# Generative AI For Software Engineering.

**Master Thesis**

submitted: 28.September 2024

by: MHD Ayham Shalaby
mshalaby@mail.uni-mannheim.de
born January 9th, 1998
in Damascus, Syria

Student ID Number: 1633598

Chair of Software Engineering
School of Business Informatics and Mathematics
University of Mannheim
D – 68159 Mannheim
Phone: +49 621-181-3912, Fax +49 621-181-3909
Internet: `http://swt.informatik.uni-mannheim.de`

# Abstract

Software debugging and code reuse are essential components of the software development lifecycle, presenting opportunities for code quality enhancements alongside development time reductions. This thesis investigates the integration of LASSO, a comprehensive platform for code searching and analysis, with generative artificial intelligence models to augment relevant code methods as RAG to AI copilot approaches. In particular, this research emphasizes the creation of a context provider, utilizing LASSO's functionalities to support interface-driven, test-driven, and code-driven search methodologies. The main objective is to improve the code generated by LLMs and explore RAG within this context. The context provider merges LASSO's search methodologies with Continue.dev. This synergy encompasses applying an LLM-as-a-Judge to enhance RAG benefits by only including beneficial RAG. This thesis elaborates on the design and implementation of the LASSO-AI copilot integration, emphasizing its technical architecture and the integration with LASSO's API and its scripting language (LSL). The assessment comprises two sets of coding problems and their results, yielding insights regarding the usability and efficacy of the context provider. This research contributes to the domain by showcasing the possible implications of merging LASSO with generative AI for software engineering applications. It also establishes a groundwork for subsequent enhancements and evaluations.

# Contents

# List of Figures

# List of Tables

# List of Abbreviations

| | |
|---|---|
| API | Application Programming Interface |
| AST | Abstract Syntax Tree |
| CDS | Code-Driven Search |
| CSV | Comma Separated Values |
| DAG | Directed Acyclic Graph |
| DSL | Domain-Specific Language |
| IDE | Integrated Development Environment |
| IDS | Interface-Driven Search |
| JSON | JavaScript Object Notation |
| LASSO | Large-scale Software Observatorium |
| LSL | LASSO Scripting Language |
| MQL | Merobase Query Language |
| PSI | Program Structure Interface |
| RDD | Resilient Distributed Dataset |
| REST | Representational State Transfer |
| ROCR | Reuse-Oriented Code Recommendation System |
| RSSE | Recommendation System for Software Engineering |
| SRM | Stimulus/Response Matrix |
| TDD | Test-Driven Development |
| TDS | Test-Driven Search |
| tf-idf | term frequency-inverse document frequency |
| TFD | Test-First Development |
| VCS | Version Control System |

# 1. Introduction

## 1.1. Motivation

The launch of ChatGPT in November 2022 marked the beginning of a transformative era in artificial intelligence, one with unprecedented potential impact. The full implications of this technology are still being understood and realized. Researchers, developers, and users across various domains have been actively exploring ways to integrate AI into their tasks and adapt to the rapidly evolving technological landscape.

In the field of software engineering, this impact has led to the development of various AI-powered tools and approaches. These range from simple chatbot interfaces to more sophisticated systems incorporating AI copilots, repository code reviewers, and intelligent agents focused on addressing specific bugs. More advanced techniques involve indexing codebases, processing targeted queries, reranking results, and providing contextually relevant outputs. According to Gartner, by 2026, more than 80% of enterprises will have used generative AI APIs or deployed generative AI-enabled applications, up from less than 5% in 2023 [29]. It is conservatively estimated to boost global GDP by over 1.5$ trillion [26]. Recent research has highlighted the potential of Large Language Models (LLMs) in various software engineering tasks, including code generation, bug fixing, and code summarization [72].

These models, known as LLM4SE(LLMs for software engineering) or LLMs4Code, are specifically trained on code data sets and made to solve software engineering tasks. However, these models are not without limitations. They work in a 'black box' fashion, making it difficult to understand how and why 'hallucinations' (instances where the model generates code that is not relevant or correct) take place. They also end up recommending code that is not always usable due to a variety of issues, particularly in code adaptability, vulnerability testing, and adherence to specific non-functional requirements such as runtime, test coverage, and usage of specific libraries [56, 59, 62]. Fortunately, empirical research has found ways to mitigate some of these issues, yet, it remains uncertain which techniques yield the highest enhancements in productivity and efficiency.

One particular approach, RA-LLM (Retrieval Augmented Large Language Model), is very promising when dealing with hallucinations [84]. RA-LLM fundamentally means retrieving and augmenting LLMs with correct knowledge that can make them produce better outputs. LASSO [43] (Large-Scale Software Observatorium) is a powerful platform that automates the collection, analysis, and execution of software components at scale. We aim to enhance code recommendation and reuse by integrating LASSO with generative AI models. LASSO's capabilities in interface-driven, test-driven, and code-driven searches provide accurate and relevant code recommendations, grounds AI-generated suggestions in real-world implementations. By leveraging LASSO's unique strengths, this integration could significantly enhance the capabilities of AI-assisted software development tools.

A notable example of an open-source AI copilot integration into an IDE is continue.dev [82] [1]. This platform provides a flexible framework for building custom AI-IDE development tools in a plugin form. Continue.dev allows developers to create context providers that feed relevant information to LLMs, enabling more accurate and context-aware code suggestions. Some key features of continue.dev include: customizable AI models, context providers, extensibility, multi-IDE support.

## 1.2. Problem statement

Software debugging has become an increasingly challenging and time-consuming task in the context of the rapidly evolving software programs. As software systems grow in complexity, identifying and resolving issues becomes more demanding. Traditional debugging techniques frequently need help to meet these mounting demands, leading to delays and increased costs in the software development life cycle. In fact, software debugging costs may account for up to 40-50% of overall expenses [10].

Meanwhile, the development of Large Language Models (LLMs) has made a significant surge, especially in their emergent abilities to handle complexity [100].

[1] `https://github.com/continuedev/continue`

These models have proven themselves in many cases by successfully delivering value across various domains [95, 68]. Especially in software engineering, where many AI developers are also software engineers, there has been many AI enhanced tools that were recently released [1]. LLMs have shown remarkable capabilities in code generation, error detection, and even providing explanations for complex software issues.

Generative AI tools, when integrated into developers' workflows, can enhance productivity and adaptability [16, 67]. However, they also have limitations, particularly in functional aspects. For example, they may produce instances where the AI generates code that does not align with the developer's intentions. They may also struggle to represent complex entity relationships in codebases, due to their reliance on pattern recognition rather than a deep understanding of code structure and semantics [16].

Non-functional challenges also exist, such as adherence to code style (organizations usually have specific methodologies for programming), staying up-to-date with libraries and packages (model training can be costly), and accounting for runtime behavior or project-wide constraints. Major AI tool providers, like GitHub Copilot, offer the best experience in terms of functionality and writing suitable code methods. However, they also raise concerns that do not exist with open-source, locally hosted models, such as developers' hesitance to grant these tools full access to codebases, worry about their code being used to train closed-source models, and exposing API keys or database secrets [16], as already experienced by large companies [80, 14].

## 1.3. Research objectives

This master's thesis introduces a novel multi-stage approach for software debugging by integrating LASSO (Large-Scale Software Observatorium) with large language models (LLMs) in a flexible setup. Although the optimal methodology for this integration is still uncertain, we present this approach as an initial attempt to advance the field and evaluate potential performance gains. By combining two open-source projects, we aim to demonstrate the potential of this integration and invite future improvements and alternative methodologies. It is important to note that this integration currently utilizes only a subset of LASSO's capabilities. Due to the scope and limitations of this thesis, neither LASSO's test generation nor dynamic runtime information about methods were included in the provided Retrieval-Augmented Generation (RAG). However, a complete integration of LASSO's capabilities with generative AI models holds even greater potential to benefit users in four key ways:

- Enhanced decision-making: By leveraging LASSO's large-scale code search and analysis capabilities combined with generative AI models, developers can receive more accurate and contextually relevant code recommendations, while also viewing RAG-used implementations that LASSO actually tested. This synergy improves the quality and reliability of code suggestions
- Improved Debugging Efficiency: The integration allows for more sophisticated debugging processes by combining LASSO's dynamic analysis capabilities with AI-driven insights. This can help developers quickly identify and resolve issues by providing intelligent suggestions based on static and runtime behavior analysis.
- Test Generation and Validation: LASSO's test-driven search capabilities and generative AI can lead to more comprehensive and effective automated test generation. This integration can create tests that cover a more comprehensive range of scenarios, including edge cases that might be missed by traditional methods, thereby improving overall code quality and reliability.
- Contextual Code Understanding and Adaptation: By leveraging the codebase context and tests within LASSO, the AI can formulate more specific LSL (LASSO Scripting Language) scripts. This capability allows the system to retrieve implementations that are better suited to the developer's needs. The integration of LASSO's dynamic and static analysis capabilities with contextual insights from a tool like Continue.dev enables the AI to understand code dependencies and behavior more effectively. This results in more precise and contextually relevant code suggestions, enhancing the overall development process.
- Adaptation Capabilities: LASSO's adaptation mechanisms allow it to adapt retrieved code from specific repositories to fit specific interfaces or coding standards. This feature is particularly useful when integrating code from different sources, ensuring that the adapted code aligns with best coding

practices and guidelines.

## 1.4. Thesis structure overview

The rest of this thesis is structured as follows:

- **Chapter 2: Background Information** provides background on Software engineering tools, AI, LLMs, RAG, and the background information needed to contextually place this integration.
- **Chapter 3: Software Reuse and Use case Scenarios** describes the software reuse process and how our integration fits into the process of pragmatic software reuse.
- **Chapter 4: Integrating Lasso With Generative AI** introduces the proposed approach for integrating LASSO as a RAG service, including the technical details that are needed to realize the integration.
- **Chapter 5: Experiments And Results** presents the experiments and the evaluation of LASSO-AI integration, analyzes its performance in code retrieval, generation and adaptation, then we discusses the implications of the results, limitations of the current approach, and directions for future work.
- **Chapter 6: Conclusion** concludes the thesis and summarizes key findings.

By leveraging the vast corpus of open-source code available through LASSO, and integrating it as a RAG service for LLMs, this thesis aims to push the boundaries of AI-assisted software engineering. The proposed approach has the potential to enhance the code generation with better adherence to non-functional requirements, ultimately leading to more productive and efficient software development practices. The techniques and tools introduced in this thesis represent a step towards overcoming these challenges and unlocking a higher potential of AI in software engineering.

# 2. Background Information

This chapter provides essential background information for this thesis and the different parts of the integration of LASSO (Large-Scale Software Observatorium) with generative AI models for software engineering tasks. We discuss key concepts, tools, and technologies relevant to this integration and establish a consistent terminology for the rest of the thesis. We begin by discussing software engineering challenges, highlighting the need for more efficient debugging approaches. Then, we introduce LASSO, a platform that automates the collection and execution of software components at scale. We explain how LASSO's capabilities can facilitate AI-enhanced software debugging. Afterwards, we explore artificial intelligence applications in software engineering, focusing on large language models (LLMs). We discuss how these technologies have been used to locate bugs, capture code context and dependencies, and assist developers in the debugging process. Furthermore, we discuss crucial concepts for leveraging LLMs in software debugging tasks, including prompt engineering and Retrieval-Augmented Generation (RAG). We also discuss related approaches within the field and how effective the different approaches are. Lastly, we discuss the various copilot tools and why we chose to Continue.dev, a platform for customising LLMs with external data sources to build domain-specific AI applications, for this integration.

## 2.1. Software Engineering Challenges

The 2020 Consortium for Information and Software Quality (CISQ) report emphasises the significant costs associated with poor software quality, including debugging, security attacks, and data leakage [38]. The report estimates that the United States incurs an astounding \$2.08 trillion due to these issues, with operational software failures being the most significant contributor at approximately \$1.56 trillion. This represents a 22% growth over the last two years. One example mentioned in the report is the Equifax data breach in 2017, which exposed the personal information of 146 million consumers. This breach resulted from an unpatched software vulnerability, leading to substantial financial losses, reputation damage, and legal consequences for the company. Effective software debugging and code analysis are crucial during development to mitigate such risks and ensure higher quality and reduced potential losses. These practices help identify and resolve errors, enhancing software reliability, maintainability, and security.

### 2.1.1. Code Analysis And Quality Assurance

Software analysis is a critical process that accompanies software development, focusing on identifying, diagnosing, and resolving errors or bugs within a software system. Two primary code analysis methods are employed for effective software debugging:

- **Static Analysis**: This method examines the code syntax without executing it to detect potential issues such as common programming errors, coding style violations, or security vulnerabilities. Static analysis tools can identify code smells and enforce coding standards, ensuring consistency and readability across the codebase [66].
- **Dynamic Analysis**: This approach analyses the code during execution to identify runtime errors, memory leaks, performance bottlenecks, and other issues that may not be apparent during static analysis, as it can lead to false positives and negatives [31, 8].

Both static and dynamic analysis methods are crucial in the debugging process, as they complement each other in identifying and fixing different issues in code functional and non-functional requirements.

Naturally, these analysis methods laid the foundation for software repair tools; they can be broadly categorised into behaviour repair tools and state repair tools. Behaviour repair tools utilise test suites, contracts, models, and crashing inputs as oracles to guide the repair process. These tools focus on identifying and fixing incorrect code behaviour by analysing the relationships between inputs and outputs or by using formal specifications for reasoning about the code's correctness. Conversely, state repair tools are more concerned with the program's runtime and maintaining its operational state. These tools include techniques such as checkpoint and restart reconfiguration and invariant restoration. Checkpoint and restart mechanisms allow the program to recover from failure by rolling back to a previously saved state. At the same time, reconfiguration techniques adjust the system's settings or components to adapt to changing conditions or recover from errors. Invariant restoration methods ensure the program maintains a consistent state by enforcing constraints on its variables and data structures [65].

Incorporating these tools into the developer's workflow and offering constructive feedback throughout the development process is the most effective way to minimise costs before technical debt builds up [94]. This approach highlights the enduring significance of code quality in software engineering, which has fueled high-quality code reuse and the creation of code search engines and related tools.

### 2.1.2. Code Search Engines

It has been established through observation long ago that the most performed activity in a developer's workflow has been conducting code search [77], with first tools appearing as simple syntax crawlers as "grep" and relied on syntax matching, which often failed to capture the semantic meaning of code. These approaches were limited in their ability to understand code semantics, frequently leading to irrelevant search results. Although Advanced syntactic search methods leveraged more abstract methods and approaches, such as Abstract Syntax Trees (AST) that allowed for more precise search queries [96]. Nevertheless, gathering dynamic information about code is inevitable. Further advancements brought search engines to apply more static and dynamic code analysis techniques to understand deeper semantics and the most recent developments can be broadly categorised between traditional information retrieval techniques and deep learning techniques which created better search models.

Deep learning models learn representations of code and queries, significantly improving the accuracy of search results by understanding their similarities [57, 15]. However, despite all these improvements, the landscape of code search projects has been characterized by a pattern of growth followed by discontinuation [50, 32, 19, 48]. While not always for the same reasons, it can be inferred that it is due to usability issues, technical difficulties and problems maintaining a business model.

This trend can be attributed, in part, to the limitations of early-generation code search engines, which relied heavily on syntactic matching and provided users somewhat related code yet no guarantees whatsoever regarding functional and non-functional requirements [52].

#### 2.1.2.1. Challenges in Code Search

A study conducted at Google analysed thousands of search log records from developers, revealing that developers perform, on average, five search sessions with a total of 12 search queries per workday [75]. This makes code searching one of the most frequent activities in their workflow. Despite the exponential growth of public code repositories [60], developers often struggle to find code examples that precisely match their needs, highlighting significant challenges in obtaining relevant search results. Understanding the functionality and usage of retrieved snippets can be difficult, mainly when documentation is sparse or the code is written in an unfamiliar style. Adapting these snippets to fit specific project contexts requires a deep understanding of the code and the target project, which can be time-consuming and prone to errors. Developers frequently need to refine their queries multiple times, as indicated by instances where search results do not receive clicks.

Query formulation is a significant challenge [71] in code search, often known as the vocabulary mismatch problem [78]. Translating information needs into effective search queries is often difficult, leading to suboptimal results. Developers rely on ad hoc methods to find relevant search results, having to do multiple tries and refinements [75, 71]. Additionally, natural language queries can be ambiguous, making it challenging for search engines to interpret developer intent accurately [71, 78].

#### 2.1.2.2. Limitations of Traditional Code Search Engines

Traditional code search engines face several structural limitations that impact their effectiveness. First, one major issue is the gap between the semantics of queries and the retrieved code. Although deep learning models have been adapted to enhance code search techniques, they often treat queries and code as sequences of tokens, failing to capture the rich semantics and behavior within the code. This results in a focus on syntactic rather than semantic matching or on aspects such as recency or popularity rather than test coverage or runtime behavior, thereby missing functional and behavioral semantics [57, 93, 15]. Consequently, these engines frequently provide contextually irrelevant results by retrieving only basic code snippets instead of fully reusable or tailored components that meet developers' specific needs. As a result, developers spend considerable time customising snippets, limiting their utility in the software development process and making them time-consuming.

Further limitations include challenges in indexing and updating large codebases, which lead to stale search results and affect the reliability of these engines. For accuracy, it is crucial to keep search indices current with recent changes. Infrequent updates compromise the relevance of the returned code, resulting in scalability and performance issues. The need for real-time results is urgent, and efficiently indexing and searching large code repositories demands advanced algorithms and infrastructure. Delivering real-time results requires substantial computing resources, but it is a necessary investment for a reliable code search engine. [7]

Lastly, to make the most efficient use of resources, it is crucial to integrate the code search engine within the tools used in the workflow. This is not just a time-saving measure; it is a game-changer that enhances the efficiency of the software development process by enabling the collection of context and a deeper understanding of the developer's activity, making it more convenient and user-friendly for developers. The following section shows how LASSO addresses many of those limitations and why it is a great candidate for this integration.

## 2.2. LASSO

LASSO (Large-Scale Software Observatorium) is a sophisticated platform designed to automate the collection, execution, and comparison of software components at a large scale and allow observations of big data in repositories. It is the predecessor of Merobase [36], significantly improving the behaviour analysis capabilities of Merobase. LASSO significantly enhances behaviour analysis capabilities by explicitly considering these limitations in its design. To successfully host and operate LASSO, it is essential to deploy two additional instances: Apache Solr and Nexus Repository Manager. These components are integral to

LASSO's functionality, providing the necessary infrastructure for indexing, retrieving, and managing software artefacts efficiently.

- **Apache Solr**: LASSO employs Apache Solr, an open-source enterprise search platform built on Apache Lucene, for indexing and retrieving software components. Solr's robust full-text search capabilities enable LASSO to manage large-scale data from diverse repositories efficiently. Solr helps address limitations through its distributed architecture, which supports scalability and fault tolerance, which are crucial for handling the vast amounts of data processed by LASSO.
- **Nexus Repository Manager**: LASSO uses Nexus to manage and store various software artefacts required by LASSO. Nexus ensures that all necessary artefacts are readily available for LASSO by notifying Solr whenever a change happens to any of the indexed repositories. Nexus helps address limitations by supporting scalability and simplifying dependency resolution and artefact retrieval.

LASSO encompasses a wide range of functionalities and supports a variety of use cases. For our use cases, we focus on the following key features that make it highly effective for software reuse:

- **Behaviour Observation**: LASSO employs an Arena structure within a containerised docker environment to execute code against predefined tests or tests automatically generated using tools such as EvoSuite (for Java) at scale. This methodology allows for safe large-scale code execution and ensures comprehensive behaviour observation, including test results and runtime behaviour. LASSO makes the use of different techniques to allow this to happen, since it is very complex considering different classes need different ways of instantiations, not only does LASSO enable dependencies retrieval (through a Direct acyclic graph representation) but also through the use of adapter code that is a middle layer between a retrieved implementation and given tests that allow retrieved code to be executable.
- **Integration of Multiple Data Sources**: LASSO allows for the integration of multiple data sources, including open public repositories and closed private ones, enabling dynamic selection over a wide range of sources. As long as the additional instances(Solr, Nexus) are correctly hosted and a

repository is indexed in Solr, LASSO simultaneously enables the collection from multiple data sources.

- **API and DSL**: LASSO is particularly well-suited for extension due to its ability to communicate through a simple API interface. This enables communication with different external systems and the extension of more actions through its dedicated domain-specific language (DSL) called LSL (LASSO Script Language). This enables a wide variety of operations on the retrieved implementations. Both concepts allow LASSO to be a versatile tool that can adapt to many use cases.

In the following we go over LSL scripts and how is a code search formulated within LASSO:

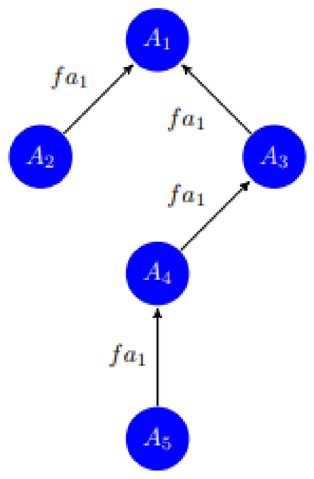
### 2.2.1. LSL Scripts

LSL is an innovative concept that significantly enhances LASSO's capabilities. LSL scripts are declarative and design workflows that generate two DAGs (Dependency DAG and Execution DAG as showsn in Figure 2.1 [43]) with each script, treating implementations as datasets flowing through a pipeline. This setup facilitates data operations (actions) that depend on each other, simultaneously acting as outputs and inputs for other actions. Actions are easily expandable, with users having full control to create and customize their own actions. Fundamentally, an LSL script consists of three main components:

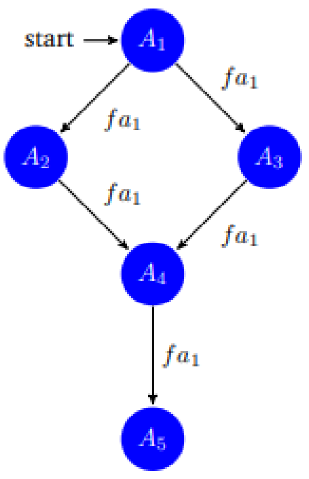
- data source definition: defines from which data source(s) lasso retrieve implementations. It is simply at the start of any LSL script and has the following syntax: dataSource 'data source'
- A study block: which defines the scope for actions to reside in and is the basis of the DAG
- actions: when stacked, creates a chain of operations executed according to the execution plan.

```
study(name:'studyName') {
    action(name:'A1', type:'A1') {
        abstraction('fa1') {
            // ...
        }
    }

    action(name:'A2', type:'A2') {
        dependsOn 'A1'
        includeAbstractions 'fa1'
    }

    action(name:'A3', type:'A3') {
        dependsOn 'A1'
        includeAbstractions 'fa1'
    }

    action(name:'A4', type:'A4') {
        dependsOn 'A3'
        includeAbstractions 'fa1'
    }

    action(name:'A5', type:'A5') {
        dependsOn 'A4'
        includeAbstractions 'fa1'
    }
}
```

(a) General Structure of an LSL Script



(b) Dependency DAG of (a)



(c) Execution DAG of (a)

Figure 2.1.: LASSO LSL Scripts

LASSO's extensibility, mainly through its LSL actions, supports various tools that can influence implementations and refine them effectively. LASSO's diverse code analysis capabilities enhance its utility for software recommendations by refining and filtering result sets. While LASSO offers extensive functionalities, this section focuses on those used by our plugin integration.

- **Code Clone Detection with NiCad** LASSO integrates the NiCad clone detection tool [20] to identify code clones within its software repository. NiCad is a text-based clone detection tool that uses a hybrid approach combining parsing and text comparison. It supports various programming languages and can detect both exact and near-miss clones. NiCad's clone detection cleans and filters code snippets to bring them to a standardised format, allowing it to identify clones at different levels. It can detect specific types of clones and reject the ones that are too similar to increase the diversity of the retrieved implementations.
    - Type-1 clones: Identical code except for variations in whitespace and comments. (Textual similarity)
    - Type-2 clones: Syntactically identical code with differences in identifiers, literals, types, etc. (Lexical similarity)
    - Type-3 clones: Copied code with further modifications such as changed, added or removed statements.(Syntactic similarity)
    - Type-4 clones: Code that is syntactically different but functionally equivalent.(Functional similarity)
- **Component Ranking with SOCORA** LASSO uses the SOCORA (Software Component Ranking) framework [47]. A testing framework was developed to facilitate selecting and recommending software components based on their respective behaviours. SOCORA is a ranking engine that prioritises software components based on various criteria and metrics. SOCORA allows the design of custom ranking strategies that are not only based on syntax but also include semantics and combine multiple factors, such as code quality metrics, test coverage, popularity, and domain-specific requirements. It provides a flexible and extensible framework for specifying ranking criteria and weights depending on the desired use case.

- **Coverage Analysis with JaCoCo** LASSO incorporates the JaCoCo code coverage library [35] to assess the quality and coverage of the test cases. JaCoCo is a widely used tool for measuring and reporting code coverage in Java projects. JaCoCo provides detailed coverage information at various levels, including methods, classes, and projects. It allows for a large-scale coverage comparison of code snippets.

### 2.2.2. REST API

LASSO provides a comprehensive REST API that can be easily integrated into larger applications and toolchains. This enables the creation of custom code search and analysis solutions that leverage LASSO's capabilities in a broader context. We use the following API points for our integration, yet we recommend expanding more on this in the discussion chapter.

| **Function** | **Type** | **Path** |
|---|---|---|
| Obtains an OAuth token | POST | `/auth/signin` |
| Executes an LSL script | POST | `/api/v1/lasso/execute` |
| Retrieves execution status | POST | `/api/v1/lasso/scripts/${executionid}/status` |
| Gets execution result | POST | `/api/v1/lasso/scripts/${executionid}` |
| Retrieves implementations | POST | `/api/v1/lasso/${datasource}/implimentations` |
| Performs an SQL query | POST | `/api/v1/lasso/report/${reportid}` |

Table 2.1.: LASSO API Endpoints

By combining the different actions within the LSL scripts, LASSO inherently supports three search methods to find code snippets:

### 2.2.3. Interface-Driven Search (IDS)

Interface-Driven Search (IDS) leverages keywords and structured queries to locate methods and classes based on their interface descriptions. This approach is rooted in signature matching techniques, where predefined method signatures (Inputs and outputs) are used to identify relevant implementations. For example, in the Merobase code search engine, this is how a signature looks like:

```
Stack (push(int):void;pop( ):int; )
```

IDS matches these signatures to retrieve software modules that conform to specified interfaces. However, IDS faces inherent limitations, primarily due to vocabulary mismatches. Developers often use varied naming conventions and identifiers, leading to discrepancies in search results. This variability can significantly impact the effectiveness of syntactic searches, as similar functionalities might be named differently across projects. Although methods like automated query expansions and natural language processing (NLP) techniques exist to enhance query precision, there are still issues in overcoming different developers' unique and/or idiosyncratic naming strategies.

To address these challenges, LASSO extends the Merobase Query Language (MQL) capabilities with its own LASSO Query Language (LQL). This extension improves recall and precision in interface-driven searches by allowing more nuanced and flexible query formulations based on interface signatures. LASSO enhances implementations retrieval by adding synonyms and antonyms, making it more robust against vocabulary mismatches. For instance, a predefined interface signature in LQL might be specified as follows:

```
Bag {
add(java.lang.Object) -> void
count(java.lang.Object) -> int
remove(java.lang.Object) -> void
contains(java.lang.Object) -> boolean
size() -> int
}
```

This signature defines the expected inputs and their respective output types for a class implementing the Bag interface.

### 2.2.4. Test-Driven Search (TDS)

Test-Driven Search (TDS) aims to identify code snippets that satisfy specific test criteria, encompassing functional and non-functional requirements. These tests can be user-defined or automatically generated, ensuring the retrieved code matches syntactic patterns and meets desired behavioural specifications. TDS bridges the gap between mere code retrieval and functional validation, providing a more reliable mechanism for discovering code that meets specific operational expectations.

In LASSO, TDS enables developers to define expected behaviours through unit tests. Unit tests serve a dual purpose in LASSO. First, Lasso follows an IDS approach to formulating inputs and outputs. Second, it executes these tests against the retrieved implementations in a containerised arena setup, identifying components that pass the specified criteria. This process ensures that the retrieved code snippets exhibit the desired functionality, which adheres to predefined inputs and outputs. In LASSO, tests are lined in a matrix-style actuation sheet where columns represent outputs, method invocations, variables, and inputs.

Example of a test sheet in an LSL script:

```
    sequences = [
        //parameterised sheet (SSN) with default input parameter values
        // expected values are given in first row (oracle)
        'pushPop': sheet(p1: 'Bag', p2: "hello", p3: "world") {
            row '', 'create', '?p1'
            row '', 'add', 'A1', '?p2'
            row '', 'add', 'A1', '?p3'
            row '', 'add', 'A1', '?p2'
            row 2, 'size', 'A1'
            row 2, 'count', 'A1', '?p2'
            row 1, 'count', 'A1', '?p3'
            row false, 'contains', 'A1', "engineering"
            row '', 'remove', 'A1', '?p2'
            row 1, 'count', 'A1', '?p2'
        }
]
```

### 2.2.5. Code-Driven Search (CDS)

Code-driven search (CDS) consists fundamentally of using a code snippet to find similar or improved implementations that adhere to more functional or non-functional requirements or have different inputs/outputs. In LASSO, this is realised by automatically generating tests (using Evosuite) and then converting them to a TDS problem, in which, as described above, a signature is generated. In our integration, this is also possible by leveraging the LLM to write those tests. However, this is influenced by many factors, as we will discuss in a later chapter. We did not adapt the Evosuite test generation in our integration

Within our integration, the complexities of these different search approaches are hidden from the user, ensuring a seamless experience. These approaches are made in the background, adapting to the user's query without disrupting their experience. Using natural language to formulate and communicate with multiple platforms is a powerful concept facilitated by LLMs. In the following, we examine the evolution and involvement of AI in software engineering.

## 2.3. Artificial Intelligence

Artificial intelligence is a computer system that aims to emulate human cognitive functions and perform tasks that typically involve human capabilities such as learning, reasoning, and problem-solving. These tasks can range from simple tasks, such as playing chess, to very complex tasks, such as high mathematical reasoning or driving a car. The evolution of AI systems was a long, multifaceted journey that encompassed many applications in many disciplines. Initially focused on automating repetitiveness and performing specific tasks with no genuine consciousness or understanding, consisting purely of rule-based systems. The availability of more and cheaper computational power and datasets led to a shift in research to machine learning and deep learning technologies, allowing for a complex modelling of non-linear relationships between data. Initial applications started appearing that were developed for complex specific tasks, such as AI-doctor interpreting medical images and enhancing diagnosis accuracy[17], the convergence of integrating AI into other fields was extremely beneficial for the development of this field.

Recently, language models have gained prominence. While at first, they were not particularly helpful, the improvements in many factors have given it emergent, unpredictable abilities, the biggest of which is scaling up a regular language model [18] (by scaling up, we mean increasing the amount of computation, number of model parameters and training dataset size [34]). With time, this leads to a range of benefits, such as improved performance and sample efficiency, yet it still needs to be fully understood why this is the case. Thus, these benefits and abilities acquired are considered emergent, unpredictable abilities [100].

This thesis focuses on integrating LLMs with LASSO to leverage these capabilities to improve software reuse and development processes. We examine in the next section what AI approaches already exist and how they perform against newer LLM AI approaches.

### 2.3.1. AI Approaches and Technologies in Software Engineering

This section delves into the various AI-driven software engineering approaches, examining their potential impact on software development and their underlying principles, strengths, and limitations through case studies. We compare these AI technologies for software engineering and debugging, emphasising their differences and how they compare against newer generative AI-driven approaches.

- AI-Static and Dynamic Analysis Approaches: Static analysis approaches, such as those implemented by Allamanis et al. [5], focus on analysing the syntax and structure of the code without executing it. These methods detect common programming errors, code style violations, and potential security vulnerabilities. However, they are limited in uncovering runtime issues or identifying more profound logical errors that might only surface during code execution, as any other static analysis tool suffers from. Dynamic analysis approaches, like the deep learning-based bug localisation method proposed by Zhang et al.[96], examine the code during execution. By doing so, they can uncover runtime errors, memory leaks, and performance bottlenecks. While these approaches can provide valuable insights into software behaviour, they may also require more computational resources and time to execute, potentially leading to longer debugging cycles.
- AI-Machine Learning and Transformer-Based Models Approaches: Machine learning-based techniques, such as those employed by DeepFix [33], and S2FIX [51], use algorithms that learn from past issues to suggest fixes for similar problems. Analysing the code and incorporating past training data can provide developers valuable insights and save time during software development. On the other hand, generative transformer-based models, like BERT [24], have shown promise in understanding natural language and can

be adapted to assist in debugging tasks. However, their performance depends on the quality of their training data, and they might struggle with complex or unfamiliar code structures. The transformer's technology was relevant in the field from even before as one very famous plugin developed by Microsoft, IntelliCode [81], was explicitly developed using this generative technology(before chatgpt3 release) and was very successful in yielding good predictive results and helping developers with what they wanted to write next.

We notice that although these approaches are trained differently to detect issues and, analyse code and push the benchmarks upwards, they all perform as well as their training data, learning approaches, and predictive accuracy, making this an inherent challenge for all AI-driven tools in software engineering. [91] As will be described in the next section, this is also a challenge that LLMs face, which shows the need for approaches and techniques that can help with those accuracy problems.

### 2.3.2. Large Language Models (LLMs)

Large Language Models (LLMs) have become transformative tools in various domains, significantly changing decade-long ways of how things are done. LLM models, such as OpenAI's GPT series with the transformers architecture [86] or DeepSeek-CoderV2 with its Mix of Experts (MoE) architecture [23], are trained on extensive text and code datasets. This comprehensive training enables LLMs to perform complex tasks involving natural language understanding, code generation, and reasoning. For Software development, we summarize the main capabilities in the next section.

#### 2.3.2.1. Capabilities of LLMs in Software Development

LLMs offer several key capabilities that enhance software development processes:

- **Interpret Natural Language Queries:** LLMs can process and understand natural language descriptions of software requirements, bug reports or for formulating queries in a query language such as(Text-To-Sql [97]). This capability allows developers to communicate freely using natural language and not waste time formulating specific query languages.

- **Generate Code Snippets:** Based on a given developer's request and/or for a given context, LLMs can generate and adapt code snippets to specific contexts, decreasing time spent in repetitive coding tasks and accelerating the development process. LLMs are also proficient in translating code from one programming language to another.[92]
- **Provide Explanations and Suggestions:** LLMs can offer detailed explanations for code behaviour, suggest potential bug fixes, and provide guidance on best practices and design patterns. LLMs are already utilized in education, functioning as tutors and evaluators of students' codes. [58]

#### 2.3.2.2. Evolution and Impact of LLMs

Initially, LLMs faced challenges due to high expectations set by the hype around them and technical limitations. At the time of writing this thesis and according to Gartner [28], we are now at the peak of the hype and the inflated expectations when talking about AI-augmented Software Engineering, new AI products are coming out every day, and approaches are being evaluated for usefulness and potential use cases and usage.

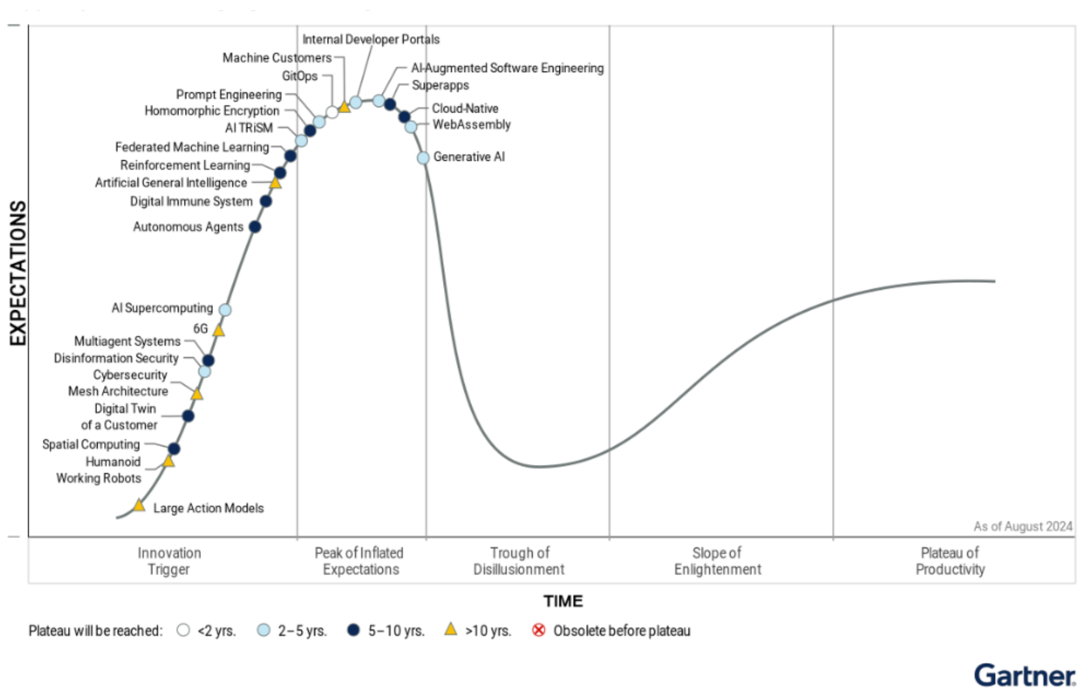


Figure 2.2.: AI-Augmented Software Engineering Hype Cycle

It is looking better in generative AI as many applications prove their usefulness more and more with time. However, in software engineering, where reasoning and logical thinking are essential when confronting any task, it was a rough start as seen by this documentation by A.Borji [11].

Nonetheless, advancements in computational power, model parameters, and training dataset sizes have significantly enhanced their capabilities and raised the bar of these failures. Nevertheless, they did not disappear and might not disappear at all [91]. It could be a benchmark that gets higher with complexity, the more these LLM models keep improving, which will keep getting higher with newer benchmarks to evaluate it. The rapidly evolving landscape and the pace, in which newer LLM models coming out in a relatively short period makes it challenging to fully understand implications of the different architectures techniques and which model with which techniques yield the highest performance in which specific tasks.

Recent developments and trends include the integration of agents powered by LLM reasoning skills, which can execute and analyze code dynamically [39]. While these agents represent a promising direction for dynamic code analysis, this thesis argues that enhancing LLM capabilities through accurate functioning code with behaviour details in the form of Retrieval-Augmented Generation (RAG) might be equally effective. (or maybe both, as in retrieval-augmented generative agents (RAGAs)

#### 2.3.2.3. Challenges and Limitations

Among those categories mentioned by A.Borji [11] in general and challenges that LLM4SE face in particular are the following limitations:

- Contextual understanding: While LLMs can generate and understand code, they often struggle to fully grasp the context of specific codebases or project requirements, leading to inconsistent or irrelevant debugging suggestions.
- Reliability and accuracy: The "hallucination" problem in LLMs can result in the generation of plausible but incorrect code fixes, which is particularly problematic in software engineering where precision is crucial[56]. Arguably, the hallucination problem might not go away [91] and only with RAG and/or other guardrails it is to be prevented.
- Lack of up-to-date knowledge: LLMs are trained on static datasets, which may not include the latest programming languages, frameworks, or best practices. It is even more of a problem when the LLM faces code that it was not trained on and did not see before. [42]
- Adaptability to specific coding standards: Different organizations and projects often have unique coding standards and best practices that are not inherently known to LLMs [12].
- Limited access to runtime information: Traditional debugging often relies on runtime information, which is not readily available to LLMs during the inference process.

By integrating LLMs into the development workflow through a well-functioning RAG system, we aim to enhance behavior-aware suggestions and improve overall software reuse and development processes, while being conscious about limited setups, costs and hostability.

LLM reasoning skills are not the only factor in output quality; other factors play equally important roles. Prompt engineering and RAG push benchmarks higher for better and more suited outputs. In the following, we examine those techniques closely, inferring which parts are essential to our integration.

## 2.4. Prompt Engineering

Prompt engineering is a critical technique in the large language model corner of artificial intelligence (AI), particularly in optimizing interactions with large language models (LLMs). It involves crafting precise inputs or instructions to guide AI models in generating accurate and contextually relevant outputs. Prompt engineering is increasingly taking a bigger role in the AI industry across various application areas and domains. Robin Li, the CEO of the Chinese AI giant Baidu, has predicted that "In ten years, half of the world's jobs will be in prompt engineering, and those who cannot write prompts will be obsolete." [79] While this statement may seem hyperbolic for the present time. There are many companies that compete in offering prompting services or techniques that include formulating queries many times, getting answers, and then evaluating the best prompts. Not only do automated systems offer these services, but it has also became a new job position for humans. This underscores the growing importance of effective, prompt design in maximizing the utility of AI models.

Prompts' nature can vary based on the specific needs of the user. They could be instructional informational-based prompts, such as the ones used in our use case where we explain the script LLM what LASSO is and how it should formulate its queries. Or they could be reformulation prompts, such as how in our thesis we request the output of the AI to be of a specific format. Another type is metaphorical prompts, which can elicit creative or more nuanced responses. For instance, a prompt could be designed to inspire a poem from an AI language model. Prompting can also be used to plan answers depending on a user's request; for example, a prompt can intercept a user's intention and invoke a particular process over another process, as in a user requesting a specific implementation. Simple prompting techniques, as the one used in this thesis are:

- **Zero-shot prompting** Providing instructions or examples in the prompt itself, without any prior fine-tuning.
- **Few-shot prompting** Including a small number of examples in the prompt to guide the model.

- **Chain-of-thought prompting** Encouraging step-by-step reasoning by asking the model to explain its thought process.
- **Explainable retrieval prompting** Encouraging step-by-step reasoning by asking the model to explain its thought process.

However, more sophisticated techniques include far more than having one system instruction that gets attached whenever a user hands in input. They include a multi-step process where the LLM gets into a recursive loop to reason about a problem, create a chain of thought, and try to find solutions, all happening dynamically while the model is generated in real time. Some of these prompt engineering techniques include:

- **Query Transformations** Encouraging step-by-step reasoning by asking the model to explain its thought process.
- **Hypothetical Document Embeddings** The core concept of HyDE is that formulating different hypothetical prompts might yield better results than the original prompt.[21]
- **Directional stimulus prompting** Providing clear direction in the prompt to reduce ambiguity and focus on user intent.
- **Superposition Prompting** The proposition of this concept is that LLMs inherently suffer from a "distraction phenomenon" from having non relevant context in the prompt and/or the RAG, where inputs get processed in a parallel fashion and get discarded when its deemed irrelevant. [61]

By their own nature, large language models produce randomized outputs, with sometimes outputs better than other times, although the user is giving the exact same input, it takes logical creativity and a deep understanding of the task that the AI is supposed to perform to build a multi-step prompt process or approach that gets the AI to fully reason about its output and try to control the randomness and the hallucinations that come by. Prompt engineering is not only the instruction on how to solve a problem or answer a query but also extends to the model evaluating its outputs and having guidelines on what it can output and what it cannot.

### 2.4.1. Prompt Engineering Applications

In the following we examine areas where using prompt engineering has been used to provide benefits and try to draw inspirations from for our work:

- Healthcare: Prompt engineering has shown significant potential in healthcare applications, particularly when dealing with sensitive and precise data such as health records. A notable study [88] demonstrates the importance of a multi-step approach when using LLMs and how to reduce complexity and costs while handling very sensetive data. The researchers implemented a 4-step process that involves:

    1. Splitting the task into multiple LLM calls
    2. Utilizing different sizes of LLMs for various subtasks
    3. Combining the results to accomplish a more complex task

- Finance: In finance, while handling very complex financial data and evaluating the financial sentiment of companies, prompt engineering has made significant improvements. Without even needing to do complex prompt engineering and simply utilizing zero-shot and few-shot prompts strategically, depending on the complexity of the data and the task, researchers have improved sentiment outcomes and enhanced the quality of LLMs answers [3].

- Software Testing: In Software testing, despite having great improvements on a variety of testing tasks, there are only few studies that try the different kinds of prompt engineering approaches, and it will definitely take time until the full potential of each technique and to which domain areas it is best applied is fully discovered. In a survey where papers are analyzed based on their approaches to prompt engineering, most papers really had zero-shot learning, few-shot learning, chain of thought, and the rest rarely, if ever, tested. as we can see in Figure 2.3 [87]

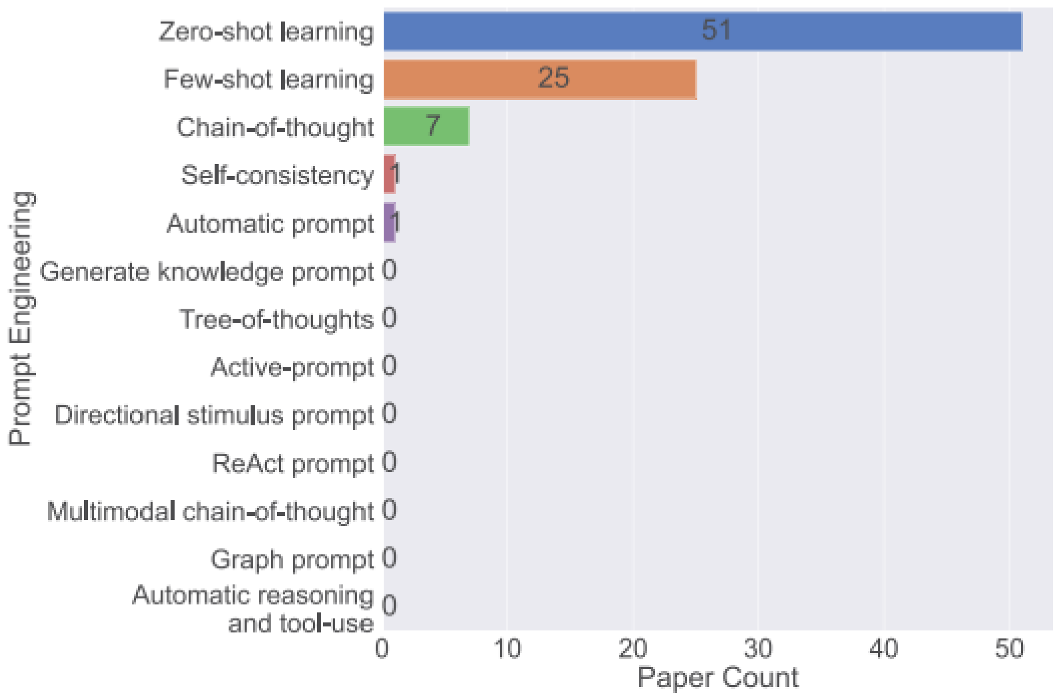


Figure 2.3.: Prompt Engineering Approaches In Recent Research Papers for Software Testing

- Software Development (Language translation): In software engineering, researchers have effectively utilized strategic prompt engineering to translate code without relying on large, complex language models [55]. They achieved this by dynamically adapting prompts to specific tasks, focusing on efficient, prompt adaptation instead. Our work draws inspiration from their approach of using few-shot learning prompts. However, rather than adopting a similar methodology, we employ a general pre-trained language model to fill in the necessary gaps of an LSL script by providing a few examples in the prompt. Our approach could also be enhanced on this approach to reduce costs in terms of GPU memory usage.

Prompt engineering face many challenges, especially in prompt effectiveness, such as knowing which prompt is better with many factors such as length, complexity, specificity, context, and even which AI model is used. In an advanced AI system, prompt engineering is a vital aspect of the AI application, and evaluating prompts using metrics like accuracy and relevance while mitigating ethical considerations such as bias mitigation, transparency, and fairness are a must.

## 2.5. RAG

Just as prompt engineering has emerged as a powerful concept to enhance the outputs of LLM, this is also true for the concept of Retrieval-Augmented Generation (RAG) in artificial intelligence, which has emerged as another piece of the puzzle to enhance the capabilities of large language models (LLMs) by integrating external knowledge retrieval processes. RAG is mainly used to address the limitations of LLMs in handling knowledge tasks, where accuracy and precision are critical [54]. RAG has evolved into a significant technique for addressing Knowledge Gaps and improving output quality. This importance is marked by the rise of its application across various domains, improvements in efficiency, and the development of hybrid approaches that combine RAG with other techniques.

As we can see in the following figure 2.4 from [27], in order to efficiently employ RAG, three core points are to be addressed:

1. Input: how to best address the user's query to formulate a precise and correct query for the retriever so we can retrieve correct information C
2. Retriever: how does the retriever cost efficiently without comprimising on correctness retrieve the knowledge gaps
3. Evaluation: how do we know if our RAG system yield successful results and improve upon the baseline.

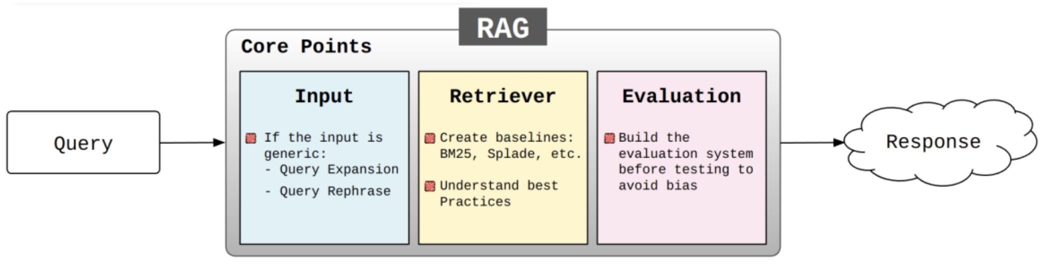

RAG
Core Points
Query
Input
If the input is generic:
- Query Expansion
- Query Rephrase
Retriever
Create baselines: BM25, Splade, etc.
Understand best Practices
Evaluation
Build the evaluation system before testing to avoid bias
Response

Figure 2.4.: RAG Components

Arguably, the last core point is the most difficult as it varies significantly by the requested knowledge, even more so since RAG is a double-edged sword [12, 98, 22]; it can greatly improve output quality or greatly reduce it. Retrieving wrong or improper knowledge is directly correlated to worse quality in the input. Even in our approach, as we will see later, utilizing filtering mechanisms and chunking strategies, dependent on the task, is very important to address those small nuances yet with a big impact. Even the amount of noise in the rag holds great power in determining the quality of the output, such as adding documents that do not contain the answer can negatively damage the effectiveness.

### 2.5.1. Origins and Development of RAG

The term RAG was first coined by Lewis et al. [53] in 2020, as a method to improve the performance of LLMs by incorporating external knowledge retrieval. Originally aimed at addressing the issue of hallucinations in LLM-generated responses, where models produce inaccurate or nonsensical outputs due to a lack of real-world knowledge. RAG, viewed as a process, has two main phases: a retrieval phase, where relevant information is fetched from external sources and given to the LLM and a generation phase, where the LLM uses this information to produce more accurate and contextually appropriate responses. This dual-phase approach allows RAG systems to extend beyond the pre-trained knowledge of LLMs, making them particularly useful in dynamic environments where information is constantly updated and efficiently reducing costs by not having to train models again.

### 2.5.2. RAG Applications

RAG has been applied in various domains, including enterprise settings and scientific research, where it helps manage and summarize large volumes of data. While most RAG applications focus on retrieving specific QA knowledge across vast amount of knowledge [63, 13], we draw inspiration from the following research papers:

- In a recently released research paper that examines the potential for leveraging open source LLM models with RAG to achieve effective handling of data at an enterprise level, thus offering an open source alternative that is cost efficient and addresses the many concerns that come with the usage of closed source models, researches found that integrating the correct techniques in retrieving relevant data from online enterprise-specific data, cleaning them thoroughly and chunking in a way that best matches incoming queries has led to smaller size models such as Llama3-8B to show superior performance when compared to GPT-3.5, this integration shows the potential of leveraging RAG with smaller sized models to solve big problems even at an enterprise size. [6]

- In an application and a research paper by Burgan et al. in Shepherd University [13], developed for its students, a Retrieval Augmented Generation (RAG) Chatbot App called RamChat using Large Language Models was described. RamChat uses API-based and local LLMs for natural language processing and a vector store system. The chatbot was optimized by leveraging RAG to produce better quality outputs than by a paid closed-source model, OpenAI's davinci-002 model, with Gemma, a local LLM based on Google's Gemini model. Furthermore, the researchers utilized the Ollama framework in Python to assist in automatic LLM selection based on user prompts. This application of an LLM chatbot, not only shows the benefits of using RAG to deliver a personalized tutor experience based on factual data, but it also shows the potential of utilizing frameworks such as Ollama to utilize different LLM for different goals within the same AI application, as we explain in a later chapter, we explored utilizing different models for multiple calls within our approach.

With the use of a multi-step setup that employs different LLM calls for different purposes in the RAG retrieval and judgement of RAG with the use of the Ollama framework, we explore and allow the use of multiple LLM models that might potentially yield better performance and be more cost-efficient.

### 2.5.3. Embedding and Chunking

Two main strategies are employed to ensure efficient content retrieval in the retrieval phase: chunking, which divides data into partitions, and embedding, which is vectorizes these partitions to allow for similarity calculations and efficient ways of retrieval. Chunking in methods, for example, could be on a class or a method level; the smaller the chunks, the more computations are required to determine which chunks are relevant and which are not. Even the slightest difference in the chunk size, strategy, or embedding has significant differences in the implementation and the score, so it is best to try as many approaches as possible and settle for what works best. This is argued by the authors of: "The Chronicles of RAG: The Retriever, the Chunk and the Generator" [27]. Sadly, given this thesis's timing and scope limitations, we could not utilize advanced chunking strategies, so we kept the chunks at a class level and with no embeddings. However, we employed a filtering strategy to reject chunks deemed improper.

We refrain from expanding on the topic of embeddings and chunking. Although it is beneficial, the use should be justified and not just added complexity, for instance in QA knowledge systems chatbots, a vast amount of knowledge is chunked, vectorized and then, through the use of cosine similarity and other methods, efficiently retrieved (although the similarity search is also another corner where things might go wrong). However, our system does not need embeddings as LASSO already retrieves context at a class level, so further embedding must make more sense for class components to be chunked and embedded to retrieve individual methods. Improper embeddings lead to worse results, harm software generation and might add unjustified complexity [90]. We also like the ability to view the RAG-used methods directly within the continue.dev framework, giving the user more knowledge about what is retrieved and generated.

### 2.5.4. Challenges and Limitations

RAG is best described as a double-edged sword [98], while it does deliver benefits in various applications, it presents new sets of challenges, especially how its components of chunking and embeddings affect the outcomes. In software engineering, and since it is a new field for many who recently started developing generative AI applications, making an integration defect when applying RAG is very probable as observed in current open source software (up to 98%) [76]. Certainly, findings such as adding random documents to the prompt can surprisingly improve LLM accuracy by up to 35% [22], which complicates things further. We are no exception; we welcome further developments to our preliminary approach and mention the challenges and pitfalls that are the most relevant to our work.

According to [2]. The biggest challenge when applying RAG to software engineering is inaccurate code retrieval, up to 83% of retrieved contents for question answering over software repositories involved retrieving bad data.

Other areas of great concern when implementing a RAG system are technical and vulnerability defects [2]. As already many open source rag systems suffer from them, it could introduce security risks and vulnerabilities into codebases. To effectively mitigate these issues, it is crucial to implement a robust RAG filtering mechanism. This mechanism should be capable of identifying and filtering out hidden issues in the code. Systems must be evaluated repeatedly to ensure these issues are not causing enormous technical debts. Otherwise, even just the suggestion of non-optimal code methods could start a circle of technical debts that is inescapable [73]. The primary response is nearly always manual inspection and refactoring of code.

## 2.6. Integrated Developer Environment Plugins

Currently, LLMs are being adopted in software engineering and provide benefits. However, their efficiency is restricted by their integration into the workflow. Daniel J Russo [74] and Wong et al. [89] describe the benefits of integrating AI tools into the workflow [74]. Some notable benefits include code completion, bug detection and repair, and optimization. Code completion uses models trained on large codebases to provide context-aware completion suggestions while simultaneously emulating developers' coding styles from the context of the integrated tools, which leads to reduced typing effort and improved code consistency. In Russo's findings [74], the author found that around half the participants found improved efficiency (55%), reduced time fixing monotonous task-specific benefits (26%), and using it as a complementary tool (18%) during development. Russo stresses the importance of AI-Workflow integration [74], as it allows developers to access these insights within the existing technical environment and codebases. This is vital to realize gains in efficiency and quality. This can be further exacerbated by providing real-time feedback on potentially non-optimized code, security vulnerabilities, and performance issues. The additional required knowledge to adhere to those requirements could be integrated in the form of RAG provided by LASSO to provide better suggestions and feedback for developers. More advanced integration that also provides improvements or additional knowledge, is how GitHub uses CodeQL [30] on codebases. Advanced integrated AI enhancement techniques can also provide recommendations for optimizing the code. This includes feedback on best practices and guidelines, as well as refactoring and simplifications. Code Optimization tools provide a time to improve the quality of the code regarding efficiency, readability, and maintainability.

### 2.6.1. Chatbot-Style Interfaces for AI Integration

Chatbot-style interfaces have emerged as a convenient AI-developer communication approach. Chatbots provide a natural and intuitive way for developers to interact with AI-powered tools, allowing them to ask questions, seek guidance, and receive intelligent responses within the IDE. Early examples of such tools, that were discontinued are Codota [1], which has been integrated into another copilot, and Kite [2]. Kite used a combination of natural language processing and machine learning to provide intelligent code completions, documentation, and examples. It also offered a search interface called "Kite Copilot," which allowed developers to retrieve code snippets related to their code while taking their context into place and following their cursor. These examples show that integrating AI-powered chatbots within IDEs has several advantages. Modern AI copilot IDEs offer the following advantages:

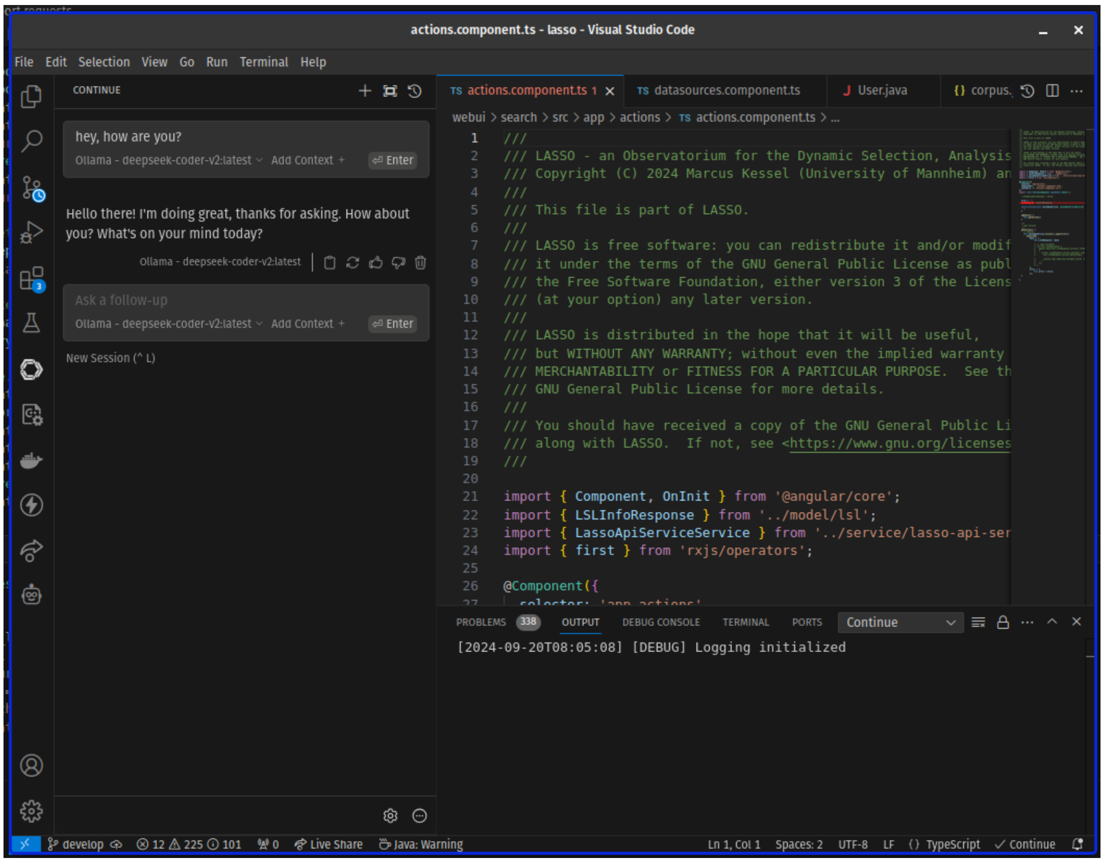
- **Natural chatbot Interaction**: Conversational interface provides a sense of intuition and familiarity, reducing the developers' learning curve associated with adopting new tools.
- **Codebase Indexing**: Codebase indexation allows for multiple benefits, one of which is to give numerical representations of the codebase to the AI rather than plain code. This allows for better retrieval of relevant code snippets to provide examples and explanations catered to their specific needs.
- **Integrating 3rd Party Platforms and Slash commands**: AI powered plugins, such as continue.dev [82], integrates multiple tools to allow for additional information through the use of context providers and the automation of recurring tasks with slash commands.

[1]Codota AI Autocomplete `https://plugins.jetbrains.com/plugin/7638-codota-ai-autocomplete-for-java-and-javascript`

[2]Kite Team. (2021) `https://www.kite.com/blog/product/kite-is-saying-farewell/`

- **Customization**: According to the interactions with the developer, AI-powered chatbots can learn and adapt their responses to yield customized recommendations based on individual preferences and coding styles.

As AI technologies advance, the potential for AI-powered IDE plugins and chatbots to improve software development expands. IDEs are becoming intelligent assistants that enhance productivity, code quality, and the overall development experience. As more research is conducted in this area and more innovative plugins are developed, a significant shift towards IDE AI-driven software development practices can be expected in the near future.



hey, how are you?

Hello there! I'm doing great, thanks for asking. How about you? What's on your mind today?

```
///
/// LASSO - an Observatorium for the Dynamic Selection, Analysis
/// Copyright (C) 2024 Marcus Kessel (University of Mannheim) an
///
/// This file is part of LASSO.
///
/// LASSO is free software: you can redistribute it and/or modif
/// it under the terms of the GNU General Public License as publ
/// the Free Software Foundation, either version 3 of the Licens
/// (at your option) any later version.
///
/// LASSO is distributed in the hope that it will be useful,
/// but WITHOUT ANY WARRANTY; without even the implied warranty
/// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See th
/// GNU General Public License for more details.
///
/// You should have received a copy of the GNU General Public Li
/// along with LASSO.  If not, see <https://www.gnu.org/licenses
///

import { Component, OnInit } from '@angular/core';
import { LSLInfoResponse } from '../model/lsl';
import { LassoApiServiceService } from '../service/lasso-api-ser
import { first } from 'rxjs/operators';

@Component({
```

```
[2024-09-20T08:05:08] [DEBUG] Logging initialized
```

Figure 2.5.: IDE Copilot plugin (continue.dev)

# 3. Software Reuse and Use Case Scenarios

This study aims to facilitate pragmatic software reuse through the integration of Lasso with generative AI. In this chapter, we discuss software reuse in general, its benefits and challenges, how our use case scenarios fit into the bigger picture and how well we support all three steps of pragmatic software reuse, specifically Selection, adaption and integration. [46]

## 3.1. Software Reuse

Software reuse is the practice of using existing software components to create or improve new software systems [40]. This can range from using small methods and importing libraries to incorporating entire modules and code infrastructure. Traditional code reuse often functions in a black-box manner, where developers import a package and call its methods for their development purposes. Although this method effectively demonstrates software reuse due to its simplicity and quick implementation, code components must be designed with reusability in mind. Frequently, code intended for reuse lacks the necessary adaptation for various applications, leading developers to experience frustration and a ’not invented here’ syndrome [37] and due to increased efforts in readaptation, reorganization and retesting that most developers at that point would give up and just rewrite the functionality from scratch, especially in large open source repositories, there are at least hundreds or hundreds of thousands of ways to split a string or encode Base64, this shows how, for many developers, it is often easier to rewrite things from scratch than to search, find code and adapt to their purposes.

### 3.1.1. Benefits

Ideally, software reuse would be implemented efficiently with minimal code modifications, minimal adaptation efforts leading to reducing development time and workload, lead to increased code quality (Difficult to quantify, usually non-functional requirements [9]), reduce development effort, and enhance maintainability [64]. Arguably, generative AI is an advanced form of software reuse [41]. These LLM models have been trained on massive code datasets to give more customized code snippets for different developers' needs. However, LLMs are advanced NLP analyzers and function as static code analyzers or in an advanced form, with time and effort, as dynamic ones using agents. This leads us to the next point: faced challenges.

### 3.1.2. Challenges

Effective reuse requires robust tools and methodologies to locate, evaluate, and integrate reusable components seamlessly [40]. Furthermore, without a realistic evaluation of how much time and effort is needed to reuse code, whether it is a few lines or a structural change, there exists a break-even point where rebuilding components from scratch is easier than reusing. this highlights the need for an effective software reuse approach that lowers time and effort costs, namely, efficient pragmatic software reuse.

### 3.1.3. Pragmatic Software Reuse

Pragmatic software reuse [46] refers to the practical approach of adapting existing components to meet specific project requirements. Unlike systematic reuse, which relies on pre-designed components intended for reuse, pragmatic reuse often involves modifying components not initially designed for reuse [46]. This approach is particularly relevant when developers must tailor components to fit unique project contexts, addressing specific functional or non-functional requirements. The process of pragmatic code reuse includes several key steps:

- Identifying suitable components
- Assessing their relevance to the current project
- Adapting components to fit specific project requirements

After identifying suitable components and during assessments, developers decide whether to reuse these components or rebuild them from scratch. The pragmatic software reuse process is sometimes time and effort-intensive, leading to many deviations, even more so if developers do not fully understand the hidden functionalities. This can lead to frustration and abandonment of the process.

LASSO addresses these challenges by providing advanced search and analysis capabilities that facilitate the discovery and evaluation of reusable software components on a large scale. It dramatically eases the first few steps of this process; given that it uses a test-driven approach and evaluates functional and non-functional requirements at a scale, it presents an easier way of quickly finding the most appropriate reusable code components. LASSO's role in the software reuse process is significant, as it streamlines the initial steps and sets the stage for further adaptation and integration. However, challenges still exist in adaptation to the new project requirements and successfully integrating it within the new context; Large Language Models (LLMs) prove invaluable here. This thesis examines whether leveraging high-quality code recommendations from LLMs can expedite and improve the adaptation process of these code components.

Considering we examine this integration in a black box fashion, in the lens of a survey, where the authors summarized 30 years of research on code search and the aspects which make code reuse and code search as efficient as possible[25]. We discuss how our approach supports these main requirements for efficient and good software reuse.

**Query Types and Formulation:** The survey categorizes query types into free-form, code-based, and custom querying languages, as shown in Figure 3.1 [25]. The survey highlights that free-form queries are the most commonly used type due to their ease of formulation and high level of expressiveness. While LASSO has its own precise and expressive language, our approach falls under both categories of free-form and custom queries, as it allows users to express their code search needs using natural language and allows for giving in the components for code search in a LASSO LSL script fashion.

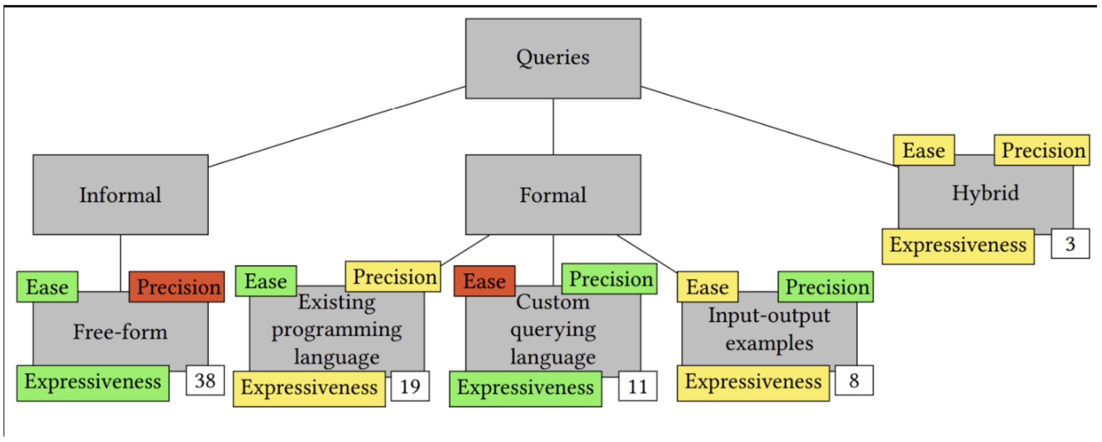


Figure 3.1.: Code Search Engines Query Formulations

**Query Preprocessing and Expansion:** The survey discusses various query preprocessing and expansion techniques using synonyms or related terms, such as query reformulation, query splitting, and query expansion. Although LASSO already has technical solutions, such as restricting and relaxing criteria for more diverse results, the LLM plays a crucial role in understanding the user's intent and generating appropriate LSL scripts based on the natural language query. The LLM enhances the query to match the desired code snippets better. Specifically, by leveraging LLMs for user intent understanding and use LASSO precisely to convey those intentions in a context-understanding manner.

**Indexing and Retrieval:** The survey covers different indexing and retrieval techniques, including text-based, graph-based, and deep learning-based approaches. LASSO employs advanced indexing and retrieval techniques to efficiently search and retrieve relevant code snippets from a large-scale corpus. The survey emphasizes the importance of scalability and efficiency in code search, which aligns with LASSO's capabilities.

**Ranking and Pruning:** The survey discusses various ranking and pruning techniques to improve the relevance and quality of search results. While LASSO can already be extended to actions to support further refinements, we employ the LLM-as-a-Judge component, which plays a crucial role in evaluating the relevance and quality of the retrieved code snippets based on predefined criteria.

Ideally, when using a perfect AI-copilot assistant, we expect to give it the problem, provide the context of the project, and get an adapted, easy-to-use, copy-paste-style implementation to reuse for our development goals. This process would mean, using LASSO and LLMs, in a white-box style:

1. Detect the scope of the needed components and in what context it is needed
2. formulate a precise code query in the form of an LSL script, which includes:
   - Determine the data source.
   - Determine the required interface.
   - Determine the tests retrieved methods need to pass to be considered in the evaluation process: LASSO, LLM or user-given tests.
   - Determine which extra actions are employable by LASSO to serve our specific purpose: Filter, Rank...etc.
3. Send LSL script (in an ideal world, the first search would yield useful implementations.
4. Retrieve implementations, specifically highly ranked implementations.
5. Check implementations for additional and project-specific requirements.
6. Provide these implementations as RAG to the LLM, with project context and requirements included.
7. Get a refined, well-adapted code to serve development requirements.

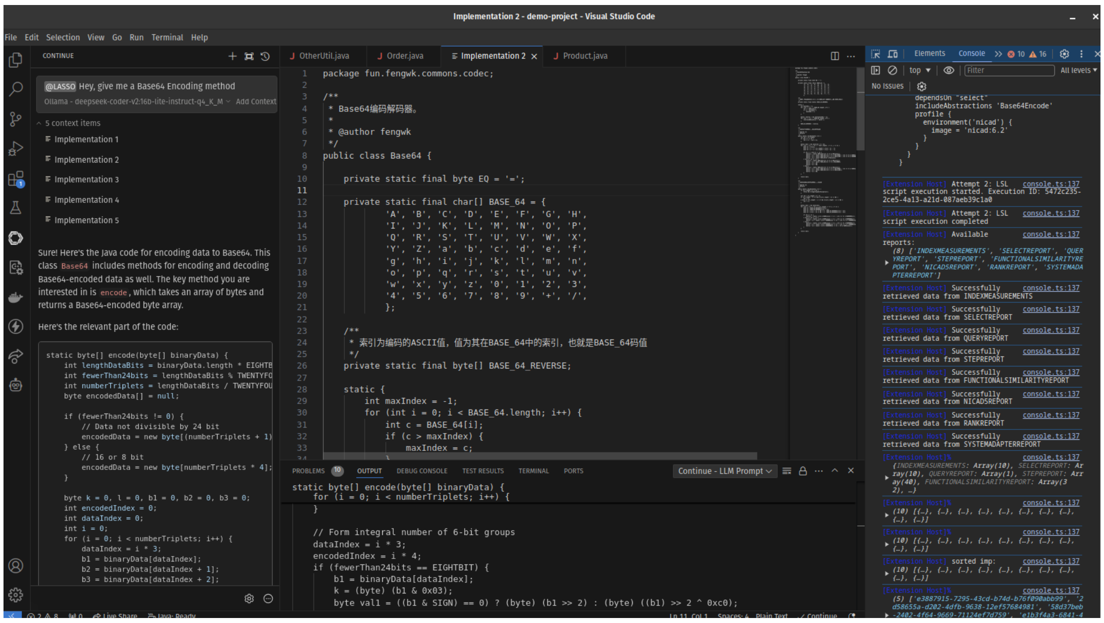
Although our integration may not represent the optimal solution for the aforementioned process, it constitutes a step towards setting the approach and seeking improvements.

## 3.2. General Considerations:

The most precise way to conduct a search using LASSO is by providing both the interface and the tests. The LLM fills these gaps in an LSL script that is then sent to LASSO. This allows the LLM to incorporate the interface and tests directly into the script. Our approach relies on a single LSL script where the LLM completes the missing elements, enabling LASSO to retrieve, filter, and rank implementations based on their functional similarity to the provided tests and how well they meet these criteria. Most importantly, the LLM takes in the context from the user and uses it to determine the interface and the tests. In the case where one of them or none of them is provided, the LLM tries to make an educated guess. The process highly depends on the user's context, instructions and the LLM's ability to understand the context and formulate the necessary query for the LSL script.

For complex implementations, such as retrieving a method that requires an object (e.g., a payment method has an input of an order object and a credit integer), precise instructions are crucial. For instance, if only the "sum" attribute of an order object is relevant, this must be specified to ensure accurate retrieval. LLM formulating an interface where it searches for an Order object in open-source repositories would likely not bring the desired results. The more precise the user query, the better the retrieval quality.

A significant finding of this thesis is that RAG (Retrieval-Augmented Generation) holds substantial power over generated code snippets. Not only by its functionality but also by its code style and granuality. For example: The following code snippet illustrates a case where the top-performing implementation retrieved by LASSO for a Base64 Encoding function may not be the most suitable case for reuse, the developer has manually crafted the encoding character by a character, which can lead to excellent performance, but has reduced readability and maintainability.



@LASSO Hey, give me a Base64 Encoding method

Sure! Here's the Java code for encoding data to Base64. This class `Base64` includes methods for encoding and decoding Base64-encoded data as well. The key method you are interested in is `encode`, which takes an array of bytes and returns a Base64-encoded byte array.

Here's the relevant part of the code:

```
static byte[] encode(byte[] binaryData) {
    int lengthDataBits = binaryData.length * EIGHTB
    int fewerThan24bits = lengthDataBits % TWENTYFO
    int numberTriplets = lengthDataBits / TWENTYFOU
    byte encodedData[] = null;

    if (fewerThan24bits != 0) {
        // Data not divisible by 24 bit
        encodedData = new byte[(numberTriplets + 1)
    } else {
        // 16 or 8 bit
        encodedData = new byte[numberTriplets * 4];
    }

    byte k = 0, l = 0, b1 = 0, b2 = 0, b3 = 0;
    int encodedIndex = 0;
    int dataIndex = 0;
    int i = 0;
    for (i = 0; i < numberTriplets; i++) {
        dataIndex = i * 3;
        b1 = binaryData[dataIndex];
        b2 = binaryData[dataIndex + 1];
        b3 = binaryData[dataIndex + 2];
```

```
package fun.fengwk.commons.codec;

/**
 * Base64编码解码器。
 *
 * @author fengwk
 */
public class Base64 {

    private static final byte EQ = '=';

    private static final char[] BASE_64 = {
            'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H',
            'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P',
            'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X',
            'Y', 'Z', 'a', 'b', 'c', 'd', 'e', 'f',
            'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n',
            'o', 'p', 'q', 'r', 's', 't', 'u', 'v',
            'w', 'x', 'y', 'z', '0', '1', '2', '3',
            '4', '5', '6', '7', '8', '9', '+', '/',
            };

    /**
     * 索引为编码的ASCII值，值为其在BASE_64中的索引，也就是BASE_64码值
     */
    private static final byte[] BASE_64_REVERSE;

    static {
        int maxIndex = -1;
        for (int i = 0; i < BASE_64.length; i++) {
            int c = BASE_64[i];
            if (c > maxIndex) {
                maxIndex = c;
```

```
static byte[] encode(byte[] binaryData) {
    for (i = 0; i < numberTriplets; i++) {
    }

    // Form integral number of 6-bit groups
    dataIndex = i * 3;
    encodedIndex = i * 4;
    if (fewerThan24bits == EIGHTBIT) {
        b1 = binaryData[dataIndex];
        k = (byte) (b1 & 0x03);
        byte val1 = ((b1 & SIGN) == 0) ? (byte) (b1 >> 2) : (byte) ((b1) >> 2 ^ 0xc0);
```

```
          dependsOn "select"
          includeAbstractions 'Base64Encode'
          profile {
            environment('nicad') {
              image = 'nicad:6.2'
            }
          }
        }
      }

[Extension Host] Attempt 2: LSL    console.ts:137
script execution started. Execution ID: 5472c235-
2ce5-4a13-a21d-087aeb39c1a0
[Extension Host] Attempt 2: LSL    console.ts:137
script execution completed
[Extension Host] Available         console.ts:137
reports:
  (8) ['INDEXMEASUREMENTS', 'SELECTREPORT', 'QUER
▸ YREPORT', 'STEPREPORT', 'FUNCTIONALSIMILARITYRE
  PORT', 'NICAD5REPORT', 'RANKREPORT', 'SYSTEMADA
  PTERREPORT']
[Extension Host] Successfully      console.ts:137
retrieved data from INDEXMEASUREMENTS
[Extension Host] Successfully      console.ts:137
retrieved data from SELECTREPORT
[Extension Host] Successfully      console.ts:137
retrieved data from QUERYREPORT
[Extension Host] Successfully      console.ts:137
retrieved data from STEPREPORT
[Extension Host] Successfully      console.ts:137
retrieved data from FUNCTIONALSIMILARITYREPORT
[Extension Host] Successfully      console.ts:137
retrieved data from NICAD5REPORT
[Extension Host] Successfully      console.ts:137
retrieved data from RANKREPORT
[Extension Host] Successfully      console.ts:137
retrieved data from SYSTEMADAPTERREPORT
[Extension Host]%                  console.ts:137
  {INDEXMEASUREMENTS: Array(10), SELECTREPORT: Ar
▸ ray(10), QUERYREPORT: Array(1), STEPREPORT: Arr
  ay(40), FUNCTIONALSIMILARITYREPORT: Array(3
  2), …}
[Extension Host]%                  console.ts:137
▸ (10) [{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
  {…}, {…}]
[Extension Host]%                  console.ts:137
▸ (10) [{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
  {…}, {…}]
[Extension Host] sorted imp:       console.ts:137
▸ (10) [{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
  {…}, {…}]
[Extension Host]%                  console.ts:137
  (5) ['e3887915-7295-43cd-b74d-b76f090abb99', '2
  d58655a-d202-4dfb-9638-12ef57684981', '58d37beb
▸ -2402-4f64-9669-71124ef7d759', 'e1b3f4a3-6841-4
```

Figure 3.2.: Asking LLM-LASSO for a Base64 Implementation

Adhering to certain principles enhances this process. For example, when retrieving open-source code methods, an alternative that simply requires an import and a single method reuse line offers a better and a simpler reuse potential.

This insight led to the development of an additional LLM-as-a-Judge feature. While the judgement may fall short in some cases, we believe in the potential of empowering the LLM dynamic judgement in this context, especially since we are not worried about functionality as much as adaptation at this stage of the process.

### 3.2.1. LLM-as-a-Judge:

Using an LLM to judge retrieved implementations has many benefits. In a conducted study by Koutcheme et al. [49] that even open source models (such as Llama 3) are on par with closed source, high reasoning GPT-4o models in terms of evaluating code and programming tasks LLMs could be great evaluators as, with employing the right frameworks or fine-tuned models, it achieves high rates of agreements with human evaluators while still using an open source small sized LLM model [85]. This highlights the possibility of a good LLM dynamic evaluator to keep bad RAG at bay.

Our LLM Judge evaluates implementations based on five criteria: functionality, readability, best practices, performance, and robustness. Each criterion is scored from 1 to 5, and implementations scoring below 15 are rejected. Typically, implementations score highly on functionality since LASSO already tested them, yet the LLM must justify its score for each criterion. A user-friendly aspect of this integration is its real-time process flow visibility and the ability to see rejected RAG and reuse it in case the developer does not agree.

## 3.3. Use Cases:

In the following, we go through the kind of use cases where the integration of LASSO with generative AI makes the most impact:

### 3.3.1. Natural Language Query:

This integration's most significant benefit and use case is the ability to conduct, retrieve, and adapt implementations based solely on natural language queries. Using the chatbot interface, developers can freely describe the functionality they need, and the AI evaluates the context to determine which implementations should be retrieved from LASSO's corpus. If the initially retrieved implementations are not satisfactory, the developer can refine their query, providing more details such as required interfaces or test cases to be used. The developer can formulate test cases with the help of the AI and use them for LASSO. All of this is done using natural language and leveraging open-source AI models. The developer can observe the retrieval process in the background, inspect the retrieval-augmented generation (RAG) results, see the LLM's evaluation scores for each implementation across multiple criteria, and adapt the code to their specific use case with the LLM's assistance. The integration leverages continue.dev capabilities enable the indexing of the developer's codebase, providing context to the LLM in a simple numerical form. It opens up the ability to integrate further AI capabilities, such as responding to slash commands and connecting extra platforms, such as combining SQL queries with LASSO-retrieved code, to solve more complex tasks.

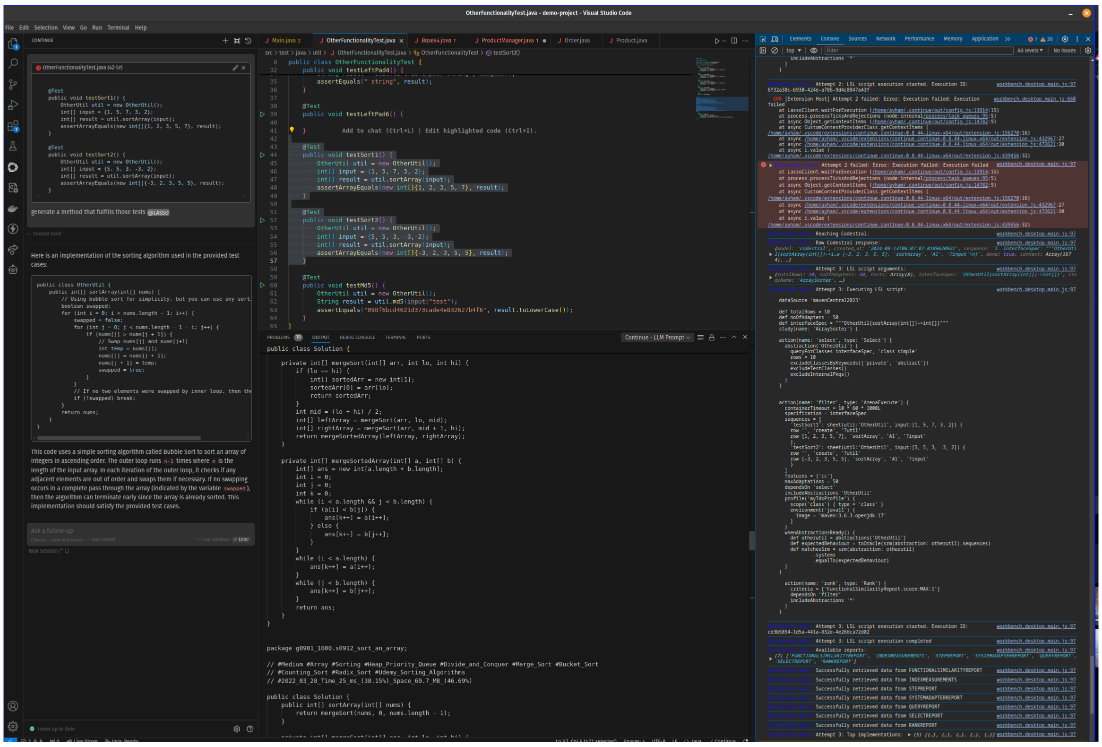
In an enterprise setting, LASSO could retrieve already deployed code from closed code repository and serve as an adapter between the different project requirements requiring similar methodologies. Leveraging already known high quality code from own database could prove very useful as the generated code is already adapted to the enterprise coding style and conventions without the explicit prompt tuning for every retrieval. A great use case for enterprises with recurring activities, such as building data pipelines for different data catalogues using the same methodology.

### 3.3.2. Unified Searching Approach:

A key benefit is the unification of LASSO's various interface-driven, test-driven, and code-driven search approaches through the AI's natural language processing capabilities. Developers can formulate queries and the AI infers the most appropriate approach based on the query. For example, if the developer provides a detailed description of the desired component's interface, the AI recognizes this and initiates an interface-driven search, automatically generating tests for it then sends an LSL script with those details. The AI triggers a test-driven search if the query includes specific test cases or expected behaviours and search. In case the user provided a code snippet, the LLM could generate input output interface based on it, tests on the expected use and search for alternative implementations. The unified search approach improves usability, flexability and seemless integration into the developer's workflow In the following Figure, we can see how just referencing the tests leads to sending an LSL query with the defined tests and retrieves implementations based on them.

### 3.3.3. Use Cases Scenarios:

- Adhering to non-functional requirements
  - Scenario: User would like to implement a function that has multiple project requirements, most of them represent test edge cases that are critical for their development purpose
  - Solution: The integration, takes in the project requirements in natural language, formulates respectice use cases and helps the developer finding concrete implementations that supports them.


OtherFunctionalityTest.java - demo-project - Visual Studio Code
File Edit Selection View Go Run Terminal Help
CONTINUE
Main.java 2
OtherFunctionalityTest.java
Base64.java 1
ProductManager.java 1
Order.java
Product.java
src > test > java > util > OtherFunctionalityTest.java > OtherFunctionalityTest > testSort2()
Elements Console Sources Network Performance Memory Application
top
Filter
All levels
No Issues
PROBLEMS 10 OUTPUT DEBUG CONSOLE TERMINAL PORTS
Continue - LLM Prompt

OtherFunctionalityTest.java (42-57)

```
    @Test
    public void testSort1() {
        OtherUtil util = new OtherUtil();
        int[] input = {1, 5, 7, 3, 2};
        int[] result = util.sortArray(input);
        assertArrayEquals(new int[]{1, 2, 3, 5, 7}, result);
    }

    @Test
    public void testSort2() {
        OtherUtil util = new OtherUtil();
        int[] input = {5, 5, 3, -3, 2};
        int[] result = util.sortArray(input);
        assertArrayEquals(new int[]{-3, 2, 3, 5, 5}, result);
    }
```

generate a method that fulfills those tests @LASSO

Context Used

Here is an implementation of the sorting algorithm used in the provided test cases:

```
public class OtherUtil {
    public int[] sortArray(int[] nums) {
        // Using bubble sort for simplicity, but you can use any sort
        boolean swapped;
        for (int i = 0; i < nums.length - 1; i++) {
            swapped = false;
            for (int j = 0; j < nums.length - 1 - i; j++) {
                if (nums[j] > nums[j + 1]) {
                    // Swap nums[j] and nums[j+1]
                    int temp = nums[j];
                    nums[j] = nums[j + 1];
                    nums[j + 1] = temp;
                    swapped = true;
                }
            }
            // If no two elements were swapped by inner loop, then th
            if (!swapped) break;
        }
        return nums;
    }
}
```

This code uses a simple sorting algorithm called Bubble Sort to sort an array of integers in ascending order. The outer loop runs `n-1` times where `n` is the length of the input array. In each iteration of the outer loop, it checks if any adjacent elements are out of order and swaps them if necessary. If no swapping occurs in a complete pass through the array (indicated by the variable `swapped`), then the algorithm can terminate early since the array is already sorted. This implementation should satisfy the provided test cases.

Ask a follow-up
Ollama - codestral:latest + Add Context ^ Use codebase Enter
New Session (^ L)

```
public class OtherFunctionalityTest {
    public void testLeftPad4() {
        assertEquals(" string", result);
    }

    @Test
    public void testLeftPad6() {

    }          Add to chat (Ctrl+L) | Edit highlighted code (Ctrl+I).

    @Test
    public void testSort1() {
        OtherUtil util = new OtherUtil();
        int[] input = {1, 5, 7, 3, 2};
        int[] result = util.sortArray(input);
        assertArrayEquals(new int[]{1, 2, 3, 5, 7}, result);
    }

    @Test
    public void testSort2() {
        OtherUtil util = new OtherUtil();
        int[] input = {5, 5, 3, -3, 2};
        int[] result = util.sortArray(input);
        assertArrayEquals(new int[]{-3, 2, 3, 5, 5}, result);
    }

    @Test
    public void testMd5() {
        OtherUtil util = new OtherUtil();
        String result = util.md5(input:"test");
        assertEquals("098f6bcd4621d373cade4e832627b4f6", result.toLowerCase());
    }
}
```

```
public class Solution {

    private int[] mergeSort(int[] arr, int lo, int hi) {
        if (lo == hi) {
            int[] sortedArr = new int[1];
            sortedArr[0] = arr[lo];
            return sortedArr;
        }
        int mid = (lo + hi) / 2;
        int[] leftArray = mergeSort(arr, lo, mid);
        int[] rightArray = mergeSort(arr, mid + 1, hi);
        return mergeSortedArray(leftArray, rightArray);
    }

    private int[] mergeSortedArray(int[] a, int[] b) {
        int[] ans = new int[a.length + b.length];
        int i = 0;
        int j = 0;
        int k = 0;
        while (i < a.length && j < b.length) {
            if (a[i] < b[j]) {
                ans[k++] = a[i++];
            } else {
                ans[k++] = b[j++];
            }
        }
        while (i < a.length) {
            ans[k++] = a[i++];
        }
        while (j < b.length) {
            ans[k++] = b[j++];
        }
        return ans;
    }
}


package g0901_1000.s0912_sort_an_array;

// #Medium #Array #Sorting #Heap_Priority_Queue #Divide_and_Conquer #Merge_Sort #Bucket_Sort
// #Counting_Sort #Radix_Sort #Udemy_Sorting_Algorithms
// #2022_03_28_Time_25_ms_(38.15%)_Space_69.7_MB_(46.69%)

public class Solution {
    public int[] sortArray(int[] nums) {
        return mergeSort(nums, 0, nums.length - 1);
    }
```

```
      includeAbstractions '*'
    }
  }

[Extension Host] Attempt 2: LSL script execution started. Execution ID:         workbench.desktop.main.js:97
6f32a38c-b930-424e-a786-9d4c8847a43f
  ERR [Extension Host] Attempt 2 failed: Error: Execution failed: Execution     workbench.desktop.main.js:660
failed
    at LassoClient.waitForExecution (/home/ayham/.continue/out/config.js:13914:15)
    at process.processTicksAndRejections (node:internal/process/task_queues:95:5)
    at async Object.getContextItems (/home/ayham/.continue/out/config.js:14782:9)
    at async CustomContextProviderClass.getContextItems (
/home/ayham/.vscode/extensions/continue.continue-0.8.44-linux-x64/out/extension.js:156270:16)
    at async /home/ayham/.vscode/extensions/continue.continue-0.8.44-linux-x64/out/extension.js:432967:27
    at async /home/ayham/.vscode/extensions/continue.continue-0.8.44-linux-x64/out/extension.js:472621:20
    at async i.value (
/home/ayham/.vscode/extensions/continue.continue-0.8.44-linux-x64/out/extension.js:439456:32)
▸ [Extension Host] Attempt 2 failed: Error: Execution failed: Execution failed  workbench.desktop.main.js:97
    at LassoClient.waitForExecution (/home/ayham/.continue/out/config.js:13914:15)
    at process.processTicksAndRejections (node:internal/process/task_queues:95:5)
    at async Object.getContextItems (/home/ayham/.continue/out/config.js:14782:9)
    at async CustomContextProviderClass.getContextItems (
/home/ayham/.vscode/extensions/continue.continue-0.8.44-linux-x64/out/extension.js:156270:16)
    at async /home/ayham/.vscode/extensions/continue.continue-0.8.44-linux-x64/out/extension.js:432967:27
    at async /home/ayham/.vscode/extensions/continue.continue-0.8.44-linux-x64/out/extension.js:472621:20
    at async i.value (
/home/ayham/.vscode/extensions/continue.continue-0.8.44-linux-x64/out/extension.js:439456:32)
[Extension Host] Reaching Codestral.                                            workbench.desktop.main.js:97
[Extension Host] Raw Codestral response:                                        workbench.desktop.main.js:97
  {model: 'codestral', created_at: '2024-08-13T06:07:07.010962092Z', response: `1. interfaceSpec: """OtherUti
▸ l{sortArray(int[])->i…w [-3, 2, 3, 5, 5], 'sortArray', 'A1', '?input'\n}`, done: true, context: Array(167
  4), …}
[Extension Host] Attempt 3: LSL script arguments:                               workbench.desktop.main.js:97
▸ {totalRows: 10, noOfAdapters: 50, tests: Array(8), interfaceSpec: 'OtherUtil{sortArray(int[])->int[]}', stu
  dyName: 'ArraySorter', …}
[Extension Host] Attempt 3: Executing LSL script:                               workbench.desktop.main.js:97

    dataSource 'mavenCentral2023'

    def totalRows = 10
    def noOfAdapters = 50
    def interfaceSpec = """OtherUtil{sortArray(int[])->int[]}"""
    study(name: 'ArraySorter') {

    action(name: 'select', type: 'Select') {
      abstraction('OtherUtil') {
        queryForClasses interfaceSpec, 'class-simple'
        rows = 10
        excludeClassesByKeywords(['private', 'abstract'])
        excludeTestClasses()
        excludeInternalPkgs()
      }
    }


    action(name: 'filter', type: 'ArenaExecute') {
      containerTimeout = 10 * 60 * 1000L
      specification = interfaceSpec
      sequences = [
        'testSort1': sheet(util:'OtherUtil', input:[1, 5, 7, 3, 2]) {
        row '', 'create', '?util'
        row [1, 2, 3, 5, 7], 'sortArray', 'A1', '?input'
        },
        'testSort2': sheet(util:'OtherUtil', input:[5, 5, 3, -3, 2]) {
        row '', 'create', '?util'
        row [-3, 2, 3, 5, 5], 'sortArray', 'A1', '?input'
        }
      ]
      features = ['cc']
      maxAdaptations = 50
      dependsOn 'select'
      includeAbstractions 'OtherUtil'
      profile('myTdsProfile') {
        scope('class') { type = 'class' }
        environment('java11') {
          image = 'maven:3.6.3-openjdk-17'
        }
      }
      whenAbstractionsReady() {
        def otherutil = abstractions['OtherUtil']
        def expectedBehaviour = toOracle(srm(abstraction: otherutil).sequences)
        def matchesSrm = srm(abstraction: otherutil)
                .systems
                .equalTo(expectedBehaviour)
      }
    }

      action(name: 'rank', type: 'Rank') {
        criteria = ['FunctionalSimilarityReport.score:MAX:1']
        dependsOn 'filter'
        includeAbstractions '*'
      }
    }

[Extension Host] Attempt 3: LSL script execution started. Execution ID:         workbench.desktop.main.js:97
cb3b5854-1d5a-441a-832e-4e266ca72d02
[Extension Host] Attempt 3: LSL script execution completed                      workbench.desktop.main.js:97
[Extension Host] Available reports:                                             workbench.desktop.main.js:97
▸ (7) ['FUNCTIONALSIMILARITYREPORT', 'INDEXMEASUREMENTS', 'STEPREPORT', 'SYSTEMADAPTERREPORT', 'QUERYREPORT',
  'SELECTREPORT', 'RANKREPORT']
[Extension Host] Successfully retrieved data from FUNCTIONALSIMILARITYREPORT    workbench.desktop.main.js:97
[Extension Host] Successfully retrieved data from INDEXMEASUREMENTS             workbench.desktop.main.js:97
[Extension Host] Successfully retrieved data from STEPREPORT                    workbench.desktop.main.js:97
[Extension Host] Successfully retrieved data from SYSTEMADAPTERREPORT           workbench.desktop.main.js:97
[Extension Host] Successfully retrieved data from QUERYREPORT                   workbench.desktop.main.js:97
[Extension Host] Successfully retrieved data from SELECTREPORT                  workbench.desktop.main.js:97
[Extension Host] Successfully retrieved data from RANKREPORT                    workbench.desktop.main.js:97
[Extension Host] Attempt 3: Top implementations: ▸ (5) [{…}, {…}, {…}, {…}, {…}] workbench.desktop.main.js:97
```

Figure 3.3.: Test Driven Approach

- Learning and Exploration
  - Scenario: A developer wants to explore different solutions to solve a problem.
  - Solution: The integration, backed by LASSO's diverse implementation retrieval offers diverse examples, that could be adapted to the different paradigms and structures.
- Updated API Usage Examples
  - Scenario: A developer is using an unfamiliar, outdated API and needs examples of how to implement certain methods.
  - Solution: Given LASSO's flexible infrastructure to keep the most recent updates indexed, it provides recent code snippets and examples of the API usage, helping the developer stay up-to-date within the workflow.
- Debugging and Issue Resolution

  - Scenario: A developer encounters a bug and needs to understand its root cause.
  - Solution: The LLM formulates tests to retrieve similar implementations and help compare the logic to find out where the bug is localized.

- Security and Compliance Checks
  - Scenario: A developer needs to ensure that the his code comply with security standards.
  - Solution: The LLM helps formulate tests, that take security risks into consideration and chooses one of LASSO retrieved implementations, that best mitigate those risks.

- Cross-Language Code Translation
  - Scenario: LASSO currently supports only java and a programmer wants to write a function in python.
  - Solution: The integration suggests equivalent code snippets in the target language, while relying on the functional java code logic.

# 4. Integrating LASSO With Generative AI

This chapter explores the technical details of integrating LASSO with LLM models. We discuss the different components and communication involved in this integration and present our integration's technical details. Figure 4.1 provides a general overview of how this integration works, where continues.dev is the main LLM call, for which RAG is provided to:

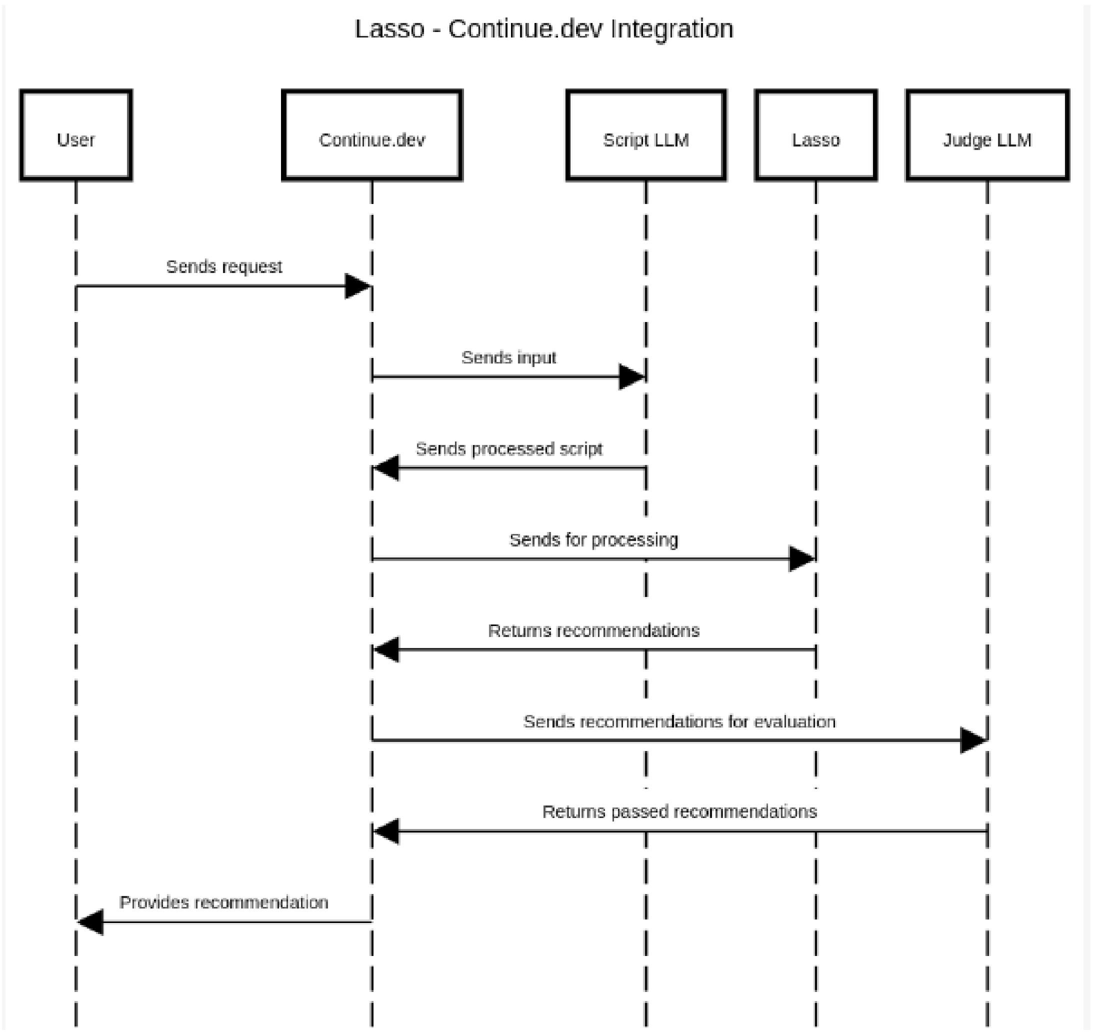


Figure 4.1.: Integration Overview

## 4.1. Continue.dev

Continue.dev is a sophisticated open-source AI assistant designed to seamlessly incorporate AI capabilities into IDEs through a plugin-based architecture. It helps developers build and deploy AI-integrated solutions that are very customizable depending on their needs. This section provides an overview of Continue.dev's key features and functionalities that are particularly relevant for integrating LASSO.

Relevant key features include:

- **AI-Driven Code Assistance:** Utilizes various AI models to support different development tasks, including: chat, embedding, autocomplete, reranking.
- **Data Persistence and Improvement:** Chat history, queries and responses are saved locally and are eused to refine the used AI model and enhance its capabilities over time, they could also be shared across teams.
- **Integration with Hosts and Model Providers:** Supports integration with a wide range of AI model providers, whether its an open-source Ollama, LM-studio or proprietary models requiring API keys. Simple copy and paste into the designated fields allow for flexible combinations across various models and host services through a unified interface.
- **Codebase Indexing:** Continue.dev automatically indexes open projects and allow for relevant context retrieval in the workspace through a combination of embeddings and keyword search.
- **Extensibility and Customization:** continue.dev allows developers to create custom structures and extend its functionality in a simple plug-and-play fashion. Some of these structures include the creation of context providers and slash commands, these concepts provide additional information to the LLM as context providers or represent actions for recurring tasks. (It is possible to integrate LASSO as a slash command but it makes more sense to do so as a context provider.)

### 4.1.1. Context Providers

Context providers in Continue.dev enable the reference or integration of information sources, particularly those unavailable within the workspace. They play a crucial role in integrating external data sources with LLMs, enhancing the AI's understanding and responsiveness to the context. Developers can type the '@' symbol in the chat window followed by a keyword to access various data sources that can be fed to the LLM as context. Built-in context providers include:

1. **@codebase and @file:** Incorporates context from within the workspace.
2. **@docs:** Enables referencing any documentation website.
3. **@google:** Provides web search functionality.
4. **@repo-map:** Offers an overview of all files and call signatures of top-level classes.

It is crucial to note that context provider references do not stack. The call for a context provider, such as @docs, followed by @LASSO does not give the documentation details to LASSO, which is the reason that mentioning @codebase and then @lasso does not give the codebase to the query sent to lasso, it has to be forwarded through the chat window. It is, however, possible to extract IDE information and user queries and forward them to a custom context provider.

We integrate LASSO as a custom context provider, as described in the continue.dev documentation [1] .

### 4.1.2. Custom Context Providers

The simplicity of adding a custom context provider ensures easy replication and seamless integration without the need to manage complex configurations. The process proceeds as follows:

1. **Development of a TypeScript Module:** The creation of a custom context provider for LASSO begins with developing a TypeScript module. This module contains the necessary logic and must conform to Continue.dev's interface.

[1]Continue.dev. Build your own context provider, Available: `https://docs.continue.dev/customize/tutorials/build-your-own-context-provider`.

2. **Integration into the Home Directory:** Once developed, the TypeScript module is placed within Continue.dev's home directory. This allows Continue.dev to detect and automatically incorporate the new context provider upon initialization. (Restarting the plugin is necessary each time the code is modified.)
3. **Definition of Functional Actions and Data Flow:** Within this module, functional actions are defined to enable robust communication with LASSO's API. These actions include executing search queries, retrieving relevant results, and processing these results to ensure their contextual relevance and utility.

The context provider effectively bridges Continue.dev with LASSO's extensive code search and analysis functionalities by establishing comprehensive Lasso client to handle API calls.

The main integration file is called **LassoContextProvider.ts**.

## 4.2. Process Flow

The `LassoContextProvider.ts` module follows a well-defined process flow to retrieve and present relevant code snippets to the user:

**Process Flow:**

1. Upon receiving a query from the user, the context provider generates an LSL script using the provided input. Based on instructions, it formulates and fills in the gaps in the script and sends it to LASSO for execution. It waits for the execution until the status FAILED or SUCCEEDED is retrieved. If the execution fails, the script is reformulated and resent to LASSO, with a maximum of three attempts, each time trying to formulate the desired interface differently.
2. The generated LSL script is then executed through LASSO using the `LassoClient`. LASSO processes the script, awaits execution, searches for the ranking report, and retrieves the top five implementations from its repository. In case there is no ranked report but there is a selection report, it selects the first five and relies on the Judge LLM for its judgment. Otherwise, another script is resent

3. The retrieved code snippets (Top five) are processed, and the top implementations are extracted. These implementations represent the most relevant code snippets based on the user's query.
4. The extracted implementations are evaluated for relevance using the `judgeRelevanceBatc` function. This evaluation depends on the criteria: Functionality, Readability, Best Practices, performance and robustness. Each from (1-5) and the final score is summed.
5. Finally, the accepted relevant code snippets(score 15 or higher from the judge LLM) are presented as context items within Continue.dev, providing the user with a curated set of code recommendations that align with their query.

**Error Handling:** Incorporates multiple robust error handling mechanisms, with a substantial amount of logging, these can be previewed in real time while the interactions are taking place by using the developer panel (usually under the option: Toggle Developer Tools) and includes retries with exponential backoff, to ensure reliability in retrieving valid implementations even when initial attempts fail.

## 4.3. LassoClient.ts

As mentioned earlier, LASSO provides a comprehensive REST API that facilitates seamless communication with external tools, to facilitate the process flow from earlier, we include the `LassoClient.ts`. Within this module, we handle the main interactions with key API endpoints:

- **Authentication Endpoint:** Obtains an OAuth token for secure communication.
- **Execution Endpoint:** Executes an LSL script based on user queries.
- **Status Endpoint:** Retrieves the execution status of scripts.
- **Result Retrieval Endpoint:** Obtains execution results for further processing.

**API Calls Order:** The order of API calls is as follows:

1. Authenticate and obtain an OAuth token.

2. Execute the generated LSL script.
3. Retrieve the execution status and wait for completion.
4. Obtain the execution result and extract the top implementations.
5. Retrieve the actual code snippets for the top implementations.

The `LassoClient` class abstracts the complexities of the API interactions, providing a clean and intuitive interface for the `LassoContextProvider` to utilize LASSO's functionalities.

## 4.4. LLMIntegration.ts

This module generates, evaluates, and refines LSL scripts based on user queries using large language models (LLMs). It bridges the gap between natural language input from users and LASSO's structured script requirements. The integration involves three distinct roles and calls for LLMs. We refer to them in the remainder of the thesis as follows:

1. **Main LLM (Continue.dev):** This LLM is the primary LLM for user interactions, processes user queries, provides initial code recommendations and receives RAG to enhance suggestions based on the provided context.
2. **Script LLM:** This LLM generates the necessary components of the LSL script. It interprets user queries to fill out script components, such as interface specifications and test sequences.
3. **Judge LLM:** As previously discussed, this LLM evaluates the relevance and quality of retrieved implementations, ensures that only high-quality and relevant code recommendations are presented to the user and enables a dynamic evaluation in case the rank report does not bring valuable results.

The Main LLM role as the developer's AI assistant is integrated into continue.dev. It manages the adaptation and integration of the pragmatic software reuse process. This is the reason we do not focus on the model selection for this LLM as much, it does make a difference in, which model is using the RAG, but objectively speaking all tested LLM models in this role were affected by RAG to a considerable degree and they based their suggested implementations around the given RAG, the better the model, the better the adaptation, but we do not view the model selection for this role as critical as the other roles.

For the other two Roles, we discuss the model selection in the following:

### 4.4.1. Selected Models

One of the primary objectives of this thesis is to improve the performance and applicability of small to mid-sized large language models (around 8-22 billion parameters) for software engineering tasks. These smaller models have become increasingly effective and offer more flexibility as they can be hosted on commercial personal-use computers. Consequently, we implemented this setup using Ollama [70], through Ollama-js [69].

One promising architecture in LLMs that achieves good results while not being very computationally costly is theMixture of Experts MoE architecture, especially in the case of DeepSeek architecture, upon which DeepSeek-Coder-V2 [99] is built, it activates only a subset of the parameters that are considered experts for the specific query [23], allowing for better results while using less GPU memory than codestral [4]. While scaling is a definite way of yielding higher results [18], it does not necessarily mean it is the most efficient way.

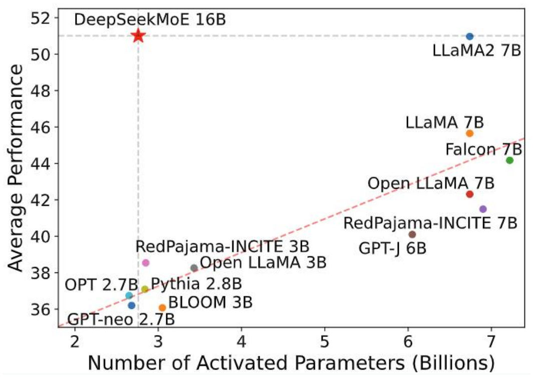


Figure 4.2.: Mixture of Experts Architecture Performance [23]

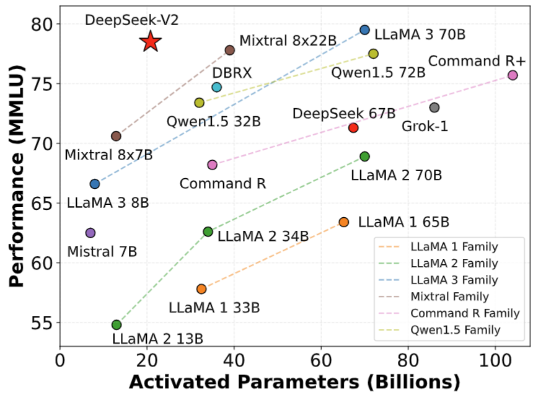
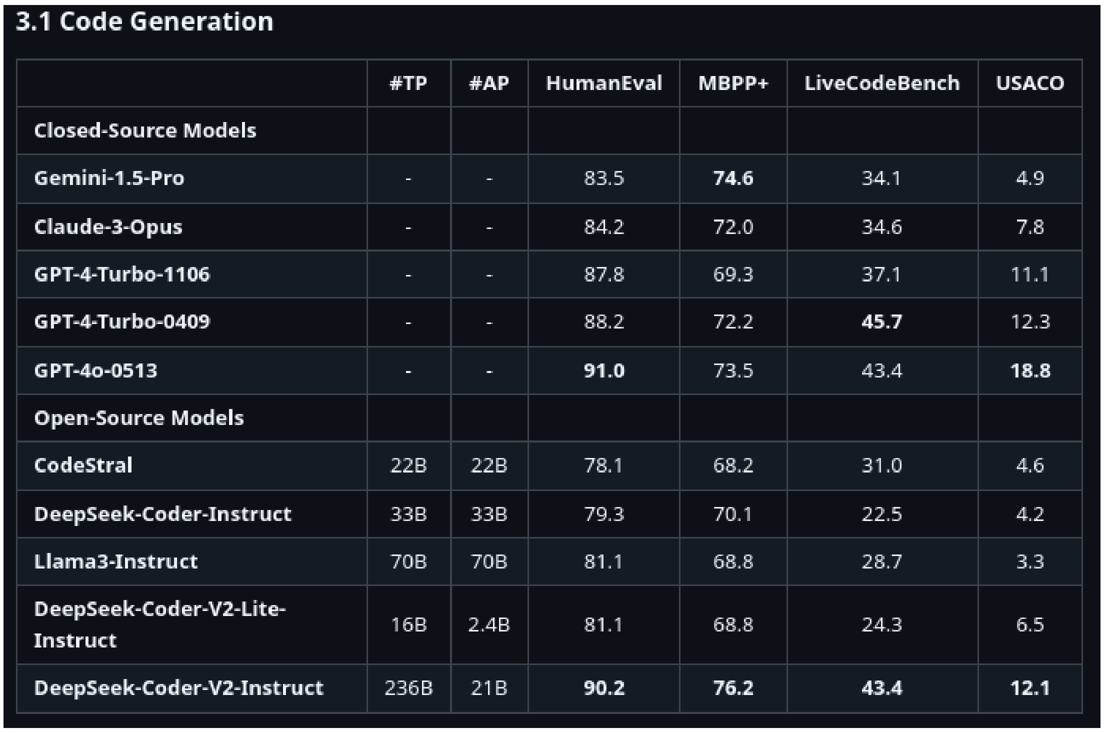


Figure 4.3.: Mixture of Experts Architecture Performance 2

In this thesis, we explore our integration using different models such as Llama 3.1 (8 Billion variant), Codestral [4], NexusRaven-V2 [83], and DeepSeek-CoderV2 [99].



3.1 Code Generation

| | #TP | #AP | HumanEval | MBPP+ | LiveCodeBench | USACO |
|---|---|---|---|---|---|---|
| **Closed-Source Models** | | | | | | |
| **Gemini-1.5-Pro** | - | - | 83.5 | **74.6** | 34.1 | 4.9 |
| **Claude-3-Opus** | - | - | 84.2 | 72.0 | 34.6 | 7.8 |
| **GPT-4-Turbo-1106** | - | - | 87.8 | 69.3 | 37.1 | 11.1 |
| **GPT-4-Turbo-0409** | - | - | 88.2 | 72.2 | **45.7** | 12.3 |
| **GPT-4o-0513** | - | - | **91.0** | 73.5 | 43.4 | **18.8** |
| **Open-Source Models** | | | | | | |
| **CodeStral** | 22B | 22B | 78.1 | 68.2 | 31.0 | 4.6 |
| **DeepSeek-Coder-Instruct** | 33B | 33B | 79.3 | 70.1 | 22.5 | 4.2 |
| **Llama3-Instruct** | 70B | 70B | 81.1 | 68.8 | 28.7 | 3.3 |
| **DeepSeek-Coder-V2-Lite-Instruct** | 16B | 2.4B | 81.1 | 68.8 | 24.3 | 6.5 |
| **DeepSeek-Coder-V2-Instruct** | 236B | 21B | **90.2** | **76.2** | **43.4** | **12.1** |

Figure 4.4.: Model Comparison

Shortly before submitting this thesis, the Qwen2.5-Coder was released, and the results were very impressive. However, it is comparable to DeepSeek-Coder-V2 and is more computationally expensive to host since it uses normal transformer architecture. We initially tried with codestral. Codestral was pretty good at producing outputs, with around a 70% success rate for writing the script. Yet, it kept writing wrong tests, even though they were sometimes given and only needed a "copy-paste" into the LSL script, which led to its discontinuation for this role.

Ultimately, DeepSeek-Coder-V2 proved highly effective in generating correct components and rivalled Qwen2.5-Coder with nearly a 100% success rate (depending on the user query). In general, we only have to ask it once, as the correct script was resolved within three tries. (Components from past failed scripts are given again to the script LLM, so it can retry with different naming strategies.)

However, even when formulating a correct LSL script, it does not necessarily mean that it retrieves any implementations in the rank report. When the rank report exists, it means that there is a ranking based on functional similarity and that we can retrieve them and provide them as RAG; however, when there are none, yet there exists multiple in the select report, we include them and rely on the Judge LLM. If no implementations are found, the LSL script is reformulated, and another script gets sent after 3 tries; if none are found, we forward no context to the main LLM, and no RAG is used.

In the following, we explore the RAG retrieval process and the Script LLM's handling of the component generation for the LSL script.

### 4.4.2. Script LLM

The way we ask LASSO to retrieve implementations is through its LSL Script. This represents a domain-specific pipeline language that allows for an efficient search for the components based on the defined Interface and execution of the tests, defined as test sequences following a specific LASSO format. In the following, we break down key components of the LSL script and examine what they do.

#### 4.4.2.1. Key Components of the LSL Script

We predefined the data source and the number of total rows and adapters as we made a minimal setup for the integration, hosting all of its components on one device. However, LASSO's scalable architecture would allow the retrieval of more components and a bigger execution arena if it was hosted separately. That also means that it could serve multiple developers working with this integration at the same time.

Another note is that we split the main script into parts and encapsulated them in methods. This was done to support AI models that are non-chatty but very good at function calling, such as the 'NexusRaven-V2 '. However, the implementation of this model was not very successful, as it did not provide the proper format or formulate the interface correctly. But with the fast pace of AI models development, this could change anytime in the near future.

**Data Source and Variables** This section defines the data source and key variables used throughout the script. The total rows and number of adapters are used to increase the number of retrievals and the number of diverse implementations; we used a setup of 10 and 50, which is considered small.

```
dataSource 'mavenCentral2023'
def totalRows = ${totalRows}
def noOfAdapters = ${noOfAdapters}
def interfaceSpec = """${interfaceSpec}"""
```

**Study Definition** The study block, serving as the cornerstone of the LSL script, encapsulates all actions that are to happen on the retrieved implementations.

```
study(name: '${studyName}') {
  // Study actions here
}
```

**Select Action** This action performs the initial selection of potential method implementations based on the interface specification.

```
action(name:'select', type: 'Select') {
  abstraction('${abstractionName}') {
    queryForClasses interfaceSpec, 'class-simple'
    rows = ${totalRows}
    excludeClassesByKeywords(['private', 'abstract'])
    excludeTestClasses()
    excludeInternalPkgs()
  }
}
```

**Filter Action** The filter action executes the selected implementations in the arena setup (containerized secure environment) to be executed on the test sequences.

```
action(name: 'filter', type: 'ArenaExecute') {
  containerTimeout = 10 * 60 * 1000L
  specification = interfaceSpec
  sequences = [
    ${tests.join(',\n          ')}
  ]
  features = ['cc']
  maxAdaptations = ${noOfAdapters}
  dependsOn 'select'
  includeAbstractions '${abstractionName}'
  profile('myTdsProfile') {
    scope('class') {
      type = 'class'
    }
    environment('java17') {
      image = 'maven:3.6.3-openjdk-17'
    }
  }
}
```

**Rank Action** This action ranks the filtered implementations based on their functional similarity scores. (how well they passed the given tests)

```
action(name: 'rank', type: 'Rank') {
  criteria = ['FunctionalSimilarityReport.score:MAX:1']
  dependsOn 'filter'
  includeAbstractions '*'
}
```

**Clone Detection** As mentioned before, LASSO includes Nicad to detect clones. We support the removal of type 1 and type 2 clones, which are code snippets that are exactly the same, syntactically or semantically. The main goal behind it is to increase diversity from the retrieved results.

```
action(name: "clones", type: 'Nicad6') {
  cloneType = "type2"
  collapseClones = true
  dependsOn "select"
  includeAbstractions '${abstractionName}'
  profile {
    environment('nicad') {
      image = 'nicad:6.2'
    }
  }
}
```

#### 4.4.2.2. Script LLM Prompt

The Script LLM Prompt is designed to generate LSL script components based on user queries. This system uses a few-shot prompting approach, providing examples for each template-filling action. Out of all combinations, we got the best responses by first defining the task, giving instructions for the expected generations, each with an example, defining the output format and then giving additional secondary instructions and complete examples. The logic behind it is that the instructions at the start are of the utmost importance, and the output instructions and examples at the end ensure the output format. (We did not include the examples for readability) Below, we break down the prompt's key components and functionalities.

**Prompt Initialization** The function begins by checking if there are any previous components from a failed generation attempt. If so, it includes instructions to modify the interface name and provides details of the last attempt.

```
export async function processUserQuery(query: string,
   lastComponents?: LslScriptArgs): Promise<string> {
  const previousInstructions = lastComponents
    ? `
- Last generation failed. Please generate a different
   interface name that describes the method.
- Last interface spec: "${lastComponents.interfaceSpec}"
- Last abstraction name: "${lastComponents.abstractionName
   }"
- Last abstraction variable name: "${lastComponents.
   abstractionVariableName}"
```

```
    `
    : '';
```

**Task Description** The main task is to generate LSL script components based on a user query. The instructions guide the generation of specific components required by LASSO for retrieving Java snippets.

```
const prompt = `
Task: Generate LSL script components based on a user query.

Instructions:
${previousInstructions}
- LASSO retrieves Java snippets and requires an LSL script.
- Based on the user query, generate the following
   components:
```

**Component Specifications** The prompt specifies how to generate the interface and each variable component It emphasizes correct formatting and provides examples for clarity.

```
1. **interfaceSpec**: Method interface in LQL notation.
   - Format: """<interface_spec>"""
   - Simplify object inputs by inferring appropriate data
      types based on context.
   - Example: """PalindromeGenerator{generatePalindrome(int
      )->int}"""

2. **studyName**: A meaningful name for the study.
   - Example: Derived from interfaceSpec, such as "
      CalculatePrice Study"

3. **abstractionName**: A meaningful abstraction name.
   - Example: Derived from studyName, such as "
      CalculatePrice"

4. **abstractionVariableName**: A meaningful abstraction
   variable name.
   - Convert abstractionName to a variable-friendly format.
   - Example: "calculatePrice"
```

**Test Sequences** The prompt includes instructions for generating comprehensive test sequences to filter candidates, ensuring outputs are in a specified format without using JSON.

```
5. **testSequences**: Comprehensive test sequences to
   filter candidates.
   - **IMPORTANT**: Generate the output in the exact format
       below. Do not use JSON.

\`\`\`
interfaceSpec: """<interface_spec>"""
studyName: <study_name>
abstractionName: <abstraction_name>
abstractionVariableName: <abstraction_variable_name>
testSequences:
'test_name_1': sheet(<parameters>) {
    row <output>, '<method_name>', <input1>, <input2>
    row <output>, '<method_name>', <input1>, <input2>
},
'test_name_2': sheet(<parameters>) {
    row <output>, '<method_name>', <input1>, <input2>
    row <output>, '<method_name>', <input1>, <input2>
}
\`\`\`
```

**Additional Instructions** Additional guidelines ensure correct formatting and context prediction, emphasizing no use of explanations or extra text.

```
Additional Instructions:
- Ensure the interface is correctly formatted, e.g., """
   PalindromeGenerator{generatePalindrome(int)->int}"""
- In case inputs or outputs are objects, guess the most
   likely basic data type instead.
```

```
- Use context to predict method functionalities and
   appropriate online method names.
- Do not use the '%' sign in test sequences.
- If tests are provided, incorporate them directly into the
    testSequences section, do not generate extra tests
   unless explicitly asked to do so.
- Exclude any explanations or additional text.

Generate the LSL script components for the following query:

**User Query:** ${query}
```

This detailed breakdown of the Script LLM Prompt ensures that each component is generated accurately and efficiently, enhancing the overall code retrieval process within LASSO.

### 4.4.3. Judge LLM

In our approach, while we do retrieve methods with LASSO based on functionality score and inputs/outputs, we implement an LLM-as-a-Judge approach, evaluate implementations and assign scores based on 5 criteria, for each we give 5 points and we ask the model to justify its scores. If the LLM fails to provide a score or if score extraction fails due to output format issues, a default middle score is assigned for every implementation. The following prompt is given to the LLM judge.

#### 4.4.3.1. Judge LLM Prompt

```
Given the following query and output,
evaluate the relevance and quality
of the RAG based on these criteria:

1. Functionality: Does the output implement
the required functionality? (1-5)
2. Readability: Is the code readable
and maintainable? (1-5)
```

```
3. Best Practices: Does the output adhere to common
coding standards and best practices? (1-5)
4. Performance: Is the code efficient
and optimized for performance? (1-5)
5. Robustness: Does the code handle edge cases
and potential errors gracefully? (1-5)

Provide a score for each criterion (1-5)
and a brief justification for your scores.

Query: \${query}

Output: \${impl.content}

Example format:
Functionality: 5
Justification: The code clearly implements
the required functionality.
Readability: 4
Justification: The code is mostly readable
but lacks some comments.
Best Practices: 4
Justification: The code follows common practices
but could be improved with better naming conventions.
Performance: 3
Justification: The code is efficient
but could be optimized further.
Robustness: 3
Justification: The code handles basic edge cases
but lacks extensive error handling.
```

# 5. Experiments, Results, and Analysis

This chapter presents our experimental study's primary observations and findings, comparing code generation with and without LASSO as a Retrieval-Augmented Generation (RAG) service. We then move on to the results, analysis, and discuss implications on software engineering and further developments and limitations.

## 5.1. Experimental Setup

In this section we explain the experiment setup and configuration, go over the dataset and coding problems and we talk about our used AI models and prompt design.

### 5.1.1. Hardware and Software Configuration

Our experimental setup consists of a machine configured as follows:

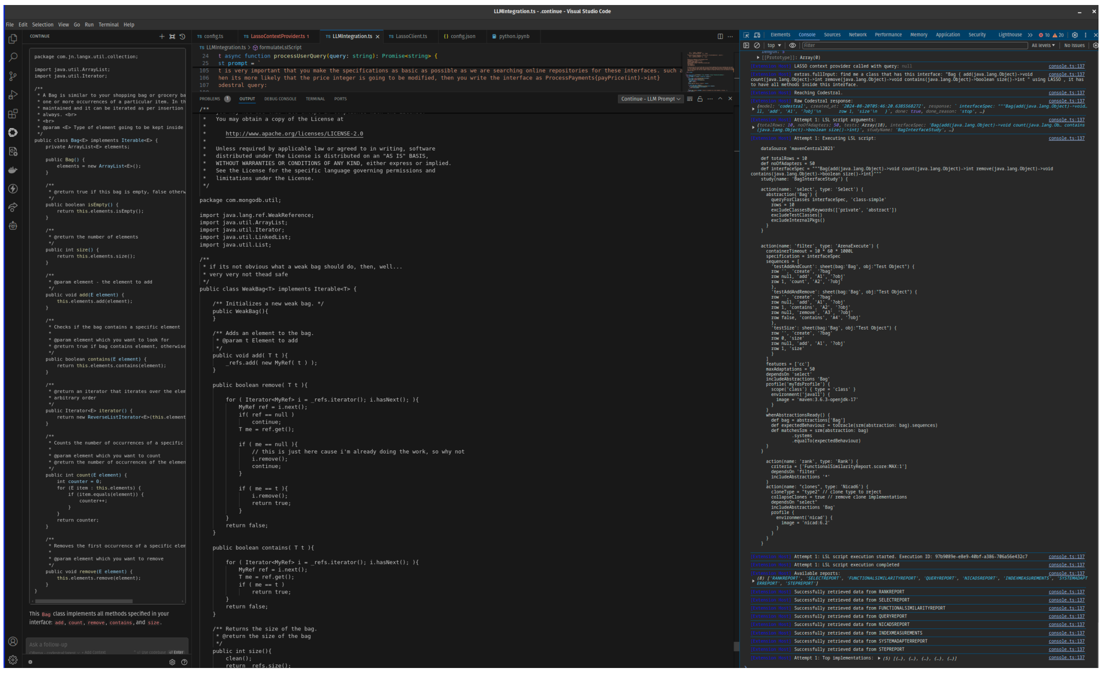
- **CPU**: Intel Core i9-13900HX
- **GPU**: 16 GB of GPU memory
- **RAM**: 32GB DDR4

It is important to note that this setup was also used to host Solr, Nexus, and LASSO as described in the LASSO repository [1]. and ran Visual Studio code(version 1.93), continue.dev plugin (0.8.49), AI models(more on that later), and Ollama(0.3.9). Only the indexed "MavenCentral2023" was accessed from the University of Mannheim.

Here is an example of running a prompt. on the right side we can see what is happening behind the scene and how LSL script is being formulated sent to LASSO, retrievals being judged and justifications for the judgement, how the results are retrieved, in the lower middle we can see how the prompts are formulated with

[1]https://github.com/SoftwareObservatorium/lasso

RAG and are being sent to the AI model, on the left side we see the response of the LLM with the RAG. (Recent continue.dev updates enable showing directly used RAG under the chat window).



```
package com.jn.langx.util.collection;

import java.util.ArrayList;
import java.util.Iterator;

/**
 * A Bag is similar to your shopping bag or grocery ba
 * one or more occurrences of a particular item. In th
 * maintained and it can be iterated as per insertion
 * always. <br>
 * <br>
 * @param <E> Type of element going to be kept inside
 */
public class Bag<E> implements Iterable<E> {
    private ArrayList<E> elements;

    public Bag() {
        elements = new ArrayList<E>();
    }

    /**
     * @return true if this bag is empty, false otherw
     */
    public boolean isEmpty() {
        return this.elements.isEmpty();
    }

    /**
     * @return the number of elements
     */
    public int size() {
        return this.elements.size();
    }

    /**
     * @param element - the element to add
     */
    public void add(E element) {
        this.elements.add(element);
    }

    /**
     * Checks if the bag contains a specific element
     *
     * @param element which you want to look for
     * @return true if bag contains element, otherwise
     */
    public boolean contains(E element) {
        return this.elements.contains(element);
    }

    /**
     * @return an iterator that iterates over the elem
     * arbitrary order
     */
    public Iterator<E> iterator() {
        return new ReverseListIterator<E>(this.element
    }

    /**
     * Counts the number of occurrences of a specific
     *
     * @param element which you want to count
     * @return the number of occurrences of the elemen
     */
    public int count(E element) {
        int counter = 0;
        for (E item : this.elements) {
            if (item.equals(element)) {
                counter++;
            }
        }
        return counter;
    }

    /**
     * Removes the first occurrence of a specific elem
     *
     * @param element which you want to remove
     */
    public void remove(E element) {
        this.elements.remove(element);
    }
}
```

This Bag class implements all methods specified in your interface: add, count, remove, contains, and size.

```
t async function processUserQuery(query: string): Promise<string> {
st prompt = `
t is very important that you make the specifications as basic as possible as we are searching online repositories for these interfaces. such a
hen its more likely that the price integer is going to be modified, then you write the interface as ProcessPayments{payPrice(int)->int}
odestral query:
```

```
/**
 *   You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 *   Unless required by applicable law or agreed to in writing, software
 *   distributed under the License is distributed on an "AS IS" BASIS,
 *   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 *   See the License for the specific language governing permissions and
 *   limitations under the License.
 */

package com.mongodb.util;

import java.lang.ref.WeakReference;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;

/**
 * if its not obvious what a weak bag should do, then, well...
 * very very not thead safe
 */
public class WeakBag<T> implements Iterable<T> {

    /** Initializes a new weak bag. */
    public WeakBag(){
    }

    /** Adds an element to the bag.
     * @param t Element to add
     */
    public void add( T t ){
        _refs.add( new MyRef( t ) );
    }

    public boolean remove( T t ){

        for ( Iterator<MyRef> i = _refs.iterator(); i.hasNext(); ){
            MyRef ref = i.next();
            if( ref == null )
                continue;
            T me = ref.get();

            if ( me == null ){
                // this is just here cause i'm already doing the work, so why not
                i.remove();
                continue;
            }

            if ( me == t ){
                i.remove();
                return true;
            }
        }
        return false;
    }

    public boolean contains( T t ){

        for ( Iterator<MyRef> i = _refs.iterator(); i.hasNext(); ){
            MyRef ref = i.next();
            T me = ref.get();
            if ( me == t )
                return true;
        }
        return false;
    }

    /** Returns the size of the bag.
     * @return the size of the bag
     */
    public int size(){
        clean();
        return _refs.size();
```

```
  ▸ [[Prototype]]: Array(0)
[Extension Host] LASSO context provider called with query: null                              console.ts:137
[Extension Host] extras.fullInput: find me a class that has this interface: "Bag { add(java.lang.Object)->void   console.ts:137
count(java.lang.Object)->int remove(java.lang.Object)->void contains(java.lang.Object)->boolean size()->int " using LASSO , it has
to have all methods inside this interface.
[Extension Host] Reaching Codestral.                                                         console.ts:137
[Extension Host] Raw Codestral response:                                                     console.ts:137
 ▸ {model: 'codestral', created_at: '2024-08-20T05:46:20.638556827Z', response: ` interfaceSpec: """Bag(add(java.lang.Object)->void…
   11, 'add', 'A1', '?obj'\n        row 1, 'size'\n    }`, done: true, done_reason: 'stop', …}
[Extension Host] Attempt 1: LSL script arguments:                                            console.ts:137
 ▸ {totalRows: 10, noOfAdapters: 50, tests: Array(18), interfaceSpec: 'Bag{add(java.lang.Object)->void count(java.lang.Ob… contains
   (java.lang.Object)->boolean size()->int}', studyName: 'BagInterfaceStudy', …}
[Extension Host] Attempt 1: Executing LSL script:                                            console.ts:137

    dataSource 'mavenCentral2023'

    def totalRows = 10
    def noOfAdapters = 50
    def interfaceSpec = """Bag{add(java.lang.Object)->void count(java.lang.Object)->int remove(java.lang.Object)->void
contains(java.lang.Object)->boolean size()->int}"""
    study(name: 'BagInterfaceStudy') {

    action(name: 'select', type: 'Select') {
      abstraction('Bag') {
        queryForClasses interfaceSpec, 'class-simple'
        rows = 10
        excludeClassesByKeywords(['private', 'abstract'])
        excludeTestClasses()
        excludeInternalPkgs()
      }
    }


    action(name: 'filter', type: 'ArenaExecute') {
      containerTimeout = 10 * 60 * 1000L
      specification = interfaceSpec
      sequences = [
        'testAddAndCount': sheet(bag:'Bag', obj:"Test Object") {
        row '', 'create', '?bag'
        row null, 'add', 'A1', '?obj'
        row 1, 'count', 'A2', '?obj'
        },
        'testAddAndRemove': sheet(bag:'Bag', obj:"Test Object") {
        row '', 'create', '?bag'
        row null, 'add', 'A1', '?obj'
        row 1, 'contains', 'A2', '?obj'
        row null, 'remove', 'A3', '?obj'
        row false, 'contains', 'A4', '?obj'
        },
        'testSize': sheet(bag:'Bag', obj:"Test Object") {
        row '', 'create', '?bag'
        row 0, 'size'
        row null, 'add', 'A1', '?obj'
        row 1, 'size'
        }
      ]
      features = ['cc']
      maxAdaptations = 50
      dependsOn 'select'
      includeAbstractions 'Bag'
      profile('myTdsProfile') {
        scope('class') { type = 'class' }
        environment('java11') {
          image = 'maven:3.6.3-openjdk-17'
        }
      }
      whenAbstractionsReady() {
        def bag = abstractions['Bag']
        def expectedBehaviour = toOracle(srm(abstraction: bag).sequences)
        def matchesSrm = srm(abstraction: bag)
                .systems
                .equalTo(expectedBehaviour)
      }
    }

      action(name: 'rank', type: 'Rank') {
        criteria = ['FunctionalSimilarityReport.score:MAX:1']
        dependsOn 'filter'
        includeAbstractions '*'
      }
      action(name: "clones", type: 'Nicad6') {
        cloneType = "type2" // clone type to reject
        collapseClones = true // remove clone implementations
        dependsOn "select"
        includeAbstractions 'Bag'
        profile {
          environment('nicad') {
            image = 'nicad:6.2'
          }
        }
      }
    }

[Extension Host] Attempt 1: LSL script execution started. Execution ID: 97b9089e-e8e9-40bf-a386-706a56e432c7   console.ts:137
[Extension Host] Attempt 1: LSL script execution completed                                   console.ts:137
[Extension Host] Available reports:                                                          console.ts:137
 ▸ (8) ['RANKREPORT', 'SELECTREPORT', 'FUNCTIONALSIMILARITYREPORT', 'QUERYREPORT', 'NICAD5REPORT', 'INDEXMEASUREMENTS', 'SYSTEMADAPT
   ERREPORT', 'STEPREPORT']
[Extension Host] Successfully retrieved data from RANKREPORT                                 console.ts:137
[Extension Host] Successfully retrieved data from SELECTREPORT                               console.ts:137
[Extension Host] Successfully retrieved data from FUNCTIONALSIMILARITYREPORT                 console.ts:137
[Extension Host] Successfully retrieved data from QUERYREPORT                                console.ts:137
[Extension Host] Successfully retrieved data from NICAD5REPORT                               console.ts:137
[Extension Host] Successfully retrieved data from INDEXMEASUREMENTS                          console.ts:137
[Extension Host] Successfully retrieved data from SYSTEMADAPTERREPORT                        console.ts:137
[Extension Host] Successfully retrieved data from STEPREPORT                                 console.ts:137
[Extension Host] Attempt 1: Top implementations:  ▸ (5) [{…}, {…}, {…}, {…}, {…}]            console.ts:137
```

Figure 5.1.: Code Generation with LASSO RAG

### 5.1.2. Dataset and Coding Problems

This work replicates the dataset and coding problems used by Kessel et al. [44]. These problems cover a range of standard programming tasks and algorithms, which provide a benchmark by comparing the performance of our approach relative to documented results. Detailed results and prompts are included in the Appendix for reproducibility.

| **Name** | **Description** |
|---|---|
| Queue | First in, first out (FIFO) structure. |
| FromJson | Parses JSON string into a Map. |
| ToJson | Serializes a Map to a JSON string. |
| B64Encode | Encodes strings to Base64 with padding. |
| B64Decode | Decodes Base64 strings with padding. |
| Fraction | Math Fraction(int, int) with methods asDouble(), getDenominator(), getNumerator(). |
| Sha256 | SHA256 hashing implementation. |
| Matrix | Mathematical matrix data structure. |
| HtmlSanitizer | Cleans untrusted HTML fragments. |
| MultiMap | Maps keys to multiple values (key-list pairs). |
| Bag | Enforces unique elements in a bag structure. |
| NGramDist | Distance metric using bi-grams. |
| NGramGen | Generates word-based bi-grams. |
| FilenameExt | Extracts file extension from java.io.File. |
| DAG | DirectedAcyclicGraph with addNode(), addEdge(), getOutgoingEdges(). |
| MinHeap | Min-heap structure with basic operations. |
| MovingAverage | Calculates rolling average over a window. |
| Cosine | Cosine similarity between string vectors. |
| TreeNode | Node in a tree structure. |
| Stack | LIFO stack with push, pop, peek, size operations. |

Table 5.1.: Dataset Summary

### 5.1.3. LLM Models

The coding problems mentioned above were tested in regard to DeepSeek-Coder-V2 and for Qwen2.5-V2. Tests with different models such as Llama3.1 (8B) or NexusRaven-V2 were discontinued because of high failure rates. A similar case to what we experienced with codestral as it had issues with formulating the LSL scripts, we expand more in the limitations section.

### 5.1.4. Prompt Design

The following prompt was used for all inputs:

> "Implement a Java class that satisfies the following interface:
>
> [Interface Specification]

Your implementation should pass the following test cases:

[Test Case]

Provide only the Java code for the class implementation, without any additional explanations or comments. ”

For the implementations with LASSO, we just had to add ”@LASSO” to the end or start of the prompt and in order not to confuse the script LLM about LASSO, we manually, through regex, extract it inside LassoContextProvider.

## 5.2. Experimental Methodology

### 5.2.1. Procedure

For each trial, a new chat window was created, and history was cleared to isolate performance and prevent context carryover. This ensures that nothing interferes with the other results, providing security about the accuracy of our experiment. Experiments were conducted twice: once using DeepSeek-Coder-v2 for all roles, and once with Qwen-2.5-Coder. we give in each experiment all roles to the respective model.

### 5.2.2. Evaluation Criteria

Most implementations, with or without LASSO, performed considerably well. This can be attributed to advancements in Language Models (LLMs) and the textbook nature of the problems.

However we decide to examine each implementation pair and note their differences, as their differences were not easy to describe as just, pass tests or does not.

- **Structural Differences**: We examined implementation pairs to note differences that impact code reuse, such as extra checks, error handling, manual custom logic, libraries used, and additional capabilities.
- **Test Case Success**: Implementations were evaluated on their ability to pass the provided test cases provided by the sequences by Kessel et al [44]. However there are undecided cases such as, the `Bag` implementation revealed

that duplicates were not allowed, a nuance often missed without LASSO. The argument is, providing the test sequence was maybe a not sufficient way to signalize uniqueness of items is a must.

- **Complexity and Overengineering**: We assessed whether LASSO introduced unnecessary complexity or overcomplicated solutions for simple tasks.

## 5.3. Results

A significant observation was the difference between how the chosen models for this integration, such as Qwen2.5-Coder, performed in the different roles. As an LLM judge Qwen gave way lower scores to RAG than DeepSeek-Coder-V2. This has led to a considerable reduction in given RAG and improved results. (However in one case, it did not reject the manual Base64 encoding and did it manually, as it accepted that RAG implementation) As discussed earlier, RAG is a double-edged sword, and we can definitely recognize that throughout our experiments.

### 5.3.1. Qwen2.5-Coder Experiment:

| Interface | Without LASSO | With LASSO |
|---|---|---|
| **JSON Parsing** | JSON parsing hardcoded values; not robust. | Utilize Jackson library |
| **Bag** | no item uniqueness. | `HashSet` to enforce uniqueness and track counts. |
| **Base64 Decode** | Java's built-in `Base64` decoder | custom Base64 decoder manual logic; complex |
| **Cosine Similarity** | similarity based on word overlap; wrong implementation | simplistic implementation; wrong implementation |
| **Directed Graph** | basic graph functionality, less type-safe. | type safety, unmodifiable views, better error handling. |
| **Filename Extract** | basic string operations without null checks | checks for null values and file validity |
| **Fraction** | Simple implementation; lacks validation for denominator zero; allows invalid fractions. | Uses immutable fields; includes validation(when asked for output only) |
| **Moving Average** | lacks validation for period; may cause errors. | Includes validation for positive period |
| **HTML Sanitize** | simple regex; potentially insecure. | allowing only specific HTML tags; improves security, prevents script injections. |
| **Matrix** | No index bounds check | bounds and dimension checks |
| **MinHeap** | efficient standard implementation. | custom array-based heap; increases complexity |
| **JSON Serialize** | Manually builds JSON string; may not handle special characters or nested structures correctly. | minimal differences; no improvement. |
| **MultiMap** | empty collection when key not present | `null` when key not present |
| **N-Gram Distance** | Simplistic algorithm; limited accuracy in distance calculation. | sophisticated algorithm; enhanced accurate results. |
| **N-Gram Generate** | Standard implementation | Similar implementation; no improvement. |
| **Queue** | `LinkedList`; efficient O(1) dequeue operations. | `ArrayList`; inefficient O(n) dequeue operations |
| **SHA-256 Hashing** | manual byte-to-hex conversion | `BigInteger` for hexadecimal conversion, better |
| **Stack** | `IllegalStateException` when empty. | `EmptyStackException` better semantics. |
| **TreeNode** | Lazy initialization of `children` list; may cause `NullPointerException`. | Eager initialization of `children`, more robust. |

Table 5.2.: Comparison of Implementations with and without LASSO using Qwen2.5-Coder

### 5.3.2. DeepSeek-Coder-V2 Experiment:

| **Interface** | **Without LASSO** | **With LASSO** |
|---|---|---|
| **Bag** | Uses `ArrayList`; inefficient counting/removal | Uses `HashMap`; efficient counting/removal |
| **Base64 Decoding** | Uses built-in `Base64` decoder | Custom decoder; complex, error-prone |
| **Base64 Encoding** | Uses built-in `Base64` encoder | Custom encoder; unnecessary complexity |
| **Cosine Similarity** | Incorrect implementation; errors | Uses Levenshtein distance; incorrect |
| **Directed Graph** | No generics; less type-safe | Uses generics; better type safety |
| **Filename Extract** | May mishandle directories | Returns `null` for directories |
| **Fraction** | Includes zero denominator check | Lacks denominator check; risk of division by zero |
| **JSON Parsing** | Uses `org.json`; standard parsing | Uses Jackson library; more robust |
| **Moving Average** | Uses `Queue`; efficient | Uses `List`; less efficient |
| **MultiMap** | Generic types; less type-safe | Specific types; better type safety |
| **Queue** | Uses `LinkedList`; efficient | Uses `ArrayList`; inefficient dequeues |
| **JSON Serialize** | Manually builds JSON; may miss edge cases | Uses `JSONObject`; robust |

Table 5.3.: Comparison of Implementations with and without LASSO using DeepSeek-Coder-v2

#### 5.3.2.1. Judge LLM Experiment

In the DeepSeek-Coder-v2 experiments, we generated three sets of implementations: without LASSO, with LASSO but without the Judge LLM, and with both LASSO and the Judge LLM.

| **Interface** | **With LASSO** | **With LASSO and Judge LLM** |
|---|---|---|
| **Base64 Encoding** | Custom encoder; unnecessary complexity | Reverts to built-in encoder |
| **Cosine Similarity** | Uses Levenshtein distance; incorrect | Corrects to proper cosine similarity |
| **Fraction** | Lacks denominator check; risk of division by zero | best implementation and error handling |
| **MinHeap** | Similar to without LASSO | Switches to array-based heap; better performance |
| **Moving Average** | Uses `List`; less efficient | Reverts to `Queue`; efficient |
| **MultiMap** | Generic types; less type-safe | Specific types; better type safety |
| **Queue** | Uses `ArrayList`; inefficient dequeues | Custom `CircularList`; potential gains |
| **JSON Serialization** | Uses `JSONObject`; robust | Uses Jackson `ObjectMapper`; more robust |

Table 5.4.: Comparison of Implementations with LASSO and with LASSO and Judge LLM using DeepSeek-Coder-v2

## 5.4. Results Analysis

The results of our experiments provide valuable insights into the impact of using LASSO as a Retrieval-Augmented Generation (RAG) service in code generation. In this section, we interpret these results, discuss the effectiveness of LASSO and the Judge LLM, consider the implications for software engineering, and acknowledge the limitations and threats to the validity of our study.

### 5.4.1. Interpretation of Results

Our experiments demonstrate that the use of LASSO can both enhance and complicate code generation. The effectiveness of LASSO is closely tied to the retrieved RAG.

In the following, we observe the results more closely and examine their implications.

#### 5.4.1.1. Qwen2.5-Coder Evaluation

Using LASSO introduced human best practices, improved robustness (e.g., in `Fraction` and `Filename`), and enhanced capabilities (e.g., dynamic resizing in

`MinHeap`). It caught small details such as enforcing uniqueness in `Bag`, using `BigInteger` in `Sha256`, and better HTML sanitization. However, it also introduced more complexity, tended to reinvent the wheel (as in `Base64`), and sometimes provided no advantages (as in `ToJson`) or overcomplicated solutions with incorrect outputs (as in `Cosine Similarity`).

#### 5.4.1.2. DeepSeek-Coder-V2 Evaluation

While it noticed small details (e.g., uniqueness in `Bag`, zero denominator check in `Fraction`), it sometimes generated entirely different requests due to irrelevant RAG (e.g., using Levenshtein distance instead of cosine similarity), leading to overcomplicated solutions for simple tasks, without the Judge LLM, the LASSO-assisted code sometimes lacked critical validation.

#### 5.4.1.3. Judge LLM Evaluation

- **Rejecting Incorrect Implementations:** In the `Cosine Similarity` task, the initial LASSO-assisted implementation incorrectly used the Levenshtein distance algorithm based on false RAG. The Judge LLM rejected this Retrieval.
- **Rejecting Overcomplicated Solutions:** For the `Base64 Encoding` task, the without LLM judge LASSO-assisted generated code provided an unnecessarily complex custom encoder. With the use of Judge LLM, it retrieved the same implementations but rejected them, which led to the recommendation to revert to Java's built-in `Base64` encoder.
- **Enhancing Robustness:** In the `Fraction` class, the LASSO-assisted implementation lacked a check for a zero denominator in the constructor, although it did not allow the division by zero errors, it did enable the creation of said Fraction. The Judge LLM included only retrievals that passed this test.
- **Enhance Performance:** In the `MinHeap` implementation, the Judge LLM rejected the list-based heap and recommended an array-based heap. Arrays are faster to access, but it highly depends on the use case; since both implementations passed all tests, array-based is more performant.

#### 5.4.1.4. Overall Evaluation

**The Good**

- **Enhanced Code Quality in Complex Scenarios**: LASSO proved invaluable in tasks where unique implementation details are critical and not explicitly specified. For instance, in the `Bag` implementation, LASSO-assisted models correctly inferred the requirement to disallow duplicates, leading to accurate implementations that passed all test cases.
- **Improved Robustness and Best Practices**: LASSO-assisted code often incorporated human best practices, including input validation, error handling, and appropriate data structures. Examples include:
  - **Fraction**: zero denominator check to prevent division by zero.
  - **Filename**: Null checks and file validity checks for safer filename extraction.
  - **HtmlSanitizer**: By only allowing specific HTML tags, it reduces security risks.
  - **Stack**: Throwing specific exceptions like offers better error semantics.
- **Utilization of Robust Libraries**: The LASSO-assisted implementations frequently leveraged well-established libraries, enhancing reliability and efficiency. Notable examples include:
  - **JSON Parsing**: Using the Jackson library for robust JSON parsing instead of manual or hardcoded methods.
  - **SHA-256 Hashing**: Utilizing `BigInteger` for hexadecimal conversions, resulting in more efficient and cleaner code.

  correctly handling edge cases in `Matrix` operations.
- **Improved Performance**: In specific tasks, LASSO-assisted code provided performance enhancements by choosing more efficient data structures or algorithms, such as:
  - **NGram Distance**: Implementing a sophisticated algorithm for more accurate distance calculation.
  - **TreeNode**: Eager initialization of children lists to avoid unnecessary checks and potential errors.

### The Bad

- **Unnecessary Complexity**: In some cases, LASSO-assisted code overcomplicated solutions by manually implementing functionality, which could have been achieved using built-in libraries. Examples include:
  - **Base64 Encoding/Decoding**: Providing custom implementations instead of utilizing Java's built-in `Base64` class, leading to increased complexity and potential for errors.
  - **MinHeap**: Implementing custom array-based heaps when standard libraries could suffice.
- **Overengineering Simple Tasks**: The use of advanced algorithms or additional features was sometimes unnecessary for the task at hand. For instance, in the `NGram Distance` implementation, a more sophisticated algorithm was used when a simpler approach would suffice.
- **Assumptions of Code Availability**: LASSO-assisted code occasionally assumed the availability of certain libraries or dependencies not part of the standard codebase, leading to potential integration challenges and additional effort to include those dependencies.
- **Inefficient Data Structure Choices**: In some implementations, such as the `Queue`, LASSO-assisted code used less efficient data structures like `ArrayList`, resulting in poorer performance compared to the non-LASSO versions that used a more appropriate structure.
- **Lack of Difference in Some Cases:** For straightforward tasks such as NGram Generate or ToJson, LASSO's RAG did not contribute any meaningful improvements.

### The Ugly - Areas for Further Improvement

- **Propagation of Incorrect Implementations:** LASSO-assisted models sometimes produced incorrect solutions due to irrelevant or inappropriate code retrievals. A notable example is the Cosine Similarity task, where the LASSO-assisted implementation incorrectly used the Levenshtein distance algorithm instead of properly computing cosine similarity. (Before implementing judge LLM)

- **Dependence on Retrieval Quality:** LASSO's effectiveness heavily depends on the retrieved code's relevance and quality. Bad RAG retrievals can actively degrade its quality and lead to incorrect or inefficient solutions.
- **Complexity in Maintenance:** Overcomplicated solutions and unnecessary dependency imports can increase technical debts, negating the benefits of time saved in software reuse.
- **Risk of Confusion and Misinterpretation:** The LLMs may misinterpret the RAG context provided by LASSO, especially if the retrieved code is not directly relevant to the task. It could confuse and misalign the developer's desired functionality by generating many complex methods that are undesired.
- **Lack of Critical Validation:** In some cases, both with LASSO and without LASSO implementations needed essential validations. For example, the Fraction implementations allows for the creation of a fraction, of which denominator is zero but does not allow retrieving it, instead of not allowing object creation.

### 5.4.2. Limitations and Threats to Validity

Our study has several limitations that should be acknowledged:

- **Model Dependency**: The results are highly dependent on the capabilities of the underlying LLMs. Different models may produce varying outcomes, and advancements in LLM technology could alter the effectiveness of approaches like LASSO.
- **Dataset Scope**: The size of the coding problems used may not fully represent the complexities encountered in real-world software development. This limits the generalizability of our findings.
- **Prompt Design**: The prompt used in our experiments may have influenced the results, especially in the case of uniqueness of items in the bag implementation. Different prompt designs or additional context could yield different generations.
- **Evaluation Metrics**: Our evaluation focused on structural differences and qualitative assessment rather than quantitative metrics such as code execu-

tion time or memory usage, which might affect performance in real world usage, this highlights the need to incorporate LASSO runtime information into the given RAG and conduct larger scale experiments.

- **Reproducibility**: The dynamic nature of LLM outputs and potential updates to models and retrieved RAG may affect the reproducibility of our results.
- **LASSO Limitations**: As with any code retrieval system, LASSO's effectiveness depends on the quality and relevance of the code in its repository. Inappropriate or outdated code could negatively impact the generated solutions, although manual implementations of Base64 might be the best performative, it may not be the best practice.
- **Integration Challenges**: Using LASSO as RAG may lead to code that assumes the availability of certain libraries or methods, which may not exist in the target codebase, which could require efforts to include those dependencies.

# 6. Discussion

In this chapter, we go over more aspects of the results and examine implications on a larger scale:

### 6.0.1. Performance Testing Results

The experimental results demonstrated that when provided with appropriate test cases, the code snippets generated by the LLMs could pass the majority of those tests, particularly for problems with reduced complexity. However, when dealing with implementations that required checking for the existence of specific objects or files (Filename extraction problem), the LLMs had to make assumptions based on unknown parameters, resulting in unverified guessed solutions. In contrast, LASSO solves this issue by incorporating and executing these tests. The experimental results also suggest that as problem complexity increases, the utility of RAG in assisting LLMs in generating applicable code becomes more evident.

### 6.0.2. Ease of Use and Integration

Rebuilding and testing different approaches within the proposed system is straightforward and can be accomplished within minutes. However, it is necessary to host LASSO and its associated Solr and Nexus components as well as a hosted Ollama service with the desired models. The continue.dev Main LLM could be replaced with an alternative model, such as a proprietary model provider accessed through an API key or other open-source models using different hosting technologies. This increases flexibility in the process. While the remaining LLM roles are hosted using Ollama. They can also be replaced with another model hosting technology, provided it offers a JavaScript or Python library to facilitate LLM calls, but that requires technical code changes.

### 6.0.3. Test Coverage and Correctness

The LLMs were relied upon to generate tests in the experiments conducted without providing tests. In these scenarios, different models generated different tests; specifically, Nexusraven-V2 and Llama 3.1 performed comparatively worse than coding models, which does not make them necessarily bad in generating LSL scripts, but it does make them inadequate in conveying contexts and requirements into tests for the LSL scripts, leading to their exclusion from the process. Codestral also generated a false test case in the LSL input despite being provided with the correct one, resulting in its exclusion. Deepseek-coder-v2 and Qwen2.5-coder exhibited the best performance and were consequently adapted for the study. However, there could be an improved process where test generation is done separately from generating the components for the LSL scripts.

### 6.0.4. Implications of the Experimental Results

Although it is a good practice, many interpretations suggest using libraries, potentially offering improved solutions. The underlying workings of these libraries are uncertain. According to our empirical evidence, many LASSO implementations are designed to handle more significant amounts of input. In contrast, implementations without LASSO tend to be more straightforward and generic.

## 6.1. Limitations

As with any code search engine, finding the perfect implementation using LASSO can be a hit or miss, just like any search engine. In some searches, we obtained better results than in others. We argue that our setup is limited and that expanding the number of searches and the number of adapters could enable more code to be considered. It may take several attempts to find better implementations. In this section, we discuss various limitations facing this approach, including those related to the AI models, LASSO itself, and the stability of the continue.dev plugin.

### 6.1.1. AI Limitations

Some persistent challenges in AI models are hallucinations, logical fallacies, and incorrect code generation. These are inherent limitations of current large language models (LLMs), as they tend to replicate patterns they have learned and may not fully understand or verify the correctness of the generated code. Throughout our observations, we cannot definitively prioritize one LLM role over another, as each internal LLM plays an equally important part ,whether in generating tests, experimenting with different scripting approaches or evaluating the relevance and quality of retrieved RAG. In future improvements (discussed in the next section), models that better leverage the potential in LASSO Script Language (LSL) scripts, such as function-calling LLMs, could be beneficial. These models perform well even with fewer parameters; for example, NexusRaven, with only 13 billion parameters, achieved results comparable to GPT-4 in function calling [83]. However, there is significant potential in integrating LASSO actions, such as the Evosuite action, to generate tests or intermediate steps focused on producing tests that best satisfy the user's natural language input. This is particularly important when the user wants to test edge cases or specific use cases. We advise to handle test generation within its subsystem to ensure overall correctness since ensuring correct RAG is crucial, as LLMs can hallucinate, and hallucinations in test generation can have significant consequences. We also had to modify prompts for each different model, and higher-quality results might have been achievable with other models given more time and effort in prompt engineering and design. However, we opted for the better-performing models that were readily available.

### 6.1.2. LASSO Limitations

LASSO's greatest challenge, in terms of our work, is defining specific LSL scripts that retrieve what can help the developer the most. This requires an understanding at a higher level rather than a granular level. This approach allows for observing multiple variations of the desired methods and generating dynamic analyses, leading to better-informed decisions based on statistical observations. In our experiments, we noticed that using LASSO and allowing LLMs to decide on the names of interfaces yields different results, leading to inconsistencies.

There could be potential for further research and codebase representations, such as graphs, which can be used to enhance the naming strategy.

### 6.1.3. continue.dev Limitations

Continue.dev, like any developing open-source software, occasionally faces development issues. While it has an engaged community that assists with questions, it suffers from bugs that may be introduced in each version. During our study, we encountered such an issue and had to revert to an older version to rerun the scripts. Another limitation is that we connect the backend as part of the `config.js` files in the home directory. A more robust approach would involve implementing additional security measures and better error handling. Additionally, the only way to get the code to recompile involves completely closing the integrated development environment (IDE) and plugin and restarting them, which is inefficient and disrupts the development workflow.

## 6.2. Improvements

The quality of the results depends significantly on the underlying AI models, as they are critical factors in determining the context window length and reasoning capabilities. However, our multi-stage LLM architecture allows us to assign different specialized models for different tasks. For example, we can use a fine-tuned model for generating LSL scripts, another for test case generation, and another for evaluating code quality. This flexibility allows us to optimize performance and make this setup accessible to many developers. In an enterprise setting, LASSO could retrieve previously deployed code from private repositories and serve as an adapter between projects requiring similar yet distinctive adaptations. This section discusses various available improvements, from enhancements to the LLM models and calls to the framework, and explores different yet similar approaches.

### 6.2.1. improvements on the AI backend

Several ways exist to extend functionality and incorporate more of LASSO's knowledge into the Judge LLM evaluation process, especially regarding runtime considerations. For instance, we could implement mechanisms to reject implementations that exceed average runtime limits specified by metrics generated from large-scale database executions. Another approach is incorporating additional actions from LASSO, such as enabling test generation for better code coverage metrics, integrating GitHub pull actions for private repository exploration and ensuring that the latest retrievals are up to date. Expanding the range of useable LASSO actions can enhance the depth and quality of code retrieval and evaluation.

### 6.2.2. Code Translation

One of the capabilities of LLMs is code translation [55], which they can perform quite efficiently, using the right models. This unlocks the potential of using LASSO to retrieve java methods and then translate them into other languages, where the same logic can be translated and applied, thereby broadening the utility of LASSO across multiple programming languages.

### 6.2.3. Alternative Technologies and Models

Exploring alternative technologies and models is crucial to further enhance LASSO's performance and reliability. Developing more advanced, "morescient" LLMs that incorporate static and dynamic knowledge of software behaviour, as highlighted by Kessel et al [45], could improve the quality of generated code and the effectiveness of LASSO as RAG.

### 6.2.4. Integration with Code Vulnerability Detection Tools:

AI-generated code and open-source code could introduce vulnerabilities and security risks into the codebase. Therefore, we recommend integrating code vulnerability detection tools such as SonarQube[1] or WhiteSource[2] into this process or as a LASSO action to help avoid recommending code that suffers from security issues, which proactively identify vulnerabilities for safer adoption and lower technical debts.

[1]https://www.sonarqube.org/
[2]https://www.whitesourcesoftware.com/

# 7. Conclusion

## 7.1. Overview

This thesis has explored the integration of the Large-Scale Software Observatorium (LASSO) with generative AI models to advance software engineering practices, particularly in pragmatic software reuse. A more efficient and reliable AI-assisted development environment could be created by combining LASSO's analytical observations with context-aware (LLMs). The primary contributions of this research include:

- **Integration Architecture:** A multi-step multi-LLM role process was designed to merge LASSO with generative AI models via the Continue.dev platform. This setup enables contextually relevant code suggestions by leveraging LASSO as a context provider.
- **Experimental Evaluation:** The system's performance was tested across two sets of coding problems, utilizing models such as Qwen2.5-Coder and DeepSeek-Coder-V2. Results showed improvements and limitations in the relevance and accuracy of code recommendations when LASSO's RAG features were applied.
- **Dynamic Evaluation:** A specialized Judge LLM was introduced to further refine these suggestions, ensuring higher quality, context-appropriate code was provided as RAG to the main AI assistant within the IDE.

As highlighted throughout this thesis and demonstrated in the experiments, RAG plays a crucial role in improving generated content quality and mitigating LLMs inherent issues. We aimed at understanding how, when, and what type of RAG is the most beneficial development tasks. Overall, although there are a significant number of limitations and areas where this integration falls short and does not produce better results, we argue that the optimal methodology for this integration is still uncertain and we present this approach as an initial attempt to advance the field and evaluate potential performance gains. We hope this constitutes a further step towards more efficient pragmatic software reuse, taking advantage of LLMs context reasoning to adapt and integrate code into different environments and an advantage of LASSOs vast capabilities, to help refine and define relevant RAG for code generation tasks in software engineering.

### 7.1.1. Future Work

This current integration utilizes only a subset of LASSO's capabilities. More helpful behavioural data about implementations, rankings based on other criteria, and the use of LASSO test generation features were not included. This offers an opportunity for further development. Further research should focus on including more of LASSO's actions and leveraging a broader range of LLMs for the different LLM roles to improve flexibility and performance. Investigating advanced prompt engineering techniques and real-time feedback mechanisms also leads to greater accuracy and reliability. However, it's important to note that conducting large-scale experiments in real-world development settings is essential to assess the system's effectiveness. Particularly by including natural language queries in the context of complex settings and improving the construction of multi-facet LSL scripts. With a well-developed RAG system, users can benefit from better implementations accompanied by interface, test-driven, and code-driven searches. The plugin developed in this research is designed to evolve with LASSO, and future work could build on this baseline by exploring additional actions and configuration changes that improve both usability and performance.

## 7.2. Final Remarks

This thesis makes a contribution to AI-assisted software engineering by demonstrating the potential of integrating LASSO's in-depth code analysis capabilities with generative AI models. The proposed framework lays the groundwork for future integrations that can enhance developer productivity, code quality, and the overall software development lifecycle as AI use in software engineering continues to evolve.

# Bibliography

[1] N.A. Aarti. „Generative Ai in Software Development : an Overview and Evaluation of Modern Coding Tools“. In: *International Journal For Multidisciplinary Research* 6.3 (2024). DOI: 10.36948/ijfmr.2024.v06i03.23271.

[2] Samuel Abedu, Ahmad Abdellatif, and Emad Shihab. „LLM-Based Chatbots for Mining Software Repositories: Challenges and Opportunities“. In: *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*. EASE '24. Salerno, Italy: Association for Computing Machinery, 2024, pp. 201–210. ISBN: 9798400717017. DOI: 10.1145/3661167.3661218. URL: https://doi.org/10.1145/3661167.3661218.

[3] Rabbia Ahmed, Sadaf Abdul Rauf, and Seemab Latif. „Leveraging Large Language Models and Prompt Settings for Context-Aware Financial Sentiment Analysis“. In: *2024 5th International Conference on Advancements in Computational Sciences (ICACS)*. 2024, pp. 1–9. DOI: 10.1109/ICACS60934.2024.10473283.

[4] Mistral AI. *Codestral: Revolutionizing Code Search and Analysis*. Accessed: September 26, 2024. Mistral AI. 2024. URL: https://mistral.ai/news/codestral/ (visited on Sept. 26, 2024).

[5] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. *Learning to Represent Programs with Graphs*. 2018. arXiv: 1711.00740 [cs.LG]. URL: https://arxiv.org/abs/1711.00740.

[6] Gautam B and Anupam Purwar. *Evaluating the Efficacy of Open-Source LLMs in Enterprise-Specific RAG Systems: A Comparative Study of Performance and Scalability*. 2024. arXiv: 2406.11424 [cs.IR]. URL: https://arxiv.org/abs/2406.11424.

[7] Sushil Bajracharya, Joel Ossher, and Cristina Lopes. „Sourcerer: An infrastructure for large-scale collection and analysis of open-source code“. In: *Science of Computer Programming* 79 (2014). Experimental Software and Toolkits (EST 4): A special issue of the Workshop on Academic Software Development Tools and Techniques (WASDeTT-3 2010), pp. 241–259. ISSN: 0167-6423. DOI: `https://doi.org/10.1016/j.scico.2012.04.008`. URL: `https://www.sciencedirect.com/science/article/pii/S016764231200072X`.

[8] Thoms Ball. „The concept of dynamic analysis“. In: *ACM SIGSOFT Software Engineering Notes* 24.6 (1999), pp. 216–234.

[9] José L Barros-Justo et al. „What software reuse benefits have been transferred to the industry? A systematic mapping study“. In: *Information and Software Technology* 103 (2018), pp. 1–21.

[10] Barry W Boehm and Victor R Basili. „Software defect reduction top 10 list“. In: *Computer* 34.1 (2001), pp. 135–137.

[11] Ali Borji. *A Categorical Archive of ChatGPT Failures.* 2023. arXiv: `2302.03494 [cs.CL]`. URL: `https://arxiv.org/abs/2302.03494`.

[12] Tilmann Bruckhaus. *RAG Does Not Work for Enterprises.* 2024. arXiv: `2406.04369 [cs.SE]`. URL: `https://arxiv.org/abs/2406.04369`.

[13] Cara Burgan, Josiah Kowalski, and Weidong Liao. „Developing a Retrieval Augmented Generation (RAG) Chatbot App Using Adaptive Large Language Models (LLM) and LangChain Framework“. In: *Proceedings of the West Virginia Academy of Science* 96.1 (Apr. 2024). DOI: `10.55632/pwvas.v96i1.1068`. URL: `https://pwvas.org/index.php/pwvas/article/view/1068`.

[14] Katherine Byers. „Amazon warns employees not to share confidential information with ChatGPT after seeing cases of its answers matching internal data“. In: *Business Insider* (Jan. 2023). URL: `https://www.businessinsider.com/amazon-chatgpt-openai-warns-employees-not-share-confidential-information-microsoft-2023-1` (visited on Sept. 22, 2024).

[15] Jose Cambronero et al. „When deep learning met code search“. In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2019. Tallinn, Estonia: Association for Computing Machinery, 2019, pp. 964–974. ISBN: 9781450355728. DOI: 10.1145/3338906.3340458. URL: https://doi.org/10.1145/3338906.3340458.

[16] Aline de Campos et al. *Some things never change: how far generative AI can really change software engineering practice*. 2024. arXiv: 2406.09725 [cs.SE]. URL: https://arxiv.org/abs/2406.09725.

[17] Chuheng Chang et al. „The path from task-specific to general purpose artificial intelligence for medical diagnostics: A bibliometric analysis“. In: *Computers in Biology and Medicine* 172 (2024), p. 108258. ISSN: 0010-4825. DOI: https://doi.org/10.1016/j.compbiomed.2024.108258. URL: https://www.sciencedirect.com/science/article/pii/S0010482524003421.

[18] Aakanksha Chowdhery et al. *PaLM: Scaling Language Modeling with Pathways*. 2022. arXiv: 2204.02311 [cs.CL].

[19] *Codase Code Search Engine*. http://www.codase.com/. Accessed: 2014-09-30.

[20] James R. Cordy. „NiCad: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization“. In: *Proceedings of the 19th IEEE International Conference on Program Comprehension* (2011), pp. 172–181. DOI: 10.1109/ICPC.2011.47.

[21] Mark Craddock. „HYDE: Revolutionising Search with Hypothetical Document Embeddings“. In: *Medium* (2023). https://medium.com/prompt-engineering/hyde-revolutionising-search-with-hypothetical-document-embeddings-3474df795af8.

[22] Florin Cuconasu et al. „The Power of Noise: Redefining Retrieval for RAG Systems“. In: *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR ’24. Washington DC, USA: Association for Computing Machinery, 2024, pp. 719–729. ISBN: 9798400704314. DOI: 10.1145/3626772.3657834. URL: https://doi.org/10.1145/3626772.3657834.

[23] Damai Dai et al. *DeepSeekMoE: Towards Ultimate Expert Specialization in Mixture-of-Experts Language Models*. 2024. arXiv: `2401.06066 [cs.CL]`. URL: `https://arxiv.org/abs/2401.06066`.

[24] Jacob Devlin et al. „BERT: Pre-training of deep bidirectional transformers for language understanding“. In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. 2019, pp. 4171–4186.

[25] Luca Di Grazia and Michael Pradel. „Code Search: A Survey of Techniques for Finding Code“. In: *ACM Computing Surveys* 55.11 (Feb. 2023), pp. 1–31. ISSN: 1557-7341. DOI: `10.1145/3565971`. URL: `http://dx.doi.org/10.1145/3565971`.

[26] Thomas Dohmke, Marco Iansiti, and Greg Richards. *Sea Change in Software Development: Economic and Productivity Analysis of the AI-Powered Developer Lifecycle*. 2023. arXiv: `2306.15033 [econ.GN]`.

[27] Paulo Finardi et al. *The Chronicles of RAG: The Retriever, the Chunk and the Generator*. 2024. arXiv: `2401.07883 [cs.LG]`. URL: `https://arxiv.org/abs/2401.07883`.

[28] Gartner. *Gartner 2024 Hype Cycle for Emerging Technologies Highlights Developer Productivity, Total Experience, AI and Security*. Gartner, Aug. 2024. URL: `https://www.gartner.com/en/newsroom/press-releases/2024-08-21-gartner-2024-hype-cycle-for-emerging-technologies-highlights-developer-productivity-total-experience-ai-and-security`.

[29] Gartner. *Gartner Says More Than 80% of Enterprises Will Have Used Generative AI APIs or Deployed Generative AI-Enabled Applications by 2026*. Oct. 2023. URL: `https://www.gartner.com/en/newsroom/press-releases/2023-10-11-gartner-says-more-than-80-percent-of-enterprises-will-have-used-generative-ai-apis-or-deployed-generative-ai-enabled-applications-by-2026`.

[30] GitHub Docs. *About Copilot Autofix for CodeQL Code Scanning*. GitHub. URL: `https://docs.github.com/en/code-security/code-scanning/`

managing-code-scanning-alerts/about-autofix-for-codeql-code-scanning.

[31] Pablo Gonzalez-de-Aledo, Pablo Sánchez, and Ralf Huuck. „An Approach to Static-Dynamic Software Analysis“. In: *International Conference on Software Engineering and Formal Methods*. Cham: Springer, 2015, pp. 225–240. DOI: 10.1007/978-3-319-29510-7_13.

[32] *Google Code Search*. https://en.wikipedia.org/wiki/Google_Code_Search. Accessed: 2023-06-16.

[33] Rahul Gupta et al. „DeepFix: Fixing common C language errors by deep learning“. In: *Thirty-Second AAAI Conference on Artificial Intelligence*. 2018.

[34] Jordan Hoffmann et al. *Training Compute-Optimal Large Language Models*. 2022. arXiv: 2203.15556 [cs.CL].

[35] Marc Hoffmann and Jesper Steen Møller. *JaCoCo: Java Code Coverage Library*. https://www.jacoco.org/. 2009.

[36] Oliver Hummel. *Semantic component retrieval in software engineering : Elektronische Ressource*. [Online]. Available: https://madoc.bib.uni-mannheim.de/1883/. 2008. URL: https://madoc.bib.uni-mannheim.de/1883/.

[37] Oliver Hummel et al. „Improving testing efficiency through component harvesting“. In: *Proc. Brazilian Workshop on Component Based Development*. 2006.

[38] Consortium for Information and Software Quality (CISQ). *The Cost of Poor Software Quality in the US: A 2020 Report*. Tech. rep. Consortium for Information and Software Quality (CISQ), 2020. URL: https://www.it-cisq.org/the-cost-of-poor-software-quality-in-the-us-a-2020-report/.

[39] Sarthak Jain et al. *LLM Agents Improve Semantic Code Search*. 2024. arXiv: 2408.11058 [cs.SE]. URL: https://arxiv.org/abs/2408.11058.

[40] Werner Janjic and Colin Atkinson. „Utilizing software reuse experience for automated test recommendation“. In: *2013 8th International Workshop on Automation of Software Test (AST)*. IEEE. 2013, pp. 100–106.

[41] Georgia M Kapitsaki. „Generative AI for Code Generation: Software Reuse Implications“. In: *International Conference on Software and Software Reuse*. Springer. 2024, pp. 37–47.

[42] Rabimba Karanjai, Lei Xudagger, and Weidong Shi. „SolMover: Feasibility of Using LLMs for Translating Smart Contracts“. In: *2024 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. 2024, pp. 1–3. DOI: 10.1109/ICBC59979.2024.10634392.

[43] Marcus Kessel. „LASSO – an observatorium for the dynamic selection, analysis and comparison of software“. PhD thesis. Mannheim: University of Mannheim, Feb. 2022. URL: https://madoc.bib.uni-mannheim.de/64107/.

[44] Marcus Kessel and Colin Atkinson. „Code search engines for the next generation“. In: *Journal of Systems and Software* 215 (2024), p. 112065. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2024.112065. URL: https://www.sciencedirect.com/science/article/pii/S0164121224001109.

[45] Marcus Kessel and Colin Atkinson. *Morescient GAI for Software Engineering*. 2024. arXiv: 2406.04710 [cs.SE]. URL: https://arxiv.org/abs/2406.04710.

[46] Matthias Kessel and Colin Atkinson. „Ranking software components for pragmatic reuse“. In: *2015 IEEE/ACM 6th International Workshop on Emerging Trends in Software Metrics*. IEEE. 2015, pp. 63–66.

[47] Matthias Kessel and Colin Atkinson. „Ranking software components for reuse based on non-functional properties“. In: *Information Systems Frontiers* 18.5 (2016), pp. 825–853.

[48] *Koders Code Search Engine*. http://www.koders.com/. Accessed: 2014-09-30.

[49] Charles Koutcheme et al. *Evaluating Language Models for Generating and Judging Programming Feedback*. 2024. arXiv: 2407.04873 [cs.AI]. URL: https://arxiv.org/abs/2407.04873.

[50] *Krugle Code Search Engine*. http://www.krugle.com/. Accessed: 2014-09-30.

[51] Xuan-Bach D Le, David Lo, and Claire Le Goues. „S2FIX: Deep learning-based approach for generating patches“. In: *arXiv preprint arXiv:1611.05144* (2016).

[52] Mu-Woong Lee, Seung-won Hwang, and Sunghun Kim. „Integrating code search into the development session“. In: *2011 IEEE 27th International Conference on Data Engineering*. 2011, pp. 1336–1339. DOI: `10.1109/ICDE.2011.5767948`.

[53] Patrick Lewis et al. *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks*. 2021. arXiv: `2005.11401 [cs.CL]`. URL: `https://arxiv.org/abs/2005.11401`.

[54] Jiarui Li, Ye Yuan, and Zehua Zhang. *Enhancing LLM Factual Accuracy with RAG to Counter Hallucinations: A Case Study on Domain-Specific Queries in Private Knowledge-Bases*. 2024. arXiv: `2403.10446 [cs.CL]`. URL: `https://arxiv.org/abs/2403.10446`.

[55] Xuan Li et al. „Few-shot code translation via task-adapted prompt learning“. In: *Journal of Systems and Software* 212 (2024), p. 112002. ISSN: 0164-1212. DOI: `https://doi.org/10.1016/j.jss.2024.112002`. URL: `https://www.sciencedirect.com/science/article/pii/S0164121224000451`.

[56] Jenny T. Liang, Chenyang Yang, and Brad A. Myers. „A Large-Scale Survey on the Usability of AI Programming Assistants: Successes and Challenges“. In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ICSE ’24. Lisbon, Portugal: Association for Computing Machinery, 2024. ISBN: 9798400702174. DOI: `10.1145/3597503.3608128`. URL: `https://doi.org/10.1145/3597503.3608128`.

[57] Shangqing Liu et al. „GraphSearchNet: Enhancing GNNs via Capturing Global Dependencies for Semantic Code Search“. In: *IEEE Transactions on Software Engineering* 49.4 (2023), pp. 2839–2855. DOI: `10.1109/TSE.2022.3233901`.

[58] Wenhan Lyu et al. „Evaluating the Effectiveness of LLMs in Introductory Computer Science Education: A Semester-Long Field Study“. In: *Proceedings of the Eleventh ACM Conference on Learning @ Scale*. L@S ’24.

ACM, July 2024, pp. 63–74. DOI: 10.1145/3657604.3662036. URL: http://dx.doi.org/10.1145/3657604.3662036.

[59] Alex Mallen et al. „When Not to Trust Language Models: Investigating Effectiveness of Parametric and Non-Parametric Memories“. In: *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers).* Ed. by Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki. Toronto, Canada: Association for Computational Linguistics, July 2023, pp. 9802–9822. DOI: 10.18653/v1/2023.acl-long.546. URL: https://aclanthology.org/2023.acl-long.546.

[60] Vadim Markovtsev and Waren Long. „Public git archive: a big code dataset for all“. In: *Proceedings of the 15th International Conference on Mining Software Repositories.* MSR '18. Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 34–37. ISBN: 9781450357166. DOI: 10.1145/3196398.3196464. URL: https://doi.org/10.1145/3196398.3196464.

[61] Thomas Merth et al. *Superposition Prompting: Improving and Accelerating Retrieval-Augmented Generation.* 2024. arXiv: 2404.06910 [cs.CL]. URL: https://arxiv.org/abs/2404.06910.

[62] Sewon Min et al. *FActScore: Fine-grained Atomic Evaluation of Factual Precision in Long Form Text Generation.* 2023. arXiv: 2305.14251 [cs.CL]. URL: https://arxiv.org/abs/2305.14251.

[63] Horia Modran et al. „LLM Intelligent Agent Tutoring in Higher Education Courses using a RAG Approach“. In: *Preprints* (July 2024). DOI: 10.20944/preprints202407.0519.v1. URL: https://doi.org/10.20944/preprints202407.0519.v1.

[64] Parastoo Mohagheghi and Reidar Conradi. „Quality, productivity and economic benefits of software reuse: a review of industrial studies“. In: *Empirical Software Engineering* 12 (2007), pp. 471–516.

[65] Martin Monperrus. „Automatic Software Repair: A Bibliography“. In: *ACM Comput. Surv.* 51.1 (Jan. 2018). ISSN: 0360-0300. DOI: 10.1145/3105906. URL: https://doi.org/10.1145/3105906.

[66] Marcus Nachtigall, Lisa Nguyen Quang Do, and Eric Bodden. „Explaining Static Analysis – A Perspective“. In: *Proceedings of the International Conference on Software Engineering*. Paderborn University and Fraunhofer IEM. 2019.

[67] Daye Nam et al. *Using an LLM to Help With Code Understanding*. 2024. arXiv: `2307.08177 [cs.SE]`. URL: `https://arxiv.org/abs/2307.08177`.

[68] Yuqi Nie et al. *A Survey of Large Language Models for Financial Applications: Progress, Prospects and Challenges*. 2024. arXiv: `2406.11903 [q-fin.GN]`. URL: `https://arxiv.org/abs/2406.11903`.

[69] Ollama. *Ollama-js: Official JavaScript library for Ollama*. `https://github.com/ollama/ollama-js`. 2024.

[70] Ollama. *Ollama: Get up and running with large language models, locally*. `https://ollama.com`. 2024.

[71] Mohammad Masudur Rahman and Chanchal K. Roy. „A Systematic Review of Automated Query Reformulations in Source Code Search“. In: *ACM Trans. Softw. Eng. Methodol.* 32.6 (Sept. 2023). ISSN: 1049-331X. DOI: `10.1145/3607179`. URL: `https://doi.org/10.1145/3607179`.

[72] Sanka Rasnayaka et al. *An Empirical Study on Usage and Perceptions of LLMs in a Software Engineering Project*. 2024. arXiv: `2401.16186 [cs.SE]`. URL: `https://arxiv.org/abs/2401.16186`.

[73] Gilberto Recupito et al. „Technical debt in AI-enabled systems: On the prevalence, severity, impact, and management strategies for code and architecture“. In: *Journal of Systems and Software* 216 (2024), p. 112151. ISSN: 0164-1212. DOI: `https://doi.org/10.1016/j.jss.2024.112151`. URL: `https://www.sciencedirect.com/science/article/pii/S0164121224001961`.

[74] Daniel Russo. „Navigating the Complexity of Generative AI Adoption in Software Engineering“. In: *ACM Trans. Softw. Eng. Methodol.* 33.5 (June 2024). ISSN: 1049-331X. DOI: `10.1145/3652154`. URL: `https://doi.org/10.1145/3652154`.

[75] Caitlin Sadowski, Kathryn T. Stolee, and Sebastian Elbaum. „How developers search for code: a case study“. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering.* ESEC/FSE 2015. Bergamo, Italy: Association for Computing Machinery, 2015, pp. 191–201. ISBN: 9781450336758. DOI: `10.1145/2786805.2786855`. URL: `https://doi.org/10.1145/2786805.2786855`.

[76] Yuchen Shao et al. *Vortex under Ripplet: An Empirical Study of RAG-enabled Applications.* 2024. arXiv: `2407.05138 [cs.SE]`. URL: `https://arxiv.org/abs/2407.05138`.

[77] Janice Singer et al. „An Examination of Software Engineering Work Practices“. In: (Mar. 2002).

[78] Raphael Sirres et al. „Augmenting and structuring user queries to support efficient free-form code search“. In: *Proceedings of the 40th International Conference on Software Engineering.* ICSE '18. Gothenburg, Sweden: Association for Computing Machinery, 2018, p. 945. ISBN: 9781450356381. DOI: `10.1145/3180155.3182513`. URL: `https://doi.org/10.1145/3180155.3182513`.

[79] Craig Smith. „Mom, Dad, I Want To Be A Prompt Engineer“. In: *Forbes* (Apr. 2023). URL: `https://www.forbes.com/sites/craigsmith/2023/04/05/mom-dad-i-want-to-be-a-prompt-engineer/`.

[80] Spiceworks News and Insights. „ChatGPT Leaks Sensitive User Data, OpenAI Suspects Hack“. In: *Spiceworks* (Apr. 2023). URL: `https://www.spiceworks.com/tech/artificial-intelligence/news/chatgpt-leaks-sensitive-user-data-openai-suspects-hack/`.

[81] Alexey Svyatkovskiy et al. „IntelliCode Compose: Code Generation Using Transformer“. In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* ESEC/FSE 2020. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 1433–1443. ISBN: 9781450370431. DOI: `10.1145/3368089.3417058`. URL: `https://doi.org/10.1145/3368089.3417058`.

[82] continue.dev Team. *continue.dev: AI-Powered Development Tools.* Web Platform. 2024. URL: `https://continue.dev/`.

[83] Nexusflow.ai team. *NexusRaven-V2: Surpassing GPT-4 for Zero-shot Function Calling.* 2023. URL: `https://nexusflow.ai/blogs/ravenv2`.

[84] Sebastian Tewes. „Retrieval-Augmented Large Language Models“. Studienarbeit. Karlsruher Institut für Technologie (KIT), 2024. 38 pp. DOI: `10.5445/IR/1000170392`.

[85] Aman Singh Thakur et al. *Judging the Judges: Evaluating Alignment and Vulnerabilities in LLMs-as-Judges.* 2024. arXiv: `2406.12624 [cs.CL]`. URL: `https://arxiv.org/abs/2406.12624`.

[86] Chenguang Wang, Mu Li, and Alexander J. Smola. *Language Models with Transformers.* 2019. arXiv: `1904.09408 [cs.CL]`. URL: `https://arxiv.org/abs/1904.09408`.

[87] Junjie Wang et al. *Software Testing with Large Language Models: Survey, Landscape, and Vision.* 2024. arXiv: `2307.07221 [cs.SE]`. URL: `https://arxiv.org/abs/2307.07221`.

[88] Lei Wang et al. „Investigating the Impact of Prompt Engineering on the Performance of Large Language Models for Standardizing Obstetric Diagnosis Text: Comparative Study“. In: *JMIR Form Res* 8 (Feb. 2024), e53216. ISSN: 2561-326X. DOI: `10.2196/53216`. URL: `http://www.ncbi.nlm.nih.gov/pubmed/38329787`.

[89] Man-Fai Wong et al. „Natural Language Generation and Understanding of Big Code for AI-Assisted Programming: A Review“. In: *Entropy* 25.6 (2023). ISSN: 1099-4300. DOI: `10.3390/e25060888`. URL: `https://www.mdpi.com/1099-4300/25/6/888`.

[90] Kevin Wu, Eric Wu, and James Zou. *ClashEval: Quantifying the tug-of-war between an LLM's internal prior and external evidence.* 2024. arXiv: `2404.10198 [cs.CL]`. URL: `https://arxiv.org/abs/2404.10198`.

[91] Ziwei Xu, Sanjay Jain, and Mohan Kankanhalli. *Hallucination is Inevitable: An Innate Limitation of Large Language Models.* 2024. arXiv: `2401.11817 [cs.CL]`. URL: `https://arxiv.org/abs/2401.11817`.

[92] Xin Yin et al. *Rectifier: Code Translation with Corrector via LLMs.* 2024. arXiv: `2407.07472 [cs.SE]`. URL: `https://arxiv.org/abs/2407.07472`.

[93] Hao Yu et al. „Incorporating Code Structure and Quality in Deep Code Search“. In: *Applied Sciences* 12.4 (2022). ISSN: 2076-3417. DOI: `10.3390/app12042051`. URL: `https://www.mdpi.com/2076-3417/12/4/2051`.

[94] Nico Zazworka et al. „Investigating the impact of design debt on software quality“. In: (May 2011). DOI: `10.1145/1985362.1985366`.

[95] K Zhang et al. „Revolutionizing Healthcare: The Transformative Impact of LLMs in Medicine“. In: *Journal of Medical Internet Research* (2024). forthcoming/in press. DOI: `10.2196/59069`. URL: `https://preprints.jmir.org/preprint/59069`.

[96] Liang Zhang et al. „A novel deep learning-based approach for software bug localization“. In: *IEEE Access* 7 (2019), pp. 63135–63147.

[97] Tingkai Zhang et al. *SQLfuse: Enhancing Text-to-SQL Performance through Comprehensive LLM Synergy.* 2024. arXiv: `2407.14568 [cs.CL]`. URL: `https://arxiv.org/abs/2407.14568`.

[98] Yujia Zhou et al. *Trustworthiness in Retrieval-Augmented Generation Systems: A Survey.* 2024. arXiv: `2409.10102 [cs.IR]`. URL: `https://arxiv.org/abs/2409.10102`.

[99] Qihao Zhu et al. „DeepSeek-Coder-V2: Breaking the Barrier of Closed-Source Models in Code Intelligence“. In: *arXiv preprint arXiv:2406.11931* (2024).

[100] Barret Zoph et al. „Emergent abilities of large language models“. In: *TMLR* (2022).

# Appendix

## A.1. Solution Code

The solution code is included in the zip file uploaded with the thesis, as well as uploaded in the GitLab repository from the chair of software engineering at Mannheim's university.

## Declaration of Authorship / Eidesstattliche Erklärung

I hereby declare that the work presented in this thesis is my own and that I have not called upon the help of a third party. In addition, I affirm that neither I nor anybody else has previously submitted this work or parts of it to obtain credits elsewhere. I have clearly marked and acknowledged all quotations and references that have been taken from the works of others. All secondary literature and other sources are marked and listed in the bibliography. The same applies to all charts, diagrams and illustrations as well as to all Internet resources. Moreover, I consent to my paper being electronically stored and sent anonymously in order to be checked for plagiarism. I know that if this declaration is not made, the paper may not be graded.

---

Hiermit versichere ich, dass diese Abschlussarbeit von mir persönlich verfasst ist und dass ich keinerlei fremde Hilfe in Anspruch genommen habe. Ebenso versichere ich, dass diese Arbeit oder Teile daraus weder von mir selbst noch von anderen als Leistungsnachweise andernorts eingereicht wurden. Wörtliche oder sinngemäße Übernahmen aus anderen Schriften und Veröffentlichungen in gedruckter oder elektronischer Form sind gekennzeichnet. Sämtliche Sekundärliteratur und sonstige Quellen sind nachgewiesen und in der Bibliographie aufgeführt. Das Gleiche gilt für graphische Darstellungen und Bilder sowie für alle Internet-Quellen.

Ich bin ferner damit einverstanden, dass meine Arbeit zum Zwecke eines Plagiatsabgleichs in elektronischer Form anonymisiert versendet und gespeichert werden kann. Mir ist bekannt, dass von der Korrektur der Arbeit abgesehen werden kann, wenn die Erklärung nicht erteilt wird.

Mannheim, September 28, 2024

A. Shalaby

MHD Ayham Shalaby

## Assignment of Usage Rights / Abtretungserklärung

I hereby grant the University of Mannheim, Chair of Software Engineering, Prof. Dr. Colin Atkinson, extensive, exclusive, unlimited and unrestricted usage rights to the work described in this thesis.

This includes the right to use the results in research and teaching. In particular, this includes the right to reproduce, distribute, translate, change and transfer the results to third parties, as well as any further results derived from them.

Whenever the results of my work are used in their original form or in a revised version, I will be mentioned by name as a co-author, in compliance with the copyright rules. This assignment does not include any commercial use.

---

Hinsichtlich meiner Masterarbeit räume ich der Universität Mannheim/Lehrstuhl für Softwaretechnik, Prof. Dr. Colin Atkinson, umfassende, ausschließliche unbefristete und unbeschränkte Nutzungsrechte an den entstandenen Arbeitsergebnissen ein.

Die Abtretung umfasst das Recht auf Nutzung der Arbeitsergebnisse in Forschung und Lehre, das Recht der Vervielfältigung, Verbreitung und Übersetzung sowie das Recht zur Bearbeitung und Änderung inklusive Nutzung der dabei entstehenden Ergebnisse, sowie das Recht zur Weiterübertragung auf Dritte.

Solange von mir erstellte Ergebnisse in der ursprünglichen oder in überarbeiteter Form verwendet werden, werde ich nach Maßgabe des Urheberrechts als Co-Autor namentlich genannt. Eine gewerbliche Nutzung ist von dieser Abtretung nicht mit umfasst.

Mannheim, September 28, 2024

A. Shalaby

MHD Ayham Shalaby