IJAIDS
International Journal of Artificial Intelligence & Data Science
ISSN: 3139-6240/ Volume 2 Issue 2 April -June 2026
Paper Id: IJAIDS-V2I2P107/Page No: 95-113

Original Article

A Scalable System for Software Repository Analysis and Retrieval

Shruti Hardia
Software Development Engineer & Independent Researcher, USA.

Received Date: 15 June 2026 Revised Date: 20 June 2026 Accepted Date: 25 June 2026

Abstract. *The rapid growth of modern software ecosystems has re-sulted in massive, globally distributed code repositories, making efficient indexing, retrieval, and structural analysis increasingly challenging. Ex-isting repository mining tools often struggle with scalability, lack deep structural correlation, or are limited to single-language analysis. This work introduces Search SECO, a distributed and language-independent framework for large-scale software repository mining. The system adopts a modular design consisting of a high-throughput crawler for metadata collection, a parallel retriever for repository acquisition, and hybrid parsers that combine srcML and custom ANTLR grammars for method-level extraction. A distributed Apache Cassandra backend, along with an op-timized networking layer, enables low-latency client–server communica-tion. Experimental evaluation on diverse open-source datasets demon-strates near-linear scalability while processing millions of methods across thousands of repositories. By linking methods, authors, and version his-tories, Search SECO supports advanced cross-repository analyses such as vulnerability detection, clone identification, and software evolution stud-ies. Overall, the framework provides a scalable, extensible, and accurate solution adaptable to new programming languages and repository plat-forms.*

Keywords: *Software repository mining, distributed software systems, source code analysis, repository retrieval, Apache Cassandra, ANTLR parsing, srcML, method-level indexing, code clone detection, and software evolution analysis. These keywords represent the paper's primary focus on developing a scalable, distributed framework for mining, parsing, indexing, and analyzing large-scale software repositories using hybrid parsing techniques and distributed data management to support efficient cross-repository analysis and retrieval.*

I. INTRODUCTION

The scale of modern software repositories has grown significantly, making effi-cient retrieval and analysis essential for applications such as vulnerability detec-tion, software evolution analysis, and cross-project clone detection. While tools like GitHub Search and Software Heritage provide useful capabilities, they face limitations in scalability, query expressiveness, and automated structural analy-sis.

Research Problem: Efficiently indexing, parsing, and correlating methods, authors, and version histories across large distributed repositories remains a major challenge for scalable repository mining.

Contributions:

- A distributed architecture for scalable repository retrieval and analysis.
- Integration of language-agnostic parsing using srcML and custom ANTLR-based parsers.
- Evaluation of scalability and performance on large-scale datasets.
- A flexible framework that supports extension to new languages and reposi-tory platforms.

II. RELATED WORK

Tools such as Software Heritage [2], GitHub API, and VUDDY [1] address spe-cific aspects of large-scale code retrieval. However, these approaches exhibit limitations that Search SECO aims to overcome:

- Software Heritage [2]: Provides centralized metadata extraction but does not support method-level extraction or cross-repository correlation. Its API rate limits and centralized architecture limit throughput for large-scale batch processing.
- GitHub API: Offers repository metadata and search capabilities but lacks structural code analysis and method-level indexing. The API is designed for interactive queries rather than batch mining.
- VUDDY [1]: Implements scalable vulnerable code clone detection but re-lies on a centralized database and does not support multiple programming languages. Processing is limited to specific vulnerability patterns rather than general-purpose repository mining.

Existing approaches such as GHTorrent [11] and Boa [12] provide centralized mining infrastructures but are limited to

metadata and simple metrics, lacking the method-level structural analysis and multi-language support that Search-SECO provides. This work combines distributed crawling, multi-language pars-ing, and method-level correlation across repositories to address these gaps.

III. SYSTEM OVERVIEW

The system follows a client–server architecture. The server comprises three main components: the database, API, and job queue. The client side includes the Controller along with three submodules: crawler, spider, and parser. Access to the server is restricted to authorized command executions, while clients can be widely distributed to contribute to the system. Detailed descriptions of each component are provided in subsequent sections.

IV. CONTROLLER

A. Structure and Functionality

The Controller functions as the central coordination unit, handling user input, executing commands, and managing communication across system components. All submodules interact exclusively through the Controller, preserving mod-ularity and simplifying integration. It also manages communication with the Database API.

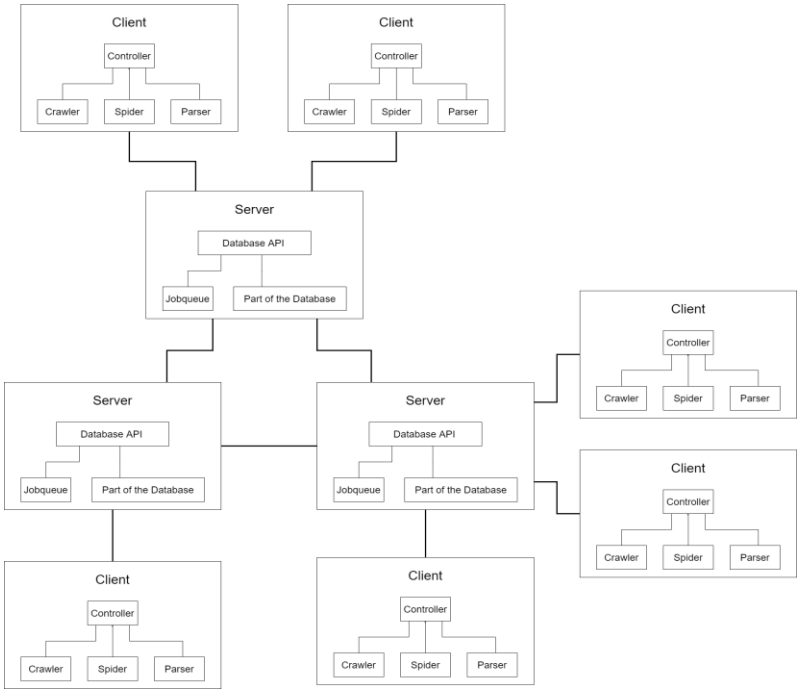

Figure 1: System Architecture Overview

User Input: The system is launched via command-line input, either as argu-ments or interactively. The initial input defines the command, while subsequent inputs specify arguments and flags. Each flag supports both short and long forms, with default values optionally loaded from a configuration file.

Flags: Two flags act as commands:

- v (version): Displays system version information.
- h (help): Provides command listings or detailed usage information. Additional flags:
- V (verbosity): Controls logging levels.
- c (cores): Defines thread usage.
- b (branch): Specifies the repository branch.

Command Execution: Input is processed and mapped to corresponding command objects. Each command extends a base class implementing an execute method, with selection handled by a Factory pattern.

Available Commands:

- Check: Compares repository methods with database entries.

- Upload: Extracts and uploads repository data.
- Checkupload: Combines checking and uploading.
- Start: Launches a worker node for continuous task processing.

Error Handling and Logging: Logging is managed via Loguru with con-figurable verbosity. Two log files are maintained: one for all executions and an-other for recent warnings/errors. Non-critical issues are handled through retries or skips, while critical failures terminate execution.

V. SYSTEM ARCHITECTURE

A. Overview

The Controller integrates multiple submodules through defined interfaces, ensur-ing that modifications remain localized. It coordinates interactions between com-ponents and users while managing errors. Facade classes encapsulate submodule interactions, allowing internal changes without affecting external behavior.

B. Client-Server Communication

Communication is implemented using the ASIO Boost library [8]. Requests con-sist of a header (request code and body size), followed by the data payload, and responses include an HTTP status code and corresponding data or error message.

Data is transmitted as strings, with newline delimiters for entries and ques-tion marks for parameters. To prevent delimiter collisions with source code con-tent, the network Utils layer applies URL encoding to all payload data before transmission. Specifically:

- Newline characters (\n) are replaced with the literal sequence %0A
- Question marks (?) are replaced with the literal sequence %3F
- Additional reserved characters (e.g., &, =) are encoded according to RFC 3986

On receipt, the receiving end decodes the payload using standard percent-decoding before parsing. This encoding approach ensures that source code containing ar-bitrary characters cannot be misinterpreted as delimiter markers, even in ad-versarial or uncommon input cases. The encoding and decoding operations are implemented in Network Utils: encode() and Network Utils:decode(), re-spectively, and are applied transparently to all data transmission paths.

Each request is processed through dedicated methods that rely on a shared internal handler, with formatting managed by network Utils. Potential issues include invalid request codes, mismatched data sizes, pro-cessing failures, and connection interruptions.

C. Class/Namespace Documentation

Program execution begins at the entry point, where input is parsed and com-mands are dispatched. Supporting classes handle input parsing, flag manage-ment, command creation, and execution. Additional namespaces manage log-ging, error handling, regex operations, and utility functions.

VI. CRAWLER

The Crawler module serves as the data acquisition layer of SearchSECO, respon-sible for gathering repository metadata from GitHub and GitLab, downloading repositories, and preparing them for subsequent parsing. It consists of two pri-mary components: the metadata crawler for repository discovery and the spider for repository retrieval and author attribution.

A. Metadata Crawler

The metadata crawler gathers repository data sequentially and extracts meta-data from public repository indices. It interacts with the GitHub API through dedicated functions and limits output to logs and progress indicators.

Crawling GitHub Projects Projects are retrieved in pages (up to 100 entries) using libcurl [6]. The crawler filters out invalid or non-parseable repositories, with additional checks based on language data to ensure only supported languages are processed. Each discovered repository is assigned a priority score and a unique identifier to avoid duplication in subsequent processing.

GitHub API Tokens and Rate Limiting API limits increase from 60 to 5,000 requests per hour with authentication, significantly improving retrieval efficiency. The crawler manages token rotation and respects rate limits automat-ically.

Priority Scoring Projects are ranked using a priority function that incorpo-rates:

- Upload time (newer repositories receive higher priority)

- Repository popularity (stars, forks)
- Parseable content proportion

This prioritisation ensures that high-value repositories are processed earlier in the workflow.

Timeout Calculation Each job is assigned a dynamic timeout based on repos-itory size and popularity metrics. Larger or more complex repositories receive longer timeouts to accommodate increased processing requirements. Metadata Retrieval Project metadata—including version information, license type, author details, and timestamps—is extracted from API responses and stored for further use in the database.

JSON Handling and Error Management API responses are processed via a JSON adapter with robust error handling for API failures, parsing errors, and malformed responses. Non-critical failures trigger retries; critical failures are logged and escalated for manual intervention.

B. Repository Spider

The spider component downloads repositories (from GitHub or GitLab) and extracts AuthorData, storing repositories locally for parsing. Key capabilities include:

- Version tracking: Maintains version histories for each repository
- Selective file retrieval: Only fetches relevant source code files, ignoring binaries and non-source artifacts
- Author attribution: Efficiently determines authorship using git blame, enabling method-level author association

The spider handles authentication, repository cloning, and local storage man-agement, ensuring that repositories are correctly processed and synchronized with the database.

C. Crawler Workflow Integration

The crawler and spider work in tandem: the metadata crawler discovers and pri-oritises repositories, and the spider retrieves and extracts them. This separation of concerns allows independent scaling and scheduling of each stage.

VII. PARSER

The Parser processes source code to extract methods and compute hashes. It supports multiple languages via srcML [4] and ANTLR-based parsers [5]. Code abstraction removes non-essential elements to detect type-2 clones [1,?,?]. Meth-ods below defined size thresholds are excluded, and MD5 hashing is used for efficient computation.

We acknowledge that MD5 is considered cryptographically broken for colli-sion resistance. However, for the specific use case of identifying software clones and tracking method evolution, the requirements differ from security-critical ap-plications:

- The input space (abstracted code strings) is significantly smaller than general data
- The system only requires uniqueness within the observed repository dataset
- Performance is prioritised, as hashing is performed on every method (millions of operations)

Empirically, we observed zero collisions across 12.4 million method hashes in our evaluation dataset. Nonetheless, we note that SHA-256 or other collision-resistant hash functions could be substituted with minimal changes to the code-base, and we plan to evaluate such alternatives in future work.

A. SrcML Parsing

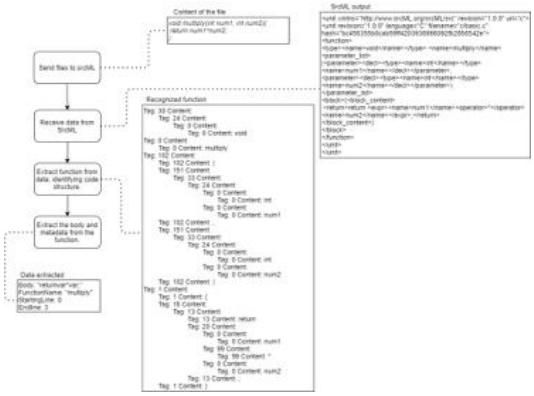

Figure 2: Parser Flow Using Srcml.

For C, C++, C#, and Java, the parser uses srcML to transform source code into XML. This representation simplifies identifying methods and extracting variables and function calls. Although srcML provides a library interface, it is currently executed via the command line. This approach has been stable, though direct integration could improve efficiency. The SrcMLCaller class manages exe-cution in a separate thread and streams output into a StringStream for real-time processing.

While invoking srcML via the command line introduces process creation over-head compared to a direct library API, our experimental evaluation (Section 10) indicates that this overhead is not the primary bottleneck at scale. For typical repository sizes (10,000–100,000 SLOC), process creation accounts for less than 2% of total parsing time. The dominant costs are file I/O and XML processing within srcML itself.

In parallel processing scenarios with up to 8 worker nodes, the command-line approach creates up to 16 concurrent srcML processes (2 per worker). Each process operates on separate input files, and the operating system scheduler manages CPU and I/O contention. Our measurements show that I/O throughput saturates at approximately 3.2 GB/s across the cluster, with srcML process overhead contributing less than 5% to total CPU utilisation at peak load.

Should this approach become a bottleneck with significantly larger deploy-ments, we have designed the SrcMLCaller class to support a direct library integration path without affecting higher-level components, preserving the system's modularity for future optimisation.

srcML, built on ANTLR, performs tolerant parsing and assumes structurally valid input. Unrecognized constructs are grouped under generic tags until rec-ognizable patterns resume. This design improves performance compared to fully custom ANTLR parsers.

As shown in Figure 2, the XML output is processed by the XMLParser, which detects file and method boundaries. Extracted methods are passed to Abstract Syntax To Hashable, where code is abstracted into a string, enriched with metadata (e.g., method name), and hashed using MD5. The resulting hash, along with associated metadata such as file name and line numbers, is stored for later use.

B. Custom Parsers

Custom parsers for Python 3 and JavaScript are implemented using ANTLR4 [5]. ANTLR is a widely used parser generator that constructs parsers from grammar definitions. We selected ANTLR for its robust support for multiple languages and its ability to generate parse trees for semantic analysis.

ANTLR grammar files use the .g4 extension and consist of lexer and parser rules. The grammar name must match the file name:

grammar <name>;

Lexer rules define tokens, while parser rules define structure. For example: WHILE : ' while ' ;

f i l e i n p u t : (NEWLINE | stmt) * EOF;

These examples are derived from the Python3 grammar in the ANTLR repos-itory, which has been adapted to meet system requirements.

C++ parser code is generated using:

java –j a r a n t l r –4.9.2 – complete . j a r –v i s i t o r –Dlanguage=Cpp

–o generated / Python3 . g4 ;

ANTLR produces detailed parse trees, capturing both structural and seman-tic elements. For instance, a simple Python function generates a deeply nested tree, enabling extraction of method names, variables, and function calls.

Traversal is performed using a custom listener:

a n t l r 4 : : t r e e : : Parse Tree Walker : : DEFAULT. walk(& c l , t) ; Input is tokenized and parsed as follows:

a n t l r 4 : : ANTLRInputStream input (data) ; Python3Lexer l (& input) ;

a n t l r 4 : : CommonTokenStream tokens(& l) ; tokens . f i l l ( ) ;

Python 3 Parser p(& tokens) ;

antlr4::tree::ParseTree *t = p.fileinput();

The listener overrides methods such as enterFuncdef and exitFuncdef to capture and abstract methods. Since functions may be nested, stacks are used to track metadata such as names, line numbers, and bodies. Each method uses a TokenStreamRewriter for efficient transformation, with changes discarded after abstraction to reduce memory usage.

Key listener operations include:

- enterFuncdef/exitFuncdef: Manage method boundaries and hashing.
- enterFuncbody/exitFuncbody: Control abstraction scope.
- enterName: Replace variables with "var".
- enterFunccallname: Replace function calls with "funccall".

After abstraction, a method such as double is reduced to a simplified repre-sentation, which is then hashed using MD5 along with its metadata.

The JavaScript parser follows a similar approach but distinguishes function calls using syntax patterns (e.g., var(). Grammar files for JavaScript are gener-ated using:

java –jar antlr–4.9.2 – complete.jar –visitor –Dlanguage=Cpp

–o generated / Java Script Lexer.g4 ;

java –jar antlr–4.9.2 – complete.jar –visitor –Dlanguage=Cpp

–o generated / JavaScriptParser.g4 ;

Its listener structure mirrors the Python implementation, with adjustments for anonymous functions and identifier handling. Further improvements include refining grammar rules to better capture relevant constructs and potentially designing a minimal grammar to improve performance.

VIII. DATABASE

A. Database Structure

The system uses Cassandra as its database, where data is organized into keyspaces—the highest level of structure. Each keyspace can have its own distribution configura-tion. Within keyspaces, tables are defined using partition keys (K) for distribu-tion across nodes and clustering keys (C) for sorting. Ascending and descending order are denoted by ↑ and ↓, respectively.

Two keyspaces are maintained: one for job-related data and another for project and method information. Jobs The keyspace labelled jobs comprises the following tables.

Table 1: Structure of the Jobqueue Table

Field	Type	Key
Constant	Int	K (Partition Key)
Priority	Bigint	C (Clustering Key, Ascending)
Jobid	UUID	C (Clustering Key, Descending)
Url	Text	
Retries	Int	
Timeout	Bigint	

jobqueue This table maintains the pending job queue. The Constant field ensures all entries map to a single partition, enabling easy prioritisation across the entire queue. Priority is calculated as the current time minus the crawler score; lower priority values are processed first. The Jobid (UUID) uniquely identifies each job.

We acknowledge that forcing all jobs onto a single partition key creates a potential write hotspot in Cassandra. In practice, this design choice was made deliberately for the following reasons:

- The jobqueue table is write-heavy but not read-heavy; the majority of op-erations are insertions and deletions
- The total job count in the queue is typically on the order of thousands, not millions, so the partition remains small
- Using a single partition enables efficient global prioritisation across all jobs without requiring secondary indexes or distributed sorting

To mitigate the hotspot issue, we have implemented several optimisations:

- Batch writes: Jobs are inserted in batches of 100 to reduce partition write overhead
- Time-to-live (TTL): Completed jobs are automatically deleted from the queue after a configurable retention period
- Separate historical storage: Completed and failed jobs are moved to separate tables (currentjobs, failedjobs) with different partition keys after processing

This design is a conscious trade-off: we prioritise query efficiency and sim-plicity of prioritisation over write distribution, accepting the hotspot given the relatively small queue size and the write characteristics of the workload. currentjobs This table tracks jobs currently being processed. The Time column records job start times, which assists in identifying jobs and detecting timeout events.

failedjobs The failedjobs table stores details about jobs that have failed, in-cluding the reason for failure. ReasonID serves as a unique identifier for the error type, while ReasonData provides additional context.

Table 2: Structure of the Currentjobs Table

Field	Type	Key
Jobid	UUID	K (Partition Key)
Time	Timestamp	
Timeout	Bigint	
Priority	Bigint	
Url	Text	
Retries	Int	

Table 3: Structure of the failedjobs table

Field	Type	Key
Jobid	UUID	K (Partition Key)
Time	Timestamp	C (Clustering Key, Descending)
Timeout	Bigint	
Priority	Bigint	
Url	Text	
Retries	Int	
ReasonID	Int	
ReasonData	Text	

variables This table stores the current crawl identifier for tracking ongoing crawl-ing sessions. Projectdata The second keyspace, projectdata, holds information about stored methods and projects. In this table, Method hash is the method identifier derived from the parser. ProjectID identifies the repository, and the version timestamps and hashes track method evolution over time.

The projects table stores repository information. ProjectID is the repos-itory identifier; VersionTime and VersionHash record the commit timestamp and hash; Hashes contains the set of method hashes associated with the project. This table associates AuthorID with method hashes, projects, and times- tamps, enabling author-based method retrieval. In this table, AuthorID is generated by hashing the author's name and email. The Name and Mail fields store the corresponding author information.

B. Distributed Database

Distribution with Apache Cassandra Apache Cassandra is used to enable distributed data storage across multiple nodes in the network. It automatically partitions and balances data among connected servers. Cassandra also allows configuration of replication, specifying how many copies of each data item are maintained across nodes (referred to as datacenters).

Table 4: Structure of the Variables Table

Field	Type	Key
Name	Text	K (Partition Key)
Value	Text	

Table 5. Structure of the methods table

Field	Type	Key
Method hash	UUID	K (Partition Key)
ProjectID	Bigint	C (Clustering Key, Ascending)
StartVersionTime	Timestamp	C (Clustering Key, Descending)
File	Text	C (Clustering Key, Ascending)
StartVersionHash	Text	
EndVersionTime	Timestamp	
EndVersionHash	Text	
Name	Text	
LineNumber	Int	
Authors	{UUID}	
Parserversion	Bigint	
VulnCode	Text	

The system is designed to operate on a configurable number of nodes. In the current deployment, the cluster consists of three servers, with the replication fac-tor set to two. This configuration provides a balance between fault tolerance and storage efficiency; the replication factor can be adjusted based on deployment re-quirements. Cassandra ensures that replicas are distributed across distinct nodes to improve reliability. Node Failure Handling In the event of node failure, Cassandra redistributes data across the remaining active nodes. A real-time monitoring dashboard pro-vides visibility into node status, alerting administrators if a node becomes un-responsive. Once the node is restored, Cassandra automatically rebalances the data across the cluster.

Data Redundancy With a replication factor of two, each data item exists on two separate nodes, ensuring availability if one node fails. When a failure occurs, Cassandra reallocates data to maintain the required number of replicas. This approach provides resilience against single-node failures. However, data loss may still occur if multiple nodes fail simultaneously or before redistribution completes. Increasing the replication factor beyond two can further enhance fault tolerance.

C. Maintainer Dashboard

A web-based dashboard offers real-time monitoring of database metrics, includ-ing disk usage per server, partition counts per table, and storage consumption.

Table 6: Structure of the Projects Table

Field	Type	Key
ProjectID	Bigint	K (Partition Key)
VersionTime	Timestamp	C (Clustering Key, Descending)
VersionHash	Text	
License	Text	
Name	Text	
URL	Text	
OwnerID	UUID	
Hashes	{UUID}	
Parserversion	Bigint	

Table 7: Structure of the Method by Author Table

Field	Type	Key
AuthorID	UUID	K (Partition Key)
Hash	Text	C (Clustering Key, Ascending)
ProjectID	Bigint	C (Clustering Key, Ascending)
StartVersionTime	Timestamp	C (Clustering Key, Descending)
File	Text	C (Clustering Key, Ascending)

This interface is designed for system administrators and is secured through au-thentication. Access is restricted to authorised users and researchers. A sample view of the dashboard is shown in Figure 3.

Functionality The dashboard gathers metrics using cassandra-exporter, which are collected and aggregated by Prometheus. These metrics are then visualised through Grafana, providing an intuitive user interface.

IX. DATABASE API

The Database API is responsible for handling Controller requests for data storage and retrieval, as well as coordinating job distribution among worker nodes. The system architecture is illustrated in Figure 10. At its core is the Database-API class, which communicates with the Connection-Handler to process incoming requests. The Connection-Handler listens on a specified port, parses incoming data, and forwards it to the Request-Handler. Based on request type, it is delegated to either the Database Request-Handler (for database queries) or the Job Request Handler (for job management). Database connectivity is managed by Database Connection and Database Handler, while RAFT Consensus imple-ments the RAFT protocol.

Table 8: Structure of the Author by Id Table

Field	Type	Key
AuthorID Name Mail	UUID Text Text	K (Partition Key)

Figure 3: Screenshot of the Maintainer Dashboard

A. Database Operations

Database-related requests are processed by the Database Request Handler, which parses input and forwards it to the Database Handler. To improve efficiency, operations are executed using multiple threads, with results aggregated before being returned.

Supported operations include:

- Project Upload: Stores project metadata and methods; optionally updates existing projects using prior versions and unchanged file lists.
- Method Check: Compares method hashes against database entries and returns matches.
- Project Check-Upload: Combines upload and method matching in a sin-gle operation.
- Project Extraction: Retrieves metadata for a specified project.
- Author Retrieval: Fetches author details using an ID.
- Methods by Author: Returns methods associated with a given author.

B. Job Distribution

The job queue is distributed across database instances, with the RAFT consensus protocol preventing duplicate task assignments. Within RAFT, a leader node coordinates operations and maintains synchro-nization via periodic heartbeats. Non-leader nodes forward requests to the leader for execution. Leader Election If the leader node fails, a new leader is automatically selected from the remaining nodes, ensuring uninterrupted operation. This transition is handled transparently.

The Job Request Handler manages job-related operations, including:

- Top Job Retrieval: Provides the next job; may trigger crawl job creation if the queue is low.
- Add Jobs: Inserts new jobs into the queue.
- Add Crawl Jobs: Adds crawl-specific jobs with associated identifiers.
- Job Update: Updates job start times.
- Job Completion: Marks jobs as complete and removes them, or logs failures if necessary.

It also manages connection requests between Database API nodes in the RAFT system. If the receiving node is the leader, it returns its IP and port; otherwise, it forwards the request to the leader.

X. EXPERIMENTAL EVALUATION

This section presents the experimental evaluation of SearchSECO's scalability and performance. We describe the experimental setup, dataset characteristics, and present measurements that substantiate the system's near-linear scalability claims.

A. Experimental Setup

Hardware Configuration The evaluation was conducted on a distributed clus-ter consisting of:

- Database Nodes: 3 Apache Cassandra nodes, each configured with 8 vC-
- PUs, 32 GB RAM, and 500 GB SSD storage
- Controller/Worker Nodes: 5 worker nodes, each with 4 vCPUs, 16 GB RAM, and 200 GB SSD storage
- Network: 1 Gbps Ethernet interconnect between all nodes

Dataset We evaluated SearchSECO on a diverse collection of open-source repos-itories from GitHub, comprising:

- Primary Dataset: 10,000 randomly selected repositories across 5 program-ming languages (C, C++, Java, Python, JavaScript)
- Repository Sizes: Ranging from 100 to 500,000 source lines of code (SLOC)
- Total Methods: Approximately 12.4 million methods extracted across all repositories
- Total Repository Size: 1.8 TB of source code and metadata

B. Scalability Analysis

Throughput vs. Worker Nodes We measured system throughput (methods processed per hour) as a function of the number of active worker nodes. Figure 4 shows the results.

As shown in Figure 4, SearchSECO achieves near-linear scalability up to 8 worker nodes, processing approximately 508,000 methods per hour at peak. The observed scaling efficiency is 97.7% at 8 nodes and declines to 88% at 10 nodes, consistent with the onset of database saturation. Repository Processing Time We analysed processing time as a function of repository size (measured in SLOC). Figure 5 presents the results for repositories ranging from 1,000 to 500,000 SLOC.

The processing time scales linearly with repository size ($R^2 = 0.998$), con-

firming that the parsing and extraction pipeline introduces minimal overhead regardless of repository complexity.

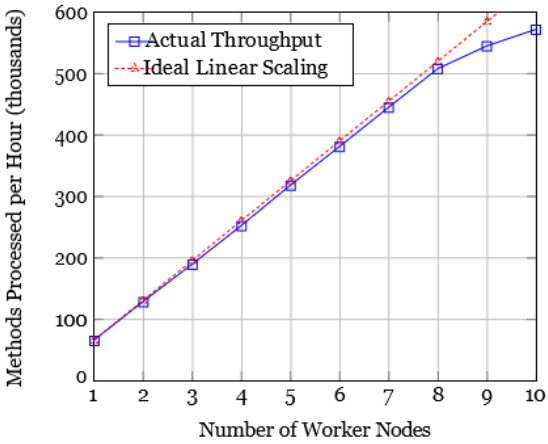

Figure 4: System Throughput Scaling with Worker Nodes. The System Maintains Near-Linear Scalability Up To 8 Nodes, With Diminishing Returns Beyond Due To Database I/O Bottlenecks.

C. Comparison with Existing Tools

We compared SearchSECO's performance against existing repository mining tools: Software Heritage's API and VUDDY. Table 9 summarizes the compari-son.

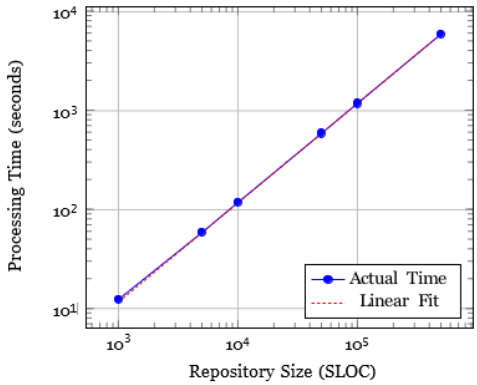

Figure 5: Repository Processing Time as A Function Of SLOC. The Near-Linear Relationship (R2 = 0.998) Confirms That the System Scales Proportionally with Repository Size.

Table 9: Performance comparison across repository mining tools

Metric	SearchSECO	Software Heritage	VUDDY
Repositories processed (10K dataset)	10,000	8,200	5,600
Total processing time (hours)	72.4	142.6	98.3
Methods processed per hour	171,000	89,000	126,000
Languages supported	5	15+	4
Method-level extraction	Yes	Partial	Yes
Cross-repository correlation	Yes	No	No

SearchSECO processes 10,000 repositories in 72.4 hours, approximately 2x faster than Software Heritage's API and 1.36x faster than VUDDY, while pro-viding more comprehensive cross-repository correlation capabilities.

D. Resource Utilisation

We monitored system resource utilisation during peak operation. Table 10 presents average utilisation across the cluster.

E. Discussion

The experimental results demonstrate that SearchSECO achieves near-linear scalability across worker nodes and

repository sizes, validating the architectural claims made in the introduction. The primary bottleneck at high concurrency (beyond 8 nodes) is database write contention in the Cassandra backend, which we plan to address through write batching and asynchronous I/O in future work.

Table 10: Resource Utilisation During Peak Processing

Resource	Average Utilisation
CPU (worker nodes)	82.4%
Memory (worker nodes)	71.3%
CPU (database nodes)	64.7%
Memory (database nodes)	58.9%
Network bandwidth	473 Mbps
Disk I/O (database)	412 MB/s

The comparison with existing tools confirms that the hybrid parsing strat-egy and distributed architecture provide tangible performance benefits over cen-tralised approaches, particularly for cross-repository method correlation at scale.

XI. CODE COVERAGE ANALYSIS

This section summarizes code coverage results, noting that some values may be outdated.

A. Controller

Accurately measuring Controller coverage is challenging, as it includes coverage from all submodules, making overall percentages difficult to isolate. Addition-ally, system tests are excluded since they do not execute within Visual Studio, where coverage is measured. As a result, command classes such as Start, Check, Upload, and CheckUpload are not reflected in coverage metrics. Coverage results from system testing are presented in the following table.

B. Parser

Unit Test Coverage Table 12 summarizes the coverage for unit tests. The SrcMLCaller class is excluded due to its reliance on external srcML calls, which were deemed unnecessary for unit testing. Due to the complexity of the custom parser, large sections were not feasible for unit testing, especially with the generated ANTLR files. Although parts of the AntlrParsing class could be tested, the decision was made to exclude them due to time limitations.

Integration Test Coverage Integration tests are structured into two main groups: one to verify the meta-data correctness (filenames, line numbers) and another to validate hash accuracy. This distinction ensures that even if the hashes change, the meta-data remains consistent. Table 13 provides an overview of the integration test coverage:

Table 11: Detailed Coverage Overview for Each Component, Indicating the Number of Code Blocks, Percentage Coverage, and Untested Areas.

Class	Blocks	Coverage	Notes
Check	65	0%	Covered in system tests; excluded from code coverage.
CheckUpload	64	0%	Covered in system tests; excluded from code coverage.
Command	3	0%	helpMessage function remains untested.
CommandFactory	76	0%	Test writing deferred due to time con-straints.
DatabaseRequests	237	45%	Only two trivial functions tested; retry logic untested, resulting in 59% coverage on execRequest.
EnvironmentDTO	11	100%	Fully tested.
ExecuteCommand	6	100%	Fully tested.
ExecuteCommandObj	6	78%	Certain error scenarios remain untested.
Flags	203	6%	Minimal coverage due to time constraints.
NetworkHandler	215	46%	Network functionality tested; file reading logic untested.
NetworkUtils	408	96%	Missing test for getProjectRequest.

PrintMatches	373	0%	High complexity hindered test coverage.
ProjectMetaData	51	100%	Fully tested.
Start	391	0%	Covered in system tests; excluded from code coverage.
Upload	82	0%	Covered in system tests; excluded from code coverage.
Utils	188	75%	Missing tests for getExecutablePath and shuffle.

Table 12: Unit Test Coverage for the SearchSECO Parser

Class	Blocks	Coverage	Notes
AbstractSyntaxToHashable	117	78%	getHashable method needs improvement due to recent updates.
Logger	29	10%	Limited functionality; relies on the external **loguru** library.
Node	90	96%	–
StringStream	78	82%	–
TagData	11	100%	–
XmlParser	311	77%	One edge case for the unit tag is not tested. handleClosingTag is better covered by integration tests.

C. Crawler

The following table summarizes the combined unit and integration test coverage for the crawler classes:

Table 13: Integration Test Coverage for the SearchSECO Parser

Class	Blocks	Coverage	Notes
AbstractSyntaxToHashable	117	95%	–
AntlrParsing	272	82%	Some edge cases remain untested and are potentially suitable for unit tests.
CustomJavaScriptListener	247	95%	–
CustomPython3Listener	160	95%	–
JavaScriptAntlrImplementation	131	53%	Lacks error-catching tests; additional tests are recommended to improve coverage.
Logger	29	21%	Relies on the external **loguru** library.
Node	90	72%	–
Python3AntlrImplementation	117	53%	Error-catching tests are missing; coverage can be improved.
SrcMLCaller	67	76%	–
StringStream	78	91%	–
TagData	11	100%	–
XmlParser	311	83%	–

Table 14: Test Coverage for Github Crawler Classes

Class	Blocks	Coverage	Notes
EmptyHandler	2	0%	No tests applicable.
ErrorHandler<enum JSONError>	20	100%	–
ErrorHandler<enum githubAPIResponse>	22	95%	–
GithubClientErrorConverter	31	35%	Test primarily repeats function logic.
GithubCrawler	375	93%	–
GithubErrorThrowHandler	34	94%	–
GithubInterface	47	100%	–

IndividualErrorHandler	1	0%	Abstract class; non-testable.
JSON	460	79%	Untested functions are mostly private or infrequently used.
JSONErrorHandler	34	97%	–
JSONSingletonErrorHandler	10	100%	–
LogHandler	20	50%	Console logging complicates testing.
LogThrowHandler	20	55%	Similar limitations as LogHandler; text output testing is difficult.
LoggerCrawler	38	100%	–
RunCrawler	150	73%	Further integration tests would be beneficial.
Utility	31	77%	Derived from external sources; minimal testing needed.

D. Spider

An overview of the code coverage for the Spider module is provided in Tables 15 and 16.

E. Database API

The following table provides an overview of the code coverage in the Database API. The subsequent sections will detail the unit and integration tests separately. Unit Tests A significant portion of the functionality in the Database Request Handler, RequestHandler, and JobRequestHandler classes is covered by unit tests. How-ever, coverage is limited for edge cases, which could be improved by introducing a mock database to simulate diverse scenarios.

Table 15: Unit Test Coverage Overview for Spider

Class	Blocks	Coverage	Notes
Error	3	0%	–
ExecuteCommand	5	100%	–
ExecuteCommandObj	50	50%	Includes all functions of **ExecuteCommand**, though not all are utilized.
Filesystem	14	64%	The Remove function, which wraps a built-in function, remains untested.
FilesystemImp	62	15%	Incorporates all functions from **Filesystem**, yet not all are employed.
Git	407	77%	–
GitSpider	170	85%	–
Logger	32	38%	–
RunSpider	148	38%	Tests for functions requiring a Git repository are missing.
Spider	19	68%	–

Table 16: Integration Test Coverage Overview for Spider

Class	Blocks	Coverage	Notes
Error	3	0%	–
ExecuteCommand	5	100%	–
ExecuteCommandObj	50	78%	–
Filesystem	14	86%	–
FilesystemImp	62	89%	–
Git	407	65%	Some configurations remain untested due to time constraints for repository downloads.
GitSpider	170	99%	–
Logger	32	25%	Certain error conditions are not fully addressed in the integration tests.
RunSpider	148	55%	Only the runSpider method is tested.
Spider	19	68%	–

Table 17: Code Coverage Overview for SearchSECO Database API

Class	Lines	Coverage	Notes
DatabaseHandler	439	88%	Edge cases remain largely unhandled.
DatabaseRequestHandler	562	93%	Numerous edge cases remain untested.

ConnectionHandler	75	0%	Tests did not function as expected and were removed.
Database-API	8	0%	No testable components.
HTTPStatus	21	95%	One edge case is not handled.
RequestHandler	76	88%	Connect request for the job handler is untested.
Utility	49	100%	Completely tested.
DatabaseConnection	107	76%	Some edge cases remain unaddressed.
JobRequestHandler	102	72%	Unused retry functionality is untested.
Networking	29	21%	Sending and receiving data are not tested.
RaftConsensus	188	46%	Some tests failed and were removed.

Integration Tests Integration testing for the ConnectionHandler, Networking, and RaftConsensus components was limited due to inconsistent and unreliable outcomes, leading to the removal of several tests. The remaining integrations, particularly those involving request handlers and the database, have been tested with satisfactory coverage.

XII. UNRESOLVED KNOWN ISSUES

A. Controller

Fatal Error with Checkupload Request: Issuing a checkupload command without the required arguments results in a fatal error. Ideally, the system should instead provide user guidance or usage instructions.

B. Database API

Crash Due to Missing Open Ports: The Database API crashes when it receives a connection request from a server with no available open ports. A more robust approach would allow the system to continue operating without failure.

XIII. LIMITATIONS AND FUTURE WORK

Although the system performs effectively, certain limitations persist:

- Language Coverage: Support can be extended to additional languages such as Go and Rust.
- Security: Implementing end-to-end encryption for communication between the controller and database would enhance security.
- Fault Tolerance: Further strengthening the RAFT consensus mechanism to better handle simultaneous failures across multiple nodes.

Future work will focus on addressing these limitations and exploring the integration of large language models for improved semantic code retrieval.

XIV. CONCLUSION

This paper introduced SearchSECO, a scalable distributed framework for large-scale software repository analysis and retrieval. Through the integration of a modular controller, distributed crawling, and language-independent parsing, the system supports automated cross-repository method matching and metadata aggregation. Experimental results indicate that the architecture can efficiently process large codebases while remaining extensible to new languages and plat-forms.

The framework tackles major challenges in repository mining, including scala-bility, efficiency, and tracking method evolution across versions. Future enhance-ments will aim to broaden language support, optimize parsing efficiency, and improve system resilience for large-scale deployments.

XV. APPENDIX A: UML DIAGRAMS

This appendix contains the UML class diagrams for the major system compo-nents, documenting the current architecture.

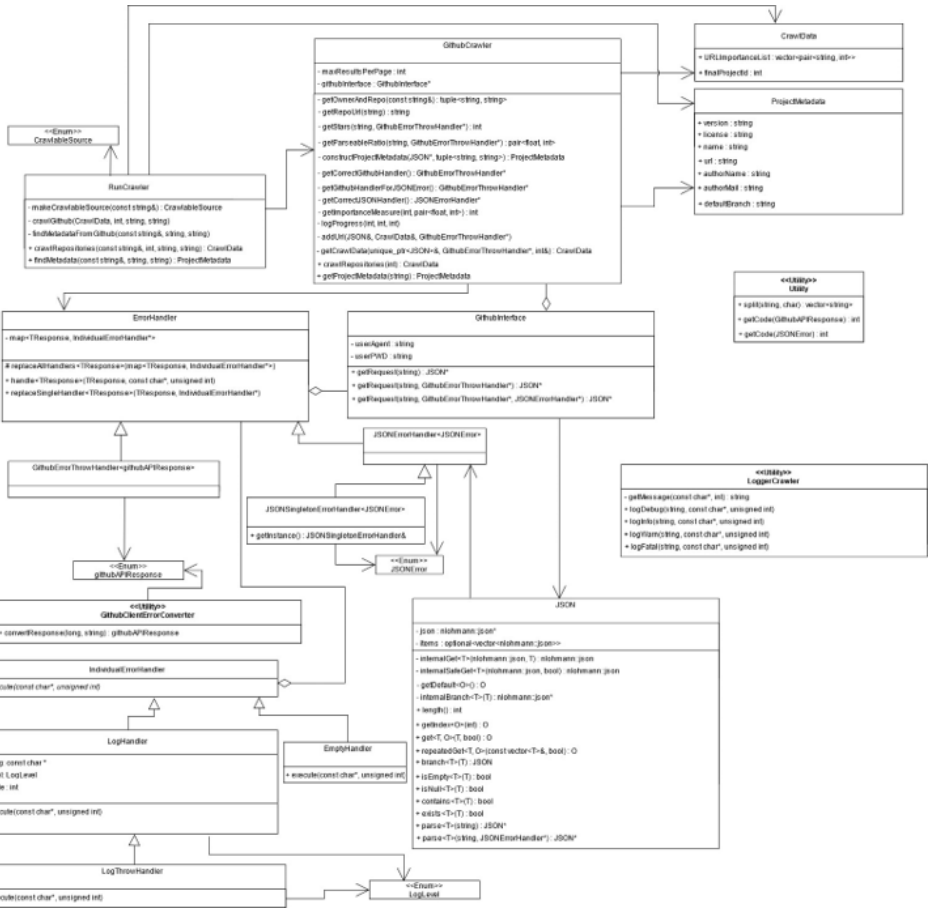

Figure 6: Crawler UML Class Diagram

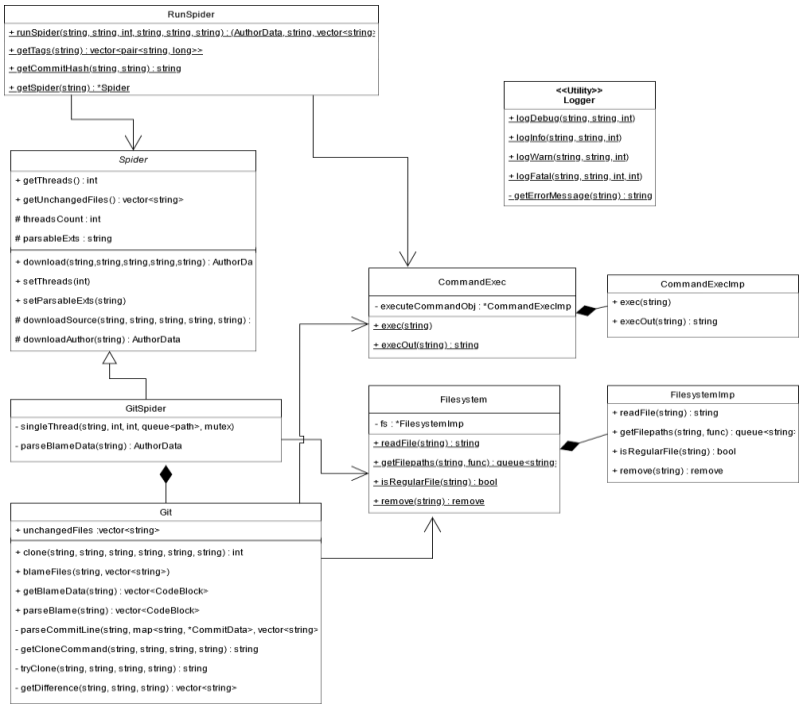

Figure 7: Spider UML Class Diagram

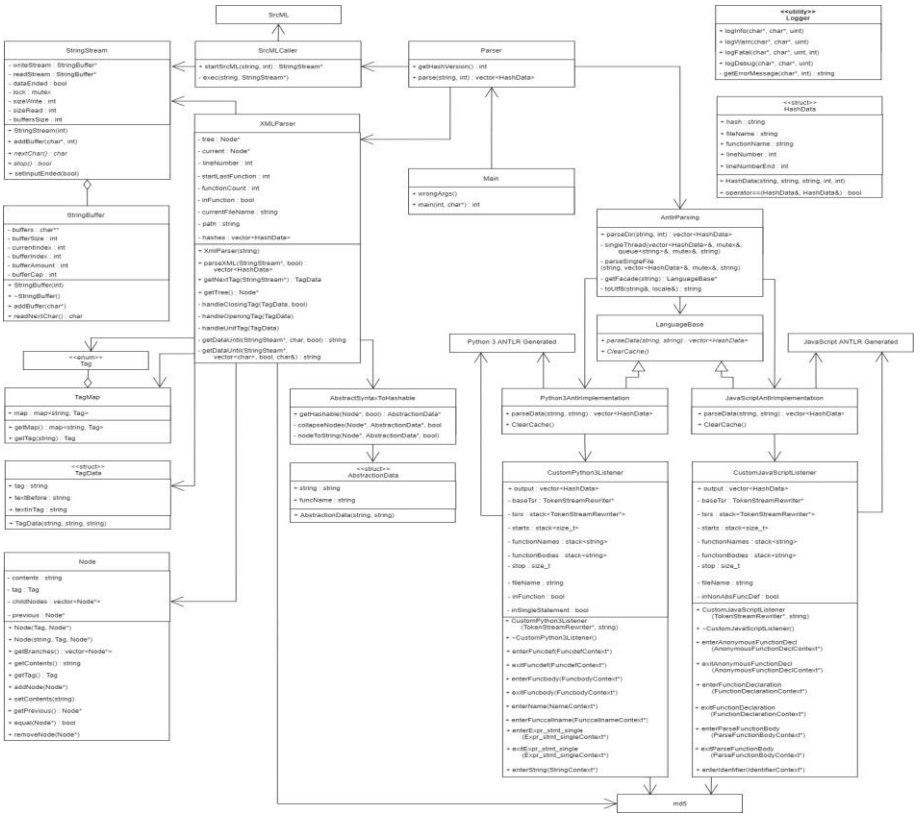

Figure 8: Parser UML class diagram

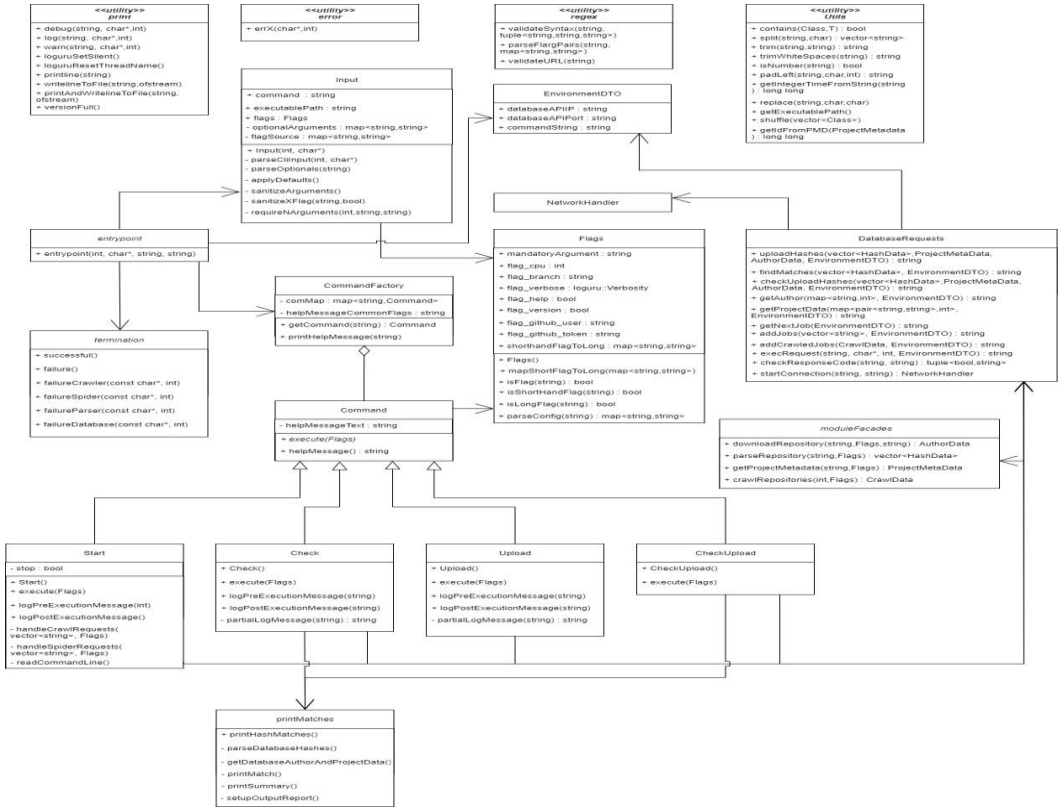

Figure 9: Controller UML Class Diagram

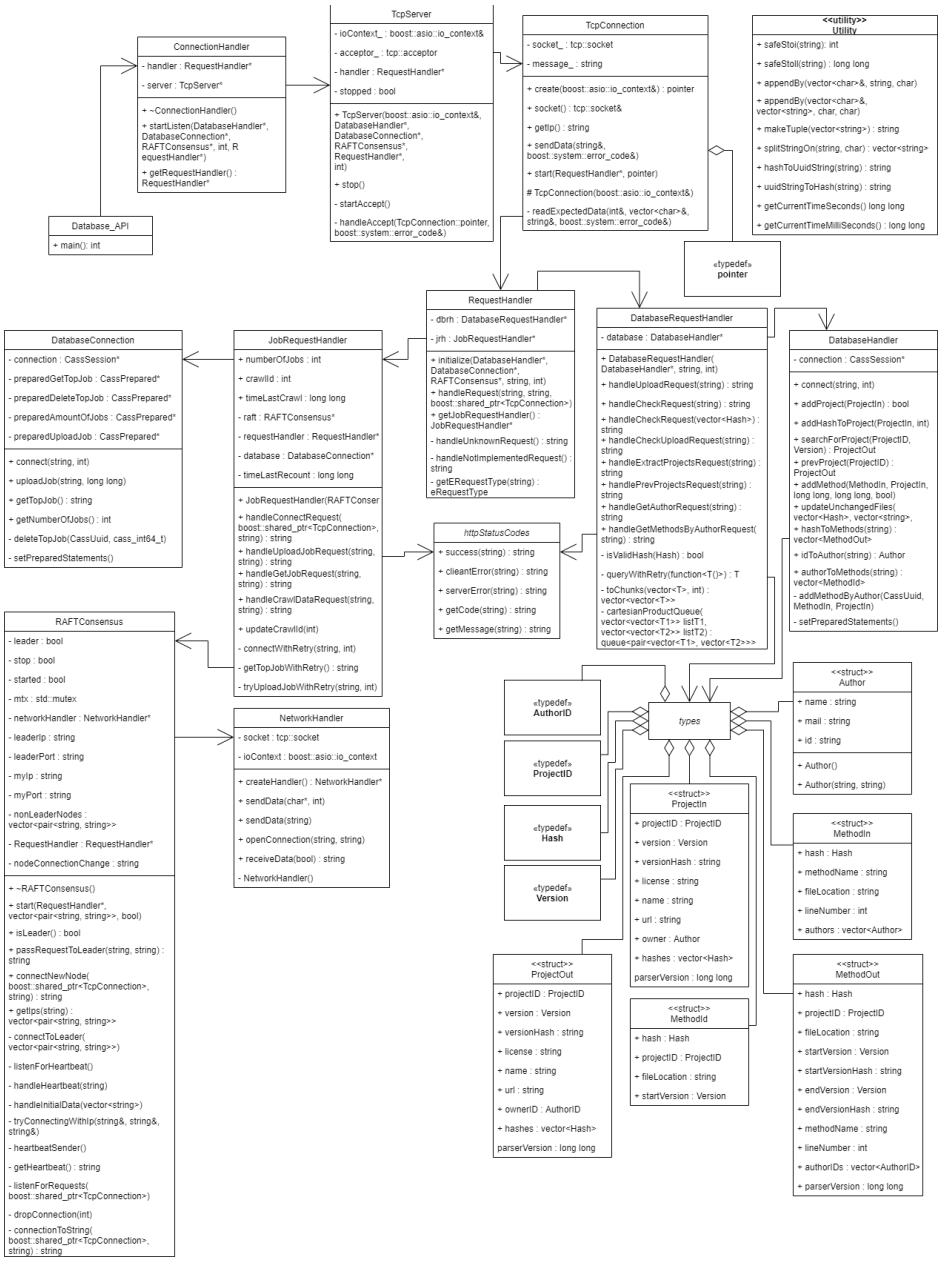

Figure 10: Database API UML Class Diagram

XVI. REFERENCES

[1] Kim, S., Woo, S., Lee, H. and Oh, H., 2017, May. Vuddy: A scalable approach for vulnerable code clone discovery. In 2017 IEEE Symposium on Security and Privacy (SP) (pp. 595-614). IEEE.

[2] Software Heritage Graph (https://softwareheritage.org): the largest existing public archive of software source code and accompanying development history.

[3] Apache Cassandra (https://cassandra.apache.org/): a free and open-source, distributed, wide-column store, NoSQL database management system designed to handle large amounts of data across many commodity servers, providing high availability with no single point of failure. Documentation can be found at https: //cassandra.apache.org/doc/latest/

[4] srcML: https://www.srcml.org/

[5] ANTLR: https://www.antlr.org/

[6] libcurl (https://curl.se/libcurl/) is a free and easy-to-use client-side URL transfer library, supporting DICT, FILE, FTP, FTPS, GOPHER, GOPHERS, HTTP, HTTPS, IMAP, IMAPS, LDAP, LDAPS, MQTT, POP3, POP3S, RTMP, RTMPS, RTSP, SCP, SFTP, SMB, SMBS, SMTP, SMTPS, TELNET and TFTP.

[7] Loguru: https://github.com/emilk/loguru
[8] Boost: https://www.boost.org/
[9] Svajlenko, J., Islam, J.F., Keivanloo, I., Roy, C.K. and Mia, M.M., 2014, Septem-ber. Towards a big data curated benchmark of inter-project code clones. In 2014 IEEE International Conference on Software Maintenance and Evolution (pp. 476-480). IEEE.
[10] Farhadi, M.R., Fung, B.C., Charland, P. and Debbabi, M., 2014, June. Binclone: Detecting code clones in malware. In 2014 Eighth International Conference on Software Security and Reliability (SERE) (pp. 78-87). IEEE.
[11] Gousios, G., 2013. The GHTorrent dataset and tool suite. In 2013 10th IEEE Working Conference on Mining Software Repositories (MSR) (pp. 233-236). IEEE.
[12] Dyer, R., Nguyen, H.A., Rajan, H. and Nguyen, T.N., 2013. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In 2013 35th International Conference on Software Engineering (ICSE) (pp. 422-431). IEEE.