

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 `pynscript-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.

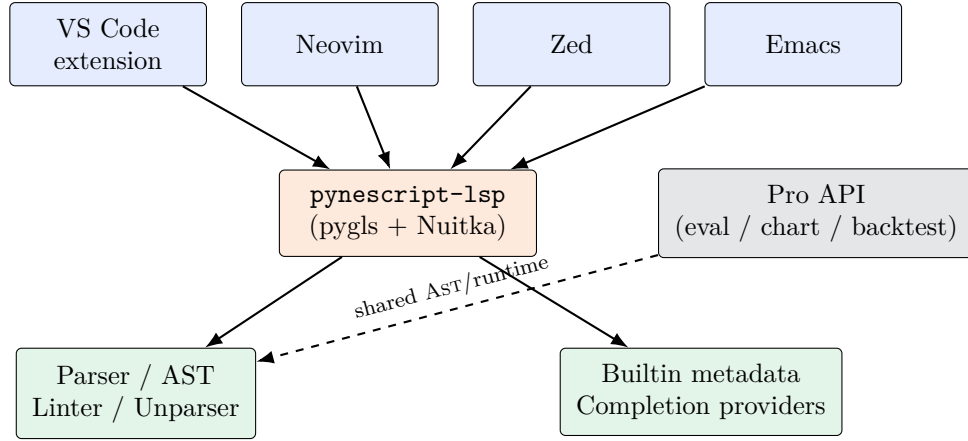


Figure 1: Multi-editor deployment topology. All editors use STDIO LSP; the Pro API shares AST/runtime but is a separate process and auth model.

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.

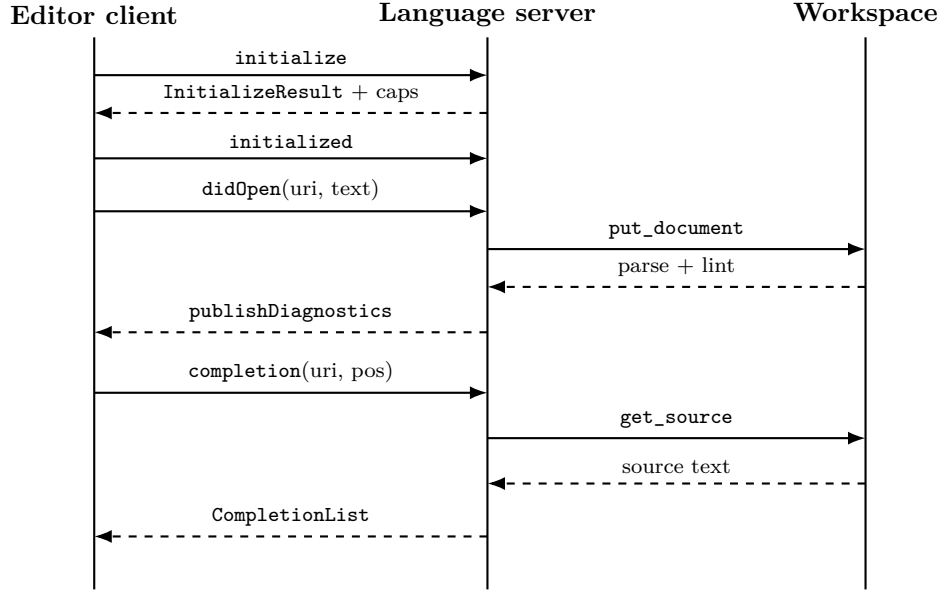


Figure 2: LSP request lifecycle: initialize, document open with push diagnostics, then a completion request served from workspace source and metadata providers.

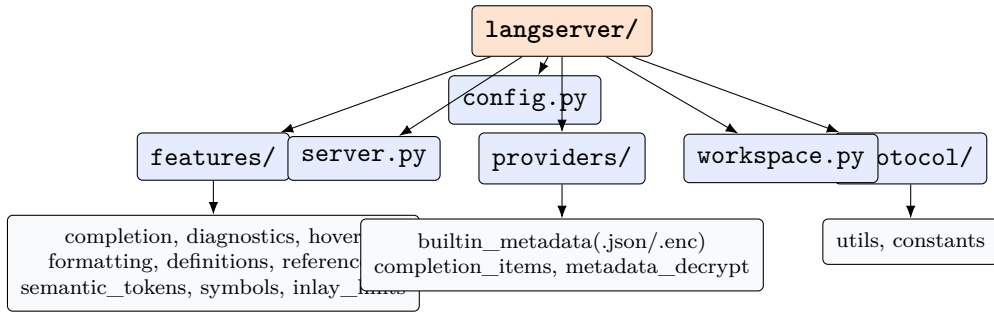


Figure 3: Feature module architecture of `pynescrypt-lsp`.

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)`.

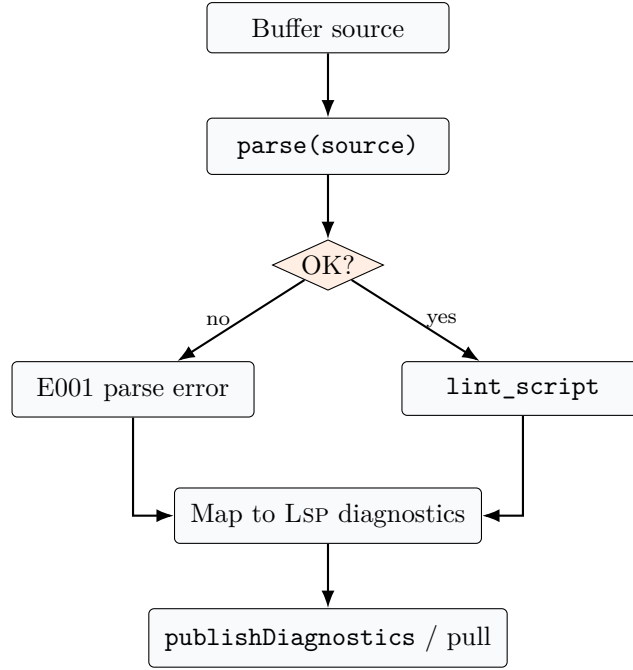


Figure 4: Diagnostics pipeline: parse, optional lint, conversion, push/pull publish.

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).

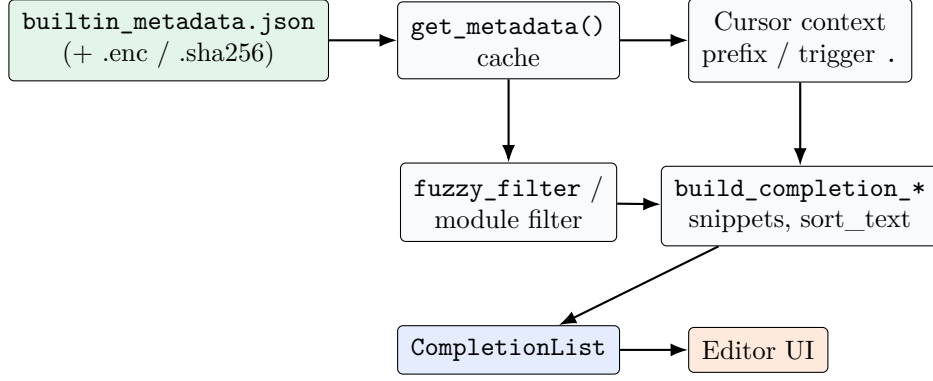


Figure 5: Completion data flow: metadata → items → client.

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 `replace 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: `pynescrypt-lsp` on PATH, else `python -m pynescrypt.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 `["pynescrypt-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 `pynescrypt-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 `pynescrypt-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`.