Orchestrating graph and semantic searches for code analysis

TNO Public ⟩ TNO 2025 R10992
12 May 2025

ICT, Strategy & Policy
www.tno.nl

TNO 2025 R10992 – 12 May 2025

Orchestrating graph and semantic searches for code analysis

Author(s)	Laurens Prast, Rosilde Corvino, Joe Reynolds, Nan Yang
Classification report	TNO Public
Title	Orchestrating graph and semantic searches for code analysis
Report text	TNO Public
Number of pages	40 (excl. front and back cover)
Number of appendices	2

All rights reserved
No part of this publication may be reproduced and/or published by print, photoprint, microfilm or any other means without the previous written consent of TNO.

Acknowledgement
The research is supported by the Netherlands Organisation for Applied Scientific Research TNO.

© 2024, 2025 TNO

Abstract

High-tech original equipment manufacturers rely on long-living, software-intensive complex systems that must continuously evolve to meet fast-paced market demands. This evolution often leads to the accumulation of technical debt, making maintenance increasingly challenging. This legacy code problem highlights the need for efficient strategies to keep these systems adaptable and fit-for-purpose.

Code analysis tools like GitHub Copilot have integrated Large Language Models (LLMs) with vector databases to answer questions about codebases. These tools excel in answering questions on local code snippets or concerning the semantics of parts of the code (we call these functional questions) but have limitations in answering questions needing precise and complete knowledge of code dependencies, such as questions about the structure of the code. The LLM4Legacy group have developed a tool integrating LLMs with code graphs to address these structural questions. In this KIP project, both approaches (leveraging vector and graph databases) were combined to enable the answering of functional, structural, and hybrid questions (questions with both a functional and structural component) about legacy software.

To achieve this, an orchestration-based retrieval approach was developed that dynamically selects the most suitable retrieval method depending on the question while considering the scale of the retrieved context not to exceed the context window of the orchestration LLM. Using LangGraph, a multi-source retrieval was implemented that leverages both vector and graph databases. Our results show improved accuracy in structural and hybrid questions while maintaining strong performance on functional questions compared to GitHub Copilot. This research advances code analysis for legacy software and provides insights into multi-agent coordination.

Contents

1 Introduction

The research of the TNO-ESI LLM4Legacy group has focused on combining Large Language Models (LLMs) with code graphs to enhance the analysis of legacy software systems. By integrating LLMs with graph databases, we can effectively address complex structural questions about codebases, such as dependency relationships and the impact of changes to specific code structures. These dependency relationships within the codebase analyzed in this study are further detailed in the evaluation section. However, when dealing with code, we also want to reason about its functionality and behaviour. This requires understanding the semantics of the code source and its runtime behaviour. In this work, we limit our scope to the code information that we can infer statically, thus, we do not consider the run time behaviour of the code. To further expand our LLM-based solution beyond mere structural questions, we propose combining multiple sources of information. Specifically, we consider:

1. The internal and external dependencies of the code, captured by a code graph.
2. The semantics of the source code, captured by encoding (or embeddings) of the source code itself.

To access and combine these two sources of information, we need mechanisms to map them to one another and to orchestrate their access. This requires the introduction of an orchestrator that can dynamically select the appropriate tools to retrieve information from different data sources, such as vector databases and graph databases. This orchestration approach aims to improve the accuracy of answering functional, structural, and hybrid questions. This leads to the following research questions:

1. How can an orchestration system be designed to efficiently select the appropriate tools for answering different types of code-related questions?
2. What database designs can be employed within the orchestration system to efficiently process large volumes of retrieved data while ensuring that the retrieved context does not exceed the context window of the orchestration LLM?
3. How does the proposed orchestration system compare to existing code analysis tools?

This report describes the information collected and experiments performed between October and December 2024 during an ESI study on using orchestration to manage legacy software. This activity aimed to build knowledge around the topic via a survey of related works and first explorations around the complexity and scalability challenges of using orchestration to retrieve information from multiple sources and answering complex questions about legacy code. In conclusion, we identify open research questions and possible solutions that must be addressed in the coming years.
This document is organized as follows. Chapter 2 introduces the research context, providing the foundational background for this work. Chapter 3 reviews related works on orchestration and the analysis of such systems. Chapter 4 details the orchestration architecture and database design, outlining the core system framework. Chapter 5 explains the experimental design and provides a discussion of the observed results. Chapter 6 presents the general conclusions and suggests directions for future work. Chapter 7 contains the references, and Chapter 8 includes the appendix.

2 Context

High-tech Original Equipment Manufacturers in the Dutch industry, such as Philips, ASML, Thermo Fisher Scientific, ITEC, Canon, Thales and Vanderlande, deal with long-living software-intensive complex systems. These systems are inherently prone to accumulating technical debt due to their complexity and the need to meet fast-paced market demands. To remain fit for purpose, they require constant rejuvenation and maintenance. This problem, known as the legacy code problem, is estimated to cost 1.52 trillion dollars in the US [1], with similar conclusions that could be drawn for Europe.

Legacy code often lacks thorough or accurate documentation, which complicates efforts to understand and maintain the system [2]. LLMs have demonstrated their effectiveness in summarizing code [3], and one logical extension of this capability is generating recommendations for improving codebases [4]. However, a primary limitation of LLMs is their inability to handle large context sizes, as the retrieved context often exceeds the model's capacity. Also, context in modern code assistants is often retrieved via a semantic similarity search, which is inherently not adapted to capture the code structure and dependencies. Thus, current code assistants cannot effectively address all types of questions about the code.

Over the past year, the TNO-ESI LLM4Legacy team investigated the potential of LLMs in managing legacy software systems. The study began by assessing current code analysis tools, revealing that LLMs integrated with vector databases excel in addressing functional questions, such as identifying which function implements a particular feature [5]. Retrieval-augmented generation (RAG) enhances LLMs by integrating external knowledge during the generation process. When using vector databases in the backend, RAG enables the analysis of codebases by retrieving relevant information based on semantic similarity. This process involves encoding both the user's query and the database contents into high-dimensional vectors, using similarity metrics to retrieve the most relevant data, which is then passed to the LLM to generate contextually informed responses.

However, while effective for tasks such as code completion and answering functional questions, vector databases are limited when it comes to handling complex structural questions such as, “Which functions are affected by changes to function X?” or “What are the dependency relationships between component X and component Y?”. This limitation highlights the need for a complementary graph database, to address structural questions about codebases.

Within TNO-ESI, a tool for code analysis and rejuvenation called Renaissance was developed to extract a code graph for large industrial codebases [6]. These code graphs are particularly well-suited to provide a context for LLMs handling structural queries. Code graphs organize data into nodes and edges, where nodes represent entities (e.g. functions, classes, or files) and edges represent dependency relationships. (e.g. functions calls to another function, inheritance or functions in a class). In a RAG workflow using a graph database, the LLM generates a Cypher query based on a user’s question, which is then executed on the graph database to retrieve relevant relationships between components. This system excels at analysing dependencies but faces challenges when it comes to answering functional questions, such as explaining how specific pieces of code operate.

Large language model-based autonomous agents (LAA) are specialised LLMs capable of executing sequential action, interacting with environments and solving complex tasks through the collection of observations [7]. In the context of LAAs, orchestration refers to the coordination of the actions of various agents to accomplish intricate tasks. The orchestrator selects and facilitates interaction between agents that perform distinct subtasks, such as reasoning, searching, or retrieving information. This modular approach is designed to improve task efficiency and scalability by distributing labour into specialized roles [7]. A multi-agent system is a computational framework consisting of multiple interacting intelligent agents that collaborate to address complex problems beyond the capability of any single agent [8]. With the integration of LLMs, these systems have evolved into LLM-Based Multi-Agent (LMA) systems [9], where LLMs serve as LAAs, enhancing decision-making and coordination within the orchestration.

This study concluded that effectively addressing a broad spectrum of questions, including functional, structural, and hybrid questions (e.g., combining structural and functional aspects, to get a question such as "Which functions are affected by changes to function X and what do these functions do?"), requires an orchestrator capable of selecting the appropriate tool to answer each question type. A tool, in this context, refers to a resource available to the orchestrator, such as a retriever from a vector database or an agent that translates a natural language question into a Cypher query to retrieve information from a graph database, to provide accurate answers. A key aspect of this orchestration is setting up the databases in a way that ensures the large volumes of context retrieved during the process do not exceed the context window of the orchestrator.

The objectives of this KIP are twofold:

1. **Reason and conclude on Orchestration Architectures**: Investigate different orchestration frameworks, develop a system integrating multiple tools, and propose strategies for improving the orchestration process.
2. **Reason and conclude on Database Design**: Explore various database designs to support the system's requirements, focusing on scalability and efficiency when working with larger codebases.

This research aims to provide insights into building orchestration systems capable of handling diverse questions about legacy software while also offering strategies for efficiently managing retrieved data to ensure that the context window of the orchestration LLM is not exceeded.

3 Related Works

As previously mentioned, the current LLM4Legacy code assistant answers questions about codebases relying on a large language model that generates Cypher queries to retrieve context information from the Renaissance code graph created using static analysis tools. While this system effectively addresses structural questions, it struggles with functional and hybrid questions. Consequently, there is a need to develop a solution with an orchestrator that selects and coordinates the appropriate tools and agents. A LAA can fulfil this role since it can manage other agents and tools that perform specialized tasks. The literature presents several systems with similar objectives and configurations, such as:

- **CodexGraph** [10]: A tool similar to the current setup of LLM4Legacy, CodexGraph enhances code generation and analysis by navigating code repositories. It builds graph databases through static analysis, where code structures are represented as nodes and their relationships as edges. CodexGraph allows LLM-based agents to interact with these graphs via query generation and execution.
- **Intelligent Code Analysis Agent (ICAA)** [11]: Used for bug detection in static code analysis, ICAA employs agents that utilize tools like web search, static analysis, and code retrieval to perform their tasks.

The orchestration system presented in this study differs from CodexGraph, which is designed specifically for graph-based search within Python codebases. In contrast, our system extends beyond graph search by incorporating orchestration capabilities to coordinate multiple agents and tools, with a specific focus on C++ codebases. Additionally, while the ICAA is primarily designed for bug detection, our system is intended for code analysis.

3.1 Orchestration architectures

Recent advancements in autonomous agent architectures and LMA systems have introduced innovative frameworks to improve task performance, reasoning, and acting. In Table 1, ReAct, PlanAct, PlanReAct, and BOLAA are discussed. These architectures have contributed novel methodologies that address key challenges in reasoning, planning, and execution.

Table 1 Orchestration architectures

Architecture	How it works	Cons
ReAct	A single agent uses Chain-of-Thoughts for generation and action execution [12]	Limited problems scale; Incoherent during complex, multi-step reasoning
PlanAct	A single agent uses a planning phase before action execution, to drive its reasoning [7]	Computation inefficiency and low adaptability in interactive, rapidly changing environments
PlanReact	Combines the plan generation of PlanAct with the Chain-of-Thought reasoning of ReAct in a single agent [7]	Computation inefficiency and low adaptability still remain
BOLAA	An orchestrator manages the work distribution and result combination	Computation inefficiency and non-determinism of the solution increase

	from a pool of specialized worker agents [7]

3.1.1 Orchestration implementation

In this research, we focused on the orchestration implementation with LangGraph [13], as this is one of the most widely recognized frameworks in the field of LLM orchestration, featuring a large and active community for support. LangGraph was developed by the LangChain team as part of their ongoing efforts to enhance and extend the capabilities of LLM-based workflows. The LangGraph framework was introduced in 2023 [14], building upon the ideas and tools established by LangChain, which had already been gaining traction as a framework for chaining together LLM tasks. LangGraph aims to incorporate graph-based reasoning into multi-agent orchestration, allowing more complex task management. While LangChain initially focused on simpler chains of thought and task execution, LangGraph added the ability to represent relationships in a graph structure, making it easier to model and manage more intricate workflows involving multiple agents.
Since LangGraph already includes a pre-built ReAct agent within its library, it was selected for this initial study on orchestration. LangGraph's ReAct agent utilizes a structured internal graph to facilitate tool calling and agent behaviour management. The diagram shown in Figure 1 visually represents the core process in LangGraph's ReAct agent, where the flow begins at the "start" node, leading to the "agent" node. From there, the agent, which is the orchestrator, either decides to continue to the tools or ends the process, depending on if it can answer the question with the retrieved context.

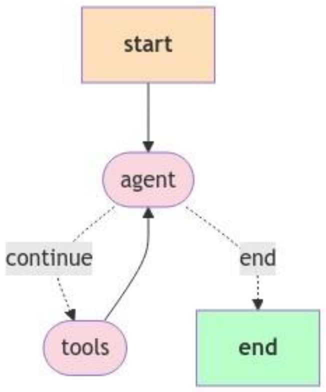

Figure 1 LangGraph's graph implementation of the ReAct orchestrator [15]

LangGraph uses a structured and reliable tool-calling mechanism. This approach allows the agent to directly call tools, pass inputs to them, and receive the outputs back in a structured format. This method not only simplifies the agent's workflow but also improves reliability by minimizing errors that may arise from parsing raw output.

The ReAct agent of LangGraph does not require the generation of a separate thought step by default. Modern LLMs are now capable of reasoning more effectively, making the explicit thought generation less necessary [15]. Although the LangGraph architecture is referred to as a ReAct agent, it aligns more closely with the BOLAA architecture due to its controller-based orchestration mechanism involving multiple specialized agents, rather than utilizing a thought process integrated into the memory of the orchestrator like the ReAct agent architecture. The ReAct agent of LangGraph directly performs actions based on the retrieved context. Being a new field of research, the terminology regarding these architectures is still evolving.

3.1.2 Evaluation orchestration architectures

In the paper introducing BOLAA [7], the researchers have also compared the different architectures by testing them in two environments: the WebShop [16] and HotPotQA [17]. WebShop is a simulated online shopping environment designed to evaluate agents' decision-making and navigation abilities. HotPotQA is a multi-hop question-answering environment that requires agents to retrieve and reason over information from multiple Wikipedia passages. A reward metric is used to evaluate how well the agents execute their tasks in the two environments.

In the WebShop environment, the reward is determined by the attribute overlap ratio between the product selected by the agent and the ground truth product [16]. The agent earns a higher reward when the attributes of its selected product closely match those of the ground truth product. This approach evaluates how accurately the agent navigates the environment and makes decisions to fulfil the given task requirements.

In the HotPotQA environment, the reward is based on the F1 score, which measures the balance between precision and recall in the agent's generated answers compared to the ground truth answers [17][18]. Precision reflects how much of the agent's answer aligns with the ground truth in content, semantic, and coverage, while recall measures how much of the ground truth answer is captured in the agent's response [19]. The F1 score assesses the agent's reasoning ability, its effectiveness in retrieving relevant information, and its capacity to synthesize an accurate answer from multiple information sources.

In both environments, the BOLAA architecture consistently outperforms the other architectures in the environments described above when larger LLMs are used. However, when smaller LLMs are utilized, the other architectures, specifically ReAct, can sometimes perform better. This difference occurs because smaller LLMs often struggle to generate effective communication messages between the controller and specialized agents in BOLAA, leading to inefficiencies and errors.

BOLAA achieves better performance because each agent focuses on a specific sub-task, such as reasoning or searching, enabling more efficient and accurate task execution. This modular approach reduces errors, enhances task efficiency, and leverages the advanced capabilities of larger LLMs, particularly in handling complex multi-step tasks.

3.2 Challenges for LMA systems

In this section, we report several challenges associated with LAAs [20]. We aim to emphasize specific challenges that we anticipate encountering in the context of utilizing LMA systems for analysing large industrial legacy codebases. Additionally, we underscore the critical challenge of agent selection within the orchestrator framework.

3.2.2 Hallucinations

LLMs frequently produce false information delivered with a high confidence level and can, therefore, negatively impact areas like code generation [21]. These inaccuracies can introduce risks, including the creation of security vulnerabilities and incorrect code. In the specific context of LLM4Legacy code analysis of legacy software, this issue may manifest as the LLM hallucinating information in the absence of retrieved context or by misinterpreting context altogether.

3.2.3 Prompt robustness

To ensure rational behaviour in LAAs, designers often integrate supplementary modules, such as memory and planning, into LLMs. However, this requires developing more complex prompts, which are prone to inconsistency, as even minor changes can lead to significantly different outcomes [22, 23]. This issue is magnified in LMA systems, where prompts for different agents can influence one another.

3.2.4 Agent selection

The orchestrator's task of determining the correct sequence of agents and tools to trigger in an LMA system is particularly challenging. The difficulty arises because the question may not clearly specify which tools should be employed or in what order. Without explicit direction, the orchestrator faces the challenge of making decisions about the appropriate sequence based on the context and the nature of the question, which can vary significantly across different scenarios and trials.

3.3 Metrics

Researchers commonly use F1 score and recall metrics to evaluate the performance of agentic systems. The F1 score provides a balanced measure of precision and recall, and recall evaluates how well the system's predicted outputs match the ground truth. Recall is evaluated independently, as it enables verification of whether all elements mentioned in the ground truth are present in the model's output without considering precision. This approach is particularly useful for fact-checking, ensuring that the output comprehensively includes all desired information [24]. Additionally, when it comes to retrieving data from a graph database by using a LLM for the querying, the F1 score is also a commonly employed evaluation metric [24]. A widely used framework for evaluating textual answers generated by LLMs is Ragas, which has a metric called factual correctness that functions similarly to the F1-score [25]. This involves an evaluation process where one LLM extracts statements from the generated answer and the ground truth. Another LLM determines if the statements are true positives, which are statements present in both the generated answer and the ground truth, false negatives, which are in the ground truth but missing from the generated answer, or false positives, which appear in the generated answer but not the ground truth.

4 Architecture and database design

This section provides an overview of the orchestration architecture and database design employed in this research. In Figure 2, the orchestration architecture is shown with the utilized database for each tool. First, the orchestration architecture is presented, outlining its implementation and the rationale behind the chosen approach. Second, the database design is discussed in detail, emphasizing how graph and vector stores were structured to address the diverse nature of queries, such as functional, structural, and hybrid questions, while also considering the scalability challenges posed by the limited context window of the LLM and strategies to mitigate this limitation.

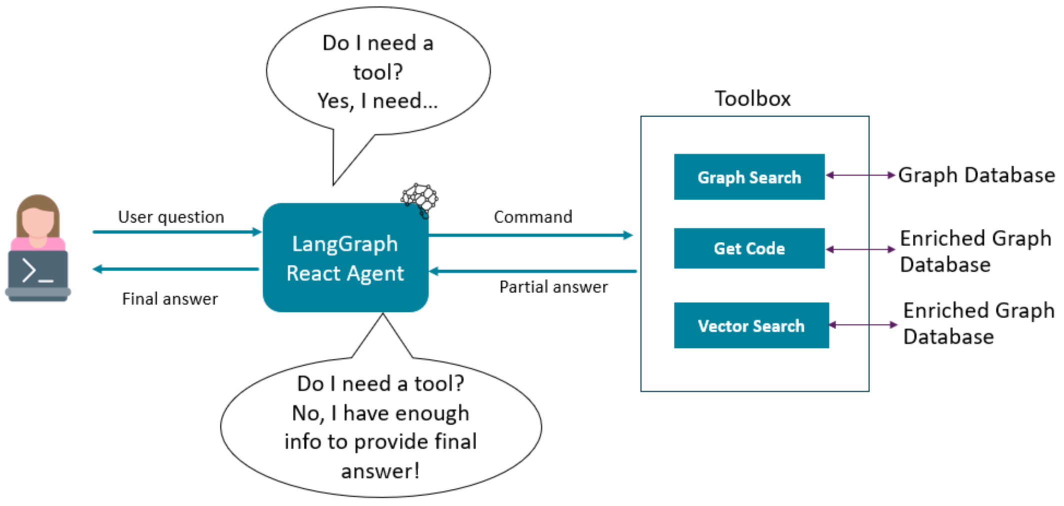

Figure 2 The orchestration architecture

4.1 Implementation orchestrator

In the related works section, several architectures are discussed, however, for this initial exploration, the BOOLA architecture was selected. This choice was driven by the availability of the implementation of the BOOLA architecture in the widely available LangGraph framework, which is further discussed in detail in Section 3.1.1.

4.2 Tools orchestrator

The orchestrator was designed to utilize the following tools:

- **Vector Search**: Addresses functional questions such as "What does this function do?" or "Which code structure implements functionality X?" It embeds the question and

performs a similarity search against the embedded summaries and code snippets, retrieving the name and summary of the relevant code structures.
- **Get Code**: Retrieves the code snippet of a function or class.
- **Graph Search**: Transforms natural language to Cypher queries and then performs a graph search on the graph database to answer structural questions. It is also useful for aggregation tasks, such as counting the number of functions.

The Vector Search tool could retrieve code snippets directly, but this approach risks exceeding the LLM's context window, as large code segments from multiple classes and functions might be retrieved. By separating the Get Code tool, only essential data is retrieved, preventing the context limit of the LLM from being exceeded.

Future iterations may include additional tools to enhance functionality further.

4.3 Database design

To effectively address functional, structural, and hybrid questions, our system required the capability to perform both graph searches and vector searches. Previously, the LLM4Legacy group developed a graph database using Renaissance, enabling graph searches to handle structural questions. Building on this foundation, a database capable of supporting vector-based retrieval for functionality-related queries was needed.
A critical design criterion was ensuring the scalability of the retrieved context, given the constrained context window size of the orchestrator's LLM. This section outlines the process of extracting and structuring the necessary information for vector-based retrieval and details the design strategies implemented to ensure scalability in the database design.

4.3.1 Code extraction

To develop a system capable of answering both functional and structural questions, the design required integration of the Renaissance code graph, which was used to retrieve structural information about the code, with an additional database containing descriptions of code snippets for answering functional queries. The Renaissance tool was also utilized to extract code snippets corresponding to the functions and classes in the Renaissance code graph. These code snippets were subsequently stored in the graph database as properties of their respective nodes. The Renaissance code graph was created using an abstract syntax tree, which offers the advantage of retrieving functions and classes directly. In contrast, traditional chunking methods typically rely on fixed sizes of characters or lines to segment the code into chunks, which may overlook the structural elements of the code.

4.3.2 Summarization and vectorization

Given the limited context window of the orchestrator's LLM, it was essential to generate summaries of the extracted code snippets, encompassing functions, classes, and higher-level structures such as files and folders. To achieve this, a hierarchical summarization strategy was implemented, ensuring scalability in summarizing the codebase [26]. This hierarchical approach was necessary because the LLM used for summarization also operates within a constrained context window. Larger code structures, such as entire files or folders, would otherwise exceed the summarization LLM's context window. By breaking down the summarization process into hierarchical levels, smaller components were summarized first, and these summaries were then aggregated to create concise representations of larger structures, preserving key information while staying within context window limitations.

Summarization began at the function level, where each function's code snippet was summarized. A depth-first search algorithm was used to include summaries of all functions a function depends on in the context provided to the LLM. Next, class-level summaries were generated by combining the extracted code for each class with the summaries of its associated functions. File-level summaries were created based on the summarized content of all functions and classes within each file. For projects, summaries were constructed from the summaries of all files and other projects linked to them in the graph database. The summaries of executables and solution files were based on the projects linked to them. Finally, folder-level summaries were generated through a depth-first search of the folder hierarchy, aggregating summaries from child folders, files, projects, executables, and solutions within each parent folder. In Figure 3, the hierarchical summarization is shown in a diagram.

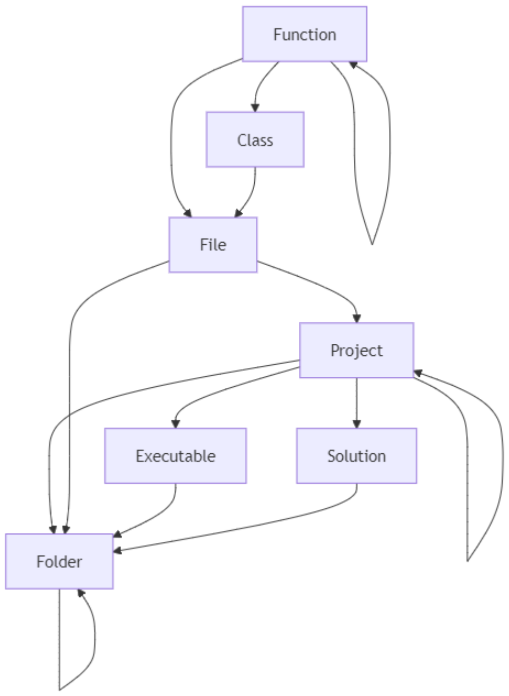

Figure 3 Diagram of the hierarchical summarization of the codebase. In this diagram, the arrows indicate that the code structure from which the arrow originates is utilized to generate a summary of the code structure to which the arrow is directed.

Subsequently, the summaries and code snippets were processed using the openai-text-embedding-3-small model, which is readily available in the Azure ecosystem, to generate vector embeddings. Each summary and code snippet was represented as a 1536-dimensional vector.

4.3.3 Databases

With the embeddings in place, enabling vector search to address functional questions, we needed to design the databases thoughtfully to ensure the LLM would not be overwhelmed with excessive context, thereby avoiding the risk of surpassing its context length limits. Neo4j, with its capability to store embeddings as properties of nodes and perform vector searches, provided a unique opportunity to leverage both graph-based and vector-based retrieval within a single system.

We explored two database design setups:

1. **An enriched graph database**: This database contains graph data, code snippet for functions and classes, as well as summaries for files, folders, projects, executables, and solution files. The associated embeddings for code snippets and summaries are also stored within this graph database. A graph search and a vector search are performed both on this database.
2. **A non-enriched graph database combined with an enriched graph database**: The non-enriched database contains the core graph structure, while the enriched database also includes code snippets, summaries and the associated embeddings for the code snippets and summaries. Graph searches are performed on the non-enriched database, containing only the core structure, ensuring efficient retrieval. Vector searches are handled by the enriched database, which stores embeddings, code snippets, and summaries to address functional questions.

The reason we explored these two options is that the first setup provides the convenience of having all information for graph and vector searches in a single database, while the second setup separates the data streams for graph and vector searches. This separation allows for more efficient retrieval tailored to the type of query being addressed.

We opted for an enriched graph database instead of a standalone vector database in the second setup because Neo4j's built-in support for vector searches removes the need to export summaries, code snippets, and embeddings to an external vector database.

The first setup was quickly found to be problematic. Performing a graph search in the enriched database required retrieving all node properties, including summaries, code snippets, and embeddings, to support both graph visualizations and graph metric computations. This retrieval significantly increased the volume of context, often exceeding the LLM's context limit. Consequently, separating the data streams was deemed necessary.

4.3.4 Considerations for future implementations

This approach has so far been applied to a static codebase. For evolving codebases, the cost of updating summaries for all classes and functions must be managed. One solution is change-based summary updates, where a change detection mechanism flags modified functions and classes. Summaries would then be regenerated only for these components and for their related higher-level structures when the lower-level changes have impact on the summaries of the higher-level structures.

5 Evaluation

This section outlines the methodologies used to assess the code assistant's performance, including the metrics for evaluation and the dataset utilized for testing. The evaluation of our research focuses on understanding the effectiveness of the proposed orchestrator in addressing diverse questions about legacy codebases. This involves assessing its ability to select the appropriate tool for specific query types, the accuracy of its responses across behavioural, structural, and hybrid questions, and its comparative performance against existing code analysis tools. The following research questions guide the evaluation:

1. **Correctness of responses:** How accurately can the orchestrator answer behavioural, structural, and hybrid questions about legacy software systems?
2. **Tool Selection:** To what extent does the orchestrator select the correct tool (e.g., graph search, vector search) for answering specific types of queries about legacy codebases?
3. **Comparative Performance:** How does the response accuracy of the orchestrator compare to established code assistant tools, such as GitHub Copilot?

5.1 Codebase

The codebase utilized for the research on orchestration is the Jsoncpp codebase [27], a compact, non-obfuscated software library specifically designed for handling JSON data. Jsoncpp facilitates the parsing, generation, and manipulation of JSON structures in C++ programming environments. Its simplicity makes it an ideal choice for this survey involving orchestration, as it allows for a clear analysis of interdependencies, functionalities, and structural relationships within a manageable and well-documented codebase.

5.2 Code graph

The code graph generated using Renaissance for the Jsoncpp codebase encapsulates dependencies, structural information and build-related components as follows:

- **Internal dependencies**: This includes function calls, inclusion dependencies, and the definition and usage of global symbols.
- **External dependencies**: References to external libraries, APIs, and frameworks utilized within the codebase.
- **Structural information**: The hierarchical organization of the code, including dependencies between elements and namespace structures.
- **Build-related components**: Representation of build containers such as Solution and Projects in Visual Studio, Executables and libraries compiled from projects and solutions.

5.3 Dataset

The dataset consists of ten functional questions, which pertain to the functionality of specific pieces of code, ten structural questions, which focus on the interdependencies among code

structures within the codebase, and ten hybrid questions, which combine aspects of both function-related and structure-related questions.
For example:

- **Behavioural question:** Which classes write values into the output in the form of a JSON value object?
- **Structural question:** How are the projects linked to the libraries in terms of dependencies? Provide the names of the projects and libraries.
- **Hybrid question:** What is the file path of the function with the highest number of calls in the codebase, and what does this function do?

5.4 Experimental design

The experimental design aims to evaluate the correctness of the proposed orchestrator in handling diverse question types about legacy codebases. By analysing the orchestrator's ability to select appropriate tools and provide accurate responses, the experiment aims to validate its capability to address behavioural, structural, and hybrid questions. Additionally, the design compares the orchestrator's performance with an existing code assistant tool to study its strengths and areas for improvement.

5.4.1 Textual metrics

Evaluating textual outputs often involves a degree of subjectivity. While numerical methods like cosine similarity can provide objective text comparison measures, they do not actually assess similarity in meaning and can consider texts with similar wording but opposite meanings as highly similar [29]. As discussed in section 3.3, metrics such as F1 score, and recall are commonly used and served as the starting point for our research. Figure 4 illustrates the answer generation and evaluation pipeline, highlighting the various textual metrics used to assess the correctness of the generated answers. After the orchestrator produces its final response, a human evaluation was conducted and compared against multiple approaches for LLM-based evaluation. This section discusses these different evaluation methods, determines which metric aligns most closely with human judgment, and subsequently reports the orchestrator's performance based on the selected metric.

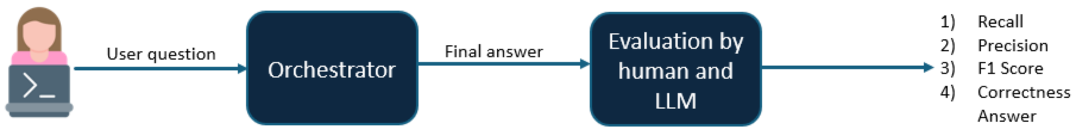

Figure 4 The answer generation and evaluation pipeline

The discussion begins with statement-level comparison using the Ragas framework, followed by an adaptation of the framework incorporating prompt engineering so the evaluation is adjusted to code analysis. Subsequently, a metric is introduced that compares individual statements from the ground truth and the generated answer against the full text of either the generated response or the ground truth. Finally, the metric ultimately employed involves a comparison of the entire ground truth with the entire generated response.

In Figure 5, an example is presented illustrating how the statements from the ground truth and the generated answer from the system are compared to compute the F1 score.

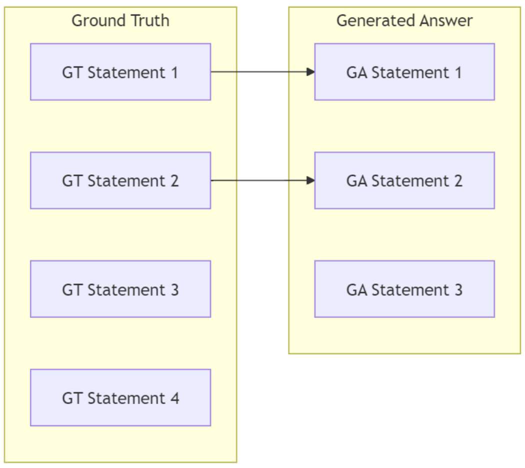

Figure 5 From the ground truth (GT), four statements were generated by the evaluation LLM, and from the generated answer (GA), three statements were produced. The arrows indicate that the statement is identified as similar as another statement. If no arrows point from a statement to another statement than it means that no statement is similar. Statements 1 and 2 from the GT were identified as similar to two statements from the GA by the evaluation LLM and were classified as True Positives (TP). Two statements in the GT were not found in the GA statements and were therefore classified as False Negatives (FN). Additionally, one statement in the GA was not found in the GT statements, classifying it as a False Positive (FP).

Firstly, when Ragas' automated evaluation was compared to human evaluation, discrepancies emerged, particularly in statement generation and classification, since the Ragas framework was not made with analysis of codebases in mind. For example, when handling a file retrieval question, it is crucial that each file is represented as a separate statement. However, Ragas did not consistently enforce this, resulting in an insufficient number of statements.

To address this problem, the examples provided to the LLM within the Ragas framework were customized to better align with our data. Specific examples were included to guide the LLM on both generating and classifying statements. While this adjustment improved the evaluation of the LLM, the results still did not meet expectations of the LLM evaluation being comparable to human evaluation. This is because the evaluation LLM compares statements one-to-one, whereas, for example, a single statement in the ground truth could be captured by multiple statements in the generated answer.

Afterward, we modified the comparison methodology. Instead of simultaneously generating and comparing statements for both the generated answers and the ground truth, we compared the statements of the generated answers against the full text of the ground truth and vice versa. In Figure 6, an example is presented illustrating how the statements from the ground truth and the generated answer from the system are compared to compute the F1 score with the changed comparison methodology. This approach significantly improved classification consistency but still exhibited considerable variance in the LLM's evaluations. This variance arises because the evaluation LLM does not consistently generate identical statements from the ground truth and the generated answer, nor does it classify them in the same way. This inconsistency can be attributed to the inherently subjective nature of interpreting the content of individual statements and determining whether they are captured by the ground truth, a challenge that humans also often find difficult to agree upon.

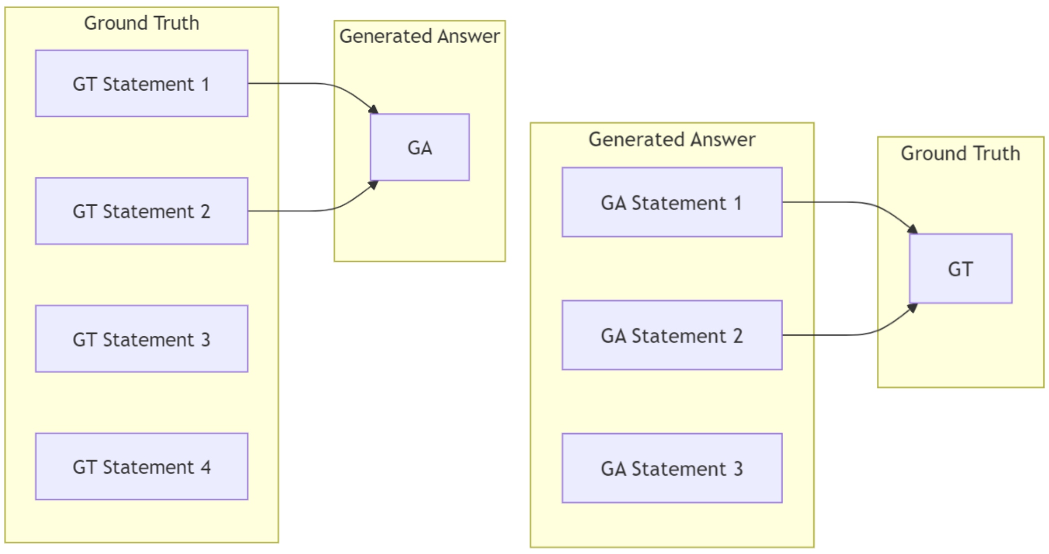

Figure 6 From the GT, four statements were generated by the evaluation LLM, and from the GA, three statements were produced. The arrows signify that a statement has been identified within the GA or GT. If no arrows connect a statement to either the GA or GT, this indicates that the statement is not identified in the GA or GT. Statements 1 and 2 from the GT were identified as present in the GA by the evaluation LLM and were classified as TP. Two statements in the GT were not found in the GA and were therefore classified as FN. Additionally, one statement in the GA was not found in the GT, classifying it as a FP.

Finally, our final evaluation methodology focused on the metric correctness answer, which compares the entirety of the generated answer against the complete ground truth. The evaluation LLM was tasked with determining whether the generated answer comprehensively captured all the information from the ground truth at once, which results in a correct or incorrect classification. This method aligned most closely with human evaluations and was ultimately the metric that was adopted to automate the evaluation process.

When analysing these metrics, it’s crucial to consider that the ground truth used for comparison may be more concise or detailed than the generated answer. This discrepancy can negatively affect the evaluation of the LLM.

Given the investigatory nature of this research and the small sample size, statistical analyses were avoided. Additionally, the structural questions were intentionally selected based on the known capabilities of the graph search tool.

As stated earlier in this section, the metric that most closely aligns with the human evaluation of the researchers of LLM4Legacy is the correctness answer, which is a binary metric (an answer can be correct 1 or not 0). Since the answer generated by the orchestrator is non-deterministic and the evaluation of the LLM is also non-deterministic, the orchestration and evaluation pipeline was executed ten times on the dataset. The result is shown in the heatmap in Figure 7.

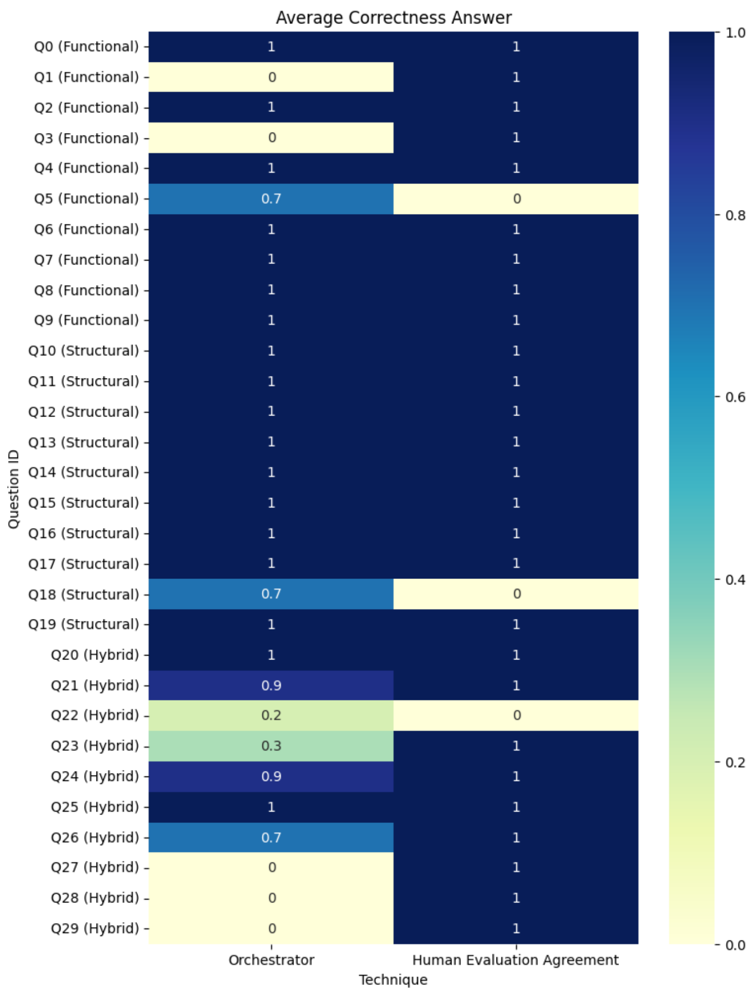

Figure 7 Heatmap of the average of the answer correctness of the orchestrator over ten runs in the left column. In the right column is shown if a human agrees with all the evaluations of the evaluation LLM for that question ID, where 1 indicates that the human does and 0 that the human does not.

In the appendix section A, on a question-to-question basis an analysis is shown for the question with an average correctness answer score of lower than 1 and in appendix section B, a table is shown that contains all the questions, ground truths and generated answers of a run that demonstrated performance closest to the median answer correctness score.

In Figure 7, the right column clearly demonstrates that the orchestrator performs exceptionally well in answering structural questions, as all but one question were correctly answered across ten runs. Seven functional questions were correctly answered in all ten runs, while the hybrid questions exhibited more varied results. Most hybrid questions were correct in some runs, but not consistently across all runs. This variability can be attributed to the fact that hybrid questions require the use of multiple tools and steps, which introduces a higher degree of non-determinism. In the left column, the agreement between the human evaluator and the automated evaluation is presented, with a value of one indicating full agreement for each run of a question. This level of agreement was observed for 27 out of 30 questions. In Section 5.5, a more detailed analysis is provided, along with the metrics from the subsequent section, to further elucidate the issues encountered with the orchestrator.

5.4.2 Other metrics

In addition to evaluating the textual output of the orchestrator, we assessed its ability to select the appropriate tools. To do this, we outlined the tool selection the orchestrator should ideally use, along with their expected order, and compared this sequence to the actual tool selection used by the orchestrator using the Levenshtein distance [29]. It is important to note that deviations in tool selection or order do not necessarily indicate errors. Instead, this metric serves as an indicator of discrepancies between expected and actual behaviour, helping to identify significant differences for further analysis. A Levenshtein distance of zero indicates that the expected and actual tool selections are identical. Conversely, if discrepancies exist between the two lists, necessitating an insertion, deletion, or substitution to align them, the Levenshtein distance is nonzero. The results are shown per question in Figure 8, 9, 10.

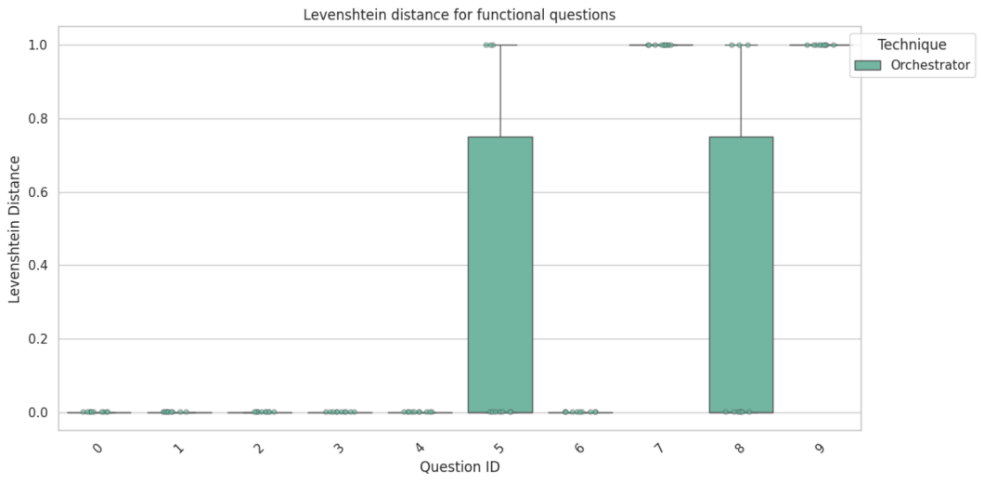

Figure 8 Boxplot of the Levenshtein distance of the orchestrator over ten runs for the functional questions

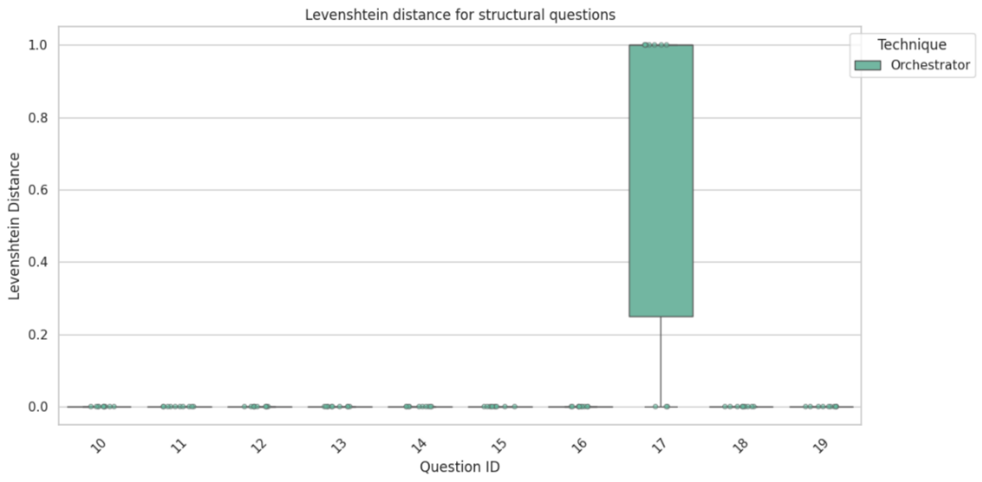

Figure 9 Boxplot of the Levenshtein distance of the orchestrator over ten runs for the structural questions

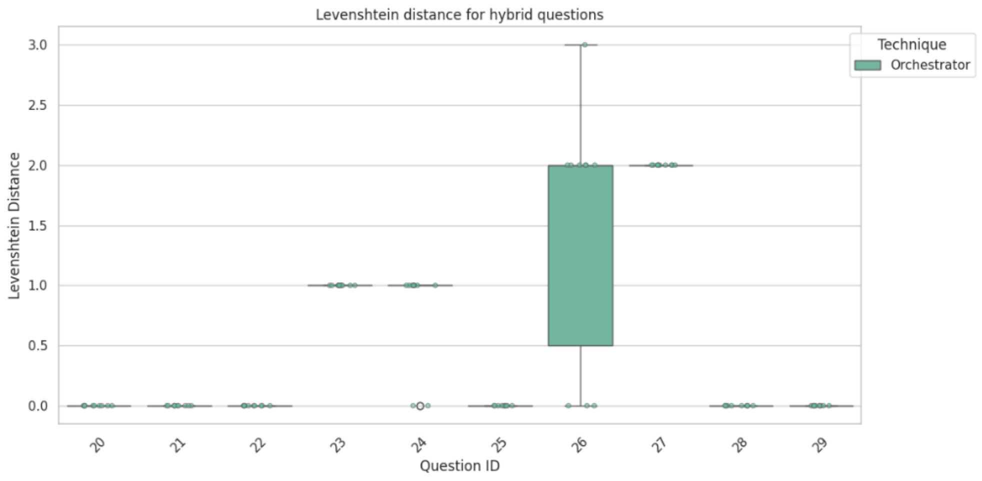

Figure 10 Boxplot of the Levenshtein distance of the orchestrator over ten runs for the hybrid questions

In instances where the Levenshtein distance deviated from the expected toolset, the orchestrator frequently invoked the "Get Code" tool. While not always necessary, this tool can enhance the completeness of the answer by providing relevant code excerpts.
Occasionally, an additional graph search is conducted on a variable, which yields no results since variables are not represented as nodes in the graph database.
At times, a vector search is performed for a structural question. However, the orchestrator typically recognizes that the required context will not be retrieved through this method and subsequently initiates a graph search to gather the necessary information.
In certain cases, multiple vector searches or "Get Code" operations are executed, particularly when information from multiple files is needed. Whether this can be accomplished with a single search or requires multiple searches depends on the query specifics made to the vector search tool.
Finally, while the execution order of tools may occasionally vary, this did not lead to incorrect answers within the dataset analysed.

5.5 Analysis of the orchestrator

Section 3.2 addressed several key challenges in LMA systems, including hallucinations, prompt robustness, and agent selection. These challenges affect the reliability of LMA systems and necessitate systematic mitigation strategies. During this study, we have identified potential solutions to these challenges.
One of the primary concerns in LMA systems is hallucination, where the system generates inaccurate or misleading information. To mitigate this issue, continuous human oversight is essential during both the development and deployment phases. Monitoring mechanisms should be implemented to detect and flag instances of incorrect or faulty outputs. User feedback can serve as a valuable source of information for system improvement, enabling post-deployment reflection, prompt engineering, retraining, or enhancements to the RAG system.
Ensuring prompt robustness across diverse LLMs remains a significant challenge in LMA systems. One approach to enhancing prompt reliability is through iterative manual refinement, where human evaluators assess the outputs of individual agents and the overall system response. Another strategy involves dynamic prompt construction, in which components of the prompt, such as rules or contextual information, are selected based on semantic similarity with the given query.
Effective agent selection is crucial for optimizing the decision-making process within LMA systems. One potential solution is the application of few-shot prompting, where the system is provided with examples of complex queries along with the correct sequence of agents or

tools required to process them. These examples guide the orchestration mechanism, enabling it to infer and execute the appropriate sequence of operations for various question types.

After analysing the generated answers of the orchestrator and comparing them to the ground truth, the following issues were identified:

1. **Redundant and incomplete information in functional questions**: The vector search can retrieve irrelevant or incorrect information based on the similarity scores between the query and the vector embeddings of code snippets and summaries. This results in the inclusion of information that does not directly address the question. Furthermore, the functional part of the generated answer can suffer from missing or inaccurate information. The search for the functional aspect of a query can be improved by combining vector-based search with keyword-based methods like BM-25, forming a hybrid approach. This technique integrates the semantic understanding of vector embeddings with the precision of keyword matching.
2. **Limited codebase coverage**: Not all code was extracted from the codebase, for instance code in example folders that is not present in the graph database. This absence hinders the ability to perform effective vector searches for those portions of the code. If the goal is to answer questions about files that are not part of the source code, we could extract the relevant portions of code for these files, enabling the model to provide answers for these parts of the code.
3. **Underutilization of Retrieved Context**: In the case of hybrid questions, which often involve multiple sub questions, not all of the retrieved context is used to answer every part of the question. This incomplete utilization of context results in partial responses to multi-part questions. This issue is expected to be addressed through few-shot prompting, which provides the LLM with examples demonstrating how to handle the context of multi-step questions effectively.
4. **Over-shortening in graph search**: The orchestrator occasionally shortens the sub questions too much, leading to incorrect Cypher queries that do not return the necessary data. This issue can be resolved through prompt engineering for the orchestrator, ensuring that it learns to utilize the full sub-question rather than excessively shortening it.
5. **Incorrect Search Parameters:** The orchestrator sometimes uses variable names as search parameters when querying function nodes, which yield no context at all. This occurs because variables, unlike functions or classes, are not represented as nodes in the graph database. To address this problem, it is crucial to include clearer instructions in the prompt for handling multi-step questions. This ensures that the model first retrieves the function using the variable before performing a graph search.
6. **Discrepancies in ground truth and generated answer**: The ground truth can either be more detailed or less exhaustive than the generated answer, which can lead to discrepancies in correctness assessments. This issue is particularly prominent when describing the functionality of specific parts of the codebase, as there are often multiple correct ways to phrase these descriptions. Additionally, formatting differences between the ground truth and the generated response, especially for large graph retrievals, can further contribute to the evaluation challenges. Therefore, when utilizing an evaluation LLM, it is essential to complement it with human evaluation and to repeat the evaluation multiple times with the LLM. This allows for averaging over multiple evaluations, thereby ensuring a more reliable and robust assessment.

5.6 The orchestrator versus Copilot

The non-determinism in correctness evaluation arises from both the answer generation and the answer evaluation processes, as both rely on LLMs. To mitigate this non-determinism, we averaged the results by generating answers ten times, selecting the answer with the correctness value closest to the median of these ten generations, and subsequently evaluating this selected answer ten times for the orchestrator and GitHub Copilot. The resulting evaluations are presented in Figure 11.

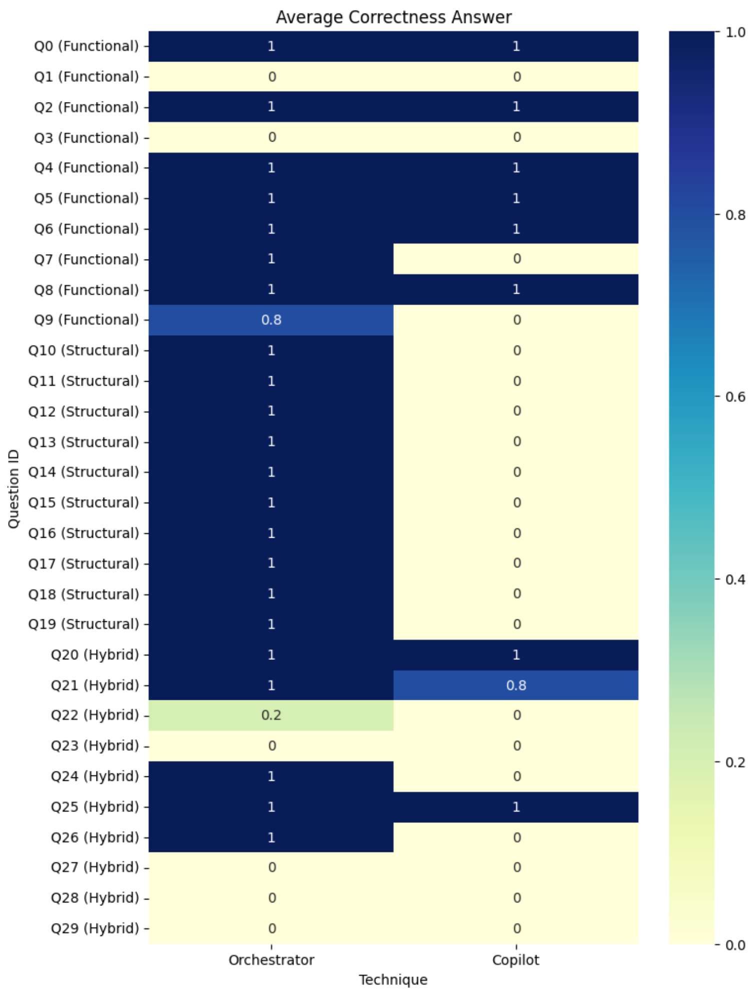

Figure 11 Heatmap of the average of the median generated answer of the orchestrator over ten evaluations and average of the generated answer of Copilot over ten evaluations

The results presented in Figure 11 demonstrate that the orchestrator exhibited superior performance in addressing structural queries and the structural part of the hybrid questions compared to Copilot. This performance differential can be attributed to the underlying architectural differences in information retrieval mechanisms. While Copilot employs a vector database that relies on semantic similarity for information retrieval, this approach appears insufficient for capturing complex interdependencies between functions, classes, files, and folders within a codebase. The graph database architecture implemented in the graph search agent in the orchestrator appears better suited for representing and querying these structural relationships.

Regarding functional queries, both systems demonstrated comparable baseline performance, with the orchestrator successfully resolving two additional test cases. A notable observation was Copilot's tendency to generate redundant information that occasionally contradicted the established ground truth. This was addressed in the orchestrator through targeted prompt engineering to minimize redundant output. While the current dataset does not provide conclusive evidence that hierarchical summarization improves functional query responses, the comparable performance suggests that this approach maintains at least equivalent efficacy in these types of questions.
Figure 11 illustrates that Question 9 and Question 22 can yield different answer correctness scores from the evaluation LLM for identical responses provided by the orchestrator. Similarly, for Question 21, the evaluation LLM assigns varying correctness scores for the same response generated by Copilot. These discrepancies arise because evaluating the functional aspects of a question can be challenging, particularly in determining whether the generated answer sufficiently covers all elements of the ground truth without including redundant or incorrect information. Moreover, the evaluation of textual output is inherently subjective, as interpreting meaning and deciding whether an answer is sufficiently comprehensive often depends on individual perspectives. This subjectivity was also evident when researchers assessed responses from different systems, as they occasionally disagreed on whether the answers adequately addressed the ground truth.

6 Conclusion and future work

In the introduction, the first research question that was asked was: "How can an orchestration system be designed to efficiently select the appropriate tools for answering different types of code-related questions?" The experimental setup involved using LangGraph's ReAct agent for orchestration, due to the fact that LangGraph is one of the most widely recognized frameworks in the field of LLM orchestration, featuring a large and active community for support.

The second research question addressed was: "What database designs can be employed within the orchestration system to efficiently process large volumes of retrieved data while ensuring that the retrieved context does not exceed the context window of the orchestration LLM?" The findings indicate that using separate databases for vector and graph search, rather than a single unified graph database, effectively mitigates the risk of exceeding the model's context window. This design allows for more targeted and efficient retrieval tailored to the nature of the question.

The third research question examined was: "How does the proposed orchestration system compare to existing code analysis tools?" The results suggest that the orchestrator offers more effective mechanisms for exploring relationships and dependencies in codebases, making it particularly valuable for addressing structural queries. However, further research is necessary to assess its performance on functional and hybrid questions. In particular, techniques such as hierarchical summarization and code extraction using Renaissance could potentially enhance the quality of responses in these cases. While the internal workings of Copilot are not fully transparent, it appears to rely on a traditional fixed chunk-size vector store and does not extract functions or classes individually. An interesting avenue for future research is to explore whether hierarchical summarization in combination with the code snippets of the functions and the classes can improve response quality, especially as the complexity of functional and hybrid queries increases.

To build on the current progress, several areas for improvement and exploration have been identified:

1. **Enhancing prompt engineering**
   Use few-shot prompting to provide the LLM examples to address underutilization of retrieved context and the tendency to over-shorten responses during graph searches.
2. **Improving instruction clarity for multi-step questions**
   Provide clearer, step-by-step instructions with examples to avoid errors like incorrect search parameters, for example when performing graph-based searches for variables, include guidance to first identify the associated function or class and then conduct the graph search.
3. **Incorporating more specialized agents**
   Introduce additional agents tailored to specific question types, such as behavioural questions focused on runtime behaviour or visualizations like making a UML diagram.
4. **Investigating plan integration for complex queries**

For challenging multi-step questions or scenarios involving multiple agents, explore integrating the planning component of PlanAct into the orchestration process. This could provide structured strategies for coordinating agents and improve overall system performance on complex questions.

By addressing these areas, the orchestration system can further improve its reliability and adaptability to a wider range of question types, paving the way for more robust applications in codebase analysis.

7 Bibliography

[1] H. Krasner, *Cost of Poor Software Quality in the U.S.: A 2022 Report*, Consortium for Information & Software Quality, Dec. 2022. Accessed: Dec. 23, 2024. [Online]. Available: https://www.it-cisq.org/the-cost-of-poor-quality-software-in-the-us-a-2022-report/
[2] T. Ahmed and P. Devanbu, “Few-shot training LLMs for project-specific code-summarization,” arxiv, Sep. 8, 2022. Accessed: Dec. 23, 2024. [Online]. Available: http://arxiv.org/abs/2207.04237
[3] M. F. Wong, S. Guo, C. N. Hang, S. W. Ho, and C. W. Tan, “Natural Language Generation and Understanding of Big Code for AI-Assisted Programming: A Review,” *Entropy*, vol. 25, no. 6, p. 888, Jun. 2023. Accessed: Dec. 23, 2024. [Online]. Available: https://www.mdpi.com/1099-4300/25/6/888
[4] P. Bhattacharya et al., “Exploring Large Language Models for Code Explanation,” arxiv, Oct. 25, 2023. Accessed: Dec. 23, 2024. [Online]. Available: http://arxiv.org/abs/2310.16673
[5] “What is retrieval-augmented generation, and what does it do for generative AI?,” *The GitHub Blog*, Apr. 4, 2024. Accessed: Dec. 23, 2024. [Online]. Available: https://github.blog/ai-and-ml/generative-ai/what-is-retrieval-augmented-generation-and-what-does-it-do-for-generative-ai/
[6] P. Van de Laar, “TNO/Renaissance-Ada,” TNO, Dec. 8, 2023. Accessed: Feb. 28, 2025. [Online]. Available: https://github.com/TNO/Renaissance-Ada
[7] Z. Liu et al., “BOLAA: Benchmarking and orchestrating LLM-augmented autonomous agents,” arXiv, 2023. Accessed: Dec. 23, 2024. [Online]. Available: https://arxiv.org/abs/2308.05960
[8] D. Zan et al., “CodeS: Natural language to code repository via multi-layer sketch,” arXiv, 2024. Accessed: Dec. 23, 2024. [Online]. Available: http://arxiv.org/abs/2403.16443
[9] J. He, C. Treude, and D. Lo, “LLM-Based Multi-Agent Systems for Software Engineering: Literature Review, Vision and the Road Ahead,” arXiv, 2024. Accessed: Dec. 23, 2024. [Online]. Available: https://arxiv.org/abs/2404.04834
[10] X. Liu et al., “CodexGraph: Bridging large language models and code repositories via code graph databases,” arXiv, 2024. Accessed: Dec. 23, 2024. [Online]. Available: http://arxiv.org/abs/2408.03910
[11] G. Fan, X. Xie, X. Zheng, Y. Liang, and P. Di, “Static Code Analysis in the AI Era: An In-depth Exploration of the Concept, Function, and Potential of Intelligent Code Analysis Agents,” arXiv, 2023. Accessed: Dec. 23, 2024. [Online]. Available: https://arxiv.org/abs/2310.08837
[12] S. Yao et al., “ReAct: Synergizing reasoning and acting in language models,” arXiv, 2023. Accessed: Dec. 23, 2024. [Online]. Available: https://arxiv.org/abs/2210.03629
[13] LangChain, “LangGraph: A framework for building graph-based applications with LLMs,” LangChain. Accessed: Dec. 23, 2024. [Online]. Available: https://www.langchain.com/langgraph
[14] LangChain, “LangGraph Studio: The first agent IDE,” LangChain Blog. Accessed: Dec. 23, 2024. [Online]. Available: https://blog.langchain.dev/langgraph-studio-the-first-agent-ide/
[15] LangChain, “Agentic concepts: React implementation,” LangGraph Docs. Accessed: Dec. 23, 2024. [Online]. Available: https://langchain-ai.github.io/langgraph/concepts/agentic_concepts/%22%20/l%20%22react-implementation
[16] S. Yao, H. Chen, J. Yang, and K. Narasimhan, “WebShop: Towards scalable real-world web interaction with grounded language agents,” arXiv, 2023. Accessed: Dec. 23, 2024. [Online]. Available: https://arxiv.org/abs/2207.01206

[17] S. Qi et al., “HotpotQA: A dataset for diverse, explainable multi-hop question answering,” arXiv, 2018. Accessed: Dec. 23, 2024. [Online]. Available: https://arxiv.org/abs/1809.09600
[18] “F-score,” Wikipedia, Apr. 14, 2025. Accessed: May. 8, 2025. [Online]. Available: https://en.wikipedia.org/wiki/F-score
[19] “Precision and recall,” Wikipedia, Mar. 21, 2024. Accessed: May. 8, 2025. [Online]. Available: https://en.wikipedia.org/wiki/Precision_and_recall
[20] L. Wang et al., “A survey on large language model based autonomous agents,” *Frontiers of Computer Science*, vol. 18, no. 6, Mar. 2024, doi: 10.1007/s11704-024-40231-1. Accessed: Dec. 23, 2024. [Online]. Available: http://dx.doi.org/10.1007/s11704-024-40231-1
[21] Z. Ji et al., “Survey of hallucination in natural language generation,” *ACM Computing Surveys*, vol. 55, no. 12, pp. 1–38, Mar. 2023. Accessed: Dec. 23, 2024. [Online]. Available: http://dx.doi.org/10.1145/3571730
[22] T. Y. Zhuo et al., “On robustness of prompt-based semantic parsing with large pre-trained language model: An empirical study on Codex,” arXiv, 2023. Accessed: Dec. 23, 2024. [Online]. Available: https://arxiv.org/abs/2301.12868
[23] Z. Gekhman et al., “On the robustness of dialogue history representation in conversational question answering: A comprehensive study and a new prompt-based method,” arXiv, 2022. Accessed: Dec. 23, 2024. [Online]. Available: https://arxiv.org/abs/2206.14796
[24] X. Liu et al., “AgentBench: Evaluating LLMs as agents,” arXiv, 2023. Accessed: Dec. 23, 2024. [Online]. Available: https://arxiv.org/abs/2308.03688
[25] Ragas, “Factual correctness,” RagasDocs. Jan. 23, 2025. Accessed: May. 8, 2025. [Online]. Available: https://docs.ragas.io/en/stable/concepts/metrics/available_metrics/factual_correctness/
[26] S. Rukmono, L. Ochoa Venegas, and M. R. V. Chaudron, "Achieving high-level software component summarization via hierarchical chain-of-thought prompting and static code analysis," in *2023 IEEE International Conference on Data and Software Engineering*, ICoDSE 2023, pp. 7–12, Article 10292037, Institute of Electrical and Electronics Engineers, 2023. Accessed: Feb. 28, 2025. [Online]. Available: https://doi.org/10.1109/ICoDSE59534.2023.10292037
[27] “jsoncpp repository,” GitHub. Sep. 12, 2025. Accessed: May 8, 2025. [Online]. Available: https://github.com/open-source-parsers/jsoncpp/tree/master
[28] “Cosine similarity,” Wikipedia, Apr. 27, 2025. Accessed: May 8, 2025. [Online]. Available: https://en.wikipedia.org/wiki/Cosine_similarity
[29] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Proceedings of the Soviet Physics Doklady*, 1966. Accessed: Dec. 23, 2024. [Online]. Available: Binder3.pdf

8 Appendix

Section A: Analysis of questions with an answer correctness score below 1

Answer correctness: functional questions

In Figure 10, question 1, 3 and 5 had a correctness answer score of lower than 1 on average of ten runs.

Question 1:
The correctness answer metric identifies the response as incorrect 10 times, which is accurate upon review. The generated answer includes two unnecessary classes that merely reuse previously mentioned classes, resulting in redundant information being provided.

Question 3:
The correctness metric identifies the response as incorrect due to a missing class in the generated answer. Additionally, the mentions of two classes are inaccurate, indicating that the retrieval process was both incomplete and over-inclusive.

Question 5:
Upon closer inspection, the generated answer may be deemed correct, albeit less detailed than the ground truth. While the ground truth includes an additional detail, the generated answer successfully retrieves the correct information and captures the core aspects of the ground truth.

Answer correctness: structural questions

In Figure 10, question 18 achieved an average correctness score below 1 across ten evaluation runs. The question elicited a lengthy response detailing folder structure, where the orchestrator successfully retrieved the correct information. However, the correctness classification depended heavily on the formatting of the retrieved information. Variations in formatting influenced whether the evaluation LLM judged the response as correct, leading to inconsistencies in its evaluation.

Answer correctness: hybrid questions

In Figure 10, questions 21, 22, 23, 24, 26, 27, and 28 achieved an average correctness score below 1 across ten evaluation runs. A detailed analysis of these cases reveals the following:

Question 21:
The functional part of the question retrieved the correct information; however, the orchestrator did not use it to address this part and instead proceeded directly to the structural component.

Question 22:
The orchestrator retrieved correct information for both the functional and structural aspects. However, the generated answer lacked the detail found in the ground truth in the functional part of the question. A human evaluation would deem the answer correct.

Question 23:
The generated answer described the functionality of a function differently from the ground truth, but not incorrect.

Question 24:
In one instance, the graph search failed to return results because the sub question was overly shortened, leading to inadequate retrieval.

Question 26:
Although the orchestrator retrieved all the necessary information, it did not use this data to fully answer the question, omitting details about the specific source file where certain classes were located.
Question 27:
Incorrect information was retrieved to answer the functional part of the question.
Question 28:
The graph search performed well, but the vector search failed to retrieve the summary for readFromString.cpp because the file contains no code in our graph database, due to its location in an example folder.
Question 29:
The file associated with a variable was not retrieved because the graph search targeted a function node based on the variable name. Variables are no nodes in the graph database and therefore can't be searched on.

Section B: Questions, ground truth and generated answer of the orchestrator that demonstrated performance closest to the median answer correctness score.

Question ID	Question	Ground Truth	Generated answer
0	What does the function deallocate do?	The function deallocate from the SecureAllocator class releases memory which was allocated for `n` items at pointer `p`.	The `deallocate` function is responsible for deallocating memory for a given pointer `p` and size `n`. It first uses `memset_s` to securely set the memory to zero, ensuring that the operation is not optimized away by the compiler. After clearing the memory, it deallocates the memory using the global `operator delete`.
1	Which functions release memory for allocated items?	The functions SecureAllocater::deallocate(), Value::releasePayload(), releaseStringValue() and releasePrefixedStringValue() all release memory for allocated items.	Here are some functions that release memory for allocated items: 1. releasePayload - Summary: The `releasePayload` function is a member of the `Value` class and is responsible for releasing resources based on the type of the value. It uses a switch statement to determine the type of the value, which is obtained by calling the `type` function. For `stringValue`, it checks if the memory is allocated using the `isAllocated` function and releases it using `releasePrefixedStringValue`. For `arrayValue` and `objectValue`, it deletes the associated map. The function ensures that no action is taken for `nullValue`, `intValue`, `uintValue`, `realValue`, and `booleanValue`, and asserts an unreachable state for any other types. 2. releaseStringValue - Summary: The function `releaseStringValue` is a static inline function that takes a `char*` and an unsigned integer as parameters. It releases the memory allocated for the `char*` by calling the `free` function. 3. deallocate - Summary: The `deallocate` function is responsible for deallocating memory for a given pointer `p` and size `n`. It first uses `memset_s` to securely set the memory to zero, ensuring that the operation is not optimized away by the compiler. After clearing the memory, it deallocates the memory using the global `operator delete`. 4. ~CZString - Summary: The destructor `~CZString` is responsible for cleaning up resources associated with a `CZString` object. It checks if the `cstr_` member is not null and if the `storage_.policy_` is set to `duplicate`. If both

			conditions are met, it calls the `releaseStringValue` function to free the memory allocated for the `cstr_`, ensuring proper memory management. 5. ~Value - Summary: The destructor `~Value` is a member of the `Value` class and is responsible for cleaning up resources when a `Value` object is destroyed. It calls the `releasePayload` function to release any resources associated with the value, such as memory for strings or maps for arrays and objects. After releasing the resources, it sets the `uint_` member of the `value_` object to zero, ensuring that the object is left in a clean state. 6. releasePrefixedStringValue - Summary: The `releasePrefixedStringValue` function is a static inline function that takes a pointer to a character array (`char* value`) as its parameter and frees the allocated memory using the `free` function. These functions are responsible for releasing memory and ensuring proper memory management in the codebase.
2	What is the purpose of the class streamWriter?	StreamWriter writes values into the output stream in the form of the JSON value object.	The `StreamWriter` class is used for writing JSON `Value` objects to an output stream. It is part of a system that includes the `StreamWriterBuilder` to configure and create instances of `StreamWriter`, which handle the actual writing process. The class is designed to facilitate the output of JSON data, ensuring that the data adheres to specified configurations for formatting and validation. This functionality is typically demonstrated in the `streamWrite.cpp` source file, which shows how to initialize a JSON object, configure a `StreamWriter`, and output the JSON data.
3	Which classes write values into the output in the form of a JSON value object?	'StreamWriter', 'FastWriter', 'StyledStreamWriter', 'StyledWriter' and 'Writer' are classes that write values in the form of a JSON object.	The classes that write values into the output in the form of a JSON value object are: 1. Value: Provides a comprehensive representation of JSON values, supporting various data types and operations. 2. FastWriter: Used for writing JSON data with options for YAML compatibility and compact output. 3. StreamWriterBuilder: Configures and creates `StreamWriter` objects with customizable settings. 4. StreamWriter: Handles the actual writing process of JSON `Value` objects to an output stream. 5. StyledStreamWriter: Provides methods for serializing JSON data with different styles and configurations. 6. StyledWriter: A subclass of `Writer` that provides functionality to serialize a `Value` into a JSON format string. These classes are utilized in various files such as `stringWrite.cpp`, `json_writer.cpp`, `streamWrite.cpp`, and the `writer.h` header file.
4	What is the role of the parseFromStream function in parsing JSON data?	parseFromStream reads all data from the input stream, uses CharReader to parse the incoming information and then populates a JSON structure with this information.	The `parseFromStream` function plays a crucial role in parsing JSON data from a stream. Here is a detailed explanation of its role: 1. Input Parameters: The function takes a `CharReader::Factory` object, an input stream (`IStream& sin`), a pointer to a `Value` object, and a pointer to a `String` for errors. 2. Reading Stream Data: It reads the entire content of the input stream into an `OStringStream`, converts it to a `String`, and then extracts the data pointers for parsing. 3. Creating CharReader: A `CharReader` is created using the factory provided. 4. Parsing Data: The function calls the `parse` method of the `CharReader`, which processes the data between the `begin` and `end` pointers. This method populates the `root` (the `Value` object) with the parsed JSON data and records any errors in the `errs` string. 5. Return Value: The function returns the result of the `parse` method, indicating whether the parsing was

			successful or not. In summary, the `parseFromStream` function is responsible for converting stream data into a JSON structure by reading the stream, creating a JSON parser, and populating the provided `Value` object with the parsed data while handling any errors that occur during the process.

5	Is there a function which uses CharReader to parse incoming information and then populate a JSON structure with it?	The global function parseFromStream() uses the CharReader object to parse incoming information and populate a JSON structure.	Yes, there is a function named `parseFromStream` that uses `CharReader` to parse incoming information and then populate a JSON structure with it. Here is the code for the `parseFromStream` function:
			```cpp
			bool parseFromStream(CharReader::Factory const& fact, IStream& sin, Value* root, String* errs) { OStringStream ssin ssin << sin.rdbuf() String doc = ssin.str() char const* begin = doc.data() char const* end = begin + doc.size() // Note that we do not actually need a null-terminator. CharReaderPtr const reader(fact.newCharReader()) return reader->parse(begin, end, root, errs) }
			```
			This function reads data from a stream, uses a `CharReader` to parse the data, and populates a `Json::Value` object with the parsed JSON structure.

6	What does function isAllocated do?	isAllocated is a boolean getter function which returns the status of the allocated_ flag for memory management.	The function `isAllocated` is a constant member function that returns a boolean value indicating the allocation status by accessing the `allocated_` member of the `bits_` object.
7	What functions in the database interact with the variable allocated_?	The functions that interact with allocated_ are isAllocated and setIsAllocated.	The functions that interact with the variable `allocated_` are: 1. `setIsAllocated` 2. `isAllocated` These functions are responsible for setting and getting the allocation status of the `allocated_` member.
8	How does the StyledWriter class ensure its output is JSON valid?	The StyledWriter class uses the function write() to create valid JSON objects using the predefined Value object as a structure.	The `StyledWriter` class ensures its output is JSON valid through a series of methods and member variables that manage the serialization process. Here is an overview of how it achieves this: 1. write Method: The `write` method is the main entry point for serializing a `Value` into a JSON format string. It handles the overall structure and ensures that the JSON document is correctly formatted. 2. writeValue Method: This private method is responsible for writing individual JSON values. It ensures that each value is correctly serialized according to its type (object, array, string, number, etc.). 3. writeArrayValue Method: This method specifically handles the serialization of JSON arrays. It ensures that arrays are written in a valid JSON format, either in a single line or across multiple lines, depending on the array's content and configuration. 4. isMultilineArray Method: This method checks whether an array should be written in a multiline format. It helps maintain readability and ensures that large arrays are formatted correctly. 5. writeIndent and writeWithIndent Methods: These methods manage indentation, ensuring that the JSON output is properly indented for readability. Proper indentation is crucial for maintaining the structure of the JSON document. 6. writeCommentBeforeValue and writeCommentAfterValueOnSameLine Methods: These methods handle comments within the JSON structure. They ensure that comments are placed correctly without breaking the JSON format. 7. normalizeEOL Method: This static method normalizes end-of-line characters, ensuring consistency in the

			output. 8. Member Variables: The class uses several member variables to maintain the state during serialization, such as \`childValues_\`, \`document_\`, \`indentString_\`, \`rightMargin_\`, \`indentSize_\`, and \`addChildValues_\`. These variables help manage the document structure, indentation levels, and child values. By combining these methods and member variables, the \`StyledWriter\` class ensures that the output is a valid and well-formatted JSON document.
9	What function selects the greatest integer value the platform running the tool can handle? How does this function do this?	The method asLargestInt() converts the given Json::Value to a 64 bit integer if available or smaller if not.	The function that selects the greatest integer value the platform running the tool can handle is \`asLargestInt\`. This function is a member of the \`Value\` class. Here is how it works: \`\`\`cpp LargestInt Value::asLargestInt() const { #if defined(JSON_NO_INT64) return asInt() #else return asInt64() #endif } \`\`\` ### Explanation: - The function \`asLargestInt\` returns the stored value as the largest integer type available. - It uses conditional compilation to determine whether to return the value as a 32-bit integer (\`asInt()\`) or a 64-bit integer (\`asInt64()\`), depending on whether the macro \`JSON_NO_INT64\` is defined. - If \`JSON_NO_INT64\` is defined, it returns the value as a 32-bit integer using \`asInt()\`. - Otherwise, it returns the value as a 64-bit integer using \`asInt64()\`.
10	If "writer.h" is updated, which directly included files might need to be reviewed or modified?	value.h may need to be updated as it is directly included in the header file writer.h.	If "writer.h" is updated, the directly included file that might need to be reviewed or modified is "value.h".
11	Which header files include "writer.h" and which header files does "writer.h" include?	writer.h is included by files jsontest.h and json.h and writer.h includes value.h.	The header file "writer.h" is included by the following header files: #NAME? #NAME? The header file "writer.h" includes the following header file: #NAME? The full details can be seen if you expand the display table.
12	How are C++ function definitions linked to the header file with the path "/include/json/value.h"?	The function definitions linked to value.h are c_str, setComment, setIsAllocated, back, Comments, StaticString, swap, setType, isAllocated and front.	The C++ function definitions linked to the header file with the path "/include/json/value.h" are as follows: 1. Function Name: \`c_str\` - File Path: \`/include/json/value.h/StaticString.c_str\` - Line Number: 154 2. Function Name: \`setComment\` - File Path: \`/include/json/value.h/Value.setComment\` - Line Number: 567 3. Function Name: \`setIsAllocated\` - File Path: \`/include/json/value.h/Value.setIsAllocated\` - Line Number: 616 4. Function Name: \`back\` - File Path: \`/include/json/value.h/Value.back\` - Line Number: 948 5. Function Name: \`Comments\` - File Path: \`/include/json/value.h/Value.Comments.Comments\` - Line Number: 653 6. Function Name: \`StaticString\` - File Path: \`/include/json/value.h/StaticString.StaticString\` - Line Number: 150 7. Function Name: \`swap\` - File Path: \`/include/json/value.h/swap\` - Line Number: 942 8. Function Name: \`setType\`

			- File Path: `/include/json/value.h/Value.setType` - Line Number: 612 9. Function Name: `isAllocated` - File Path: `/include/json/value.h/Value.isAllocated` - Line Number: 615 10. Function Name: `front` - File Path: `/include/json/value.h/Value.front` - Line Number: 944 The full details can be seen if you expand the display table.
13	Can you identify the static libraries that are generated by projects that another project depends on or references?	Relevant libraries are lib.name "readFromStream.lib", "readFromString.lib", "streamWrite.lib", "stringWrite.lib", "jsontestrunner_exe.lib", "jsoncpp.lib", "jsoncpp_object.lib", "jsoncpp_static.lib" and "jsoncpp_test.lib".	The static libraries that are generated by projects that another project depends on or references are: 1. readFromStream.lib - File Path: unresolved//build/lib/Debug/readFromStream.lib 2. readFromString.lib - File Path: unresolved//build/lib/Debug/readFromString.lib 3. streamWrite.lib - File Path: unresolved//build/lib/Debug/streamWrite.lib 4. stringWrite.lib - File Path: unresolved//build/lib/Debug/stringWrite.lib 5. jsontestrunner_exe.lib - File Path: unresolved//build/lib/Debug/jsontestrunner_exe.lib 6. jsoncpp.lib - File Path: unresolved//build/lib/Debug/jsoncpp.lib 7. jsoncpp_object.lib - File Path: unresolved/jsoncpp_object.dir/Debug/jsoncpp_object.lib 8. jsoncpp_static.lib - File Path: unresolved//build/lib/Debug/jsoncpp_static.lib 9. jsoncpp_test.lib - File Path: unresolved//build/lib/Debug/jsoncpp_test.lib The full details can be seen if you expand the display table.
14	How are the projects linked to the libraries in terms of dependencies? Give me the name of the project and libaries.	Projects that share the lib references relationship are readFromStream.vcxproj and jsoncpp.lib, readFromString.vcxproj and jsoncpp.lib, streamWrite.vcxproj and jsoncpp.lib, stringWrite.vcxproj and jsoncpp.lib, jsontestrunner_exe.vcxproj and jsoncpp.lib, jsoncpp_test.vcxproj and jsoncpp.lib.	The projects are linked to the libraries in terms of dependencies as follows: - Project: readFromStream.vcxproj, Library: jsoncpp.lib - Project: readFromString.vcxproj, Library: jsoncpp.lib - Project: streamWrite.vcxproj, Library: jsoncpp.lib - Project: stringWrite.vcxproj, Library: jsoncpp.lib - Project: jsontestrunner_exe.vcxproj, Library: jsoncpp.lib - Project: jsoncpp_test.vcxproj, Library: jsoncpp.lib The full details can be seen if you expand the display table.
15	Which C or Cpp projects use Assembly to generate the executable named "readFromString.exe" and what are their parent folders?	The project 'readFromString.vcxproj' is stored in the folder '/build/example' and generates readFromString.exe.	The C or Cpp project that uses Assembly to generate the executable named "readFromString.exe" is `readFromString.vcxproj`, and its parent folder is `/build/example`.
16	Identify C++ declarations containing 'include' in their filepath which are	The relevant source file is streamWrite.cpp and the corresponding C++ declarations	Here are the C++ declarations containing 'include' in their filepath which are used by each source file with file path containing '/streamWrite/': 1. Source File: `

	used by each source file with file path containing '/stream-Write/'.	are StreamWriterBuilder, StreamWriter and Value.	/example/streamWrite/streamWrite.cpp` - Declaration Name: `StreamWriterBuilder` - Declaration File Path: `/include/json/writer.h/Stream-WriterBuilder` - Declaration Name: `StreamWriter` - Declaration File Path: `/include/json/writer.h/Stream-Writer` - Declaration Name: `Value` - Declaration File Path: `/include/json/value.h/Value` The full details can be seen if you expand the display table.
17	Can you list all in-stances where a source file, which compiles a C or C++ project, uses the macro definition 'is-nan'?	The are three relevant sets being "jsoncpp_object.vcx-proj" "json_writer.cpp" "is-nan" "jsoncpp_static.vcxproj", "json_writer.cpp" "isnan" and "jsoncpp_lib.vcxproj" "json_writer.cpp" "isnan"	The macro definition `isnan` is used in the following source files and projects: 1. Source File: `json_writer.cpp` - Project: `jsoncpp_object.vcxproj` - Project: `jsoncpp_static.vcxproj` - Project: `jsoncpp_lib.vcxproj` The full details can be seen if you expand the display table.
18	What projects are connected to folders through compliation and nested parent folders?	The following projects are connected to folders through compilation and nested par-ent folders: 1. Project: readFrom-Stream.vcxproj - Fold-ers: - /include/json - /include - / -/example/readFrom-Stream - /example 2. Project: readFrom-String.vcxproj - Folders: - /include/json - /include - / - /example/readFrom-String - /example 3. Project: streamWrite.vcx-proj - Folders: - /include/json - /include - / - /example/streamWrite - /example 4. Project: stringWrite.vcx-proj - Folders: - /include/json - /include - / - /example/stringWrite - /example 5. Project: jsontestrun-ner_exe.vcxproj - Fold-ers: - /include/json - /include - / - /src/jsontestrunner - /src 6. Project: jsoncpp_lib.vcx-proj - Folders: - /src/lib_json - /src - / - /include/json	The following projects are connected to folders through compilation and nested parent folders: 1. Project: `readFromStream.vcxproj` - Folders: - `/include/json` - `/include` - `/` - `/example/readFromStream` - `/example` 2. Project: `readFromString.vcxproj` - Folders: - `/include/json` - `/include` - `/` - `/example/readFromString` - `/example` 3. Project: `streamWrite.vcxproj` - Folders: - `/include/json` - `/include` - `/` - `/example/streamWrite` - `/example` 4. Project: `stringWrite.vcxproj` - Folders: - `/include/json` - `/include` - `/` - `/example/stringWrite` - `/example` 5. Project: `jsontestrunner_exe.vcxproj` - Folders: - `/include/json` - `/include` - `/` - `/src/jsontestrunner` - `/src` 6. Project: `jsoncpp_lib.vcxproj` - Folders: - `/include/json` - `/include` - `/` - `/src/lib_json` - `/src` 7. Project: `jsoncpp_object.vcxproj` - Folders: - `/src/lib_json` - `/src` - `/` - `/include/json` - `/include`

		- /include 7. Project: jsoncpp_object.vcxproj - Folders: - /src/lib_json - /src - / - /include/json - /include 8. Project: jsoncpp_static.vcxproj - Folders: - /include/json - /include - / - /src/lib_json - /src 9. Project: jsoncpp_test.vcxproj - Folders: - /src/test_lib_json - /src - / - /include/json - /include	8. Project: `jsoncpp_static.vcxproj` - Folders: - `/include/json` - `/include` - `/` - `/src/lib_json` - `/src` 9. Project: `jsoncpp_test.vcxproj` - Folders: - `/include/json` - `/include` - `/` - `/src/test_lib_json` - `/src` The full details can be seen if you expand the display table.
19	What is the file path of the function definition that is called the most by other functions in the codebase? How many times is it called?	The most used function in the repository is 'checkEqual' which is called 50 times. It is stored under '/src/test_lib_json/jsontest.h/checkEqual'	The function definition that is called the most by other functions in the codebase is located at the file path `/src/test_lib_json/jsontest.h/checkEqual`. It is called 50 times by other functions.

20	What does the function isAllocated do and what files would be impacted by changing it?	'isAllocated' is a boolean getter function which returns the status of the allocated_ flag for memory management. 'isAllocated' is used by function in 'json_value.cpp' so this should be considered before making changes.	### Function: `isAllocated`
			Definition:
			```cpp
			bool isAllocated() const { return bits_.allocated_ }
			```
			Behavior:
			The function `isAllocated` is a constant member function that returns a boolean value indicating the allocation status by accessing the `allocated_` member of the `bits_` object. ### Impacted Files: Changing the function `isAllocated` would impact the following file:
			- `/src/lib_json/json_value.cpp`
			The full details can be seen if you expand the display table.

21	What does the function removeSuffix do and what effect would changing it have on other functions?	'removeSuffix' takes two parameters, a path and an extension. If the extension is present in the path the path is returned with the extension removed. 'removeSuffix is used by 'runTest' therefore this function should be inspected.	### Functionality of `removeSuffix` The `removeSuffix` function is a static function that takes two `Json::String` arguments, `path` and `extension`. It performs the following steps:
			1. Checks if the length of the `extension` is greater than or equal to the length of the `path`. If true, it returns an empty `Json::String`.
			2. Extracts a suffix from the `path` that is the same length as the `extension`.
			3. Compares the extracted suffix to the `extension`. If they do not match, it returns an empty `Json::String`.
			4. If they match, it returns the `path` without the `extension` suffix.
			Here is the code for the `removeSuffix` function:
			```
			cpp
			static Json::String removeSuffix(const Json::String& path, const Json::String& extension) { if (extension.length() >= path.length()) return Json::String("") Json::String suffix = path.substr(path.length() - extension.length()) if (suffix !=
			```

			extension) return Json::String("") return path.substr(0, path.length() - extension.length()) } ``` ### Effect of Changing `removeSuffix` Changing the `removeSuffix` function would affect the `runTest` function, as it is the function that calls `removeSuffix`. The full details can be seen if you expand the display table.
22	What does header-file json.h do and what files would be affected by changing it?	'json.h' is a header file from jsoncpp and includes all core components, like config, features, reader, value and writer, required to work with JSON data. Users only need to include this file to access all the functionality of jsoncpp. Changing the file to remove any of these components would greatly affect the rest of the program. It is included in 'jsontestrunner/main.cpp', 'streamWrite.cpp', 'readFromString.cpp', 'stringWrite.cpp','readFromStream.cpp', 'test_lib_json/main.cpp' and fuzz.cpp	### Summary of `json.h`: The `json.h` header file is part of a JSON library that includes several header files defining classes and functions for JSON parsing, serialization, and memory management. It provides a comprehensive set of tools for handling JSON data, including secure memory management, flexible parsing options, robust data manipulation, and versatile serialization capabilities. ### Files Affected by Changes to `json.h`: The following files would be affected by changes to the `json.h` header file: 1. Source Files: - `/src/jsontestrunner/main.cpp` - `/example/readFromStream/readFromStream.cpp` - `/example/streamWrite/streamWrite.cpp` - `/src/test_lib_json/main.cpp` - `/src/test_lib_json/fuzz.cpp` - `/example/readFromString/readFromString.cpp` - `/example/stringWrite/stringWrite.cpp` These files include the `json.h` header file and would be impacted by any modifications made to it.
23	What is the file path of the function that has the highest number of calls in the codebase and what does this function do?	The function 'checkEqual' from '/src/test_lib_json/jsontest.h' has the highest number of calls in the codebase. It compares an expected value against an actual value, and if they don't match, it records the failure details (including file, line number and mismatched value).	The function with the highest number of calls in the codebase is `checkEqual`, and it is located in the file `/src/test_lib_json/jsontest.h`. ### Function Summary: The `checkIsEqual` function is a member of the `ValueTest` class and is designed to verify the equality of two `Json::Value` objects, `x` and `y`. It uses a series of assertions to ensure that the two values are equal in terms of equality (`==`), less than or equal (`<=`), greater than or equal (`>=`), and not less than (`<`) or greater than (`>`). Additionally, it checks that the `compare` function of the `Value` class returns 0, indicating that the two values are equal. The function relies on the `JSONTEST_ASSERT` macro to perform these checks, ensuring that all conditions for equality are met.
24	Which function releases memory for allocated items in the Value class and what other functions are in this class?	The function that releases memory for allocated items is `releasePayload`, and it is a member of the `Value` class. Other functions in the `Value` class include: - `isAllocated` - `setComment` - `setIsAllocated` - `setType` The full details can be seen if you expand the display table.	The function that releases memory for allocated items in the `Value` class is `releasePayload`. Here is the code for the `releasePayload` function: ```cpp void Value::releasePayload() { switch (type()) { case nullValue: case intValue: case uintValue: case realValue: case booleanValue: break case stringValue: if (isAllocated()) releasePrefixedStringValue(value_.string_) break case arrayValue: case objectValue: delete value_.map_ break default: JSON_ASSERT_UNREACHABLE } } ``` Other functions in the `Value` class include: - `isAllocated` - `setComment` - `setIsAllocated` - `setType` The full details can be seen if you expand the display table.
25	What would be the impact of removing the ValueType enum	Removing the ValueType enum from value.h would break the core type system of	The `ValueType` enum in `value.h` defines the fundamental JSON data types such as null, int, uint, real, string, boolean, array, and object. Removing this enum

	that defines the fundamental JSON data types (null, int, uint, real, string, boolean, array, object) from value.h? How would this affect the functionality of the JsonCpp library?	JsonCpp as the Value class relies on these enums for all type checking and data handling operations. Since value.h is included by writer.h, this would also break JSON serialization capabilities as the writer would have no way to determine what type of value it's writing to JSON format.	would have significant impacts on the functionality of the JsonCpp library. Here are the potential effects: 1. Type Identification: The `ValueType` enum is used to identify the type of JSON values. Without it, the library would lose the ability to distinguish between different JSON data types, which is crucial for parsing, manipulating, and validating JSON data. 2. Type Checking and Conversion: Many functions within the `Value` class and related classes rely on `ValueType` for type checking and conversion. Removing the enum would break these functions, leading to errors in type handling and conversion operations. 3. Iterators and Utility Functions: Iterators like `ValueIterator` and `ValueConstIterator`, as well as various utility functions for type checking and conversion, depend on `ValueType`. Their functionality would be compromised, affecting the traversal and manipulation of JSON arrays and objects. 4. Memory Management: Internal structures that manage memory allocation and type information, such as `ValueHolder`, use `ValueType` to store and retrieve the correct data types. Removing the enum would disrupt memory management, potentially causing memory leaks or corruption. 5. Error Handling: Functions that handle errors and exceptions related to JSON data types would lose their ability to provide meaningful error messages, making debugging and error resolution more difficult. 6. Backward Compatibility: Removing `ValueType` would break backward compatibility with existing code that relies on the JsonCpp library. Users of the library would need to refactor their code to accommodate the changes, which could be a significant effort. In summary, removing the `ValueType` enum from `value.h` would severely impact the core functionality of the JsonCpp library, affecting type identification, type checking and conversion, iterators, utility functions, memory management, error handling, and backward compatibility.
26	Identify C++ declarations containing 'include' in their file path which are used by each source file with file path containing 'streamWrite'. What do these declarations do?	The relevant source file is streamWrite.cpp and the corresponding C++ declarations are StreamWriterBuilder, StreamWriter and Value. StreamWriterBuilder is a factory class that constructs and configures StreamWriter instances. StreamWriter is a class that defines an interface for writing JSON value objects to output streams. Value is a class which can represent a variety of json types such as null, boolean, number, string, array or object), it provides type checking, conversion methods and access to nested elements through array indices.	The C++ declarations containing 'include' in their file path which are used by the source file with file path containing 'streamWrite' are: 1. StreamWriterBuilder in `/include/json/writer.h/StreamWriterBuilder` 2. StreamWriter in `/include/json/writer.h/StreamWriter` 3. Value in `/include/json/value.h/Value` ### What these declarations do: 1. StreamWriterBuilder: - File: `/include/json/writer.h/StreamWriterBuilder` - Summary: The `StreamWriterBuilder` class provides configuration and creation of `StreamWriter` instances. It initializes with default settings for JSON stream writing and allows customization of these settings. The `newStreamWriter` function within this class is responsible for creating a new `StreamWriter` object with the specified settings, ensuring proper validation and configuration. 2. StreamWriter: - File: `/include/json/writer.h/StreamWriter` - Summary: The `StreamWriter` class is an abstract class that defines the interface for converting `Value` objects into strings or writing them to streams. It is part of the JSON serialization process, providing methods for writing JSON data with different styles and configurations. 3. Value: - File: `/include/json/value.h/Value`

			- Summary: The `Value` class represents a JSON value, which can be of various types such as object, array, string, number, boolean, or null. It provides methods for manipulating and accessing JSON data, serving as the core data structure for JSON parsing and serialization. These declarations are part of a JSON handling library, focusing on the serialization and formatting of JSON data. The `StreamWriterBuilder` and `StreamWriter` classes are used to configure and write JSON `Value` objects to output streams, ensuring the JSON output adheres to specified configurations.
27	Can you list all instances where a source file, which compiles a C or C++ project, uses the macro definition 'isnan'. Provide the source files and the C or C++ projects. Explain how these source files use this macro?	There are three relevant sets being "jsoncpp_object.vcxproj", "json_writer.cpp", "isnan", "jsoncpp_static.vcxproj", "json_writer.cpp", "isnan" and "json_lib.vcxproj", "json_writer.cpp" and "isnan". The macro is instigated and logs a bug message when an error occurs - in this cases when isnan is true.	I am unable to retrieve the code directly for the file `json_writer.cpp`. However, based on the information provided, the `isnan` macro is used in the `json_writer.cpp` file within the following projects: - `jsoncpp_object.vcxproj` - `jsoncpp_static.vcxproj` - `jsoncpp_lib.vcxproj` To understand how the `isnan` macro is used in these source files, you can manually check the `json_writer.cpp` file in the mentioned projects. The `isnan` macro is typically used to check if a floating-point number is "Not-a-Number" (NaN), which is a common operation when dealing with JSON data that may include special float values. If you need further assistance or specific details about the usage, you may need to access the source code repository where these projects are maintained.
28	Which C or Cpp projects use Assembly to generate the executable named "readFromString.exe" and what are their parent folders? Considering the relevant cpp file what is the behaviour of this executable?	The C or Cpp project that uses Assembly to generate the executable named "readFromString.exe" is `readFromString.vcxproj`. The parent folder of this project is `/build/example` Based on readFromString.cpp this example parses a JSON string containing name and age fields and prints the extracted values to the console.	### Projects and Parent Folders The C or C++ project that uses Assembly to generate the executable named `readFromString.exe` is: - Project Name: `readFromString.vcxproj` - Parent Folder: `/build/example` ### Behaviour of the Executable The `readFromString.exe` executable is part of the `readFromString.vcxproj` project, which focuses on reading and parsing JSON strings using the JsonCpp library. The project includes a source file, `readFromString.cpp`, that demonstrates the JSON parsing process with the `CharReader` and `CharReaderBuilder` classes, while also handling potential errors. Additionally, the project incorporates several header files for secure memory management, JSON parsing configuration, JSON reading and error handling, JSON data representation and manipulation, and JSON serialization. These components work together to provide comprehensive JSON operations, including parsing, error handling, memory management, and serialization.
29	What functions in the database interact with the variable allocated_ and in what file are these functions stored?	The functions isAllocated and setIsAllocated from '/include/json/value.h' interact with the variable allocated_.	The functions that interact with the variable `allocated_` are: 1. `setIsAllocated` 2. `isAllocated` These functions are stored in the following files: - `setIsAllocated` is stored in the file: `bits_` - `isAllocated` is stored in the file: `bits_` The full details can be seen if you expand the display table.

ICT, Strategy & Policy

www.tno.nl

TNO innovation for life