RFC 324 Approved: Intelligent reorganization of code intelligence code

Editor: roux@sourcegraph.com
Status: Review
Requested reviewers (please review by EOD 2021-02-22): TJ Devries
Approvals: Olaf Geirsson thorsten@sourcegraph.com Eric Fritz , Noah Santschi-Cooney
Team: Code Intel

1.0 Background / Problem

Code intelligence code has been slowly growing over time, and we've historically initiated code cleanup/reuse efforts on an adhoc/as needed basis. Questions like "where should I put this code intel related command line utility?", "what if I want to let customers use it?", or "where should I move this code so I can re-use it in multiple packages/repos?" don't have good answers.

The particular motivation for creating this RFC is wanting to create tooling to semantically validate, compare, and diff LSIF indexes. A lot of the necessary code already exists in the sourcegraph repo, but it's coupled heavily to the server. We've already reimplemented some of the logic in lsif-test for basic validation, but that's duplicated logic now. There's no path forward without rewriting a lot more logic, or moving a lot of code out of internal libraries in the sourcegraph repo.

1.1 Repos

The following table shows which repos code intelligence code lives in, and what code lives in those repos. By my count there are at least 67 such repos.

The Sourcegraph Sphere	
sourcegraph	The big cheese.
src-cli	The small cheese.
code-intel-extensions	Client code for fetching and munging code intel data
codeintelutils	Common library code. Right now just LSIF upload logic, used by src-cli and sourcegraph.
go-lsp	Go LSP protocol definitions, used by go-langserver, and in a few places in sourcegraph (not sure if critical?).
The LSIF sphere	
lsif-go, lsif-node, lsif-java, lsif-clang, lsif-semanticdb	Actively developed / maintained LSIF indexers
lsif-protocol	Go LSIF protocol definitions, used by lsif-go, lsif-semanticdb, and

	sourcegraph.
lsif-test	CLI utilities.
Misc maybe used?	
lsif-java-action, lsif-go-action, lsif-node-action, lsif-upload-action, lsif-dart-action	GitHub actions
bazel-compilation-database	Our fork of GrailBIO's bazel-compilation-database code
Seems unused?	
coif-to-lsif, merlin-to-coif	seems to be a one-off experiment with a simpler format than LSIF
go-langserver, java-langserver, javascript-typescript-langserver, langserver, OLD-java-langserver, vscode-javascript-typescript, python-langserver, javascript-typescript-buildserver, python-language-server, language-server-protocol, java-langserver-integration-tests, typescript-language-server, sourcegraph-langserver-http, java-langserver-docs, vscode-languageserver-node, swift-langserver, sublime-lsp, emacs-lsp, css-langserver, lsp-client	LSP related things
lsif-demos, sample-public-go-repo	Demo projects to run LSIF indexers on.
lsif-node-fork	A fork of lsif-node which still tracks lsif-node unlike our fork
lsif-semanticdb-build-tooling	Not sure exactly what this is for but it seems abandoned.
sourcegraph-graphql, sourcegraph-go, sourcegraph-python, lang-ruby	Superseded by code-intel-extensions.
old-lsif-dart-research	Seems old.
14 srclib-related things (got tired of copying links).	Abandoned project.

2.0 Proposal

2.1 Archive all the things

Archive everything under “seems unused?”

2.2 Create a lib module in sourcegraph/sourcegraph

Create a new `enterprise/lib` directory at the root of sourcegraph/sourcegraph with its own go.mod file. Code which is intended for reuse outside of sourcegraph/sourcegraph can live here, and sourcegraph/sourcegraph can require it directly in-tree. Code in `enterprise/lib` should *never* import code from the root module, and this is currently impossible by circumstance because the replace directives in the root module’s go.mod make it impossible to import or install. External repos similarly would only be able to import from `enterprise/lib` but not from the root sourcegraph module. So we would be effectively introducing a new dependency, usable by sourcegraph/sourcegraph and external repos, but versioned alongside sourcegraph to reduce development friction. We should document the usage of different folders in enterprise in a top level README.

CLI tools that we want to be easily accessible outside of sourcegraph/sourcegraph would live in `enterprise/lib/cmd` so they can be directly ‘go install’d.

It might also be desirable to have a top level `lib` module for OSS licensed code we want to expose to other repos, but I’m not sure how to enforce that only enterprise can import from lib but not vice versa, which is definitely something we want to enforce. None of the code that would be moved right now would live there so it’s out of scope for the RFC, but (please correct me on this, not sure) we also don’t do anything to enforce that relationship with the root sourcegraph module so it wouldn’t be a regression to just do it.

2.3 Consolidate code intel code

codeintelutils, lsif-protocol, go-lsp, lsif-test, and the enterprise/internal/codeintel/lsif should all move to `enterprise/lib`. Other repos which depend on this code can instead depend on sourcegraph/sourcegraph. After this, the only (non-indexer) repos we’ll need to work with are sourcegraph/sourcegraph, src-cli, and

code-intel-extensions. (Would be nice to collapse code-intel-extensions into sourcegraph/sourcegraph as well, but I have no idea what the implications of that would be so it's out of scope for this RFC).

2.4 Code restructuring

At this point it would also become easy to liberate generally useful code from internal packages so we can use it for other cool things. For example, if we factored out types and functions from lsifstore we could easily build an in-memory code intel DB over Go structs which would be really useful for testing lsif indexers. We should also rename lsifstore because the types in it are not coupled to LSIF, LSIF is just the only currently supported input format for our internal representation.

3.0 Definition of success

- All code intel related repos which are not developed or maintained are archived.
- The number of repos storing code intel logic (minus lsif indexers) is at most 2.
- There is a canonical and ergonomic place to store logic and CLI tools so that they can be reused anywhere code intel needs to happen.

4.0 Historical proposal

This was the original proposal section where I suggested several options to generate discussion.

4.1 Process for deprecating repos

1. Archive the repo: This is a nondestructive action which clearly signals the repo is unmaintained, and we can do this as soon as we decide the repo is deprecated. We should put a timeline'd deprecation notice on the repo stating it will be deleted in 1 year. If it still has known users, we should communicate with them.
2. (Optional) After a year, we make the repo private for a year. This is a destructive but reversible action, and provides a grace period where we can confirm that the repo is unused before:
3. (Optional) After a year, remove the repo from the Sourcegraph org by either deleting it, or create a Sourcegraph archive org to move it to if we want to keep everything for historical interest.

Step 1 is the most/only important one, 2 and 3 are just to keep things tidy but we totally don't have to do them. I'd propose everything under the "Seems unused?" header in the table above be archived immediately.

4.2 Code reuse

I propose to collapse codeintelutils, lsif-protocol, go-lsp, and the enterprise/internal/codeintel/lsif folder in sourcegraph into one repo. I can't see any particular reason the development of any of the non-sourcegraph logic needs development to be decoupled. The codeintel logic in sourcegraph won't need to be decoupled if we go with Option 1:

4.2.1 Option 1: Move everything into sourcegraph (**Proposed**)

Create a new lib/codeintel folder at the root of the sourcegraph repo and store all of the above library logic in there. This repo is not currently imported into other repos for use as a library, and this change would require at least src-cli and some LSIF indexers to do so. But it would also simplify common workflows and allow most dev to happen in one repo which would be really nice. There's also some logic in e.g. codeintelutils which is very tightly coupled to the server (e.g. upload endpoints), and it would make a lot more sense to have those in the same source tree.

4.2.2 Option 2: Have a single code intel library repo

Create a new sourcegraph/libcodeintel (name up for debate) to repo and store all of the above logic there. This seems undesirable since a lot of logic that's very important to the server would be versioned in a separate repo, but I'm presenting it as basically the only alternative to keeping everything in sourcegraph.

4.3 Code restructuring

We can also move the code in enterprise/internal/codeintel/stores/lsifstore out of sourcegraph internal packages if we abstract over the backing store. This would be useful for building in-memory code intel DBs that we could build an LSP on top of and use for local development and testing. We should also rename lsifstore

because the types in it are not coupled to LSIF, LSIF is just the only currently supported input format for our internal representation.

4.4 Where to put CLI tools

4.4.1 Option 1: Internal tools in sourcegraph (**Proposed**)

This is preferred for internal tools if we move all of the code into sourcegraph so everything can live together. In this case they would likely go in cmd/codientel or enterprise/cmd/codeintel so they can still be accessible by LSIF indexers wanting to use them for testing.

4.4.2 Option 2: Internal tools in a separate repo

This is preferred if we move all of the code into a separate repo (the same one as the CLI tools) for the same reasons.

4.4.3 Option 3: Deliver tools in src-cli (**Proposed**)

This seems like the best place to put tools we actually want to deliver to users. There's already a lot of infrastructure around building and distributing the tool, users won't have to maintain installs of several different sourcegraph related tools, and I just think it'd be neat.

4.4.4 Option 4: Internal tools in src-cli

The line between internal and external tools is blurry, and I could see us developing something for internal use and then wanting to let a customer use it for debugging if there's a really weird language bug. An option here *could* be to include tools in src-cli but have them hidden or documented separately as dev/experimental. But this seems premature and provides limited benefit given that it would by default introduce cross-repo dependency for new tools.