

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-compiles 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

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

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

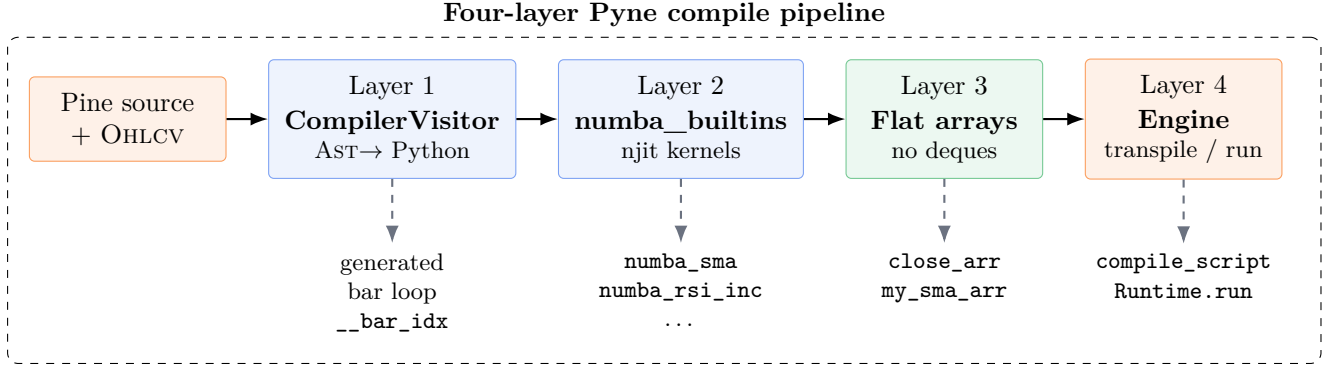


Figure 1: Four-layer compilation architecture. Layer 1 emits readable Python; Layer 2 provides numeric `ta` kernels; Layer 3 stores series as contiguous arrays; Layer 4 packages results for hosts and manages caches.

```
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 ~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
```

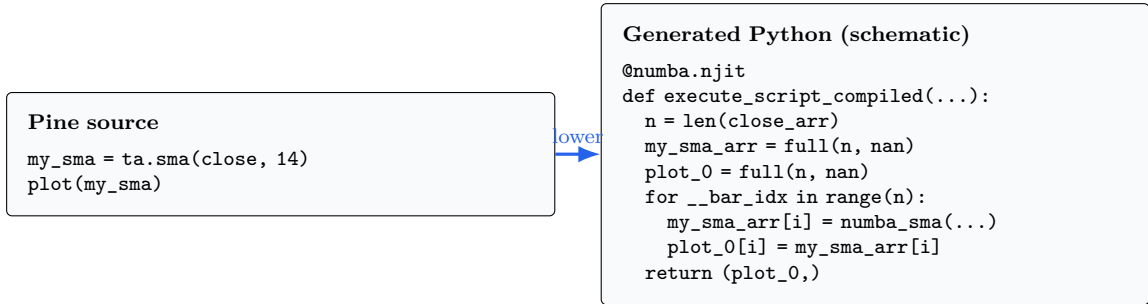


Figure 2: Numeric-mode lowering: series assignment and `plot` become full-length array writes inside an explicit bar loop. The host packs plot titles from visitor metadata.

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`.

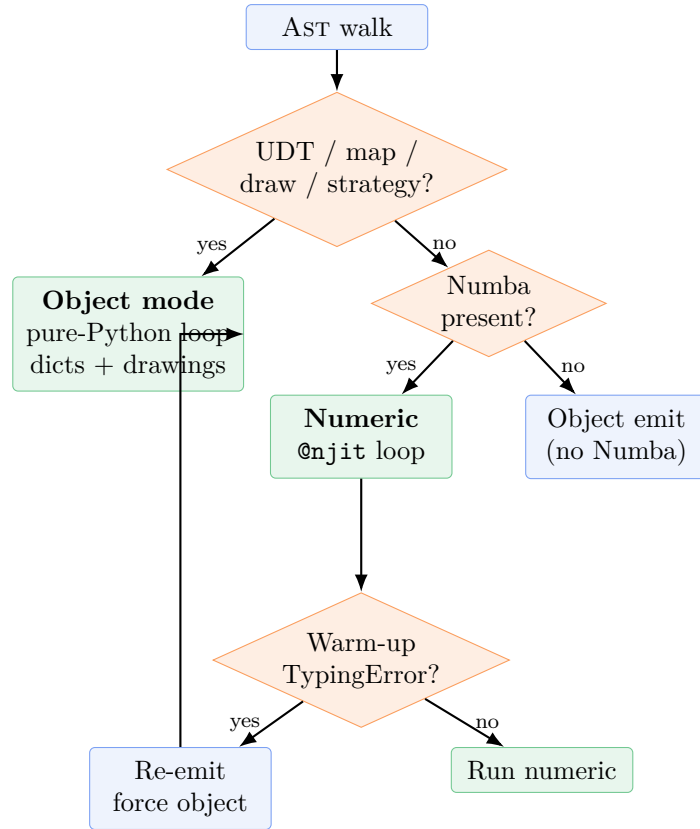


Figure 3: Numeric vs. object mode decision tree. Structural features force object emit; nopython warm-up failures recover by re-visiting with `force_object_mode=True`.

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

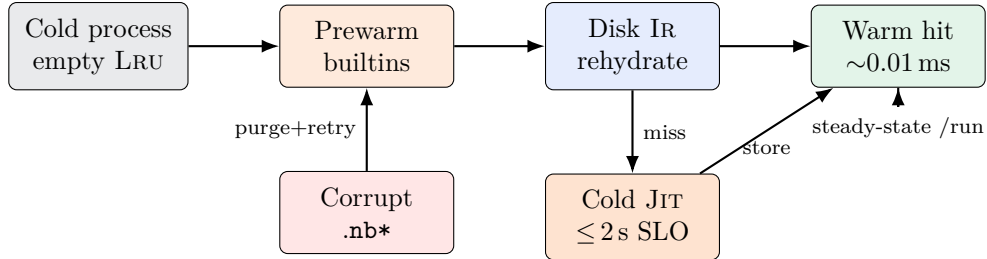


Figure 4: Warm-cache state machine. Deploy readiness should pay cold JIT via prewarm; interactive paths prefer in-process and disk hits.

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.

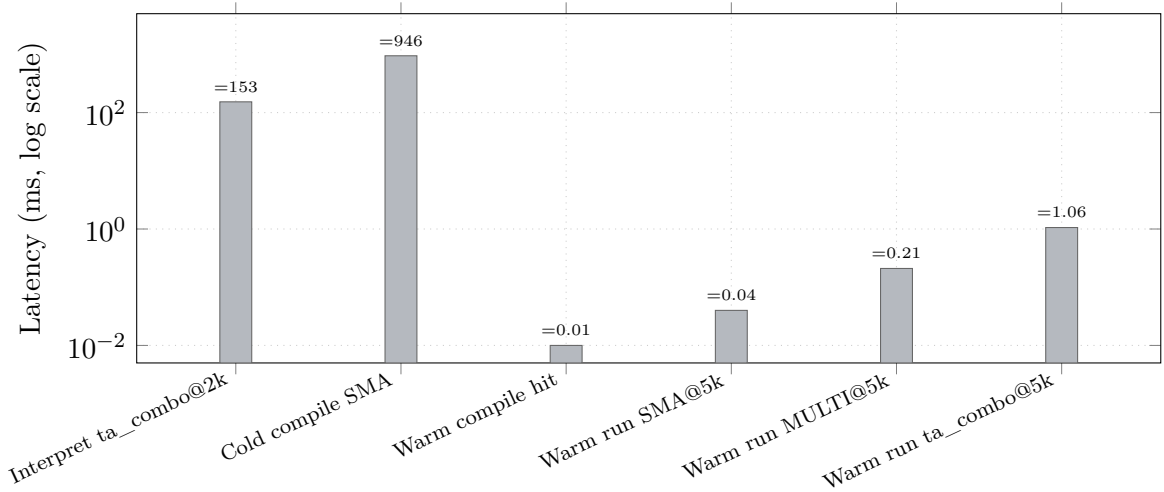


Figure 5: Indicative latencies (internal 2026): interpret combo residual vs. cold JIT, warm compile cache hits, and warm numeric executes. Log scale emphasizes the multi-order-of-magnitude gap between cold JIT and warm hits.

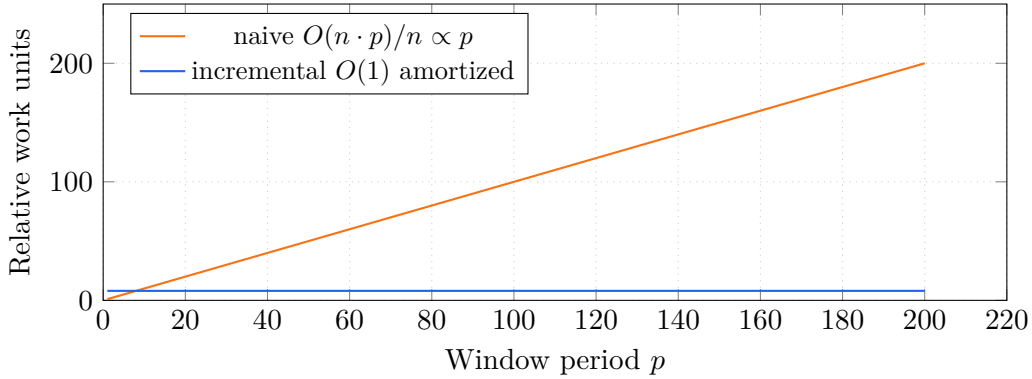


Figure 6: Asymptotic per-bar work for sliding SMA-style kernels: naive recompute grows with p ; incremental state remains flat after seeding.

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.