

SCIP File Format and Indexes (Detailed Overview)

What is a SCIP index?

SCIP (SCIP Code Intelligence Protocol) is a binary encoding for code-index data. Sourcegraph created it to replace the JSON-based LSIF format. SCIP is designed to be an efficient transmission format (not an on-disk database) for delivering semantic information from an *indexer* to a consumer such as Sourcegraph's code-navigation backend or an LLM. Each `.scip` file contains exactly one `scip.Index` message encoded using [Protocol Buffers 3]. Protobuf gives SCIP several advantages over LSIF:

- A compact TLV-based binary encoding that can be streamed; tools can read/write documents one at a time.
- A machine-readable schema makes it easier to generate type-safe bindings in many languages and to validate or upgrade indexes.
- Human-readable string identifiers are used for symbols instead of numeric IDs, improving debuggability and reducing the blast radius of bugs ¹.
- Producers can emit multiple partial `.scip` files and later concatenate them to form a complete index ².

A typical workflow with Rust uses the `rust-analyzer` CLI. Running `rust-analyzer scip --path <crate-dir>` invokes the Rust compiler front-end, computes semantic information and writes an `index.scip` file. The file is a raw protobuf message (it is not zipped by default). You can inspect it using the protobuf compiler:

```
# assuming scip.proto and index.scip are in the current directory
protoc --decode=scip.Index scip.proto < index.scip > decoded.txt
```

This command decodes the binary file into a human-readable representation ³. Tools such as the `scip` CLI also provide `snapshot` to pretty-print occurrences back into source code ⁴.

Top-level structure: Index

The root message is `scip.Index`. An index consists of metadata, a list of documents and optional external symbols:

- `metadata` – information about the index: protocol version, tool info, project root and text encoding ⁵.
- `documents[]` – an array of `scip.Document` messages, one per source file ⁶.
- `external_symbols[]` – a list of `SymbolInformation` entries for symbols referenced from external packages. This allows the index to supply hover documentation for dependencies that are not indexed alongside the current project ⁷.

Metadata

The `Metadata` message contains:

Field	Purpose
<code>version</code>	Indicates the SCIP protocol version ⁵ . Currently only <code>UnspecifiedProtocolVersion</code> exists; future versions may add features.
<code>tool_info.name</code> , <code>tool_info.version</code> , <code>tool_info.arguments[]</code>	Identify the indexer tool (e.g., <code>rust-analyzer</code> and its version) and how it was invoked ⁵ .
<code>project_root</code>	URI-encoded absolute path to the root of the project being indexed ⁵ . All document paths are relative to this directory.
<code>text_document_encoding</code>	Encoding of the source files on disk (UTF-8, UTF-16 or UTF-32). This setting only applies to offsets used in position ranges and is independent of the encoding used in protobuf strings ⁸ .

Documents

Each `Document` message represents one source file and contains:

Field	Meaning
<code>language</code>	A string naming the programming language of the file (e.g., <code>Rust</code>). A language enumeration is provided to prevent incompatible names ⁹ .
<code>relative_path</code>	The canonical path (no leading <code>/</code> , uses <code>/</code> as separator) to the file, relative to <code>metadata.project_root</code> ¹⁰ .
<code>occurrences[]</code>	A list of <code>Occurrence</code> messages describing positions of symbols and associated syntax/semantic information ¹¹ .
<code>symbols[]</code>	A list of <code>SymbolInformation</code> messages describing symbols defined in this document (definitions and metadata) ¹² .
<code>text</code> (optional)	Inline text contents of the document. Indexers usually omit this because consumers can read the file from disk, but it can be included for testing or for virtual documents ¹³ .
<code>position_encoding</code>	Determines how characters in ranges are measured: <code>UTF8CodeUnitOffsetFromLineStart</code> , <code>UTF16CodeUnitOffsetFromLineStart</code> or <code>UTF32CodeUnitOffsetFromLineStart</code> ¹⁴ . Indexers implemented in Rust use UTF-8 code unit offsets.

Occurrences

An `Occurrence` ties a source-range to a symbol and/or syntax highlighting. Indexers should group related information into a single occurrence to reduce payload size ¹⁵.

Field	Description
<code>range</code>	A list of three or four 0-based integers encoding <code>[startLine, startCharacter, endLine?, endCharacter]</code> ¹⁶ . If three elements are given, the end line is the same as the start line ¹⁶ . Editors display line/character numbers as 1-based. The character offset is interpreted using the document's <code>position_encoding</code> .
<code>symbol</code> (optional)	The symbol identifier whose definition or reference appears at this range ¹⁷ . Absent if the occurrence is purely syntactic (e.g., punctuation).
<code>symbol_roles</code>	A bitset describing roles: definition, import, write access, read access, generated code, test code and forward declaration ¹⁸ . A <code>Definition</code> role marks that the symbol is defined at this location; a <code>Reference</code> role is implied when <code>symbol</code> is set but <code>Definition</code> is not.
<code>override_documentation[]</code>	Markdown strings providing specific documentation for this occurrence, overriding the symbol's general documentation; useful when generic functions are instantiated with concrete types ¹⁹ .
<code>syntax_kind</code>	A <code>SyntaxKind</code> enum value for token-level syntax highlighting (e.g., <code>Keyword</code> , <code>Identifier</code> , <code>StringLiteral</code> , etc.) ²⁰ .
<code>diagnostics[]</code>	A list of diagnostics (errors, warnings, hints) reported at this range. Each diagnostic contains a severity, code, message, source and optional tags ²¹ .
<code>enclosing_range</code>	A range describing the nearest non-trivial enclosing AST node. This enables features like call hierarchies, breadcrumb display and expand-selection ²² . The enclosing range must fully contain the <code>range</code> and, for definitions, should include preceding documentation comments ²³ .

Symbols and Symbol Strings

Symbols identify entities such as functions, types or local variables. SCIP uses *human-readable* strings instead of integer IDs. Each symbol string has the form:

```
<scheme> <manager> <package> <version> <descriptor1> <descriptor2> ...
```

- **Scheme** – identifies the indexer (e.g., `rust-analyzer`, `scip-typescript`).

- **Package** – a triple `(manager, name, version)` describing the module or package where the symbol is defined (e.g., `cargo`, `serde`, `1.0.0`). Local symbols use the special scheme `local`.
- **Descriptors** – a sequence of descriptors that form a fully qualified name. Each descriptor has a `name`, optional `disambiguator` and a suffix indicating its kind (namespace, type, method, term, type parameter, parameter, meta, local, or macro) ²⁴. Descriptors must together form a unique path from the root of the file's AST to the symbol.

For example, a Rust function `fn foo` defined in module `my_mod` of crate `my_crate` version `0.1.0` might be encoded as:

```
rust-analyzer cargo my_crate 0.1.0 my_mod/ foo().
```

Local variables within a function use the special `local` scheme followed by an integer that uniquely identifies them within the document.

Symbol Information

The `SymbolInformation` message records metadata about a symbol:

Field	Description
<code>symbol</code>	The unique string identifier for this symbol ²⁵ .
<code>documentation[]</code>	CommonMark documentation strings for the symbol (excluding code signatures). Multiple strings can be provided to separate paragraphs ²⁶ .
<code>relationships[]</code>	A list of <code>Relationship</code> messages describing how this symbol relates to other symbols (implements, references, type definitions or definitions) ²⁷ .
<code>kind</code>	A fine-grained category (e.g., <code>Function</code> , <code>Class</code> , <code>Struct</code> , <code>Method</code> , etc.) ²⁸ . The <code>kind</code> enumeration distinguishes language-specific constructs even when their descriptor suffix is the same.
<code>display_name</code>	A human-friendly name for this symbol. Tools may use it instead of parsing the symbol string.
<code>signature_documentation[]</code>	Markdown strings describing the symbol's signature. For a function, this might include the parameter and return types; for a type, this might include generic parameters and trait bounds.
<code>enclosing_symbol</code> (optional)	For local symbols, the symbol string of the definition that encloses this one. This allows the local symbol to be part of the documentation hierarchy and call graphs ²⁹ .

Relationships

A `Relationship` connects the current symbol to another symbol and records how they interact:

Field	Meaning
<code>symbol</code>	The identifier of the related symbol ³⁰ .
<code>is_reference</code>	If <code>true</code> , occurrences of this symbol should be considered references for “Find references” on the related symbol ³¹ . For example, when class <code>Dog</code> implements interface <code>Animal</code> , <code>Dog#sound</code> is marked as an implementation of <code>Animal#sound</code> , and a reference on <code>Dog#sound</code> should count towards references for <code>Animal#sound</code> ³¹ .
<code>is_implementation</code>	If <code>true</code> , occurrences of this symbol should be returned by “Find implementations” on the related symbol. This is similar to <code>is_reference</code> but for implementation relationships ³² .
<code>is_type_definition</code>	If <code>true</code> , the symbol provides the type definition for the related symbol (used by “Go to type definition”) ³³ .
<code>is_definition</code>	Allows overriding “Go to definition” and “Find references” when a symbol does not have its own definition or may have multiple definitions. Due to current limitations, only global symbols may set this flag ³⁴ .

The `relationships` field allows indexers to record complex inheritance, implementation and aliasing patterns so that tools like Sourcegraph or an LLM can navigate between interfaces and implementations.

Languages and encoding

The SCIP schema defines an enumeration of common programming languages. Each language is associated with a numeric code so that `Document.language` can be a stable string (e.g., `Rust` corresponds to the code `40` in the enumeration) ³⁵. Indexers may use the language names directly as strings for unlisted or experimental languages.

The `position_encoding` field influences how column offsets in ranges are interpreted. Indexers written in Rust, Go or C++ typically set this to `UTF8CodeUnitOffsetFromLineStart`, whereas indexers written in JVM/.NET or JavaScript/TypeScript use UTF-16 offsets ¹⁴.

Diagnostic messages

SCIP allows indexers to embed diagnostics (compiler errors, warnings, hints) directly into occurrences. Each diagnostic has a severity (`Error`, `Warning`, `Information`, `Hint`), an optional code, a message, a source (tool name) and optional tags marking it as `Unnecessary` or `Deprecated` ²¹. Embedding diagnostics enables tools to surface compile-time errors alongside semantic navigation data.

Using SCIP with Rust

The `rust-analyzer` command-line tool can generate SCIP indexes for Rust projects. When you run `rust-analyzer scip --path <dir>`, it loads the Cargo workspace, analyses all crates, resolves references and writes a binary `index.scip` file to the output directory. The file’s

`metadata.tool_info` identifies the indexer as `rust-analyzer` and records its version and CLI arguments ³⁶. By default, the project root in metadata is the absolute path passed via `--path`, and the text encoding is UTF-8 ³⁷. Indexers follow the convention of naming their output `index.scip` ³⁸.

After generating the file, you can inspect it with the `protoc` command described earlier or use the `scip` CLI's `snapshot` subcommand to pretty-print occurrences. The snapshot format re-creates the source file with caret annotations that show definitions and references and the symbol strings associated with them ⁴. This makes it easy to verify that occurrences have the correct ranges, symbol identifiers and packages ³⁹.

Key design goals (Why SCIP?)

Sourcegraph designed SCIP to address limitations of LSIF. The design document lists several primary goals:

- Provide IDE-level code navigation across repositories and packages ⁴⁰.
- Make it easy to implement indexers that can run incrementally and in parallel ⁴¹.
- Use human-readable string identifiers and Protobuf rather than ad-hoc integer IDs to improve robustness and debugging ⁴².
- Avoid encoding the index as a graph; instead use arrays (documents and occurrences) so producers and consumers can stream data and merge partial indexes ⁴³.
- Optimise for producers (many indexers) rather than consumers (fewer clients); compactness is achieved via standard compression rather than custom integer IDs ⁴⁴.

Because of these goals, a `.scip` file is straightforward to generate from compiler information and straightforward to consume. The index can be merged simply by concatenating multiple protobuf messages (provided that only one `metadata` appears), enabling incremental and distributed indexing.

Summary

The SCIP file format is a Protobuf-based index for code-intelligence data. A `.scip` file contains one `scip.Index` message with:

1. **Metadata** describing the protocol version, the indexing tool and its arguments and the project root ⁵.
2. **Documents**, one per source file, each holding a list of **occurrences** (positions of tokens with semantic information) and **symbol definitions** ⁴⁵.
3. **External symbols** for dependencies, providing hover documentation for symbols defined in other packages ⁷.

Within each document, **occurrences** specify half-open source ranges and assign roles (definition, reference, read/write, etc.) ¹⁵. They may include syntax-highlight information, diagnostics and the range of the enclosing AST node. **Symbols** are globally unique strings formed from an indexer scheme, package coordinates and a sequence of descriptors ⁴⁶. **SymbolInformation** entries attach documentation, kind, relationships and other metadata to symbols ⁴⁷. **Relationships** capture implementation, type-definition and reference links between symbols, which improves features like “find references” and “find implementations” ⁴⁸.

Overall, SCIP indexes provide rich semantic context for codebases in a compact, extensible format. Tools like `rust-analyzer` can emit `.scip` files to power cross-repository navigation, semantic search or LLM-based code assistance.

- 1 5 6 7 8 9 10 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34
35 45 46 47 48 [raw.githubusercontent.com](https://raw.githubusercontent.com/sourcegraph/scip/main/scip.proto)
<https://raw.githubusercontent.com/sourcegraph/scip/main/scip.proto>
- 2 40 41 42 43 44 [scip/DESIGN.md at main · sourcegraph/scip · GitHub](https://github.com/sourcegraph/scip/blob/main/DESIGN.md)
<https://github.com/sourcegraph/scip/blob/main/DESIGN.md>
- 3 [Decoding SCIP index file – Sourcegraph Help Center](https://help.sourcegraph.com/hc/en-us/articles/15045932124941-Decoding-SCIP-index-file)
<https://help.sourcegraph.com/hc/en-us/articles/15045932124941-Decoding-SCIP-index-file>
- 4 11 12 38 39 [Indexers - Sourcegraph docs](https://sourcegraph.com/docs/code-search/code-navigation/writing_an_indexer)
https://sourcegraph.com/docs/code-search/code-navigation/writing_an_indexer
- 36 37 [scip.rs - source](https://rust-lang.github.io/rust-analyzer/src/rust_analyzer/cli/scip.rs.html)
https://rust-lang.github.io/rust-analyzer/src/rust_analyzer/cli/scip.rs.html