ChatGPT

Generating Complete AST, CST, and SCIP Datasets from a Python Repository

In this guide we walk through an end-to-end process for extracting a **Concrete Syntax Tree (CST)**, a derived **Abstract Syntax Tree (AST)**, and a **SCIP index** from a Python codebase. We assume a standard repository with source files under `src/`, running on Linux with all necessary tools installed (LibCST, `scip-python`, etc.) in an active `uv` Python environment. The goal is to produce comprehensive metadata outputs – including `ast_nodes.jsonl`, `cst_nodes.jsonl`, `index.scip.json`, `goids.jsonl`, and associated crosswalk and graph files – that feed into a code intelligence enrichment pipeline. The focus will be on using **LibCST** for structural extraction and **SCIP** for language-agnostic symbol indexing, with detailed steps for alignment and data persistence.

Prerequisites and Environment Assumptions

- **Dependencies Installed**: We assume LibCST (the Python Library for Concrete Syntax Trees) is installed and working, as well as the Sourcegraph SCIP Python indexer (`scip-python`) and the `scip` command-line tool. The environment is configured via `uv` (ensuring the correct Python interpreter and packages) – no additional installation steps are needed in this guide.
- **Repository Structure**: The Python codebase is located under a `src/` directory (treated as the top-level package). All `.py` files in this tree will be analyzed.
- **Output Directory**: We will refer to an output location (e.g., an `enriched/` folder or similar) where JSONL and Parquet files will be written. In a full pipeline, Parquet files might reside under a build directory (like `enriched/ast/ast_nodes.parquet`), with JSONL copies in a document output directory for easy consumption [1] [2].

With the groundwork set, we proceed in three major phases: **(1)** extracting the CST with LibCST (and then deriving the AST from it), **(2)** indexing symbols with `scip-python` to produce the SCIP data, and **(3)** linking these results and saving them in the desired formats.

1. CST Extraction with LibCST (Concrete Syntax Tree)

LibCST will be used to parse every Python file and produce a concrete syntax tree that preserves all details of the code's syntax. Unlike Python's built-in `ast` library, LibCST captures every character of source (including formatting and comments) in an immutable tree structure [3]. This makes it ideal for building a *complete* CST dataset that tools can use for precise source-to-metadata mapping.

a. Repository-wide Parsing Setup: To handle parsing across the whole repository (especially to resolve imports for qualified names), we utilize LibCST's metadata framework with a **FullRepoManager**. The FullRepoManager allows certain metadata providers (like fully qualified names) to consider the entire codebase at once [4]. We initialize it with the repo root and the list of all Python file paths under `src/`. For example:

```
import libcst as cst
from libcst import metadata
from pathlib import Path

repo_root = Path("<repo-root>")             # e.g., path to the parent of src/
py_files = [str(p) for p in repo_root.glob("src/**/*.py")]  # all Python files

# Choose the metadata providers we need for CST extraction
providers = {
    metadata.FullyQualifiedNameProvider,
    metadata.QualifiedNameProvider,
    metadata.PositionProvider,
    metadata.ByteSpanPositionProvider,
    metadata.ParentNodeProvider
}
manager = metadata.FullRepoManager(repo_root, py_files, providers)
```

In the above, we request providers that will supply *positions* (line/col spans), *parent links*, and *qualified names* for symbols. The FullRepoManager will pre-compute repository-wide caches for things like fully-qualified names (ensuring that relative imports are resolved in context) [4] [5]. We set `use_pyproject_toml=False` unless the project structure requires it (if package roots are defined via pyproject).

b. Parsing Files and Visiting CST Nodes: For each file in `py_files`, we obtain a MetadataWrapper (which wraps the parsed module CST with the metadata providers) and traverse the CST. For example:

```
for file_path in py_files:
    wrapper = manager.get_metadata_wrapper_for_path(file_path)
    module_node = wrapper.module  # LibCST Module representing the file
    # Traverse the module_node to collect CST nodes...
```

We can implement a LibCST `CSTVisitor` to walk through every node in the parse tree. The visitor's `visit_*` methods or a generic override (like `visit_Default` for all nodes) can capture each node's information. LibCST's matcher DSL is also available to find specific patterns, but for the CST dataset we want **all nodes**. The visitor will record for each node:

- **File Path**: The repo-relative path of the file (e.g., `src/package/module.py`).
- **Node Kind**: The concrete syntax kind or LibCST node type (e.g., `"Module"`, `"FunctionDef"`, `"If"`, `"Name"`, `"Indent"`, or even `"Token"` for punctuation). This is typically `node.__class__.__name__`.
- **Span (Start & End Position)**: Exact source span of the node, recorded as start and end line/column. Using the `PositionProvider` and `ByteSpanPositionProvider`, we retrieve the node's starting position and its full span. For instance, `wrapper.get_metadata(PositionProvider, node)` gives the start `(line, col)` of `node` [6]. To get the end position, we use

`ByteSpanPositionProvider` which yields the byte-index range of the node's code; we map the end byte back to line/col using the source content.

- **Text Preview**: Optionally, a snippet of the source code for that node (for example, the exact token text or a shortened representation). This can be extracted by slicing the file's source text at the byte span positions.
- **Parent Stack**: The hierarchy of parent node kinds/IDs. We can build a list of parent identifiers (for example, using `ParentNodeProvider` or by keeping a stack in the visitor). Each node can be assigned a unique `node_id` (e.g., a concatenation of file path and node location or an incremental ID) and we store the chain of parent node_ids or types. This provides context (e.g., a `Name` might have parents `Attribute -> Call -> Expr -> Module`, etc.).
- **Qualified Names**: If the node represents an import or name that LibCST can resolve, we attach its qualified name. Using `QualifiedNameProvider` / `FullyQualifiedNameProvider`, we can obtain possible fully-qualified symbols for name nodes [7]. For example, if a `Name` node is an alias to an imported symbol, the provider might return the actual absolute name. We include these in a list (could be empty for nodes that don't correspond to symbols). For definition nodes (like a ClassDef or FunctionDef's identifier), LibCST might treat the name differently; in those cases, we rely on our AST extraction for the defined symbol's name (see next section).

All this information is recorded for every node in the tree. The result is a **CST nodes dataset** capturing the full syntax of the repository. In our pipeline, this is written to `cst_nodes.jsonl` (with an optional compressed interim file). Each line in `cst_nodes.jsonl` is a JSON object with fields like the following (one per node) [8]:

- `path`: the file path of the node's file (relative to repo root).
- `node_id`: a stable identifier for the node (unique within the dataset, used for cross-references).
- `kind`: the concrete syntax kind or node type (e.g., `"If"`, `"Name"`, `"Keyword", "Indent"`).
- `span`: an object `{ "start": [line, col], "end": [line, col] }` giving the 1-indexed start and end position in the file [9].
- `text_preview`: a snippet of the source text for this node (for example, the exact token text if `kind` is `Name` or `Token`).
- `parents`: a list representing the stack of parent node kinds or IDs up to the root (useful for reconstructing context).
- `qnames`: a list of qualified names associated with the node (if applicable, e.g., the fully qualified name for an identifier token when resolvable).

By preserving every token and structural element, `cst_nodes.jsonl` ensures we have **lossless syntax context**. This can be joined later with semantic info (e.g., linking a GOID or symbol to the exact source span) [10]. The CST data is complementary to the AST data we extract next, as it retains formatting and comments needed for exact references or transformations.

2. AST Extraction with LibCST (Abstract Syntax Tree)

While the CST captures all syntax nodes, the AST view focuses on **semantic structure** – namely, the definitions (modules, classes, functions, methods, etc.) and their relationships. We derive the AST from the LibCST parse trees by selectively visiting only the relevant node types and annotating them with additional metadata (qualified names, parent relationships, docstrings, etc.). This AST dataset will feed into downstream components like the GOID registry, call graph builder, and control-flow graph builder [11] [12].

a. Identifying Structural Nodes: Using the same parsed `Module` nodes (and metadata) from LibCST, we traverse to collect nodes that represent **code entities**: - **Modules**: Each file itself is a module. We treat the top-level as an entity with a qualified name corresponding to its module path (e.g., `package.submodule` derived from file path). - **Class Definitions** (`ClassDef` in LibCST): Each class declaration is a node of interest. - **Function Definitions** (`FunctionDef` in LibCST): This includes top-level function definitions and methods (functions defined inside classes). In LibCST, both are `FunctionDef` nodes; we will differentiate by context (a function inside a class gets a method kind in GOID classification, but for AST data we can just record it with its qualname). - Possibly other constructs if needed: e.g., we might include module-level assignments or variables if we wanted, but typically for the enrichment pipeline, the focus is on the above entity types (which get GOIDs). We do not include every small statement in AST, since the CST already covers them. The AST is meant to catalog definitions and their scopes.

We can implement a visitor or simply filter the CST for these node types. For example, LibCST's matcher could find all `ClassDef` and `FunctionDef` nodes. Alternatively, a custom visitor can override `visit_ClassDef` and `visit_FunctionDef` methods to capture them. We maintain a **scope stack** or use `ParentNodeProvider` to determine the current qualified name context while traversing (e.g., when visiting a class, push its name; when leaving, pop it). LibCST's providers can also supply qualified names for definitions: for instance, `FullyQualifiedNameProvider` will give the fully-qualified module path plus class name for a class definition's name node [13]. In practice, one can derive the qualified name manually: - For a top-level class or function, prefix with the module name. - For a nested class or a method, prefix with the parent class's qualname. - For a nested function (inner function), prefix with the enclosing function's qualname. - The module's own qualname is its dotted import path (e.g., `package.module` without the `.py`).

b. Extracting Attributes for AST Nodes: For each identified AST node (module, class, function), we gather:

- **Name**: The symbol name (e.g., class name, function name). For modules, this can be the file stem or module path.
- **Node Type**: The kind of node (`"Module"`, `"ClassDef"`, `"FunctionDef"`). This corresponds to the LibCST node type.
- **Qualified Name (Qualname)**: The fully qualified name, i.e., the dotted path including any enclosing scopes. For example, a class `Foo` in module `pkg.mod` has qualname `pkg.mod.Foo`; a method `bar` inside that class would have `pkg.mod.Foo.bar`. We either compose this during traversal or use the metadata providers (FullyQualifiedNameProvider yields absolute names useful for cross-package contexts [13]). This qualname is critical as a canonical identifier for the entity in the repository.
- **Parent Qualname**: The qualname of the enclosing scope. This is `null` (or empty) for a module (since modules have no parent in Python), the module's qualname for top-level classes/functions, the class's qualname for methods, etc. This field lets us reconstruct the hierarchy or do parent-child joins.
- **Location (Start and End)**: The line and column where the definition starts, and where it ends. We get the start from `PositionProvider`. To find the end line/col, we can use the node's body last element or `ByteSpanPositionProvider` to capture the full span of the def including its suite. For example, if a function spans lines 10-25, we record `lineno=10, end_lineno=25` (and similarly columns, though columns are most relevant for the start of the definition). These positions correspond to the def header and end of the block, helping align with both CST spans and SCIP ranges.

- **Decorators**: If the node has decorators (applicable to classes and functions), we collect the list of decorator names or expressions. LibCST provides the `.decorators` attribute on FunctionDef/ClassDef, from which we can extract each decorator's name (for simplicity, store the qualified name or code of the decorator).
- **Docstring**: The extracted docstring, if any. We can get this via LibCST by checking if the first statement in the class/function body is a `SimpleStatementLine` containing a `String` literal. LibCST's convenience method `Module.get_docstring()` or similar on ClassDef/FunctionDef can retrieve it [14]. If present, we record the docstring text (or a truncated version if needed) for documentation purposes.
- **Content Hash**: A hash of the node's content, used to detect duplicate bodies. The pipeline uses this to avoid processing identical code twice. We can compute a stable hash (e.g., MD5 or SHA) of the source substring for this node. For instance, take the bytes from start to end of the node (from the original file content) and hash them. This `hash` acts as an identifier for the code body (helpful if the same function is defined identically in multiple places).

Each collected AST node becomes a record in the `ast_nodes.jsonl` file (and Parquet equivalent). This dataset is the *semantic index* of code entities. The fields in `ast_nodes.jsonl` appear as [15] [16]:

- `path`: File path of the entity's source file.
- `node_type`: Node type (LibCST class name, e.g. `"ClassDef"`, `"FunctionDef"`, or `"Module"`).
- `name`: Name of the symbol (class/function name; for modules this could be the module name).
- `qualname`: Fully qualified name of the entity (as described above).
- `lineno` / `col_offset`: Start line and column of the definition.
- `end_lineno` / `end_col_offset`: End line and column for the end of the entity's span [17].
- `parent_qualname`: Qualname of the parent scope (or null if none) [18].
- `decorators`: List of decorators applied (names or code strings).
- `docstring`: Docstring string if present, else null [19].
- `hash`: A content hash of the node's source (string) [20].

This **AST index** is the primary input for building the GOID registry and various graphs. It captures the hierarchy (through qualified names and parent link), locations (for mapping to code), and basic metadata like decorators and docs. The AST also feeds into metrics (like counting functions, calculating complexity, etc.) during enrichment [21], but here our focus is on structural extraction.

Using LibCST Visitors/Matchers: Throughout CST and AST extraction, LibCST's visitor pattern and matcher DSL prove useful. For example, we could use a matcher to find all function definitions easily instead of manually checking node types. A snippet using matchers could be:

```
import libcst.matchers as m

# Within a MetadataWrapper context:
func_defs = list(m.findall(module_node, m.FunctionDef()))
class_defs = list(m.findall(module_node, m.ClassDef()))
```

However, using a visitor with explicit methods may be more straightforward when collecting many fields. The key is that LibCST provides *ergonomic traversal* and *metadata* so we don't have to compute qualified

names or scopes from scratch – these are provided by `QualifiedNameProvider`, `ScopeProvider`, etc., once we've set up the metadata wrapper [22] [23]. For instance, to get all references of a defined name, we could combine `ScopeProvider` and `QualifiedNameProvider` to map definitions to usage sites [24], but in this guide we limit ourselves to recording definitions and letting SCIP handle cross-reference mapping.

By the end of this phase, we have two complementary datasets: - `cst_nodes.jsonl` – the complete concrete syntax tree nodes of all files (for exact spans, tokens, etc.). - `ast_nodes.jsonl` – the abstracted syntax tree of definitions (for semantic structure).

These will be used to generate stable identifiers (GOIDs) and form the backbone of further analysis.

3. Generating the SCIP Symbol Index with scip-python

Next, we leverage Sourcegraph's **SCIP** (Semantic Code Indexing Protocol) to index the repository for symbol definitions and references. The `scip-python` tool analyzes the Python code to produce a language-agnostic symbol graph. This step gives us the **SCIP dataset** capturing where each symbol is defined and all the places it's referenced, along with additional symbol metadata (signatures, documentation, etc.) [25] [26]. We will produce an `index.scip` file (in SCIP's binary format) and then convert it to a JSON representation for easier integration.

a. Running the SCIP Indexer: From the repository root (or the directory containing `src`), we execute:

```
scip-python index ./src -o index.scip
```

This command runs the Python indexer on all files under `src/` and outputs a SCIP index (here we specify output to `index.scip`). The indexer analyzes the code, resolves imports and definitions, and encodes the information into SCIP's binary schema. Once this completes, we convert the binary index to JSON:

```
scip print --json index.scip > index.scip.json
```

The `scip print --json` command reads the `.scip` file and prints a JSON serialization. The resulting `index.scip.json` is typically an array of JSON objects (one per source file, known as *documents*) [27]. We store this file in our output (for example, under `Document Output/index.scip.json` as noted in the pipeline docs [28]).

b. Understanding the SCIP JSON structure: Each document in the `index.scip.json` contains: - `relative_path`: the path of the file (relative to the repository root or the indexing root, e.g., `"src/package/module.py"`). - `occurrences`: a list of symbols occurrences in that file. Each occurrence typically has: - a `range`: the location of the symbol usage in the file (often as `[start_line, start_char, end_line, end_char]`, 0-indexed positions), - a `symbol`: the global identifier of the symbol (a string in SCIP format, which encodes the package, module, and symbol signature), - a `symbol_roles`: a bitmask or enum indicating if this occurrence is a definition, reference, import, etc. - `symbols`: a dictionary of symbol metadata for symbols defined in this file. For each symbol (keyed by the

symbol string), it provides details like the symbol's name, kind (function, class, variable, etc.), signature or hover text, and documentation string if available.

For example, if `foo()` function is defined in `module.py`, the SCIP index will have an occurrence entry for `foo` at its definition location with a symbol like `python pkg/module.py foo().` (exact format is not crucial here), marked as a definition role. If `foo()` is called in other files, those files' occurrences will list the same symbol at the call site with a reference role. The symbols section for `foo` in `module.py` might include its signature `(params) -> return_type` and docstring. In summary, the SCIP data provides a *graph of symbol definitions and uses across the repository* [26].

c. Applications of SCIP Data: The SCIP index serves as a language-agnostic symbol graph. It's particularly useful for cross-reference queries (finding all references to a definition) and for linking definitions across repositories or languages. In our enrichment pipeline, we will use SCIP to augment our Python-specific analysis: - We will **match SCIP symbols to our AST entities** (so we can enrich the GOID crosswalk with external symbol IDs). - We will extract **def→use edges** from the SCIP index to build a *Symbol Use Graph* – essentially edges linking a definition to each file that uses it [29] [30]. - The SCIP data is also the foundation for cross-language code navigation features (outside the scope of our single-repo pipeline, but important for integration into larger systems).

At this stage, we have `index.scip.json` ready, which we will align with the AST data.

4. Aligning SCIP Symbols with AST/CST Anchors

With both LibCST-derived data (AST and CST) and the SCIP index in hand, the next step is to **cross-reference** them. The goal is to tie together the purely semantic view from SCIP with our AST nodes and GOIDs, and to enable precise source mapping via CST spans. This alignment produces the **GOID crosswalk** and ensures all datasets speak a common language of identifiers.

a. Generating GOIDs for AST Entities: First, we create GOIDs (Global Object IDs) for each code entity captured in `ast_nodes.jsonl`. The GOID is a stable hash-based identifier that uniquely represents a code entity (module, class, function, or even basic blocks in CFG) in a specific repo and commit [31] [32]. In practice, a GOID might be computed by hashing a tuple of `(repo_id, commit_hash, language, file_path, entity_kind, qualname, span)`. Our pipeline uses a `GOIDBuilder` to walk through the AST nodes and assign each a 128-bit hash and a human-readable URN. For example, a function `pkg.module.foo` defined from lines 10–20 might get a GOID URN like:

```
goid:<repo>/<path>#python:function:pkg.module.foo?s=10&e=20
```

(where `s` and `e` query params indicate the start and end lines) [33]. The `goids.jsonl` (and `goids.parquet`) file will list all entities with their GOID hash, URN, and metadata such as repo, commit, path, kind, qualname, start_line, end_line, etc. [34] [35]. This GOID registry is our master list of code entities.

b. Building the GOID Crosswalk: We then create a **crosswalk** table that links each GOID to related identifiers from different sources [36]. Each entry in the crosswalk maps a GOID to: - the language and file path, - the module path (dotted name of the file, sans ".py"), - the source span (start_line and end_line) of

that entity, - the `ast_qualname` (qualified name from AST) [37], - and placeholders for other IDs: `scip_symbol`, `cst_node_id`, `chunk_id`, etc. [38].

At the time of GOID generation, we fill in what we know: the ast qualname, path, lines, etc., and likely leave `scip_symbol` null initially [39]. Now we use the SCIP index to fill the SCIP symbol field. For each AST entity (GOID) that represents a definition, we find the corresponding symbol in the SCIP data: - Use the file path and line number to locate the SCIP occurrence marked as a definition in that file at that location. The SCIP occurrence's `symbol` string is then the identifier for that entity. - We insert that symbol string into the crosswalk entry's `scip_symbol` column [37]. This effectively links our GOID (and thus AST node) to the SCIP global symbol graph.

This process may involve a small tolerance (e.g., if SCIP ranges and AST lines differ slightly due to 0-indexing or inclusive/exclusive ranges, or if the definition spans multiple tokens). But generally, matching by file and a near-equal start position is reliable. As a result, the GOID crosswalk now has enriched entries: where available, each GOID knows its SCIP symbol ID.

c. Utilizing CST Node IDs: In the future (or if implemented already), we can also connect CST nodes to GOIDs. In our `cst_nodes.jsonl`, every node had a `node_id`. For definition nodes (like the `FunctionDef` or its name token), we could mark that with the corresponding GOID. The crosswalk has a `cst_node_id` column reserved for this [10]. For now, this might remain null or only be used internally, but the idea is that one could map back from an entity to the exact CST node(s) representing it in source.

After cross-referencing, we have unified the data: any GOID can be resolved to a qualname, a source file span, and a SCIP symbol if one exists. Likewise, any SCIP symbol found in references can be mapped back to a GOID via this crosswalk. This is powerful for downstream analysis, as we can join on GOID or symbol to combine information.

d. Cross-Referencing Symbol Uses: With GOIDs linked to SCIP symbols, we can now produce a **symbol use graph**. Using the SCIP occurrences, we generate edges from each symbol's defining GOID to the file where it's used: - For each occurrence in SCIP marked as a reference (not a definition), find the symbol string and the file of occurrence. - Look up the GOID that has that symbol as its SCIP symbol (the crosswalk provides this mapping). - Emit an edge: GOID -> (file of occurrence). In practice, our pipeline creates a table `symbol_use_edges` with columns like `symbol`, `def_path`, `use_path`, etc. [40]. Since GOID already contains `file_path`, we may also link GOID -> GOID if the reference is to another GOID, but generally symbol use is captured as file-to-file or module-to-module references.

The symbol use edges (also output as Parquet/JSONL) help identify where each entity is used across the repo, and can be filtered by same-file or same-module flags [30]. These are directly derived from the SCIP index.

At this point, we have the core datasets required: the AST nodes (with GOIDs), CST nodes, and the SCIP index, all cross-linked. We proceed to final output steps and mention how these feed into further analyses.

5. Persisting Outputs to JSONL and Parquet

For each dataset we've built, we will output it in two formats: **JSON Lines (JSONL)** for easy consumption (e.g., streaming into an LLM or quick scripting) and **Parquet** for efficient querying (columnar storage for analytics). The enrichment pipeline typically writes Parquet files during processing, then converts them to JSONL as a final step [1]. We ensure that the JSONL files contain the same fields/records as the Parquet tables, one JSON object per line, making them suitable for line-by-line reading.

Key outputs and how to write them:

- **AST Nodes** (`ast_nodes.parquet` and `ast_nodes.jsonl`): After collecting all AST node info in memory (e.g., as a list of dictionaries or a pandas DataFrame), we write it to Parquet using a library like PyArrow or DuckDB. For example, using PyArrow:

```
import pyarrow as pa
import pyarrow.parquet as pq

ast_table = pa.Table.from_pylist(ast_nodes_list)  # ast_nodes_list is list
of dicts
pq.write_table(ast_table, "enriched/ast/ast_nodes.parquet")
```

  Then convert to JSONL. One convenient method is using DuckDB:

```
COPY (SELECT * FROM 'enriched/ast/ast_nodes.parquet')
TO 'Document Output/ast_nodes.jsonl' (FORMAT JSON, HEADER false);
```

  This ensures the JSONL has one line per row [41]. The same approach applies to other Parquet outputs.

- **CST Nodes** (`cst_nodes.parquet` and `cst_nodes.jsonl`): Similarly, we output the comprehensive CST node list. Given the potential size (every token of every file), these might be large. We first write `cst_nodes.parquet`. (In the pipeline, a temporary gzip JSON may have been produced and then normalized [42] – but directly writing Parquet is fine.) Then produce the JSONL via conversion.

- **GOID Registry** (`goids.parquet` / `goids.jsonl`): This is created after running the GOID builder on ast_nodes. We output each GOID entry with its metadata columns (hash, URN, repo, commit, path, kind, qualname, start/end lines, etc.) [34] [35]. The JSONL variant `goids.jsonl` will be used by other tools to identify entities.

- **GOID Crosswalk** (`goid_crosswalk.parquet` / `goid_crosswalk.jsonl`): After aligning SCIP and CST, we output the crosswalk mapping. Each row links a GOID (by its URN or hash) to potentially multiple other IDs: the `scip_symbol` (if matched), the `ast_qualname`, `file_path` and lines, and possibly a `cst_node_id` if implemented [43] [44]. One GOID can have multiple crosswalk

entries if it appears multiple times (e.g., a function defined once but perhaps referenced multiple times in AST – though definitions are single, so crosswalk mainly helps with linking definitions to other representations).

- **SCIP Index** (`index.scip.json`): We already obtained this via `scip print`. We store it as is. For completeness, we might also keep the raw `index.scip` binary (for reuse or for Sourcegraph ingestion). The JSON is primarily for our pipeline's consumption and verification [28].

- **Graph Datasets**: Using the AST and cross-referenced data, the pipeline can generate various static analysis graphs:

- **Import Graph** (`import_graph_edges.parquet/jsonl`): Using LibCST's import metadata, we record edges between modules (who imports whom) [45]. This can be built by scanning AST or using LibCST's `Import` and `ImportFrom` nodes with FullyQualifiedNameProvider to resolve module names. Each edge has `src_module`, `dst_module` and maybe fan-in/out counts and cycle group id (if we compute SCCs) [46].
- **Call Graph** (`call_graph_nodes/edges.parquet/jsonl`): Using the AST and possibly SCIP, we identify call relationships. The `CallGraphBuilder` goes through each function's body (AST or LibCST tree) to find `Call` expressions. For each call, it tries to resolve the target function's GOID. Resolution can use lexical scope analysis (via LibCST ScopeProvider), import resolution, and augmented by SCIP for global symbol matches [47] [48]. We then output nodes (which are basically the callable GOIDs with some attributes like arity, is_public, etc.) and edges (caller_goid -> callee_goid, with metadata about how it was resolved and confidence) [49] [50]. Calls that cannot be resolved to a known GOID get a null callee but are still recorded with their callsite info [51] [52].
- **Control Flow Graph (CFG)** (`cfg_blocks.parquet/jsonl`, `cfg_edges.parquet/jsonl`): Using each function's AST, we construct a CFG to break the function into basic blocks. Our CFG builder uses the AST to identify control structures (ifs, loops, exceptions) and splits the function accordingly [53]. Entry and exit blocks are added for completeness. Each block and function gets a GOID (functions already had one; blocks get a special GOID of kind "block") [54] [35]. We output block nodes (with block index, corresponding function GOID, etc.) and edges representing control flow jumps (block i -> block j). These tables allow analysis of function structure.
- **Data Flow Graph (DFG)** (`dfg_edges.parquet/jsonl`): Building on the CFG, we analyze variable definitions and uses to create data-flow edges. For each basic block, we track where variables are defined and where they are used. If a variable defined in block X is used in block Y (and Y is reachable from X in CFG), we emit a def-use edge [55] [56]. We also mark phi-nodes for merges (e.g., a variable coming from two branches) [57]. Each edge can be identified by source/destination block indices and includes the variable name and whether it's a read/write/etc. The DFG edges can be joined with CFG blocks by those indices to pinpoint the flow in context [58].

Each of these graph datasets uses GOIDs (hash or URN) as foreign keys to link back to the entities in `goids.jsonl`. For example, call graph edges reference `caller_goid_h128` and `callee_goid_h128` which match entries in the GOID registry [59] [60]. This consistent use of GOIDs allows all data to be interconnected: we can start from a GOID and find its properties, source code span, calls, uses, etc., by simple joins [61].

- **Analytics and Metrics**: The pipeline may produce additional analysis outputs (hotspots, typedness, function metrics, etc.) as JSONL, which derive from combining the above data with external info (git

history, type checker results). For instance, `ast_metrics.jsonl` summarizes counts of AST nodes per file [62] [21], `hotspots.jsonl` merges git churn with complexity [63], etc. These are not extracted directly from LibCST or SCIP but computed later using the extracted AST plus other inputs. They can be included in the pipeline as needed for a comprehensive metadata suite.

In all cases, after writing Parquet files, we perform a JSONL export (the pipeline's `generate_documents.sh` automates copying Parquets to a docs directory and converting them) [2]. The final JSONL files (e.g., `goids.jsonl`, `ast_nodes.jsonl`, `cst_nodes.jsonl`, `index.scip.json`, `call_graph_edges.jsonl`, `cfg_blocks.jsonl`, etc.) form the complete set of datasets described in **README_METADATA.md**. These can be loaded by downstream tools or even fed into language model prompts (the JSONL is convenient for streaming into an LLM context, as each line is a self-contained JSON object).

6. Utilizing LibCST Codemods for Transformations (Future Work)

While the above focuses on extracting structural and semantic information, it's worth noting that LibCST also supports transformations via its **codemod framework**. Once we have the CST and AST in hand, one could use codemods to perform automated code modifications or remediation across the repository. A codemod is essentially a specialized `CSTTransformer` with context about the file and repository [64] [65]. For example, if analysis revealed certain APIs to update, a codemod can systematically apply those changes. The LibCST codemod tool can run a transform over all files in the repo via a single command [66]. In our pipeline, codemods are not executed during extraction, but the structured data we produced (especially CST with exact positions) could help target transformations. Developers implementing enrichment pipelines can plan separate codemod steps after analysis – for instance, using the AST/CST mapping to locate code to rewrite, then employing LibCST to safely apply edits.

In summary, codemods provide a pathway from **insight to action**: the enriched datasets (AST, CFG, etc.) highlight where changes are needed, and LibCST codemods can make those changes in a controlled manner. This guide, however, prioritizes building the knowledge base (datasets) on which such transformation steps could later operate.

Conclusion

By following this guide, we achieve a comprehensive extraction of Python code structure and semantics: - A **LibCST-derived CST** for every file, preserving complete syntax and exact source positions for all nodes. - A refined **AST index** of all definitional elements (modules, classes, functions) with qualified names and metadata, suitable for generating stable identifiers (GOIDs) and powering static analyses. - A **SCIP index** capturing cross-file symbol definitions and references, providing a language-neutral graph of how symbols connect. - Alignment of these outputs via GOIDs and crosswalks, enabling unified queries (e.g., mapping a function to its source, documentation, and usage in one step). - The data persisted in accessible formats (JSONL for consumption, Parquet for analytics) as described in the metadata reference [2] [12].

Armed with `goids.jsonl`, `ast_nodes.jsonl`, `cst_nodes.jsonl`, `index.scip.json`, and the linked crosswalk and graph files, developers can implement robust enrichment pipelines. These datasets feed into building call graphs, control-flow and data-flow graphs, import dependencies, and various analytics that help understand and navigate the codebase. Downstream systems or AI agents can leverage this rich metadata to reason about the code without re-parsing it, since all the heavy analysis is done

upfront. By systematically using LibCST and SCIP together, we ensure both **precision** (through CST exactness and qualified AST info) and **coverage** (through SCIP's whole-repo symbol connectivity), laying a solid foundation for advanced code intelligence features.

Sources:

- LibCST documentation and usage guides [3] [67] [4]
- Enrichment pipeline metadata reference (README_METADATA.md) [12] [28]
- Sourcegraph SCIP protocol reference (for generating and interpreting `index.scip.json`) [26] [25]
- CodeIntel pipeline descriptions for GOIDs, crosswalks, and graph outputs [32] [68] [9]

---

[1] [2] [8] [9] [10] [11] [12] [15] [16] [17] [18] [19] [20] [21] [25] [26] [27] [28] [29] [30] [31] [32] [33] [34] [35] [36] [37] [38] [39] [40] [41] [42] [43] [44] [45] [46] [47] [48] [49] [50] [51] [52] [53] [54] [55] [56] [57] [58] [59] [60] [61] [62] [63] [68]

README_METADATA.md

file://file_0000000021cc722f995705430cc774b9

[3] [4] [5] [6] [7] [13] [14] [22] [23] [24] [64] [65] [66] [67] libcst.md

file://file_00000000d294722fbb5c6a514c1b530f