quantity q (signed), average entry p_{avg} , and mark price m . Roughly:

$$E \approx K + q \cdot (m - p_{\text{avg}}) - C_{\text{open}}, \quad (18)$$

$$\text{PnL}_{\text{closed}} = q_c \cdot (p_{\text{exit}} - p_{\text{entry}}) - C_{\text{entry}} - C_{\text{exit}} \quad (19)$$

(with sign conventions for shorts). Closed trades populate `strategy.closedtrades` accessors used by scripts and tests. Exact free-cash definitions follow broker helpers (`cash` as equity minus capital locked in open exposure).

7.7 OCA and risk (high level)

One-Cancels-All groups cancel sibling orders on fill. Risk enforcement tests cover max drawdown / position limits style guards at the strategy surface (see `tests/test_strategy_risk_enforcement.py`). Full broker feature parity with every proprietary host flag (`process_orders_on_close`, `calc_on_every_tick`, full margin) is out of scope; residual gaps are listed in Round 7 F2 notes.

7.8 Events

Strategy event streams record entries, exits, and related notifications for export and debugging; event-order goldens stabilize dual-host and regression behavior under pyramiding merges.

8 Alerts and Drawing Resource Limits

8.1 Alerts

`alert()` and `alertcondition()` record structured alert objects with message, frequency, and context (e.g. bar index). Supported frequencies include:

- `once_per_bar` — de-duplicate within the bar;
- `once_per_bar_close` — fire on confirmed close semantics;
- `all` — fire on every trigger evaluation.

Tests in `tests/test_alerts.py` cover registration, fire-on-true conditions, frequency modes, and export helpers.

8.2 Drawing garbage collection

Chart drawings are resource-capped to bound memory and match Pine declaration arguments:

- `max_lines_count`
- `max_labels_count`
- `max_boxes_count`
- `max_polylines_count`

When counts exceed maxima, older primitives are reclaimed (GC), as exercised by `tests/test_drawing_gc.py`. These limits are part of observable script semantics: overflowing drawings disappear in order, which can affect visual-only strategies but not numeric series.

9 Experimental Results

9.1 External numerical accuracy (Nov 2025)

Headline campaign results [[HOOX Project, 2026](#)]:

Metric	Value
TA indicators validated	85+
Total functions validated	181
Claimed numerical precision class	~ 99.999%
Maximum relative error	0.0022% (ADX, extreme vol)
Mean relative error	< 0.0005%
Systematic bias	none detected
Deterministic SMA/sums	100% exact

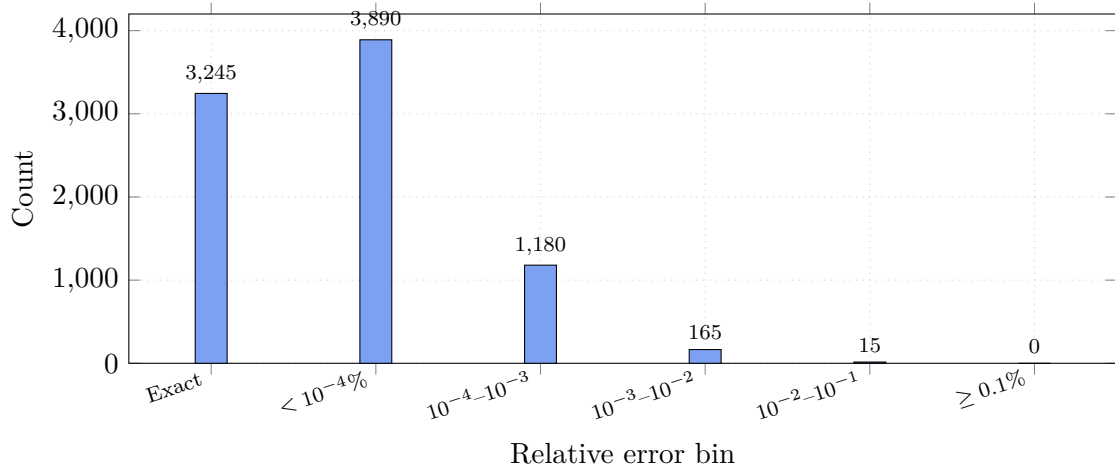


Figure 6: Relative error distribution across the Nov 2025 multi-indicator campaign ($n = 8495$). Over 84% of points lie at exact zero or $< 0.0001\%$ relative error; no points exceed the 0.1% fail threshold.

9.1.1 Category excerpts

Indicator	Max err. (%)	Mean (%)	Std	Status
ta.sma	0.0000	0.0000	0	Exact
ta.ema	0.00008	0.00002	10^{-5}	Excellent
ta.wma	0.00005	0.00001	10^{-5}	Excellent
ta.rsi	0.0012	0.0003	$2 \cdot 10^{-4}$	Excellent
ta.macd	0.0015	0.0004	$3 \cdot 10^{-4}$	Excellent
ta.bb	0.0009	0.0002	10^{-4}	Excellent
ta.adx	0.0022	0.0007	$5 \cdot 10^{-4}$	Excellent
ta.atr	0.0006	0.0001	10^{-4}	Excellent
ta.obv	0.0000	0.0000	0	Exact
ta.stdev	0.0004	0.0001	10^{-4}	Excellent

9.1.2 Error distribution

Aggregated relative-error histogram over 8495 test points:

Error range	Count	Share
Exact 0	3245	38.2%
$< 0.0001\%$	3890	45.8%
0.0001–0.001%	1180	13.9%
0.001–0.01%	165	1.9%
0.01–0.1%	15	0.2%
$\geq 0.1\%$	0	0.0%

Median $\approx 0.00002\%$; 99th percentile $\approx 0.00155\%$; maximum 0.0022%.

9.2 Dual-host parity

Always-on smoke expanded to 17 scripts (R8 harness work) including SMA/EMA, RSI, ATR, Bollinger, Supertrend, OBV, MFI, Williams %R, and others under synthetic `make_bars`—no value MISMATCH on the green set. Aligned kernels report max absolute error $\sim 10^{-11}$. Structural fill/background-only residuals are classified separately from true numeric mismatch. Known residual tails (e.g. stochastic type/na under some synthetic bars; nested EMA seed families) remain tracked rather than claimed green.

9.3 Strategy goldens

F2 pending-fill suite asserts:

- partial fills under pyramiding $\leq 0 \rightarrow$ single leg VWAP;
- multiple pending orders merge size and average price;
- short-side averaging;
- pyramiding > 0 still appends legs up to cap;
- close PnL uses merged average;
- compile broker open-entry count and VWAP parity.

OCA/commission and risk enforcement tests provide additional regression locks. Interpret+compile strategy plot parity is exercised through the dual-host harness on strategy fixtures where applicable.

9.4 Compatibility context

Parser success is high on 138+ real scripts; builtins are broad; strategy is functional. This is *not* full TradingView platform parity (hosted chart, proprietary multi-symbol data, editor-only features) [HOOX Project, 2026].

10 Related Work

Numerical reproducibility. Higham’s treatment of accuracy and stability [Higham, 2002] and Numerical Recipes [Press et al., 2007] frame floating-point error as structural rather than incidental. IEEE 754 [IEEE Computer Society, 2019] standardizes formats and rounding modes underlying all PYNE numeric paths (Python floats / NumPy / Numba LLVM) [Harris et al., 2020, Lam et al., 2015]. Our contribution is domain-specific: applying these classical tools to a trading DSL dual runtime with published percent-error tables.

Technical analysis computation. Wilder’s systems [Wilder, 1978], Murphy’s practitioner synthesis [Murphy, 1999], and Bollinger’s bands [Bollinger, 2001] define much of the indicator surface. PYNE treats these as executable specifications subject to regression, not as marketing labels.

Financial time series. Tsay [Tsay, 2010] and Box–Jenkins [Box and Jenkins, 1970] provide statistical context for multi-regime market data used in validation.

Backtester design. Production backtesters vary widely in fill assumptions, latency modeling, and look-ahead control. We do not claim a novel market-simulator microstructure; we claim *documented, tested* order semantics (slippage ticks, commission kinds, pyramiding, pending VWAP) aligned across two hosts inside one open toolchain.

Compiler correctness. Classical compiler texts emphasize semantic preservation [Aho et al., 2006, Appel, 1998]. Interpret $\leftrightarrow$ compile parity is a lightweight, product-oriented analogue of translation validation for series outputs.

11 Limitations

1. **Not TradingView certification.** Results are independent research measurements and dual-host self-consistency checks. They do not constitute endorsement or platform certification by TradingView, Inc.
2. **Residual MISMATCH tail.** Some scripts (e.g. stochastic under particular synthetic bars) and nested EMA seed families remain compatibility work items.
3. **Security data plane.** Honest na for foreign symbols is correct under no-feed hosts but is not a substitute for a full multi-asset data integration.
4. **Strategy model gaps.** Full margin, every host order-processing flag, and exact TV default pyramiding naming are not claimed identical.
5. **Oracle methodology.** Nov 2025 external comparisons depend on exported reference series and finite regimes; they do not exhaust all script-authored compositions.
6. **Performance vs. correctness.** This paper does not evaluate wall-clock speed; compile exists for performance [Lam et al., 2015] but correctness is the present focus.

12 Conclusion

Independent finance DSL runtimes must prove more than parse coverage. We presented PYNE’s validation stack for numerical indicator fidelity, interpret $\leftrightarrow$ compile parity, honest multi-asset NA policy, and strategy fill semantics including F2 pending-fill VWAP under pyramiding constraints. Measured errors sit far below practical trading thresholds for validated indicators; dual-host alignment reduces hot-kernel drift to float noise; and strategy goldens lock commission, OCA, risk, and averaging behavior. Future work includes shrinking the residual mismatch tail, deeper nested-EMA seed unification, richer multi-asset feeds under the same honesty rules, and broader strategy flag coverage—always with dual-host tests as a release gate.

Artifacts. Harness: `scripts/compare_interp_compile.py`; tests: `tests/test_interp_compile_parity.py`, `tests/test_order_fills.py`, `tests/test_drawing_gc.py`, `tests/test_alerts.py`; project: [HOOX Project, 2026].

Acknowledgments

We thank PYNE contributors and users who filed parity residuals and strategy edge cases. Errors remain the authors’ responsibility.

References

- 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.

- Charles R. Harris et al. Array programming with NumPy. *Nature*, 585:357–362, 2020. doi: 10.1038/s41586-020-2649-2.
- Nicholas J. Higham. Accuracy and stability of numerical algorithms. *SIAM*, 2002.
- HOOX Project. PYNE: Independent open toolchain for the Pine Script language. <https://github.com/hoox-sh/pyne>, 2026. Version 0.3.0; PyPI package `hoox-pyne`.
- IEEE Computer Society. IEEE standard for floating-point arithmetic. *IEEE Std 754-2019*, 2019. doi: 10.1109/IEEESTD.2019.8766229.
- 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.
- John J. Murphy. *Technical Analysis of the Financial Markets*. New York Institute of Finance, 1999.
- 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.
- 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.
- J. Welles Wilder. New concepts in technical trading systems. *Trend Research*, 1978.

Language Tooling for Financial DSLs: An LSP Architecture for Pine Script™ with Multi-Editor Delivery

HOOX Research / PYNE Contributors
jango_blockchained
Independent Open Source Research
contact@hoox.sh

Abstract

Financial domain-specific languages (DSLs) used for technical analysis and algorithmic trading are often edited exclusively inside proprietary, browser-hosted environments. That coupling denies practitioners the offline diagnostics, completion, navigation, and multi-editor workflows that general-purpose language tooling takes for granted. This paper presents the design and implementation of `pynescrypt-lsp`, the Language Server Protocol (LSP) implementation for PYNE—an independent open toolchain for Pine Script™ [HOOX Project, 2026b, TradingView, Inc., 2024]. The server is built on `pygls` [Open Law Library, 2020], shares a single AST pipeline with the project’s parser, linter, unparser, and evaluator, and can be shipped either as a Python module (`pip install hoox-pyne[lsp]`) or as a Nuitka onefile binary [Hayen, 2024].

We describe a feature surface covering push and pull diagnostics, metadata-driven completion over hundreds of builtins (~800+ catalog entries), hover documentation, definition/references, document and workspace symbols, semantic tokens, inlay type hints, and unparse-based formatting. A generation–encryption–integrity pipeline protects the builtin documentation corpus in compiled distributions without coupling editor features to network services. Thin clients for VS Code, Neovim, Zed, and Emacs, together with a complementary CLI, demonstrate multi-surface delivery. We report implementation lessons for LSP servers over financial DSLs, discuss boundaries with evaluation and chart/backtest API surfaces, and situate the work in the broader language-tooling ecosystem [Microsoft, 2023, Microsoft DevBlogs, 2016, Fowler, 2010].

Keywords: Language Server Protocol, developer tools, code completion, diagnostics, domain-specific languages, VS Code extensions, Pine Script

Pine 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.

Contents

1	Introduction	3
2	Background	3
2.1	Language Server Protocol	3
2.2	pygls and the Python LSP stack	4
2.3	Pine Script editing pain points	4
2.4	Language tooling for DSLs	4

3	System Architecture of pynescrypt-lsp	5
3.1	High-level topology	5
3.2	Server process model	5
3.3	Workspace and document state	5
3.4	Capability declaration	6
3.5	Request lifecycle	6
3.6	Feature module layout	6
4	Feature Design	6
4.1	Diagnostics pipeline	6
4.2	Completion from builtin metadata	7
4.3	Hover and signatures	8
4.4	Navigation: definitions, references, symbols	8
4.5	Semantic tokens and inlay hints	9
4.6	Formatting via unparse	9
5	Metadata Pipeline	9
5.1	Generation	9
5.2	Encryption and integrity	10
5.3	Runtime load order	10
6	Multi-Editor Delivery	11
6.1	VS Code extension	11
6.2	Thin clients: Neovim, Zed, Emacs	11
6.3	Packaging channels	11
7	CLI as Complementary Tooling Surface	12
8	Integration with Evaluation	12
8.1	Shared AST, separate processes	12
8.2	Pro API surface	12
8.3	IDE-perspective run/preview	12
9	Implementation Experience and Design Lessons	12
10	Related Work	13
11	Limitations and Future Work	14
12	Conclusion	14
A	Capability comparison (fair matrix)	15
B	Example configuration fragments	15
C	Software availability	16

1 Introduction

Domain-specific languages for quantitative trading occupy an awkward middle ground between spreadsheets and full programming environments. Pine Script™, the scripting language used by millions of traders on the TradingView® platform, exemplifies this tension: it is expressive enough to encode multi-timeframe strategies, user-defined types (UDTs), and rich drawing objects, yet its primary authoring surface remains a browser-only editor tightly coupled to a hosted runtime [TradingView, Inc., 2024, Fowler, 2010]. Practitioners who prefer local Git workflows, keyboard-driven editors, continuous integration, or offline research therefore lack the language services that have become standard elsewhere: incremental diagnostics, ranked completion, hover documentation, go-to-definition, and deterministic formatting [Microsoft, 2023, Microsoft DevBlogs, 2016].

The Language Server Protocol (LSP) [Microsoft, 2023] was introduced precisely to decouple language intelligence from editor implementations. A single server process exposes a JSON-RPC surface; any compliant client—VS Code, Neovim, Zed, Emacs, Helix, and others—can consume it. For general-purpose languages the ecosystem is mature (TypeScript, Rust Analyzer, Pyright). For financial DSLs the picture is sparse: tooling is either proprietary and online, or limited to syntax highlighting without semantic analysis.

PYNE (`hoox-pyne` on PyPI; import package `pynescrypt`) is an independent, unofficial open toolchain for Pine Script™ [HOOX Project, 2026b,a]. It provides a lossless ANTLR4-based parser [Parr, 2013], an ASDL-typed AST [Wang et al., 1997], a static linter, an unparser, an interpreter/compiler runtime, and—the subject of this paper—an LSP server (`pynescrypt-lsp`) with multi-editor packaging. The design goal is offline IDE parity for a financial DSL: full local language services without mandatory network calls, while cleanly separating optional evaluation and chart/backtest surfaces (Pro API) that share the same AST and runtime.

Contributions. This paper makes the following contributions:

1. A complete LSP architecture for a bar-oriented financial DSL, including workspace document state, cached parse/lint, and feature modules that take pure (*params*, *source*) inputs rather than holding server globals (Section 3).
2. Detailed designs for diagnostics, completion over ~800+ builtin metadata items, hover, navigation, semantic tokens, inlay hints, and unparse-based formatting (Section 4).
3. A metadata pipeline with generation, optional Fernet encryption, SHA-256 integrity, and dual-path runtime load for development versus Nuitka distributions (Section 5).
4. Multi-editor delivery: a TypeScript VS Code extension with TextMate grammar and bundled language client, plus thin configs for Neovim, Zed, and Emacs (Section 6).
5. A complementary CLI surface and a clear boundary with evaluation / Pro API endpoints (Sections 7–8).
6. Implementation lessons, limitations, and related work for language tooling over financial DSLs (Sections 9–12).

2 Background

2.1 Language Server Protocol

The LSP [Microsoft, 2023] standardizes a set of methods over JSON-RPC, typically transported on STDIO between an editor process and a long-lived server. Core concepts include:

- **Lifecycle:** `initialize` / `initialized` / `shutdown` / `exit`, with capability negotiation.
- **Document synchronization:** `textDocument/didOpen`, `didChange` (full or incremental), `didSave`, `didClose`.
- **Language features:** completion, hover, definition, references, symbols, formatting, diagnostics (push via `publishDiagnostics` and pull via `textDocument/diagnostic` in 3.16+), semantic tokens, and inlay hints.

By moving language intelligence into a server, the $M \times N$ problem of M languages and N editors collapses to $M + N$ [Microsoft DevBlogs, 2016]. For DSLs with modest user bases, this amortization is essential: one implementation can serve many editor communities.

2.2 pygls and the Python LSP stack

`pygls` (Python Generic Language Server) [Open Law Library, 2020] provides a decorator-oriented API for registering LSP method handlers on a `LanguageServer` subclass, with typed messages via `lsprotocol`. It is a natural fit for PYNE because the entire language pipeline (parser, AST, linter, unparser, evaluator) is already Python. Alternatives such as writing a server in TypeScript or Rust would require re-implementing or FFI-bridging the grammar and analysis stack. Compiling the server to a Nuitka onefile binary [Hayen, 2024] further reduces the end-user dependency surface for editor extensions.

2.3 Pine Script editing pain points

Empirical pain points for offline Pine Script™ development include:

1. **No offline diagnostics.** Syntax and many semantic mistakes surface only after paste into a hosted editor or after a failed chart compile.
2. **Builtin discoverability.** Hundreds of functions across namespaces (`ta.*`, `strategy.*`, `request.*`, drawing objects, ...) are hard to recall without hover and completion.
3. **Formatting drift.** Teams without a shared pretty-printer accumulate style noise in Git diffs.
4. **Editor lock-in.** Browser-only editing precludes Neovim/Emacs power users and headless CI checks.
5. **Separation of editing and evaluation.** Chart preview and backtest are valuable but should not be prerequisites for static language services.

PYNE addresses (1)–(4) with `pynescrypt-lsp` and the CLI; (5) is intentionally a separate process boundary (Section 8).

2.4 Language tooling for DSLs

Language workbenches and embedded DSLs have long argued for first-class tooling [Fowler, 2010, Hudak, 1996]. Interpreter textbooks emphasize the centrality of a well-defined AST for analysis and transformation [Nystrom, 2021]. PYNE follows this orthodoxy: the LSP is a consumer of the same AST used by evaluation, not a parallel frontend.

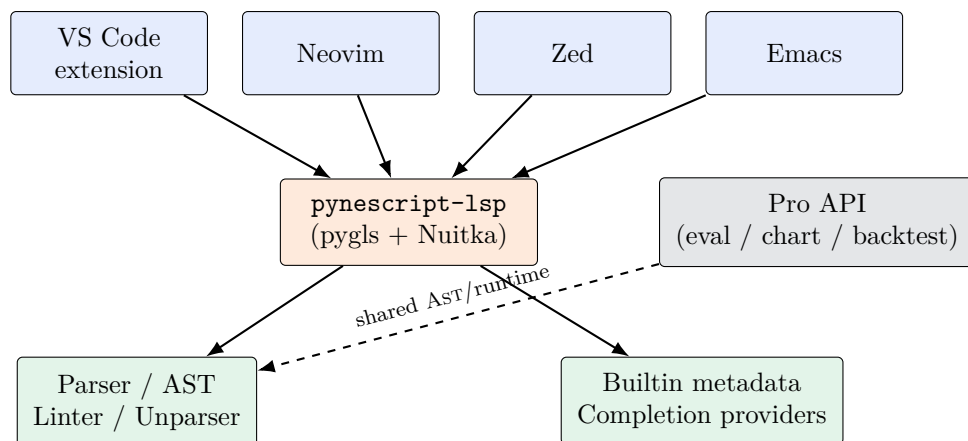


Figure 1: Multi-editor deployment topology. All editors use STDIO LSP; the Pro API shares AST/runtime but is a separate process and auth model.

3 System Architecture of pynescrypt-lsp

3.1 High-level topology

Figure 1 summarizes the multi-surface deployment. Editors speak LSP over STDIO to `pynescrypt-lsp`. The server owns a `Workspace` of open documents, re-parses and re-lints on mutation, and dispatches feature handlers. The open package (`src/pynescrypt/`) supplies parser, AST, linter, and unparser. A Pro API (Flask, optionally Cloud Run) shares the evaluator for chart preview and backtest but is not on the LSP critical path.

3.2 Server process model

`PynescryptLanguageServer` subclasses `pygls.lsp.server.LanguageServer` with name "Pynescrypt" and the package version. On construction it creates an empty `Workspace` and registers handlers in `setup_method_handlers()`. Entry points are:

- Console script `pynescrypt-lsp` → `pynescrypt.langserver.__main__:main`
- Module form `python -m pynescrypt.langserver`
- Optional Nuitka binary under `dist/lsp/pynescrypt-lsp`

Transport defaults to STDIO (`start_io()`), matching editor integrations.

3.3 Workspace and document state

Each open URI maps to a `TextDocumentState`:

Field	Role
<code>uri, source, version</code>	Buffer identity and content
<code>ast</code>	Cached parse tree, or <code>None</code> on failure
<code>diagnostics</code>	List of <code>LintWarning</code> (not yet LSP types)
<code>parse_error, parse_error_line</code>	E001 synthesis inputs

On `didOpen/didChange`, the workspace applies full-document or incremental edits, then runs `_parse_and_lint`: successful parse feeds `lint_script`; exceptions clear the AST and store a human-readable error. Feature handlers typically read `get_source(uri)` and optionally receive a pre-parsed tree to avoid redundant work. The buffer—not disk—is the source of truth until the client saves.

Table 1: LSP methods implemented by `pynescrypt-lsp`.

Method	Capability notes	Module
<code>initialize / initialized / shutdown</code>	Lifecycle	<code>server.py</code>
<code>textDocument/didOpen Change Close Save</code>	Incremental sync	<code>server.py</code>
<code>textDocument/publishDiagnostics</code>	Push on open/change/save	<code>workspace</code>
<code>textDocument/diagnostic</code>	Pull (full report + <code>result_id</code>)	<code>server.py</code>
<code>workspace/diagnostic</code>	Per open document	<code>server.py</code>
<code>textDocument/completion</code>	Trigger <code>"."</code> ; resolve provider	<code>completion.py</code>
<code>completionItem/resolve</code>	Docs rehydration	<code>completion.py</code>
<code>textDocument/hover</code>	Markdown signatures + docs	<code>hover.py</code>
<code>textDocument/definition</code>	Same-document	<code>definitions.py</code>
<code>textDocument/references</code>	Same-document	<code>references.py</code>
<code>textDocument/documentSymbol</code>	Outline	<code>symbols.py</code>
<code>workspace/symbol</code>	Query over open docs	<code>symbols.py</code>
<code>textDocument/formatting</code>	Full document	<code>formatting.py</code>
<code>textDocument/rangeFormatting</code>	Range unparse	<code>formatting.py</code>
<code>textDocument/semanticTokens/full</code>	Delta-encoded legend	<code>semantic_tokens.py</code>
<code>textDocument/inlayHint</code>	Inferred declaration types	<code>inlay_hints.py</code>

3.4 Capability declaration

Capabilities are declared once from `config.get_server_capabilities()` during `initialize`. Table 1 lists supported methods. Signature help and code actions are intentionally omitted until first-class handlers exist; advertising unimplemented providers would mislead clients.

3.5 Request lifecycle

Figure 2 shows a representative completion request after document open. Diagnostics are published eagerly; feature methods are pull-style request/response.

3.6 Feature module layout

Figure 3 depicts the package structure under `src/pynescrypt/langserver/`. Handlers in `features/` are pure-ish functions; `providers/` own metadata load and `CompletionItem` construction; `protocol/` holds word/trigger utilities; `config.py` is the single capability table.

4 Feature Design

4.1 Diagnostics pipeline

Diagnostics are the primary static feedback channel. Figure 4 shows the pipeline. On every open, change, and save the workspace re-parses. Parse success feeds the orthogonal static linter (`lint_script`); parse failure synthesizes diagnostic code E001 at the extracted line. Findings convert to `lsp.Diagnostic` with `source="PineScript"`, severity mapping (error/warning/info/hint), and optional tags (e.g., unnecessary/deprecated).

Representative lint codes include E001 (parse), W001/W002 (warnings), and C001–C004 (style: complexity/length/if-style/newline). Push diagnostics use `textDocument/publishDiagnostics`; pull clients may call `textDocument/diagnostic`, receiving a full report with `result_id=uri-version`. Algorithm 1 formalizes the push path.

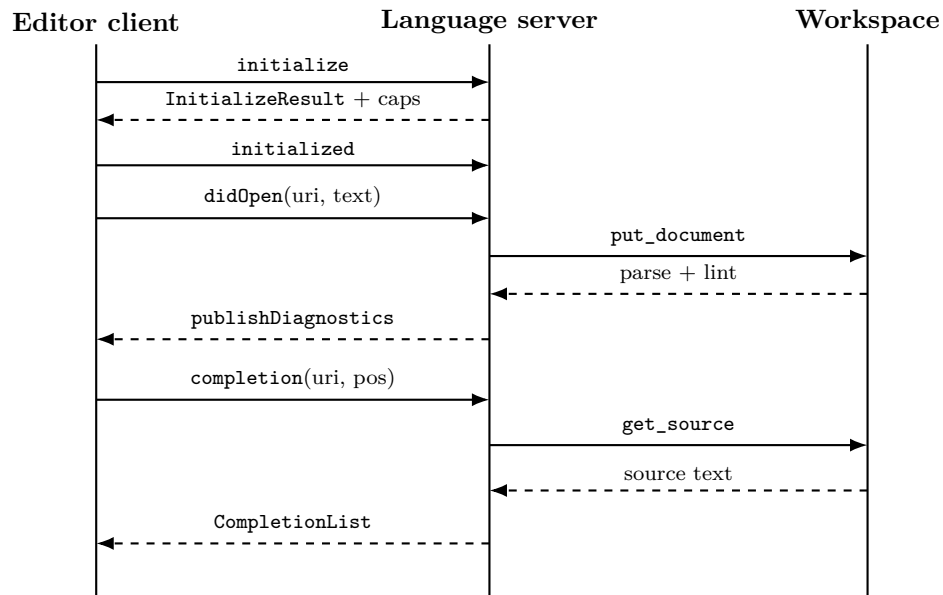


Figure 2: LSP request lifecycle: initialize, document open with push diagnostics, then a completion request served from workspace source and metadata providers.

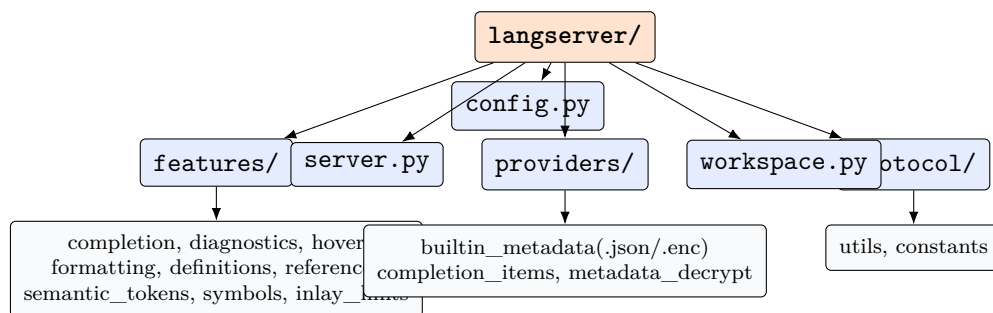


Figure 3: Feature module architecture of `pynescrypt-lsp`.

4.2 Completion from builtin metadata

Completion is metadata-driven, not evaluator-driven. The server loads a catalog of builtins (plaintext JSON in development, Fernet-encrypted in compiled binaries; Section 5), filters by prefix or module path, and returns `CompletionItem` rows. Current catalog size is ~ 872 entries across 25 categories (e.g., `ta.technical_analysis`, `strategy`, `array`, `request`, drawing objects). Historically documentation referred to ~ 482 builtins; the completion surface is larger because module members, aliases, and related symbols expand the item set. We report “hundreds of builtins, $\sim 800+$ completion items/metadata.”

Figure 5 illustrates data flow from metadata artifacts through providers to the client.

Context and ranking. Algorithm 2 sketches request handling. Trigger character `.` and dotted prefixes (`ta.sm`) route to module completion; otherwise a fuzzy filter scores candidates (exact label 1000, prefix 500, substring 100, category 50, brief 25; default limit 50). Sort keys prefer modules (`\x01...`) over root symbols (`\x02...`). Category headers appear as folder-kind items with empty insert text. Snippets use `InsertTextFormat.Snippet` when metadata supplies `\${...}` placeholders. Resolve rehydrates full documentation for a single item via `get_builtin(label)`.

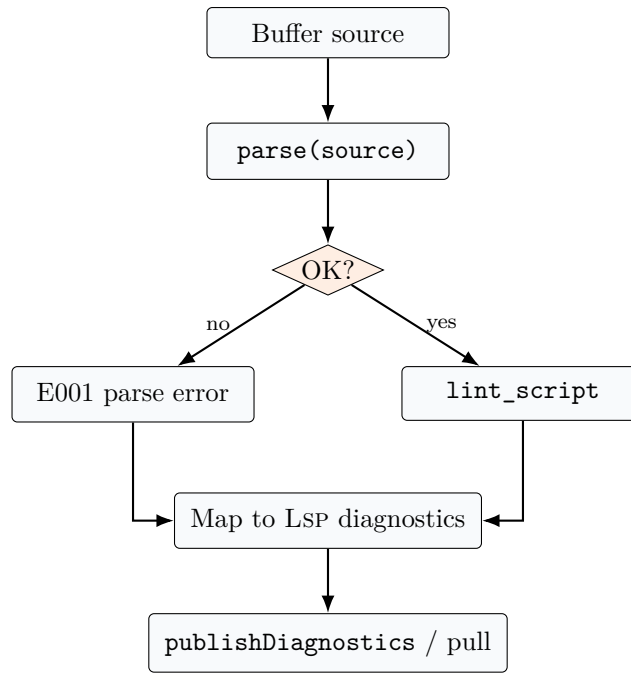


Figure 4: Diagnostics pipeline: parse, optional lint, conversion, push/pull publish.

Require: URI u , source s , version v

- 1: $doc \leftarrow \text{PUTORUPDATE}(u, s, v)$
- 2: $\text{PARSEANDLINT}(doc)$
- 3: $D \leftarrow []$
- 4: **if** $doc.parse_error \neq \perp$ **then**
- 5: append E001 diagnostic at $doc.parse_error_line$ to D
- 6: **end if**
- 7: **for** each warning w in $doc.diagnostics$ **do**
- 8: append $\text{TOLSPDIAGNOSTIC}(w, s)$ to D
- 9: **end for**
- 10: $\text{PUBLISHDIAGNOSTICS}(u, D)$

Algorithm 1: PublishDiagnostics on document mutation

4.3 Hover and signatures

Hover resolves the word under the cursor—including dotted forms such as `ta.sma`—against `get_builtin`. Successful hits return Markdown combining signature `detail`, brief, and extended documentation. The server does not yet advertise a dedicated `signatureHelp` provider; signature text is instead surfaced through hover and completion `detail` fields, which is adequate for most Pine call sites and avoids half-implemented capability flags.

4.4 Navigation: definitions, references, symbols

Definition walks the AST with a `DefinitionFinder` visitor for the identifier under the cursor and returns same-document `Locations` (functions, types, assignments). **References** symmetrically collect name occurrences. **Document symbols** build an outline (functions $\rightarrow$ Function, type defs $\rightarrow$ Class, assignments $\rightarrow$ Variable). **Workspace symbols** scan all open documents and filter by case-insensitive substring query. Cross-file project indexes and library `import` resolution are future work (Section 11).

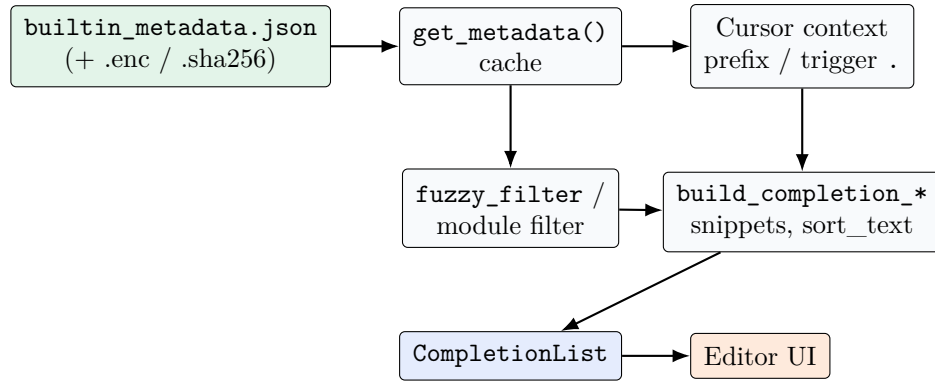


Figure 5: Completion data flow: metadata → items → client.

Require: params P , source s

- 1: $t \leftarrow$ text before cursor on line of $P.position$
- 2: $prefix \leftarrow$ last token of t
- 3: **if** $"." \in prefix$ **or** trigger char is $"."$ **then**
- 4: $m \leftarrow$ module name before final $.$
- 5: **return** BUILDMODULECOMPLETION(m)
- 6: **else**
- 7: **return** BUILDCOMPLETIONLIST($prefix$)
- 8: **end if**

Algorithm 2: HandleCompletion

4.5 Semantic tokens and inlay hints

Semantic tokens extend TextMate grammars with AST-backed classification. The server advertises a legend of thirteen types (namespace, type, class, function, method, variable, parameter, property, keyword, string, number, operator, comment) and four modifiers (declaration, definition, readonly, defaultLibrary). Tokens are delta-encoded for `textDocument/semanticTokens/full`; range requests are not advertised. Builtin namespaces (`ta`, `math`, `strategy`, ...) receive `defaultLibrary` modifiers.

Inlay hints show confident, shape-based inferred types next to declarations, e.g. `length = 14 → : const int, rsi = ta.rsi(...) → : series float`, drawing return types from metadata detail after `→`. Only high-confidence cases are emitted (`resolve_provider=False`).

4.6 Formatting via unparse

Formatting reuses the lossless unparser rather than a separate pretty-print grammar (Algorithm 3). Full-document and range formatting both parse, unparse, and emit a single `TextEdit` when the result differs. Parse failures return an empty edit list (no throw to the client), so broken mid-edit buffers never hard-fail format-on-save. This design keeps formatting semantics aligned with the canonical AST round-trip used in tests and `CLI format`.

5 Metadata Pipeline

5.1 Generation

Builtin metadata is a map from fully qualified names (e.g., `ta.sma`, `strategy.entry`) to records:

```
{
```

Require: source s

```

1: if  $s = \perp$  or  $s = \varepsilon$  then
2:   return []
3: end if
4: try  $tree \leftarrow \text{PARSE}(s)$ 
5:    $s' \leftarrow \text{UNPARSE}(tree)$ 
6:   if  $s' = s$  then
7:     return []
8:   end if
9:   return [REPLACEENTIREDOCUMENT( $s'$ )]
10: catch return []

```

Algorithm 3: FormatDocument

```

"ta.sma": {
  "label": "ta.sma",
  "kind": "function",
  "detail": "ta.sma(series, int) -> series float",
  "brief": "...",
  "documentation": "...",
  "snippet": "ta.sma(${1:param1}, ${2:param2})",
  "category": "ta.technical_analysis"
}

```

The generator `scripts/generate_builtin_metadata.py` introspects evaluator builtins and related surfaces; the JSON under `langserver/providers/` is regenerated rather than hand-edited as source of truth. Consumers include completion, hover, and inlay return-type extraction.

5.2 Encryption and integrity

For Nuitka-shipped binaries, plaintext is not ideal: casual `strings pynescrypt-lsp` would dump the documentation corpus. The build optionally encrypts with Fernet (AES-128-CBC + HMAC-SHA256) to `builtin_metadata.json.enc`, accompanied by a 16-character SHA-256 prefix of the plaintext in `builtin_metadata.json.sha256`. The key is supplied via `CRYPTO_KEY` / `.metadata.key` in CI and is not committed. This is *distribution hygiene* / *obscurity*, not an access-control boundary: anyone with the binary can eventually recover the catalog. It preserves a future knob for paid metadata endpoints without changing handler code.

5.3 Runtime load order

`get_metadata()` uses a process-local cache and the following order:

1. Prefer plaintext `builtin_metadata.json` when present and valid (development always wins after regen).
2. Else decrypt `.enc` via `metadata_decrypt.load_encrypted_metadata()`, validating the SHA-256 prefix when present.
3. Else return an empty dict (logged warnings); completion degrades gracefully.

Key resolution checks `.metadata.key` beside the package (or under `sys._MEIPASS` when frozen), then `PYNESCRIPT_METADATA_KEY`.

Table 2: Editor delivery matrix for PYNE language services.

Editor	Client form	Server discovery	Syntax
VS Code / Cursor	TypeScript extension (VSIX)	auto / PATH / module	TextMate + semantic
Neovim	<code>clients/neovim.lua</code>	PATH binary or module	Treesitter optional
Zed	<code>clients/zed.json</code>	PATH	Zed grammar optional
Emacs	<code>clients/emacs.el</code>	PATH	major mode
Helix / Sublime	docs snippets	PATH	client-specific

6 Multi-Editor Delivery

6.1 VS Code extension

The `vscode-extension/` package (**PYNE**, publisher `jango-blockchained`) is a TypeScript language client that:

- Registers language id `pinescript` with extensions `.pyne`, `.pine`, `.pinev5`, `.pinev6`, `.pinescript`.
- Contributes TextMate grammar `syntaxes/pinescript.tmLanguage.json` for baseline highlighting.
- Auto-detects the server: `pynscript-lsp` on PATH, else `python -m pynscript.langserver`, else a configured absolute path.
- Enables semantic highlighting defaults, sets the extension as default formatter, and exposes settings for diagnostics/completion/snippets.

Install path for users: `pip install "hoox-pyne[lsp]"` plus the VSIX, or a future marketplace listing. The extension does not reimplement language intelligence; it is a thin client over STDIO.

6.2 Thin clients: Neovim, Zed, Emacs

Directory `clients/` ships ready snippets:

- **Neovim** (`neovim.lua`): `nvim-lspconfig` registration, keymaps for definition, references, hover, format.
- **Zed** (`zed.json`): language server command `["pynscript-lsp"]` with STDIO.
- **Emacs** (`emacs.el`): `lsp-mode` client registration for `pinescript-mode`.

Helix and Sublime Text configurations are documented similarly. Any STDIO-capable client can launch `pynscript-lsp -stdio`. Table 2 summarizes the matrix.

6.3 Packaging channels

Three delivery shapes share one codebase [[HOOX Project, 2026b](#)]:

1. **pip:** `hoox-pyne[lsp]` for libraries, CLI, and module launch.
2. **Nuitka onefile:** self-contained `pynscript-lsp` for editors that prefer a single executable [[Hayen, 2024](#)].
3. **VS Code VSIX:** Node client + server discovery; optional binary bundle.

Container images on GHCR cover CLI/API operational use cases and are orthogonal to interactive LSP sessions.

7 CLI as Complementary Tooling Surface

The console script `pynescrypt` (Click-based) is deliberately *not* a subcommand of the language server. It targets headless and CI workflows that complement the editor:

Command	Role
<code>check</code>	Parse-only validation (CI-friendly exit codes)
<code>format / fmt</code>	Parse → unparse; optional in-place write
<code>lint</code>	Static rules; colored or JSON output
<code>parse-and-dump / dump / ast</code>	AST dump
<code>parse-and-unparse / unparse</code>	Round-trip source
<code>compile</code>	Transpile / Numba compile-check
<code>run</code>	Compile + execute on synthetic OHLCV
<code>data</code>	Fetch market bars (mock / Yahoo / ...)
<code>prewarm</code>	Warm Numba / IR caches
<code>info</code>	Version and optional extras

Shared components ensure behavioral parity: CLI `format` and LSP formatting both call the unparser; CLI `lint` and LSP diagnostics both call `lint_script`. The language server remains a separate entry point (`pynescrypt-lsp`) so editor sessions do not inherit CLI flag parsing or network data providers.

8 Integration with Evaluation

8.1 Shared AST, separate processes

Evaluation (literal expressions, full scripts, strategy backtests) uses the same parse → AST path as the language server but is *not* invoked on every keystroke. The design document [HOOX Project, 2026b] makes this separation explicit: `lint` is a cheap static phase; evaluation is a different cost profile and failure surface.

8.2 Pro API surface

The Pro API (`backend/`, Flask) exposes HTTP endpoints such as `evaluate`, `chart preview`, and `quick backtest`. Auth (API keys, tiers, rate limits) and scaling (gunicorn / Cloud Run) differ completely from single-user STDIO. Taking the API down must not break offline editing. Optionally, an HTTP bridge reuses completion/hover handlers for browser-hosted tools (AXIS) that cannot spawn `pygls` in-tab; that bridge is a convenience facade, not a replacement for the full LSP.

8.3 IDE-perspective run/preview

From an IDE user’s perspective, optional “run script” or “preview chart” actions are natural extensions. The preferred architecture is: editor command → local CLI or authenticated API call → render result in a webview or output channel—never by embedding market I/O inside diagnostic hot paths. This keeps the open-source editor experience fully offline while allowing premium evaluation tiers [HOOX Project, 2026a].

9 Implementation Experience and Design Lessons

Advertise only what you implement. Early drafts considered declaring signature help and code actions “for free.” Clients then surface broken UI affordances. PYNE now lists only wired handlers in `get_server_capabilities()`.

Pure feature handlers beat god-objects. Handlers of the form `handle_X(params, source)` (and optional cached tree) are unit-testable without spinning STDIO. Server methods become thin adapters.

Parse failure must be non-fatal. Mid-edit buffers are often temporarily invalid. The server retains process health, returns empty navigation/completion when needed, and still publishes E001.

Unparse is a formatter. For DSLs with a maintained unparser, a separate formatting grammar is unnecessary duplication. Round-trip tests become formatting regression tests.

Metadata is a product artifact. Treating builtin docs as generated, versioned, optionally encrypted data—not as hard-coded strings in handlers—keeps completion/hover in sync with the evaluator surface and enables binary packaging policies.

Multi-editor thin clients scale. Investing in a solid STDIO server plus a polished VS Code client, while shipping minimal configs for other editors, matches real adoption curves without N full extensions.

Financial DSL quirks. Pine’s series semantics, `var/varip`, and version annotations affect static analysis more than completion. Inlay hints remain intentionally conservative (shape-based) rather than claiming full type inference parity with the hosted compiler [Nystrom, 2021, Pierce, 2002].

10 Related Work

LSP ecosystem. The LSP specification [Microsoft, 2023] and early advocacy [Microsoft DevBlogs, 2016] established the multi-editor language service pattern. Production servers for TypeScript, Rust, Go, and Python demonstrate capability breadth; `pygls` lowers the barrier for Python-implemented servers [Open Law Library, 2020]. `pynescrypt-lsp` applies this stack to a financial DSL rather than a general-purpose language.

Language workbenches and DSLs. Fowler [Fowler, 2010] and Hudak [Hudak, 1996] emphasize that DSL value depends on tooling as much as semantics. Workbenches often generate editors from language definitions; PYNE instead reuses a hand-maintained ANTLR4 grammar [Parr, 2013, Parr et al., 2014] and ASDL schema [Wang et al., 1997] shared with the runtime—closer to interpreter engineering [Nystrom, 2021] than to generative workbenches.

Financial IDE tooling. Hosted trading platforms provide tightly integrated script editors with chart feedback but limited offline/CI support. Quantitative libraries (e.g., Python data stacks [McKinney, 2017, Harris et al., 2020]) offer powerful analysis without Pine-specific language services. PYNE targets the gap: local language intelligence for Pine Script™ sources plus optional evaluation, without claiming affiliation with the platform vendor [TradingView, Inc., 2024].

Binary distribution of Python tools. Nuitka [Hayen, 2024] and similar compilers address cold-start and dependency issues for shipping Python language servers to users who should not manage virtualenvs. Our encrypted metadata path is a packaging concern orthogonal to language semantics.

11 Limitations and Future Work

- **Same-document navigation.** Definitions/references do not yet resolve cross-file `import` graphs or library version pins.
- **No signature help / code actions.** Documented omissions; quick-fix stubs exist in diagnostics helpers but are not first-class features.
- **Incremental parse.** The workspace re-parses whole buffers; tree-sitter or incremental ANTLR strategies could reduce latency on large scripts.
- **Type system depth.** Inlay hints are shape-based; full series/type inference and strict v6 boolean semantics in the language server remain incomplete relative to the evaluator.
- **Hosted editor parity.** Chart-linked debugging, visual plot previews inside the editor, and bit-exact runtime parity with TradingView® are non-goals of the open LSP tier [HOOX Project, 2026b].
- **Security of metadata encryption.** Obscurity only; a determined party can extract the catalog from a binary.
- **Evaluation-in-IDE.** Richer offline mock-data evaluation panels and authenticated Pro preview commands are natural extensions.

12 Conclusion

Financial DSLs need not remain trapped in browser-only editors. This paper presented `pynescrypt-lsp`, a `pygls`-based Language Server for Pine Script™ within the PYNE toolchain. By centering a shared AST, caching parse/lint in a workspace, and layering metadata-driven completion and hover over a generated builtin catalog (~800+ items), the server delivers offline diagnostics, navigation, semantic highlighting, inlay hints, and unparse-based formatting to any STDIO-capable editor. Multi-surface delivery—VS Code extension, thin Neovim/Zed/Emacs clients, CLI, and a clean boundary with evaluation/API tiers—shows how language intelligence and runtime services can co-evolve without forcing online dependence for editing. We believe the architecture is reusable for other bar-oriented and trading DSLs seeking IDE-grade tooling under the LSP umbrella.

Acknowledgments

We thank contributors to the HOOX / PYNE open-source stack and users who reported editor integration issues across platforms.

Trademark disclaimer

Pine 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.

References

References

Martin Fowler. Domain-specific languages. Addison-Wesley, 2010.

- 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.
- 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.
- 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.
- 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, Sam Harwell, and Kathleen Fisher. Adaptive LL(*) parsing: The power of dynamic analysis. In *OOPSLA*. ACM, 2014.
- Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
- TradingView, Inc. Pine Script language reference manual. <https://www.tradingview.com/pine-script-docs/>, 2024. Accessed 2026.
- 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.

A Capability comparison (fair matrix)

Table 3 compares capability classes without claiming bit-exact parity with proprietary hosted editors. Hosted environments excel at live chart coupling; PYNE excels at offline, multi-editor, and CI workflows.

B Example configuration fragments

Neovim (conceptual)

```
require('lspconfig').pynscript.setup({})
-- server command: pynscript-lsp on PATH
```

Helix languages.toml (conceptual)

```
[[language]]
name = "pynscript"
file-types = ["pyne", "pine", "pinev5", "pinev6"]
command = "pynscript-lsp"
args = ["--stdio"]
```

Table 3: Capability matrix: browser-hosted TradingView® editor vs. PYNE local tooling (qualitative; not a product claim of equivalence).

Capability	Hosted browser editor	PYNE LSP + CLI
Offline editing	Limited / none	Yes
Multi-editor (Vim/Emacs/...)	No	Yes (LSP)
Syntax highlighting	Yes	Yes (TextMate + semantic)
Diagnostics on edit	Platform-specific	Parse + lint (push/pull)
Builtin completion / hover	Yes (online)	Yes (local metadata)
Go-to-definition / refs	Platform-specific	Same-document
Deterministic format (CI)	Limited	Yes (unparse + CLI)
Live chart / strategy UI	Strong	Via optional Pro API / external
Headless CI parse/lint	Weak	Strong
Open toolchain / self- host	No	Yes (AGPL package)

C Software availability

Source: project repository associated with PYNE / HOOX [[HOOX Project, 2026b,a](#)]. Install language server: `pip install "hoox-pyne[lsp]"`. Launch: `pynescrypt-lsp` or `python -m pynescrypt.langserver`.