Mälardalen University
Department of Computer Science and Engineering
Västerås, Sweden

---

Thesis for the degree of Master of Science in Computer Science with Specialization in Software Engineering - 15 credits

# AUTOMATED C4 DIAGRAM GENERATION FROM SOURCE CODE: STATIC-ANALYSIS GROUNDING AND THE CONSISTENCY OF LARGE LANGUAGE MODEL OUTPUT

Arben Dokovic and Balsa Rakocevic
adc25002@student.mdu.se and brc25001@student.mdu.se

Examiner: Antonio Cicchetti
Mälardalen University, Västerås, Sweden

Supervisor: Malvina Latifaj
Mälardalen University, Västerås, Sweden

11 October 2026

# Table of Contents

**1 Introduction** **2**

**2 Background and Related Work** **4**
2.1 Software Architecture Recovery and Reconstruction . . . . . 4
2.2 Software Reverse Engineering from Code to Models . . . . . 4
2.3 Static Analysis and Large Language Models for Model Extraction . . . . . 4
2.4 Hybrid Approaches Combining Static Analysis and LLMs . . . . . 5
2.5 Summary and Research Gap . . . . . 5

**3 Problem Formulation** **6**

**4 Research Methodology** **7**
4.1 Design Science Research for Pipeline Development (RQ1) . . . . . 7
4.2 Controlled Repeated-Run Evaluation for Consistency (RQ2) . . . . . 7

**5 Pipeline Design and Implementation** **9**
5.1 Static-Analysis Extraction . . . . . 9
5.1.1 Information Requirements for C4 Diagrams . . . . . 9
5.1.2 Tool Selection and Justification . . . . . 9
5.1.3 Extracted Information . . . . . 10
5.1.4 Output Filtering . . . . . 11
5.2 Transformation into LLM Input . . . . . 11
5.2.1 Choice of Representation . . . . . 11
5.2.2 Package and Class Summaries . . . . . 11
5.2.3 Source-Code Input . . . . . 12
5.2.4 Differences Between C2 and C3 . . . . . 12
5.2.5 Target-Container Selection for C3 . . . . . 12
5.3 Prompt Construction . . . . . 13
5.3.1 C2/C3 Instructions and Output Format . . . . . 13
5.3.2 Differences Between the Templates . . . . . 13
5.3.3 Combining Source Code and Static Analysis . . . . . 13
5.4 LLM Invocation and Output Processing . . . . . 13
5.5 Worked Example . . . . . 14

**6 Evaluation Protocol** **17**
6.1 Study Factors and Variables . . . . . 17
6.1.1 Factors . . . . . 17
6.1.2 Controlled Variables . . . . . 17
6.1.3 Dependent Variables . . . . . 18
6.2 LLM Selection and Configuration . . . . . 18
6.2.1 Model Selection and Justification . . . . . 18
6.2.2 API Parameter Configuration . . . . . 18
6.3 Target Projects . . . . . 19
6.3.1 Rationale for Restricting to the Java Ecosystem . . . . . 19
6.3.2 Selection Criteria . . . . . 19
6.3.3 Selected Projects . . . . . 19
6.3.4 Container Selection for C3 Analysis . . . . . 19
6.4 Study Execution . . . . . 20
6.4.1 Factorial Structure and Total Conditions . . . . . 20
6.4.2 Number of Repetitions . . . . . 20
6.4.3 Execution Procedure . . . . . 20
6.5 Metrics and Measurement . . . . . 20
6.5.1 Syntactic Validity . . . . . 20
6.5.2 From a Diagram to Nodes and Edges . . . . . 20
6.5.3 Label Alignment . . . . . 21

6.5.4 Output Consistency . . . 22
6.5.5 Number of Distinct Diagrams . . . 22
6.5.6 Within-Diagram Repetition Ratio . . . 22
6.5.7 Aggregation and Statistical Analysis . . . 22

**7 Results** **24**
7.1 Syntactic Validity . . . 24
7.2 Main Comparison: Code-Only versus Combined Input . . . 24
7.3 Consistency by Model . . . 25
7.4 Consistency by Project . . . 26
7.5 Number of Distinct Diagrams . . . 26
7.6 Repetitive Outputs . . . 26
7.7 Effect of Label Alignment . . . 27
7.8 Robustness to the Treatment of Undeclared Elements . . . 28
7.9 Data and Code Availability . . . 28

**8 Discussion** **29**
8.1 Addressing RQ1: Pipeline Design and Operation . . . 29
8.2 Addressing RQ2: Effect of Static Analysis on Consistency . . . 29
8.3 Variation Is Structural, Not Terminological . . . 30
8.4 Differences Between the Models . . . 30
8.5 Repetitive Outputs as a Separate Quality Problem . . . 31
8.6 Consistency Does Not Establish Architectural Correctness . . . 31
8.7 Correctness as Future Work . . . 31

**9 Threats to Validity and Limitations** **32**
9.1 Construct Validity . . . 32
9.2 Internal Validity . . . 32
9.3 External Validity . . . 32
9.4 Conclusion Validity . . . 33

**10 Conclusions and Future Work** **34**

**References** **37**

**Appendix A Complete Prompt Templates** **38**
A.1 Source-Code-Only, C2 . . . 38
A.2 Source-Code-Only, C3 . . . 39
A.3 Static-Analysis-Only, C2 . . . 40
A.4 Static-Analysis-Only, C3 . . . 40
A.5 Combined, C2 . . . 41
A.6 Combined, C3 . . . 42

**Abstract**

*Software architectural documentation is frequently missing or outdated, and large language models (LLMs) are increasingly proposed as a way to recover architectural views from source code. A practical obstacle is that LLM output is not deterministic: repeated generation from the same input can yield different diagrams, even at temperature 0. This thesis studies this problem for the Container (C2) and Component (C3) levels of the C4 model, and investigates whether providing structural evidence extracted by static analysis changes how consistent LLM-generated diagrams are across repeated runs. The study measures consistency only; it does not assess whether the diagrams are architecturally correct.*

*A five-stage pipeline (static-analysis extraction with Spoon, transformation into LLM input, prompt construction, LLM invocation, and output processing) was implemented and applied to three open-source Java projects of increasing size (jhipster ≈4,500 LOC, spring-boot-admin ≈15,300 LOC, apache-maven-3.9.15 ≈84,000 LOC). Three models (GPT-4.1 mini, GPT-4.1 nano, Gemini 2.5 Flash) were each given three input conditions: source code only, static analysis output only, and both combined. Every condition was run ten times, yielding 810 PlantUML diagrams. Consistency was measured as node and edge Jaccard similarity over all pairs of runs, after an automated, deterministic label-alignment procedure. Differences between conditions were tested with Wilcoxon signed-rank tests over model × project × target cells, with Holm correction.*

*All 810 diagrams were syntactically valid PlantUML. Repeated generation was not deterministic: only 4 of 27 C2 conditions produced the same diagram in all ten runs. The combined condition had the highest mean C2 consistency (node Jaccard 0.83 versus 0.78 for source code only; edge Jaccard 0.71 versus 0.60), but none of the differences was statistically significant after Holm correction for multiple comparisons, and the effect of adding static analysis was positive for one model, negative for another and negligible for the third. At C3, consistency was high in all conditions (node Jaccard 0.92–0.97); however, for two of the three projects every run of every model produced an identical set of components, so the C3 differences stem almost entirely from the largest project. Gemini 2.5 Flash produced the most stable output overall and GPT-4.1 nano the least. Label normalisation changed the measured consistency by at most 0.032, which indicates that run-to-run variation is structural rather than terminological. The lowest consistency values were concentrated in outputs with large numbers of repeated declarations. Overall, adding static analysis did not reliably make LLM-generated architectural views more stable.*

***Keywords:*** *software architecture recovery, C4 model, static analysis, large language models, LLM non-determinism, output consistency, architectural views, PlantUML, Spoon, empirical evaluation*

# 1. Introduction

Software systems are continuously increasing in size and complexity. Because of that, it is often difficult for developers to understand the overall structure of a system, especially when they are joining a new project or working with code they are unfamiliar with. In modern software development, systems are constructed using a variety of artefacts such as documentation, architectural models, configuration files, frameworks, and code-generation tools. While these artefacts contribute to system development, they may be incomplete or become inconsistent over time. Regardless of whether the code is written manually or generated automatically, the source code is what is ultimately executed, and it therefore represents the primary and authoritative description of the system [1].

Although source code constitutes the ground truth of a software system, it is not well suited for direct human comprehension at the architectural level. Source code exposes fine-grained implementation details, while developers often require higher-level abstractions to understand system structure, responsibilities, and dependencies [2]. For this reason, architectural documentation and architectural views are commonly used to support tasks such as onboarding, change impact analysis, refactoring, and technical debt management [3]. However, architectural documentation is frequently missing, outdated, or inconsistent with the current implementation, largely because it is created and maintained manually and tends to diverge from the evolving codebase over time. This problem is further amplified by the increasing use of large language models (LLMs) to generate source code directly: as a growing share of code is produced by AI coding assistants rather than written by developers, documentation derived from that code becomes important not only for communicating a system's design, but also for understanding and reviewing what was actually produced.

A wide range of approaches for architectural representation and recovery have been proposed to address this challenge. These include viewpoint-based frameworks, such as the 4+1 view model, which define which architectural views should be documented; standardised modelling notations, such as the Unified Modeling Language (UML), which define how architectural elements are represented; and architecture recovery techniques, such as static analysis, which derive architectural information from existing systems [1]. Representation frameworks and notations provide conceptual structures for describing an architecture, but their practical usefulness depends on the availability of accurate input. Architecture recovery techniques, on the other hand, can derive up-to-date structural information directly from the implemented system.

Static analysis extracts structural information from source code without executing the system. Depending on the type of analysis, it can reveal elements such as modules, packages, classes, and their static dependencies. Because static analysis is grounded in the implemented code, it provides an objective and reproducible view of these structural aspects: running the same analysis on the same code always yields the same result. However, its output is low-level and large; dependency graphs of real systems quickly become visually complex and difficult to read as a whole.

The C4 model [4] addresses the readability problem from the notation side. It organises the structure of a system into four hierarchical levels with increasing detail (system context, containers, components, and code), uses a small, fixed vocabulary of element types, and is intended to be read by developers without training in a formal modelling language. This makes it a natural target format when the goal is a view that a developer can read, rather than a complete dependency graph.

This thesis targets the Container (C2) and Component (C3) levels. The system-context level (C1) is excluded because it depends largely on information that is not present in the source code, such as users and external systems. The code level (C4) is excluded because it is close to a class diagram and adds little over what static analysis already provides; in practice, architectural views at this level of detail are also rarely created: an empirical study of open-source projects found that only 9% of the analysed architectural views describe the system at a low level close to the code [5], and a recent study on LLM-based generation of C4 views likewise omits the code level [6]. C2 and C3 are therefore the levels at which a structural summary of the code is both possible and useful.

Large language models are able to summarise and transform technical artefacts, which makes them a candidate tool for turning source code or static-analysis output into such views. Two properties, however, limit their use for this purpose. First, an LLM is not grounded in the analysed

system and may introduce elements or relationships that do not exist. Second, and the focus of this thesis, LLM output is not deterministic: repeated requests with the same input can return different outputs, even when sampling temperature is set to zero and nothing about the system has changed. For architecture recovery this is a practical problem. A diagram that changes every time it is generated cannot serve as a stable reference for developers, cannot be compared meaningfully across versions of a system, and cannot be reproduced by another person, regardless of how plausible any single output looks. Whether such variation can be reduced, and by which input, is therefore an open question that has to be answered before the correctness of LLM-generated views can even be assessed.

The scope of the study is deliberately limited. The generated diagrams show only structure: which elements exist and how they depend on each other. They do not explain why the system was designed that way. The study measures *consistency* only. It does not evaluate whether the generated diagrams are architecturally correct or faithful to the codebase, which would require a reference architecture for each project and is left for future work. In this study, a container is represented by a group of Java packages, because the source code of a single repository does not clearly define which parts are deployed separately; the consequences of this choice are discussed in Section 9.

The remainder of this thesis is structured as follows. Section 2 describes the background and related work. Section 3 formulates the research problem and states the research questions. Section 4 describes the research methodology, which combines design science research for the construction of the pipeline with a controlled, repeated-run empirical study for its evaluation. Section 5 presents the design and implementation of the pipeline. Section 6 describes the evaluation protocol, including the target projects, models, factors, and metrics. Section 7 reports the results. Section 8 interprets them in relation to the research questions. Section 9 discusses threats to validity and limitations. Finally, Section 10 concludes the thesis and outlines directions for future work.

# 2. Background and Related Work

Recovering software architecture from source code is a long-standing problem in software engineering, motivated by the need to understand, maintain, and evolve systems whose documentation is missing or out of date. This section reviews related work in four areas: software architecture recovery and reconstruction, reverse engineering from code to models, static analysis and large language models for model extraction, and hybrid approaches that combine the two. For each area, the main contributions and limitations are discussed, with particular attention to whether the reproducibility of the generated output has been studied.

## 2.1 Software Architecture Recovery and Reconstruction

Software architecture recovery reconstructs architectural structures and views from existing software artefacts, most commonly source code. Many approaches extract structural dependencies between software entities and group them into higher-level elements using clustering and visualisation techniques [7]. These methods are effective at exposing subsystem boundaries and dependency relationships, but their output usually has to be interpreted manually and is not presented in a standard view format.

Other approaches recover architectural models through reverse engineering pipelines that transform source code into higher-level representations such as UML models [8]. These techniques improve readability, but the recovered models often remain close to the low-level code structure.

Overall, existing architecture recovery approaches extract structural information reliably, but the result is usually a large dependency structure rather than a compact view in a standard notation that a developer can read directly. Turning that structure into such a view is the step that this thesis automates with an LLM. The views produced in this work are still structural: they show elements and their dependencies, not design decisions or the reasons behind them.

## 2.2 Software Reverse Engineering from Code to Models

Software reverse engineering transforms source code into higher-level models that support system understanding and maintenance. Model-driven reverse engineering (MDRE) approaches dominate this area; they rely on static analysis and model transformations to generate UML artefacts such as class, component, and sequence diagrams [9, 10].

Systematic literature reviews show that these approaches provide formal structure and traceability between code and models, but they cannot capture abstractions that are not explicitly represented in the source code [9]. Many of them also require substantial manual configuration and tool-specific expertise, which limits their scalability and industrial use [10].

These limitations motivate the investigation of more automated and flexible techniques for reverse engineering.

## 2.3 Static Analysis and Large Language Models for Model Extraction

Static analysis is a fundamental technique for reverse engineering. It extracts structural information from source code precisely and reproducibly: the same code always produces the same result. Its output, however, is a detailed list of elements and dependencies, which is too large to serve directly as an architectural view.

Recent research uses large language models to generate models directly from source code. Studies show that LLMs can extract UML class diagrams, including classes, attributes, and relationships, with relatively low manual effort [11, 12]. A follow-up empirical study comparing five LLMs (StarCoder2, LLaMA, CodeLlama, Mistral, and DeepSeek) on UML class diagram extraction from Java and Python code found substantial differences in accuracy between models, with DeepSeek and Mistral performing best and LLaMA the weakest across both languages and all metrics [13]. Comparative evaluations indicate that LLM-based approaches can produce more abstract and readable models than traditional model-driven techniques [11].

LLM-based approaches also have known limitations: they may produce model elements that do not exist in the code, miss constraints, and are not reproducible, because repeated generation from the same input can give different results [12, 14]. None of these studies, however, repeats

the same generation task several times with a fixed input to measure this variation directly; it is reported as a limitation rather than evaluated. In addition, most existing work targets class-level models rather than architectural views.

The work closest to this thesis in its use of the C4 model is that of Szczepanik and Chudziak [6], who use a multi-agent LLM system to generate the Context, Container, and Component views of the C4 model. Their input, however, is a textual description of system requirements rather than source code, and the generated views are assessed with structural and syntactic checks and an LLM-as-a-judge score of a single output. Whether the same input leads to the same views across repeated runs is not studied. This thesis differs in both respects: the C2 and C3 views are generated from existing source code and static-analysis output, and their consistency across repeated runs is measured.

## 2.4 Hybrid Approaches Combining Static Analysis and LLMs

To address the complementary weaknesses of static analysis and LLM-based techniques, recent work investigates hybrid approaches. These typically use static analysis to extract reliable structural information, which is then summarised by an LLM into a higher-level architectural description [15, 16].

Results indicate that hybrid techniques can improve the quality of the abstraction and reduce manual effort compared to purely static or purely LLM-based approaches [15]. Existing hybrid solutions, however, usually produce a single architectural representation or textual description rather than views at several levels of a standard model [16].

More importantly for this thesis, none of the reviewed hybrid approaches evaluates whether the generated view stays the same when the same input is processed several times under identical conditions. It is therefore unknown whether adding static-analysis evidence makes the output more stable, or only improves the perceived quality of a single output.

## 2.5 Summary and Research Gap

The reviewed literature shows a clear trend toward more automation in architecture recovery. Static analysis and MDRE techniques are precise and reproducible but produce low-level output that is hard to read as an architectural view. LLM-based approaches produce more readable models but are not reproducible. Hybrid approaches combine both, but have so far been evaluated on the quality of a single output. None of the reviewed studies, whether based on static analysis, LLMs, or a combination, systematically measures whether a generated architectural view remains the same across repeated generations from the same input.

This thesis addresses this gap. It does not propose a new extraction technique and does not evaluate the correctness of the generated views. Instead, it measures how consistent LLM-generated Container (C2) and Component (C3) diagrams are across repeated runs, and whether this consistency changes when the LLM receives static-analysis evidence instead of, or in addition to, the source code, across different models and target projects.

# 3. Problem Formulation

Static analysis extracts structural information from source code deterministically: the same code always yields the same result. Its output, however, is a detailed list of elements and dependencies that has to be further abstracted before it can serve as a readable architectural view. Large language models can perform this abstraction directly, but they are not deterministic: repeated generation from the same input can produce structurally different diagrams, even though nothing about the underlying system has changed. A view that changes every time it is generated cannot serve as a stable reference for developers.

The research problem addressed in this thesis is therefore the run-to-run variability of LLM-generated architectural views, and whether this variability depends on the kind of input the LLM receives. As a concrete focus, the thesis targets **Container (C2) and Component (C3) diagrams** of the C4 model (Section 1). The goal is to design and implement an automated pipeline that generates such diagrams from three alternative inputs (source code only, static-analysis output only, and both combined) and to measure how consistent the generated diagrams are across repeated runs under each input. The study measures consistency only; the correctness of the generated diagrams is outside its scope.

Based on this, the thesis addresses the following research questions:

1. **How can static-analysis information be structured and combined with a large language model to automatically generate Container (C2) and Component (C3) diagrams from source code?**
   This question concerns the design and construction of the pipeline: which structural information is extracted from the codebase, how it is filtered and formatted as input for the LLM, and how the prompt and the output processing are organised so that each run produces a C2 or C3 diagram in PlantUML.

2. **To what extent does providing static-analysis output to a large language model affect the consistency of generated C2 and C3 diagrams across repeated runs, compared to providing source code alone?**
   This question investigates whether grounding the LLM with static-analysis data changes output variability across repeated runs under identical conditions. Consistency is examined by comparing the structural similarity of diagrams generated under different input conditions — source code only versus static analysis combined with source code — across multiple LLMs and projects of varying size and complexity.

# 4. Research Methodology

This thesis addresses two research questions that require different methodological treatments. RQ1 concerns the design and construction of an automated pipeline, while RQ2 concerns the empirical evaluation of that pipeline's behaviour across repeated executions. Accordingly, the overall research approach combines two complementary methodologies: Design Science Research (DSR) for the design and construction of the pipeline artefact (RQ1), followed by a controlled repeated-run empirical evaluation of that artefact (RQ2).

## 4.1 Design Science Research for Pipeline Development (RQ1)

The design and construction of the automated generation pipeline is guided by a Design Science Research (DSR) methodology, which is appropriate for studies that focus on the design, construction, and evaluation of an artefact intended to address a specific problem [17]. In this work, the artefact is an automated pipeline for generating high-level architectural views from source code using static analysis and large language models. The specific research process followed is based on the Design Science Research Methodology [18], comprising the following activities:

- **Problem Identification and Motivation.** The research is motivated by the difficulty of obtaining up-to-date, high-level architectural views of software systems from existing codebases. Architectural documentation is often missing or outdated, while available static-analysis results are typically low-level and difficult to interpret at an architectural level. Furthermore, LLMs used for architectural view generation exhibit non-deterministic behaviour, producing structurally different outputs across repeated runs under identical conditions, which undermines the reliability of generated views.
- **Objectives of a Solution.** The objective is to design an automated approach that generates Container (C2) and Component (C3) diagrams in the C4 notation from source code, static-analysis output, or both, and to evaluate whether grounding the LLM with structured static-analysis output affects the consistency of the generated diagrams across repeated runs. Assessing the architectural correctness of the diagrams is not an objective of this work.
- **Design and Construction.** The core artefact is an automated generation pipeline. It was developed in a small number of iterations: early versions often produced PlantUML that could not be rendered, which led to the explicit syntax rules and the valid example diagram now included in every prompt template. The final design is described in Section 5, which directly addresses RQ1.
- **Demonstration.** The proposed approach is demonstrated by applying the pipeline to three open-source Java projects. A complete example, from a source-code excerpt to the generated diagram, is given in Section 5.
- **Evaluation.** The pipeline is evaluated through a controlled empirical study, described in Section 6, which directly addresses RQ2.

In this structure, the pipeline keeps the structural evidence obtained from static analysis separate from the abstraction performed by the language model, so that the effect of each input on the generated diagrams can be studied in isolation.

## 4.2 Controlled Repeated-Run Evaluation for Consistency (RQ2)

The pipeline produced by the DSR process is evaluated through a controlled empirical study following established guidelines for experimentation in software engineering [19]. This framework provides the structure for factor identification, treatment design, outcome measurement, and validity analysis used in the study. The reporting structure follows established guidelines for reporting empirical studies in software engineering [20].

Rather than a controlled experiment in the strict sense, which requires random assignment of treatments to subjects, this is a controlled empirical study in which conditions are systematically

varied and outcomes are measured [19]. The study investigates whether providing structured static-analysis output to the LLM affects the consistency of generated diagrams across repeated runs, compared to providing source code alone. The full evaluation protocol, including factors, controlled variables, target projects, and metrics, is described in Section 6.

Together, these two methodological components ensure that RQ1 is addressed through a systematic design-and-construction process with a demonstrable artefact, and that RQ2 is addressed through a reproducible, factor-based empirical evaluation of that artefact's output behaviour.

# 5. Pipeline Design and Implementation

This section describes the design and implementation of the automated generation pipeline, directly addressing RQ1: how static-analysis information can be structured and combined with a large language model to automatically generate Container (C2) and Component (C3) diagrams from source code. The section distinguishes explicitly between components that were implemented as part of this work and external tools or services that were used as-is. A high-level overview of the pipeline is shown in Figure 1, and a complete worked example is given in Section 5.5.

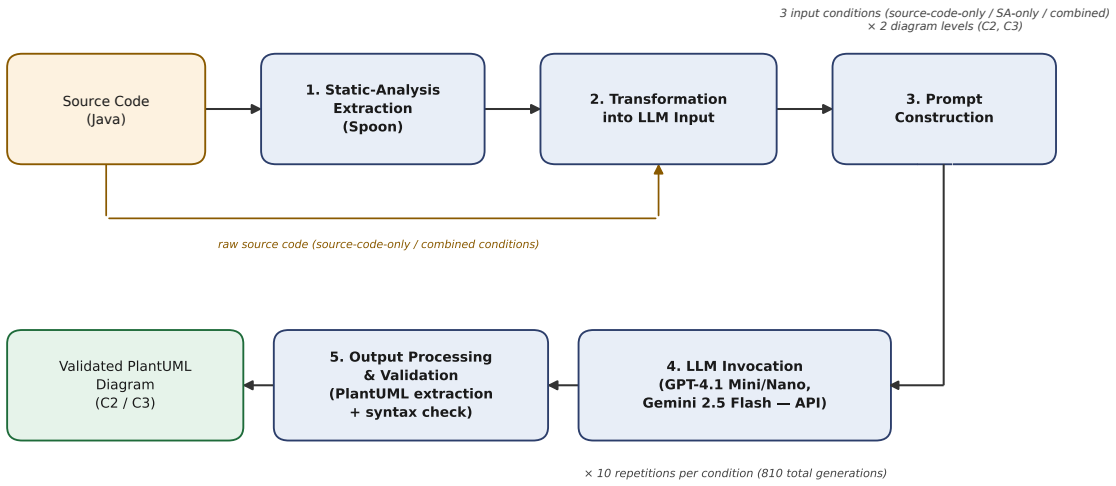


Figure 1: Overview of the automated generation pipeline: static-analysis extraction, transformation into LLM input, prompt construction, LLM invocation, and output processing and validation.

The pipeline consists of five stages: (1) static-analysis extraction, (2) transformation of the extracted data into LLM input, (3) prompt construction, (4) LLM invocation, and (5) output processing and validation. Each stage is described below.

## 5.1 Static-Analysis Extraction

### 5.1.1 Information Requirements for C4 Diagrams

Before selecting a static-analysis tool, the structural information required to generate C2 and C3 diagrams was identified. A C2 diagram requires the identification of containers, the technologies associated with each container, and the dependencies between containers. A C3 diagram requires the identification of the components inside one container and the dependencies between them.

In C4, a container is a separately deployable or runnable unit. The source code of a single Java repository does not state explicitly which parts are deployed separately, so in this work a container is represented by a group of Java packages, and the LLM is asked to form these groups. A component is represented by a single class or interface. The first choice is discussed as a limitation in Section 9.

### 5.1.2 Tool Selection and Justification

Several static-analysis tools for Java were considered, including Spoon, SonarQube, and jQAssistant. Spoon was ultimately selected because it is designed for direct source-code analysis and transformation, building a detailed Java AST metamodel with an analysis API that supports precise program inspection at the code-element level. This made it well suited for extracting the structural information needed in this thesis, such as packages, classes, qualified names, inheritance

relations, implemented interfaces, and type dependencies. Spoon can also build its model without a fully resolved classpath (`noClasspath` mode), which is required when analysing open-source projects without first building them.

SonarQube was considered because it can analyse code structure, extract classes, modules, and inter-dependencies, and verify architecture rules during CI/CD through an architecture-as-code approach based on YAML or JSON constraints. However, its primary focus is architectural validation and issue reporting rather than offering a programmable source model for custom extraction of diagram elements, which made it less appropriate for this thesis.

jQAssistant was also a strong candidate because it scans source code, bytecode, build metadata, and related artefacts into a Neo4j graph database, where analysis is performed through concepts, constraints, and plugins. Its workflow, however, is centred on graph queries and rule evaluation, which introduces an additional modelling layer, whereas Spoon gives direct access to the Java source structure.

*Tool used externally:* Spoon (version 10.2.0) is an existing open-source static-analysis framework and was not developed as part of this thesis. *Implemented as part of this work:* the extraction program that invokes Spoon's analysis API, collects the relevant fields, computes the package summary, and writes the result to a JSON file.

### 5.1.3 Extracted Information

For every top-level class and interface, the extraction program records the fields listed in Table 1. Enums and records are treated as classes; annotation types (`@interface`) are not included. The internal dependencies of a class are collected from the types of its fields, the return and parameter types of its methods and constructors, local variable types, constructor calls and method-call targets inside method and constructor bodies, caught exception types, its superclass, and its implemented interfaces. Generic type arguments and array element types are included. A dependency is kept only if it refers to another class of the analysed project.

Table 1: Information extracted by Spoon and the diagram level it is used for.

| Information needed | Relevant for | Spoon output field |
|---|---|---|
| Package membership | C2 and C3 | `package` |
| Class names and qualified names | C3 | `name`, `qualified_name` |
| Kind of type | C3 | `is_interface`, `is_abstract` |
| Architectural annotations | C2 and C3 | `annotations` |
| Role derived from annotations | C3 | `role` |
| Inheritance relationships | C3 | `extends` |
| Implemented interfaces | C3 | `implements` |
| Dependencies on other project classes | C2 and C3 | `depends_on_internal` |

The `role` field is derived from the annotations by a fixed rule (for example, `@RestController` gives "REST Controller" and `@Entity` gives "Entity (Domain Model)"); a class without any of the listed annotations is labelled "Interface", "Abstract Class", or "Class". In addition to the per-class entries, the extraction program writes a per-package summary, described in Section 5.2. An example per-class entry is shown in Listing 1. For an interface, the interfaces it extends are recorded under `implements`, and `extends` is used only for the superclass of a class. This is why the interface `UserRepository`, which extends `JpaRepository` in the source code, has `JpaRepository` under `implements` in Listing 1.

```
{
  "name": "UserRepository",
  "qualified_name": "io.github.jhipster.sample.repository.UserRepository",
  "package": "io.github.jhipster.sample.repository",
  "is_interface": true,
  "is_abstract": true,
  "annotations": ["Repository"],
  "role": "Repository (Data Access)",
  "extends": null,
```

```
  "implements": ["JpaRepository"],
  "depends_on_internal": ["User"]
}
```

Listing 1: Spoon JSON entry for one class of the jhipster project.

#### 5.1.4 Output Filtering

Not every type reference is architecturally relevant. Dependencies on primitive types, common JDK types (such as `String`, `List`, or `Optional`), logging types, and single-letter generic type variables are excluded through a fixed list (`IGNORED_TYPES`) in the extraction program. Test code, build output, and generated sources are excluded by path (`/test/`, `/tests/`, `/target/`, `/build/`, `/generated/`), and classes whose qualified name contains "test" are also removed. Only annotations from a fixed list of architecturally meaningful annotations (Spring stereotypes, JPA, AOP, configuration and security annotations) are kept. The filtering is deterministic and was implemented as part of this work.

### 5.2 Transformation into LLM Input

The JSON file produced by Spoon is not given to the LLM as JSON. It is converted into a plain-text summary, which is the only form of static-analysis data that the LLM receives. This step, implemented as part of this work, is described below.

#### 5.2.1 Choice of Representation

The static-analysis output could also have been stored as a knowledge graph, for example in a graph database as done by jQAssistant. A graph representation is well suited for querying the structure of a system and would be a natural basis for a later comparison between a generated diagram and the real structure of the code. In this thesis, however, the static-analysis data has one purpose: to be inserted into a prompt. Any representation, graph or JSON, has to be turned into text before an LLM can read it. JSON was therefore chosen as the simplest format that stores the extracted information without loss and can be turned into prompt text with a short, deterministic script. The JSON contains exactly the nodes (classes and packages) and edges (dependencies) that a graph would contain, so it can be loaded into a graph database without loss if a later study needs it. The choice of JSON is therefore a choice of storage format, not a claim that JSON is a better representation than a knowledge graph.

#### 5.2.2 Package and Class Summaries

For C2, the summary lists every package with its number of classes, its centrality score (defined below), the technologies detected for it, and up to eight packages it depends on, followed by a complete list of all package-to-package dependencies with their reference counts. The packages are listed in order of decreasing centrality score.

The technologies of a package are found from the annotations on its classes. For example, if a class in the package has the annotation `@Entity`, the package is labelled "JPA/Hibernate"; if it has `@Service`, it is labelled "Spring Framework". A package whose name contains the word `security` is also labelled "Spring Security". If no technology is found for a package, it is labelled "Standard Java".

For C3, the summary lists every class of the selected target package with its kind (class, interface, or abstract class), role, annotations, superclass, implemented interfaces, and internal dependencies. Listing 4 in Section 5.5 shows a complete C3 summary.

The centrality score is computed by the extraction program for each package as the number of class-level dependencies that leave the package plus the number that enter it from other packages. Each dependency is counted once per pair of classes, and dependencies between classes of the same package are not counted.

### 5.2.3 Source-Code Input

In the source-code-only and combined conditions, the source code is included in the prompt as plain text. All `.java` files of the project are collected, except those in directories named `test`, `tests`, `target`, `build`, `generated`, or `resources`. The files are sorted alphabetically by path and joined into one long text, one file after another. Before each file, a comment line with the file name (for example `// FILE: src/main/java/.../UserService.java`) is added, so that the model can see where each file begins. The same loading procedure is used for both conditions and both diagram levels, so the source-code-only and combined conditions receive exactly the same source code; the only difference between them is the additional static-analysis summary in the combined condition.

The loader has an upper limit of 4,000,000 characters, which corresponds roughly to the one-million-token context window of the models used. This limit was not reached for any of the three projects, so every project was included in full.

### 5.2.4 Differences Between C2 and C3

For C2 diagrams, the static-analysis summary covers the whole project at the package level, and one diagram is generated per project. For C3 diagrams, the static-analysis summary is restricted to the classes of one selected target package. The source code, where it is part of the input, is not restricted: the whole project is provided, and the prompt instructs the model to describe only the classes of the target package. Restricting the source files by directory was avoided because directory paths do not match package names reliably across the build layouts of the three projects.

### 5.2.5 Target-Container Selection for C3

For each project, two packages are selected as C3 targets. The selection is made once, before the experiment, and the same two targets are used for all models and all three input conditions. The target is therefore a fixed experimental setting, chosen by the researchers, and not part of the input that is being compared: in the source-code-only condition the model receives only the name of the target package, and no static-analysis data.

The rule used to choose the targets is the following: the two packages with the highest centrality score are selected. The centrality score is taken from the Spoon output for convenience, because it is already available there; the same packages can be identified without static analysis by counting, for each package, the `import` statements in other files that refer to it. The selected targets are listed in Table 2, together with the number of classes and of dependencies between classes of the same package.

Table 2: Selected C3 target packages. “Internal deps.” is the number of dependencies between classes of the same package, as reported by Spoon.

| Project | Target package (index) | Classes | Internal deps. |
|---|---|---|---|
| jhipster | `...sample.web.rest` (0) | 8 | 0 |
| | `...sample.domain` (1) | 6 | 6 |
| spring-boot-admin | `...server.domain.entities` (0) | 5 | 5 |
| | `...server.domain.values` (1) | 8 | 1 |
| apache-maven | `...model.building` (0) | 32 | 78 |
| | `...maven.project` (1) | 40 | 76 |

The targets differ strongly in size. Four of the six targets have at most eight classes and almost no internal dependencies, so a C3 diagram of these packages contains few elements and few or no relationships. This affects the C3 results and is discussed in Section 8.

## 5.3 Prompt Construction

### 5.3.1 C2/C3 Instructions and Output Format

For every combination of diagram level and input condition there is one fixed prompt template. The prompt asks the model to act as a software architect and to produce only a PlantUML diagram; no textual explanation is requested. The diagrams use plain PlantUML elements: containers and components are drawn as `rectangle` elements with a short alias, C3 components are placed inside a `package` block that represents the target container, and relationships are drawn as `-->` arrows with a quoted label. The C4-PlantUML macro library is not used, and the prompt forbids `!include` and `!define` directives, so that every diagram can be rendered without external files. Each template contains a small valid example diagram and a list of syntax rules (for example, no colour names after `#` and no nested rectangles at C2). The C2 templates additionally ask the model to group related packages into at most 20 containers. All templates are listed in Appendix A.

### 5.3.2 Differences Between the Templates

The templates for the three input conditions share the same structure, output format, and syntax rules. They differ only where the instructions refer to the input itself, which is different in each condition:

- **C3 components.** In the source-code-only condition the model must find the classes of the target package in the source code. In the static-analysis-only condition every class in the summary becomes a component. In the combined condition only classes listed in the summary may be used.
- **C3 relationships.** In the source-code-only condition a relationship is drawn where the source code shows a usage. In the static-analysis-only condition a relationship is drawn only where `depends_on_internal` lists one. In the combined condition a relationship is drawn where `depends_on_internal` lists one *or* where the source code shows a usage.
- **C2 relationships and technologies.** In the conditions with static analysis, container relationships and technology labels are to be taken from the static-analysis data; in the source-code-only condition they are to be derived from the code.

These differences follow directly from the different inputs and are taken into account when the C3 results are interpreted (Section 8.2).

### 5.3.3 Combining Source Code and Static Analysis

In the combined condition, the prompt contains the static-analysis summary ("Source 1") followed by the source code ("Source 2"), separated by fixed headings. The model is told to treat Source 1 as the structural ground truth and to use Source 2 to understand responsibilities and technologies. All templates were fixed before the experiment and were not changed during it.

## 5.4 LLM Invocation and Output Processing

Each run is executed as follows:

1. A new API request is sent, without any previous conversation history, containing the fixed system instruction and the prompt for the condition. The parameters are described in Section 6.2.
2. The PlantUML block is extracted from the response as the text between the first `@startuml` and the following `@enduml`. If the response contains no such complete block, the run is counted as invalid and the raw response is stored for inspection; no attempt is made to repair the output.
3. Two cleaning steps are applied to the extracted block: lines containing `!include` or `!define` are removed, and a line that is identical (after removing surrounding whitespace) to an earlier line of the same diagram is removed.

4. The cleaned diagram is stored as a `.puml` file, named by project, model, diagram level, condition, C3 target, and run number.

The runs of a condition are sent one after another with a two-second pause. The removal of identical lines in step 3 affects the diagrams that contain many repeated declarations: exact repetitions are removed before the analysis, so the repetition ratio defined in Section 6 captures only repeated elements that differ in their alias or text.

The stored PlantUML files are then rendered with the PlantUML command-line tool to check that they are syntactically valid. This check establishes that the pipeline produces well-formed diagrams; it does *not* establish that the content of a diagram is architecturally correct.

*Implemented as part of this work:* the scripts that build the prompts, call the APIs, and extract and store the diagrams. *Used externally:* the LLM APIs (Azure OpenAI and Google AI) and the PlantUML command-line tool.

## 5.5 Worked Example

This section follows one input through the pipeline: the C3 diagram of the `io.github.jhipster.sample.domain` package of the jhipster project in the combined condition.

**Source code.** Listing 2 shows an excerpt of one class of the target package.

```
@Entity
@Table(name = "bank_account")
public class BankAccount implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE)
    private Long id;

    @NotNull
    private String name;

    @NotNull
    private BigDecimal balance;

    @ManyToOne(fetch = FetchType.LAZY)
    private User user;

    @OneToMany(fetch = FetchType.LAZY, mappedBy = "bankAccount")
    private Set<Operation> operations = new HashSet<>();
    ...
}
```

Listing 2: Excerpt of `BankAccount.java` (jhipster); getters, setters and other methods omitted.

**Static-analysis output.** From this class, Spoon records the annotations `@Entity` and `@Table`, the implemented interface `Serializable`, and the internal dependencies `User` and `Operation`, taken from the types of the fields `user` and `operations`. The fields `id`, `name`, and `balance` have JDK types (`Long`, `String`, `BigDecimal`) and are therefore not recorded as dependencies. The extraction program stores this as the JSON entry shown in Listing 3.

```
{
  "name": "BankAccount",
  "qualified_name": "io.github.jhipster.sample.domain.BankAccount",
  "package": "io.github.jhipster.sample.domain",
  "is_interface": false,
  "is_abstract": false,
  "annotations": ["Entity", "Table"],
  "role": "Entity (Domain Model)",
  "extends": null,
  "implements": ["Serializable"],
  "depends_on_internal": ["Operation", "User"]
}
```

Listing 3: Spoon JSON entry for `BankAccount` (jhipster).

**Text summary.** For the C3 prompt, the entries of all six classes of the package are converted into the text summary shown in Listing 4. This text is inserted into the prompt as Source 1.

```
CONTAINER TO DECOMPOSE: io.github.jhipster.sample.domain
Total classes: 6
============================================================
  [Abstract] AbstractAuditingEntity
    Role             : Entity (Domain Model)
    Annotations      : JsonIgnoreProperties, MappedSuperclass
    Extends          : -
    Implements       : Serializable
    Internal deps    : -
  [Class] Authority
    Role             : Entity (Domain Model)
    Annotations      : Entity, JsonIgnoreProperties, Table
    Extends          : -
    Implements       : Persistable, Serializable
    Internal deps    : -
  [Class] BankAccount
    Role             : Entity (Domain Model)
    Annotations      : Entity, Table
    Extends          : -
    Implements       : Serializable
    Internal deps    : Operation, User
  [Class] Label
    Role             : Entity (Domain Model)
    Annotations      : Entity, Table
    Extends          : -
    Implements       : Serializable
    Internal deps    : Operation
  [Class] Operation
    Role             : Entity (Domain Model)
    Annotations      : Entity, Table
    Extends          : -
    Implements       : Serializable
    Internal deps    : BankAccount, Label
  [Class] User
    Role             : Entity (Domain Model)
    Annotations      : Entity, Table
    Extends          : AbstractAuditingEntity
    Implements       : Serializable
    Internal deps    : AbstractAuditingEntity
```

Listing 4: C3 static-analysis summary for `io.github.jhipster.sample.domain`, as inserted into the prompt.

**Prompt.** The prompt consists of the fixed combined C3 template (Appendix A) with four parts filled in: the project name, the target package, Source 1 (Listing 4), and Source 2 (the source code of the whole project, loaded as described in Section 5.2.3).

**Generated diagram.** Listing 5 shows the PlantUML returned in one run, and Figure 2 shows the rendered diagram. Every class of the summary appears as a component inside the package block. The five dependencies listed in the summary for `BankAccount`, `Label`, and `Operation` appear as `uses` arrows, and the superclass of `User` appears as an `extends` arrow.

```
@startuml
title io.github.jhipster.sample.domain - Component Diagram

package "io.github.jhipster.sample.domain" {
    rectangle "AbstractAuditingEntity\n<<Entity (Domain Model)>>" as abstractAuditingEntity
    rectangle "Authority\n<<Entity (Domain Model)>>" as authority
    rectangle "BankAccount\n<<Entity (Domain Model)>>" as bankAccount
    rectangle "Label\n<<Entity (Domain Model)>>" as label
    rectangle "Operation\n<<Entity (Domain Model)>>" as operation
    rectangle "User\n<<Entity (Domain Model)>>" as user
    bankAccount --> operation : "uses"
    bankAccount --> user : "uses"
    label --> operation : "uses"
```

```
    operation --> bankAccount : "uses"
    operation --> label : "uses"
    user --|> abstractAuditingEntity : "extends"
}
@enduml
```

Listing 5: PlantUML output of one run (combined condition, C3, jhipster domain package, Gemini 2.5 Flash, run 1).

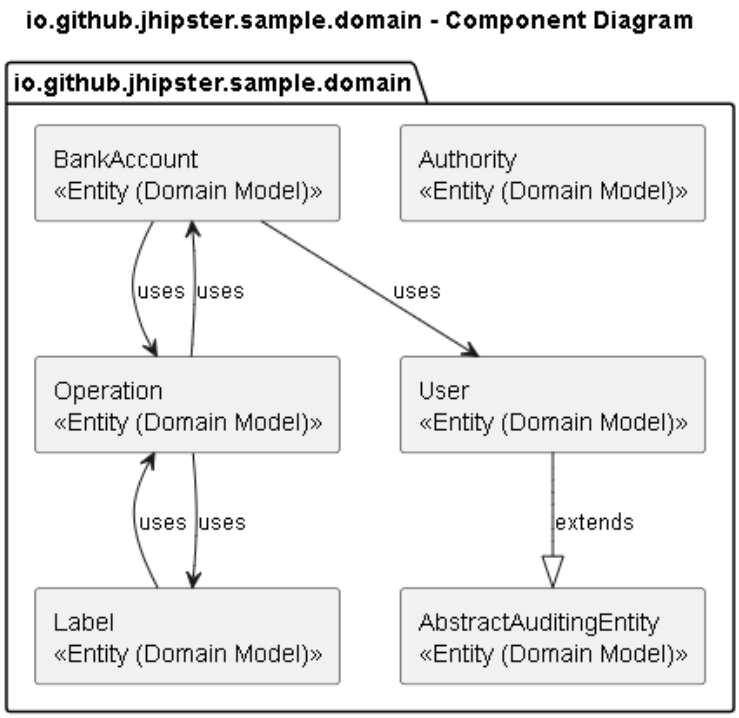


Figure 2: Rendered C3 diagram of Listing 5.

For comparison, Figure 3 shows a C2 diagram generated for the same project in the combined condition.

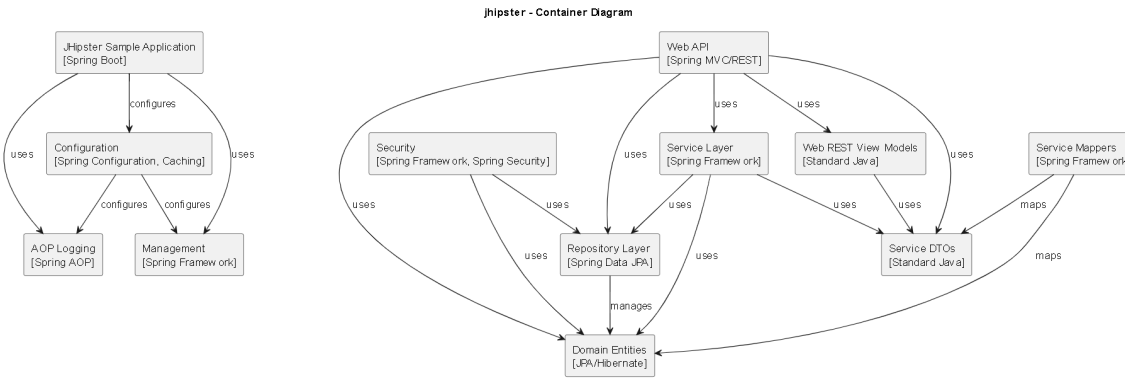


Figure 3: Example C2 (Container) diagram for the jhipster project under the combined condition (Gemini 2.5 Flash, run 1).

# 6. Evaluation Protocol

This section describes how the empirical evaluation of the pipeline (Section 5) was designed and executed, directly addressing RQ2. Results are reported separately in Section 7.

## 6.1 Study Factors and Variables

Following established guidelines for experimentation in software engineering [19], this section identifies the factors deliberately varied in the study, the variables held constant, and the dependent variables that constitute the measured outcomes.

### 6.1.1 Factors

A factor is any variable that is deliberately varied in order to observe its effect on the outcomes [19]. The study involves the following factors:

- **Input type.** The type of information provided to the LLM. Three levels are defined: (1) source code only, (2) static analysis output only, and (3) static analysis output combined with source code. *The primary comparison for RQ2 is source-code-only versus combined; the static-analysis-only condition is treated as an additional condition that shows what static analysis alone can support without source code*, and is reported separately.
- **LLM model.** The large language model used for diagram generation. Three models are included so that findings are not specific to a single model, as described in Section 6.2.
- **Target project.** The open-source software project used as input. Projects of different size are selected to see whether the effect of static analysis changes with the size of the codebase, as described in Section 6.3.
- **Diagram level.** The C4 level targeted: Container (C2) or Component (C3). The two levels are analysed separately because C2 produces one diagram per project, while C3 describes one selected container and is applied to two containers per project.

### 6.1.2 Controlled Variables

The following variables are held constant so that observed differences can be attributed to the factors above:

- **Prompt template.** The same prompt template is used for all runs of the same diagram level and input type (see Section 5.3).
- **API parameters.** Temperature, top-$p$, and maximum output tokens are the same for all models and all runs (see Section 6.2).
- **Output format.** All runs produce a plain PlantUML diagram in the format described in Section 5.3.
- **Static-analysis tool version and configuration.** Spoon is used in one fixed version with identical configuration for all projects (see Section 5.1).
- **Source-code version.** Each target project is fixed to one version (see Table 4). The same version is used for all runs involving that project.
- **Session isolation.** Each run is a new API request without any conversation history.

### 6.1.3 Dependent Variables

The following dependent variables are measured. Their exact definitions are given in Section 6.5.

- **Syntactic validity.** Whether the PlantUML output can be rendered without errors.
- **Output consistency.** How similar the diagrams of repeated runs of the same condition are, measured separately for nodes and for directed edges.
- **Number of distinct diagrams.** How many different diagrams appear among the ten runs of a condition.
- **Within-diagram repetition ratio.** How often the same element is declared again inside one diagram.

The correctness of the diagrams with respect to the source code is not a dependent variable. It is discussed as future work in Section 10.

## 6.2 LLM Selection and Configuration

### 6.2.1 Model Selection and Justification

Locally hosted open-weight models, accessed via Ollama, were evaluated first. Two models were tested, `qwen2.5-coder:7b` and `qwen3:4b`, chosen for their small size (4.7 GB and 2.5 GB) and, in the case of the former, its specialisation in code. This approach was abandoned for two reasons. First, the context window of these local configurations was too small for the combined condition, which requires the complete source code of the project together with the static-analysis summary in one prompt. Second, running the full study, 810 generations, was not feasible on the available student hardware in reasonable time. These limitations led to the use of API-based models.

Model selection was guided by three criteria: API access with control over the generation parameters, a context window large enough for the source code of the largest project together with the static-analysis summary, and cost within the available student budget. Web interfaces were excluded because they do not allow the generation parameters to be fixed.

Two models were accessed through Azure for Students, which provided $100 in credits: GPT-4.1 mini and GPT-4.1 nano. Both belong to the GPT-4.1 family, have a context window of about one million tokens, and offer a good ratio of cost to capability. Gemini 2.5 Flash was accessed through the Google AI API and was included so that the findings do not depend on a single vendor.

Table 3: Selected LLMs.

| Model name | Vendor | API access | API model string | Version reported by API |
|---|---|---|---|---|
| GPT-4.1 mini | OpenAI | Azure for Students | `gpt-4.1-mini` | `gpt-4.1-mini-2025-04-14` |
| GPT-4.1 nano | OpenAI | Azure for Students | `gpt-4.1-nano` | `gpt-4.1-nano-2025-04-14` |
| Gemini 2.5 Flash | Google | Google AI API | `gemini-2.5-flash` | `gemini-2.5-flash` |

The last column of Table 3 gives the model version reported by the API during the experiment. For the GPT models, Azure reports a dated model version. For Gemini 2.5 Flash, the API reports only the stable model name, and no more detailed version identifier is available.

### 6.2.2 API Parameter Configuration

The following parameters are set identically for all runs and all models:

- **Temperature = 0.0.** A temperature of zero makes the model choose the most likely next token at every step, which removes most of the randomness of sampling. This is the usual setting when consistency is measured. It does not guarantee identical outputs, because the provider's infrastructure can still produce small numerical differences between requests; measuring how much variation remains is exactly the purpose of this study.

- **Top-$p$ = 1.0.** With temperature fixed at zero, top-$p$ has no practical effect. It is set to its neutral value and reported for completeness.
- **Maximum output tokens = 32,768.** This is the largest output length supported by the GPT-4.1 models and is far above the length of a normal diagram, so that valid outputs are not cut off. The same value is used for Gemini.
- **Thinking disabled (Gemini only).** Gemini 2.5 Flash can perform an internal reasoning step before answering. This step was disabled (`thinking_budget = 0`) so that all three models produce their answer directly and the reasoning step does not add another source of variation.

## 6.3 Target Projects

### 6.3.1 Rationale for Restricting to the Java Ecosystem

The study is restricted to Java for three reasons. First, Java has mature static-analysis tooling, including Spoon, which provides the AST-level access required for this thesis. Second, Java is widely used in open-source enterprise software, so suitable projects of different sizes are available. Third, using a single language removes language-specific differences as a confounding factor.

### 6.3.2 Selection Criteria

Projects are selected according to the following criteria:

1. Open-source with a permissive licence (MIT or Apache 2.0) permitting publication of derived artefacts.
2. The backend is written in Java. Modules in other languages, such as web frontends, may be present, but only the `.java` files are analysed and provided to the model.
3. Compatible with Spoon, i.e., parseable without a fully resolved classpath.
4. Architecturally non-trivial: at least three packages, so that a container diagram with several elements can be drawn.

### 6.3.3 Selected Projects

Three projects are selected, one from each size category.

Table 4: Selected target projects.

| Project | Approx. LOC | Size category | Rationale |
|---|---|---|---|
| jhipster | 4,500 | Small (<10k) | Open-source, Apache 2.0, Java Spring Boot application, clear package hierarchy |
| spring-boot-admin | 15,300 | Medium (10k–50k) | Open-source, Apache 2.0, Java, separate client and server modules |
| apache-maven-3.9.15 | 84,000 | Large (>50k) | Open-source, Apache 2.0, large multi-module Java build tool |

### 6.3.4 Container Selection for C3 Analysis

For C2, one diagram is generated per project. For C3, two target packages are selected per project; the procedure and the selected packages are described in Section 5.2.5.

## 6.4 Study Execution

### 6.4.1 Factorial Structure and Total Conditions

The study crosses input type (3 levels), LLM model (3), and target project (3). At C2 there is one diagram per project, and at C3 there are two target packages per project:

- C2 conditions: $3 \times 3 \times 3 \times 1 = 27$
- C3 conditions: $3 \times 3 \times 3 \times 2 = 54$
- Total conditions: $27 + 54 = 81$

### 6.4.2 Number of Repetitions

Each condition is executed $r = 10$ times. Ten repetitions give $\binom{10}{2} = 45$ pairs of runs per condition. The total number of API calls is $81 \times 10 = 810$.

### 6.4.3 Execution Procedure

The execution of a single run is described in Section 5.4. Prompts and outputs were not changed manually at any point.

## 6.5 Metrics and Measurement

All metrics are computed automatically by scripts that are published together with the generated diagrams (Section 7.9). No step of the analysis involves manual judgement.

### 6.5.1 Syntactic Validity

Each stored PlantUML file is rendered with the PlantUML command-line tool. The outcome is binary (valid or invalid). The validity rate of a condition is the share of valid diagrams among its ten runs.

### 6.5.2 From a Diagram to Nodes and Edges

To compare diagrams, every PlantUML file is turned into a set of nodes and a set of directed edges.

**Nodes.** A node is every element declared with a PlantUML element keyword (such as `rectangle`, `component`, or `interface`). The node label is the text of the element without its second line: for a declaration such as `rectangle "Service Layer\n[Spring]" as service` the label is `Service Layer`. Stereotypes (`<<...>>`) and a trailing technology in brackets are removed as well, so that only the name of the element is compared.

**Edges.** An edge is a pair (source, target) taken from an arrow such as `service --> db`. The arrow type and the text label of the relationship are not part of the edge. An arrow pointing to the left is stored in the reverse direction, and an arrow with heads on both sides is stored as two edges.

**Explicit and implicit nodes.** An arrow can refer to an element that was never declared, for example `user --|> Serializable`. PlantUML still draws such an element. These *implicit* nodes are kept separately. The main analysis uses only explicitly declared nodes and only edges whose two ends are both declared (*explicit* variant). A second analysis that includes implicit nodes and all edges (*all* variant) is used to check that the conclusions do not depend on this choice.

**Unrecognised lines.** Every line that is neither a node, an edge, nor a structural line (such as `@startuml` or a closing brace) is written to an audit list instead of being ignored silently. Package declarations of C3 diagrams appear in this list, because the package is the container boundary and not a component. In the whole dataset, only five lines with an arrow were not recognised; in all of them the model wrote a placeholder such as `--> (none)` instead of a target element, and they were not counted as edges.

### 6.5.3 Label Alignment

Two runs can name the same element slightly differently, for example `UserService` and `User Service`. If labels are compared as raw text, such differences count as disagreement. Node labels are therefore compared in three forms, from strict to tolerant:

1. **Raw:** the label exactly as generated.
2. **Format-normalised:** camelCase and acronyms are split into words, hyphens, underscores, and dots become spaces, other punctuation is removed, and the text is lower-cased. `UserService`, `user_service`, and `User Service` all become `user service`.
3. **Similarity clusters:** format-normalised labels that are similar in both wording and meaning are grouped automatically, as described below. In the analysis scripts and output files this representation is called `semantic`.

**Automatic clustering.** For every pair of format-normalised labels a similarity between 0 and 1 is computed as the average of two scores:

- a *lexical* score: the RapidFuzz `token_sort_ratio`, which compares the words of the two labels independently of their order;
- a *semantic* score: the cosine similarity of the two labels encoded with the sentence-embedding model `all-MiniLM-L6-v2` (version fixed by its commit hash).

Labels are then grouped by agglomerative clustering with complete linkage: two labels can only be in the same group if *every* pair of labels in that group has a similarity of at least a chosen threshold. Three thresholds are used: 0.85, 0.90, and 0.95. The threshold 0.90 is the primary setting; the other two show how sensitive the results are to this choice. Giving the lexical score half of the weight makes the clustering conservative: two labels are grouped only if they are similar both in meaning and in wording. This avoids merging labels of different elements that share a topic, at the cost of not merging some synonyms with different words (for example `database` and `data store`).

Clustering is done separately for every project and diagram level, over the labels of all models and all input conditions together. All conditions are therefore compared with the same set of groups. The procedure is deterministic: running it again on the same diagrams gives the same groups. The static-analysis output is not used to form the groups, and there is no manual step.

**Scope of the groups.** The groups are computed for one given set of generated diagrams. If new diagrams are added, for example for a new version of a project, the clustering is simply run again on the new set; the groups are not carried over by hand. A group only states that two labels are very similar in wording and meaning. It is based on the labels alone and is not a claim that they denote the same element in the code.

**Check of the clusters.** At C3, many labels are class names, so the clusters can be checked automatically against the class names extracted by Spoon. A cluster that contains two labels which are *different* Spoon class names is a demonstrable wrong merge. The share of such clusters is reported in Section 7. Spoon is used here only to check the groups after they have been formed; the groups are not changed based on this check. At C2, labels are free text and this check is not possible.

### 6.5.4 Output Consistency

For each pair of runs within a condition, the Jaccard similarity is computed separately for the node sets $N$ and the edge sets $E$:

$$J_{\text{nodes}}(A, B) = \frac{|N_A \cap N_B|}{|N_A \cup N_B|}, \qquad J_{\text{edges}}(A, B) = \frac{|E_A \cap E_B|}{|E_A \cup E_B|} \tag{1}$$

A value of 1 means that the two runs contain exactly the same nodes (or edges); a value of 0 means they have nothing in common. The consistency of a condition is the mean over its 45 pairs.

Two special cases are handled explicitly:

- **Two empty edge sets.** If neither run of a pair contains an edge, $J_{\text{edges}}$ is set to 1, because the two runs agree. This happens mainly at C3 for packages with few internal dependencies, and it can raise the edge score without any edge being drawn. The share of such pairs is therefore reported, and the edge score is also computed over the remaining pairs only ("non-empty edges"). Groups in which no run contains any edge have no such pairs; for them, edge stability cannot be measured over non-empty pairs, and they are left out of this comparison.
- **Merged endpoints.** If the two ends of an edge fall into the same similarity cluster, the edge becomes a self-loop and is kept as such.

### 6.5.5 Number of Distinct Diagrams

For each condition, the number of different diagrams among the ten runs is counted. Two runs give the same diagram if their node sets and edge sets are identical. The value ranges from 1 (all ten runs identical) to 10 (all runs different). The size of the largest group of identical runs is also reported.

### 6.5.6 Within-Diagram Repetition Ratio

Some outputs repeat the same element many times, for example by declaring `Service Layer` as `svc1`, `svc2`, `svc3`, and so on until the output limit is reached. Such degenerate outputs are not invalid PlantUML, but they distort the node and edge sets. To detect them, the repetition ratio of a diagram is defined as

$$\text{repetition ratio} = \frac{\text{number of explicit node declarations}}{\text{number of different format-normalised labels}} \tag{2}$$

A normal diagram declares every element once and has a ratio of 1.0. A ratio of 3.0 means that, on average, every element is declared three times. A ratio above 2.0 is used as a warning flag. Flagged diagrams are *not* removed from the main analysis; instead, consistency is computed a second time without them to show how much they affect the results. Because identical lines are removed during output processing (Section 5.4), the ratio counts only repetitions that differ in alias or text, and therefore underestimates the true amount of repetition.

### 6.5.7 Aggregation and Statistical Analysis

Pairs of runs share runs with each other and are therefore not independent observations. The unit of statistical analysis is therefore a *cell*: one combination of model, project, and (for C3) target package. The value of a cell is its mean consistency over the 45 pairs. This gives 9 cells at C2 ($3 \times 3$) and 18 cells at C3 ($3 \times 3 \times 2$).

Input conditions are compared within each cell with the two-sided Wilcoxon signed-rank test, which compares paired values without assuming a normal distribution. Cells with a difference of zero are left out of the test. When zero differences or tied values occur, the p-value is computed with the normal approximation instead of the exact distribution. The size of the effect is reported as the matched-pairs rank-biserial correlation, which ranges from $-1$ (the second condition is higher in every cell) to $+1$ (the first condition is higher in every cell).

Because several tests are run, the $p$-values are adjusted with the Holm method within each group of related tests. The groups were fixed before the tests were run:

- **Primary group (4 tests):** combined versus source code only, on the primary label representation (similarity clusters, threshold 0.90), for C2 and C3, for nodes and edges.
- **Non-empty edges (2 tests):** the same comparison for edges, using only pairs in which at least one edge set is non-empty.
- **Secondary group:** all other label representations and the comparison of combined with static analysis only. These results are reported as sensitivity checks.

With 9 cells, the smallest possible two-sided $p$-value is $2/2^9 \approx 0.004$, so the tests can only detect differences that are consistent across almost all cells. Cells that share a model or a project are not fully independent, which is listed as a threat to validity. Besides the tests, results are reported per model and per project, so that differences that are hidden in the overall averages remain visible.

# 7. Results

This section reports the results of the evaluation described in Section 6. This section only presents the results; what they mean for RQ1 and RQ2 is discussed in Section 8. Unless stated otherwise, all consistency values use the explicit variant (Section 6.5.2) and similarity clusters with threshold 0.90 (Section 6.5.3). Pooled values are means and standard deviations over all pairs of runs in a group (45 pairs per condition). These pooled values summarise the data but are not used for statistical tests, because pairs share runs and are therefore not independent. Statistical tests use the cell means described in Section 6.5.

## 7.1 Syntactic Validity

All 810 generated diagrams contained a complete `@startuml` ... `@enduml` block and were rendered by the PlantUML command-line tool without errors (Table 5). No output required manual repair; only the automatic cleaning described in Section 5.4 was applied.

Table 5: Syntactic validity of the generated PlantUML diagrams.

| Level | Valid diagrams | Total diagrams | Validity rate (%) |
|---|---|---|---|
| C2 | 270 | 270 | 100.0 |
| C3 | 540 | 540 | 100.0 |
| Overall | 810 | 810 | 100.0 |

## 7.2 Main Comparison: Code-Only versus Combined Input

Table 6 shows the pooled consistency of the three input conditions. The static-analysis-only condition is shown for reference.

Table 6: Pooled consistency per input condition (mean and standard deviation over all pairs of runs).

| | | Code-only | | Combined | | | SA-only | |
|---|---|---|---|---|---|---|---|---|
| Level | Metric | Mean | SD | Mean | SD | Δ | Mean | SD |
| C2 | Nodes | 0.778 | 0.296 | 0.833 | 0.287 | +0.055 | 0.669 | 0.396 |
| C2 | Edges | 0.603 | 0.370 | 0.705 | 0.372 | +0.102 | 0.574 | 0.430 |
| C3 | Nodes | 0.923 | 0.172 | 0.969 | 0.123 | +0.046 | 0.974 | 0.096 |
| C3 | Edges | 0.828 | 0.314 | 0.888 | 0.247 | +0.060 | 0.876 | 0.239 |
| C3 | Edges (non-empty) | 0.754 | 0.351 | 0.855 | 0.272 | +0.101 | 0.859 | 0.250 |

Each C2 value pools 405 pairs (9 model–project cells × 45 pairs); each C3 value pools 810 pairs (18 model–project–target cells × 45 pairs). "Edges (non-empty)" excludes pairs in which both runs have no edges; such pairs make up 30.1% (code-only), 22.3% (combined), and 11.9% (SA-only) of all C3 pairs, and less than 1% at C2.

At both levels the combined condition has a higher mean consistency than the code-only condition. At C2 the difference is +0.055 for nodes and +0.102 for edges; at C3 it is +0.046 for nodes and +0.060 for edges. The SA-only condition has the lowest mean consistency at C2 and the highest node consistency at C3. The standard deviations are large at C2 (0.29–0.43), which means that the pooled means hide large differences between cells.

Table 7 reports the Wilcoxon signed-rank tests over cells for the primary comparison.

Table 7: Wilcoxon signed-rank tests, combined versus code-only, per cell. "Higher" counts the cells in which each condition has the higher mean; cells with equal values are not counted. The median $\Delta$ is computed over all cells, including cells with a difference of zero, whereas the test and $r$ use only the cells with a non-zero difference. $r$ is the matched-pairs rank-biserial correlation (positive values favour combined). The $p$-values at C2 are exact; at C3, where zero differences or tied values occur, they are computed with the normal approximation.

| Level | Metric | Cells | Higher | | Median $\Delta$ | $r$ | $p$ | $p_{\text{Holm}}$ |
|---|---|---|---|---|---|---|---|---|
| | | | Comb. | Code | | | | |
| *Primary group* | | | | | | | | |
| C2 | Nodes | 9 | 5 | 4 | +0.088 | 0.29 | 0.496 | 0.992 |
| C2 | Edges | 9 | 5 | 4 | +0.115 | 0.20 | 0.652 | 0.992 |
| C3 | Nodes | 18 | 6 | 0 | 0.000 | 1.00 | 0.028 | 0.111 |
| C3 | Edges | 18 | 8 | 4 | 0.000 | 0.46 | 0.158 | 0.474 |
| *Non-empty edges* | | | | | | | | |
| C2 | Edges | 9 | 5 | 4 | +0.115 | 0.20 | 0.652 | 0.652 |
| C3 | Edges | 12 | 7 | 2 | +0.097 | 0.82 | 0.027 | 0.055 |

None of the differences is statistically significant after Holm correction. At C2, the combined condition is higher in five of the nine cells and the code-only condition in four. At C3, the combined condition is never lower for nodes, but twelve of the eighteen cells have exactly the same node consistency in both conditions (all of them 1.0, for jhipster and spring-boot-admin), so the test is based on six cells only. The comparison of combined with SA-only (secondary group) is also not significant at either level (all $p_{\text{Holm}} = 1.0$). At C3, only 12 of the 18 cells could be used for the non-empty edges test. In the other six cells, the combined or the code-only condition produced no edges in any of its ten runs, so there was nothing to compare.

## 7.3 Consistency by Model

Table 8 breaks the comparison down by model, pooled over the three projects (and, for C3, both targets).

Table 8: Pooled consistency by model.

| | | Node Jaccard | | | Edge Jaccard | | | SA-only | |
|---|---|---|---|---|---|---|---|---|---|
| Level | Model | Code | Comb. | $\Delta$ | Code | Comb. | $\Delta$ | Node | Edge |
| C2 | Gemini 2.5 Flash | 0.907 | 0.924 | +0.017 | 0.861 | 0.799 | −0.062 | 0.555 | 0.519 |
| C2 | GPT-4.1 mini | 0.849 | 0.633 | −0.216 | 0.544 | 0.539 | −0.005 | 0.917 | 0.856 |
| C2 | GPT-4.1 nano | 0.576 | 0.942 | +0.366 | 0.404 | 0.777 | +0.373 | 0.535 | 0.347 |
| C3 | Gemini 2.5 Flash | 0.969 | 1.000 | +0.031 | 0.873 | 0.958 | +0.085 | 1.000 | 0.979 |
| C3 | GPT-4.1 mini | 0.866 | 0.963 | +0.097 | 0.733 | 0.856 | +0.123 | 1.000 | 0.905 |
| C3 | GPT-4.1 nano | 0.934 | 0.945 | +0.011 | 0.878 | 0.849 | −0.029 | 0.923 | 0.743 |

Each C2 value pools 135 pairs (3 projects × 45); each C3 value pools 270 pairs (3 projects × 2 targets × 45).

At C2, the effect of adding static analysis differs in direction between models. For GPT-4.1 nano, node consistency rises from 0.576 to 0.942. For GPT-4.1 mini, it falls from 0.849 to 0.633. For Gemini 2.5 Flash, it is almost unchanged (0.907 and 0.924), while its edge consistency is lower in the combined condition. The SA-only condition shows the same pattern: it is the most consistent condition for GPT-4.1 mini (0.917) and the least consistent for Gemini 2.5 Flash (0.555). At C3, the combined condition has equal or higher node consistency than code-only for all three models.

Across conditions, Gemini 2.5 Flash is the most consistent model at both levels. At C3 it reaches a node consistency of 1.000 in the combined and SA-only conditions.

## 7.4 Consistency by Project

Table 9 breaks the comparison down by project, pooled over the three models (and, for C3, both targets).

Table 9: Pooled consistency by project.

| | | Node Jaccard | | | Edge Jaccard | | | SA-only | |
|---|---|---|---|---|---|---|---|---|---|
| Level | Project | Code | Comb. | Δ | Code | Comb. | Δ | Node | Edge |
| C2 | jhipster | 0.786 | 1.000 | +0.214 | 0.526 | 0.854 | +0.328 | 0.797 | 0.715 |
| C2 | spring-boot-admin | 0.802 | 0.799 | −0.003 | 0.717 | 0.738 | +0.021 | 0.732 | 0.612 |
| C2 | apache-maven | 0.745 | 0.700 | −0.045 | 0.567 | 0.522 | −0.045 | 0.478 | 0.394 |
| C3 | jhipster | 1.000 | 1.000 | 0.000 | 0.905 | 0.962 | +0.057 | 1.000 | 0.936 |
| C3 | spring-boot-admin | 1.000 | 1.000 | 0.000 | 0.913 | 0.932 | +0.019 | 1.000 | 0.904 |
| C3 | apache-maven | 0.769 | 0.908 | +0.139 | 0.665 | 0.770 | +0.105 | 0.923 | 0.788 |

At C2, adding static analysis raises consistency clearly for jhipster, where every combined run of every model produced the same set of containers (node consistency 1.000). For spring-boot-admin and apache-maven the difference is close to zero or slightly negative. The SA-only condition is least consistent for the largest project (0.478).

At C3, jhipster and spring-boot-admin have a node consistency of 1.000 in all three conditions: for these projects, every run of every model produced the same set of components. The C3 differences in node consistency between conditions therefore come entirely from apache-maven, where the combined (0.908) and SA-only (0.923) conditions are more consistent than code-only (0.769). Edge consistency varies in all three projects.

## 7.5 Number of Distinct Diagrams

Table 10 reports how many different diagrams were produced among the ten runs of a condition.

Table 10: Distinct diagrams among the ten runs of a condition (mean over conditions), and number of conditions in which all ten runs produced the same diagram.

| Level | Condition | Conditions | Mean distinct diagrams | All ten identical |
|---|---|---|---|---|
| C2 | Code-only | 9 | 4.67 | 2 |
| C2 | Combined | 9 | 3.44 | 1 |
| C2 | SA-only | 9 | 5.00 | 1 |
| C3 | Code-only | 18 | 2.94 | 7 |
| C3 | Combined | 18 | 2.44 | 9 |
| C3 | SA-only | 18 | 3.28 | 6 |

Only 4 of the 27 C2 conditions and 22 of the 54 C3 conditions produced the same diagram in all ten runs. By model, the mean number of distinct C2 diagrams is 1.78 for Gemini 2.5 Flash, 5.44 for GPT-4.1 mini, and 5.89 for GPT-4.1 nano.

## 7.6 Repetitive Outputs

Thirty of the 810 diagrams (3.7%) have a repetition ratio above 2.0. All of them were produced by the GPT-4.1 models (8 by mini, 22 by nano); none by Gemini 2.5 Flash. Table 11 lists the groups that contain at least one such diagram.

Table 11: Groups with at least one diagram with repetition ratio $> 2.0$.

| Model | Project | Level | Condition | Flagged / total | Max ratio | Mean ratio |
|---|---|---|---|---|---|---|
| GPT-4.1 mini | apache-maven | C2 | combined | 8 / 10 | 46.4 | 16.5 |
| GPT-4.1 nano | apache-maven | C3 | code-only | 11 / 20 | 72.1 | 17.2 |
| GPT-4.1 nano | spring-boot-admin | C3 | code-only | 10 / 20 | 101.0 | 51.0 |
| GPT-4.1 nano | apache-maven | C2 | SA-only | 1 / 10 | 27.5 | 3.8 |

Table 12 shows how consistency changes in these groups when the flagged diagrams are left out.

Table 12: Pooled consistency with all diagrams and without diagrams with repetition ratio > 2.0 (pooled over projects).

| | | | | Nodes | | Edges | |
|---|---|---|---|---|---|---|---|
| Model | Level | Condition | Excluded | All | Without | All | Without |
| GPT-4.1 mini | C2 | combined | 8 / 30 | 0.633 | 0.787 | 0.539 | 0.753 |
| GPT-4.1 nano | C2 | SA-only | 1 / 30 | 0.535 | 0.564 | 0.347 | 0.364 |
| GPT-4.1 nano | C3 | code-only | 21 / 60 | 0.934 | 0.970 | 0.878 | 0.808 |

For GPT-4.1 mini at C2 in the combined condition, the low consistency is largely caused by the repetitive outputs: without them, node consistency rises from 0.633 to 0.787. For GPT-4.1 nano at C3 in the code-only condition, node consistency rises but edge consistency falls (from 0.878 to 0.808), because many of the repetitive diagrams had no edges and pairs of empty edge sets count as agreement.

## 7.7 Effect of Label Alignment

Table 13 compares consistency for raw labels, format-normalised labels, and similarity clusters.

Table 13: Pooled consistency for the three label representations.

| Level | Condition | Metric | Raw | Format | Clusters | Δ Raw→Fmt | Δ Fmt→Clust. |
|---|---|---|---|---|---|---|---|
| C2 | Code-only | Nodes | 0.775 | 0.775 | 0.778 | 0.000 | +0.003 |
| C2 | Code-only | Edges | 0.601 | 0.601 | 0.603 | 0.000 | +0.002 |
| C2 | Combined | Nodes | 0.819 | 0.822 | 0.833 | +0.003 | +0.011 |
| C2 | Combined | Edges | 0.700 | 0.701 | 0.705 | +0.001 | +0.004 |
| C2 | SA-only | Nodes | 0.643 | 0.645 | 0.669 | +0.002 | +0.024 |
| C2 | SA-only | Edges | 0.542 | 0.542 | 0.574 | 0.000 | +0.032 |
| C3 | Code-only | Nodes | 0.921 | 0.921 | 0.923 | 0.000 | +0.002 |
| C3 | Code-only | Edges | 0.825 | 0.825 | 0.828 | 0.000 | +0.003 |
| C3 | Combined | Nodes | 0.969 | 0.969 | 0.969 | 0.000 | 0.000 |
| C3 | Combined | Edges | 0.888 | 0.888 | 0.888 | 0.000 | 0.000 |
| C3 | SA-only | Nodes | 0.974 | 0.974 | 0.974 | 0.000 | +0.001 |
| C3 | SA-only | Edges | 0.875 | 0.875 | 0.876 | 0.000 | +0.001 |

Label alignment changes the measured consistency by at most 0.032. The largest changes occur in the SA-only condition at C2. The results also change little with the clustering threshold: between thresholds 0.85, 0.90, and 0.95, the pooled values differ by at most 0.029 for any condition and metric.

**Check of the clusters.** At C3, the clusters were checked against the class names extracted by Spoon (Section 6.5.3). At the primary threshold of 0.90, six clusters in total merged two different class names: five in apache-maven and one in spring-boot-admin. All six merge an interface or class with a closely named one, for example `Endpoint` and `Endpoints`, or an interface such as `PluginConfigurationExpander` and its implementation `DefaultPluginConfiguration-Expander`. In apache-maven these clusters cover 2.5% of all C3 node occurrences. At threshold 0.85 the number of wrong merges in apache-maven rises to 45 (30.1% of node occurrences); at 0.95 there are none. No wrong merges were found for jhipster. These wrong merges change the consistency values only slightly, because the merged labels, such as an interface and its default implementation, usually appear together in the same runs. Merging them affects all runs of a condition in the same way, so the overlap between runs, and therefore the Jaccard similarity, changes little.

## 7.8 Robustness to the Treatment of Undeclared Elements

When implicit nodes and all edges are included ("all" variant, Section 6.5.2), the pooled values change but the pattern stays the same: at C2, node consistency is 0.766 (code-only), 0.825 (combined), and 0.653 (SA-only); at C3 it is 0.860, 0.892, and 0.937. None of the primary tests is significant in this variant either (all $p_{\text{Holm}} \geq 0.12$).

## 7.9 Data and Code Availability

The scripts, static-analysis outputs, generated diagrams, and result data supporting the analyses in this section are publicly available in our Zenodo repository[1].

[1] https://doi.org/10.5281/zenodo.23298354

# 8. Discussion

This section interprets the results reported in Section 7 in relation to the two research questions, discusses possible explanations for the observed patterns, and clarifies what the results do and do not establish.

## 8.1 Addressing RQ1: Pipeline Design and Operation

RQ1 asked how static-analysis information can be structured and combined with a large language model to automatically generate Container (C2) and Component (C3) diagrams from source code. Answering this question required several concrete design decisions.

The first decision was to turn the Spoon output into a short plain-text summary instead of passing the JSON file to the model (Section 5.2). A flat list of every class and dependency is long and gives no indication of which parts of the system matter most. The summary therefore adds two package-level values, the number of classes and a centrality score, and lists the packages in order of centrality, so that the most connected parts of the system appear first.

The second decision concerned the source code. In the source-code-only and combined conditions the complete source code of the project is included, loaded in the same way for both conditions (Section 5.2.3). This keeps the comparison between the two conditions fair: the only difference between them is the static-analysis summary. The context windows of the selected models were large enough to hold even the largest project.

The third decision was how to restrict a C3 diagram to one container. The source files were not filtered by directory, because the three projects organise their folders differently and a file's location does not always match its package. Instead, the model receives the whole project and is told in the prompt which package to describe. The static-analysis summary, in contrast, is filtered directly, because every class record already states its package.

The fourth decision concerned the reliability of the output format. Early generations often produced PlantUML that could not be rendered, mostly because of colour names after `#` or nested elements. These problems were addressed by adding explicit syntax rules and a valid example to the prompt templates (Section 5.3), rather than by repairing the output afterwards.

With these decisions, all 810 generated diagrams were syntactically valid without manual repair (Table 5); only the automatic cleaning described in Section 5.4 was applied. This shows that the pipeline reliably produces well-formed PlantUML.

## 8.2 Addressing RQ2: Effect of Static Analysis on Consistency

RQ2 asked to what extent providing static-analysis output to the LLM affects the consistency of the generated diagrams across repeated runs, compared to source code alone. The short answer is that **no reliable effect was found**. The combined condition had a higher mean consistency than the code-only condition at both levels (Table 6), but none of the differences was statistically significant (Table 7), and the direction of the effect differed between models and projects.

**Repeated generation is not deterministic.** Before comparing the conditions, the results show that the problem motivating this thesis is real. Even at temperature 0, only 4 of the 27 C2 conditions produced the same diagram in all ten runs, and on average a C2 condition produced between 3.4 and 5.0 different diagrams (Table 10). This holds for all three input conditions, including the one with static analysis. Providing structural evidence therefore does not by itself make the output deterministic.

**At C2, the effect depends on the model.** The pooled C2 difference between combined and code-only (+0.055 for nodes) hides opposite effects (Table 8). For GPT-4.1 nano, adding static analysis made the output much more consistent (node consistency from 0.576 to 0.942). For GPT-4.1 mini, it made it less consistent (from 0.849 to 0.633). For Gemini 2.5 Flash, it made almost no difference. Section 8.5 shows that the drop for GPT-4.1 mini is mainly caused by repetitive outputs in one project. Even so, the three models do not react to the additional input in the same way, and with only three models it is not possible to say which reaction is typical.

**At C2, the effect also depends on the project.** The clearest improvement occurs for jhipster, where every combined run of every model produced the same set of containers (Table 9). For spring-boot-admin and apache-maven, the difference is close to zero or slightly negative. One possible explanation is that jhipster follows the standard layered structure of a Spring application, so the package summary maps almost directly onto a small set of obvious containers, whereas the larger projects offer many more ways to group packages.

**The static-analysis-only condition is the least consistent at C2.** This is the opposite of what one would expect if structural evidence simply made the model more stable. A likely reason is that the C2 task asks the model to group packages into containers, and the package summary gives no indication of how this grouping should be done; the model therefore has to decide it again in every run. With the source code available, the model has more information to rely on, such as module names in the file paths, which can help it group the packages in the same way each time.

**At C3, the results are dominated by a ceiling effect.** For jhipster and spring-boot-admin, every run of every model produced the same set of components in every condition (Table 9). The C3 target packages of these projects have only five to eight classes and almost no internal dependencies (Table 2), so there is little that can vary. The C3 differences in node consistency between the conditions therefore come entirely from apache-maven, whose target packages contain 32 and 40 classes. For apache-maven, the combined and SA-only conditions are clearly more consistent than code-only (0.908 and 0.923 versus 0.769). This is the strongest indication in the data that static analysis can stabilise the output.

**Role of the C3 instructions.** When static analysis is available, the C3 prompt asks the model to use the classes listed in the summary, whereas in the code-only condition the model has to find them in the code. Part of the higher C3 node consistency of these conditions may therefore come from this instruction rather than from the static-analysis input alone.

**Summary.** Taken together, the results do not support the claim that grounding an LLM with static-analysis output makes the generated C2 and C3 diagrams reliably more consistent. A positive effect appears for some models and projects, most clearly for the C3 diagrams of the largest project, but it is not consistent enough across the nine cells at C2 or the eighteen cells at C3 to be statistically detectable, and part of it may be caused by differences in the prompt instructions.

## 8.3 Variation Is Structural, Not Terminological

Label alignment changed the measured consistency by at most 0.032 (Table 13), and the choice of clustering threshold by at most 0.029. When two runs differ, they therefore usually do not differ only in the names they give to the same elements: they contain different elements, or group the code into containers in a different way. The variation between runs is mainly a variation in structure.

One limitation of this conclusion is that the clustering is conservative (Section 6.5.3): synonyms written with different words, such as `database` and `data store`, are usually not merged. Some of the remaining variation at C2, where labels are free text, could therefore still be a difference in wording. At C3, where most labels are class names, this is unlikely.

## 8.4 Differences Between the Models

The three models differed clearly in how stable their output was. Gemini 2.5 Flash produced on average 1.8 different C2 diagrams per condition, compared to 5.4 for GPT-4.1 mini and 5.9 for GPT-4.1 nano (Section 7.5), and it produced no repetitive outputs at all. This observation is limited to the three models studied, and a more stable model is not necessarily a model that produces more correct diagrams (Section 8.6).

## 8.5 Repetitive Outputs as a Separate Quality Problem

Thirty diagrams (3.7%) declared the same elements many times, in the most extreme case with a repetition ratio of 101 (Table 11). All of them were produced by the two GPT-4.1 models, and all four affected groups belong to the two larger projects. These outputs are valid PlantUML, so they would pass a syntax check, but they are not usable as diagrams.

The repetitive outputs strongly affect some consistency values. For GPT-4.1 mini at C2 in the combined condition, leaving them out raises node consistency from 0.633 to 0.787 (Table 12), so the drop in consistency for this model, discussed above, is mostly a result of repetitive outputs and not of ordinary variation between runs. Even without them, however, the combined condition remains below the code-only condition for this model (0.787 versus 0.849). For GPT-4.1 nano at C3, many repetitive outputs had no edges at all; since two empty edge sets count as agreement, they actually *raised* the edge consistency. Because flagged diagrams were kept in the main analysis, all reported consistency values include these effects, and the robustness check shows how large they are.

Repetitive output is therefore a quality problem of its own, which a consistency measure alone does not capture. A practical pipeline should detect it, for example with the repetition ratio used in this thesis.

## 8.6 Consistency Does Not Establish Architectural Correctness

The consistency metrics measure whether repeated runs of the same condition produce *similar* diagrams; they do not measure whether those diagrams are *correct* with respect to the system. A model could produce the same wrong diagram in all ten runs and still reach a perfect Jaccard similarity. The worked example in Section 5.5 shows a case in which the generated C3 diagram contains exactly the classes and dependencies reported by static analysis, but such agreement was not measured systematically. Likewise, the 100% syntactic validity only confirms that the PlantUML is well-formed. High consistency and syntactic validity are necessary for a diagram generator to be useful in practice, but neither shows that the diagrams are correct.

## 8.7 Correctness as Future Work

Assessing whether the generated diagrams are architecturally correct was deliberately left out of this thesis. Comparing generated labels with the names extracted by static analysis is not sufficient for this: a valid C2 container such as “Persistence Layer” does not correspond to any single name in the code, and the result of fuzzy or embedding-based matching depends strongly on the chosen threshold. A sound assessment of correctness would require a reference architecture for each project, created or validated by people who know the system, or an independent expert assessment of the generated elements and relationships. The static-analysis data produced by the pipeline could be loaded into a graph and serve as one input to such an assessment. This is proposed as future work in Section 10.

# 9. Threats to Validity and Limitations

The threats to validity are discussed using the four categories proposed by Wohlin et al. [19].

## 9.1 Construct Validity

**What consistency measures.** Node and edge Jaccard similarity measure how similar the diagrams of repeated runs are. They do not measure whether a diagram is correct, readable, or useful (Section 8.6).

**Definition of an edge.** An edge is a directed pair (source, target); the arrow type and label of a relationship are not part of it. This keeps the measure simple and comparable across conditions.

**Empty edge sets.** Two runs without any edges are counted as full agreement ($J = 1$). To make the effect of this convention visible, the edge score is also reported without such pairs (Section 7.2).

**Label alignment.** The similarity clustering decides which labels of different runs count as the same node. To reduce the influence of this step, three thresholds are reported, and at C3 the clusters are checked automatically against the class names extracted by Spoon (Section 7.7). The results change by at most 0.032 between raw labels, format-normalised labels, and the three thresholds, so the conclusions do not depend on this step.

**Containers are represented by packages.** In the C4 model, a container is a separately deployable or runnable unit. A Java repository does not state this explicitly, so the pipeline gives the model a package-level summary and asks it to group packages into containers. The C2 diagrams are therefore closer to a logical grouping of the code than to a deployment view.

## 9.2 Internal Validity

**Prompt instructions.** Some instructions differ between the conditions because they refer to input that exists only in some of them (Section 5.3). At C3, for example, the conditions with static analysis refer to the listed classes. Differences between the conditions therefore reflect the input together with these instructions.

**Static-analysis output.** The static-analysis output is used as input to the model, so its quality may influence the generated diagrams. It is, however, identical for every run of a condition, and the study measures consistency between runs rather than correctness.

**Selection of the C3 targets.** The C3 target packages were selected once with a fixed rule (Section 5.2.5) and are identical for all models and conditions. A different selection rule could lead to different C3 results.

**Analysis scripts.** The results depend on the correctness of the scripts that parse the diagrams and compute the metrics. To reduce this risk, the parser writes every line it does not recognise to an audit list instead of ignoring it, and the scripts and all intermediate results are published (Section 7.9).

**External APIs.** The models are accessed through APIs whose internal behaviour cannot be inspected. The exact model versions and all parameters are reported (Section 6.2).

## 9.3 External Validity

The study covers three open-source Java projects, three LLMs, and the C2 and C3 levels of the C4 model. The findings may not apply to other programming languages, to industrial systems, to other models or later versions of the same models, or to the C1 and C4 levels. One prompt template per condition was used, and other prompts may lead to different consistency.

## 9.4 Conclusion Validity

**Number of cells.** The statistical tests compare 9 cells at C2 and 18 at C3. With this number of cells, only effects that point in the same direction in almost all cells can be detected. The absence of a significant result therefore means that no consistent effect was found, not that static analysis has no effect.

**Dependence between cells.** Cells that share a model or a project are not fully independent. The per-model and per-project tables are reported so that such dependencies can be seen.

**Multiple comparisons.** The $p$-values were adjusted with the Holm method within groups of tests that were fixed before the tests were run (Section 6.5).

# 10. Conclusions and Future Work

## Summary of Contributions

This thesis studied the run-to-run variability of LLM-generated architectural views, using the Container (C2) and Component (C3) levels of the C4 model as a concrete case. For RQ1, we designed and implemented a five-stage pipeline (static-analysis extraction, transformation into LLM input, prompt construction, LLM invocation, and output processing; Section 5) that produces PlantUML diagrams from source code, from static-analysis output, or from both. For RQ2, the pipeline was run in 81 conditions (3 projects $\times$ 3 LLMs $\times$ 3 input conditions, with one C2 diagram and two C3 targets per project), ten times each, which produced 810 diagrams. Consistency was measured with node and edge Jaccard similarity after a fully automated label alignment, and the conditions were compared with statistical tests over model $\times$ project $\times$ target cells. All scripts, inputs, and generated diagrams are publicly available.

## Answering the Research Questions

**RQ1** asked how static-analysis information can be structured and combined with a large language model to generate C2 and C3 diagrams. The pipeline answers this question with a plain-text summary of the Spoon output (package-level for C2, class-level for C3), the complete source code loaded identically for all conditions that use it, fixed prompt templates with explicit syntax rules, and an automatic extraction of the PlantUML block. All 810 generated diagrams were syntactically valid without manual repair. This shows that the pipeline works reliably end to end; it does not show that the diagrams are architecturally correct, which was not evaluated.

**RQ2** asked to what extent providing static-analysis output affects the consistency of the generated diagrams across repeated runs, compared to source code alone. The answer is that no reliable effect was found. The main findings are:

- *Generation is not deterministic.* Even at temperature 0, only 4 of 27 C2 conditions produced the same diagram in all ten runs, including conditions that received static analysis.
- *No significant effect of static analysis.* The combined condition had a higher mean consistency than the code-only condition at both levels, but none of the differences was statistically significant, and at C2 the effect was positive for one model, negative for another, and negligible for the third.
- *C3 is affected by a ceiling effect.* For two of the three projects, every run produced the same components in every condition. The only clear C3 improvement was found for the largest project.
- *Variation is structural.* Label alignment changed the measured consistency by at most 0.032, so the runs differ mainly in which elements and relationships they contain, not in how they name them.
- *The models differ in stability.* Gemini 2.5 Flash produced far fewer distinct diagrams than the two GPT-4.1 models and no repetitive outputs, whereas the GPT-4.1 models produced all 30 repetitive outputs.

Consistency is a necessary property of a useful diagram generator, but not a sufficient one: these results say nothing about whether the generated diagrams correctly describe the systems (Section 8.6).

## Future Work

**Correctness of the diagrams.** The most direct extension is to evaluate whether the generated elements and relationships correspond to the real structure of the system. This requires a reference architecture for each project, created or validated by people who know the system, or an independent assessment by software architects. The static-analysis output of the pipeline, loaded into a graph, could be one input to such an evaluation.

**Larger C3 targets and more projects.** Selecting larger target packages, and evaluating more projects and more models, including open-weight models run locally, would give the statistical tests more power and show how far the findings generalise.

**Containers from build modules.** Containers were represented by groups of packages. Deriving container candidates from build modules (for example Maven or Gradle modules) would bring the C2 diagrams closer to the C4 definition of a container.

**Richer comparison of edges.** Edge similarity in this study considers only the source and target of a relationship. Including the type or label of a relationship would distinguish diagrams that agree on which elements are connected from diagrams that also agree on how they are connected.

**Repetitive outputs.** The repetitive outputs of the GPT-4.1 models occurred only for the two larger projects. Investigating why they occur, and detecting or preventing them automatically, would make the pipeline more robust.

# References


[1] R. Koschke, "Software architecture recovery and reconstruction: Techniques and tools," in *Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 2009, pp. 3–10.

[2] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 3rd ed. Addison-Wesley, 2013.

[3] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford, *Documenting Software Architectures: Views and Beyond*, 2nd ed. Addison-Wesley, 2010.

[4] S. Brown, *The C4 Model for Visualising Software Architecture*, 1st ed. Leanpub, 2022.

[5] S. Migliorini, R. Verdecchia, I. Malavolta, P. Lago, and E. Vicario, "Architectural views: The state of practice in open-source software projects," in *Software Architecture – 18th European Conference on Software Architecture (ECSA 2024)*, ser. Lecture Notes in Computer Science. Springer, 2024.

[6] K. Szczepanik and J. A. Chudziak, "Collaborative LLM agents for C4 software architecture design automation," arXiv preprint arXiv:2510.22787, 2025.

[7] A. Savidis and C. Savaki, "Software architecture mining from source code with dependency graph clustering and visualization," in *Proceedings of the 17th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP) – IVAPP*. SciTePress, 2022, pp. 179–186.

[8] A. Radermacher, M. Didonet Del Fabro, S. Anvar, F. Chateau, and Y. Abdeljalil, "AI-based recovery of high-level models in the context of reverse engineering of C++ to UML," in *International Conference on Enterprise Information Systems (ICEIS)*. Springer, 2024, pp. 303–322.

[9] C. Raibulet, F. Arcelli Fontana, and M. Zanoni, "Model-driven reverse engineering approaches: A systematic literature review," *IEEE Access*, vol. 5, pp. 14 516–14 542, 2017.

[10] H. A. Siala, K. Lano, and H. Alfraihi, "Model-driven approaches for reverse engineering: A systematic literature review," *IEEE Access*, vol. 12, pp. 62 558–62 580, 2024.

[11] H. Siala and K. Lano, "A comparison of large language models and model-driven reverse engineering for reverse engineering," *Frontiers in Computer Science*, vol. 7, p. 1516410, 2025.

[12] H. A. Siala and K. Lano, "Using large language models to extract UML class diagrams from Java programs," in *Proceedings of the 8th International Conference on Software and System Engineering (ICoSSE)*. IEEE, 2025, pp. 70–74.

[13] ——, "Using LLMs to extract UML class diagrams from Java and Python programs: An empirical study," *CEUR Workshop Proceedings*, vol. 4122, 2025.

[14] V. Campanello, S. Shahbaz, V. Indykov, and D. Strüber, "On the use of GPT-4 in the reverse engineering of class diagrams," *Journal of Object Technology*, vol. 24, no. 2, pp. 2:1–14, 2025.

[15] D. Amalfitano, M. De Luca, T. Santilli, P. Pelliccione, and A. R. Fasolino, "Automated software architecture design recovery from source code using LLMs," in *European Conference on Software Architecture (ECSA)*. Springer, 2025, pp. 73–89.

[16] A. Hatahet, C. Knieke, and A. Rausch, "Generating software architecture description from source code using reverse engineering and large language model," arXiv preprint arXiv:2511.05165, 2025.

[17] A. R. Hevner, S. T. March, J. Park, and S. Ram, "Design science in information systems research," *MIS Quarterly*, vol. 28, no. 1, pp. 75–105, 2004.

[18] K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee, "A design science research methodology for information systems research," *Journal of Management Information Systems*, vol. 24, no. 3, pp. 45–77, 2007.

[19] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering.* Springer, 2012.

[20] A. Jedlitschka, M. Ciolkowski, and D. Pfahl, "Reporting experiments in software engineering," in *Guide to Advanced Empirical Software Engineering.* Springer, 2008, pp. 201–228.

# A. Complete Prompt Templates

This appendix lists the complete prompts used for each of the three input conditions (source-code-only, static-analysis-only, and combined) and both diagram levels (C2 and C3), as referenced in Section 5.3. Every request consists of two messages: a short *system instruction*, which sets the role of the model, and the *user prompt*, which contains the task and the input. Both are shown for each condition below. Placeholders in curly braces (e.g. {`project_name`}, {`source_code`}) are filled in by the generation scripts with the project-specific inputs described in Section 5.2. The same prompt is used, unchanged, for all ten runs, all three models, and all three projects of a given condition and level.

## A.1 Source-Code-Only, C2

```
=== SYSTEM INSTRUCTION ===
You are a software architect specialising in C4 model diagrams. Generate
only valid PlantUML. Never fabricate containers or relationships not
supported by the provided source code. Never use colour names after # and
never nest rectangles inside rectangles.

=== USER PROMPT ===
You are a software architect. Generate a C4 Container (C2) diagram
in PlantUML format for the Java project: {project_name}.

The following is the source code of the project. Analyse it to identify the
major deployable/runnable units (containers) in the project and the
dependencies between them.
Use this source code as the authoritative source -- do not invent containers
or relationships that are not supported by it.

--- SOURCE CODE ---
{source_code}

INSTRUCTIONS:
1. Group related packages into a reasonable number of logical containers.
   Do NOT create more than 20 containers total. Do NOT create separate
   containers for individual subpackages, helper classes, or exceptions --
   group them into their parent container.
2. Label each container with its technology stack where identifiable from
   the source code.
3. Draw inter-container relationships only where the source code shows
   clear usage between packages.
4. Keep the diagram at the C2 level: containers, not individual classes.
5. Output ONLY valid PlantUML. Do NOT use !include or !define directives.
6. Do NOT nest rectangles inside rectangles.
7. Do NOT put [ ] component syntax inside rectangles.
8. Use 'rectangle' for containers and '-->' for relationships.
9. Keep aliases short (no spaces). Add a title with the project name.

PLANTUML COLOR RULES -- CRITICAL:
- Do NOT use color names after #. Colors are NOT supported in rectangle syntax.
- Do NOT write: rectangle "Name" as alias #ColorName
- ONLY valid rectangle syntax: rectangle "Label" as alias
- Relationship labels must be quoted: alias1 --> alias2 : "label"
- Aliases must be short, with no spaces (e.g. 'webapi', 'service').

VALID EXAMPLE:
@startuml
title MyProject - Container Diagram

rectangle "Web API\n[Spring MVC]" as webapi
rectangle "Service Layer\n[Spring]" as service
rectangle "Database\n[JPA/Hibernate]" as db

webapi --> service : "calls"
service --> db : "reads/writes"
@enduml
```

```
Now generate the diagram:
@startuml
```

## A.2 Source-Code-Only, C3

```
=== SYSTEM INSTRUCTION ===
You are a software architect specialising in C4 Component (C3) diagrams.
Generate only valid PlantUML. Never fabricate components or relationships
not supported by the provided source code. Never use colour names after #
and never put [ ] syntax inside package or rectangle blocks.

=== USER PROMPT ===
You are a software architect. Generate a C4 Component (C3) diagram
in PlantUML format for the Java project: {project_name}.

A specific container has been selected for decomposition.
Container package : {target_pkg}

The source code below contains the classes belonging to this container
(and the surrounding project). Analyse it to identify the classes, their
responsibilities, and the dependencies between them. Focus specifically on
classes belonging to the '{target_pkg}' package.
Use this source code as the authoritative source -- do not invent components
or relationships that are not supported by it.

--- SOURCE CODE ---
{source_code}

INSTRUCTIONS:
1. Focus ONLY on classes belonging to the '{target_pkg}' package.
   Do NOT include classes from other packages as components.
2. Each class or interface in the target container becomes a component
   inside the container boundary.
3. Label each component with its architectural role (e.g. Service,
   Repository, Entity, Controller) where identifiable from naming or
   annotations.
4. Draw relationships only where the source code clearly shows a usage or
   dependency relationship.
5. Show inheritance (extends/implements) as relationships where meaningful.
6. Output ONLY valid PlantUML. Do NOT use !include or !define directives.
7. Wrap all components in a 'package' block representing the container
   boundary.
8. Use 'rectangle' for components, '-->' for dependencies. Add a title.

PLANTUML SYNTAX RULES -- CRITICAL:
- Do NOT use color names after #. Colors are NOT supported on rectangle.
- Do NOT write: rectangle "Name" as alias #ColorName
- Do NOT put [ ] component syntax inside package or rectangle blocks.
- ONLY valid rectangle syntax: rectangle "Label\n<<role>>" as alias
- Aliases must be short with no spaces (e.g. 'ownerSvc', 'petRepo').
- Relationship labels must be quoted: ca --> cb : "uses"

VALID EXAMPLE:
@startuml
title ContainerName - Component Diagram

package "ContainerName" {
    rectangle "OwnerService\n<<Service>>" as ownerSvc
    rectangle "OwnerRepository\n<<Repository>>" as ownerRepo
    rectangle "Owner\n<<Entity>>" as owner
    ownerSvc --> ownerRepo : "uses"
    ownerRepo --> owner : "persists"
}
@enduml

Now generate the diagram:
@startuml
```

## A.3 Static-Analysis-Only, C2

```
=== SYSTEM INSTRUCTION ===
You are a software architect specialising in C4 model diagrams. Generate
only valid PlantUML. Never fabricate containers or relationships not
supported by the provided static analysis data. Never use colour names
after # and never nest rectangles inside rectangles.

=== USER PROMPT ===
You are a software architect. Generate a C4 Container (C2) diagram
in PlantUML format for the Java project: {project_name}.

The following structural information was extracted by static analysis
(Spoon). Analyse it to identify the major deployable/runnable units
(containers) in the project and the dependencies between them. Each entry
shows a package with its class count, centrality score (inbound + outbound
reference totals), detected technologies, and outbound dependencies.
Use this static analysis data as the authoritative source -- do not invent
containers or relationships that are not supported by it.

--- STATIC ANALYSIS DATA ---
{formatted_packages}
{dependency_list}

INSTRUCTIONS:
1. Group related packages into a reasonable number of logical containers.
   Do NOT create more than 20 containers total. Do NOT create separate
   containers for individual subpackages, helper classes, or exceptions --
   group them into their parent container.
2. Use technology labels detected by the static analysis wherever available.
3. Draw inter-container relationships only where static analysis shows
   package-level dependencies.
4. Keep the diagram at the C2 level: containers, not individual classes.
5. Output ONLY valid PlantUML. Do NOT use !include or !define directives.
6. Do NOT nest rectangles inside rectangles.
7. Do NOT put [ ] component syntax inside rectangles.
8. Use 'rectangle' for containers and '-->' for relationships.
9. Keep aliases short (no spaces). Add a title with the project name.

PLANTUML COLOR RULES -- CRITICAL:
- Do NOT use color names after #. Colors are NOT supported in rectangle syntax.
- Do NOT write: rectangle "Name" as alias #ColorName
- ONLY valid rectangle syntax: rectangle "Label" as alias
- Relationship labels must be quoted: alias1 --> alias2 : "label"
- Aliases must be short, with no spaces (e.g. 'webapi', 'service').

VALID EXAMPLE:
@startuml
title MyProject - Container Diagram

rectangle "Web API\n[Spring MVC]" as webapi
rectangle "Service Layer\n[Spring]" as service
rectangle "Database\n[JPA/Hibernate]" as db

webapi --> service : "calls"
service --> db : "reads/writes"
@enduml

Now generate the diagram:
@startuml
```

## A.4 Static-Analysis-Only, C3

```
=== SYSTEM INSTRUCTION ===
You are a software architect specialising in C4 Component (C3) diagrams.
Generate only valid PlantUML. Never fabricate components or relationships
not supported by the provided static analysis data. Never use colour names
after # and never put [ ] syntax inside package or rectangle blocks.
```

```
=== USER PROMPT ===
You are a software architect. Generate a C4 Component (C3) diagram
in PlantUML format for the Java project: {project_name}.

A specific container has been selected for decomposition.
Container package : {target_pkg}

The classes inside this container were extracted by static analysis (Spoon).
Analyse them to identify the classes, their responsibilities, and the
dependencies between them. Each entry includes: type, architectural role
(derived from annotations), inheritance, and internal dependencies.
Use this static analysis data as the authoritative source -- do not invent
components or relationships that are not supported by it.

--- STATIC ANALYSIS DATA ---
{formatted_classes}

INSTRUCTIONS:
1. Focus ONLY on classes belonging to the '{target_pkg}' package. Do NOT
   include classes from other packages as components.
2. Each class/interface becomes a component inside the container boundary.
3. Use the 'role' field as a label or stereotype for each component.
4. Draw relationships only where 'depends_on_internal' lists a dependency.
5. Show inheritance (extends/implements) as relationships where meaningful.
6. Output ONLY valid PlantUML. Do NOT use !include or !define directives.
7. Wrap all components in a 'package' block representing the container
   boundary.
8. Use 'rectangle' for components, '-->' for dependencies. Add a title.

PLANTUML SYNTAX RULES -- CRITICAL:
- Do NOT use color names after #. Colors are NOT supported on rectangle.
- Do NOT write: rectangle "Name" as alias #ColorName
- Do NOT put [ ] component syntax inside package or rectangle blocks.
- ONLY valid rectangle syntax: rectangle "Label\n<<role>>" as alias
- Aliases must be short with no spaces (e.g. 'ownerSvc', 'petRepo').
- Relationship labels must be quoted: ca --> cb : "uses"

VALID EXAMPLE:
@startuml
title ContainerName - Component Diagram

package "ContainerName" {
    rectangle "OwnerService\n<<Service>>" as ownerSvc
    rectangle "OwnerRepository\n<<Repository>>" as ownerRepo
    rectangle "Owner\n<<Entity>>" as owner
    ownerSvc --> ownerRepo : "uses"
    ownerRepo --> owner : "persists"
}
@enduml

Now generate the diagram:
@startuml
```

## A.5 Combined, C2

```
=== SYSTEM INSTRUCTION ===
You are a software architect specialising in C4 model diagrams. Generate
only valid PlantUML. Never fabricate containers or relationships not
supported by the provided static analysis or source code. Never use colour
names after # and never nest rectangles inside rectangles.

=== USER PROMPT ===
You are a software architect. Generate a C4 Container (C2) diagram
in PlantUML format for the Java project: {project_name}.

You have TWO sources of information about this project. Analyse them to
identify the major deployable/runnable units (containers) and the
dependencies between them. Treat Source 1 (static analysis) as the
authoritative structural ground truth. Use Source 2 (source code) to
```

```
enrich container descriptions, understand responsibilities, and clarify
technology usage.
Do NOT invent containers or relationships not supported by either source.

--- SOURCE 1: STATIC ANALYSIS DATA ---
{package_text}
{dep_text}

--- SOURCE 2: SOURCE CODE ---
{source_code}

INSTRUCTIONS:
1. Group related packages into a reasonable number of logical containers.
   Do NOT create more than 20 containers total. Do NOT create separate
   containers for individual subpackages, helper classes, or exceptions --
   group them into their parent container.
2. Use technology labels detected by the static analysis wherever available.
3. Draw inter-container relationships only where static analysis shows
   package-level dependencies.
4. Keep the diagram at the C2 level: containers, not individual classes.
5. Output ONLY valid PlantUML. Do NOT use !include or !define directives.
6. Do NOT nest rectangles inside rectangles.
7. Do NOT put [ ] component syntax inside rectangles.
8. Use 'rectangle' for containers and '-->' for relationships.
9. Keep aliases short (no spaces). Add a title with the project name.

PLANTUML COLOR RULES -- CRITICAL:
- Do NOT use color names after #. Colors are NOT supported in rectangle syntax.
- Do NOT write: rectangle "Name" as alias #ColorName
- ONLY valid rectangle syntax: rectangle "Label" as alias
- Relationship labels must be quoted: alias1 --> alias2 : "label"
- Aliases must be short, with no spaces (e.g. 'webapi', 'service').

VALID EXAMPLE:
@startuml
title MyProject - Container Diagram

rectangle "Web API\n[Spring MVC]" as webapi
rectangle "Service Layer\n[Spring]" as service
rectangle "Database\n[JPA/Hibernate]" as db

webapi --> service : "calls"
service --> db : "reads/writes"
@enduml

Now generate the diagram:
@startuml
```

## A.6 Combined, C3

```
=== SYSTEM INSTRUCTION ===
You are a software architect specialising in C4 Component (C3) diagrams.
Generate only valid PlantUML. Never fabricate components or relationships
not supported by the provided static analysis or source code. Never use
colour names after # and never put [ ] syntax inside package or rectangle
blocks.

=== USER PROMPT ===
You are a software architect. Generate a C4 Component (C3) diagram
in PlantUML format for the Java project: {project_name}.

A specific container has been selected for decomposition.
Container package : {target_pkg}

You have TWO sources of information about this container. Analyse them to
identify the classes, their responsibilities, and the dependencies between
them. Treat Source 1 (static analysis) as the authoritative structural
ground truth. Use Source 2 (source code) to enrich component descriptions
and clarify responsibilities.
```

```
Do NOT invent components or relationships not supported by either source.

--- SOURCE 1: STATIC ANALYSIS DATA ---
{formatted_classes}

--- SOURCE 2: SOURCE CODE ---
{source_code}

INSTRUCTIONS:
1. Focus ONLY on classes belonging to the '{target_pkg}' package that are
   listed in Source 1. Do NOT add components that are not listed in
   Source 1, and do NOT include classes from other packages.
2. Each class/interface from Source 1 becomes a component inside the
   container boundary.
3. Use the 'role' field as a label or stereotype for each component.
4. Draw relationships only where 'depends_on_internal' lists a dependency
   OR where the source code clearly shows a usage relationship.
5. Show inheritance (extends/implements) as relationships where meaningful.
6. Output ONLY valid PlantUML. Do NOT use !include or !define directives.
7. Wrap all components in a 'package' block representing the container
   boundary.
8. Use 'rectangle' for components, '-->' for dependencies. Add a title.

PLANTUML SYNTAX RULES -- CRITICAL:
- Do NOT use color names after #. Colors are NOT supported on rectangle.
- Do NOT write: rectangle "Name" as alias #ColorName
- Do NOT put [ ] component syntax inside package or rectangle blocks.
- ONLY valid rectangle syntax: rectangle "Label\n<<role>>" as alias
- Aliases must be short with no spaces (e.g. 'ownerSvc', 'petRepo').
- Relationship labels must be quoted: ca --> cb : "uses"

VALID EXAMPLE:
@startuml
title ContainerName - Component Diagram

package "ContainerName" {
    rectangle "OwnerService\n<<Service>>" as ownerSvc
    rectangle "OwnerRepository\n<<Repository>>" as ownerRepo
    rectangle "Owner\n<<Entity>>" as owner
    ownerSvc --> ownerRepo : "uses"
    ownerRepo --> owner : "persists"
}
@enduml

Now generate the diagram:
@startuml
```