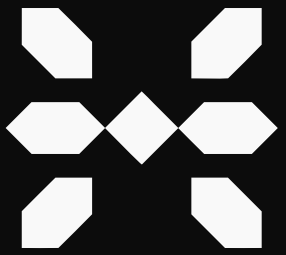
HOOX · PYNE · DOCUMENTATION



EDITOR INTEGRATORS · IDE AUTHORS

LANGUAGE SERVER MANUAL

LSP features, builtin metadata, VS Code, editor clients

Open-source Pine Script evaluation framework

CONTENTS

OVERVIEW

title: "Language Server" description: "PYNE Language Server Protocol surface — diagnostics, completion, hover, navigation, formatting, and editor clients."

LANGUAGE SERVER

The PYNE **Language Server** is the editor-facing half of the evaluation stack: a pygls process that speaks LSP over STDIO (or TCP in tooling), parses Pine with the same ANTLR4 → ASDL pipeline as the runtime, and never requires a proprietary chart host.

Abstract

Editors want *static* intelligence at typing cadence. Evaluation over OHLCV is *dynamic* and bar-loop expensive. PYNE therefore splits the stack:

Process	Cadence	Shared substrate
LSP (`pynescript-lsp`)	keystroke / open / save	parse, lint, unparse, builtin metadata
Pro API (`backend.app`)	request / batch	parse, evaluate, plots / events
CLI / library	batch / embed	same AST + evaluator

The LSP publishes diagnostics from the linter and parse errors, completes and hovers from Fernet-ready builtin metadata, navigates user symbols via AST visitors, and formats by round-tripping through the unparser. Feature modules live under `src/pynescript/langserver/features/` ; the VS Code extension and `clients/` configs are thin protocol hosts.

Conceptual model

```
flowchart TB
  subgraph Editors
    VS[VS Code extension]
    NV[Neovim / Helix / Zed / Emacs]
  end
  subgraph LSP["pynescript-lsp (pygls)"]
    S[PynescriptLanguageServer]
    W[Workspace + TextDocumentState]
    F[Feature handlers]
    M[Builtin metadata provider]
  end
  subgraph Core
    P[parse / lint / unparse]
    A[ASDL AST]
  end
  VS -->|stdio LSP| S
  NV -->|stdio LSP| S
  S --> W
  S --> F
  F --> M
  W --> P
  F --> P
  P --> A
```

Invariant: open/change/close always re-parse and re-lint the in-memory buffer. Features that need an AST re-parse on demand if the workspace cache is missing source.

Interface surface

Capability	LSP method(s)	Module
Sync	`textDocument/didOpen	didChange
Diagnostics (push + pull)	`publishDiagnostics`, `textDocument/diagnostic`	`workspace.py`, `features/diagnostics.py`
Completion (+ resolve)	`textDocument/completion`, `completionItem/resolve`	`features/completion.py`
Hover	`textDocument/hover`	`features/hover.py`
Definition / references	`textDocument/definition`, `textDocument/references`	`definitions.py`, `references.py`
Symbols	`textDocument/documentSymbol`, `workspace/symbol`	`symbols.py`, `server.py`
Formatting	`textDocument/formatting`, `rangeFormatting`	`features/formatting.py`
Inlay hints	`textDocument/inlayHint`	`features/inlay_hints.py`
Semantic tokens	`textDocument/semanticTokens/full`	`features/semantic_tokens.py` (stub data)

Declared capabilities: `src/pynescript/langserver/config.py` (`get_server_capabilities`).

Internals

Path	Role
`src/pynescript/langserver/server.py`	`PynescriptLanguageServer`, method registration
`src/pynescript/langserver/workspace.py`	Document map, incremental edits, parse+lint
`src/pynescript/langserver/config.py`	ServerCapabilities
`src/pynescript/langserver/features/*`	Per-method handlers
`src/pynescript/langserver/providers/*`	Metadata + completion builders
`src/pynescript/langserver/__main__.py`	STDIO entry (`pynescript-lsp`)
`vscode-extension/`	Language client + TextMate grammar
`clients/`	Neovim, Zed, Emacs, Helix snippets

Console scripts are separate: `pynescript` (Click CLI) vs `pynescript-lsp` (pygls). Do not conflate them — see architecture.

Tracks in this tab

1. Architecture — server lifecycle, workspace, capabilities
2. Diagnostics through Inlay hints — feature manuals
3. Builtin metadata — generation + Fernet `.enc` / `CRYPTO_KEY`
4. VS Code extension — client packaging
5. Clients — Neovim, Zed, Emacs, Helix, Sublime

See also

- Pro API — HTTP evaluate path sharing the same AST
- Evaluate contract
- Linter
- Unparser
- Editors guide (end user)

ARCHITECTURE

title: "LSP Architecture" description: "PynescriptLanguageServer lifecycle, Workspace document model, capability declaration, and STDIO transport."

LSP ARCHITECTURE

Abstract

The language server is a single-process pygls application. It owns a `Workspace` of open `TextDocumentState` records, re-parses on every mutation, and dispatches LSP methods to pure-ish feature handlers that take `(params, source)` rather than holding server globals. Capabilities are declared once at `initialize`; transport defaults to STDIO for editor integration.

Conceptual model

```
sequenceDiagram
  participant C as Editor client
  participant S as PynescriptLanguageServer
  participant W as Workspace
  participant P as parse + lint

  C->>S: initialize
  S-->>C: InitializeResult + ServerCapabilities
  C->>S: initialized
  C->>S: textDocument/didOpen
  S->>W: put_document(uri, text, version)
  W->>P: parse + lint_script
  S-->>C: publishDiagnostics
  C->>S: textDocument/completion | hover | ...
  S->>W: get_source(uri)
  S-->>C: feature result
  C->>S: textDocument/didChange (incremental)
  S->>W: update_document + re-parse/lint
  S-->>C: publishDiagnostics
  C->>S: shutdown / exit
```

Interface surface

Process entry

```
# Editable install
pip install -e ".[lsp]"
python -m pynescript.langserver   # same as pynescript-lsp
make run-lsp
```

Entry: `src/pynescript/langserver/__main__.py` constructs `PynescriptLanguageServer()` and calls `server.start_io()` (STDIO JSON-RPC).

`pyproject.toml` registers:

```
pynescript-lsp = "pynescript.langserver.__main__:main"
```

Server class

`PynescriptLanguageServer` (`server.py`):

- Subclasses `pygls.lsp.server.LanguageServer` with `name="Pynescript"`, `version="0.1.0"`.
- Instantiates `self.pine_workspace = Workspace()`.
- Registers handlers in `setup_method_handlers()` via `@self.feature(...)`.

Document sync

Event	Behavior
`didOpen`	`put_document` → parse/lint → push diagnostics
`didChange`	Incremental or full-text apply → re-parse/lint → push diagnostics
`didClose`	Remove document; publish empty diagnostic list
`didSave`	Re-publish diagnostics for current buffer

Sync options (`config.get_server_capabilities`):

- `TextDocumentSyncKind.Incremental`
- `open_close=True`
- `save` with `include_text=True`
- `will_save` / `will_save_wait_until` off

Capability set (declared)

From `config.py`:

- Diagnostic provider (`identifier="pynescript-diagnostics"`, `workspace_diagnostics=False` in options; workspace pull handler still exists)
- Completion: trigger `"."`, `resolve_provider=True`
- Hover, definition, references
- Document + workspace symbols
- Full + range formatting
- Inlay hints (`resolve_provider=False`)
- Semantic tokens full (legend of standard token types/modifiers; range=false)
- Signature help and code action *options are advertised*; dedicated handlers are not fully wired as first-class feature modules yet — treat as capability stubs for clients that probe the table

File filters (`get_filter_options`): `*.pine`, `*.pinev5`, `*.pinev6` under language id `pinescript`.

Internals

Workspace

`Workspace` (`workspace.py`) maps `uri → TextDocumentState`:

```
TextDocumentState
  uri, source, version
  ast | None
  diagnostics: list[LintWarning]
  parse_error, parse_error_line
```

`_parse_and_lint`:

1. `parse(source, filename=uri)` — on success, `lint_script(source, filename=uri)`.
2. On exception: clear AST, store `parse_error` string, extract line via regex `line[:\s]+(\d+)`.

Incremental edits: `_apply_text_edit` splits on `\n`, replaces the LSP range, rejoins. Whole-document change events replace `source` wholesale.

Diagnostics conversion (`_lint_warnings_to_diagnostics`):

- Map severity strings → `DiagnosticSeverity`.
- Source tag `"PineScript"`, code from lint rule.
- Append synthetic **E001** error for parse failures at the extracted line.

Feature dispatch pattern

Handlers are functions, not server methods, e.g.:

```
@self.feature(lsp.TEXT_DOCUMENT_COMPLETION)
def text_completion(params):
    source = self.pine_workspace.get_source(params.text_document.uri)
    return completion_feature.handle_completion(params, source)
```

Definition / references / symbols also pass `uri` for `Location` construction. Inlay hints and semantic tokens re-parse from `source` inside the feature module.

Pull diagnostics

- `textDocument/diagnostic` → `RelatedFullDocumentDiagnosticReport` with `result_id=f"{uri}-{version}"`.
- `workspace/diagnostic` → one full report per open document from `get_all_diagnostics()`.

Workspace symbols

`workspace/symbol` walks each open document's AST via `_collect_workspace_symbols`: `FunctionDef` → Function, `TypeDef` → Class, `Assign` to `Name` → Variable. Filter is case-insensitive substring on `params.query`.

Invariants and edge cases

1. **Parse failure is non-fatal for the process.** AST-dependent features return `None` or `[]`; diagnostics still surface E001.
2. **Source of truth is the open buffer**, not disk. Save only re-publishes; it does not re-read files.
3. **Incremental sync assumes the client sends coherent ranges.** Out-of-bounds ranges leave source unchanged (`_apply_text_edit` early-return).
4. **Version** is stored and used in pull-diagnostic `result_id`; the server does not reject out-of-order versions itself.
5. **Nuitka onefile** embeds providers data and may rely on encrypted metadata — see builtin metadata. Architecture of handlers is identical to the pure-Python path.

Worked example — minimal initialize

Client → server (conceptual):

```
{
  "method": "initialize",
  "params": {
    "capabilities": {},
    "clientInfo": { "name": "example", "version": "1" }
  }
}
```

Server returns `InitializeResult` with `serverInfo.name = "Pynescript Language Server"` and the full `ServerCapabilities` table from `get_server_capabilities()`.

Failure modes

Symptom	Likely cause
No diagnostics on open	Client did not send `didOpen` or language id is not `pinescript`
Completions empty	Metadata file missing/undecryptable; see builtin metadata
Format no-ops	Parse error (handler returns `[]`) or unparsed text equals source
Extension can't spawn server	`pynescript-lsp` not on `PATH`; check VS Code extension

See also

- Diagnostics
- Builtin metadata
- VS Code extension
- Nuitka build

DIAGNOSTICS

title: "Diagnostics" description: "Push and pull diagnostics: parse errors, lint rules, severity mapping, and code-action stubs."

DIAGNOSTICS

Abstract

Diagnostics are the primary static feedback channel. The workspace re-parses and re-lints on every open/change/save, converts `LintWarning` objects (and parse failures) into `lsp.Diagnostic`, then **pushes** them with `textDocument/publishDiagnostics`. Clients that support LSP 3.16+ pull diagnostics can also request `textDocument/diagnostic` or `workspace/diagnostic`.

Conceptual model

```
flowchart LR
  SRC[Buffer source] --> PARSE[parse]
  PARSE -->|ok| LINT[lint_script]
  PARSE -->|err| E001[Diagnostic E001]
  LINT --> LW[LintWarning list]
  LW --> MAP[lint_warnings_to_diagnostics]
  E001 --> PUB[publishDiagnostics]
  MAP --> PUB
  PUB --> ED[Editor squiggles]
```

Interface surface

Push (always on)

Triggered from `server.py` on `didOpen`, `didChange`, `didSave`. Close clears diagnostics (`items=[]`).

Pull

Method	Response
`textDocument/diagnostic`	`RelatedFullDocumentDiagnosticReport` with `kind=full`, `result_id="{uri}-{version}"`
`workspace/diagnostic`	One `WorkspaceFullDocumentDiagnosticReport` per open URI

Capability: `diagnostic_provider` with `identifier="pynescript-diagnostics"`, `inter_file_dependencies=False`.

Diagnostic shape

Field	Value
`range`	0-based line; column from warning or 0; end spans remainder of line (capped)
`severity`	error / warning / information / hint
`message`	Human-readable lint or parse message
`source`	`"PineScript"`
`code`	Lint code (e.g. `W001`) or `E001` for parse errors

The richer conversion in `features/diagnostics.py` also supports:

- `code_description` hrefs for codes starting with `E` / `W` (docs URLs)
- `tags`: `W001` → Unnecessary, `W002` → Deprecated

Workspace conversion in `workspace.py` is the path used on push; the feature module API is available for shared logic and quick-fix helpers.

Internals

Path	Role
`src/pynescript/langserver/workspace.py`	`_parse_and_lint`, `_lint_warnings_to_diagnostics`
`src/pynescript/langserver/features/diagnostics.py`	Conversion helpers, tags, `create_quick_fix`
`src/pynescript/ast/linter.py`	Rule engine producing `LintWarning`

Severity map

```
error        → DiagnosticSeverity.Error
warning      → Warning
info|information → Information
hint         → Hint
(default)    → Warning
```

Parse errors

On exception from `parse`:

- `ast = None`, `diagnostics = []` (lint skipped)
- Message stored as `parse_error`
- Synthetic diagnostic code **E001**, severity Error, line from regex on the exception text

Quick fixes (feature module)

`create_quick_fix` (not yet fully wired through `workspace/executeCommand` / codeAction handlers in `server.py`):

- **W001** → insert `//@version=5\n` at document start
- **C002** (long line) → suggest format document command

Capability advertises QuickFix / Refactor / SourceOrganizeImports kinds for future wiring.

Invariants and edge cases

1. **Lint does not run when parse fails** — only E001 is shown.
2. Warnings without a line number are dropped (`None` conversion).
3. End column heuristic uses full line length (or +10) and clamps to 2000 characters — not a precise token span.
4. `inter_file_dependencies=False`: no cross-file analysis; each URI is independent.
5. Push and pull should agree for a given buffer version; both read the same `TextDocumentState`.

Worked example

Source:

```
indicator("x")
plot(close)
```

If the linter emits a missing-version warning (code `W001`), the client receives a Warning diagnostic on the relevant line with `source: "PineScript"` and may offer the version-header quick fix when code actions are connected.

On a hard parse failure at line 3, expect:

```
{
  "severity": 1,
  "code": "E001",
  "source": "PineScript",
  "message": "<exception text>"
}
```

Failure modes

Symptom	Cause
Stale squiggles after edit	Client using full-document sync incorrectly, or version mismatch
Empty diagnostics on broken file	Unexpected: parse errors should still emit E001 — check client filter on `source`
No workspace pull results	Only **open** documents are tracked; closed files are absent

See also

- Linter (core)
- Error model
- Architecture

COMPLETION

title: "Completion" description: "textDocument/completion and completionItem/resolve from builtin metadata, module-dot triggers, and fuzzy filter."

COMPLETION

Abstract

Completion is metadata-driven, not evaluator-driven. The server loads a catalog of builtins (plaintext JSON in development, Fernet-encrypted in compiled binaries), filters by prefix or module, and returns `CompletionItem` rows with optional snippet `insertText`. Resolve re-hydrates documentation for a single item.

Conceptual model

```
flowchart TD
  CUR[Cursor context] --> TR{trigger / prefix}
  TR -->|module.| MOD[build_module_completion]
  TR -->|prefix text| ALL[build_completion_list + fuzzy_filter]
  MOD --> META[get_metadata]
  ALL --> META
  META --> ITEMS[CompletionList]
  RES[completionItem/resolve] --> GET[get_builtin label]
  GET --> FULL[build_completion_item]
```

Interface surface

Method	Capability
`textDocument/completion`	`trigger_characters=["."]`, returns `CompletionList`
`completionItem/resolve`	`resolve_provider=True`

Handler: `features/completion.py` → `handle_completion` / `handle_completion_resolve`.

Trigger logic

1. Compute `text_before_cursor` on the current line.
2. `get_trigger_char` — if the previous character is `.` (or `(`, `,`, space), treat as trigger.
3. If the last token contains `.` (e.g. `ta.` or `ta.sm`), call `build_module_completion(module)`.
4. Else `build_completion_list(prefix=prefix)` over the full catalog.

Word boundaries for Pine identifiers: `[a-zA-Z_][a-zA-Z0-9_.]*` (`protocol/utils.py`).

Item fields

Built by `providers/completion_items.py`:

Field	Source
`label`	metadata `label` (e.g. `ta.sma`)
`kind`	`CompletionItemKind.Function`
`detail`	signature / detail string
`documentation`	Markdown from metadata
`insert_text`	Snippet if `${...}` present, else plain label
`insert_text_format`	Snippet or PlainText
`filter_text`	dotted parts + brief
`sort_text`	modules first (`\x01…`), root second (`\x02…`)

Category headers may appear as Folder-kind items with empty insert text and labels like `--- Technical Analysis (ta.*) (N) ---`.

Internals

Path	Role
`features/completion.py`	Request context + dispatch
`providers/completion_items.py`	List / item / module builders
`providers/builtin_metadata.py`	Load cache, `fuzzy_filter`, `get_builtin`
`protocol/utils.py`	Word + trigger helpers

Fuzzy filter scores

`fuzzy_filter(query, items, limit=50)`:

Match	Score
Exact label	1000
Prefix	500
Substring in label	100
Category	50
Brief	25

Results sorted by score descending, capped at `limit`.

Resolve

`handle_completion_resolve` looks up `params.label` in metadata; if found, rebuilds a full `CompletionItem`. Unknown labels return unchanged.

Invariants and edge cases

1. **No user-symbol completion yet** — only catalog builtins (not local `myFunc` unless it appears in metadata).
2. Empty metadata dict yields an empty list (failed decrypt / missing files).
3. Module completion uses `startswith(module + ".")` — nested namespaces beyond one dot still work if labels are fully qualified.
4. Snippets are generated at metadata build time (`scripts/generate_builtin_metadata.py`); incompleteness of placeholders is a generator concern, not the LSP handler.

Worked example

User types `ta.` → module completion for all `ta.*` labels.

User types `sma` without a module → fuzzy filter may return `ta.sma` among others (substring score).

Resolve on `ta.sma` reloads full documentation markup for the detail pane.

Failure modes

Symptom	Cause
Zero items always	Metadata not loaded — check builtin metadata
Dot trigger ignored	Client did not set completion trigger characters from server capabilities
Snippets inserted raw	Client disabled snippet support; extension setting `pynescript.completion.snippets`

See also

- Hover — shares metadata
- Builtin metadata
- Inlay hints — uses return types from `detail`

HOVER

title: "Hover" description: "textDocument/hover — Markdown documentation cards for Pine builtins from metadata."

HOVER

Abstract

Hover answers “what is this identifier?” for **builtins**. The handler extracts the word under the cursor (including dotted forms), looks up metadata, and returns a Markdown card: signature fence, brief, optional long documentation, curated examples, and related symbols. User-defined symbols without metadata return `null`.

Conceptual model

```
flowchart LR
  POS[Position] --> WORD[get_word_at_position]
  WORD --> META{get_builtin}
  META -->|hit| CARD[MarkupContent Markdown]
  META -->|miss| DOT{module. prefix?}
  DOT -->|ta.sma| META
  DOT -->|no| NULL[null]
```

Interface surface

- Method: `textDocument/hover`
- Capability: `hover_provider=True`
- Handler: `features/hover.py` → `handle_hover`

Lookup order

1. Exact word at cursor via `get_word_at_position` (identifiers may include `.`).
2. If no metadata hit, inspect text before the word: if it ends with `module.`, try `module.word`.
3. Build hover with range covering `[start, end)` of the word on the line.

Card structure

```
```pinescript
{detail}
```

{brief}

{first documentation paragraph, ≤500 chars}

Example: ... **See also:** `ta.ema`, ...
```

```
Examples and related maps are **hardcoded** for a small high-value set (`ta.sma`, `ta.ema`, `ta.rsi

External TradingView doc links are intentionally omitted (`_build_docs_link` returns empty) to keep

## Internals

| Path | Role |
| --- | --- |
| `features/hover.py` | Word resolution + Markdown assembly |
| `providers/builtin_metadata.py` | `get_builtin` |
| `protocol/utils.py` | Word extraction |

## Invariants and edge cases

1. **No source → null.** Missing buffer aborts early.
2. **Line OOB → null.** Cursor past last line is ignored.
3. **Empty word → null.** Whitespace / punctuation yields no hover.
4. **User symbols** (local variables, UDTs) are not inferred for hover; use [definition](/pyne/docs
5. Documentation truncation is first-paragraph / 500-char only — full docs live in metadata and com

## Worked example

Cursor on `sma` in `plot(ta.sma(close, 14))`:

- Word may be `ta.sma` if the dotted identifier is one match, or `sma` with module recovery from `t
- Response `contents.kind = markdown` with fence of the metadata `detail` line.

## Failure modes

| Symptom | Cause |
| --- | --- |
| Hover never appears | Metadata empty or word not in catalog |
| Wrong symbol | Identifier regex swallowed neighboring text — rare with `_.` pattern |
| Stale docs after adding builtin | Regenerate metadata — [builtin metadata](/pyne/docs/lsp/builtin

## See also

- [Completion](/pyne/docs/lsp/features/completion)
- [Builtin metadata](/pyne/docs/lsp/builtin-metadata)
- [Navigation](/pyne/docs/lsp/features/navigation)
```

NAVIGATION

title: "Navigation" description: "Go-to-definition, find-references, document outline, and workspace symbol search over the ASDL AST."

NAVIGATION

Abstract

Navigation features operate on **user AST symbols**, not the builtin catalog. Definition and references walk a freshly parsed tree with `NodeVisitor` subclasses; document symbols build a hierarchical outline; workspace symbols scan all **open** documents. Locations currently anchor to line starts (coarse ranges) — enough for jump lists, not character-precise rename.

Conceptual model

```
flowchart TB
  SRC[Source buffer] --> PARSE[parse]
  PARSE --> AST[ASDL tree]
  AST --> DEF[DefinitionFinder]
  AST --> REF[ReferencesFinder]
  AST --> SYM[DocumentSymbolCollector]
  DEF --> LOC[Location list]
  REF --> LOC
  SYM --> TREE[DocumentSymbol tree]
  WS[workspace/symbol] --> OPEN[All open ASTs]
```

Interface surface

Method	Handler	Returns
`textDocument/definition`	`definitions.handle_definition`	`list[Location] | None`
`textDocument/references`	`references.handle_references`	`list[Location]` (possibly empty)
`textDocument/documentSymbol`	`symbols.handle_document_symbols`	`list[DocumentSymbol]`
`workspace/symbol`	`server._collect_workspace_symbols`	`list[SymbolInformation]`

Capabilities: `definition_provider`, `references_provider`, document + workspace symbol options with work-done progress flags.

Definition (`DefinitionFinder`)

Matches target word against:

- `FunctionDef.name`
- `TypeDef.name`
- `Assign` targets (`Name` or nested names)

Ignores mere call sites (`visit_Call` returns early when the callee name matches).

References (`ReferencesFinder`)

- `Name` with Load / Store contexts
- Function and type definitions when `include_declaration` is true
- Call sites of bare `Name` callees

`params.context.include_declaration` controls whether declarations appear.

Document symbols

Hierarchical:

AST	SymbolKind	Children
`FunctionDef`	Function	Variables assigned inside body
`TypeDef`	Class	Fields (`Assign` targets)
Top-level `Assign`	Variable	—

Function `detail` is a coarse `function name()` string; assignments may show callee module attribute as detail when RHS is a call.

Workspace symbols

Flat `SymbolInformation` for functions, types, and assigned names across open buffers; filtered by lowercase query substring.

Internals

Path	Role
`features/definitions.py`	Go-to-definition visitor
`features/references.py`	Find-all-references visitor
`features/symbols.py`	Document outline
`server.py`	Workspace symbol aggregation

Parse is **per request** inside definition/references/symbols (not the workspace cached AST). That trades a little CPU for isolation if cache and buffer ever diverge.

Invariants and edge cases

1. **Parse failure → empty / null** (definition `None`, others `[]`).
2. **Builtin names** have no definition in-file; clients should not expect `ta.sma` to resolve to a library file URI.
3. **Ranges are line-coarse** for many locations (`character=0`); selection ranges for document symbols use `col_offset` when present.
4. **Scope is naive** — same identifier in nested functions may collect multiple definitions; no shadowing analysis.
5. **Closed documents** disappear from workspace symbol search.

Worked example

```
//@version=5
indicator("nav")
length = 14
mySma(src, len) =>
    ta.sma(src, len)
v = mySma(close, length)
```

- Definition on `mySma` at the call site → function line.
- References on `length` → assign + call argument (and declaration if included).
- Outline → `length` variable, `mySma` function (with locals), `v` variable.

Failure modes

Symptom	Cause
Jump lands column 0	Expected with current Location encoding
Missing references inside function when targeting another function	`visit_FunctionDef` only walks body when the def name **differs** from target — intentional for declaration handling; calls inside same-named functions are a known edge
Empty outline	Parse error or empty source

See also

- Architecture
- Hover — builtins vs user symbols
- Diagnostics

FORMATTING

title: "Formatting" description: "Document and range formatting via AST parse → NodeUnparser round-trip."

FORMATTING

Abstract

Formatting is **canonical unparse**, not a style linter. The handler parses the buffer, runs `NodeUnparser`, and if the result differs from the source, replaces the document (or the requested line range) with a single `TextEdit`. There is no prettier-like option matrix; `FormattingOptions` from the client are currently unused.

Conceptual model

```
flowchart LR
  SRC[Source] --> PARSE[parse]
  PARSE --> AST[ASDL]
  AST --> UNP[NodeUnparser.visit]
  UNP --> OUT[Formatted text]
  OUT -->|equals source| EMPTY[empty edit list]
  OUT -->|differs| EDIT[TextEdit replace]
```

Interface surface

Method	Handler
`textDocument/formatting`	`handle_formatting`
`textDocument/rangeFormatting`	`handle_range_formatting`

Capabilities: both providers `True` in `config.py`.

Full document

1. Parse with filename `"<format>"`.
2. `formatted = NodeUnparser().visit(tree)`.
3. If identical to source → `[]`.
4. Else one `TextEdit` spanning `(0,0)` → last line/col with `new_text=formatted`.

Range

1. Unparse the **entire** document.
2. Slice **formatted** lines and **source** lines by the client range's start/end line indices.
3. If the in-range slices match → `[]`.
4. Else replace `params.range` with the formatted line slice.

This is line-aligned, not AST-node-aligned: partial mid-line ranges may produce surprising splices when unparse changes line structure.

Internals

Path	Role
`features/formatting.py`	LSP handlers
`src/pynescript/ast/helper.py`	`parse`
`src/pynescript/ast/unparser.py`	`NodeUnparser`

VS Code command `pynescript.formatDocument` delegates to `editor.action.formatDocument`, which hits this provider when the language client is active.

Invariants and edge cases

1. **Parse failure** → `[]` (no partial format; silent no-op from the client's perspective).

2. **Idempotence goal:** format twice should stabilize if unparse is canonical; any non-idempotent unparse is a core bug, not an LSP bug.
3. **Comments / trivia:** fidelity is whatever the unparser preserves — not a full CST pretty-printer.
4. **Range formatting** still requires a full successful parse of the whole file.
5. Tab size / insert spaces from `FormattingOptions` are ignored today.

Worked example

Before:

```
//@version=5
indicator("x")
plot(  close  )
```

After full format (illustrative — exact whitespace follows `NodeUnparser`):

```
//@version=5
indicator("x")
plot(close)
```

If source already matches unparse output, the server returns an empty edit list and the editor leaves the buffer alone.

Failure modes

Symptom	Cause
Format does nothing on broken syntax	Expected: exception → `[]`
Range format rewrites wrong lines	Unparse shifted line count; prefer full-document format
Client “format on save” loops	Unparse not idempotent — fix unparser, not client

See also

- Unparser
- Architecture
- VS Code extension

SEMANTIC TOKENS

title: "Semantic Tokens" description: "textDocument/semanticTokens/full — capability, legend, and current stub implementation."

SEMANTIC TOKENS

Abstract

Semantic tokens let the server paint identifiers with roles richer than TextMate scopes (e.g. distinguish library functions from user variables). PYNE **advertises** a full semantic-tokens provider with a standard legend, and implements `textDocument/semanticTokens/full` as a **safe stub**: parse succeeds, `data` is an empty integer array. Editors that require tokens degrade to grammar-only highlighting without protocol errors.

Conceptual model

```
flowchart LR
  REQ[semanticTokens/full] --> SRC{source?}
  SRC -->|empty| EMPTY[SemanticTokens data=[]]
  SRC -->|present| PARSE[parse]
  PARSE -->|fail| EMPTY
  PARSE -->|ok| STUB[SemanticTokens data=[]  // future visitor]
  STUB -.->|planned| ENC["delta encoding: line, col, len, type, mods"]
```

Interface surface

Capability (`config.py` → `semantic_tokens_provider`):

```
legend.token_types:
  namespace, type, class, enum, interface, struct, typeParameter,
  parameter, variable, property, enumMember, event, function,
  method, macro, keyword, modifier, comment, string, number,
  regexp, operator

legend.token_modifiers:
  declaration, definition, readonly, static, deprecated,
  abstract, async, modification, documentation, defaultLibrary

range: false
full: true
```

Handler: `features/semantic_tokens.py` → `handle_semantic_tokens`.

Response type: `lsp.SemanticTokens(data=[...])` — LSP-encoded five-tuples as a flat `int` array.

Internals

Path	Role
`features/semantic_tokens.py`	Stub handler
`config.py`	Legend + capability
`server.py`	`TEXT_DOCUMENT_SEMANTIC_TOKENS_FULL` registration

The module docstring states intent: a visitor would emit `(line, col, len, token_type_index, modifiers_bitfield)` with relative line/character deltas per the LSP spec.

Until that lands, **TextMate** highlighting from `vscode-extension/syntaxes/pinescript.tmLanguage.json` remains the primary colorization path for VS Code.

Invariants and edge cases

1. **Never throws** — exceptions from parse yield empty data.
2. **No range provider** — clients must not request `semanticTokens/range`.
3. **Legend is fixed** at initialize; changing indices later would break cached clients.
4. Empty `data` is valid protocol, not an error.

Worked example

Request `textDocument/semanticTokens/full` for any open buffer:

```
{ "data": [] }
```

Client falls back to TextMate / tree-sitter grammar scopes.

Failure modes

Symptom	Cause
“Semantic highlighting” setting appears on but changes nothing	Stub returns empty data — expected
Client errors on range request	`range=False` in capabilities; client should not call it

See also

- VS Code extension — TextMate grammar
- Architecture
- Inlay hints — complementary visual annotations

INLAY HINTS

title: "Inlay Hints" description: "Inferred type annotations on simple assignments — const, series, and input kinds."

INLAY HINTS

Abstract

Inlay hints surface **inferred types** next to simple `name = expr` assignments when no explicit type annotation is present. Inference is deliberately shallow: constants, bare series builtins (`close`, …), `input.*` constructors, and `ta` / `math` / `str` / `color` calls whose metadata `detail` contains a `→` return type. Ambiguous operators are skipped rather than guessed wrong.

Conceptual model

```
flowchart TD
  PARSE[parse source] --> WALK[walk Assign nodes]
  WALK --> CHK{simple Name target, no type, has value?}
  CHK -->|no| SKIP[skip]
  CHK -->|yes| INF[_infer_type RHS]
  INF -->|const / series / input / → type| HINT[": type_label" at end of name]
  INF -->|None| SKIP
```

Example mental model (from module docstring):

```
length = 14                 →  length: const int
rsi = ta.rsi(close, 14)     →  rsi: series float   (from metadata detail)
n = input.int(5, "n")       →  n: input int
```

Interface surface

- Method: `textDocument/inlayHint`
- Capability: `InlayHintOptions(resolve_provider=False)`
- Handler: `features/inlay_hints.py` → `handle_inlay_hints`

Each hint:

Field	Value
`position`	End of target identifier (`lineno-1` , `col_offset + len(id)`)
`label`	`": {type_label}"`
`kind`	`InlayHintKind.Type`
`tooltip`	`Inferred type: {type_label}`

Returns `[]` for empty source, `None` if parse fails (client typically treats as no hints).

Internals

Constant mapping

Python value	Label
`bool`	`const bool`
`int`	`const int`
`float`	`const float`
`str`	`const string`

Builtin bare names

`open` , `high` , `low` , `close` , `volume` , `hl2` , `hlc3` , `ohlc4` → `series float` ; time / bar index vars → `series int` ; `na` → `na` .

Calls

- `input.{int,float,bool,string,color,symbol,session,source,time,timeframe,price}` → `input {attr}`
- `ta|math|str|color.{attr}` → parse `detail` after `→`, else fallback `"series"`

Walk

Recursive over `_fields` (same pattern as workspace symbol collection) — not limited to script top-level; nested function assigns can receive hints.

Path	Role
`features/inlay_hints.py`	Collection + inference
`providers/builtin_metadata.py`	Return-type detail for modules

Invariants and edge cases

1. **Explicit `name: type = …` suppresses hints** (`node.type is not None`).
2. **Tuple / attribute targets** are ignored.
3. **BinOp / Compare / Conditional** return `None` — no speculative arithmetic types.
4. **Metadata quality bounds accuracy** — missing `→` in detail yields coarse `series`.
5. **No resolve step** — tooltips are final.

Worked example

```
//@version=5
indicator("hints")
length = 14
src = close
avg = ta.sma(src, length)
period = input.int(20, "Period")
```

Expected hints (when parse + metadata succeed):

- `length` → `: const int`
- `src` → `: series float`
- `avg` → type from `ta.sma` detail (typically series float) or `series`
- `period` → `: input int`

`avg` will not get a hint derived from `length`'s type through the call graph — only the call's known return shape.

Failure modes

Symptom	Cause
No hints on file	Parse error → `None`
`ta.*` always `: series`	Metadata detail lacks `→` segment
Hints mid-identifier	Incorrect `col_offset` on AST node — builder issue

See also

- Completion
- Builtin metadata
- Type system

BUILTIN METADATA

title: "Builtin Metadata" description: "JSON catalog for LSP completion and hover; generation pipeline; Fernet encryption, CRYPTO_KEY / METADATA_KEY, and decrypt path."

BUILTIN METADATA

Abstract

Builtin metadata is the **LSP documentation corpus**: a map from fully qualified names (`ta.sma`, `strategy.entry`, ...) to labels, signatures, briefs, snippets, and categories. It is generated from code (not hand-edited as source of truth), loaded by the language server for completion/hover/inlay, and optionally **Fernet-encrypted** for Nuitka onefile distribution so casual string extraction does not dump the full catalog.

This is obscurity for distribution hygiene, not a cryptographic access-control boundary.

Conceptual model

```
flowchart TB
  BUILTINS[evaluator builtins + generator] --> GEN[scripts/generate_builtin_metadata.py]
  GEN --> JSON[providers/builtin_metadata.json]
  JSON --> SHA[builtin_metadata.json.sha256]
  JSON --> ENC[Fernet encrypt]
  KEY[.metadata.key / CRYPTO_KEY] --> ENC
  ENC --> ENCFILE[builtin_metadata.json.enc]
  subgraph Runtime
    LOAD[get_metadata]
    LOAD -->|dev| JSON
    LOAD -->|binary| DECRYPT[metadata_decrypt]
    KEY --> DECRYPT
    ENCFILE --> DECRYPT
    SHA --> DECRYPT
  end
```

Interface surface (metadata record)

Typical entry in `builtin_metadata.json`:

```
{
  "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"
  }
}
```

Categories drive completion headers (`ta.technical_analysis` → “Technical Analysis (ta.*)”, `builtin` → “Built-in Variables”, etc.).

Loader API

`providers/builtin_metadata.py`:

Function	Behavior
`get_metadata()`	Singleton cache; prefer `.enc` decrypt, else plaintext JSON, else `{}`
`get_builtin(name)`	Single entry or `None`
`get_builtins_by_category` / `get_all_categories`	Category queries
`fuzzy_filter`	Scored substring match for completion

Consumers

- Completion list / module / resolve
- Hover cards
- Inlay return-type extraction from `detail` after `→`

Internals

Path	Role
`scripts/generate_builtin_metadata.py`	Introspect builtins → JSON
`src/pynescript/langserver/providers/builtin_metadata.json`	Dev plaintext
`…/builtin_metadata.json.enc`	Encrypted blob for release
`…/builtin_metadata.json.sha256`	First 16 hex chars of SHA-256 of **plaintext**
`…/metadata_decrypt.py`	Fernet load + integrity check
`scripts/build/compile.py`	`encrypt_metadata()`, Nuitka include of providers dir
`scripts/build/ci_build.py`	`stage_metadata()` for CI
`scripts/build/.metadata.key`	**Gitignored** Fernet key material

Generation

After adding evaluator builtins:

```
python scripts/generate_builtin_metadata.py
```

Do not hand-edit the JSON as the long-term source of truth; regenerate from code.

Encryption (build)

`scripts/build/compile.py` → `encrypt_metadata()`:

1. Require `cryptography` and plaintext JSON.
2. Generate Fernet key → write `scripts/build/.metadata.key` (mode `0o600`).
3. Encrypt → `builtin_metadata.json.enc`.
4. Write truncated SHA-256 of plaintext to `.sha256`.

Local `encrypt_metadata` currently **always generates a new key** when invoked. For **byte-stable** encrypted artifacts across CI runs, supply a stable secret:

Context	Mechanism
GitHub Actions	`secrets.METADATA_KEY` → env (documented as `CRYPTO_KEY` in workflows / AGENTS)
Cloud Build	substitution `${_METADATA_KEY}` → `CRYPTO_KEY`
Runtime decrypt	File `.metadata.key` next to providers **or** env `PYNESCRIPT_METADATA_KEY`

Without a stable key, each CI encrypt produces a different `.enc` blob — not a functional bug, but noisy diffs and cache misses.

Decrypt path

`metadata_decrypt.py`:

1. Resolve key: PyInstaller/Nuitka data dir `.metadata.key`, else providers dir, else `PYNESCRIPT_METADATA_KEY`.
2. Fernet-decrypt `.enc`.
3. If `.sha256` present, compare `sha256(plaintext).hexdigest()[:16]`.
4. `json.loads` → dict.

`get_metadata_cached()` prefers **plaintext** when both exist (developer-friendly).

`get_metadata()` in `builtin_metadata.py`: if `.enc` exists, try decrypt cache; on failure empty dict; elif plaintext, load it.

Invariants and edge cases

1. **Regenerate after builtin changes** or completion/hover/inlay drift.
2. **Empty dict on failure** fails soft (no completions) rather than crashing the server.
3. **Integrity mismatch** raises in decrypt — compiled binary should not silently serve corrupt docs.
4. **Key not in git** — never commit `.metadata.key`.
5. **Fernet is symmetric** — anyone with the binary + key material can recover plaintext; design goal is casual reverse-engineering friction and future paid-metadata swap hooks.

Worked example — local dev

```
python scripts/generate_builtin_metadata.py
# edit-free check
python -c "from pynescript.langserver.providers.builtin_metadata import get_builtin; print(get_buil
```

Worked example — encrypted binary path

```
# build (encrypts + Nuitka) — see DevOps
python scripts/build/compile.py

# or embed .metadata.key in the onefile data dir during compile
```

Failure modes

Symptom	Cause
Completions empty in release binary	Missing key / wrong `PYNESCRIPT_METADATA_KEY`
`Metadata integrity check failed`	`.enc` and `.sha256` out of sync
CI churn on `.enc`	New random key each build without `CRYPTO_KEY` / `METADATA_KEY`
Stale docs for new `ta.*`	Forgot to re-run generator

See also

- Metadata crypto (DevOps)
- Nuitka build
- Completion
- Hover

VSCODE EXTENSION

title: "VS Code Extension" description: "PYNE — Pine Script™ for VS Code: HOOX branding, TextMate grammar, LanguageClient wiring, settings, and VSIX build."

VS CODE EXTENSION

Abstract

PYNE — Pine Script™ for VS Code is a **LanguageClient host** plus a rich TextMate grammar, branded as part of the **HOOX open trading stack**. It does not reimplement diagnostics or completion in TypeScript; it spawns `pynescript-lsp` (or a configured command) over STDIO via `vscode-languageclient`, maps `*.pine` / `*.pinev5` / `*.pinev6` to language id `pinescript`, and exposes a small settings surface for enablement and feature toggles.

> Pine Script™ and TradingView® are trademarks of TradingView, Inc. This project is independent and is not affiliated with or endorsed by TradingView, Inc.

Conceptual model

```
flowchart LR
  VS[VS Code] --> EXT[extension.ts activate]
  EXT --> LC[LanguageClient]
  LC -->|stdio| LSP[pynescript-lsp]
  EXT --> TM[TextMate grammar]
  TM --> HL[Token colors]
  LSP --> DIAG[Diagnostics / completion / …]
```

Interface surface

Package identity

From `vscode-extension/package.json`:

Field	Value
name	`pyne`
displayName	`PYNE — Pine Script™ for VS Code`
icon	`media/icon.png` (HOOX mark)
engines.vscode	`^1.85.0`
main	`./out/extension.js`
dependency	`vscode-languageclient` ^9
galleryBanner	`#050505` dark

Language contribution

- **id:** `pinescript`
- **aliases:** `Pine Script™`, `Pine Script`, `pinescript`, `Pine`
- **extensions:** `.pine`, `.pinev5`, `.pinev6`
- **configuration:** `language-configuration.json`
- **grammar:** `syntaxes/pinescript.tmLanguage.json` (`scopeName: source.pinescript`) — namespaces, annotations, hex colors, UDT/enum, multiline strings, plot/strategy builtins

Activation

```
onLanguage:pinescript
workspaceContains:**/*.pine
workspaceContains:**/*.pinev5
workspaceContains:**/*.pinev6
```

Settings (`pynescript.*`)

Key	Default	Meaning
`lsp.enabled`	`true`	Skip activation when false
`lsp.command`	`auto`	Server executable / auto-detect
`lsp.python`	`python3`	Interpreter for module launch
`lsp.args`	`[]`	Extra server args
`formatting.enabled`	`true`	Passed as init option
`diagnostics.enabled`	`true`	Passed as init option
`completion.snippets`	`true`	Passed as init option

Initialization options object:

```
{
  "formattingEnabled": true,
  "snippetsEnabled": true,
  "diagnosticsEnabled": true
}
```

Commands

Command	Action
`pynescript.restartServer`	`client.stop()` then `client.start()`
`pynescript.formatDocument`	`editor.action.formatDocument`

Client options

- Document selector: `pinescript` for `file` and `untitled` schemes
- File watcher: `**/*.pine` (not automatically the v5/v6 suffixes)
- Diagnostic collection name: `pynescript`

Internals

Path	Role
`vscode-extension/src/extension.ts`	Activate / deactivate / client
`vscode-extension/package.json`	Contributes + scripts
`vscode-extension/syntaxes/pinescript.tmLanguage.json`	Grammar
`vscode-extension/language-configuration.json`	Brackets / comments
`out/extension.js`	Compiled JS

Server launch:

```
const serverOptions: ServerOptions = {
  command: lspCommand,           // default pynescript-lsp
  args: ['--parent-dir', scriptPath],
  transport: TransportKind.stdio,
  options: { env: { ...process.env, PYTHONPATH: scriptPath } },
};
```

`scriptPath` is resolved to `../src/pynescript/langserver` relative to the extension for monorepo development. Release packaging typically expects `pynescript-lsp` on `PATH` (pip or Nuitka binary).

Build

```
cd vscode-extension
npm ci
npm run compile          # tsc → out/
npx vsce package         # VSIX
# or monorepo:
make build-vscode
```

Requires Node 22 tooling as noted in project agents docs for packaging.

Invariants and edge cases

1. **`lsp.enabled === false` short-circuits activate** — no client, no restart command registration beyond early return (commands not registered).
2. **Binary vs module:** production users should install `pynescript[lsp]` or the Nuitka onefile and set `pynescript.lsp.command` if not on PATH.
3. **Grammar works without LSP** — open a `.pine` file offline and still get TextMate highlighting.
4. File watcher only lists `**/*.pine`; v5/v6 still activate via language id on open.

Worked example — extension development

```
pip install -e ".[lsp]"
cd vscode-extension && npm ci && npm run compile
# Launch Extension Development Host pointing at vscode-extension
```

Open a fixture `.pine` file; confirm Problems panel shows parse/lint diagnostics from the Python server.

Failure modes

Symptom	Cause
“Language client failed to start”	`pynescript-lsp` missing / wrong command
No squiggles, highlighting OK	LSP disabled or server crash; check Output → Pine Script Language Server
Restart does nothing	Client never started (`lsp.enabled` false)

See also

- Clients — non-VS Code hosts
- Architecture
- Nuitka build
- Editors guide

CLIENTS

title: "Editor Clients" description: "Wire pynescript-lsp into Neovim, Zed, Emacs, Helix, Sublime, and generic LSP hosts."

EDITOR CLIENTS

Abstract

Any editor that speaks LSP over STDIO can host PYNE intelligence. First-class snippets live in `clients/` (Neovim Lua, Zed JSON, Emacs Lisp) plus documented Helix/Sublime fragments. The VS Code path is a dedicated extension (vscode-extension); this page covers the rest.

Prerequisite: `pynescript-lsp` on `PATH` (Python 3.10+):

```
pip install "hoox-pyne[lsp]"
# development
pip install -e ".[lsp]"
```

Conceptual model

```
flowchart LR
  subgraph Hosts
    NV[Neovim]
    ZD[Zed]
    EM[Emacs]
    HX[Helix]
    SB[Sublime LSP]
  end
  NV --> STDIO
  ZD --> STDIO
  EM --> STDIO
  HX --> STDIO
  SB --> STDIO
  STDIO[pynescript-lsp stdio] --> FEAT[diagnostics completion hover …]
```

Interface surface — repo files

File	Editor
`clients/neovim.lua`	Neovim (`nvim-lspconfig` or manual)
`clients/zed.json`	Zed `settings.json` fragment
`clients/emacs.el`	Emacs `lsp-mode` + `pinescript-mode`
`clients/README.md`	Human-oriented install notes
`vscode-extension/`	VS Code / Cursor-compatible

Neovim

Return table from `clients/neovim.lua`:

```
return {
  cmd = { 'pynescript-lsp' },
  filetypes = { 'pinescript' },
  root_dir = function(fname)
    return vim.fs.root(fname, { '.git', '*.pine', '*.pinev5', '*.pinev6', 'pyproject.toml' })
      or vim.fn.getcwd()
  end,
  settings = {
    pinescript = {
      formatting = { enabled = true },
      diagnostics = { enabled = true },
      completion = { snippets = true },
    },
  },
}
```

Suggested maps when attaching: `gd` definition, `gr` references, `K` hover, format via `vim.lsp.buf.format`.

Register filetype if needed:

```
au BufRead,BufNewFile *.pine,*.pinev5,*.pinev6 setfiletype pinescript
```

Zed

Merge `clients/zed.json` into `~/.config/zed/settings.json`:

```
{
  "languages": {
    "Pine Script": {
      "language_servers": ["pynescript"]
    }
  },
  "language_servers": {
    "pynescript": {
      "command": "pynescript-lsp",
      "arguments": ["--stdio"],
      "languages": ["Pine Script"]
    }
  }
}
```

Emacs

`clients/emacs.el` registers an `lsp-mode` client:

```
(lsp-register-client
 (make-lsp-client
  :new-connection (lsp-stdio-connection '("pynescript-lsp" "--stdio"))
  :major-modes '(pinescript-mode)
  :server-id 'pynescript))
```

Defines a minimal `pinescript-mode` with font-lock keywords and `auto-mode-alist` for `\\.pine\\'` / `\\.pinev[0-9]+\\'`. Optional keys: `C-c C-c` format, `M-.` definition, `M-?` references.

Helix

`~/.config/helix/languages.toml`:

```
[[language]]
name = "pinescript"
scope = "source.pinescript"
file-types = ["pine", "pinev5", "pinev6"]
roots = ["pyproject.toml"]
command = "pynescript-lsp"
args = ["--stdio"]
```

Sublime Text

With Package Control **LSP**:

```
{
  "clients": {
    "pynescript": {
      "command": ["pynescript-lsp", "--stdio"],
      "selector": "source.pinescript",
      "initializationOptions": {}
    }
  }
}
```

Generic / Cursor

```
pynescript-lsp
# or explicitly:
pynescript-lsp --stdio
```

Point the host's "external language server" command at that binary. Cursor users can also load the VS Code extension in compatible mode when packaged as VSIX.

Internals

Server entry always uses STDIO (`start_io` in `__main__.py`). Capability negotiation is identical across hosts — differences are client UI only (how inlay hints render, whether pull diagnostics are used, etc.).

Tests: `tests/test_lsp_features.py` (handlers with fake params), `tests/test_langserver.py` (e2e with pytest-asyncio) — not editor-specific.

Invariants and edge cases

1. **Language id** should be `pinescript` for VS Code parity; some editors invent their own names — map filetypes carefully.
2. **Root directory** heuristics vary; wrong root rarely breaks single-file Pine editing (workspace is URI-keyed open buffers).
3. **Settings keys** under `pinescript` in Neovim are conventional; the Python server does not currently enforce a rich `workspace/configuration` schema for all of them.
4. Nuitka onefile can replace the module entry for air-gapped machines — same command name if installed on PATH.

Failure modes

Symptom	Cause
Server exits immediately	Missing pygls / incomplete install `.[lsp]`
Filetype never attaches	Extension not mapped to `pinescript`
Features missing in Helix	Older Helix without inlay / semantic support — grammar still useful

See also

- VS Code extension
- Architecture
- End-user editors guide