Guidedog Discussions GDS 0010

Typed build plugins

STATE
committed
TYPE
Standards Track
AUTHORS
Vikrant Rathore
CREATED
2026-10-13
UPDATED
2026-10-15
DISCUSSION
No external discussion URL assigned
LABELS
architecture, library, cli, documentation

Record status

The catalog defines the review process and the meaning of each state. This numbered record forms part of the catalog.

Contents

1. Abstract 1
2. Motivation and scope 1
3. Specification 1
4. Rationale and alternatives 4
5. Security and operational considerations 4
6. Implementation and verification 4
7. Open questions 4
8. References 4

1. Abstract

Extend Guidedog with typed source, HTML and static-asset services. Keep the released publisher API and wire protocol intact. A normal build may invoke only build services selected explicitly in project configuration. Publication remains a separate operation after a successful, complete build.

2. Motivation and scope

GDS 0009 delivered a publisher interface. That interface cannot extend document processing. Calling it a general plugin tutorial concealed this boundary and forced readers to learn deployment before writing their first extension.

Provide a small, useful build interface without exposing mutable syntax-tree internals, injecting arbitrary events, or running scripting interpreters. This record does not introduce installed directives, roles, custom builders or a dynamic-library ABI. Compiled reader and renderer registries remain available.

3. Specification

A build plugin implements the `BuildPlugin` trait of `gd-plugins`: `info` (identity, product version, interface version and the hooks it offers), an optional `load`, and any of `process_source`, `process_html` and `collect_assets`.

A default method means “service absent”. At least one build service is required, unless the plugin only adds commands. `Plugin::build` wraps it; a publisher is wrapped by `Plugin::publisher` instead. A `Plugin` is build hooks or a publisher, never both.

`Registry::new` validates explicit plugins. A program links them into `guidedog` through `gd_cli::run_with(Registry::new(vec![badge::plugin()])?)`, and the project receives them in `LoadOptions::plugins`. Installed programs serve the same value through `gd_plugins::serve_plugin` and the bounded JSON Lines transport. A build looks up a selected ID in the linked registry first, then in the machine registry (`select_build_plugin`). Neither route discovers or executes code merely because a descriptor exists.

`conf.toml` selects IDs in `plugins` and their configuration paths in `[plugin_configs]`. Paths resolve from the project configuration directory. Hooks run in listed order. `plugin_timeout` defaults to 60 seconds and accepts 1 through 86400 seconds per whole installed-program request. An untrusted build refuses native plugins before starting them.

3.1. Lifecycle and ownership

Each build has private sessions. Load copies options, establishes private state, and declares dependencies. The host serializes callback access across reading workers. A plugin need not implement synchronization to use this API. A native callback is trusted code. A pipe deadline cannot forcibly interrupt it.

`PluginSession::load` opens a session, and dropping it ends the session. `process_source` and `process_html` return text, and `collect_assets` returns assets; every result is an owned value. Inputs are borrowed for the call. Session state must not retain borrowed inputs. Dropping a session closes its resources, also after a failed load.

Primary RST and Markdown sources pass through source hooks before parsing. Includes are still read by the document engine. Source transformations affect all builders. When the source changes, root-document markup diagnostics identify plugin-expanded input and retain its real excerpt. They do not claim original source coordinates. A future origin-map interface would require a separate design.

HTML hooks receive complete rendered document, index and additional pages. Their asset URL is relative to the actual output page. They execute before staging. Assets are generated once per HTML build and placed under `_static/plugins/ID/`. PDF does not promise a plugin-generated resource service. The transformed source may use ordinary project resources already supported by the PDF engine.

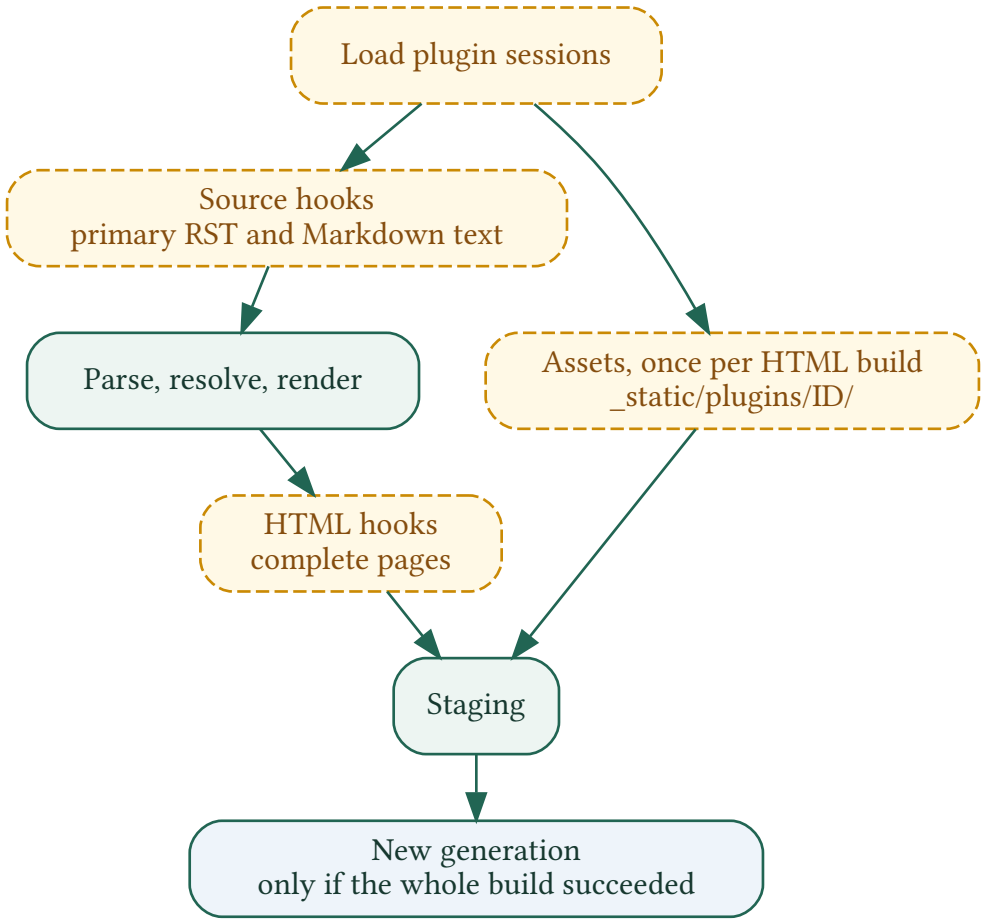

Fig. 12 Build hooks sit at the edges of a build; publication comes after it.

3.2. Limits and failure semantics

Text input and replacement must be valid UTF-8 and at most 8 MiB. Metadata and paths have explicit bounds. A plugin may return at most 256 assets totalling 16 MiB. Asset names must be portable, relative paths: no traversal, drives, backslashes, control bytes or reserved Git metadata. Reject duplicates, case-insensitive aliases and file-as-parent conflicts before copying results. Refuse collisions with files supplied by the project instead of overwriting them.

What the host reads for a plugin uses the existing build budget. Child RSS is outside that budget. No operating-system sandbox is promised. Hook text is `Cow`, so unchanged text is never copied. A hook never turns a failure into partial success.

Every failure carries a title, context, explanation and concrete correction. Malformed plugin problems are contract errors. Timeouts before a completed build are build failures, not uncertain remote publication outcomes. Failed hooks, markup rejected by -W, budget refusal and interrupted builds abandon staging and preserve the complete previous generation.

3.3. Cache identity

Each plugin contributes its identity, product version, executable SHA-256 (for a linked plugin, the SHA-256 of its ID, versions and the host's build identity), loaded configuration, declared dependency files, `cache_key`, builder, language and asset bytes to the build's cache key.

`read_plugin_config` and `read_plugin_file` declare files and append the fingerprint of the exact loaded bytes to `cache_key`. This separates loaded state from a later file change. Hashing only the on-disk file after load could incorrectly pair old state with new content. A callback that reads files itself must declare their paths and append fingerprints of the values actually loaded. Other inputs, including selected environment variables, also belong in `cache_key`. Do not erase SDK fingerprints. State remains fixed for a build.

3.4. Version negotiation

Build-plugin API 1.0 and process protocol `guidedog-build-plugin` 1.0 are new contracts. The handshake declares exactly the offered source/html/assets hooks. The host checks identity, hook set and compatible versions when starting a pinned program. JSON requests have strictly increasing IDs. Session load precedes hooks. Unload is bounded and terminates the process.

`check_plugin` exercises two fresh sessions, repeated source/HTML values and asset validation. The CLI's `plugin check PATH` adapts only an isolated process and does not alter the registry. Build-plugin registration runs these checks before pinning a checksum, with project/configuration supplied explicitly. Passing is interoperability evidence, not certification. Every runtime response is still checked. Publisher qualification uses provider tests and dry runs rather than generic remote operations.

Publisher SDK 1.1, `guidedog-publisher` 1.0 and snapshot schema 1 retain their released contracts. Machine registry schema 1 adds optional protocol-name and hook fields only for build programs. Legacy publisher entries remain readable. Old hosts reject the distinct build protocol. Compatibility is not simulated by advertising a publishing capability.

4. Rationale and alternatives

A trait keeps inputs, ownership and available services visible. The same package can be linked into an application or served by a small executable wrapper. There is no reflection, inheritance, global callback registry or event bus. Explicit source and output boundaries minimize coupling to parser internals.

An arbitrary mutable AST interface would couple extensions to internal node layout, reference resolution and ownership. A dynamic-library ABI would add compiler and platform compatibility contracts without improving this first extension. Neither is justified by the worked example.

5. Security and operational considerations

Build hooks execute trusted native code. Checksums detect changed programs. They do not prove the programs safe. Selection is deliberate, untrusted mode disables them, and no publication credentials or authority are handed to build contexts.

6. Implementation and verification

The implementation is the `BuildPlugin` trait of `gd-plugins` and the plugin run of a build in `gd-env`. `crates/gd-plugins/examples/build_plugin` (the badge) demonstrates bounded configuration, source expansion, HTML output, a generated CSS asset, and a `guidedog badge preview` command. Its standalone `main` contains only protocol dispatch. The manual follows that exact source.

`LoadContext::read_config` and `read_file` extend the cache key with the digests of the bytes read, and only append. Each plugin has one session per build, shared by reading workers behind a `Mutex`. `--untrusted` refuses any selected plugin before code starts. The limits above stay, with configuration files at 1 MiB, 256 dependencies, and build requests timing out after 60 s.

SDK tests cover separate services, owned results and portable asset conflicts. Integration tests (`tests/integration/tests/tools/plugins.rs`) build the example, use a private machine registry, and exercise builders, cache changes, nested asset links, untrusted mode and preservation of failed generations. The badge runs linked and served in a test that requires identical outputs. Build hooks first ship in 0.6.2. Earlier 0.6.x releases contain publishers only.

A plugin of either kind may add commands to `guidedog`'s command line; GDS 0009 ("Command contributions") describes the interface.

7. Open questions

Origin maps, typed directive/role registration and installed custom builders need separate proposals with their own cache, ownership and diagnostic contracts. The maintainer accepted the implemented scope on 14 October 2026. See Plugin design acceptance, 14 October 2026 for the decision and qualification evidence.

8. References

- Native plugin API and registry defines the released publisher contract.

- Publish documentation to GitHub Pages defines the first remote publisher.
- `crates/gd-plugins` contains the typed SDK and transport.
- `tests/integration/tests/tools/plugins.rs` exercises the complete installed build plugin.

Lifecycle event

```
2026-10-13: prediscussion -> discussion. Assigned a permanent number and
opened for discussion.
```

Lifecycle event

```
2026-10-14: discussion -> accepted. Maintainer requested acceptance after
review of the implemented 0.6.2 scope.
```

Lifecycle event

```
2026-10-14: accepted -> committed. Implemented scope reviewed against source
and published qualification.
docs/gds/evidence/plugin-acceptance-20261014.rst; successful 0.6.2 release
run 38434657070.
```