Winnow: Software architecture specification

Ayan Das

March 9, 2025

Contents

1 Introduction

1.1 Purpose of the Document

This **Software Architecture Specification (SAS)** aims to describe the *architectural structure*, *components*, and *interfaces* of the proposed Python-based system that (1) parses large volumes of BibTeX entries, (2) augments missing abstracts from external sources, and (3) filters references relevant to the intersection of Physics and Machine Learning (ML). This document draws upon the requirements established in the *Software Requirements Specification (SRS)* and serves as a bridge between the higher-level requirements and the detailed design (which will be covered in the *Software Design Specification, SDS*).

1.2 Scope

The scope of this SAS is limited to:

- The *internal* architectural components (e.g., modules for parsing, augmentation, filtering, classification).
- *Integration* with external services (Semantic Scholar, OpenAlex, arXiv, Google Scholar, etc.), as well as potential third-party scrapers.

- *Deployment* concerns and concurrency (process or threading models).
- *Quality attributes* such as scalability, reliability, and maintainability.

This document follows the general structure recommended by *ISO/IEC/IEEE 42010* and standard industry practices for architecture documentation.

1.3 Definitions, Acronyms, and Abbreviations

SRS
Software Requirements Specification, describing functional/non-functional requirements.

SAS
Software Architecture Specification (this document).

SDS
Software Design Specification, describing module/class-level designs.

API
Application Programming Interface.

CLI Command-Line Interface.

ML Machine Learning.

Parsing Module
A software component responsible for reading ‘.bib‘ files and converting them into internal data structures.

Filtering Module
A software component that applies text-based or classification-based filtering logic.

Augmentation Module
A software component that retrieves missing abstracts from external sources.

1.4 References

- SRS: *Software Requirements Specification for the BibTeX Aggregation and Filtering Pipeline.*
- IEEE 29148 (Requirements) and ISO/IEC/IEEE 42010 (Architecture).
- Documentation for external services:
  - Semantic Scholar: `https://api.semanticscholar.org`
  - OpenAlex: `https://docs.openalex.org`
  - arXiv: `https://arxiv.org/help/api/index`

2 Architectural Drivers

2.1 Key Functional Requirements (Summary)

From the SRS, the most critical functional requirements that *shape* this architecture are:

FR-1 **FR-1: Directory-Based Discovery.** We must parse large, hierarchical directories of '.bib' files efficiently.

FR-2 **FR-2: Configurable Invariant Checks.** The system must validate that each reference meets certain minimal requirements (title, etc.).

FR-3 **FR-3: Abstract Augmentation with Multiple External Services.** The pipeline must call external APIs in a prioritized or fallback sequence.

FR-4 **FR-4: Keyword-Based Filtering (Physics & ML Lexicons).** The system must do text-based intersection checks over title + abstract.

FR-5 **FR-5: Optional Text Classification.** For advanced filtering that ensures minimal misses and high precision.

FR-6 **FR-6: Custom Scraper Interface.** A plugin-like mechanism to integrate specialized scrapers or data sources.

FR-7 **FR-7: Logging, Error Handling, and Reporting.** Must be robust, with structured logs, partial failure handling.

FR-8 **FR-8: Final BibTeX Generation.** Combine the curated references into a single or multiple '.bib' outputs.

2.2 Non-Functional Requirements / Quality Attributes

- **Scalability & Performance:** The architecture must handle potentially hundreds of thousands of references without undue memory usage or excessive runtime.
- **Maintainability:** The code must be modular. Each stage (parse, augment, filter, export) is separated into self-contained components.
- **Reliability & Fault Tolerance:** External API calls can fail or be rate-limited; the system must degrade gracefully.
- **Extensibility:** Domain lexicons can change; new scrapers can be added without rewriting core modules.
- **Usability:** Must provide a simple CLI or library interface for the main pipeline.
- **Security:** API keys must not be logged in plaintext; sensitive data must be restricted in the code or config.

2.3 Constraints and Assumptions

- **Python 3.x**: The architecture will leverage Python's ecosystem (e.g., concurrency with multiprocessing or async).
- **File-Based Input and Output**: '.bib' source files are on a local or networked file system. Output is a single '.bib' or multiple '.bib' files.
- **Limited Control Over External APIs**: Must handle rate-limits, timeouts, inconsistent data from external services.

3 Architectural Principles

3.1 Design Philosophy

- **Pipeline-Oriented Architecture:** A linear (or nearly linear) sequence of well-defined stages: *parse* → *augment* → *filter* → *export*. Each stage is loosely coupled but can communicate intermediate results.
- **Modular and Extensible:** We favor a design that allows *pluggable scrapers* for abstract augmentation and *pluggable classifiers* for advanced text analysis.
- **Declarative/Functional Approach Where Feasible:** Configuration-based operation, minimal shared mutable state. Where possible, data flows through pure functions or well-defined transformations.
- **Error Tolerance and Graceful Degradation:** The system logs errors but keeps processing unaffected references.

3.2 Key Architectural Patterns

- **Layered / Tiered Approach:**
  1. **Data Ingestion Layer**: reading '.bib' files, applying invariants.
  2. **Augmentation Layer**: external lookups to gather more data.
  3. **Filtering Layer**: text intersection + classification.
  4. **Export Layer**: final assembly to '.bib'.
- **Plugin or Strategy Pattern for Scrapers**: Each external scraper or data source can implement a standard interface, enabling easy substitution and fallback.
- **Configuration-Driven Behavior**: The pipeline's steps, concurrency, lexicons, etc., are specified in a configuration file or CLI arguments.

4 System Context and Overview

4.1 System Context

Figure 4.1 (a conceptual diagram) shows the environment in which the pipeline operates:

- **Users / Researchers**: Provide the root directory with '.bib' files and request the system to produce a curated '.bib'.
- **File System**: Contains the input '.bib' files. The system outputs curated '.bib' files to it.
- **External APIs**: The pipeline queries these for missing abstracts.
- **Plugin Ecosystem**: Specialized scrapers can be plugged in for certain venues or subscription-based APIs.

System Context Diagram (Conceptual)

Users (CLI) $\longrightarrow$ [Pipeline] $\longrightarrow$ **File System**
[Pipeline] $\longleftrightarrow$ **External APIs**
[Pipeline] $\longleftrightarrow$ **Custom Plugins**

4.2 Overview of Major Use Cases

The pipeline addresses these primary use cases at the system boundary:

1. **Ingest & Validate References:** Read '.bib' files, parse them, apply invariants, store references in a structured internal format.
2. **Augment Data:** For references missing abstracts, query external services. Possibly also store a local cache for repeated runs.
3. **Filter & Classify:** Perform keyword-based intersection or advanced text classification to produce a *candidate set*.
4. **Output Final BibTeX:** Write curated references with new abstract fields to an output file.

5 Architecture Views

5.1 Logical View

Purpose: The Logical View focuses on the *main modules*, their *responsibilities*, and how they *interact* to satisfy the functional requirements.

5.1.1 Main Subsystems / Modules

a) **InputParser**:

- *Responsibility:* Locate all '.bib' files, parse them into an *internal reference object*, apply basic data invariants.
- *Input:* Root directory path, optional config specifying ignore patterns or advanced validation rules.
- *Output:* List (or generator) of reference objects.

b) **AbstractAugmenter**:

- *Responsibility:* For each reference lacking an abstract, query external APIs in a fallback sequence (Semantic Scholar, OpenAlex, etc.).
- *Input:* Reference objects from `InputParser`.
- *Output:* Updated references with newly added `abstract` fields (if found).

c) **FilterEngine**:

- *Responsibility:* Apply broad *lexicon-based intersection* or advanced *text classification*.
- *Input:* References from `AbstractAugmenter`.
- *Output:* A subset (candidate references) or flagged references with probability scores.

d) **Exporter**:

- *Responsibility:* Merge or group final references, write them to a single '.bib' file.
- *Input:* Filtered references from `FilterEngine`.
- *Output:* A curated '.bib' file on the file system.

e) **ScraperPlugins** (Optional):

- *Responsibility:* Specialized modules for certain journals or websites. They implement an interface recognized by `AbstractAugmenter`.

f) **ClassifierPlugins** (Optional):

- *Responsibility:* Provide advanced text classification logic, implementing a standard interface for model inference or training.

5.1.2 Interactions among Modules (High-Level Workflow)

1. **InputParser** scans directories, loads references, and provides them as a list (or a lazy iterator) to the next stage.
2. **AbstractAugmenter** receives references, checks which ones *lack* abstracts, and calls a pipeline of scraper plugins or direct API calls. It updates the references in-place with retrieved abstracts.
3. **FilterEngine** then reads the updated references, runs textual analysis (lexicons or classifiers), and separates references into *kept* vs. *excluded*.
4. **Exporter** writes the kept references to a user-specified '.bib' file with updated fields.

5.2 Process View / Concurrency Model

5.2.1 Process Model and Threads

Because Python can handle concurrency in multiple ways, we highlight two potential concurrency models:

A. **Multi-threaded approach using *ThreadPoolExecutor***:

- Typically beneficial for *I/O-bound* tasks (like external API calls).
- Each API call can be dispatched asynchronously while other references or tasks proceed.
- Care must be taken with Python's GIL (Global Interpreter Lock) for CPU-bound tasks (like heavy text classification).

B. **Multi-process approach (e.g., `multiprocessing` or `Ray`)**:

- CPU-bound tasks such as *model inference* can be parallelized effectively across processes.
- External API calls can also be parallelized if we want to avoid blocking in a single process.

Recommended Hybrid Strategy:

- Use an async or thread-pool concurrency for the *abstract augmentation* stage (I/O-bound).
- Use a process-based concurrency (if needed) for *ML classification* if the model is large or CPU-intensive.

5.2.2 High-Level Concurrency Steps

1. **Parsing Step**: Typically runs single-threaded or multi-threaded, but usually it is fast relative to augmentation, so concurrency might be optional here.
2. **Augmentation Step**: Potentially spawns multiple threads to query external services. Each reference that lacks an abstract is assigned to a worker thread. Results are re-assembled in the main thread.
3. **Filtering Step**:
   - *Lexicon-based intersection* is straightforward and can be done sequentially or with data parallelism.
   - *Classification-based* approaches might spin up multiple processes or a single GPU-based process, depending on user configuration.
4. **Export Step**: Typically single-threaded, but trivial to parallelize if the final '.bib' writing becomes a bottleneck.

5.3 Physical / Deployment View

- The **typical deployment** is on a single server or workstation with network access to external APIs.
- **Optional scaling** can occur on HPC clusters or in the cloud (AWS, GCP) if the user wants to handle extremely large '.bib' sets.
- Minimal infrastructure components:
  - **Local File System**: Input, output, logs.
  - **(Optional) Caching Database**: Could be a local SQLite or Redis store for abstract queries.

5.4 Data View

5.4.1 Conceptual Data Model

Reference Record: • **Key Fields:** `bibtex_key`, `title`, `authors`, `year`, `abstract`, `venue`, `doi`, `url`, `raw_bibtex`.

Relationships and transformations:

1. **Raw '.bib' → Reference Object:** The **InputParser** extracts structured data from the raw BibTeX entry.
2. **Reference Object + External Service → Augmented Reference Object**: `abstract` is added or updated.

3. **Reference Object → Filter Label**: The **FilterEngine** adds *candidate or excluded* or a *confidence score*.
4. **Reference Object → Raw '.bib'** for final export.

5.4.2 Data Storage and Format

- **In-Memory Representation:** Python dictionaries or data classes.
- **Optional Caching:** AbstractAugmenter might maintain a local store keyed by `(title, doi)` to avoid repeated queries.
- **Final Output:** Plain text '.bib' with updated `abstract` fields.

6 Interfaces and Integration

6.1 Module Interfaces

1. **InputParser Interface:**

```
parse_bib_files(root_dir: str, config: Config) -> Iterator[ReferenceObject]
```

   It returns an iterator (or list) of `ReferenceObject`.

2. **AbstractAugmenter Interface:**

```
augment_references(refs: Iterable[ReferenceObject],
                   config: Config,
                   scrapers: List[AbstractScraper]) -> None
```

   This function updates each reference in place with an abstract if found.

3. **FilterEngine Interface:**

```
filter_references(refs: Iterable[ReferenceObject],
                  config: Config,
                  classifier: Optional[Classifier]) -> List[ReferenceObject]
```

   Returns references that pass the filter.

4. **Exporter Interface:**

```
export_to_bib(refs: List[ReferenceObject],
              output_path: str,
              config: Config) -> None
```

   Writes the final list of references to a '.bib' file.

6.2 Plugin Integration

- **ScraperPlugins**:

```
class AbstractScraper:
    def get_abstract(self, ref: ReferenceObject) -> Optional[str]:
        ...
```

  The **AbstractAugmenter** can call multiple scrapers in a chain, stopping once an abstract is retrieved.

- **ClassifierPlugins**:

```
class Classifier:
    def load_model(self, model_path: str):
        ...
    def predict(self, text: str) -> float:
        # returns a probability or confidence score
```

  The **FilterEngine** can optionally call this to refine the candidate set.

6.3 External APIs and Integration Points

- **Semantic Scholar API**:
  - Integration via HTTP GET or POST with `requests`.
  - A rate-limit or daily quota might apply; the system tracks request counts, logs errors if exceeded.
- **OpenAlex API**:
  - JSON-based results. The system needs to parse the field that corresponds to the paper's abstract or summary.
- **arXiv API**:
  - XML-based feed results. We parse out the `summary` node for the abstract.
- **Google Scholar Scraping**:
  - Potentially less stable. We handle it carefully with fallback, logging if blocked or captcha appears.

7 Quality Attributes and Tactics

7.1 Reliability and Fault Tolerance

- **Error Handling and Logging:** Each module logs errors in a structured manner (timestamp, severity, reference ID). If `AbstractAugmenter` fails on a reference, it logs the error but does not crash the entire pipeline.
- **Retry Mechanisms:** For external API calls, a short *exponential backoff* can be used if rate limits are encountered.
- **Fallback to Next Service:** If one service fails or returns no data, the system tries the next. This ensures maximum coverage for abstract retrieval.

7.2 Scalability and Performance Tactics

- **Parallel I/O for Augmentation:** Using multi-threading or async to handle potentially thousands of references for external calls.
- **Batch or Bulk Queries (if supported):** Some APIs might allow bulk queries. The system architecture can incorporate a *bulk retrieval plugin* if an API supports it.
- **Efficient Text Search:** The lexicon-based intersection can use *compiled regex* or a *trie-based approach* for speed.
- **Cache Results**: Storing `(title, abstract)` pairs in a local DB to prevent repeated queries in subsequent runs.

7.3 Maintainability

- **Modular Codebase**: Each functional stage is in a separate Python module.
- **Plugin Interfaces**: Clear contracts for scrapers and classifiers reduce friction when adding new ones.
- **Configuration-Driven Behavior**: Minimizes the need to change source code to alter lexicons, concurrency, or external endpoints.

7.4 Security Considerations

- **API Keys or Tokens**:
  - Must be stored in external config files or environment variables, not in source code.
  - Logging must redact any sensitive information.
- **HTTPS Everywhere**: Ensures calls to external services are secure by default.

7.5 Usability Tactics

- **CLI Interface with Clear Help/Docs**: `--help` shows usage, arguments, config paths, etc.
- **Human-Readable Logging**: In addition to structured logs, provide console-friendly logs with color-coding or labeling.
- **Summary Report**: At the end of a run, the system can print a summary: how many references were parsed, how many abstracts were retrieved, how many references made it into the final file.

8 Design Constraints and Rationale

8.1 Key Decisions and Their Rationale

1. **Use of Python:**
   - Rationale: Large ecosystem for text processing, straightforward concurrency for I/O.
   - Trade-off: Python can be slower for CPU-bound tasks, but typically external calls are the bottleneck.
2. **Pipeline Structure vs. Monolith:**
   - Rationale: Separation of concerns among parse, augment, filter, export stages fosters maintainability and clarity.
   - Trade-off: Passing data among stages can add overhead but fosters clarity and testing.
3. **Lexicon + Classification Hybrid:**
   - Rationale: We want high recall (hence broad lexicon-based filter) and high precision (classification to remove false positives).
   - Trade-off: Additional complexity requires labeled data for the classification approach.
4. **Falling back across multiple abstract providers:**
   - Rationale: Minimizes missed opportunities to retrieve abstracts.
   - Trade-off: Requires more complex logic to handle partial or out-of-order successes/failures.

8.2 Limitations

- **Partial Control Over Abstract Quality:** If external APIs return incorrect or truncated abstracts, the system can do minimal validation, but it might still store an erroneous abstract.
- **Data Overlap / Duplicates:** If the same reference is present in multiple '.bib' files with slightly different metadata, the system merges them or picks one. This might introduce duplicates in the final '.bib'.

9 Risks and Technical Debt

9.1 Known Risks

- **External Service Change**: If external API endpoints or formats change, the **AbstractAugmenter** might break, requiring quick updates.
- **Google Scholar Scraping**: Unofficial approach can fail unpredictably due to captcha or UI changes.
- **Large Data Processing Time**: If user has *extremely large* input sets and classification is CPU-heavy, runs could be slow.
- **Classifier Accuracy**: If the classification module is not well-trained, it might incorrectly exclude relevant references.

9.2 Technical Debt Areas

- **No Universal Bulk Query**: We currently do multiple single queries. Bulk or batched queries are not standardized across all external services, representing a potential optimization area.
- **Duplicate Checking Logic**: We may need a robust approach for merging duplicates or near-duplicates across multiple input '.bib' files. This is not thoroughly addressed in the core design yet.
- **Multi-Language Support**: If references are in non-English languages, the lexicon-based or classification approach may degrade. The architecture doesn't fully address multilingual support.

10 Appendices

10.1 Detailed Diagrams or References

Detailed UML-like diagrams (class diagrams, sequence diagrams) will be provided in the *SDS (Software Design Specification)* to maintain separation between architecture-level and implementation-level details.

10.2 Glossary

(See also SRS for domain definitions.)

Scraper Plugin
: A module implementing a standard interface for retrieving data from a specific external source.

Classifier Plugin
: A module that provides advanced ML-based text classification to refine references beyond lexicon matching.

Reference Object
: An in-memory representation of a single bibliographic record, with fields such as title, abstract, authors, etc.

Data Pipeline
: The sequential series of transformations from raw ‘.bib‘ input to curated ‘.bib‘ output.

10.3 References to Related Documents

- **SRS (Software Requirements Specification)** for the comprehensive list of functional and non-functional requirements.
- **SDS (Software Design Specification)** for the detailed module design, data structures, class diagrams, and code-level details.