POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Integrating Large Language Models in Microservice Architecture Decomposition

Tesi di Laurea Magistrale in
Computer Science and Engineering - Ingegneria Informatica

Author: **Raffaele Russo**

Student ID: 11016624
Advisor: Prof. Giovanni Ennio Quattrocchi
Academic Year: 2025-26

Abstract

In the early stages of software development, microservices architecture design is often driven by informal and semi-structured functional requirements, typically captured as user stories, and relies heavily on the experience of architects. In contrast, optimization-based decomposition and evaluation frameworks require structured specifications describing application operations, data access patterns, and architectural constraints. This misalignment creates an "input gap" that limits the applicability of such tools in early-stage contexts. This thesis explores the integration of Large Language Models (LLMs) into an optimization-based workflow to automate the transformation of semi-structured functional requirements into formal architectural specifications and to make implicit architectural constraints explicit. The proposed approach consists of a three-stage experimental pipeline. First, an LLM extracts a functional structure from user stories and the domain entity model, identifying read and write accesses to relevant entities and attributes for each operation. Second, an additional generation step infers co-location constraints between operations, representing architectural decisions not captured by optimization criteria alone. Third, the resulting specification is evaluated using Cromlech, a multi-objective optimization framework for microservice decomposition, which synthesizes candidate service partitions and assesses them through quantitative metrics such as Cohesion, Communication Cost, and Total Value. The methodology is evaluated on the Trainticket and Tutored case studies, comparing different LLMs and analyzing both deterministic (T=0.0) and stochastic (T=1.0) configurations to estimate performance and stability. The results show that inferred constraints significantly influence the structure of the generated architectures and that models differ substantially in terms of consistency and robustness. The main contribution of this work is a reproducible methodology for integrating automated text extraction with optimization-based architectural synthesis, enabling systematic microservice decomposition even in the absence of detailed technical specifications.

Keywords: Large Language Models, Software Architecture, Microservices, Optimization-based Decomposition, Prompt Engineering

Abstract in lingua italiana

Nelle fasi inziali dello sviluppo software, la progettazione dell'architettura dei microservizi è spesso guidata da requisiti funzionali informali e semi-strutturati, solitamente rappresentati come user story, e si basa in larga misura sull'esperienza degli architetti. Al contrario, i framework di scomposizione e valutazione basati sull'ottimizzazione richiedono specifiche strutturate che descrivono operazioni applicative, i modelli di accesso ai dati e i vincoli architettonici. Questo disallineamento crea un "input gap" che ne limita l'applicabilità nelle fasi iniziali. Questa tesi esplora l'integrazione di Large Language Models (LLM) in un flusso di lavoro basato sull'ottimizzazione per automatizzare la trasformazione di requisiti funzionali semi-strutturati in specifiche architettoniche formali e per rendere espliciti i vincoli architettonici impliciti. L'approccio proposto consiste in una pipeline sperimentale in tre fasi. In primo luogo, un LLM estrae una struttura funzionale dalle user story e dal modello di entità di dominio, identificando gli accessi in lettura e scrittura alle entità e agli attributi rilevanti per ciascuna operazione. In secondo luogo, un ulteriore passaggio di generazione deduce vincoli di co-locazione tra le operazioni, rappresentando decisioni architettoniche non catturate solo da criteri di ottimizzazione. In terzo luogo, la specifica risultante viene valutata utilizzando Cromlech, un framework di ottimizzazione multiobiettivo per la scomposizione dei microservizi, che sintetizza le partizioni di servizio candidate e le valuta attraverso metriche quantitative come coesione, costo di comunicazione e valore totale. La metodologia viene valutata sui casi di studio Trainticket e Tutored, confrontando diversi LLM e analizzando configurazioni sia deterministiche (T=0.0) che stocastiche (T=1.0) per stimare prestazioni e stabilità. I risultati mostrano che i vincoli dedotti influenzano significativamente la struttura delle architetture generate e che i modelli differiscono sostanzialmente in termini di coerenza e robustezza. Il contributo principale di questo lavoro è una metodologia riproducibile per integrare l'estrazione automatizzata del testo con la sintesi architettonica basata sull'ottimizzazione, consentendo la scomposizione sistematica dei microservizi anche in assenza di specifiche tecniche dettagliate.

Parole chiave: Large Language Models, Software Architecture, Microservices, Optimization-based Decomposition, Prompt Engineering

Contents

Introduction

Software architecture design represents one of the most complex and critical activities in the development of modern information systems. In particular, the adoption of microservices architectures has introduced significant benefits in terms of scalability, maintainability, and independent component evolution. At the same time, however, it has increased the complexity of architectural decisions to be made during the early stages of the software lifecycle. Defining an appropriate system decomposition requires a deep understanding of functional requirements, logical dependencies, and data access patterns-information that is rarely fully available at the beginning of a project.

In industrial practice, early architectural decisions are often derived from informal requirements, typically expressed in the form of *user stories*. While these artifacts effectively describe the expected system behavior from the user's perspective, they do not provide a structured representation of relationships among application components or of implicit architectural constraints. As a result, architectural design largely depends on the experience of software architects, making the process costly, unsystematic, and difficult to reproduce.

In recent years, several optimization-based frameworks have addressed the architectural decomposition problem by proposing formal approaches to derive microservices architectures from structured system models. In particular, tools such as *Cromlech* [1] formulate decomposition as a mathematical optimization problem, balancing criteria such as cohesion, coupling, and operational costs. However, the effectiveness of such approaches strongly depends on the availability of detailed and formalized input specifications, which are rarely present in the early stages of development.

In parallel, recent advances in *Large Language Models* (LLMs) have opened new opportunities for automated support in cognitively demanding tasks. Thanks to their ability to interpret and process large amounts of unstructured text, LLMs have emerged as potential assistants for software engineers in requirements analysis and structured knowledge extraction. When used in a controlled and informed manner, these models can reduce the manual effort associated with complex and error-prone activities, without replacing the decision-making role of the architect.

In this context, employing LLMs as architectural reasoning assistants represents a promising yet still underexplored research direction. A key open question concerns how to integrate their inferential capabilities into structured engineering workflows while ensuring verifiability, decision stability, and objective evaluation of the produced results. This issue becomes particularly critical when model-generated decisions directly influence the structure of the resulting software architecture.

This thesis addresses these challenges by proposing an experimental approach for integrating LLMs into the early-stage design of microservices architectures. The goal is to bridge the gap between informal requirements and formal architectural models, enabling the automated synthesis and evaluation of microservices architectures starting solely from user stories. In particular, LLMs are employed to extract functional specifications and to infer implicit architectural constraints, which encode co-location requirements among functionalities not explicitly represented in optimization models.

The generated specifications are subsequently evaluated using the Cromlech framework, which produces concrete microservices decompositions and assesses their quality through quantitative metrics. The entire process is structured as a multi-stage experimental pipeline, designed to analyze both the effectiveness of the generated solutions and the stability of architectural decisions suggested by stochastic language models.

The main contributions of this thesis can be summarized as follows:

- the definition of a methodology for generating microservices architectures from informal requirements through the integration of LLMs and optimization frameworks;
- the use of LLMs to infer co-location architectural constraints to support the decomposition process;
- an empirical evaluation of the impact and stability of architectural decisions suggested by different language models;
- a reproducible experimental pipeline for the quantitative and comparative analysis of the resulting architectures.

1 | State of the Art & Background

This chapter presents the fundamental concepts and the state of the art required to understand the context in which this thesis is positioned. Starting from an overview of traditional architectural paradigms, it discusses the challenges associated with software system decomposition and the emerging role of Large Language Models (LLMs) in automating software engineering processes.

1.1. Software Architectural Patterns

The choice of an architectural pattern represents one of the most critical decisions in the lifecycle of a software system, as it directly influences its non-functional properties, such as scalability, maintainability, and resilience. Over the past decades, the evolution of business requirements and the transition toward cloud-native paradigms have led organizations to reconsider traditional models in favor of more flexible and distributed structures[2].

This section analyzes the two predominant paradigms in the modern landscape: monolithic architecture and microservices architecture. The discussion begins with the fundamental characteristics of the monolith and its natural evolutionary limitations, and then presents microservices as both a technological and organizational response to these challenges. Finally, the intrinsic difficulties associated with the transition between these two models are examined, laying the groundwork for understanding the need for automated decomposition tools.

1.1.1. Monolithic Architecture

Monolithic architecture represents the classical and historically predominant paradigm in software system development. In this model, the entire system is conceived as a single indivisible unit [1], where business logic, data access, user interfaces, and supporting components are tightly intertwined within a single deployable artifact[3]. Typically, a monolithic application runs as a single process, which may be scaled by replicating the entire application across multiple instances.

Structural Characteristics and Design Philosophy According to classical literature [4], a monolith is characterized by tight coupling among its internal components. Although the source code may be logically organized into distinct modules or packages, this separation is often purely structural: at runtime, all modules share the same address space and memory, operating within a single application process.

A central element of the monolithic design philosophy is the adoption of a centralized data model. The application typically interacts with a centralized database, often relational, which simplifies transaction management through support for ACID properties [5]. However, this choice may introduce architectural bottlenecks, limiting the evolutionary and operational independence of different functional modules.

Among the main advantages of monolithic architecture is its conceptual and implementation simplicity. The presence of a single codebase facilitates early development phases, as well as debugging and testing activities, reducing the complexity associated with communication between distributed components. Moreover, the absence of network overhead for internal calls often results in lower latency compared to distributed architectures.

The "Big Ball of Mud" Phenomenon As complexity increases and maintenance iterations accumulate, monolithic systems tend to degenerate into what Foote and Yoder [6] defined as a *"Big Ball of Mud"*. This term describes a system in which the lack of clear modular boundaries leads to a tangled web of dependencies. In such scenarios, understanding the logical flow becomes increasingly burdensome for developers, and the risk of introducing regressions during bug fixes or feature additions grows exponentially, progressively eroding the original architectural structure.

Technical Limitations and Scalability The main drawbacks of the monolithic model become particularly evident under high workloads or when the system is subject to frequent functional evolution [7]. These limitations can be summarized as follows:

- *Undifferentiated Scalability:* It is not possible to selectively scale a single critical functionality. Instead, the entire monolithic system must be replicated across multiple instances, leading to inefficient use of resources such as CPU and memory for components that do not require additional capacity.
- *Fragility and Reliability:* Since the application runs as a single process, a malfunction in a non-critical module, such as a memory leak, may compromise the stability of the entire system, effectively creating a single point of failure.
- *Barrier to Innovation:* The adoption of new technologies is hindered by the need

to maintain compatibility with the entire existing stack, forcing teams to rely on potentially outdated languages or frameworks for new developments.

Maintainability and Lifecycle From a development process perspective, monolithic architecture may introduce what is commonly referred to as a *deployment bottleneck*. Because the entire system must be tested, validated, and deployed as a single artifact, even minor modifications require full build and release cycles, increasing deployment complexity and duration [8]. These dynamics are widely recognized as a major obstacle to the adoption of Continuous Integration and Continuous Delivery practices, particularly in large codebases and multi-team environments [9].

The limitations discussed above, particularly in terms of scalability, maintainability, and release efficiency, have progressively highlighted the inadequacy of monolithic architecture in complex and dynamic environments. These considerations have motivated the adoption of alternative architectural approaches aimed at greater modularity, component autonomy, and operational flexibility.

1.1.2. Microservices Architecture

In contrast to the monolithic model, microservices architecture has emerged as a reference paradigm for the development of scalable and maintainable cloud-native systems. According to Lewis and Fowler [10], a microservices architecture structures an application as a suite of small, independently deployable services, each running in its own process and communicating with others through lightweight mechanisms, typically HTTP-based APIs (REST) or message brokers such as Kafka or RabbitMQ.

Core Principles The success of this paradigm is grounded in several technical principles aimed at addressing the intrinsic limitations of monolithic systems:

- *Decoupling and Independence:* Each microservice is an autonomous deployment unit. This enables individual services to be updated, replaced, or scaled without redeploying the entire application.
- *Data Decentralization:* Each microservice manages its own data independently [5, 11], reducing undesired coupling and ensuring that schema modifications within one service do not directly impact others.
- *Polyglot Governance:* Development teams are free to choose the most appropriate technology stack for each microservice, including programming languages, frameworks,

and databases. This flexibility facilitates technological evolution and allows services to be tailored to specific functional and performance requirements [12].

The Concept of Bounded Context The primary challenge in microservices design lies not in the underlying technologies, but in defining appropriate service boundaries. Architectural best practices recommend adopting the concept of *Bounded Context*, derived from Domain-Driven Design (DDD) [4], so that each service corresponds to a well-defined business domain boundary characterized by high internal cohesion and minimal external dependencies.

Trade-offs and Distributed Complexity Despite the benefits in terms of agility and scalability, adopting a microservices architecture shifts complexity from internal application logic to distributed communication management. Issues such as data consistency, coordination of distributed transactions, service monitoring, and network latency introduce significant design challenges [13]. These aspects require careful architectural decomposition to avoid excessive inter-service dependencies and operational overhead.

Scalability and Fault Isolation Microservices enable selective scaling of individual services based on workload demands, optimizing resource utilization without replicating the entire application [4]. Furthermore, decomposing the system into autonomous units improves fault isolation: a malfunction or performance degradation in a single service does not necessarily compromise the entire system, thereby reducing the risk of global outages and facilitating targeted maintenance interventions.

Deployment and Continuous Delivery The independence of microservices supports rapid and frequent release cycles, facilitating Continuous Integration and Continuous Delivery (CI/CD) practices [4]. Updates or bug fixes can be applied to individual services without redeploying the whole application, reducing deployment time and operational complexity.

Observability and Monitoring The distributed nature of microservices introduces new operational challenges, making advanced monitoring and centralized logging strategies essential. Techniques such as distributed tracing, per-service metrics collection, and alerting systems provide visibility into the overall system state and enable timely anomaly detection [4]. These practices help mitigate the complexity inherent in distributed systems, ensuring long-term reliability and maintainability[14].

1.1.3. Monolithic vs Microservices Architecture

Monolithic and microservices architectures represent two fundamentally different approaches to software system design, each characterized by specific architectural trade-offs. Comparing these paradigms helps clarify the technical and organizational motivations that have driven the increasing adoption of microservices in complex and dynamic environments.

Structure and Coupling In the monolithic model, application components share the same execution space and are tightly coupled at both code and data levels. Although logical separations may exist, these boundaries are not enforceable at runtime. In contrast, within a microservices architecture, each component is isolated as an independent process, interacting with others through well-defined interfaces. This explicit separation reduces coupling and promotes stronger architectural modularity.

Data Management Another substantial difference concerns the data management model. Monolithic architectures typically rely on a centralized database, which simplifies transaction management but introduces strong dependencies among application modules. In microservices architectures, data decentralization at the service level enhances component autonomy and independent evolution. However, this comes at the cost of increased complexity in managing consistency and distributed operations.

Scalability and Reliability Scalability in monolithic systems is generally achieved by replicating the entire application, which can be inefficient under non-uniform workloads. Microservices, on the other hand, allow selective scaling of individual services according to specific load requirements. Similarly, while a monolith may act as a Single Point of Failure, decomposing the system into independent services improves fault isolation and enhances overall system resilience.

Technological Evolution and Release Cycle From an evolutionary perspective, monolithic systems impose significant constraints on the adoption of new technologies, as any modification must preserve compatibility across the entire application stack. Microservices architectures, enabled by polyglot governance and independent deployment units, support incremental technological evolution and faster release cycles, aligning with Agile methodologies and Continuous Delivery practices.

Architectural Complexity Finally, the complexity of the two paradigms manifests in different forms. In monolithic systems, complexity is concentrated within the codebase and tends to grow over time as the system evolves. In microservices architectures, part of

this complexity shifts toward infrastructure and inter-service communication, requiring advanced expertise in distributed systems, observability, and operational management.

This comparison highlights that the transition from a monolithic architecture to a microservices-based system does not represent a linear evolution, but rather a structural transformation that introduces new benefits alongside new challenges. These considerations provide the foundation for analyzing the issues related to architectural migration, which are discussed in the following section.

1.1.4. The Transition Challenge

The migration from a monolithic architecture to a microservices-based system is not just a technological change, but a highly complex engineering challenge. Empirical studies have shown that many organizations fail in their decomposition efforts, resulting in what is commonly referred to as a *Distributed Monolith*: a system that incurs the operational costs of a distributed architecture without gaining its decoupling benefits [7].

Boundary Identification and Data Entanglement The primary challenge lies in correctly identifying the architectural boundaries to introduce within the existing codebase. In monolithic systems that have evolved over several years, often degenerating into a *Big Ball of Mud*, data dependencies tend to become opaque, meaning they are neither explicitly documented nor easily reconstructed from the code structure alone. Splitting a centralized database into service-specific databases requires a meticulous analysis of relationships among entities. Mistakes at this stage may lead to excessive inter-service communication or complex data consistency issues, ultimately undermining the intended benefits of the migration.

The Gap Between Requirements and Implementation Another significant obstacle is the lack of up-to-date documentation. Business logic is often implicitly embedded within the code and may no longer reflect the original requirements. To employ semi-automated decomposition tools such as Cromlech, designers must first map each system functionality in terms of the involved operations and entities. This reverse engineering process is highly time-consuming and subject to subjective interpretation, effectively becoming a bottleneck for the adoption of architectural optimization tools.

The Need for Automation Due to these challenges, recent research has increasingly focused on automating the extraction of architectural knowledge. Addressing the transition problem requires tools capable of bridging the gap between the natural language

of requirements, such as user stories, and the formal models required by partitioning algorithms. Such tools aim to reduce human error and accelerate the architectural design phase.

Overall, the transition from a monolithic architecture to a microservices-based system emerges as a non-trivial process, in which incorrect decisions during the decomposition phase may compromise the resulting architecture. The difficulties associated with boundary identification, domain understanding, and reconstruction of architectural knowledge highlight the limitations of purely manual approaches. These considerations underscore the need for systematic and automated methodologies to support architectural design.

1.2. Monolith Decomposition Strategies

Decomposing a monolithic system into microservices requires a solid methodological foundation. The core problem consists in identifying a functional partitioning that maximizes the benefits of service independence while avoiding excessive communication overhead among distributed components. This process is traditionally guided by two fundamental principles: *cohesion* and *coupling*.

1.2.1. Cohesion and Coupling

The decomposition of a monolithic system into microservices demands a rigorous methodological approach. The main objective is to determine a partitioning of functionalities that maximizes the advantages of service autonomy while minimizing communication overhead between distributed components.

This process is traditionally driven by two fundamental principles of modular design: *cohesion* and *coupling*, which represent the primary criteria for assessing the quality of an architectural decomposition [15]

- **Cohesion:** measures the degree of correlation among the responsibilities assigned to a single microservice. A highly cohesive service groups closely related functionalities centered around a specific business domain, in line with the *Single Responsibility Principle*. High cohesion improves comprehensibility, maintainability, and independent evolution of services [16].
- **Coupling:** refers to the level of dependency between distinct microservices. High coupling occurs when the execution of a functionality requires frequent inter-service interactions or when local changes propagate across multiple services. This increases operational complexity and distributed communication costs[17].

The goal of decomposition strategies is therefore to minimize inter-service coupling while ensuring that operations frequently accessing the same data are grouped within the same microservice or functional cluster [4]. This reduces inter-service communication traffic and enhances overall architectural efficiency.

1.2.2. Static vs Dynamic Approaches

Architectural decomposition strategies can be classified according to the type of information used to analyze the system. In particular, the literature distinguishes between static analysis approaches and dynamic analysis approaches [18, 19], which differ in the data sources

considered and in the types of dependencies they can identify.

1. **Static Analysis:** These approaches rely on the examination of structural artifacts such as source code, domain models, and database schemas. They enable the identification of syntactic and structural dependencies, including method calls, relationships among persistent entities, and data access patterns. However, static techniques do not fully capture runtime behavior, as they do not account for how functionalities are actually exercised by users.
2. **Dynamic Analysis:** These approaches observe the system during execution, leveraging runtime data such as application logs, execution traces, and monitoring information. They allow the identification of real interaction patterns among components, highlighting temporal dependencies and operational flows that are difficult to detect through static analysis. Nevertheless, their effectiveness strongly depends on the quality and coverage of the collected data, which may be incomplete or not representative of all usage scenarios.

In practice, many architectural decomposition approaches combine both static and dynamic information in order to obtain a more comprehensive view of system dependencies and support more accurate partitioning decisions.

1.2.3. Decomposition Criteria

Beyond the general principles of cohesion and coupling, the literature identifies specific criteria that guide architectural decomposition processes in defining microservice boundaries. These criteria act as design constraints and optimization objectives, and are typically embedded within partitioning algorithms.

- **Data Semantics and Ownership:** Decomposition must respect data semantics and ownership responsibilities. The *database per service* principle states that each microservice should exclusively own its data schema to ensure autonomy and independent evolution. Consequently, entities frequently involved in the same transactions should be colocated within the same service to reduce the need for distributed transactions and simplify consistency management [4].
- **Communication Overhead:** Interactions among microservices introduce additional costs related to network communication, data serialization, and remote invocation handling. An effective decomposition strategy aims to minimize such costs by grouping operations with high interaction frequency within the same service.
- **Business Capabilities:** In line with *Domain-Driven Design* principles, microser-

vices should align with business capabilities and domain boundaries. This perspective promotes a modularization that is not only technical but also organizational, supporting team autonomy and long-term system scalability.

1.2.4. State of the Art in Decomposition Tools

The decomposition of monolithic systems into microservices has emerged as a prominent research topic in both academia and industry. Over the past decade, numerous tools and methodologies have been proposed to assist software architects in identifying suitable service boundaries. These approaches differ substantially with respect to the data sources they leverage, the analytical techniques they employ, the optimization strategies they adopt, and their degree of automation.

Although the literature does not provide a universally accepted taxonomy, existing solutions can be broadly classified according to their primary data sources and underlying analytical paradigms. Based on this perspective, decomposition tools can be organized into five macro-categories:

1. domain-driven and rule-based approaches
2. graph-based clustering techniques
3. static code analysis tools
4. dynamic and data-driven approaches
5. optimization-based frameworks

Domain-Driven and Rule-Based Approaches

Early service decomposition approaches are based on the principles of Domain-Driven Design (DDD) [20]. In these approaches, *Bounded Contexts* are identified through domain analysis activities and workshops involving domain experts and software architects.

These methods are not fully automated, as they rely mainly on human expertise and structured discussions. Several semi-structured tools have been inspired by DDD and apply predefined coupling and cohesion rules to support the identification of service boundaries.

Compared to purely technical approaches, DDD-based methods focus more on business capabilities and domain logic. However, their outcomes strongly depend on the experience of the involved experts, which may reduce reproducibility and introduce subjectivity in the decomposition process.

Graph-Based Clustering Approaches

Graph-based techniques represent a software system as a dependency graph, where nodes correspond to classes, entities, or operations, and edges capture structural or semantic relationships among them. This representation enables the application of graph partitioning and clustering algorithms to identify potential service boundaries.

Among the most well-known tools in this category is **Service Cutter** [21]. Service Cutter defines 16 coupling criteria, including semantic proximity, transactional consistency, shared ownership, and data access dependencies. These criteria are used to construct a weighted graph, and the decomposition problem is formulated as a clustering task over this graph.

Although Service Cutter introduces a structured and systematic methodology, its practical effectiveness largely depends on the manual definition of the coupling matrices. In legacy or poorly documented systems, gathering the required information can become a significant bottleneck. Moreover, the clustering strategy is heuristic and does not provide formal guarantees of optimality.

Other studies have investigated the use of community detection algorithms, such as modularity maximization and spectral clustering, to partition software dependency graphs [22]. These approaches are generally computationally efficient; however, they focus primarily on structural properties and do not explicitly account for higher-level architectural or domain constraints.

Static Code Analysis Tools

Static analysis-based approaches derive system dependencies directly from source code. By examining call graphs, package relationships, and database access patterns, these methods aim to identify groups of components that can be reorganized into candidate microservices.

Several works have explored clustering techniques on static dependency information. In [19], hierarchical clustering is applied to class-level dependencies extracted from the codebase. Similarly, the approach presented in [18] combines structural metrics with measures of semantic similarity to support the identification of service candidates.

Despite their structured methodology, static approaches present intrinsic limitations. Since they rely exclusively on compile-time information, they cannot capture runtime behavior. Call graphs often over-approximate actual interactions, including execution paths that may never be exercised in practice. Consequently, the inferred dependencies do not always reflect real usage patterns. Furthermore, business semantics expressed in natural language requirements remain outside the scope of purely code-based analysis.

Dynamic and Data-Driven Approaches

To address the limitations of purely static analysis, dynamic approaches leverage runtime information, such as execution traces, monitoring logs, and workload profiles. By observing how components interact during execution, these methods aim to derive service boundaries that better reflect actual system usage [23].

One of the most mature industrial solutions in this area is **Mono2Micro** [24], developed by IBM. The tool applies machine learning techniques to cluster classes that frequently co-occur within runtime traces, based on the assumption that components involved in the same business transaction should be assigned to the same microservice.

While dynamic approaches are effective in capturing real usage patterns, they also introduce several challenges. Their results depend heavily on the availability of representative execution traces and adequate test coverage. They may also be sensitive to workload bias, leading to decompositions that reflect specific usage scenarios rather than the overall system structure. Furthermore, rarely executed functionalities can remain underrepresented or entirely undetected.

In addition, the infrastructure required for trace collection and monitoring is not always available, particularly in early-stage development contexts or legacy systems.

Search-Based and Evolutionary Approaches

Another significant research direction formulates service decomposition as a search-based software engineering problem. In this context, evolutionary algorithms—such as genetic algorithms—are employed to optimize multiple objectives, typically aiming to maximize cohesion while minimizing coupling [25].

These approaches are well suited to exploring large and complex solution spaces, as they do not require strictly defined linear formulations. By iteratively refining candidate decompositions, they can identify configurations that balance competing structural metrics.

Nevertheless, their effectiveness depends on the design of appropriate fitness functions, which are often heuristic in nature. As a result, there is no formal guarantee of global optimality. Moreover, due to their stochastic components, different executions may produce different decompositions, potentially affecting reproducibility.

Optimization-Based Frameworks

More recently, service decomposition has been formulated as a formal optimization problem [26]. In this setting, the system is abstracted through a structured representation of entities and operations, and the partitioning task is expressed as a Mixed-Integer Linear Programming (MILP) model.

Among the most advanced solutions in this category is the Cromlech framework [1]. The framework explicitly incorporates multiple architectural factors into the optimization model, including:

- cohesion among operations,
- communication costs between services,
- data replication overhead,
- transactional constraints,
- co-location requirements.

In contrast to heuristic clustering techniques, MILP-based approaches provide guarantees of optimality with respect to the defined objective function. Moreover, they allow the trade-off between organizational cohesion and operational cost to be explicitly parameterized, enabling architects to systematically explore alternative design configurations.

At the same time, this formal rigor introduces practical challenges. Optimization-based frameworks require highly structured and detailed input models, including explicit specifications of entities, operations, data access patterns, invocation frequencies, and architectural constraints. The effort needed to construct such models can represent a significant barrier in real-world contexts.

Overall, the evolution of decomposition tools reflects a progressive shift toward higher levels of automation and formal rigor. While early approaches rely primarily on expert knowledge and domain modeling, more recent solutions adopt structured representations and algorithmic optimization techniques.

However, a fundamental limitation persists across all categories. None of the reviewed approaches explicitly address the transformation of informal or semi-structured requirements into the formal models required by optimization-based frameworks.

This disconnect between natural language requirements and formal architectural representations highlights a structural gap between requirement elicitation and automated

architectural synthesis, which we refer to as the *Input Gap*.

1.2.5. The Common Limitation: The Input Gap

Despite significant advances in architectural decomposition tools, a fundamental challenge still limits their large-scale adoption: the need for detailed formal input models [27]. Most existing tools require users to provide precise technical specifications of operations, involved entities, and their interactions, often represented through structured configurations such as `YAML` files or complex dependency graphs.

Constructing such models remains a predominantly manual, time-consuming, and error-prone process, especially in contexts where knowledge of the original system is fragmented, incomplete, or poorly documented.

In summary, the state of the art shows a progressive shift toward increasingly rigorous approaches based on mathematical optimization models, among which the Cromlech framework represents one of the most advanced solutions. However, the effectiveness of these methods critically depends on the ability to translate informal documentation and domain requirements into formal representations interpretable by optimization algorithms. This observation lays the groundwork for the analysis of the Cromlech framework and motivates the exploration of automation methodologies based on Large Language Models.

1.3. Cromlech Framework

Cromlech is a semi-automatic framework developed at Politecnico di Milano, designed to support the decomposition of monolithic systems into microservice-based architectures [1]. The framework adopts a hybrid approach that integrates organizational aspects, such as business-domain cohesion, with operational aspects, including communication costs and data management. Architectural decomposition is formulated as a mathematical optimization problem based on Mixed-Integer Linear Programming (MILP).

1.3.1. Conceptual Model

The conceptual model of Cromlech abstracts system complexity through two fundamental elements, defined independently of the underlying implementation technology: entities and operations.

- **Entities (E):** represent the informational resources managed by the system, such as database tables or logical data aggregates. The framework adopts the *Single Writer* principle, according to which each entity is associated with a single *leader replica* assigned to one microservice, responsible for all write operations. This ensures data consistency, while optional read-only replicas may be distributed across other services to reduce access latency.
- **Operations (O):** represent logical execution units of the system and constitute the functional layer of the model. Each operation is characterized by several attributes:
  - *Frequency:* an estimate of the invocation frequency of the operation, used to weight the impact of operational costs within the optimization model.
  - *Data access patterns:* the set of entities accessed by the operation, specifying for each whether access occurs in read or write mode.
  - *Transactional requirements:* a boolean property indicating whether the operation requires ACID transactional guarantees. In this case, a co-location constraint is imposed between the operation and the *leader replica* of the entities on which it performs write operations.
  - *Co-location constraints:* a set of constraints enforcing the joint placement of specific operations within the same microservice. These constraints allow domain knowledge or architectural requirements not explicitly captured by the formal model to be incorporated.

Interactions between operations and entities are formalized through an access matrix, where for each (operation, entity) pair the access mode is specified, either read or write. This representation is central to the evaluation of cohesion and to the definition of partitioning constraints: in particular, colocating operations with frequently accessed entities contributes to reducing latency and inter-service communication costs.

1.3.2. Optimization Engine and MILP Formulation

Cromlech translates the conceptual model into a Mixed-Integer Linear Programming (MILP) problem, solved using the Gurobi solver [28]. The optimization formulation aims to identify a partitioning of operations and entities into microservices that balances two primary objectives: maximizing architectural cohesion and minimizing runtime operational costs.

This trade-off is regulated by the parameter α, referred to as the *organization–operations ratio*, which controls the relative weight of the two components in the objective function:

$$Obj_\alpha = \alpha \cdot Coh - (1 - \alpha) \cdot \frac{OpCost}{OpCost_{max}} \tag{1.1}$$

The terms of the objective function are defined as follows.

1. **Cohesion (Coh):** a normalized metric in the interval $[0, 1]$ that measures the affinity between operations assigned to the same microservice. Cromlech computes this value based on the overlap of entities accessed by pairs of operations, considering the intersection of their respective data sets. A high cohesion value indicates that the service aggregates closely related functionalities, promoting alignment with business domains and improving system maintainability.

2. **Operational cost ($OpCost$):** represents the total estimated runtime cost resulting from the distribution of operations and data among microservices. It consists of two main components:

   - *Communication cost ($CommCost$):* models the impact of network interactions between services. It includes both the cost of read operations, incurred when accessing a remote replica, and the cost of write operations required to interact with a *leader replica* located in a different service.

   - *Data management cost ($MngCost$):* quantifies the overhead associated with propagating updates from the *leader replica* to distributed read replicas in order to maintain data consistency across services.

The total operational cost is therefore expressed as:

$$OpCost = CommCost + MngCost \tag{1.2}$$

Since $OpCost$ is expressed in absolute terms, it is normalized with respect to $OpCost_{max}$, which represents the theoretical cost of a fully decentralized architecture. This normalization enables a direct balance between cohesion and operational cost within the objective function.

1.3.3. Input Specification and the YAML Gap

The Cromlech optimization engine requires a formal `YAML` specification describing entities, operations, and their data access patterns (read/write), including invocation frequencies, transactional requirements, and co-location constraints. As discussed in the previous section, manually constructing such a specification represents the primary barrier to the adoption of the framework, constituting the so-called *Input Gap*. In real-world scenarios, available documentation is often incomplete or outdated, and extracting the necessary information requires significant manual effort and is prone to errors.

The contribution of this thesis addresses this limitation by proposing the integration of Large Language Models (LLMs) to automate the generation and refinement of the `YAML` specification. The adopted approach consists of two main phases.

1. **Initial extraction:** In the first phase, the LLM analyzes the system's *user stories* to generate a structured representation in `YAML` format. The model automatically extracts operations, involved entities, data access patterns, usage frequencies, and consistency requirements. This step enables the rapid production of a formal configuration ready to be processed by the optimization engine.
2. **Constraint refinement (co-location):** In a second phase, the generated specification is further analyzed by the LLM to identify constraints corresponding to *forced operations*, that is, groups of functionalities that must be placed within the same microservice due to logical, transactional, or functional dependencies not explicitly captured by the formal model.

Once the process is completed, the resulting `YAML` file is used as input for Cromlech. This approach automates the most labor-intensive phase of system modeling and enables a systematic evaluation of the impact of LLM-suggested co-location constraints on architectural decomposition results.

1.4. Large Language Models in Software Engineering

The rise of Large Language Models has reshaped the boundaries of Artificial Intelligence, transforming language models from simple text prediction engines into sophisticated tools that can support engineering activities. In software engineering, their relevance does not only depend on the ability to generate fluent text, but also on the ability to interpret complex meanings and translate informal requirements into rigorous data structures [29].

This section introduces the theoretical foundations that make such an integration possible. It begins with the Transformer architecture and the attention mechanism, then discusses interaction strategies through prompt engineering and the parameters that influence model behaviour during generation. Finally, it discusses the emerging distinction between efficiency-oriented models and reasoning-focused architectures, establishing the conceptual framework for evaluating their suitability in automated monolith decomposition.

1.4.1. Generative Artificial Intelligence and Transformers

The rapid development of generative Artificial Intelligence, which culminated in the widespread adoption of Large Language Models, is strongly connected to the introduction of the **Transformer** architecture proposed by Vaswani and colleagues in 2017 [30]. Before this innovation, the state of the art in Natural Language Processing was dominated by sequential models, especially Recurrent Neural Networks and Long Short-Term Memory architectures.

Although effective in many application scenarios, these models exhibited relevant structural limitations. First, their sequential processing made training difficult to parallelise. In addition, modelling long-range dependencies was inefficient due to the well-known *vanishing gradient* problem, which is only partially mitigated by Long Short-Term Memory networks [31].

The Transformer architecture addresses these issues by introducing the **self-attention** mechanism, which allows the model to represent relationships among all tokens in a sequence independently of their positional distance. In this framework, each token is projected into three vector representations, namely *Query*, *Key*, and *Value*. The attention between two elements is computed through the dot product between Query and Key, enabling the model to integrate global contextual information into the representation of each token. This mechanism significantly improves the ability to capture complex semantic dependencies [30].

1.4.2. Large Language Models for Software Engineering Tasks

The use of Large Language Models in software engineering has introduced a significant change, transforming them from text completion tools into assistants that can support design activities. This evolution was enabled by training on large datasets that include not only natural language, but also billions of lines of source code collected from public repositories such as GitHub [32, 33].

The effectiveness of Large Language Models in software engineering tasks can be traced back to three main capabilities:

1. **Semantic understanding of code:** unlike traditional static parsers, Large Language Models can capture the logical intent behind code fragments. They can map natural language descriptions to logical structures and architectural patterns, supporting the translation between requirements and implementation.

2. **Multi-language and cross-domain versatility:** these models can operate across different programming languages and across multiple data serialization formats, including `JSON`, `XML`, and `YAML`. This property is particularly relevant for this thesis, where the model must interpret user stories and generate structured specifications in `YAML` format.

3. **Support for requirements engineering:** Large Language Models have shown usefulness in identifying ambiguities, completing missing information, and extracting formal models from informal requirement descriptions [34].

In the context of decomposing monolithic systems, the use of Large Language Models helps address the cognitive complexity associated with dependency analysis. While a human analyst would require significant time to map relationships among many operations and the involved database entities, a Large Language Model can process such information at scale, identifying data access patterns and invocation frequencies in a way that reduces manual effort [35].

However, these models also introduce relevant risks, including *hallucinations*, meaning the generation of plausible but incorrect information, and a strong dependency on the quality and structure of the input. These issues are discussed further in the sections dedicated to prompt engineering.

1.4.3. Prompting and Large Language Model Interaction Paradigms

Prompt engineering has emerged as a crucial discipline for using Large Language Models effectively. It can be defined as the practice of formulating textual inputs, called prompts, that guide the model towards specific and accurate outputs [36]. When partial retraining is not performed, the prompt remains the primary mechanism for conditioning model behaviour, leveraging in-context learning.

Recent literature distinguishes several fundamental interaction strategies:

- *Zero-shot prompting:* the model must perform a task based only on the instruction, without any reference examples. This approach tests generalisation and instruction-following capabilities acquired during pretraining [37].
- *Few-shot prompting:* the prompt includes a small number of examples, typically as input-output pairs. This technique helps the model infer the expected format and domain semantics, often reducing output variability [38].
- *Chain-of-thought prompting:* the prompt explicitly requests intermediate reasoning steps before the final answer. This strategy has been shown to improve performance on arithmetic, symbolic, and logical reasoning tasks, because it encourages the model to allocate additional computation to the reasoning process [39].

A particularly relevant concept for software engineering is *system prompting*. Modern systems allow a system-level message that defines role, tone, and persistent behavioral constraints for the model, for example by stating that it should behave as an expert software architect. This abstraction helps separate operational instructions from persistent context, improving consistency in multi-step generation pipelines.

Nevertheless, prompting faces an intrinsic sensitivity to small variations. Minor changes in instruction wording can lead to significantly different outcomes. For this reason, current research focuses on designing robust prompts and on formally validating generated outputs, especially when the objective is to produce code or structured configuration files.

1.4.4. Large Language Models for Structured Output Generation

A major challenge when integrating Large Language Models into engineering pipelines is the transition from unstructured natural language to serialized data formats such as `JSON`, `XML`, or, in this thesis, `YAML` [40]. While ambiguity is intrinsic to human language, optimization frameworks such as Cromlech require inputs that strictly comply with syntactic rules and predefined semantic constraints.

The ability of a model to generate correct structured output is referred to in the literature as *schema adherence* or *instruction following for data extraction* [41]. Even though state-of-the-art models are trained on large code corpora, generating complex configuration files presents several difficulties:

- *Syntax and formatting:* even a single indentation error or an incorrect special character in a `YAML` file can make it unreadable to standard parsers, interrupting the automated decomposition workflow.
- *Schema hallucination:* models may introduce optional keys or attributes that are not required, based on probabilistic patterns learned during pretraining, violating the specification expected by the target tool.
- *Referential integrity constraints:* in architectural modelling tasks, the model must ensure that entity names referenced in operations exactly match those defined in the entity catalogue, preserving terminological consistency across the document.

To mitigate these risks, research has explored grammar-constrained decoding techniques and prompting approaches that specify an explicit output contract. In this context, the ability of Large Language Models to translate natural language requirements into structured and formally valid representations, while respecting syntactic constraints, semantic constraints, and internal consistency, is a fundamental prerequisite for integrating them into automated software analysis and optimisation pipelines.

1.4.5. Decoding Strategies: Temperature and Sampling

Text generation in a Large Language Model is inherently stochastic. At each step, the model produces a probability distribution over the full vocabulary. The next token is selected according to a decoding strategy, which determines the balance between determinism, meaning exploitation of the most likely continuation, and exploration, meaning the introduction of variability.

The central parameter in this process is **temperature** (T). From an analytical perspective, this parameter originates from statistical mechanics and was later adopted in Deep Learning to calibrate model confidence [42]. Temperature modifies the softmax applied to the logits, which are the raw numerical scores produced by the final layer of the network, as follows:

$$P(x_i) = \frac{\exp(z_i/T)}{\sum_j \exp(z_j/T)}$$

The impact of T on the probability distribution defines the behavior of the model:

- **Low temperature ($T \to 0$):** when temperature approaches zero, the probability distribution becomes sharper, increasing the difference between the most probable token and the others. At the limit $T = 0$, the model operates with greedy decoding, always selecting the token with the highest probability. This approach maximizes determinism and is common in data extraction and code generation tasks, where syntactic correctness and reproducibility are prioritized over linguistic variety.
- **High temperature ($T \geq 1$):** as temperature increases, the probability distribution becomes smoother, assigning more probability mass to less frequent tokens. While this introduces variability, in engineering contexts it substantially increases the risk of syntactic errors and hallucinations, because the model is more likely to deviate from the most reliable structured patterns learned during training [43].

From an operational perspective, higher temperature increases expressive freedom. With $T = 0$, the model is constrained to produce the safest and most repeatable output, while with $T = 1$, the model explores less frequent tokens and may generate syntactic or lexical variants that would not appear under greedy decoding. While this can produce richer outputs, it can also lead to structural hallucinations, where creativity is favoured over the formal rigour required by programming or configuration languages [44].

1.4.6. The Reasoning Paradigm: System One Thinking and System Two Thinking

Recent research on Large Language Models (LLMs) distinguishes between two different inference paradigms: immediate, pattern-based generation and structured, multi-step reasoning. This distinction can be interpreted through the lens of dual-process theories of cognition [45] and has recently been adapted to characterize reasoning behaviors in LLMs [46, 47].

- **Fast inference:** This paradigm is characterized by direct output generation based on learned statistical associations between inputs and outputs. It relies primarily on probabilistic next-token prediction and pattern recognition capabilities acquired during pretraining. Fast inference is computationally efficient and often effective for tasks involving text transformation, structured format generation, or syntactic completion. However, it may exhibit limitations when addressing problems that require long-horizon planning, global constraint satisfaction, or consistency verification across multiple interdependent elements.
- **Deliberate inference:** This paradigm allocates additional inference-time com-

putation to structured reasoning processes. Instead of producing an answer in a single pass, the model generates intermediate reasoning steps, decomposes the task into subproblems, and performs internal consistency checks before outputting a final result. Such behavior is commonly implemented through multi-step reasoning techniques, including chain-of-thought prompting [39], and has been shown to improve performance on tasks requiring logical coherence and constraint-aware decision-making.

In software engineering, this distinction is crucial. While the syntactic generation of a `YAML` schema can often be handled effectively by a System One model due to its linguistic competence, resolving conflicts among functional requirements and maintaining architectural consistency can benefit from the stronger analytical behavior typical of System Two models. Including both types of models in this study makes it possible to evaluate the trade-off between execution speed and logical rigor in monolith decomposition tasks.

2 | Problem and Solution Overview

This chapter introduces the context of the problem addressed in this thesis and provides a general description of the elements that characterize the reference scenario. The objective is to define the conceptual framework within which the work is situated.

The first part of the chapter outlines the problem in question, highlighting the challenges associated with the design and evaluation of complex software architectures. Next, an overview of the approach considered is provided, which is useful for understanding the structure of the work and the choices that guide the subsequent phases.

2.1. Problem Overview

Breaking down a monolithic system into microservices is one of the most complex challenges in modern software engineering, as it requires architectural decisions that directly affect the maintainability, scalability, and functional cohesion of the system [48]. Although optimization frameworks such as *Cromlech* have automated the search for the optimal network topology, the effectiveness of such tools is closely dependent on the quality and accuracy of the input model.

In recent years, the use of LLMs has shown promising results in supporting software engineering activities, particularly in understanding and generating requirements from natural language descriptions. Among these activities, the transformation of user stories into structured formal specifications is a problem of particular interest, both in academia and industry.

User stories are commonly adopted in agile development processes to describe the functional requirements of a system in a simple and understandable way for stakeholders [49]. However, they are inherently characterized by a high level of ambiguity, incompleteness, and linguistic variability. In contrast, formal specifications, such as those expressed using `YAML` files, require a precise, unambiguous, and structurally consistent representation, as they are intended to be interpreted automatically by software tools.

The problem addressed in this thesis therefore consists in the automatic generation of

structured YAML specifications from user stories, using LLM models, while ensuring:

- semantic correctness with respect to the original requirements,
- completeness of the requested information,
- adherence to a predefined structural schema.

Although LLMs are capable of producing syntactically valid and semantically plausible output, their application to this context presents several challenges. In particular, the generation of complex and constrained structures, such as `YAML` specifications for software operations, requires rigorous control over the output that goes beyond the simple ability to generate coherent text. Even minor errors in structure or content can compromise the usability of the generated specifications.

Furthermore, the stochastic nature of LLMs introduces significant variability in the results produced, making it difficult to evaluate the quality of the generations and to compare different models or configuration [50]. Consequently, there is a need to define a systematic process that allows for the objective analysis, comparison, and evaluation of automatically generated specifications, especially in the presence of complex architectural constraints.

The problem addressed in this work is therefore the search for a systematic methodology that, by exploiting the cognitive capabilities of *Large Language Models*, is able to automate the extraction of knowledge from requirements, while ensuring a metric rigor comparable to or superior to that of humans.

2.1.1. Input Requirements for Cromlech

Cromlech is a multi-objective optimization framework designed to identify optimal partitions of a monolithic system. Its operation is based on solving an optimization problem that aims to balance the maximization of internal cohesion within microservices and the minimization of coupling and infrastructure costs. To operate, the framework requires a structured specification in `YAML` format that must reflect three fundamental pillars of information:

1. *Data Model (Entities):* a formal definition of the system's domain entities, used to map how data is distributed among potential microservices.
2. *Functional Model (Operations):* an exhaustive list of application operations, specifying their data access profile, distinguishing between read and write attribute sets. This granularity is essential for calculating functional dependencies.
3. *Dynamic Behavior (Workload):* an indication of the invocation frequency of each

operation, allowing the solver to weigh the importance of keeping certain operations co-located to avoid network communication overhead on critical paths.

The quality of the decomposition produced by Cromlech depends directly on the accuracy and completeness of the specification provided as input. Inaccurate or incomplete representations of operations, data accesses, or quantitative information can lead to suboptimal or inconsistent architectural solutions with respect to system requirements.

2.1.2. Forced Operations as an Architectural Constraint

In addition to the structural and quantitative information described above, Cromlech allows the introduction of explicit architectural constraints that restrict the space of admissible solutions to the optimization problem. These constraints reflect the fact that, in the process of decomposing a monolithic system into microservices, not all architectural decisions can be entrusted exclusively to automatic optimization criteria.

In many real-world contexts, there are requirements that specific application operations must be located within the same microservice, regardless of the objectives of minimizing coupling or maximizing functional cohesion. These constraints, which are often not explicitly expressed in structural models, derive from architectural requirements such as data consistency requirements, the need to perform atomic transactions, performance constraints, or dependencies imposed by the existing system architecture.

The problem therefore lies in the vulnerability of the input model: dependence on a manual extraction process, which is inherently prone to omissions and terminological discrepancies, transforms the `YAML` file into a bottleneck that limits the effectiveness of advanced tools such as *Cromlech*. There is therefore a critical need to investigate approaches capable of extracting such complex relationships from unstructured requirements, while preserving the syntactic and semantic rigor required to avoid compromising the entire architectural analysis pipeline.

In this context, the work lies at the intersection between software engineering, microservice architectures, and natural language processing techniques, addressing the problem of automating architectural modeling based on unstructured requirements.

2.2. Solution Overview

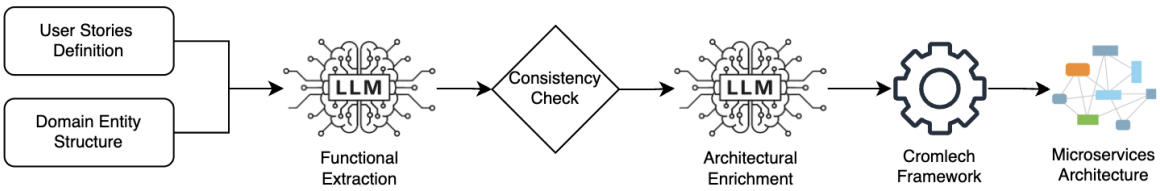

Figure 2.1: Proposed solution workflow overview

The goal of the proposed solution is to define a semi-automatic workflow that bridges the gap between business requirements, expressed in natural language, and the technical design of a microservices architecture. The approach leverages the reasoning capabilities of LLMs to formalize architectural knowledge, reducing the manual burden and risk of errors in drafting specifications for automatic partitioning tools such as Cromlech.

As illustrated in Figure 2.1, the process is organized into a sequential pipeline in which human input and automatic generation cooperate to produce a formal specification in YAML format, which is then used to define the microservices architecture.

2.2.1. Requirements Definition: User Stories and Entity Structure

The initial phase of the workflow is dedicated to formalizing knowledge of the application domain, i.e., defining the information perimeter within which the entire process operates. For automatic generation to be effective and controllable, it is necessary to provide a structured and complete description of the functional requirements and data model of the system.

This phase consists of two main components:

User Stories Definition Each feature or use case of the software must be translated into a user story following a defined standard format (e.g., "As a [role], I want [goal], so that [benefit]"). Standardization of language is a key element, as it allows the actor, action, and business goal to be made explicit, providing linguistic models with clear and consistent semantic signals for identifying application operations.

Entity Structure Mapping At the same time, the structure of the domain entities is defined, including relevant relationships and attributes. This data model acts as a reality

constraint for generative models, preventing the introduction of entities or attributes not present in the original schema and contributing to the semantic consistency of the specification produced.

Requirements can be defined in two ways: an entirely manual approach, based on the functional analyst's experience, or a hybrid approach, in which the human operator uses an LLM as a support to accelerate the drafting of User Stories from existing technical documentation. In both cases, the output of this phase must ensure complete functional coverage, as any omission at this early stage would directly result in a missing operation in the final specification.

2.2.2. Functional Structure Extraction (LLM-1)

The first generation phase is dedicated to extracting the functional structure of the system and is entrusted to an initial LLM. The purpose of this stage is to translate the User Stories and entity structure into a technical specification compliant with the YAML schema required by the Cromlech framework.

In this phase, the model operates as a technical analyst, performing a semantic mapping process aimed at formalizing the functional requirements in terms of elementary operations and data access. For each user story input, the LLM must first identify an operation that represents the action described by the requirement, assigning it a unique name expressed in *camelCase* format. Subsequently, the model is called upon to identify the entities and attributes involved, strictly distinguishing between read and write accesses in order to correctly express the data access pattern.

In addition to the structure of the operations, the model must also deduce certain fundamental transactional properties, such as the parameters of consistency and frequency, from the semantic clues present in the natural language of the user stories.

The use of a structured and constrained prompt is essential to prevent hallucination phenomena, limiting access to the database exclusively to the entities and attributes defined in the initial schema. The result of this process is a preliminary functional specification which, although syntactically complete and semantically consistent, does not yet include the co-location constraints between operations. This design choice allows the correctness of the functional modeling and data access patterns to be isolated and validated before introducing the complexity of architectural reasoning.

2.2.3. Consistency Check

Once the preliminary functional specification has been generated, the workflow includes an intermediate validation phase called *Consistency Check*. This step is essential to ensure that the output produced by the LLM is not only syntactically correct, but also semantically consistent with the domain constraints defined in the input phase.

The control activity is divided into three main aspects:

1. *Schema Alignment:* it is verified that each entity and attribute mentioned in the generated `YAML` actually belongs to the entity structure provided. This check allows any model hallucinations to be identified and neutralized, ensuring that the specification can be mapped to the system's actual database.

2. *Syntactic Integrity:* the file's compliance with the `YAML` standard and the conventions required by Cromlech (*camelCase* for operations and *PascalCase* for entities) is checked, ensuring that the output can be correctly processed by subsequent parsers.

3. *Functional Traceability:* each generated operation is verified to be correctly associated with the source user story, ensuring that functional requirements are covered and identifying any omissions or duplications.

The Consistency Check acts as a quality filter; only specifications that pass this phase are propagated to the next stage of architectural enrichment. This approach prevents errors from propagating between different stages of the workflow and helps to increase the overall reliability of the final architecture produced.

2.2.4. Architectural Enrichment: Forced Operations (LLM-2)

Once the functional structure has been consolidated and validated, the workflow proceeds to the architectural enrichment phase. At this stage, a second model is tasked with identifying *Forced Operations*, i.e., co-location constraints that require the solver to assign certain operations to the same microservice.

Unlike the previous phase, which was mainly focused on extracting the data structure and access patterns, the activity of *LLM-2* requires a higher level of Architectural Reasoning. At this stage, the model is called upon to analyze the interdependencies between the operations already defined, with the aim of identifying logical affinities motivated by engineering criteria.

In particular, the model must recognize cases in which multiple operations critically access the same data and require transactional atomicity guarantees, making their co-location

preferable to avoid the complexity of distributed transactions. Similarly, performance implications are considered: operations frequently performed sequentially or jointly can benefit from residing in the same microservice, reducing the overhead introduced by inter-service communication. Finally, the model is incentivized to preserve the cohesion of the application domain by grouping operations belonging to the same logical subdomain in order to improve the maintainability and evolvability of the architecture in the long term.

The generation process adopts the *Context-Augmented Update* paradigm: the LLM receives the validated YAML as input and, operating under the constraint of *Strict Integrity*, populates only the `forced_operations` field, without modifying the names of operations, entities, or attributes already defined. This approach ensures that architectural enrichment occurs as a coherent logical progression, transforming a purely functional specification into an input ready for formal optimization.

2.2.5. Solver Execution and Microservices Architecture

The final stage of the workflow represents the convergence point between LLM-assisted generation and formal mathematical modeling. The enriched `YAML` file, containing both the functional structure and the co-location constraints, is fed into the Cromlech framework for the optimization process to be performed.

The solver operates as a constrained optimization engine, processing the received specifications to determine a coherent and balanced architectural partitioning. At this stage, the decisions made by the language models in the previous stages directly influence the final output. In particular, forced operations are treated as hard constraints, ensuring that operations identified as logically related are effectively placed within the same service boundary.

At the same time, the solver considers data access patterns, frequency values, and consistency levels to optimize the overall partitioning of the system. The goal is to minimize coupling between microservices and maximize internal cohesion, producing a decomposition that respects both explicit architectural constraints and operational efficiency criteria.

The final result of this process is the definition of the system's **microservices architecture**. The output consists of a formally valid architectural decomposition, obtained by applying a constrained optimization process to a specification derived from the initial functional requirements.

In this context, linguistic models act as a mechanism for translating and structuring domain knowledge, while the solver is responsible for processing architectural decisions according

to formal criteria. This separation of responsibilities allows the mathematical rigor of the partitioning process to be maintained, delegating the interpretation of requirements expressed in natural language to LLMs.

In the next chapter, this workflow is instantiated in a controlled experimental context in order to evaluate the quality of the generated specifications, the stability of architectural decisions, and the impact of constraints derived from LLMs on the final partitioning.

3 | Experimental Methodology

To address the critical issues identified in the previous section, the proposed work adopts a systematic methodology that combines the generation capabilities of Large Language Models with rigorous quantitative evaluation, aimed at producing structured, complete, and consistent specifications.

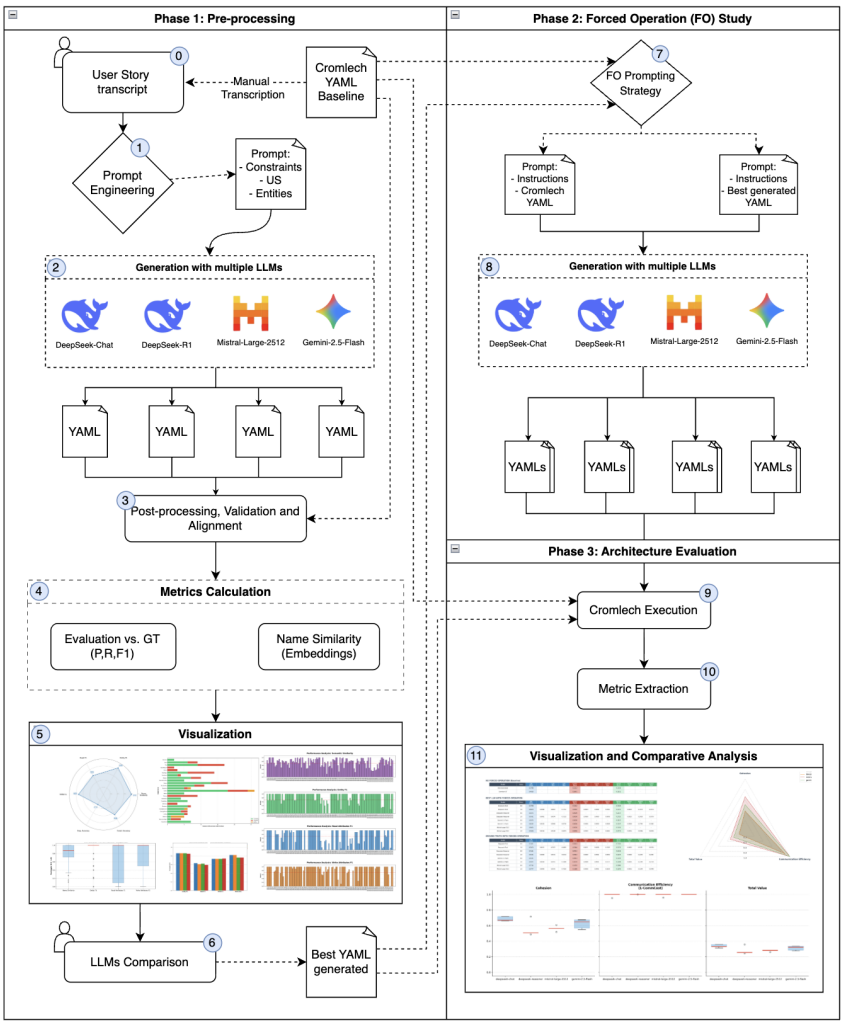

Figure 3.1: Methodological workflow: from User Stories acquisition to `YAML` specification synthesis.

The workflow architecture, illustrated schematically in Figure 3.1, is conceived as an incremental process divided into three main macro-phases. The first phase is oriented towards the systematic extraction of the functional structure and the selection of the best-performing linguistic model through a quantitative comparison with the *Ground Truth* (GT). The second phase, on the other hand, exploits the reasoning capabilities of the reference models to enrich the specification with complex architectural constraints, such as forced operations, analyzing the impact of stochasticity on output consistency. The workflow concludes with the synthesis of results through data visualization tools.

The result is a replicable and measurable approach that reduces dependence on manual intervention and allows the quality of the generations to be evaluated in terms of semantic correctness, completeness, and structural adherence. The phases in the workflow will be described in detail in the subsections, highlighting both the methodological choices and the evaluation criteria used.

3.1. Selection of LLMs for Structured Specification Generation

The experimental study requires the comparison of multiple Large Language Models (LLMs) in a controlled structured-generation task. The objective is not to identify an absolute "best" model, but to analyze how different architectural design choices and inference strategies affect the reliability of formal specification generation under strict syntactic and semantic constraints.

Model selection was driven by the specific requirements of the experimental task. The generation of Cromlech-compatible `YAML` specifications imposes strict syntactic constraints, semantic consistency across multiple entities, and complete adherence to a predefined schema. Furthermore, the prompt integrates extended contextual information, including multiple User Stories and a full entity structure, requiring robust long-context handling.

For these reasons, the benchmark includes models characterized by different inference profiles, ranging from general-purpose instruction-following architectures to reasoning-oriented variants, in order to evaluate how these design choices affect structured specification generation.

Based on these criteria, four models were selected: DeepSeek-Chat, DeepSeek-Reasoner, Mistral-Large-2512, and Gemini-2.5-Flash. The following subsections describe the motivations for their inclusion in the benchmark.

DeepSeek Family: V3 and R1

The DeepSeek family represents one of the most relevant contributions in the open-weights ecosystem, introducing architectural solutions aimed at balancing reasoning capabilities and computational cost. This thesis considers two main variants: **DeepSeek V3**, optimised for throughput, and **DeepSeek R1**, designed to support more structured logical reasoning tasks.

The inclusion of DeepSeek models is motivated by the following technological aspects:

1. *Multi-head Latent Attention:* DeepSeek V3 introduces an attention mechanism that significantly reduces the impact of the key-value cache, enabling more efficient handling of long contexts compared to traditional Transformer architectures. This property is relevant when processing large and complex inputs [51].

2. *Test-time reasoning and chain-of-thought behaviour:* DeepSeek R1 adopts a training paradigm based on Reinforcement Learning that encourages multi-step reasoning. These mechanisms allow the model to address tasks requiring more explicit logical structuring before producing the final output [52].

3. *Mixture-of-Experts and multi-token prediction:* the adoption of a Mixture-of-Experts architecture combined with multi-token prediction techniques improves inference efficiency while maintaining high semantic consistency in structured outputs.

Mistral Family: Mistral Large 2512

Models developed by Mistral AI are characterised by a focus on efficient sparse architectures and strong instruction-following capabilities. In particular, **Mistral Large 2512** is considered a reference model for analysing structured output generation under strict syntactic and semantic constraints.

The main motivations for including Mistral Large 2512 are:

1. *Advanced instruction following:* the model is trained to adhere consistently to prompt instructions, reducing deviations from the expected output format. This property is particularly relevant when generating configuration files where high precision is required [53].

2. *Mixture-of-Experts architecture:* selective activation of subsets of parameters during inference helps balance high performance and efficient resource usage, making the model suitable for complex tasks without excessive latency.

3. *Reasoning over hierarchical data:* Mistral Large demonstrates strong capability in

handling nested and hierarchical data structures, which contributes to reducing structural errors in `YAML` generation.

Gemini Family: Gemini 2.5 Flash

The Gemini family, developed by Google DeepMind, integrates Transformer architectures optimised for scalability and advanced Reinforcement Learning techniques. This thesis includes the **Gemini 2.5 Flash** variant, designed to maximise throughput and reduce inference latency.

The selection of this model is motivated by the following technical properties:

1. *Support for long contexts:* the model is designed to handle large context windows while maintaining effective attention over information distributed across the input. This property is useful when processing large collections of user stories [54].
2. *Efficiency-oriented design:* the Flash variant prioritizes high performance and low latency through optimization and distillation techniques, making it relevant for analyzing efficiency-optimized models in systematic data transformation tasks.
3. *Structural consistency of output:* the model shows a strong ability to preserve terminological and structural consistency in technical output, which is essential to ensure interoperability with automated optimization tools.

Overall, the selected models span heterogeneous architectural strategies and training paradigms, enabling a comparative analysis of how inference design, context management, and reasoning-oriented mechanisms influence the generation of structured and solver-compatible `YAML` specifications.

The selection is intentionally diversified in order to reduce bias toward a single model family and to strengthen the generalizability of the experimental findings.

3.2. Phase 1: Pre-processing

Phase 1 represents the starting point of the entire experimental study and aims to transform functional requirements, expressed in natural language, into a structured technical specification that serves as a baseline for subsequent analyses. In this phase, language models are used exclusively to extract the functional structure of the system, without introducing architectural constraints.

From a methodological point of view, this separation is essential to isolate the semantic extraction capabilities of LLMs from more complex architectural reasoning. Ensuring

a correct and consistent representation of operations, entities involved, and data access patterns is a necessary prerequisite to avoid the propagation of errors in subsequent stages of the workflow.

As illustrated in Figure 3.1, the phase includes pre-processing, generation, validation, and comparison with Ground Truth activities in order to select a reliable functional specification to be used as a reference in subsequent experiments.

3.2.1. User Story Transcription (Step 0)

The starting point of the workflow is the creation of the input dataset. Starting from a Ground Truth `YAML` file containing the list of operations and their associated fields, a **manual transcription** was performed in User Stories. This process of "reverse engineering" the requirements allowed the technical specifications to be converted into functional descriptions in natural language following the standard format:

"As a [role], I want [goal], so that [benefit]"

This template specifies the actor, the objective, and the business value, representing a de facto standard for drafting requirements in Agile development processes [55].

At this stage, particular attention was paid to preserving the informational integrity of the original GT: the attributes and data entities involved in the operations were translated into discursive actions and usage contexts, ensuring that the complexity of the domain was not altered during conversion. This approach made it possible to faithfully simulate the source material that a software architect usually receives from stakeholders, laying the groundwork for testing the models' ability to extract a formal structure from unstructured input.

It is important to note that this phase is a necessary preliminary step in building the experimental dataset and defining a controlled baseline for comparative evaluation of the models.

3.2.2. Prompt Engineering (Step 1)

The second step of Phase I concerns the automated generation of specifications using the models selected and described in Section 3.1. This activity is governed by a rigorous *Prompt Engineering* strategy, aimed at transforming textual input into structured output that complies with the formal constraints of the Cromlech solver.

The prompt design, shown in full in Appendix A.1, follows a structured instruction-based

approach, in which the information provided to the model is organized into distinct logical blocks, each with a well-defined semantic function. This organization, illustrated in Figure 3.2, is intended to improve the model's understanding of the task and reduce interpretative ambiguity.

In particular, the prompt is divided into the following main sections:

1. **Role Definition:** the operational identity of the model is established as *"Software Architect Assistant"*, specializing in the generation of Cromlech-compatible specifications.
2. **Context and Tools:** the Cromlech framework and its purpose (automatic decomposition) are presented, providing the model with an understanding of the final purpose of the output.
3. **Hard Constraints:** collects the set of non-negotiable logical constraints that must be satisfied in order for the generated specification to be semantically correct and processable by the solver.
4. **Input Specification:** provides the two necessary datasets: the list of User Stories and the `YAML` schema of the domain entities of the monolithic system.
5. **Modeling Rules:** defines the operational rules that guide the modeling of operations, imposing a standardized and minimal output structure.
6. **Output Format Enforcement:** a mandatory instruction that requires the output to be in pure `YAML` format, without discursive explanations or additional text blocks, to facilitate automatic parsing.

Hard Constraints These constraints form the logical basis of the prompt and cannot be violated. Compliance with them is essential to ensure that the generated specification is semantically correct and usable by the solver.

- *Consistency Rule:* the model must guarantee data integrity by enforcing strong consistency (consistency: 'H') whenever a write operation is present (`write_attributes` not empty). This constraint instructs the solver on the need to maintain transactional atomicity, directly influencing microservice partitioning decisions.
- *Absolute Disjoint and Atomic Mutation:* these two constraints work together to enforce isolation of data access. It is established that the sets `read_attributes` and `write_attributes` must be mutually exclusive for each entity.
- *Zero-Hallucination Policy and Exact Match:* total adherence to the provided entity

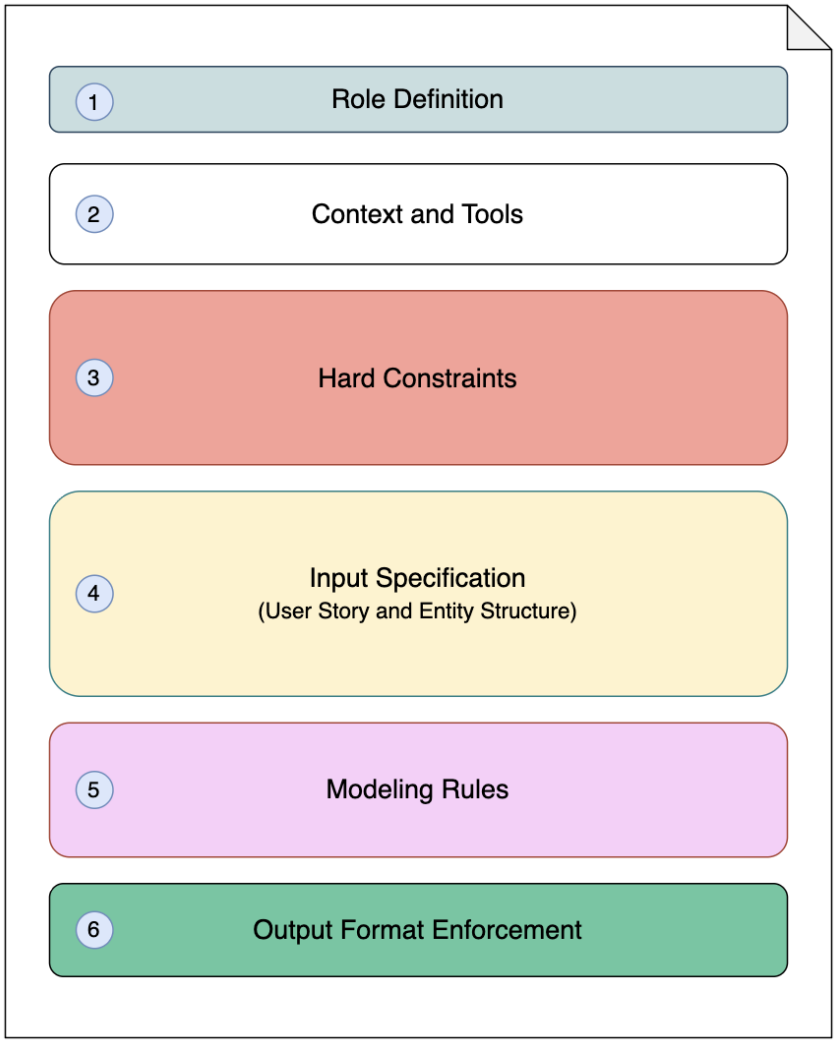

Figure 3.2: Structural blocks of the initial YAML generation prompt.

schema is enforced. The model is prohibited from "inventing" attributes and must strictly respect case sensitivity. This ensures that every reference generated is mappable 1:1 with the system's actual data dictionary.

- *Update/Delete Protocol:* specifies the structural rules for state change operations. In update operations, the `read_attributes` field must contain only the target entity identifier, while `write_attributes` must include only the modified attributes. In delete operations, the `write_attributes` field must be empty.
- *Existence and Uniqueness Constraints:* each operation must be uniquely mapped to a specific User Story and must have an identifying name. In addition, each operation must declare at least one database access, ensuring that no empty specifications or architectural impact specifications are generated.

Modeling Rules Modeling rules define the operating protocols for generating a specification that is not only correct, but also optimized for automatic parsing. These guidelines force the model to maintain a minimal, standardized data structure.

- *Schema Minimization:* the model is instructed to generate only the keys required by the framework, explicitly avoiding the inclusion of optional fields if they are empty.
- *Operational Mapping and Tracing:* a 1:1 mapping constraint is set between each generated operation and the corresponding User Story *(US-ID)*, allowing for deterministic alignment between the model output and the operations defined in the Ground Truth (GT) during evaluation.
- *Quantitative Standardization:* to allow for deterministic evaluation of workloads, the *frequency* parameter is limited to a discrete domain $\{1, 5, 10\}$. Similarly, the *consistency* field is restricted to the values $\{H, L\}$, standardizing the categorization of transactional constraints.
- *Naming Convention:* Operation names must strictly follow the *camelCase* format and be globally unique. Similarly, entity names must be expressed in *PascalCase*. Compliance with these conventions prevents ambiguity in the system dependency graph and facilitates the unique identification of services and domain components in the final model.
- *Unidirectional Access Protocol:* this rule requires that each access to an attribute be strictly unidirectional within a single operation (read-only or write-only). The model cannot simulate *Read-Modify-Write* processes; if an operation requires the current value of an attribute to calculate a new one, it must assume that this information has been retrieved from a separate, previous lookup operation.

At the end of this phase, the structured prompt is fully defined and validated from a syntactic and semantic point of view. This prompt is then used, without further modification, as input for the next phase of automatic specification generation using the selected LLM models, described in Step 2.

3.2.3. Generation with multiple LLMs (Step 2)

The generation activity constitutes the operational core of Phase 1. In this phase, the structured prompt is submitted to the selected models in order to transform the User Stories and Entity Structure into a formal technical specification. The execution follows a controlled strategy, based on the determinism of the response and the independence of the generation sessions.

Selected Models The generation phase involves the set of LLMs previously introduced in Section 3.1. The selected models represent heterogeneous architectural and inference profiles, including general-purpose instruction-tuned architectures, reasoning-oriented variants, and efficiency-optimized long-context models.

Their inclusion enables a controlled comparative evaluation of structured specification generation under identical prompt and decoding configurations. The focus of this phase is not on model selection itself, but on the operational assessment of their behavior when subjected to the same deterministic generation strategy and single-turn interaction protocol.

Deterministic Generation ($T = 0$) All models were queried by setting the generation temperature to $T = 0.0$. In LLMs, the temperature controls the degree of randomness in token selection during the decoding process: values close to zero reduce the stochastic component, inducing deterministic generation and therefore fully reproducible with the same input.

This choice responds to specific engineering requirements. In a comparative evaluation context, the elimination of randomness guarantees the replicability of results and allows the differences observed to be attributed exclusively to the intrinsic capabilities of the models, avoiding interference due to random variations in the generation process.

Furthermore, the production of structured technical specifications requires a high degree of syntactic and semantic rigor. A deterministic process reduces the risk of uncontrolled deviations and hallucination phenomena, promoting compliance with the imposed structural constraints and ensuring the validity of the output for processing by the solver.

Single-Turn Generation Strategy The adoption of a Single-Turn generation strategy, based on a single atomic interaction for the entire input dataset, is motivated by the need to evaluate the analytical capabilities of models in a zero-shot scenario, without any form of iterative assistance. In this configuration, the model receives both the input data and the complete set of constraints in a single block of information, leaving it to its internal architecture to manage the complexity and overall consistency of the output.

This methodology allows two central aspects to be analyzed. First, it evaluates the model's ability to maintain contextual consistency across extended input windows, consistently applying global rules throughout the entire generated specification. Second, it eliminates any form of bias introduced by human interaction: in multi-turn scenarios, any corrections or suggestions can influence the final output, masking the actual limitations of the model.

As a result, the output produced reflects only the intrinsic capabilities of the LLM, providing a neutral and objective basis for comparison between the models analyzed and configuring the generation phase as a controlled test of the ability to translate unstructured requirements into coherent and processable technical specifications.

The result of this phase is a set of `YAML` specifications generated independently by each model, which constitute the raw input for the subsequent post-processing, validation, and alignment phase described in Step 3.

3.2.4. Post-processing, Validation and Alignment (Step 3)

This phase acts as a bridge between the raw output of the models and quantitative analysis, transforming the generated `YAML` files into datasets ready for evaluation. The process consists of three automated sub-phases:

Structural Post-processing During Phase I, it was decided not to include Forced Operations in the prompt provided to the models, with the aim of focusing exclusively on extracting the basic functional structure of the system, without introducing dependencies between operations.

To ensure compatibility between the generated specifications and the Cromlech solver, a structural post-processing phase was introduced through a dedicated script. This phase adds the `forced_operations` field to each operation, guaranteeing a uniform and complete schema before execution.

Rigorous Syntactic and Functional Validation To ensure the usability of the generated specifications and their correct interpretation by the Cromlech framework, a dedicated validation script has been developed. This tool performs a series of cross-checks between the `YAML` files produced by the models, the domain entity schema, and the reference User Stories dataset.

The validation process is structured on multiple levels:

- *Schema and Referential Integrity:* verified that each operation refers to a valid User Story identifier and that all entities and attributes declared in the `database_access` block belong to the data schema provided as input. Any reference to attributes not present in the schema is reported as an error, allowing hallucination phenomena to be identified.
- *Cromlech Semantic Rules:* the semantic constraints required by the framework are

checked, including the Consistency Rule, which imposes the value H in the presence of write operations, and the uniqueness of operation names. In addition, it is verified that each entity defined in the schema is used at least once, in order to ensure complete coverage of the modeled system.

- *Attribute Overlap:* identifies any violations of the *Disjoint Rule*, reporting cases where the same attribute is declared both read and write within the same operation for a given entity.
- *Naming Convention Enforcement:* verifies compliance with the required naming conventions, such as the use of *PascalCase* for entity names and *camelCase* for attributes and operations. This check provides a qualitative indication of the models' ability to comply with formal constraints and formatting instructions.

The output of this phase consists of a detailed report that classifies the anomalies detected as blocking errors or simple warnings. This report forms the basis for a granular assessment of the reliability of the specifications generated and the behavior of the various LLMs analyzed.

Data Aggregation and Ground Truth Alignment Once the structural validation of the files generated by the LLMs has been completed, the alignment phase with the Ground Truth is carried out. This step is essential to transform the individual `YAML` specifications produced by the models into a comparative dataset, which is necessary for calculating performance metrics.

The main criticality in comparing the output of a generative model with a baseline lies in the unique identification of operations. Due to the stochastic nature of LLMs, the names of the generated operations can indeed vary, even though they refer to the same functional requirement. To overcome this problem, during the Ground Truth design phase, a unique identifier was associated with each operation, corresponding to the reference User Story.

The aggregation script uses this identifier to couple each operation generated by the model with the corresponding ideal operation present in the Ground Truth. If a model omits the generation of a specification for a given identifier, the operation is marked as *missing*, allowing omissions to be accounted for in the evaluation process as well.

Importantly, the introduction of the User Story identifier represents an extension of the input format envisioned by Cromlech, adopted exclusively for requirements search and traceability needs. In order to preserve a clear separation between the experimental evaluation logic and the framework optimization logic, aggregated files are used solely for calculating metrics and visualizing results.

At the end of this phase, each output generated by the LLMs is structurally valid, semantically consistent and aligned with the reference Ground Truth. The aggregated files therefore constitute a comparable and controlled dataset, suitable for calculating the quantitative metrics described in the next Step.

3.2.5. Metrics Calculation (Step 4)

Once the aggregation and alignment phase with the GT is completed, the workflow proceeds with the calculation of the evaluation metrics. This step aims to objectively quantify the quality of the `YAML` specifications generated by the models, evaluating both their structural correctness and semantic adherence to the original requirements.

The evaluation process was designed to reflect different dimensions of the problem of automatic generation of architectural specifications, being divided into two main axes:

- Formal and functional correctness of the generated operations.
- Semantic correctness between the operations produced and those present in the GT.

Structural Evaluation vs Ground Truth The first set of metrics is aimed at measuring the structural accuracy of the generated specifications with respect to the Ground Truth. In particular, each generated operation is compared with the corresponding reference operation, identified by the identifier of the associated user story.

This comparison allows the generated operations to be classified as correct, incorrect, or missing, allowing the calculation of classification-type metrics. This approach provides a quantitative measure of the model's ability to correctly reconstruct the functional structure of the system, including coverage of User Stories and compliance with required constraints.

Semantic Name Similarity Since generative models can assign names to operations different from those in GT while maintaining the same logical purpose, exact text comparison cannot be used. A semantic similarity metric based on Natural Language Processing (NLP) techniques is then employed.

In particular, the *all-MiniLM-L6-v2* embedding model, provided by the `SentenceTransformers` library, is used. This model was chosen because it represents a good compromise between semantic quality of representations and computational cost, being widely adopted in reading for text similarity and semantic clustering tasks. Furthermore, its compact architecture ensures stability and reproducibility of the evaluations, fundamental characteristics in a

controlled experimental context.

Each operation name generated by the LLM and the corresponding name in Grounf Truth are projected into dense vectors in a shared semantic space. The similarity between the two vectors is calculated using the *Cosine Similarity*:

$$\text{Similarity}(n_{llm}, n_{gt}) = \frac{\mathbf{v}_{llm} \cdot \mathbf{v}_{gt}}{\|\mathbf{v}_{llm}\|\|\mathbf{v}_{gt}\|}$$

The obtained value, in the range $[0, 1]$, provides a quantitative measure of the degree of semantic alignment between the identifier proposed by the model and the functional intent expressed in the Ground Truth, allowing to evaluate the quality of the naming regardless of its syntactic form.

Structural Metrics: Entity and Attributes For structural elements (entities involved, reading and writing attributes), the evaluation follows a binary classification approach. For each operation, the set of elements predicted by the model is compared with the set of GT. Standard Information Retrieval metrics are then calculated:

- *Precision:* measures the ability of the model not to include extraneous or incorrect attractants.
$$\text{Precision} = \frac{|S_{llm} \cap S_{gt}|}{|S_{llm}|}$$
- *Recall:* measures the model's ability to identify all attributes required by the original specification.
$$\text{Recall} = \frac{|S_{llm} \cap S_{gt}|}{|S_{gt}|}$$
- *F1-Score:* represents the harmonic mean between Precision and Recall, providing a synthetic index of generation quality for that given field.
$$F1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

Categorical Accuracy: Frequency and Consistency For categorical parameters, such as frequency and consistency, a point accuracy is calculated. The value is considered correct only in case of an exact match with the GT. This rigor is necessary because a cariation of these parameters drastically alters the architectural decisions of the solver when partitioning microservices.

The results of these analyses are finally consolidated into a global `YAML` report, which aggregates both the average statistics per model and the granular details for each user story. This structured representation constitutes the input for the next phase of data visualization, aimed at comparative model analysis.

3.2.6. Data Visualization (Step 5)

This phase of the workflow is dedicated to the visual synthesis of the results obtained in the quantitative evaluation phase. To facilitate the interpretation of data extracted from `YAML` reports and allow for immediate comparison between different LLMs, a suite of visualization scripts based on the `Matplotlib`, `Seaborn`, and `Pandas` libraries has been developed.

Each component of the suite is designed to analyze a specific aspect of generation:

- *Radar Charts:* provide a holistic view of the average performance of a single model. By simultaneously visualizing heterogeneous metrics (semantic similarity, F1-score, and categorical accuracy) on radial axes, they allow us to immediately identify systematic strengths and weaknesses of each LLM.
- *Comparative Radar:* extends the logic of the radar chart by superimposing the profiles of all analyzed models into a single graph. This tool is essential in the benchmarking phase to discriminate which model offers the best compromise between syntactic rigor and semantic understanding.
- *Performance Variability:* uses box plots to analyze the distribution of scores. This allows us to evaluate the stability of the model: a small dispersion indicates a consistent performance, while the presence of outliers highlights specific scenarios in which the model systematically fails.
- *Granular Analysis for User Story:* generates bar charts that show the performance of metrics for each individual operation. It is the main tool for debugging the prompt, allowing us to trace the textual requirements that caused the greatest interpretative difficulties.
- *Entity Truth Analysis:* focused on database mapping accuracy, this script generates aggregated histograms that compare real GT entities with those predicted by the LLM. Allows you to view omission errors (false negatives) or hallucinations of extra tables (false positives)

The integration of such tools ensures that evaluation is not limited to single aggregated

numerical values, but provides a multidimensional perspective on the capabilities of LLMs to operate as reliable assistants in the process of automatically generating software specifications.

3.2.7. Selection of the Optimal Generation (Step 6)

The final phase of *Phase I* concerns the experimental selection of the specification to be used as a baseline for the subsequent phases of the study. Starting from the set of `YAML` files generated by the different LLMs and evaluated through the metrics described in the previous Steps, a single representative specification is identified, called *Best-Generated*.

Within the experimental design adopted, this selection is not performed purely automatically, but occurs through a guided analysis process (*Human-in-the-loop*). This methodological choice is motivated by the exploratory nature of the experiment: quantitative metrics provide an objective assessment of model performance, but do not always allow for the exhaustive capture of architecturally relevant trade-offs.

In particular, the final decision jointly takes into account the aggregated numerical results and the qualitative evidence emerging from the data visualization phase. This approach allows you to select the specification that has the best overall balance between formal correctness, adherence to requirements, and performance stability across the entire set of User Stories.

The *Best-Generated* specification then represents the final output of *Phase I*; the corresponding `YAML` file is used as input for subsequent steps, where architectural constraints are explicitly introduced (definition of forced operations).

3.3. Phase 2: Forced Operation study

Phase 2 represents the transition from functional extraction to the definition of concrete architectural constraints. If in Phase 1 the goal was to reconstruct the structure of operations and data, here we focus on Forced Operations, constraints that indicate which operations should be colocated in the same microservice to meet consistency, logical affinity, or critical dependency requirements.

The analysis starts from the *Best-Generated* specification and the *Cromlech-Baseline*; before being submitted to the models, this specification is cleaned of the User Stories tracking fields introduced for evaluation, ensuring fully Cromlech-compliant input.

Unlike the initial phase, here we introduce a study of the stochasticity of the models,

allowing us to evaluate the stability of the architectural decisions suggested by the LLMs and the consistency of the constraints generated between different executions.

3.3.1. FO Prompting Strategy (Step 7)

The starting point of this phase is the integration of architectural knowledge within the consolidated functional specifications. The goal is to enrich the *Ground Truth* (GT) and the *Best-Generated* file with co-location constraints, without altering the pre-existing data structure.

The prompt structure, reported in full in Appendix A.2, follows the *Context-Augmented Update* paradigm. Unlike Phase 1, where generation was driven by unstructured requirements (User Stories), here the model operates on a technical basis already formatted in `YAML`. As illustrated in Figure 3.3, the prompt is organized into five sequential logic blocks:

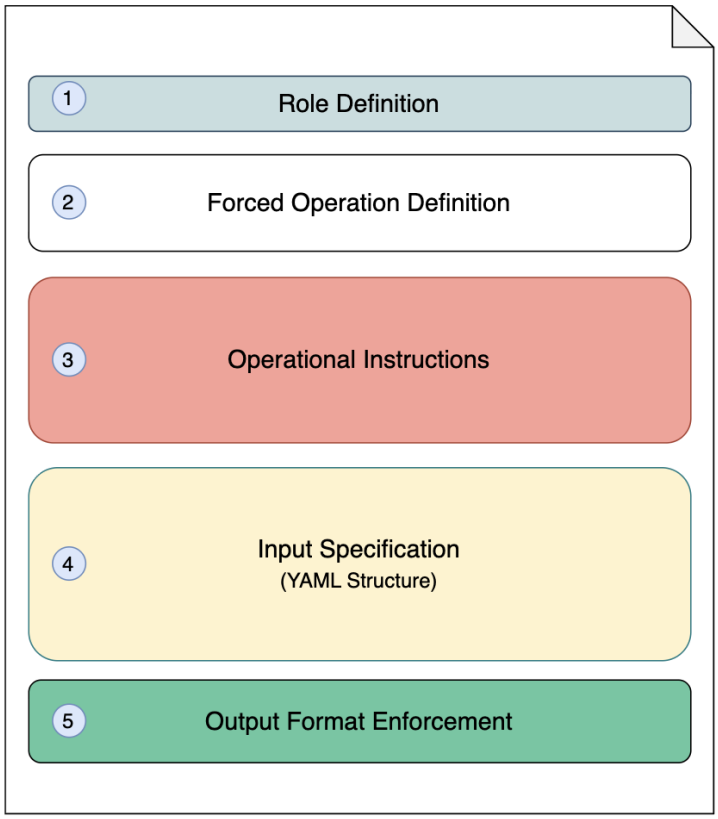

Figure 3.3: Structural blocks of the Forced Operations extraction prompt.

1. **Role Definition:** sets the LLM as *"Software Architect Assistant"*, providing the context needed to identify architectural consistencies and suggest which operations should be colocated in the same microservice.
2. **Forced Operation Definition:** provides a rigorous definition of *Forced Operation*

in the context of the Cromlech framework, avoiding arbitrary interpretations of the concept.

3. **Operational Instructions:** contains the main operating rules. In particular, the constraint of not modifying existing attributes is imposed, focusing the output exclusively on populating the list `forced_operations`.

4. **Input Specification (`YAML` Structure):** provides the complete `YAML` file to enrich, allowing the model to analyze dependencies between operations and logically deduce which ones should be colocated.

5. **Output Format Enforcement:** ensures that the model response is in pure `YAML` format, ready to be processed automatically without the need for further manual intervention.

This strategy transforms a "creation" task into a "constrained reasoning" task, significantly reducing the error space and maintaining the LLM's focus exclusively on the architectural logic of service affinities.

3.3.2. Generation with Multiple LLMs (Step 8)

At this stage, the goal is to evaluate the robustness and effectiveness of the architectural decisions suggested by the models in generating Forced Operations. For each LLM, two distinct prompts were used: one based on the *Cromlech Baseline* and one on the *Best-Generated* file obtained in Phase 1. This approach allows us to compare the impact of FO insertion from two different starting points: the original baseline and the optimal functional specification.

There are four selected models: *DeepSeek-Chat*, *DeepSeek-Reasoner*, *Mistral-Large-2512* and *Gemini-2.5-Flash*. For each model and each prompt, six independent generations were performed:

1. *Deterministic Inference* ($T = 0.0$)*:* a single deterministic execution, used as a stable reference to observe architectural decisions without stochastic variability.

2. *Stochastic Inference* ($T = 1.0$)*:* five independent generations with high temperature. The use of multiple runs allows us to measure the variability and stability of the FOs suggested by the models, allowing us to evaluate how consistent the architectural choices are between different runs.

This multiple strategy allows FO generation to be treated as an observable process across multiple instances, providing insights into the consistency of solutions and enabling a

systematic comparison between the baseline and the optimal specification.

At the end of this phase, each model produces six `YAML` files for each prompt (one deterministic and five stochastic), containing both the data structure and the partitioning assumptions suggested by the FOs. These files represent the generative output of Phase 2 and are the fundamental input for Phase 3, which is dedicated to running the Cromlech framework and benchmarking the results.

3.4. Phase 3: Architecture Evaluation

Phase 3 represents the final phase of the workflow, in which the functional specifications enriched with co-location constraints are evaluated in terms of architectural quality. The aim is to verify the validity of the partitionings generated by the LLM models and compare the stability and effectiveness of the produced architectures, taking into account the variability introduced by stochastic generations.

This step transforms the `YAML` files produced in Phase 2 into measurable results, providing an empirical basis for evaluating the reliability of the models in supporting the design of microservices systems. The workflow includes running the Cromlech framework, calculating architectural metrics, and visualizing the results for comparison between models and different starting points.

3.4.1. Cromlech Execution (Step 9)

The main component of architectural evaluation is the execution of the Cromlech framework, designed for the automated decomposition of monolithic systems into microservices. In this step, the `YAML` files produced in Phase 2 are processed to identify the optimal partitionings.

In total, the solver processes 50 `YAML` files: 48 generated by the models with the addition of Forced Operations (FO) and 2 baselines (*Cromlech Baseline* and *Best-Generated*), used as a reference to evaluate the impact of architectural constraints.

α Setting For architectural evaluation, Cromlech uses the MILP model-based objective function described in Section 1.3, which combines inter-service communication cost and load imbalance between microservices. In this experiment, the α parameter was set to 0.5, following the indications of [1], to ensure a balance between the two main metrics: inter-service traffic reduction and computational load balancing. This neutral configuration allows for an objective assessment of the impact of the Forced Operations generated by the LLMs, preventing the solver from favoring partitioning that is too granular or too

aggregated. In doing so, Cromlech acts as an impartial observer, clearly highlighting the effect of architectural constraints suggested by language models.

Solver Configuration and Time Constraints The microservices partitioning problem is inherently *NP-hard*, with a search space that grows combinatorially as the number of operations and entities increases. To ensure reliable results within a reasonable timeframe, the configuration follows the guidelines proposed in [1].

In particular, a *timeout* of 1 hour has been set for each solver run, an interval considered sufficient for Cromlech to converge to optimal solutions or, in cases of high complexity, return the best feasible solution with a contained *optimality gap*. All runs were conducted in a controlled environment, with dedicated and constant computational resources, in order to reduce the variability of solver performance and ensure objective comparability of results between the different inputs and models analyzed. This configuration allows to isolate the dynamics arising from architectural constraints and Forced Operations, without introducing distortions due to time limits or hardware fluctuations.

Running Cromlech produces two types of output: `.txt` files, containing the metrics calculated for each partitioning (*Cohesion*, *Communication Cost* and *Total Value*), and `.html` files, which display the microservices structure resulting from the decomposition. These data represent the objective basis of the next phase for quantitative analysis and comparison between models.

3.4.2. Metric Extraction (Step 10)

After executing Cromlech, this step collects and organizes the metrics produced by the solver in order to quantitatively evaluate the resulting microservice partitionings.

The metrics extracted from the 50 `YAML` specifications (48 generated by the models with Forced Operations and 2 baseline configurations) are aggregated using Python scripts based on the `Pandas` library. This aggregation enables the computation of descriptive statistics, such as mean values and variance, which are subsequently used to assess model stability and to compare the different experimental configurations.

3.4.3. Visualization and Comparative Analysis (Step 11)

The final step of *Phase 3* consists in transforming the numerical data produced by Cromlech into visual evidence, in order to compare the different LLMs with respect to the Ground Truth and evaluate the impact of temperature on the stability of architectural decisions.

Visual analysis is mainly divided into three types of output:

1. *Comparative Radar Charts:* They display the trade-off between *Cohesion*, *Communication Efficiency* and *Total Value*, allowing you to compare the performance of the models (*deepseek, mistral, gemini*) both with Cromlech baselines and against file versions without Forced Operations.

2. *Statistical Box Plots:* They represent the distribution of the *Total Value*, *Cohesion* and *Communication Efficiency* $(1 - CommCost)$ over the five stochastic runs ($T = 1.0$). Narrow boxes indicate stability in the architectural decisions of the model, while greater dispersion signals uncertainty in the identification of Forced Operations.

3. *Summary Tables:* Automated tables that report means, variances, and extreme values for each tested configuration, providing a comprehensive quantitative framework for comparative analysis.

Additionally, visual comparison of the `.html` files generated by Cromlech allows you to qualitatively evaluate the structure of the microservices. This allows us to observe whether Forced Operations have caused excessive aggregations ("super-services") or have maintained a balanced partitioning, offering direct evidence of the consistency of the solutions suggested by the LLMs.

The integration of quantitative metrics and stochastic analyses ensures that the comparison is not limited to single, fortuitous instances, but reflects the true reasoning ability and robustness of the generated architectures. This methodological framework forms the basis for the critical analysis presented in the next chapter, where the collected data will be discussed to evaluate the effectiveness of LLMs in supporting the design of microservices systems.

4 | Results

This chapter presents and analyzes the experimental results obtained by applying the workflow described in Chapter 3 to the case studies considered. The main objective of the analysis is to evaluate the impact of Forced Operations generated by Large Language Models on the quality of microservices architectures produced by the Cromlech framework, with particular attention to the stability of the solutions and the variability introduced by the stochastic component of the models.

The case studies selected for experimental evaluation are:

- *Trainticket:* a benchmark for microservices architectures, simulating a large-scale rail reservation system;
- *Tutored:* a platform operating in the EdTech (Educational Technology) sector, designed to support students and young graduates in orientation and integration into the world of work.

The results are organized according to a dual analysis perspective. On the one hand, the behavior of different LLMs under deterministic conditions ($T = 0.0$) is examined, in order to compare the architectural decisions suggested by the models in the absence of stochastic variability. On the other hand, the stochastic setting ($T = 1.0$) is analyzed, which allows studying the robustness and consistency of Forced Operations generated on multiple independent runs.

The analysis is structured by case studies, distinguishing between starting architectures and specifications enriched with Forced Operations, in order to isolate the effect of architectural constraints suggested by LLMs.

4.1. Trainticket

4.1.1. Initial Architectural Structure

The evaluation of the architectural structures generated for the *Trainticket* case study was conducted by applying the *metrics calculation step* described in Section 3.2.5. In order to provide a synthetic view of the overall performance of the analyzed models, the results are initially summarized in a comparative table.

Model	Entity F1	Read F1	Write F1	Name Sim.	Freq. Acc.	Cons. Acc.
DeepSeek-Chat	0.86	0.63	0.73	0.82	26%	55%
DeepSeek-Reasoner	0.86	0.61	0.73	0.82	33%	66%
Mistral-Large-2512	0.86	0.62	0.73	0.76	49%	56%
Gemini-2.5-Flash	0.85	0.59	0.74	0.76	37%	56%

Table 4.1: (Trainticket) Performance comparison across LLM models for architectural synthesis.

To facilitate the interpretation of the results reported in Table 4.1, a comparative radar chart was also used, shown in Figure 4.1. The graph allows you to simultaneously visualize the main evaluation metrics for each model, providing an immediate representation of the different performance profiles.

The radar chart shows how the models exhibit different behaviors on the various dimensions considered. In particular, structural metrics (Entity, Read, and Write F1) are overall aligned, while more marked differences are observed in categorical and semantic accuracy metrics. This visual representation allows us to highlight the main trade-offs between structural correctness, semantic consistency, and architectural decision stability.

In the following, the distribution of evaluation metrics and the coverage of domain entities are analyzed for each model. For completeness, the performance of the metrics at the single user story level, useful for highlighting intra-model variability and the presence of outliers, is reported in Appendix A.3.

Figure 4.1: (Trainticket) Comparative radar chart of architectural synthesis performance across LLM models.

DeepSeek-Chat Analysis

The box plot shown in Figure 4.2 shows the distribution of scores obtained on the main structural and semantic metrics. The *Name Similarity* metric has a high median, with a mean value of 0.82, indicating that in most cases the model is able to assign semantically consistent names to the generated operations. However, the presence of some outliers with significantly lower values suggests that specific user stories are particularly ambiguous or complex to interpret, leading to lexical choices less aligned with the Ground Truth.

Regarding the *Entity F1*, the distribution highlights a strong concentration of values close to unity, consistent with the aggregate mean value of 0.86. This result indicates a good ability of the model to correctly identify relevant entities in most operations. However, there are some isolated cases with lower scores, attributable to omissions or incorrect associations between entities and operations.

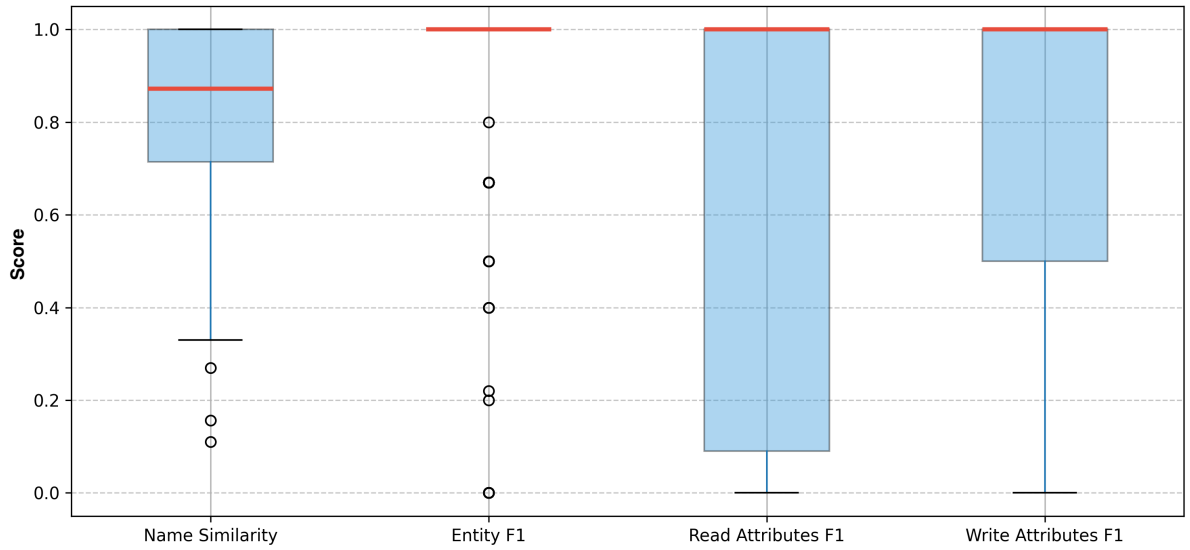

Figure 4.2: (Trainticket) Box Plot for DeepSeek-Chat.

A more marked variability emerges instead in the metrics of *Read Attributes F1* and *Write Attributes F1*. The mean values, 0.63 and 0.73, respectively, reflect a greater difficulty for the model in consistently modeling access to entity attributes. This dispersion is particularly evident in user stories that describe complex interactions or multi-entity operations, where the distinction between read and written attributes is not always uniquely made explicit.

The entity coverage analysis, illustrated in Figure 4.3, provides a more granular view of the model's behavior. For each entity in the Trainticket domain, the graph distinguishes between correctly identified occurrences, omitted entities, and incorrectly introduced entities. It is observed that some entities have a high frequency of correct recognitions, while others show a significant number of omissions, highlighting a non-uniform coverage of the entire domain. The presence of extra entities, although limited to specific cases, also indicates episodes of overgeneralization, in which the model introduces concepts not explicitly required by the reference specification.

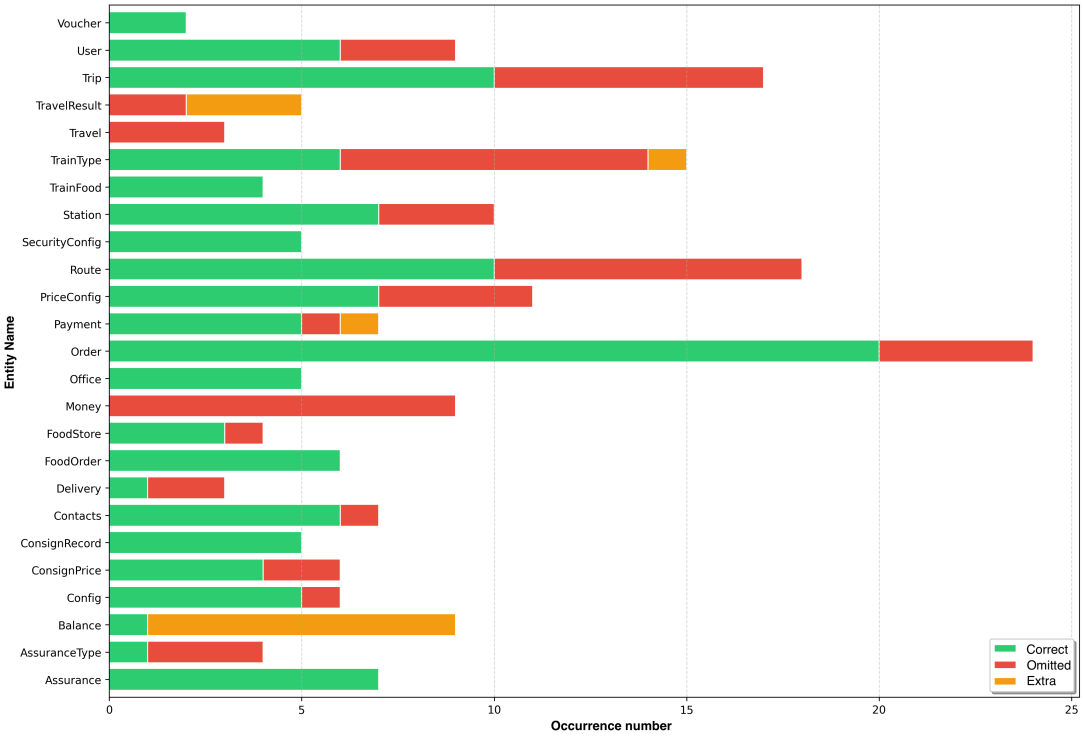

Figure 4.3: (Trainticket) Entity Coverage for DeepSeek-Chat.

DeepSeek-Reasoner Analysis

The box plot shown in Figure 4.4 shows an overall distribution similar to that observed for DeepSeek-Chat. The *Name Similarity* metric has a high median, with a mean value of 0.82, indicating a good ability of the model to generate semantically consistent names for operations. Again, the presence of some outliers with lower values suggests that certain user stories are more difficult to interpret correctly lexically.

Regarding the *Entity F1*, the distribution highlights a significant concentration of values close to unity, consistent with the aggregate mean value of 0.86. This result confirms a solid ability of the model to correctly identify the entities involved in the operations described by user stories. Cases with lower scores remain limited and can be traced back to omissions or errors in association between entities and operations.

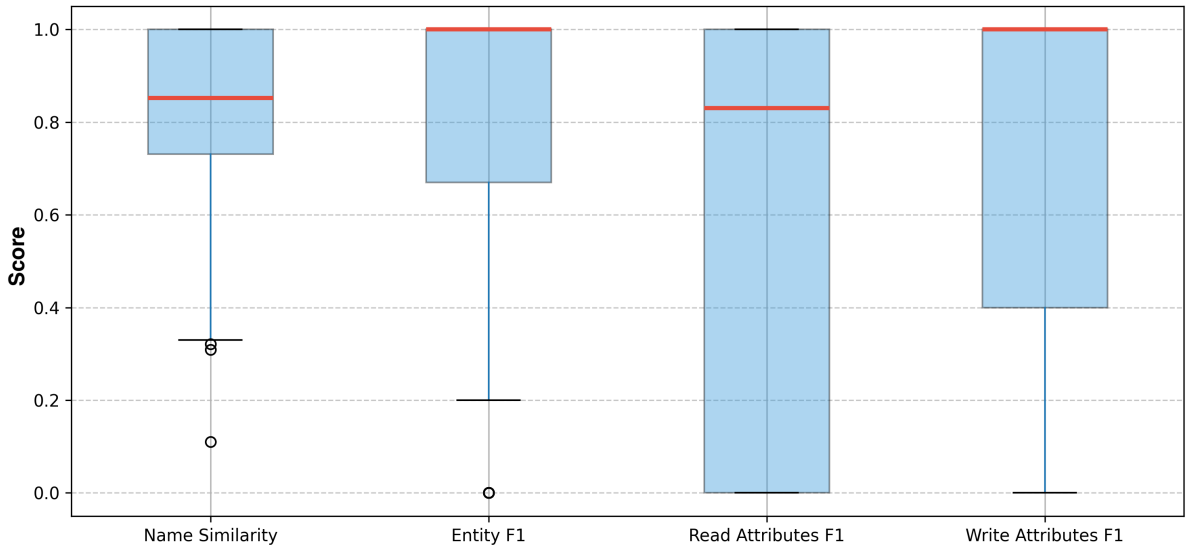

Figure 4.4: (Trainticket) Box Plot for DeepSeek-Reasoner.

Greater variability is observed in the *Read Attributes F1* and *Write Attributes F1* metrics. In particular, the mean value of the Read Attributes F1 of 0.61 is slightly lower than that of DeepSeek-Chat, suggesting a greater difficulty for the model in consistently identifying the read attributes. The Write Attributes F1, with an average value of 0.73, shows a more stable behavior, although characterized by a significant dispersion of the scores. This variability reflects the inherent complexity of operations involving state updates, especially in multi-entity or implicitly described scenarios.

The entity coverage analysis, illustrated in Figure 4.5, highlights an overall behavior in line with that observed for DeepSeek-Chat. Some domain entities show a high frequency of correct acknowledgements, while others are characterized by a significant number of omissions. The presence of incorrectly introduced entities, although limited to specific cases, indicates episodes of overgeneralization, in which the model tends to introduce concepts not explicitly required by the reference specification.

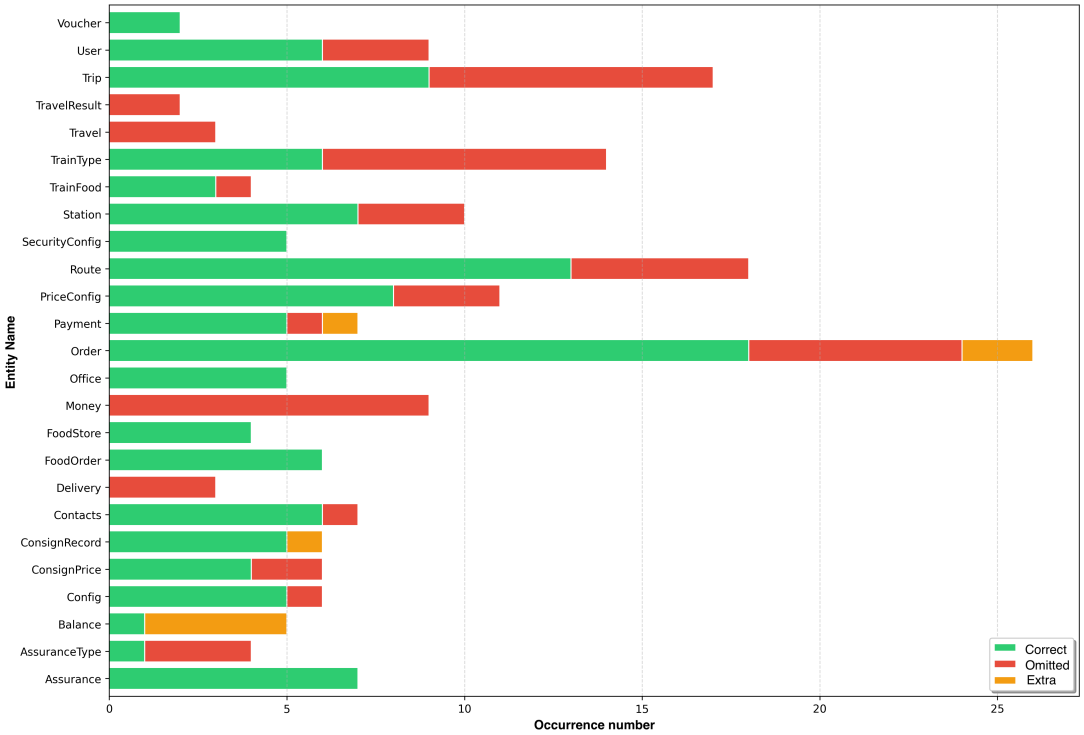

Figure 4.5: (Trainticket) Entity Coverage for DeepSeek-Reasoner.

Mistral-Large-2512 Analysis

The box plot shown in Figure 4.6 highlights a distribution of scores characterized by good overall stability. The *Name Similarity* metric has a high median, with a mean value of 0.76, slightly lower than the DeepSeek models. This indicates a generally adequate ability to generate semantically consistent denominations, albeit with greater variability and the presence of some outliers associated with more complex or less lexically explicit user stories.

The distribution of the *Entity F1* shows a strong concentration of values close to unity, consistent with the aggregate mean value of 0.86. This result indicates that the model is generally able to correctly identify the relevant entities involved in the operations described by user stories. However, the presence of some isolated cases with lower scores suggests that, in certain instances, the model may incur omissions or partial coverage of the required entities.

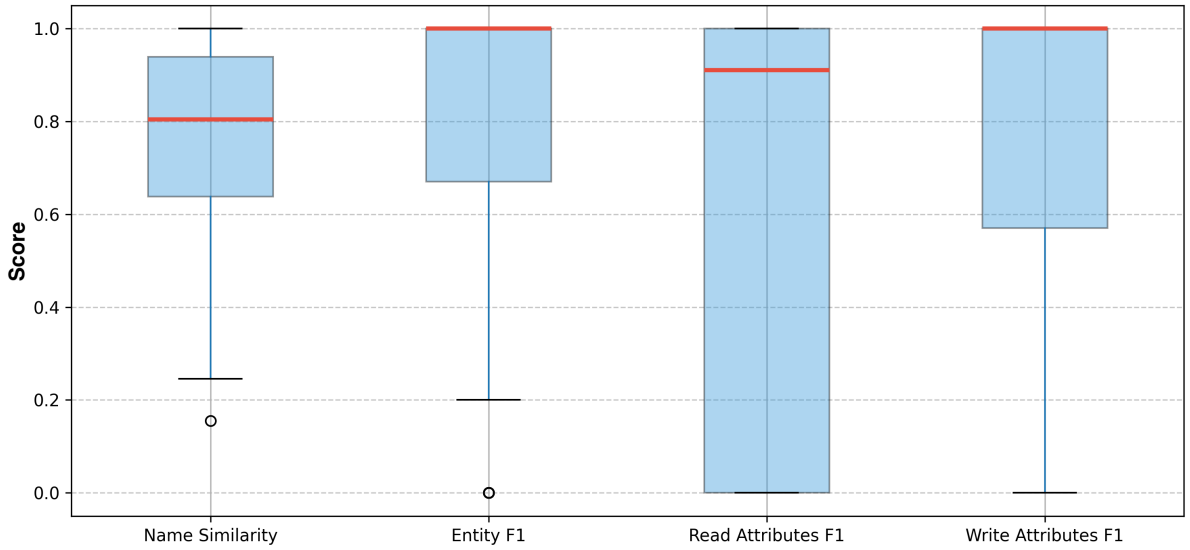

Figure 4.6: (Trainticket) Box Plot for Mistral-Large-2512.

Regarding attribute access metrics, the *Read Attributes F1* and *Write Attributes F1* have average values of 0.62 and 0.73, respectively. The dispersion observed in box plots indicates significant variability, particularly for attribute reading, suggesting a difficulty in consistently modeling operations involving multiple entities or implicitly describing relevant attributes. The writing metric, on the other hand, is slightly more stable, although it also shows a non-negligible variability.

The entity coverage analysis, illustrated in Figure 4.7, highlights overall behavior similar to that observed for DeepSeek models. Some domain entities have a high frequency of correct recognitions, while others are characterized by a significant number of omissions. The presence of incorrectly introduced entities, although limited to specific cases, suggests episodes of overgeneralization, in which the model tends to introduce concepts not explicitly required by the reference specification.

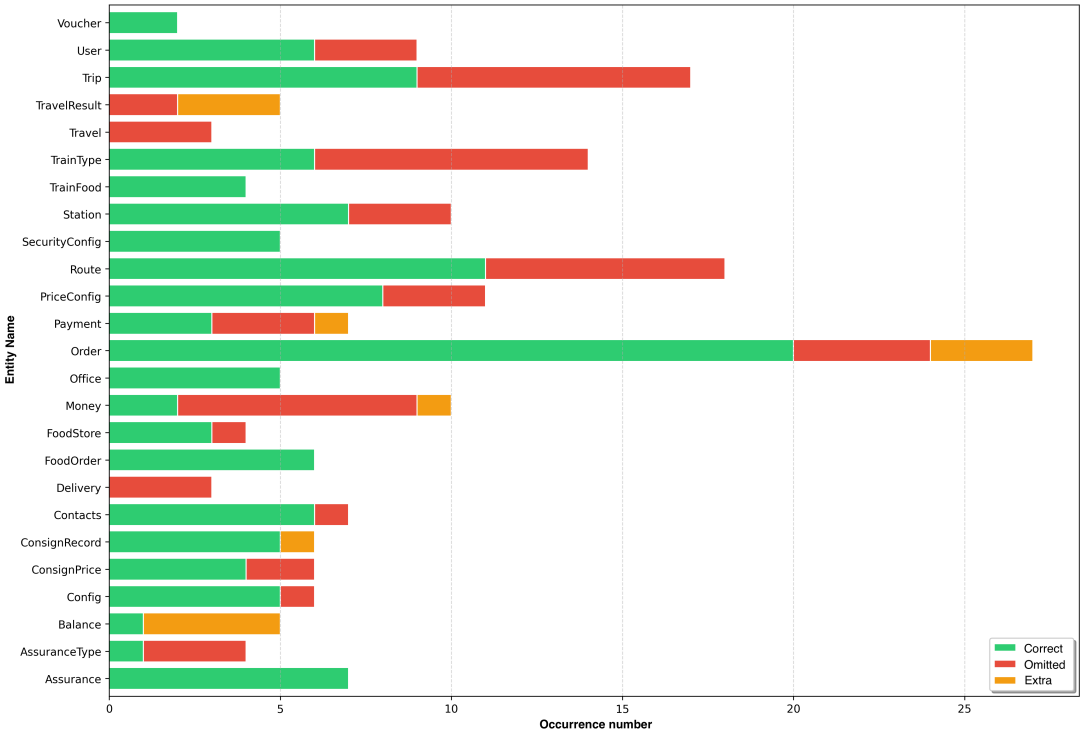

Figure 4.7: (Trainticket) Entity Coverage for Mistral-Large-2512.

Gemini-2.5-Flash Analysis

The box plot shown in Figure 4.8 highlights a distribution of scores characterized by good overall stability. The *Name Similarity* metric has a high median, with a mean value of 0.76, consistent with what was observed for Mistral-Large-2512 and slightly lower than the DeepSeek models. This result indicates a generally adequate ability to assign semantically consistent names to operations, although there are several outliers with lower values, which can be associated with particularly ambiguous or poorly formulated user stories.

The distribution of the *Entity F1* shows a concentration of values close to unity, consistent with the aggregate mean value of 0.85. This result suggests that the model is generally able to correctly identify the relevant entities involved in the operations described by user stories. However, the presence of some cases with low scores highlights episodes of omission or incomplete coverage of the required entities.

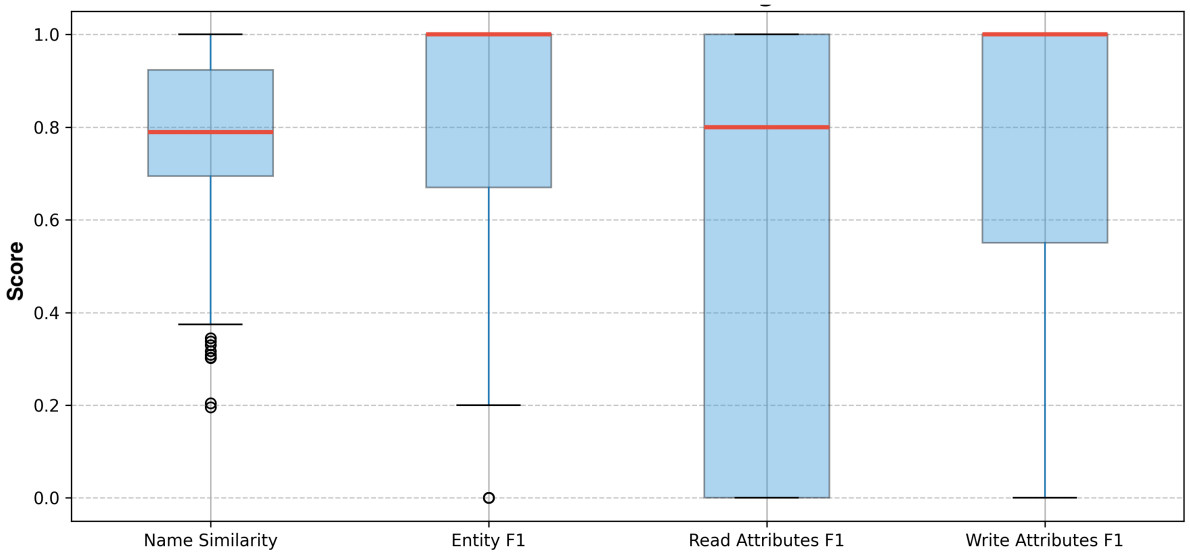

Figure 4.8: (Trainticket) Box Plot for Gemini-2.5-Flash.

Particularly marked variability emerges in attribute access metrics. The *Read Attributes F1* has an average value of 0.59, the lowest among the analyzed models, indicating a greater difficulty of the model in consistently identifying the read attributes. The *Write Attributes F1*, with an average value of 0.74, is more stable, although it shows a significant dispersion of the scores. This behavior reflects the inherent complexity of operations involving state updates, especially in multi-entity or implicitly described scenarios.

The entity coverage analysis, illustrated in Figure 4.9, highlights heterogeneous domain coverage. Some entities have a high frequency of correct recognitions, while others are characterized by a significant number of omissions. Furthermore, the non-negligible presence of incorrectly introduced entities indicates a tendency towards overgeneralization, in which the model introduces concepts not explicitly required by the reference specification.

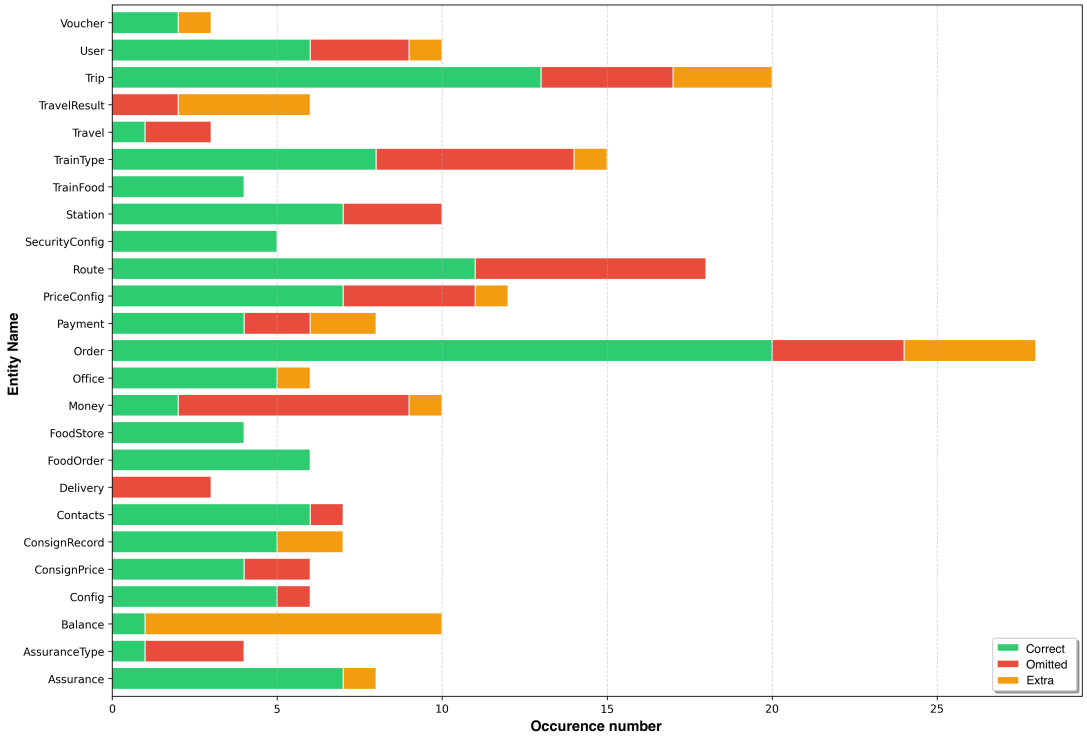

Figure 4.9: (Trainticket) Entity Coverage for Gemini-2.5-Flash.

In order to select the best generated architecture for the Trainticket case study, a set of criteria oriented towards the overall quality of the produced specification was considered, including (i) domain entity coverage, (ii) semantic consistency in denominations, and (iii) stability and consistency of architectural decisions. In particular, in addition to the aggregate mean values reported in Table 4.1, the variability of the metrics at the single user story level and the per-entity coverage highlighted by the *entity coverage* graphs were considered.

Based on these considerations, the architecture generated by **DeepSeek-Reasoner** was selected as *best generated*. The model shows a performance profile characterized by greater stability on semantic consistency and categorical accuracy metrics, highlighted in particular by the higher value of Consistency Accuracy and good levels of Name Similarity. With adequate coverage of the domain's core entities (Figure 4.5), DeepSeek-Reasoner produces an overall consistent architectural specification that is robust to user story variability, making it an appropriate basis for subsequent evaluation phases and for analyzing the impact of Forced Operations.

4.1.2. Forced Operations Impact on Architecture Evaluation

This section analyzes the impact of Forced Operations on the architectural properties evaluated using the Cromlech framework, considering both the *Cromlech Baseline Architecture* and the *Best Generated Architecture*. The goal is to evaluate how introducing guided changes to architectural decomposition influences quality metrics, isolating the effect of Forced Operations from the initial architecture configuration.

All the evaluations reported in this section were obtained by running the framework with the configuration parameters adopted in the study on the Trainticket [1] system, in order to ensure the comparability of the results and adherence to the original experimental scenario. In particular, the following were used for all executions:

- a balance parameter value $\alpha = 0.5$;
- a maximum number of services equal to 27;
- an execution time limit of 1 hour for each evaluation.

Within this section, the analysis is divided into two main subsections: the first dedicated to the impact of Forced Operations on Cromlech Baseline Architecture, and the second focused on Best Generated Architecture. For each case, the results are analyzed separately in deterministic ($T = 0.0$) and stochastic ($T = 1.0$) configurations, combining the analysis of Cromlech metrics with the observation of structural differences in terms of architectural decomposition.

Baseline Comparison without Forced Operations

Before analyzing the impact of Forced Operations, it is useful to compare the two reference architectures in their absence. Table 4.2 reports the evaluation metric values obtained via Cromlech for both architectures, considering a configuration without Forced Operations.

Model	**Cohesion**	**Communication Cost**	**Total Value**	**# Services**
Cromlech Baseline	0.4063	0.1041	0.1511	25
Best Generated	0.3798	0.1112	0.1343	26

Table 4.2: (Trainticket) Comparison between Cromlech Baseline and Best Generated architecture without Forced Operations.

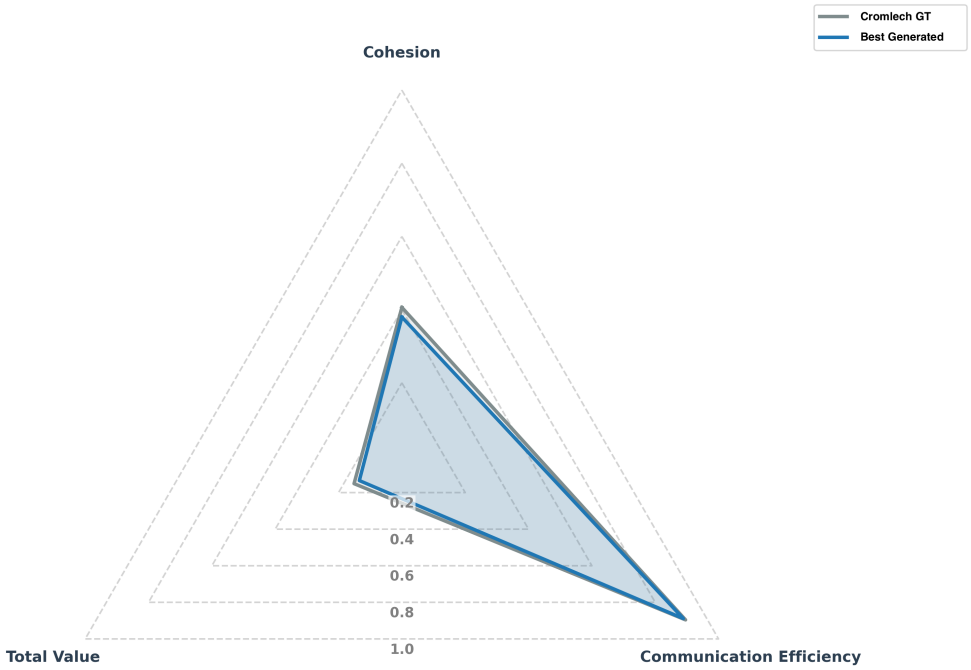

Figure 4.10: (Trainticket) Radar chart comparison between Cromlech Baseline and Best Generated architecture without Forced Operations.

The results show that the Best Generated has overall comparable values to those of the Cromlech Baseline on all the metrics considered. In particular, the observed differences in terms of *Cohesion*, *Communication Cost*, and *Total Value* are small, indicating that the generated architecture significantly approximates the reference configuration.

For completeness, Table 4.2 also shows the number of resulting services for each architecture. In the absence of Forced Operations, the Best Generated has a number of services substantially aligned with that of the Cromlech Baseline (26 vs. 25), indicating that the generation process does not introduce significant structural variations in terms of architectural decomposition.

These differences are visually summarized in the radar chart in Figure 4.10, which highlights similar architectural profiles between the two solutions. This initial comparison allows us to establish a common reference point for subsequent analysis, allowing us to more clearly isolate the effect of Forced Operations on the architectural properties evaluated.

Impact of Forced Operations on: Cromlech Baseline Architecture

This section analyses the impact of Forced Operations applied to the *Cromlech Baseline Architecture*, using different LLMs for their generation. Because the input architecture is

kept fixed, the observed differences in Cromlech metric values are directly attributable to the Forced Operations generated by the models.

Model	T	Cohesion	Comm. Cost	Total Value	# Services
Baseline (No FO)	-	0.4063	0.1041	0.1511	25
DeepSeek-Chat	0.0	0.5341	0.0953	0.2194	13
DeepSeek-Chat	1.0	0.5015 ± 0.0181	0.0746 ± 0.0011	0.2135 ± 0.0028	11,4 ± 3.8 (7-16)
DeepSeek-Reasoner	0.0	0.5341	0.0953	0.2194	13
DeepSeek-Reasoner	1.0	0.5341 ± 0.0	0.0953 ± 0.0	0.2194 ± 0.0	13 ± 0.0 (13-13)
Gemini-2.5-Flash	0.0	0.5341	0.0953	0.2194	13
Gemini-2.5-Flash	1.0	0.5674 ± 0.0056	0.1078 ± 0.0008	0.2298 ± 0.0005	13.4 ± 0.9 (13-15)
Mistral-Large-2512	0.0	0.7714	0.1535	0.3090	18
Mistral-Large-2512	1.0	0.5777 ± 0.0095	0.1288 ± 0.0056	0.2245 ± 0.0001	14 ± 2.2 (13-18)

Table 4.3: (Trainticket) Cromlech evaluation metrics (mean ± standard deviation) for Cromlech Baseline architecture with Forced Operations.

In addition to the evaluation metrics (*Cohesion*, *Communication Cost* and *Total Value*), the resulting number of services is also considered, as a direct indicator of the degree of architectural decomposition. In particular, significant variations in the number of services may explain differences in observed structural properties, such as increased cohesion or reduced communication costs.

Table 4.3 reports the values of the evaluation metrics obtained considering both the baseline architecture without Forced Operations, and the architectures resulting from the' application of the Forced Operations generated by the different LLMs, in deterministic ($T = 0.0$) and stochastic ($T = 1.0$) configurations. For stochastic setting, the results are expressed in aggregate form as the mean ± standard deviation.

In the following paragraphs, the results are analyzed separately for deterministic and stochastic settings.

Deterministic setting (T=0.0) In the deterministic setting, applying Forced Operations to the Cromlech Baseline Architecture produces a systematic improvement in evaluation metrics compared to the Forced Operations-free configuration. As reported in Table 4.3, all solutions generated by LLMs show an increase in Cohesion and Total Value, despite small changes in Communication Cost.

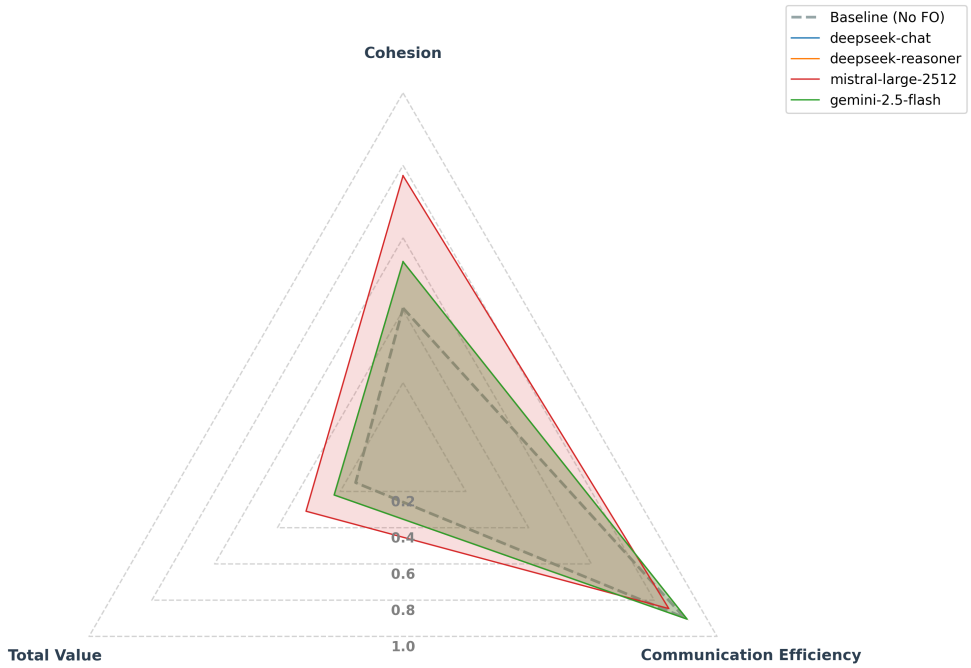

Figure 4.11: (Trainticket) Radar chart of Cromlech evaluation metrics for the Cromlech Baseline architecture with Forced Operations at T=0.0.

In particular, the Forced Operations generated by *DeepSeek-Chat*, *DeepSeek-Reasoner* and *Gemini-2.5-Flash* lead to numerically identical results on the three metrics considered (Cohesion = 0.5341, Communication Cost = 0.0953, Total Value = 0.2194), as also highlighted by the complete overlapping of the profiles in the radar chart in Figure 4.11. This behavior is accompanied by a number of services equal to 13 for all three models, indicating convergence towards the same architectural decomposition starting from the same input.

A different behavior is observed for *Mistral-Large-2512*, which in the deterministic setting generates Forced Operations associated with a larger number of services (18). This structural choice is reflected in a significantly higher value of Cohesion (0.7714), compared to an increase in the Communication Cost. Overall, this configuration leads to the highest value of Total Value (0.3090), as visible both in Table 4.3 and in the radar chart.

Overall, deterministic setting highlights how, in the absence of stochasticity, different models tend to converge towards similar architectural solutions, characterized not only by comparable metrics but also by an identical decomposition in terms of the number of services.

Stochastic setting (T=1.0) In the stochastic setting, each LLM was used to generate 5 independent instances of Forced Operations from the same Architecture. The results, reported in Table 4.3 as mean ± standard deviation, allow us to jointly evaluate the average performance, the stability of the solutions and the structural differences in terms of architectural decomposition.

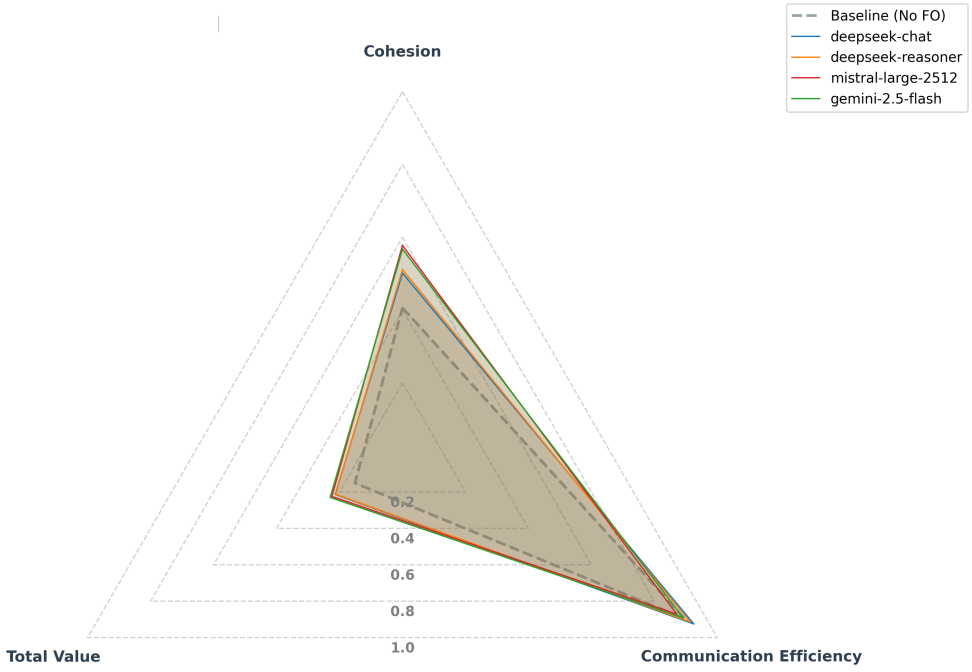

Figure 4.12: (Trainticket) Radar chart of Cromlech evaluation metrics for the Cromlech Baseline architecture with Forced Operations at T=1.0.

The radar chart in Figure 4.12 provides a summary of the average performance of the models, showing overall similar profiles across all metrics. However, this representation does not capture the variability introduced by different runs nor the differences in the resulting structure.

These aspects emerge clearly from the analysis of the box plot in Figure 4.13, which highlights the dispersion of the metrics and their relationship with the number of services generated. In this context, communication is represented by the *Communication Efficiency* ($1 - \textit{Communication Cost}$), so higher values indicate a lower penalty due to communication between services.

DeepSeek-Chat shows the greatest structural variability, with a number of services equal to 11.4 ± 3.8 (range 7–16). This dispersion is also reflected in Cromlech metrics, particularly in Cohesion (0.5015 ± 0.0181) and Total Value (0.2135 ± 0.0028). At the same time, the

model achieves the lowest average Communication Cost (0.0746 ± 0.0011), suggesting that more clustered solutions contribute to greater communication efficiency.

DeepSeek-Reasoner instead exhibits completely stable behavior: all executions produce the same architectural decomposition, with 13 services, and identical metrics (Cohesion $= 0.5341$, Total Value $= 0.2194$). This absence of variability, visible in the box plot as a collapse of the distribution on a single line, indicates that the Forced Operations generation process is in fact deterministic even at $T = 1.0$.

For *Mistral-Large-2512*, moderate variability in the number of services is observed (14.0 ± 2.2, range 13–18), associated with relatively high Cohesion values (0.5777 ± 0.0095) but a higher Communication Cost than the other models.

Gemini-2.5-Flash instead shows a more limited structural variability (13.4 ± 0.9 services), consistent with limited dispersions of metrics and a comparable average Total Value.

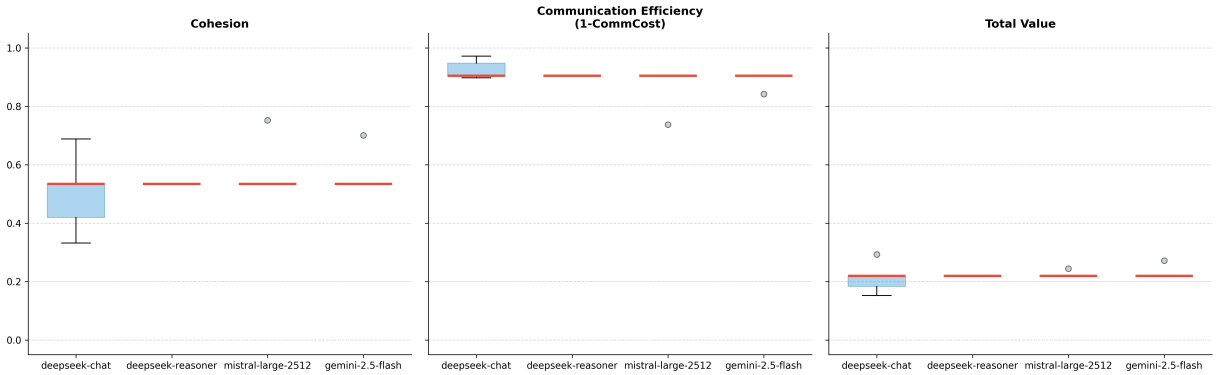

Figure 4.13: (Trainticket) Radar chart of Cromlech evaluation metrics for the Cromlech Baseline architecture with Forced Operations at T=1.0.

Overall, while the radar chart suggests similar average performance, the joint analysis of box plots and number of services highlights substantial differences in terms of stability and architectural decomposition strategies adopted by different LLMs.

Impact of Forced Operations on: Best Generated Architecture

This subsection analyses the impact of Forced Operations applied to the *Best Generated Architecture*, i.e. the architecture obtained from the automatic generation process. Unlike the previous case, where Forced Operations act on a manually designed baseline configuration, in this scenario they intervene on an architecture already optimized according to the generation framework criteria.

The aim of the analysis is to evaluate whether and to what extent Forced Operations are

able to further improve the architectural properties measured by Cromlech, or whether the generated architecture already represents a balance point with respect to the metrics considered.

Model	T	Cohesion	Comm. Cost	Total Value	# Services
Best Generated (No FO)	-	0.3798	0.1112	0.1343	26
DeepSeek-Chat	0.0	0.3798	0.1112	0.1343	26
DeepSeek-Chat	1.0	0.6843 ± 0.0008	0.0092 ± 0.0004	0.3375 ± 0.0004	14.6 ± 0.55 (14-15)
DeepSeek-Reasoner	0.0	0.5075	0.0	0.2537	10
DeepSeek-Reasoner	1.0	0.5451 ± 0.0091	0.0 ± 0.0	0.2725 ± 0.0023	12 ± 1.73 (11-15)
Gemini-2.5-Flash	0.0	0.5075	0.0	0.2537	11
Gemini-2.5-Flash	1.0	0.6214 ± 0.0035	0.0 ± 0.0	0.3107 ± 0.0009	12.8 ± 1.3 (11-14)
Mistral-Large-2512	0.0	0.6605	0.0	0.3303	14
Mistral-Large-2512	1.0	0.5626 ± 0.0010	0.0080 ± 0.0003	0.2773 ± 0.0001	12.4 ± 1.52 (11-15)

Table 4.4: (Trainticket) Cromlech evaluation metrics (mean ± standard deviation) for Best Generated architecture with Forced Operations.

Table 4.4 reports the results obtained considering both the Forced Operations-free configuration and the architectures resulting from the application of Forced Operations in deterministic ($T = 0.0$) and stochastic ($T = 1.0$) configurations. Again, for stochastic setting the results are expressed as mean ± standard deviation over 5 independent runs.

Deterministic setting (T=0.0) In the deterministic setting, applying Forced Operations to the Best Generated Architecture produces differentiated effects depending on the model used, as shown by the radar chart in Figure 4.14. Unlike the case of the Cromlech Baseline, where different models converged towards identical solutions, in this scenario more heterogeneous behaviors emerge, indicating a greater sensitivity of the starting architecture to Forced Operations.

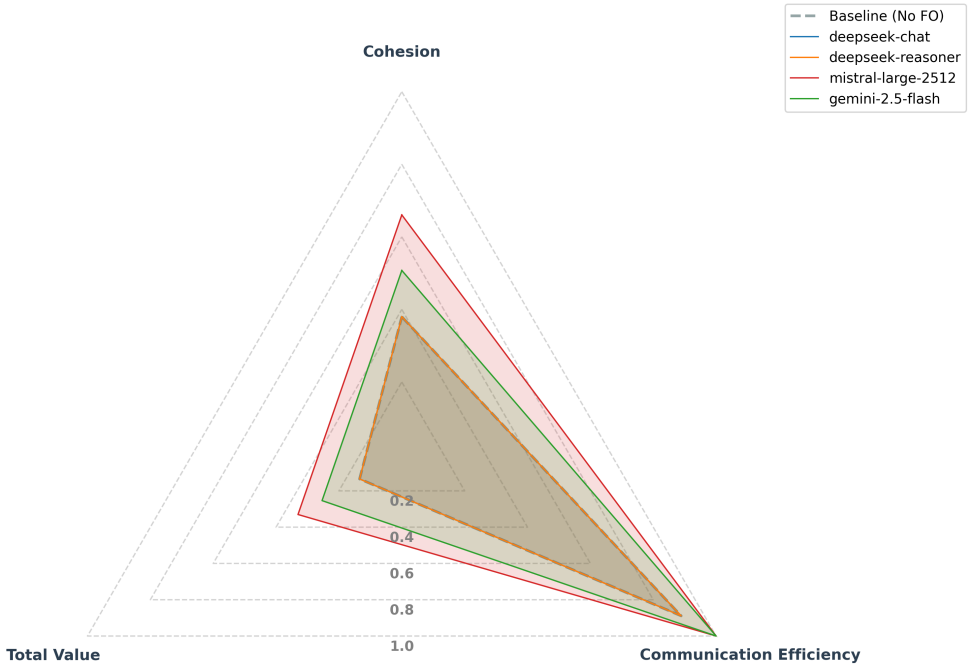

Figure 4.14: (Trainticket) Radar chart of Cromlech evaluation metrics for the Best Generated architecture with Forced Operations at T=0.0.

In particular, *DeepSeek-Chat* does not introduce any changes compared to the configuration without Forced Operations: the evaluation metrics remain unchanged (Cohesion = 0.3798, Communication Cost = 0.1112, Total Value = 0.1343), indicating that, in deterministic configuration, The model fails to identify Forced Operations that can further improve an already optimized architecture.

Conversely, *DeepSeek-Reasoner* and *Gemini-2.5-Flash* produce Forced Operations that dramatically reduce the Communication Cost to zero, while significantly increasing the Cohesion. This trade-off leads to a significant increase in Total Value, as evidenced by the expansion of the respective areas in the radar chart. Structurally, these improvements are associated with a reduction in the number of services, indicating a more consolidated decomposition.

The most marked behavior is observed for *Mistral-Large-2512*, which in the deterministic setting reaches the highest values of Cohesion and the maximum Total Value (0.3303), as also highlighted by the radar chart.

Overall, the deterministic setting analysis highlights how, in the case of Best Generated Architecture, Forced Operations can lead to significant improvements only for some models, while others tend to preserve the original configuration. This suggests that the effectiveness of Forced Operations depends not only on the model used, but also on the degree of initial

optimality of the starting architecture.

Stochastic setting (T=1.0) In stochastic setting, the application of Forced Operations to the Best Generated Architecture highlights significant differences both in terms of Cromlech performance and the stability of the solutions produced by the different LLMs.

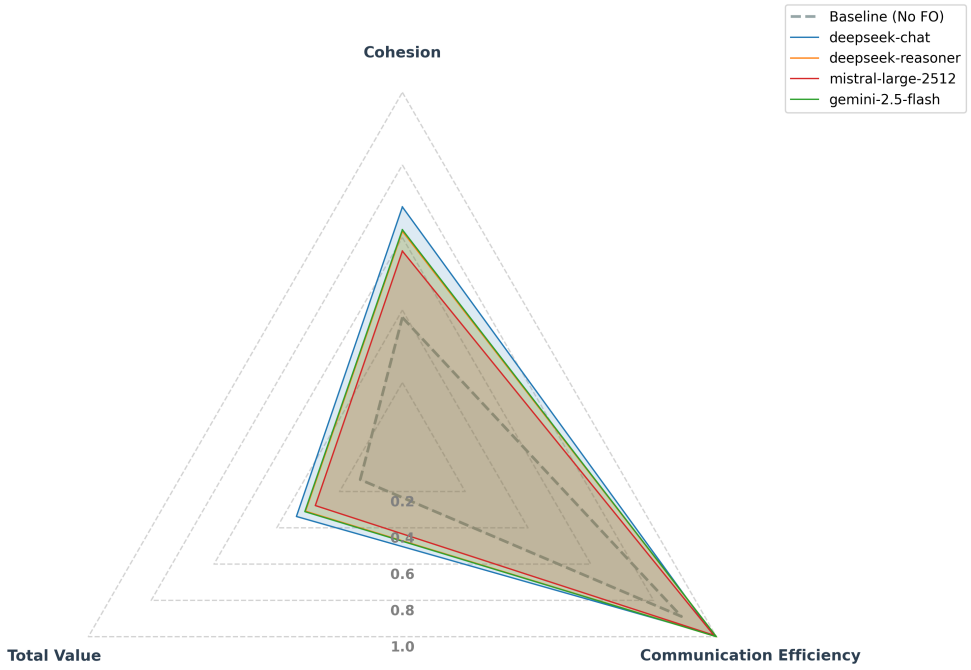

Figure 4.15: (Trainticket) Radar chart of Cromlech evaluation metrics for the Best Generated architecture with Forced Operations at T=1.0.

The radar chart in Figure 4.15 shows that, in terms of average values, the solutions produced by the different models present comparable profiles, in particular for the Communication Efficiency, which is close to the maximum value for all LLMs. However, this representation does not highlight the dispersion of the results nor the structural differences between individual runs.

These aspects emerge clearly from the analysis of the box plot in Figure 4.16, which reports the distribution of Cromlech metrics over the 5 runs for each model.

DeepSeek-Chat shows good overall stability, with an average Cohesion of 0.6843 ± 0.0008 and a Total Value of 0.3375 ± 0.0004, which represents the highest average value among the models considered. The box plot shows minimal dispersion of metrics, consistent with a relatively stable number of services (14.6 ± 0.55, range 14–15), indicating that the model explores similar solutions both structurally and qualitatively.

DeepSeek-Reasoner instead has greater variability in Cohesion (0.5451 ± 0.0091) and Total Value (0.2725 ± 0.0023), while maintaining zero Communication Cost in all runs. The box plot shows a more marked dispersion than DeepSeek-Chat, consistent with the greater variability in the number of services (12 ± 1.73, range 11–15), suggesting a greater exploration of the decomposition space.

For *Gemini-2.5-Flash*, the distribution shows intermediate stability: the Cohesion is equal to 0.6214 ± 0.0035 and the Total Value is equal to 0.3107 ± 0.0009, with a dispersion contained in the box plot. The number of services (12.8 ± 1.3) indicates moderate structural variability, reflected in slight fluctuations in metrics but without significant impacts on overall performance.

Finally, *Mistral-Large-2512* shows comparable structural variability to Gemini, with 12.4 ± 1.52 services (range 11–15), but lower mean values of Cohesion (0.5626 ± 0.0010) and Total Value (0.2773 ± 0.0001). The box plot shows a reduced dispersion of the metrics, indicating that, while exploring different decompositions, the model converges to qualitatively similar solutions.

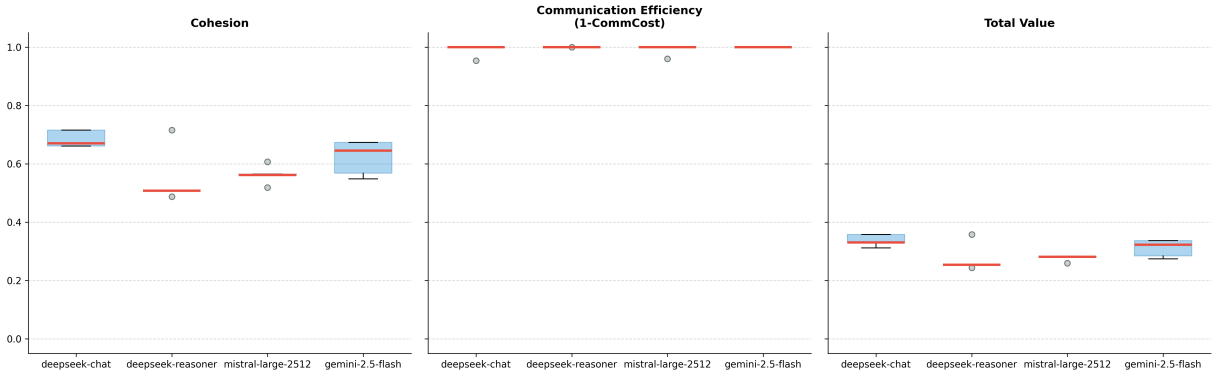

Figure 4.16: (Trainticket) Box plot of Cromlech evaluation metrics for the Best Generated architecture with Forced Operations at T=1.0.

Overall, while the radar chart suggests comparable average performance across models, box plot analysis highlights substantial differences in stability and space exploration capability of the solutions introduced by Forced Operations. In this context, *DeepSeek-Chat* emerges as the most stable and performing model on average, while *DeepSeek-Reasoner*, *Gemini-2.5-Flash* and *Mistral-Large-2512* show greater structural variability, associated with slightly lower overall performance.

4.2. Tutored

4.2.1. Initial Architectural Structure

The evaluation of the architectural structures generated for the *Tutored* case study was conducted by applying the *metrics calculation step* described in Section 3.2.5. In order to provide a synthetic view of the overall performance of the analyzed models, the results are initially summarized in a comparative table.

Model	Entity F1	Read F1	Write F1	Name Sim.	Freq. Acc.	Cons. Acc.
DeepSeek-Chat	0.62	0.19	0.59	0.85	54%	52%
DeepSeek-Reasoner	0.62	0.28	0.51	0.81	39%	54%
Mistral-Large-2512	0.65	0.24	0.57	0.83	37%	51%
Gemini-2.5-Flash	0.64	0.21	0.53	0.8	45%	52%

Table 4.5: (Tutored) Performance comparison across LLM models for architectural synthesis.

To facilitate the interpretation of the results reported in Table 4.5, a comparative radar chart was also used, shown in Figure 4.17. The graph allows you to simultaneously visualize the main evaluation metrics for each model, providing an immediate representation of the different profiles of performance.

The radar chart shows how the models exhibit different behaviors on the various dimensions considered. In particular, structural metrics (Entity, Read, and Write F1) are overall aligned, while more marked differences are observed in categorical and semantic accuracy metrics. This visual representation allows us to highlight the main trade-offs between structural correctness, semantic consistency, and architectural decision stability.

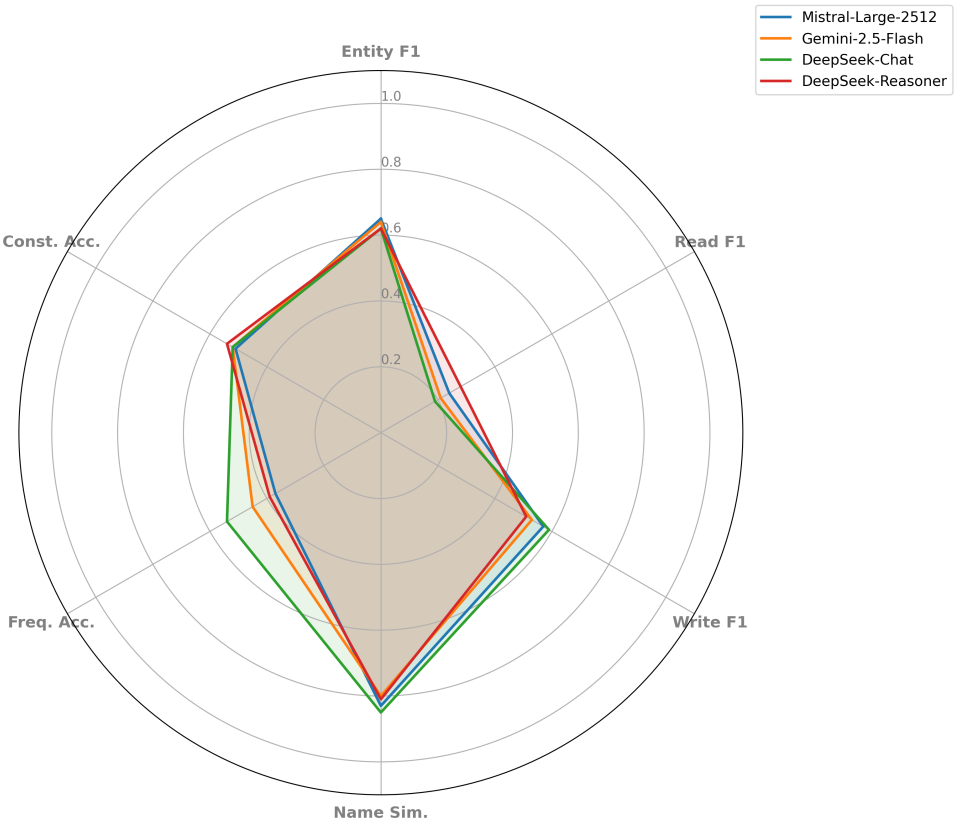

Figure 4.17: (Tutored) Comparative radar chart of architectural synthesis performance across LLM models.

In the following, the distribution of evaluation metrics and the coverage of domain entities are analyzed for each model. For completeness, the performance of the metrics at the single user story level, useful for highlighting intra-model variability and the presence of outliers, is reported in Appendix A.4.

DeepSeek-Chat Analysis

The box plot shown in Figure 4.18 shows the distribution of scores obtained by DeepSeek-Chat on the main structural and semantic evaluation metrics. The *Name Similarity* metric has a high median, close to 0.9, indicating that in most cases the model is able to assign semantically consistent names to the generated operations. The distribution, however, is characterized by the presence of some outliers with significantly lower values, suggesting that specific user stories, characterized by greater ambiguity or lexical complexity, can lead to nominal choices less aligned with the Ground Truth.

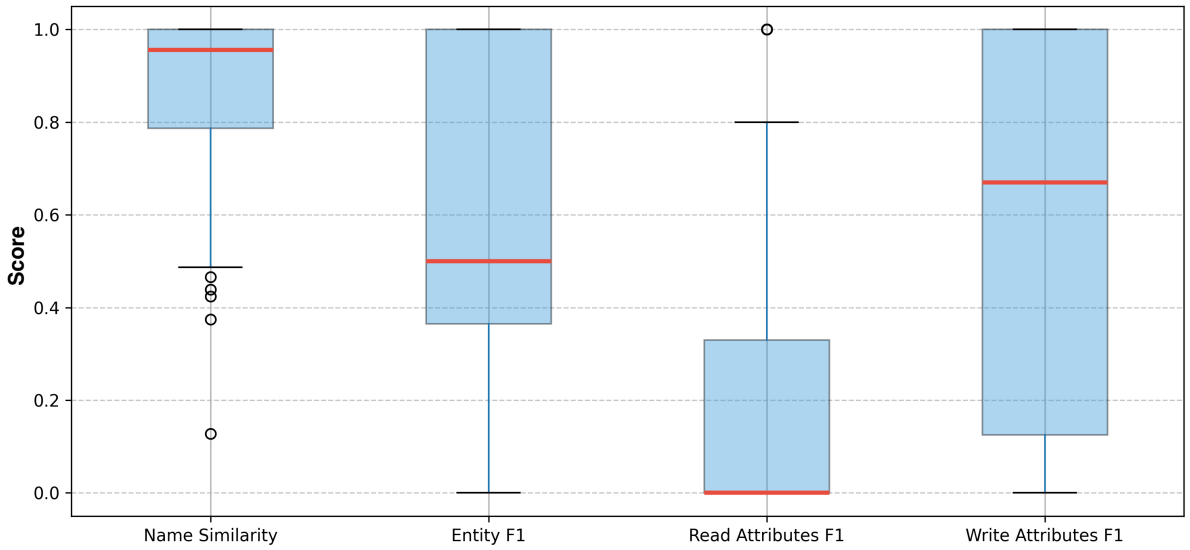

Figure 4.18: (Tutored) Box Plot for DeepSeek-Chat.

Regarding the *Entity F1*, the distribution shows marked variability, with scores ranging from values close to zero up to cases of complete entity recognition. The median is placed on intermediate values, consistent with the aggregate mean value reported in Table 4.5, reflecting an overall adequate ability of the model to identify relevant entities in the domain. This variability suggests that the correct identification of entities depends significantly on the clarity and level of detail with which they are made explicit in the textual requirements.

Even more pronounced variability emerges in the metric of *Read Attributes F1*, characterized by a median of zero and a distribution strongly skewed towards low values. This result indicates that, in most cases, DeepSeek-Chat encounters difficulties in identifying attributes read from operations, confirming the inherent complexity of this task when data access patterns are not explicitly defined in user stories. In contrast, the *Write Attributes F1* shows a more balanced distribution, with a higher median and a significant presence of high values, consistent with a mean value close to 0.6, suggesting that application state modification operations are more easily recognizable than simple read operations.

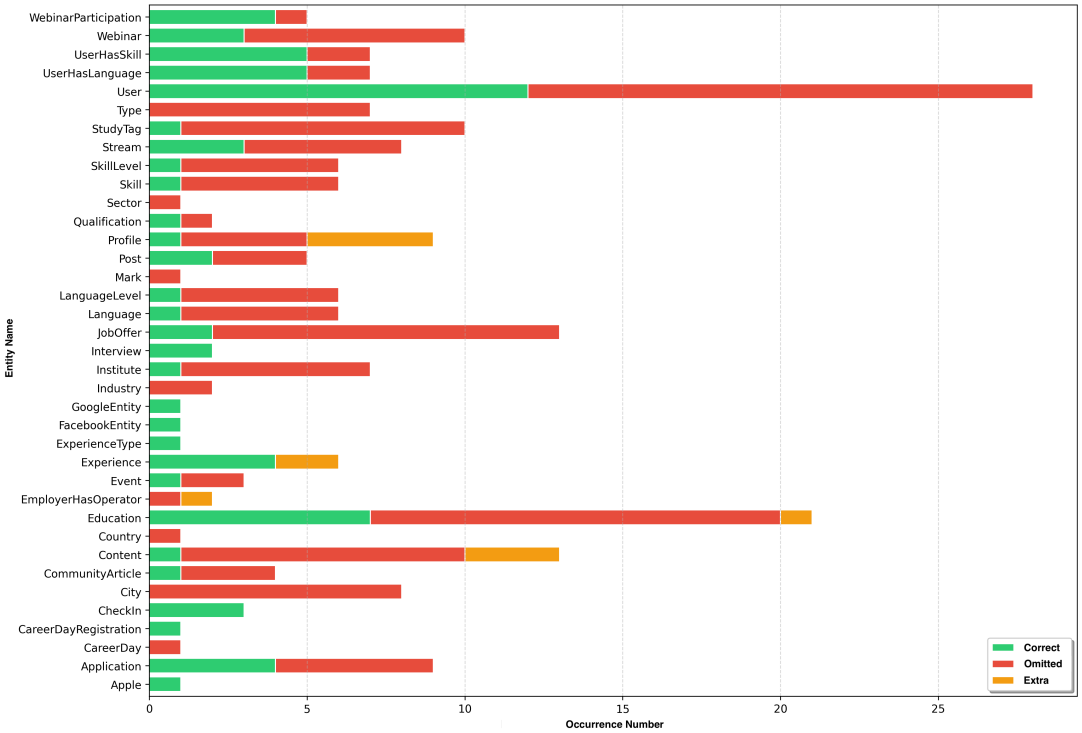

Figure 4.19: (Tutored) Entity Coverage for DeepSeek-Chat.

The entity coverage analysis, illustrated in Figure 4.19, provides a more granular view of the model's behavior on the Tutored domain. For each entity, the graph distinguishes between correctly identified occurrences, omitted entities, and incorrectly introduced entities. It is observed that some central entities of the domain, such as *User*, *Education* and *Application*, present a high number of correct recognitions, while more specific or less frequently cited entities show a significant rate of omissions.

DeepSeek-Reasoner Analysis

The box plot shown in Figure 4.20 shows the distribution of scores obtained by DeepSeek-Reasoner on the main structural and semantic evaluation metrics. The *Name Similarity* metric has a high median, close to 0.85, indicating that in most cases the model is able to generate names semantically consistent with the Ground Truth. The distribution, however, highlights the presence of some outliers with lower values, suggesting that specific user stories, characterized by lexical ambiguity or a less structured description, may lead to less aligned nominal choices.

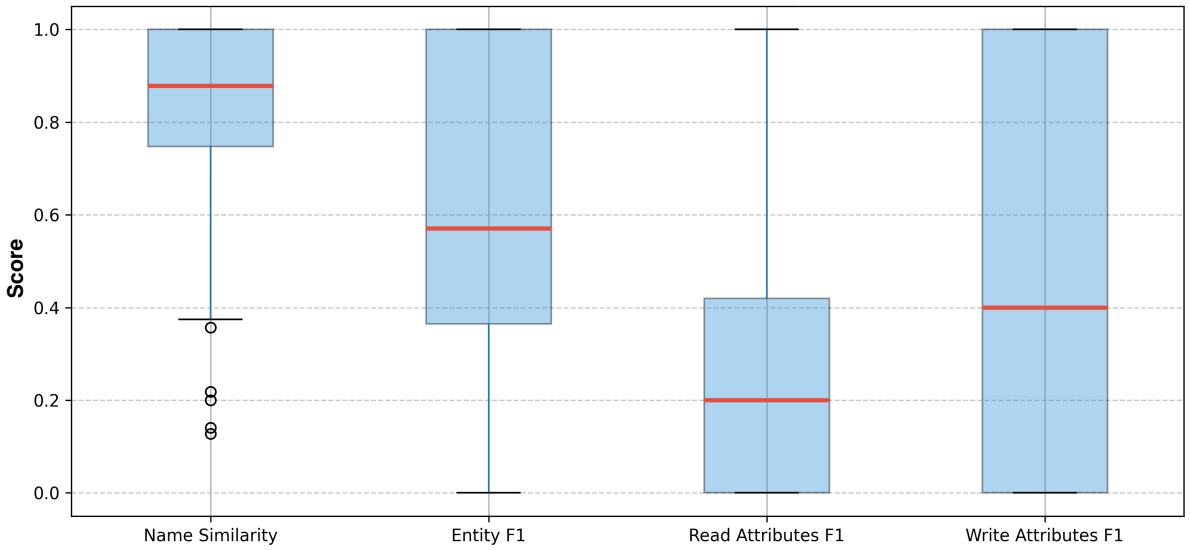

Figure 4.20: (Tutored) Box Plot for DeepSeek-Reasoner.

The distribution of the *Entity F1* shows significant variability, with values ranging from cases of null recognition to situations of complete entity coverage. The median is placed on intermediate values, in line with the aggregate mean value reported in Table 4.5, indicating an overall adequate ability of the model in identifying the relevant entities of the domain. The wide dispersion observed suggests that performance is strongly influenced by the clarity and level of detail with which entities are made explicit in textual requirements.

As regards the *Read Attributes F1*, the distribution shows marked variability, with a non-zero but still relatively small median and a significant presence of values close to zero. This trend indicates that the model is able to correctly identify the attributes read in a subset of cases, while in others it encounters difficulties attributable to the often implicit nature of data access patterns in user stories. The presence of higher values, however, suggests that when the requirements are sufficiently explicit, the model is also able to correctly represent the reading operations.

The *Write Attributes F1* has a broad distribution, characterized by an intermediate median and a significant dispersion of scores, consistent with an overall mean value close to 0.5. This behavior indicates that DeepSeek-Reasoner is able to correctly identify application state update operations in several cases, but that this capability is not uniform across the entire set of user stories, reflecting the inherent complexity of write operations in the Tutored domain.

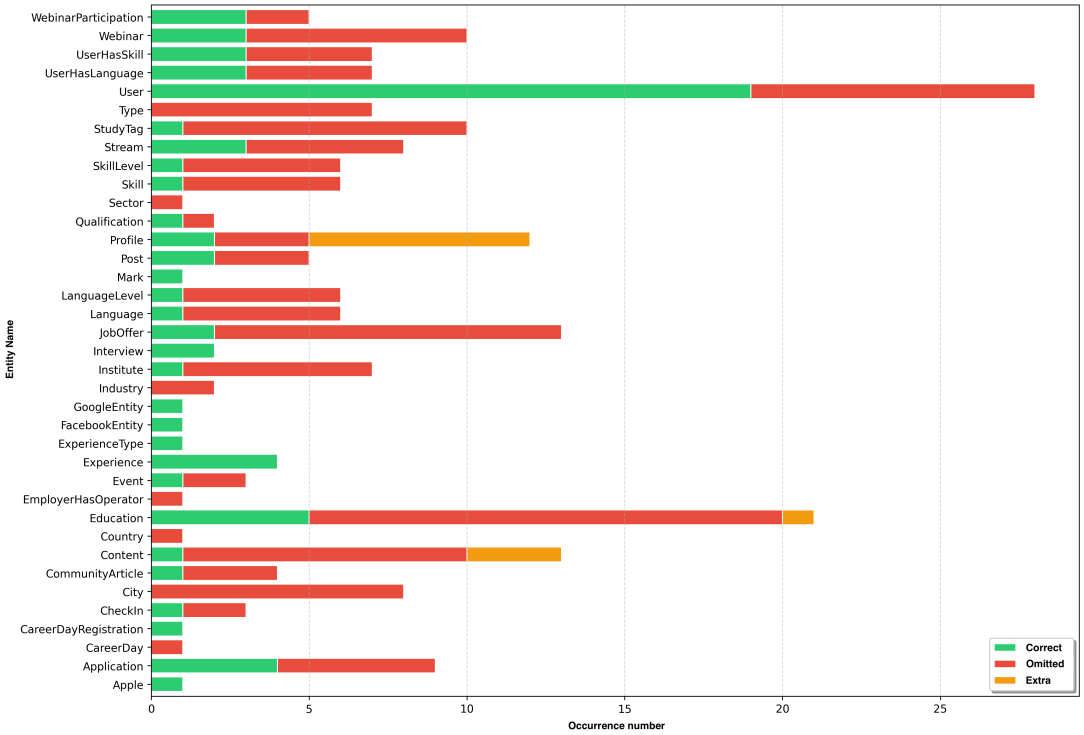

Figure 4.21: (Tutored) Entity Coverage for DeepSeek-Reasoner.

The entity coverage analysis, shown in Figure 4.21, provides a more granular view of the model's behavior at the single-entity level. The core domain entities, have a high number of correct recognitions, indicating a good ability of the model to capture the main elements of the system. Conversely, for numerous secondary or less frequently cited entities, a significant number of omissions are observed, highlighting a non-uniform coverage of the entire domain.

Mistral-Large-2512 Analysis

The box plot shown in Figure 4.22 shows the distribution of scores obtained by Mistral-Large-2512 on the main structural and semantic evaluation metrics. The *Name Similarity* metric has a high median, close to 0.9, indicating that the model is generally able to assign semantically consistent names to the generated operations. The distribution, however, highlights the presence of some outliers with lower values, suggesting that in the presence of less structured user stories or those characterized by lexical ambiguities, the model can produce denominations less aligned with the Ground Truth.

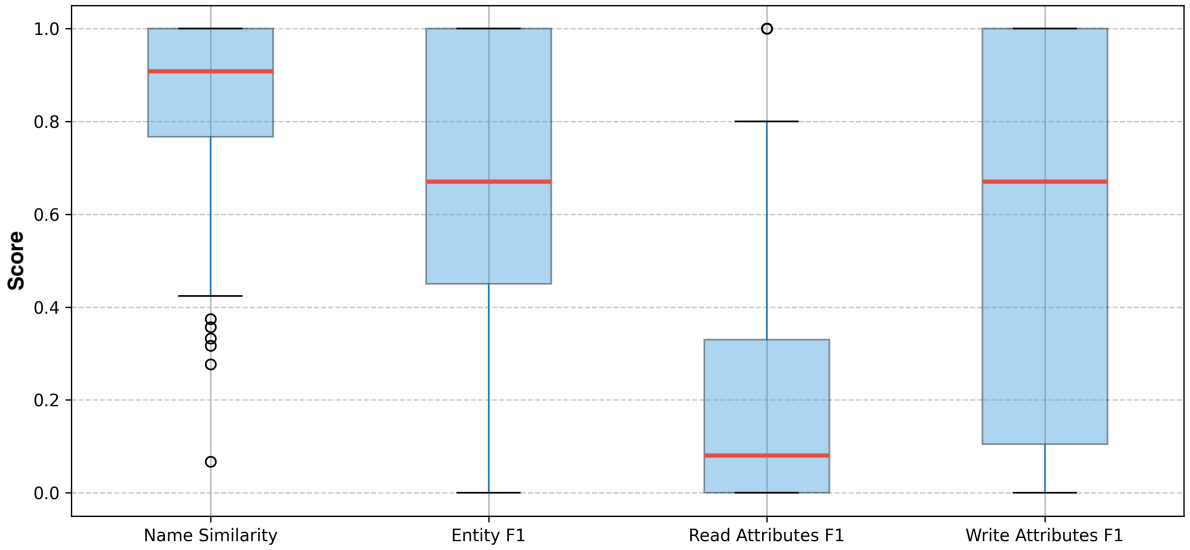

Figure 4.22: (Tutored) Box Plot for Mistral-2512-Large.

The distribution of the *Entity F1* shows significant variability, with values ranging from cases of partial recognition to situations of complete entity coverage. The median is placed on intermediate values, consistent with the aggregate mean value reported in Table 4.5, indicating an overall adequate ability to identify relevant entities in the domain. The wide dispersion of scores suggests that the effectiveness of the model in structural modeling depends significantly on the level of detail and clarity with which entities are made explicit in textual requirements.

Regarding the *Read Attributes F1*, the distribution is strongly skewed towards low values, with a median close to zero and a limited presence of high scores. This trend indicates that the model encounters significant difficulties in identifying attributes read by operations, particularly when data access patterns are not explicitly defined in user stories. The presence of some higher values, however, suggests that, in specific cases, the model is also able to correctly capture reading operations.

The *Write Attributes F1* has a wide distribution, characterized by an intermediate median and a significant dispersion of scores, consistent with an overall mean value close to 0.6. This result indicates that Mistral-Large-2512 is generally able to identify application state update operations, although this capability is not uniform across the entire set of user stories, reflecting the complexity of write operations in the Tutored domain.

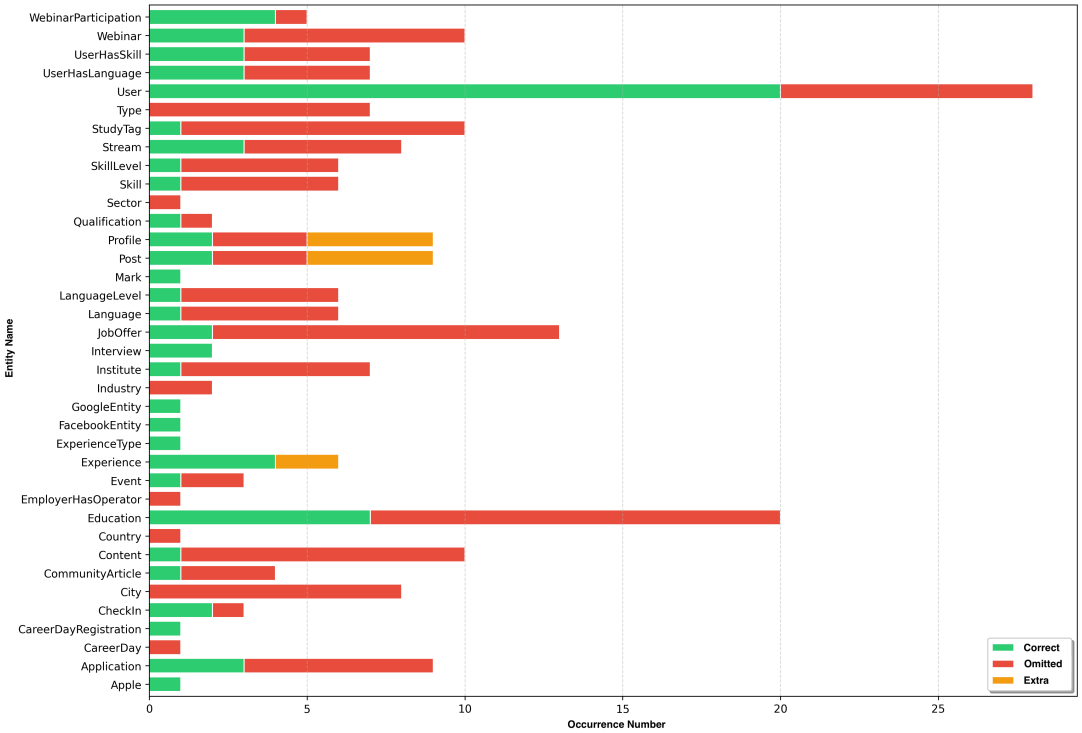

Figure 4.23: (Tutored) Entity Coverage for Mistral-2512-Large.

Entity coverage analysis, illustrated in Figure 4.23, provides a detailed view of the model's behavior at the single-entity level. Domain core entities, have a high number of correct recognitions, while numerous secondary entities show a significant rate of omissions. In some cases, the presence of incorrectly introduced entities is also observed, indicating episodes of over-generalization in the representation of the domain.

Gemini-2.5-Flash Analysis

The box plot shown in Figure 4.24 shows the distribution of scores obtained by Gemini-2.5-Flash on the main structural and semantic evaluation metrics. The *Name Similarity* metric has a high median, between 0.8 and 0.9, indicating that the model is generally able to assign semantically consistent names to the generated operations. The distribution, however, highlights the presence of some outliers with lower values, suggesting that in the presence of less structured user stories or those characterized by lexical ambiguity, the model can produce denominations less aligned with the Ground Truth.

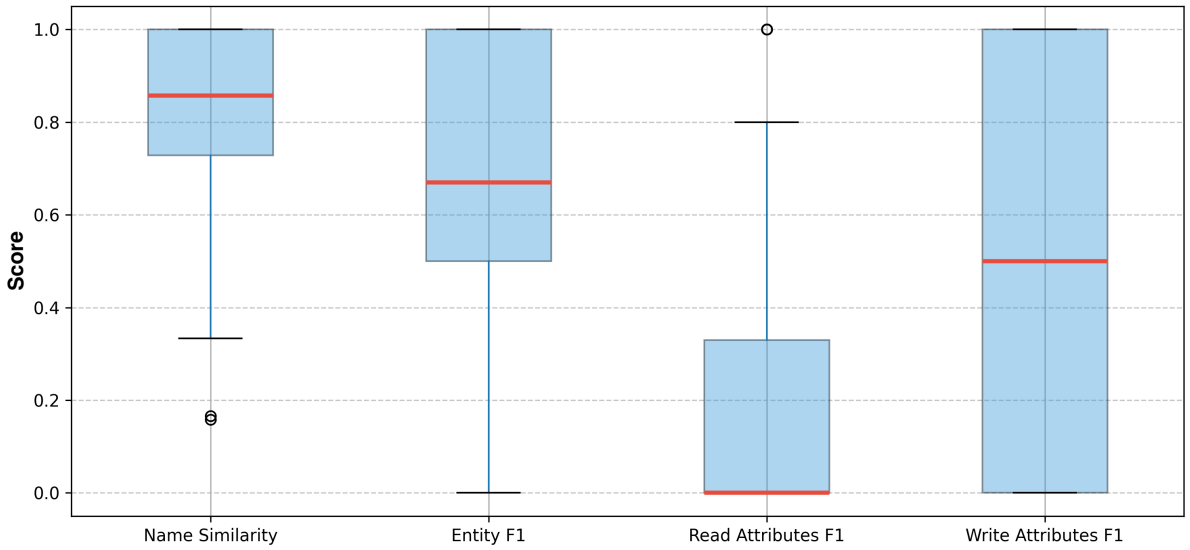

Figure 4.24: (Tutored) Box Plot for Gemini-2.5-Flash.

The distribution of the *Entity F1* shows significant variability, with scores ranging from values close to zero to cases of complete entity recognition. The median is placed on intermediate values, consistent with the aggregate mean value reported in Table 4.5, indicating an overall adequate ability to identify relevant entities in the domain. The wide dispersion of values suggests that model performance is strongly dependent on the clarity and level of detail with which entities are made explicit in textual requirements.

Regarding the *Read Attributes F1*, the distribution is strongly skewed towards low values, with a median close to zero and a limited presence of high scores. This trend indicates that Gemini-2.5-Flash encounters significant difficulties in identifying attributes read by operations, particularly when data access patterns are not explicitly defined in user stories. The presence of some higher values, however, suggests that, in specific cases, the model is also able to correctly capture reading operations.

The *Write Attributes F1* has a broad distribution, characterized by an intermediate median and a significant dispersion of scores, consistent with an overall mean value close to 0.5. This result indicates that the model is able to correctly identify application state update operations in a relevant subset of cases, although this capability is not uniform across the entire set of user stories, reflecting the inherent complexity of write operations in the Tutored domain

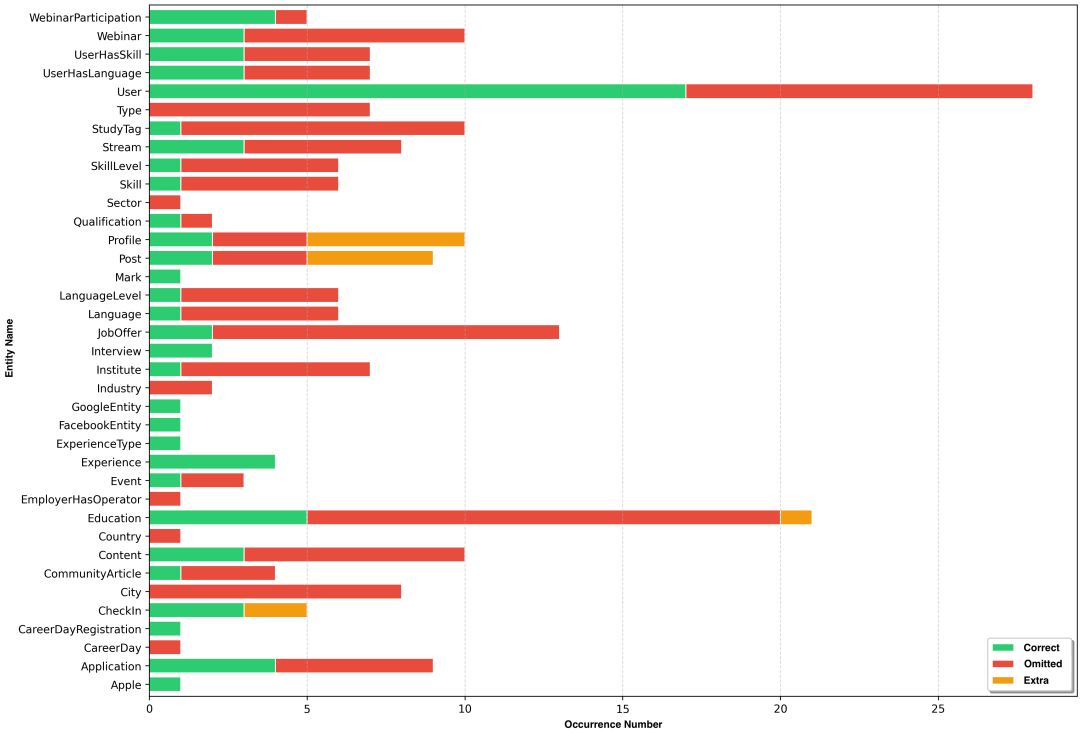

Figure 4.25: (Tutored) Entity Coverage for Gemini-2.5-Flash.

Entity coverage analysis, illustrated in Figure 4.25, provides a detailed view of the model's behavior at the single-entity level. The central entities of the domain have a high number of correct recognitions, while numerous secondary entities show a significant rate of omissions. In some cases, the presence of incorrectly introduced entities is also observed, indicating episodes of over-generalization in the representation of the domain.

In order to select the *best generated* architecture for the *Tutored* case study, a set of criteria oriented to the overall quality of the produced specification was considered, including (i) domain entity coverage, (ii) semantic consistency in denominations, and (iii) the ability to represent application operations in a structurally consistent way. In particular, in addition to the aggregate mean values reported in Table 4.5, the variability of the metrics at the single user story level and the percentity coverage highlighted by the *entity coverage* graphs were considered.

Based on these considerations, the architecture generated by **DeepSeek-Chat** was selected as *best generated*. The model in fact shows a good compromise between structural correctness and semantic consistency, highlighted by an average value of Name Similarity

equal to 0.85 and a Write F1 close to 0.6, compared to an adequate coverage of the main entities of the domain (Figure 4.19). Although modeling read accesses is complex for all models and has room for improvement, DeepSeek-Chat produces an overall consistent and interpretable specification, making it a suitable basis for subsequent analyses and for evaluating the impact of Forced Operations in the next phase.

4.2.2. Forced Operations Impact on Architecture Evaluation

This section analyzes the impact of Forced Operations on the architectural properties evaluated using the Cromlech framework, considering both the *Cromlech Baseline Architecture* and the *Best Generated Architecture* for the Tutored case study. The goal is to evaluate how introducing guided changes to the architectural decomposition affects quality metrics, isolating the effect of Forced Operations with respect to the initial architecture configuration.

All evaluations reported in this section were obtained by running the Cromlech framework with a configuration consistent with the experimental scenario defined for the domain Tutored [1], in order to ensure comparability of the results and adherence to the adopted experimental workflow. In particular, the following were used for all executions:

- un valore del parametro di bilanciamento $\alpha = 0.5$;
- un numero massimo di servizi pari a 7;
- un limite di tempo di esecuzione pari a 1 ora per ciascuna valutazione.

Within this section, the analysis is divided into two main subsections: the first dedicated to the impact of Forced Operations on Cromlech Baseline Architecture, and the second focused on Best Generated Architecture. For each case, the results are analyzed separately in deterministic ($T = 0.0$) and stochastic ($T = 1.0$) configurations, combining the analysis of Cromlech metrics with a qualitative comparison of the resulting architectural decompositions.

Baseline Comparison without Forced Operations

In this subsection, the comparison between the two architectures is analyzed in the case where Forced Operations are not applied. The aim is to evaluate the intrinsic differences between the reference architecture and the one generated by the selected model, considering only the initial output of the architectural synthesis process and the optimization performed by the Cromlech framework in the absence of additional constraints.

Model	**Cohesion**	**Communication Cost**	**Total Value**	**# Services**
Cromlech Baseline	0.4864	0.0855	0.2005	7
Best Generated	0.4843	0.0	0.2421	7

Table 4.6: (Tutored) Comparison between Cromlech Baseline and Best Generated architecture without Forced Operations.

The quantitative results of the comparison are reported in Table 4.6. Both architectures are characterized by the same number of services, equal to 7, allowing a direct comparison of quality metrics without the influence of different levels of granularity of the decomposition.

Regarding the metric, the values obtained from the two architectures are substantially aligned, indicating a comparable distribution of operations within services from the point of view of internal cohesion. This result suggests that, in the absence of Forced Operations, the basic structure of the decomposition generated by the model maintains a level of functional consistency similar to that of the reference architecture.

A more marked difference emerges instead in the metric of *Communication Cost*. The Best Generated Architecture has a null value, indicating the absence of inter-service dependencies according to the evaluation model adopted. On the contrary, the Cromlech Baseline Architecture shows a non-zero, albeit low, communication cost, highlighting the presence of interactions between services that contribute to increasing communication complexity.

This difference is reflected in the *Total Value* metric, which is higher for the Best Generated Architecture. This result indicates that, given the same number of services and level of cohesion, reducing communication costs contributes to improving the overall quality of the architecture generated by the model.

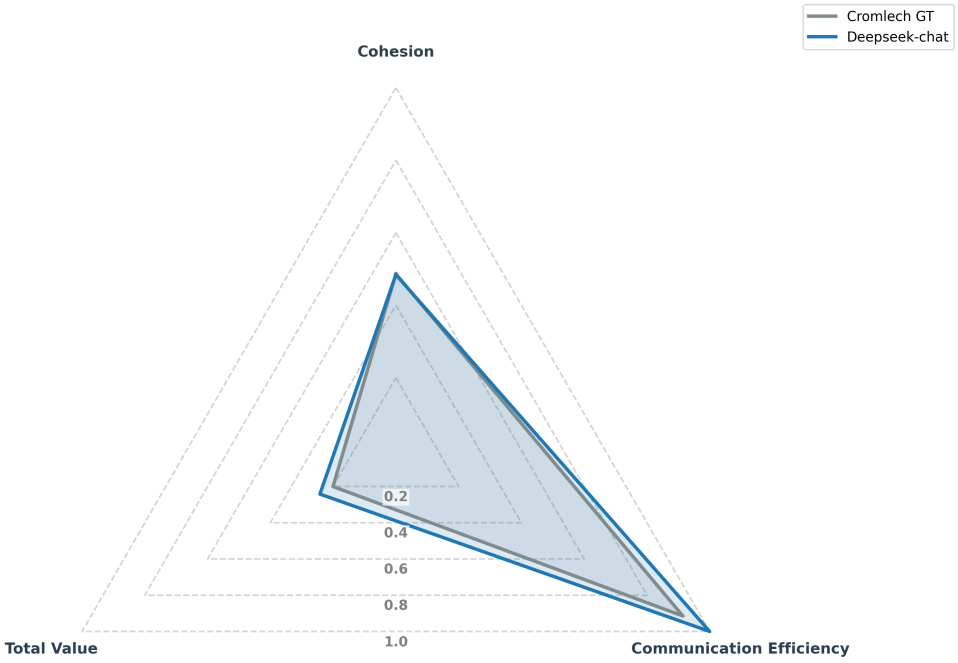

Figure 4.26: (Tutored) Radar chart comparison between Cromlech Baseline and Best Generated architecture without Forced Operations.

The comparison is further summarized in the radar chart in Figure 4.26, which visually highlights the profile of the two architectures on the main Cromlech metrics. In particular, the graph highlights how the Best Generated Architecture achieves an advantage mainly related to communication efficiency, while differences in cohesion are marginal.

Overall, this analysis shows that, in the absence of Forced Operations, the Best Generated Architecture represents a valid alternative to the Cromlech Baseline Architecture, ensuring a better trade-off between cohesion and communication cost and providing a solid basis for analyzing the impact of Forced Operations in subsequent sections.

Impact of Forced Operations on: Cromlech Baseline Architecture

This section analyzes the impact of Forced Operations integration with the aim of evaluating how the logical constraints suggested by LLMs influence the metrics and the final number of microservices identified, comparing them with the Cromlech Baseline. The latter represents the reference configuration without external constraints, operating exclusively

on the optimization of data access patterns.

Model	T	Cohesion	Comm. Cost	Total Value	# Services
Baseline (No FO)	-	0.4864	0.0855	0.2005	7
DeepSeek-Chat	0.0	0.4197	0.0393	0.1902	7
DeepSeek-Chat	1.0	0.3802 ± 0.0033	0.0351 ± 0.0005	0.1725 ± 0.0006	6.8 ± 0.4 (6-7)
DeepSeek-Reasoner	0.0	0.4075	0.0448	0.1814	7
DeepSeek-Reasoner	1.0	0.3458 ± 0.0040	0.0564 ± 0.0012	0.1447 ± 0.0007	4.8 ± 1.33 (3-5)
Gemini-2.5-Flash	0.0	0.2047	0.0	0.1024	2
Gemini-2.5-Flash	1.0	0.2110 ± 0.0018	0.0141 ± 0.0002	0.0984 ± 0.0005	2.2 ± 0.75 (1-3)
Mistral-Large-2512	0.0	0.4064	0.0231	0.1917	7
Mistral-Large-2512	1.0	0.3755 ± 0.0004	0.0461 ± 0.0004	0.1647 ± 0.0003	7.0 ± 0.0 (7-7)

Table 4.7: (Tutored) Cromlech evaluation metrics (mean ± standard deviation) for Cromlech Baseline architecture with Forced Operations

Table 4.7 summarizes the results obtained for the Tutored case study. The data correlate the deterministic configuration ($T = 0.0$) with the stochastic one ($T = 1.0$). For the latter, the results are reported in terms of mean ± standard deviation over 5 iterations, including the range of the number of services generated in brackets. This representation allows us to observe not only the point performance, but also the stability of the partitioning process under the influence of forced constraints.

Deterministic setting (T=0.0) In the deterministic setting, applying Forced Operations to the Cromlech Baseline Architecture for the Tutored case study has a significant impact on evaluation metrics, highlighting a trade-off between structural cohesion and communication efficiency compared to the unconstrained configuration. In particular, the baseline without Forced Operations has a Cohesion value of 0.4864 and a Communication Cost of 0.0855, as reported in Table 4.7.

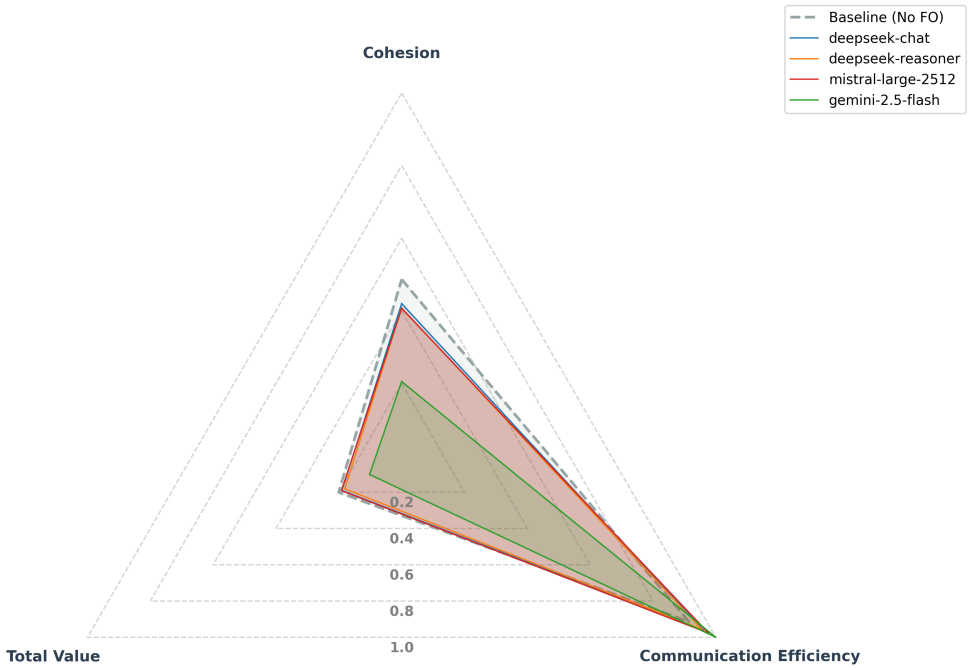

Figure 4.27: (Tutored) Radar chart of Cromlech evaluation metrics for the Baseline architecture with Forced Operations at T=0.0.

The Forced Operations generated by DeepSeek-Chat, DeepSeek-Reasoner and Mistral-Large-2512 lead to architectural configurations characterized by Cohesion values in the range [0.40, 0.42], compared to a reduction in Communication Cost, which stands at values lower than 0.05. In all three cases, the number of services remains unchanged from the baseline, equal to 7, indicating that the introduced constraints do not change the granularity of the decomposition, but mainly affect the redistribution of operations.

A distinct behavior is observed for Gemini-2.5-Flash, which in the deterministic setting generates a significantly more compact decomposition, reducing the number of services to 2. This structural choice is reflected in a Cohesion value of 0.2047, compared to a zero Communication Cost. Overall, this configuration leads to a significantly lower Total Value than the other solutions, as highlighted both in Table 4.7 and in the radar chart in Figure 4.27.

Overall, the deterministic setting highlights how, in the Tutored domain, the introduction of Forced Operations does not lead to convergence towards a single architectural solution, but produces different decompositions depending on the model considered. This result suggests that, in the absence of stochasticity, the effect of forced constraints is strongly dependent on the nature and structure of the indications provided by individual LLMs.

Stochastic setting (T=1.0) In the stochastic setting, each LLM was used to generate 5 independent instances of Forced Operations from the same Cromlech Baseline Architecture. The results, reported in Table 4.7 as mean ± standard deviation, allow to jointly evaluate the average performance, the stability of the solutions and the structural differences introduced by the Forced Operations in terms of architectural decomposition.

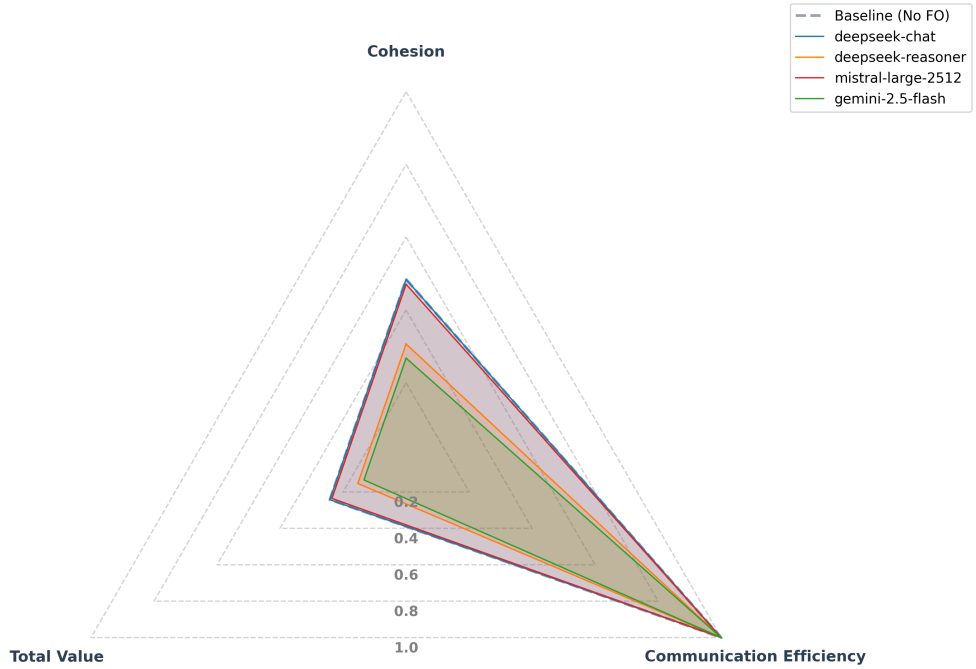

Figure 4.28: (Tutored) Radar chart of Cromlech evaluation metrics for the Baseline architecture with Forced Operations at T=1.0.

The radar chart in Figure 4.28 provides a summary of the average performance of the models, showing differentiated profiles on the Cohesion, Communication Cost and Total Value metrics. However, as in the case of Trainticket, this aggregate representation does not allow us to fully capture the variability introduced by the stochastic component nor the differences in the structure of the resulting decompositions.

These aspects emerge most clearly from the analysis of the box plot in Figure 4.13, which highlights the dispersion of the metrics and their relationship with the number of services generated.

DeepSeek-Chat shows limited structural variability, with an average number of services of 6.8 ± 0.4 (range 6–7). This limited dispersion is reflected in compact metric distributions, with Cohesion values of 0.3802 ± 0.0033 and Total Value values of 0.1725 ± 0.0006, indicating relatively stable behavior of the partitioning process under the effect of Forced Operations.

More variable behavior is observed for *DeepSeek-Reasoner*, which generates a number of services equal to 4.8 ± 1.33, with a range between 3 and 5. This structural variability is clearly visible in the box plot and is accompanied by a greater dispersion of metrics, particularly for Cohesion (0.3458 ± 0.0040) and Total Value (0.1447 ± 0.0007). This result suggests that, in the Tutored domain, Forced Operations produced by DeepSeek-Reasoner can induce significantly different decompositions depending on the instance generated.

Gemini-2.5-Flash has a very compact average decomposition, with a number of services equal to 2.2 ± 0.75 (range 1–3). The observed variability in the number of services is reflected in a moderate dispersion of the metrics, with Cohesion values of 0.2110 ± 0.0018 and a low average Communication Cost. However, the low level of structural cohesion helps to keep Total Value at lower values overall than the other models.

Finally, *Mistral-Large-2512* shows completely stable behavior in the stochastic setting: all runs produce an identical decomposition, with 7 overlapping Cromlech services and metrics. This absence of variability, visible in the box plot as a collapse of the distribution on a single line, indicates that the effect of stochasticity is negligible for this model in the context of Cromlech Baseline Architecture.

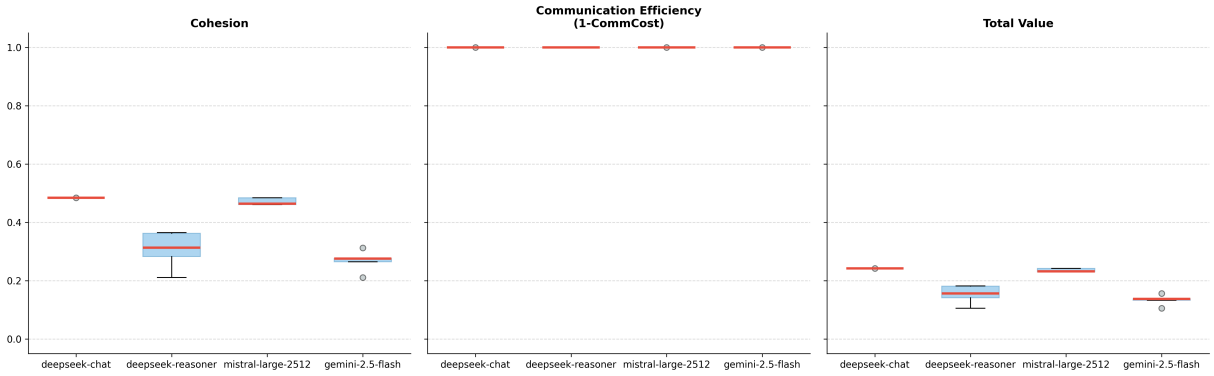

Figure 4.29: (Tutored) Box plot of Cromlech evaluation metrics for the Baseline architecture with Forced Operations at T=1.0.

Overall, while the radar chart provides a concise view of average performance, the joint analysis of the box plot and the number of services highlights substantial differences between the models in terms of stability and architectural decomposition strategies. In particular, stochastic setting highlights how, in the Tutored domain, the introduction of Forced Operations can lead to structurally different solutions even starting from the same basic architecture, depending on the model considered.

Impact of Forced Operations on: Best Generated Architecture

This subsection analyzes the impact of Forced Operations applied to Best Generated Architecture, i.e., the architecture obtained from the automatic generation process. Unlike the previous case, where Forced Operations act on a manually designed baseline configuration, in this scenario they intervene on an architecture already optimized according to the generation framework criteria.

Model	T	Cohesion	Comm. Cost	Total Value	# Services
Best Generated (No FO)	-	0.4843	0.0	0.2421	7
DeepSeek-Chat	0.0	0.4843	0.0	0.2421	7
DeepSeek-Chat	1.0	0.4843 ± 0.0	0.0 ± 0.0	0.2421 ± 0.0	7.0 ± 0.0 (7-7)
DeepSeek-Reasoner	0.0	0.4641	0.0	0.2320	7
DeepSeek-Reasoner	1.0	0.3066 ± 0.0041	0.0 ± 0.0	0.1533 ± 0.0010	4.8 ± 1.17 (3-6)
Gemini-2.5-Flash	0.0	0.2047	0.0	0.103	4
Gemini-2.5-Flash	1.0	0.2678 ± 0.0013	0.0 ± 0.0	0.1339 ± 0.0003	4.0 ± 0.63 (3-5)
Mistral-Large-2512	0.0	0.4221	0.0	0.2111	6
Mistral-Large-2512	1.0	0.4707 ± 0.0002	0.0 ± 0.0	0.2354 ± 0.0	7.0 ± 0.0 (7-7)

Table 4.8: (Tutored) Cromlech evaluation metrics (mean ± standard deviation) for Best Generated architecture with Forced Operations

Table 4.8 reports the results obtained considering both the Forced Operations-free configuration and the architectures resulting from the application of Forced Operations in deterministic ($T = 0$) and stochastic ($T = 0$) configurations. Again, for stochastic setting the results are expressed as mean ± standard deviation over 5 independent runs.

Deterministic setting (T=0.0) In the deterministic setting, applying Forced Operations to the Best Generated Architecture produces an overall limited impact on evaluation metrics, highlighting how the starting architecture is already well aligned with the optimization objectives of the Cromlech framework.

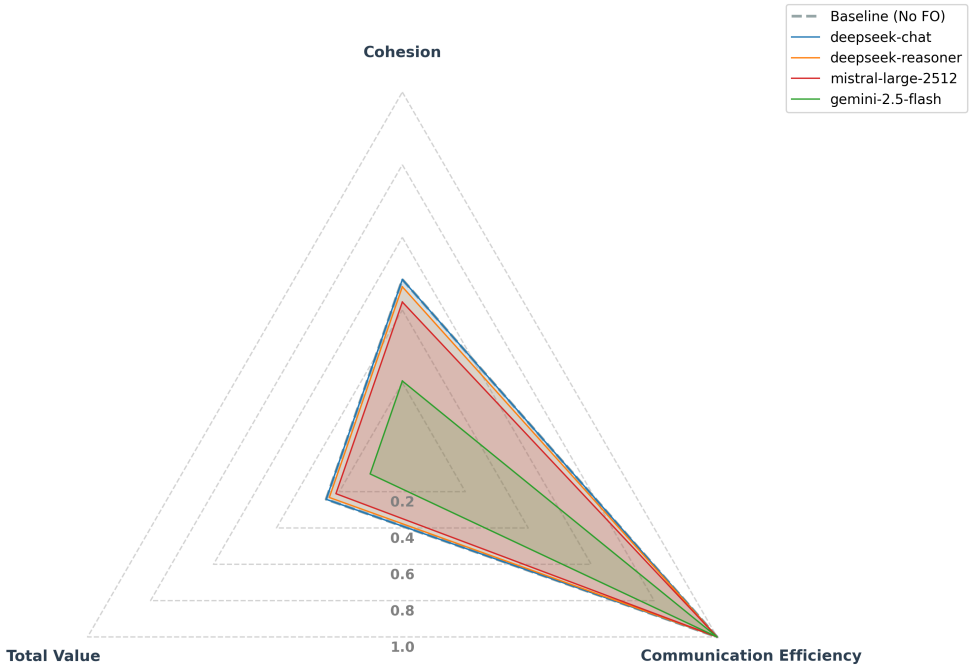

Figure 4.30: (Tutored) Radar chart of Cromlech evaluation metrics for the Best Generated architecture with Forced Operations at T=0.0.

As reported in Table 4.8, the Forced Operations generated by *DeepSeek-Chat* do not introduce any changes compared to the unconstrained configuration: the Cromlech metrics and the number of services remain unchanged, with a Total Value value of 0.2421. This behavior is clearly visible in the radar chart in Figure 4.30, where the DeepSeek-Chat profile completely overlaps with that of the Best Generated Architecture without Forced Operations.

For *DeepSeek-Reasoner* and *Mistral-Large-2512*, however, smaller variations are observed, associated with slight structural reorganizations. In both cases, these changes result in a reduction in overall cohesion compared to the starting configuration, as evidenced by the contraction of the polygon along the Cohesion axis in the radar chart. This suggests that the introduced constraints are not fully compatible with an already optimized structure.

A distinct behavior is observed for *Gemini-2.5-Flash*, which in the deterministic setting produces a significantly more compact decomposition, with a number of services equal to 4. This structural reduction is reflected in a marked decline in cohesion and overall value, as clearly shown in the radar chart, confirming that an excessively consolidated decomposition penalizes architectural quality in the Tutored domain.

Overall, the deterministic setting highlights how, when Forced Operations are applied to an already optimized architecture, their contribution strongly depends on the consistency

between the constraints suggested by the model and the starting structure. In this context, DeepSeek-Chat emerges as the only model capable of fully preserving the quality of the Best Generated Architecture, without introducing unwanted side effects.

Stochastic setting (T=1.0) In the stochastic setting, each LLM was used to generate 5 independent instances of Forced Operations from the same Best Generated Architecture. The results, reported in Table 4.8 as mean ± standard deviation, allow us to jointly evaluate the average performance, the stability of the solutions and the impact of stochasticity on the resulting architectural structure.

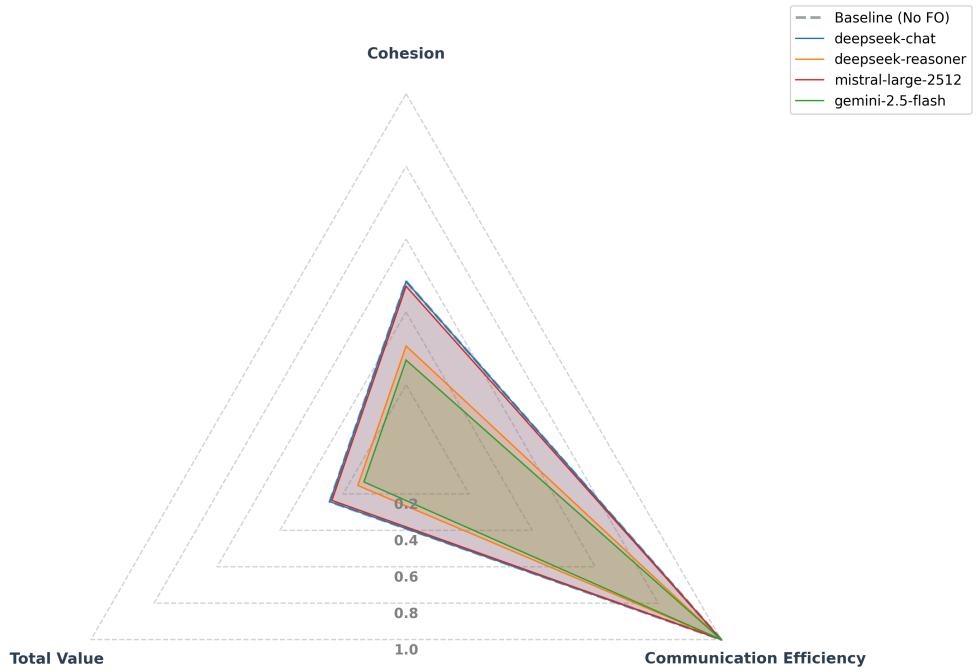

Figure 4.31: (Tutored) Radar chart of Cromlech evaluation metrics for the Best Generated architecture with Forced Operations at T=1.0.

The radar chart in Figure 4.31 provides a summary of the average performance of the models, showing profiles differentiated mainly along the Cohesion and Total Value dimensions, while Communication Efficiency remains consistently maximum for all configurations. In particular, DeepSeek-Chat perfectly overlaps with the profile of the Best Generated Architecture without Forced Operations, keeping the value of Total Value unchanged at 0.2421. This lack of variation is also evident from the analysis of the box plot in Figure 4.32, which allows us to observe the dispersion of the metrics and their relationship with the number of services generated.

DeepSeek-Chat indeed shows completely stable behavior in the stochastic setting: all runs

produce an identical decomposition, with 7 overlapping Cromlech services and metrics. In the box plot, this stability manifests itself as a collapse of distributions on single lines, indicating that the introduction of Forced Operations does not alter an already optimized structure.

Significantly different behavior is observed for *DeepSeek-Reasoner*, which introduces marked structural variability, with the number of services ranging from 3 to 6. This variability is reflected in a reduction in the average Total Value value, which stands at around 0.15, as visible both in the radar chart and in the dispersion observed in the box plot, indicating a negative impact of Forced Operations on the overall architectural quality.

Gemini-2.5-Flash has a moderately compact decomposition, with a number of services close to 4 and moderate variability between different runs. Despite a slight increase in mean values compared to the deterministic case, Total Value remains low, at approximately $0,134$, suggesting a penalty due to an excessively consolidated structure.

Finally, *Mistral-Large-2512* shows completely stable behavior even in the stochastic setting: all runs converge to the same decomposition with 7 services and an average Total Value of 0.2354. Similar to DeepSeek-Chat, the effect of stochasticity is negligible, indicating good robustness of the model to the application of Forced Operations on the Best Generated Architecture.

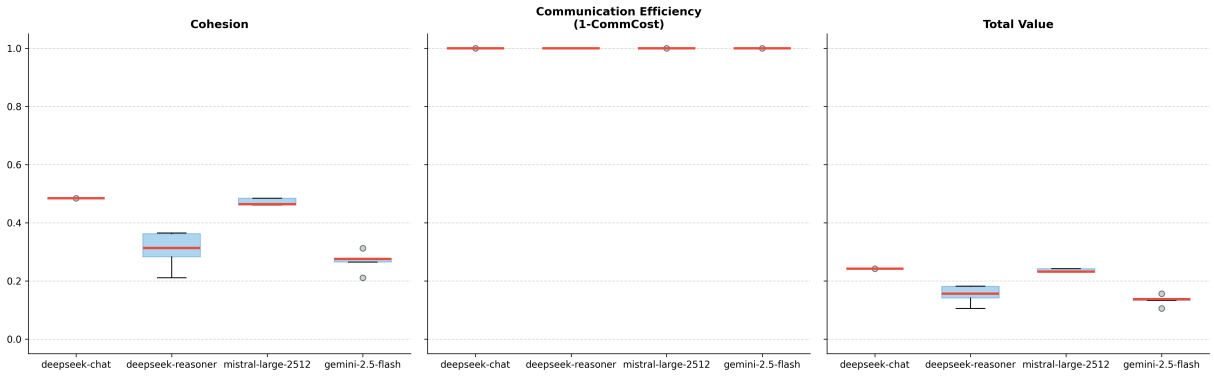

Figure 4.32: (Tutored) Box plot of Cromlech evaluation metrics for the Best Generated architecture with Forced Operations at T=1.0.

Overall, while the radar chart provides a concise view of average performance, the joint analysis of the box plot and the number of services highlights substantial differences between the models in terms of stability and architectural decomposition strategies. In particular, in the case of Best Generated Architecture, stochastic setting shows how Forced Operations can have no, moderate or highly degrading impact depending on the model considered.

5 | Conclusions and Future Work

This thesis addressed the problem of translating informal requirements into formal architectural models required by optimization frameworks for decomposing monolithic systems into microservices. In particular, the work focused on the so-called *Input Gap*, i.e. the difficulty of deriving structured specifications from non-formalized textual artifacts, such as user stories.

The proposed approach integrates Large Language Models (LLMs) into a structured pipeline aimed at automatically generating and refining architectural specifications in `YAML` format. The workflow allows you to extract entities, operations, data access patterns, and transactional constraints directly from natural language requirements, while also introducing the concept of Forced Operations to model colocation constraints inferred from language models.

The experimental evaluation, conducted on the Trainticket and Tutored case studies, allowed us to systematically analyze the impact of the specifications generated by the LLMs on the quality of the resulting architectures. The results highlight some relevant aspects.

First, LLMs proved capable of generating syntactically valid structured specifications that were semantically consistent enough to be processed by the Cromlech optimization engine. This confirms the feasibility of using generative models as support in the initial stages of architectural modeling.

Second, the introduction of Forced Operations significantly impacts optimization results. In some scenarios, additional constraints improve cohesion and guide the solver toward architectural decompositions that adhere more closely to domain logic; in other cases, however, they narrow the space of possible solutions, introducing tradeoffs in terms of operating cost.

Third, the comparison between different model families highlighted significant differences in terms of stability and structural consistency. Reasoning-oriented models show greater consistency in constraint generation in some contexts, while efficiency-optimized models

show greater sensitivity to stochastic decoding parameters. The analysis in the deterministic and stochastic settings also highlighted the central role of temperature in balancing reproducibility and exploratory variability.

Overall, the results suggest that LLMs can provide valuable support in the preliminary stages of microservices architecture design, provided that their outputs are placed in a controlled and verifiable context, such as a formal optimization framework. The proposed integration does not replace the role of the software architect, but helps reduce the cognitive load associated with formalizing requirements.

Despite the contributions presented, the work highlights some limitations. The evaluation was conducted on two case studies and relies exclusively on user stories as an information source, without integrating static code analysis or execution traces. Furthermore, the generation of the `YAML` specification depends on the prompt design and does not use grammar-constrained decoding techniques, which could impact robustness in more complex industrial contexts. Finally, the process does not involve an explicit iterative human validation mechanism.

Among the possible evolutions of the presented work, a particularly promising direction concerns the adoption of a multi-agent architecture. In this configuration, several specialized models could collaborate in a coordinated manner within the pipeline: a first agent dedicated to structured specification extraction, a second focused on architectural constraint inference, and a third responsible for consistency verification and formal validation of the generated output.

Such an approach would allow for the separation of different cognitive responsibilities, reducing the risk of structural hallucinations and improving the overall robustness of the process. Furthermore, agent interaction could introduce iterative self-verification and review mechanisms, further bringing the proposed automation closer to a more controlled and reliable architectural design process.

In conclusion, the thesis demonstrates that the controlled integration of Large Language Models with architectural optimization frameworks represents a promising direction for automated support for microservices decomposition.

Bibliography

[1] Giovanni Quattrocchi, Davide Cocco, Simone Staffa, Alessandro Margara, and Gianpaolo Cugola. Cromlech: Semi-automated monolith decomposition into microservices. *IEEE Transactions on Services Computing*, 2024.

[2] Len Bass, Paul C. Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, 4 edition, 2021.

[3] Mark Richards and Neal Ford. *Fundamentals of Software Architecture: An Engineering Approach*. O'Reilly Media, 2020.

[4] Sam Newman. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, 2021.

[5] Rodrigo Laigner, Yongluan Zhou, Marcos Antonio Vaz Salles, Yijian Liu, and Marcos Kalinowski. Data management in microservices: State of the practice, challenges, and research directions. *Proceedings of the VLDB Endowment*, 2021.

[6] Brian Foote and Joseph Yoder. Big ball of mud. In *Conference on Pattern Languages of Programs (PLoP)*, 1997.

[7] Jacopo Soldani, Damian Andrew Tamburri, and Willem-Jan Van Den Heuvel. The pains and gains of microservices: A systematic grey literature review. *Journal of Systems and Software*, 2018.

[8] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. *IEEE Access*, 2017.

[9] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley, 2010.

[10] Martin Fowler and James Lewis. Microservices: a definition of this new architectural term, 2014. URL `https://martinfowler.com/articles/microservices.html`.

[11] Chris Richardson. *Microservices Patterns: With Examples in Java*. Manning, 2018.

[12] Festim Halili, Anila Nuhiji, and Diellza Mustafai Veliu. Polyglot persistence in microservices: Managing data diversity in distributed systems, 2025. URL `https://arxiv.org/abs/2509.08014`.

[13] Martin Kleppmann. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly Media, 2017.

[14] Opentelemetry documentation and specification. Online, 2025. URL `https://opentelemetry.io/docs/`.

[15] Zakarya Abdullah Alzamil. Software coupling and cohesion model for measuring the quality of software components. *Computers, Materials & Continua*, 2023.

[16] M. G. Moreira et al. Analysis of microservice evolution using cohesion metrics. *ACM Transactions on Software Engineering and Methodology*, 2021.

[17] Sebastiano Panichella, Mohammad Imranur Rahman, and Davide Taibi. Structural coupling for microservices. In *International Conference on Cloud Computing and Services Science (CLOSER)*, 2021.

[18] Genc Mazlami, Jens Cito, and Philipp Leitner. Extraction of microservices from monolithic software architectures. In *IEEE International Conference on Web Services (ICWS)*, 2017.

[19] Ying Jin, Shuai Wang, and Alessandro Armando. Towards automated microservices extraction using static analysis. *Journal of Systems and Software*, 2019.

[20] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2003.

[21] Lukas Gysel, Lukas Kölbener, Wolfgang Giersche, and Olaf Sorg. Service cutter: A systematic approach to service decomposition. In *European Conference on Service-Oriented and Cloud Computing*, pages 185–200. Springer, 2016.

[22] Brian S. Mitchell and Spiros Mancoridis. On the automatic modularization of software systems using the bunch tool. *IEEE Transactions on Software Engineering*, 2006.

[23] Anas Shatnawi and Abdelhak-Djamel Seriai. Identifying microservices using dynamic analysis. In *ICSME*, 2018.

[24] Shriti Kalia et al. Mono2micro: A practical and effective tool for decomposing monolithic java applications to microservices. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1214–1224, 2021.

[25] Seda Eski and Fevzi Belli. A systematic approach to microservices architecture migration. *Information and Software Technology*, 2018.

[26] Ghada Boussaidi. *Optimization Models in Software Engineering*. Springer, 2014.

[27] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. Architectural patterns for microservices: A systematic mapping study. *Journal of Systems and Software*, 2019.

[28] Gurobi Optimization, LLC. *Gurobi Optimizer Reference Manual*, 2023. URL `https://www.gurobi.com`.

[29] Fabiano Dalpiaz and Alessio Ferrari. Natural language processing for requirements engineering: The best is yet to come. *IEEE Software*, 2018.

[30] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, 2017.

[31] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 1997.

[32] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large language models for software engineering: A systematic literature review, 2024. URL `https://arxiv.org/abs/2308.10620`.

[33] Jerry Tworek Mark Chen et al. Evaluating large language models trained on code, 2021. URL `https://arxiv.org/abs/2107.03374`.

[34] Mohammad Amin Zadenoori, Jacek Dąbrowski, Waad Alhoshan, Liping Zhao, and Alessio Ferrari. Large language models (llms) for requirements engineering (re): A systematic literature review, 2025. URL `https://arxiv.org/abs/2509.11446`.

[35] Jianzhang Zhang, Yiyang Chen, Nan Niu, Yinglin Wang, and Chuang Liu. Empirical evaluation of chatgpt on requirements information retrieval under zero-shot setting, 2023. URL `https://arxiv.org/abs/2304.12562`.

[36] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C. Schmidt. A prompt pattern catalog to enhance prompt engineering with chatgpt, 2023. URL `https://arxiv.org/abs/2302.11382`.

[37] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. *Advances in Neural Information Processing Systems*, 2022.

[38] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, 2020.

[39] Jason Wei, Xuezhi Wang, Dale Schuurmans, Bosma Maarten, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 2022.

[40] Saibo Geng et al. Grammar-constrained decoding for structured text generation. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*, 2023.

[41] Xukun Liu, Zhiyuan Peng, Xiaoyuan Yi, Xing Xie, Lirong Xiang, Yuchen Liu, and Dongkuan Xu. Toolnet: Connecting large language models with massive tools via tool graph, 2024. URL `https://arxiv.org/abs/2403.00839`.

[42] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network, 2015. URL `https://arxiv.org/abs/1503.02531`.

[43] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration, 2020. URL `https://arxiv.org/abs/1904.09751`.

[44] Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q. Weinberger. On calibration of modern neural networks. In *ICML*, 2017.

[45] Daniel Kahneman. *Thinking, Fast and Slow*. Farrar, Straus and Giroux, 2011.

[46] Duzhen Zhang Zhong-Zhi Li et al. From system 1 to system 2: A survey of reasoning large language models, 2025. URL `https://arxiv.org/abs/2502.17419`.

[47] Alireza S. Ziabari, Nona Ghazizadeh, Zhivar Sourati, Farzan Karimi-Malekabadi, Payam Piray, and Morteza Dehghani. Reasoning on a spectrum: Aligning llms to system 1 and system 2 thinking, 2025. URL `https://arxiv.org/abs/2502.12470`.

[48] Paolo Di Francesco, Ivano Malavolta, and Patricia Lago. A systematic mapping study on microservice architectures. *Journal of Systems and Software*, 2020.

[49] Mike Cohn. *User Stories Applied: For Agile Software Development*. Addison-Wesley, 2004.

[50] Gian Wiher, Clara Meister, and Ryan Cotterell. On decoding strategies for neural text generators. *Transactions of the Association for Computational Linguistics*, 2022.

[51] DeepSeek-AI, Aixin Liu, et al. Deepseek-v3 technical report, 2025. URL https://arxiv.org/abs/2412.19437.

[52] Daya Guo et al. Deepseek-r1 incentivizes reasoning in llms through reinforcement learning. *Nature*, 645(8081):633–638, September 2025. ISSN 1476-4687. doi: 10.1038/s41586-025-09422-z. URL http://dx.doi.org/10.1038/s41586-025-09422-z.

[53] Mistral AI. Mistral large 3: The next frontier in frontier models, 2025. URL https://mistral.ai/news/mistral-large-2512/.

[54] Google DeepMind. Gemini 2.5: High-performance long-context models, 2025. URL https://deepmind.google/technologies/gemini/.

[55] Tor Thorsrud Sporsem, Torgeir Dingsøyr, and Klaas-Jan Stol. User stories as boundary objects in agile requirements engineering: A theoretical literature review. *Journal of Systems and Software*, 2025. URL https://doi.org/10.1016/j.jss.2025.112693.

A | Appendix

A.1. Prompt per la generazione del file YAML di partenza

```
Role: You are a Software Architect Assistant specializing in generating
    Cromlech-compatible YAML files for microservices decomposition.

Task: Update the provided YAML specification by populating the co-location
    constraints within each operation.

Fundamental hard constraints:
1. CONSISTENCY RULE: If write_attributes is not empty for any entity,
    consistency MUST be 'H'.
2. UNIQUE NAME MANDATE: Every operation name must be unique.
3. MAPPING: Exactly one operation per User Story.
4. ABSOLUTE DISJOINT RULE: In any single database_access block for an entity,
     the read_attributes and write_attributes lists must be mutually
    exclusive.
     - Rule: If an attribute is being changed, it must appear ONLY in
    write_attributes.
     - Forbidden: It is strictly forbidden to "read" an attribute to calculate
     its new value within the same access block.
5. ZERO-HALLUCINATION POLICY: Use ONLY attributes explicitly listed in the '
    entities' YAML. You MUST use the correct name from the schema. Never
    invent any field.
6. UPDATE/DELETE PROTOCOL (not only for operation whose name begins with
    update/delete ):
     - Modification: read_attributes = [identifier], write_attributes = [only
    fields being updated].
     - Deletion: read_attributes = [identifier], write_attributes = []. (Leave
     empty, do not repeat the identifier here).
7. EXACT MATCH: Attribute names are case-sensitive. Use the exact spelling
    from the entity definition.
8. DATABASE ACCESS: Each operation must have at least one entity in
    database_access with at least one non-empty attribute list.
9. THE ATOMIC MUTATION RULE (STRICT): In this system, an attribute cannot
    serve as both a search key and a modified value in the same operation.
     - Functional Split: Use read_attributes ONLY for the fixed identifiers
    needed to target the record. Use write_attributes ONLY for the values
    being changed or created.
```

```
    - Isolation: Never include an attribute in read_attributes just because
    the logic needs to "know" it or "calculate" a difference. Assume any
    value required for logic was already retrieved by a previous query
    operation.
    - Disjoint Enforcement: If an attribute appears in write_attributes, it
    MUST be excluded from read_attributes. It is a target of the change, not
    a filter for the search.


Instructions:
STRICT INTEGRITY: Do not modify or delete any existing data (names, entities,
     attributes, consistency, or frequency).
LOCAL UPDATE: Populate the forced_operations field inside each operation
    block.
FORMAT: The forced_operations field must be a list containing the exact names
     of the other operations that must be co-located with the current one.
LOGIC: Analyze the requirements to identify which operations belong together
    and ensure the mapping is consistent across all related operations.

Input provided:
1) List of atomic User Stories:
<User Story list>
...
</User Story list>

2) YAML file of entities:
<Entities YAML file>
...
</Entities YAML file>

Modeling rules (mandatory, never violate):
- Generate ONLY the keys required by Cromlech: name, us_id, frequency,
    consistency, database_access
- For each database_access item: entity_name, read_attributes,
    write_attributes
- frequency must be 1, 5, or 10
- consistency must be 'H' or 'L'
- If write_attributes is non-empty for ANY entity in database_access,
    consistency must be 'H'
```

```
- read_attributes and write_attributes MUST be disjoint for each entity. The
    same attribute string MUST NOT exist in both and do not invent attributes.

- database_access list cannot be empty
- Operation names must be in camelCase and unique
- Map each operation to exactly one US-ID from the provided list
- Do NOT include optional fields if they are empty, omit fields if empty list
- You are only allowed to use attributes explicitly listed in the 'entities'
    YAML provided.
- Each access to an entity attribute must be unidirectional within a single
    operation: it is either a lookup (read) or a mutation (write). No Read-
    Modify-Write: An operation cannot read an attribute to use it as a basis
    for its own update.
- If you need the current value of X to calculate the new one, assume the
    system already knows it from a previous, separate operation.


Required output format (YAML only):
You must output only the YAML content, without any additional text,
    explanations, or markdown code blocks.
```

A.2. Prompt per la generazione delle Forced Operations

```
Role: You are a Software Architect Assistant specializing in generating
    Cromlech-compatible YAML files for microservices decomposition.

Task: Update the provided YAML specification by populating the co-location
    constraints within each operation.

Context: You are provided with a YAML file containing a mapping of software
    operations. Each operation is defined with precision (entities,
    attributes, consistency, frequency).

Definition of Forced Operation:
A "Forced Operation" is a logical co-location constraint indicating that the
    current operation must be assigned to the same physical microservice as
    one or more existing operations.
```

```
The goal is to identify logical affinities between operations that, for
    architectural reasons, should reside within the same execution context.
Such affinities may arise, for instance, from the need to preserve
    transactional atomicity across multiple operations, or from the objective
     of reducing latency and consistency overhead introduced by inter-service
     communication.


Instructions:
STRICT INTEGRITY: Do not modify or delete any existing data (names, entities,
     attributes, consistency, or frequency).
LOCAL UPDATE: Populate the forced_operations field inside each operation
    block.
FORMAT: The forced_operations field must be a list containing the exact names
     of the other operations that must be co-located with the current one.
LOGIC: Analyze the requirements to identify which operations belong together
    and ensure the mapping is consistent across all related operations.

<YAML File>
...
</YAML File>

Required output format (YAML only):
You must output only the YAML content, without any additional text,
    explanations, or markdown code blocks.
```

A.3. Trainticket: Per-Instance Metrics Analysis

A.3.1. DeepSeek-Chat

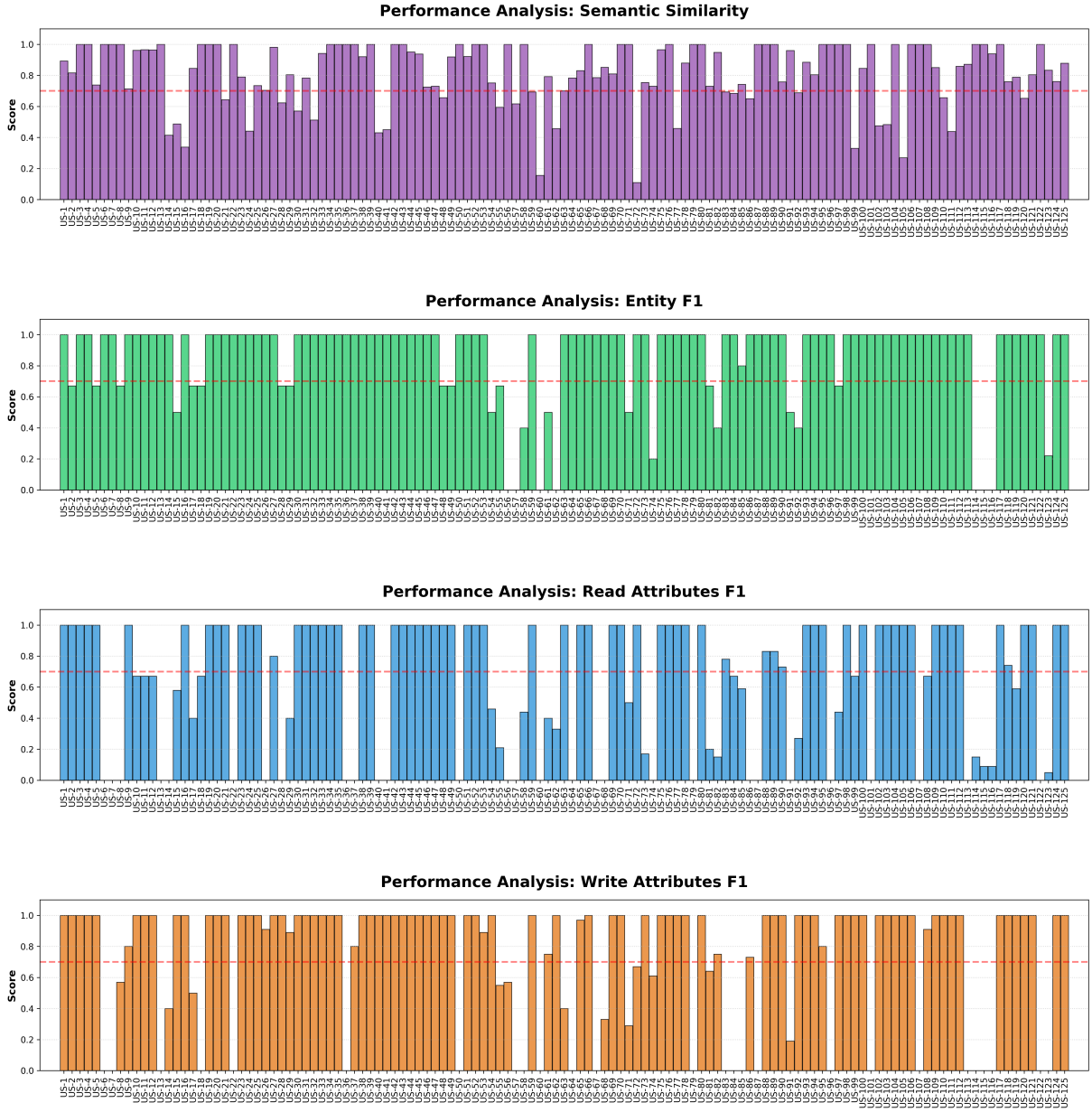

Figure A.1: (Trainticket) Per-instance metrics analysis with DeepSeek-Chat.

A.3.2. DeepSeek-Reasoner

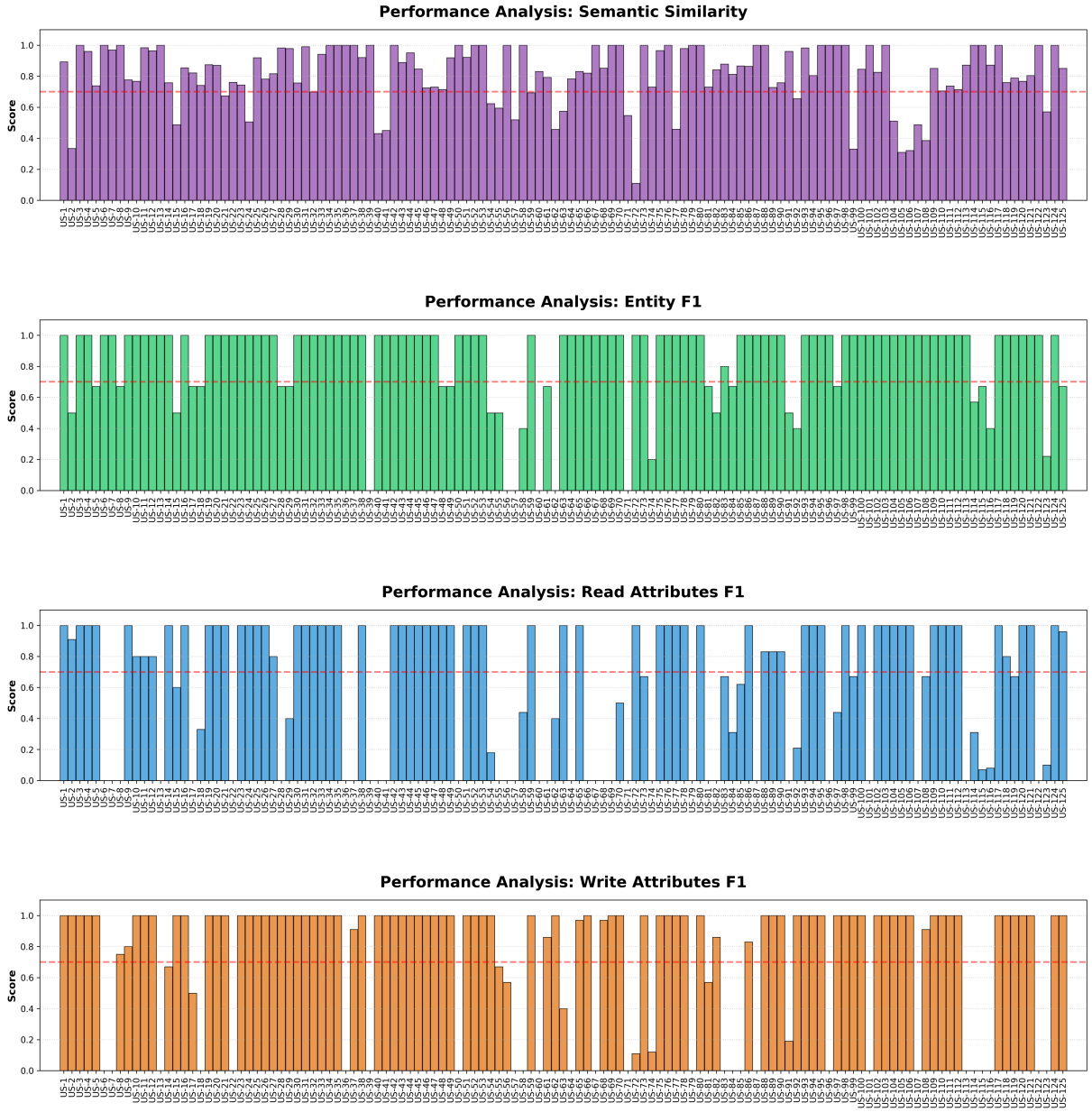

Figure A.2: (Trainticket) Per-instance metrics analysis with DeepSeek-Reasoner.

A.3.3. Mistral-Large-2512

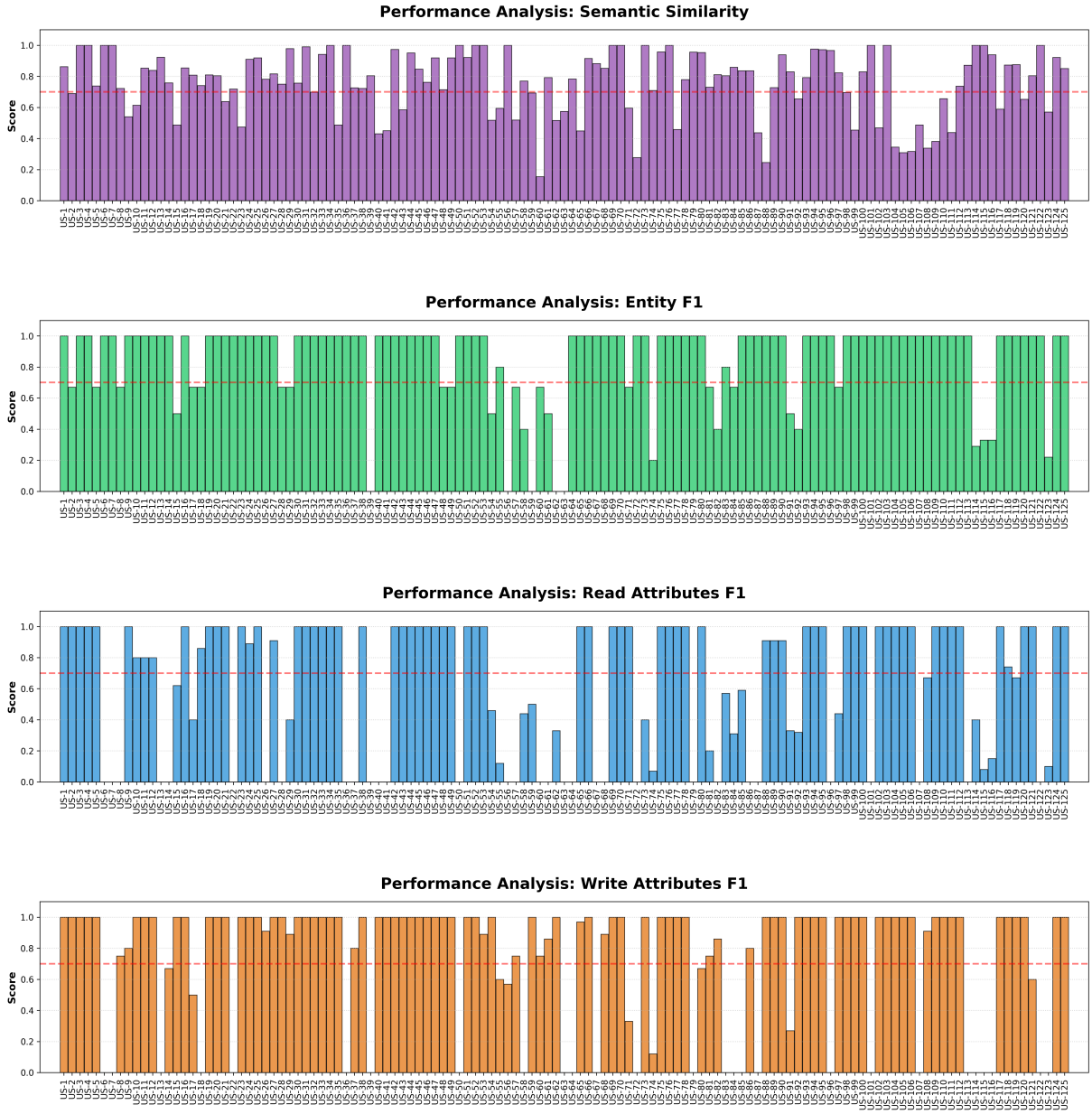

Figure A.3: (Trainticket) Per-instance metrics analysis with Mistral-Large-2512.

A.3.4. Gemini-2.5-Flash

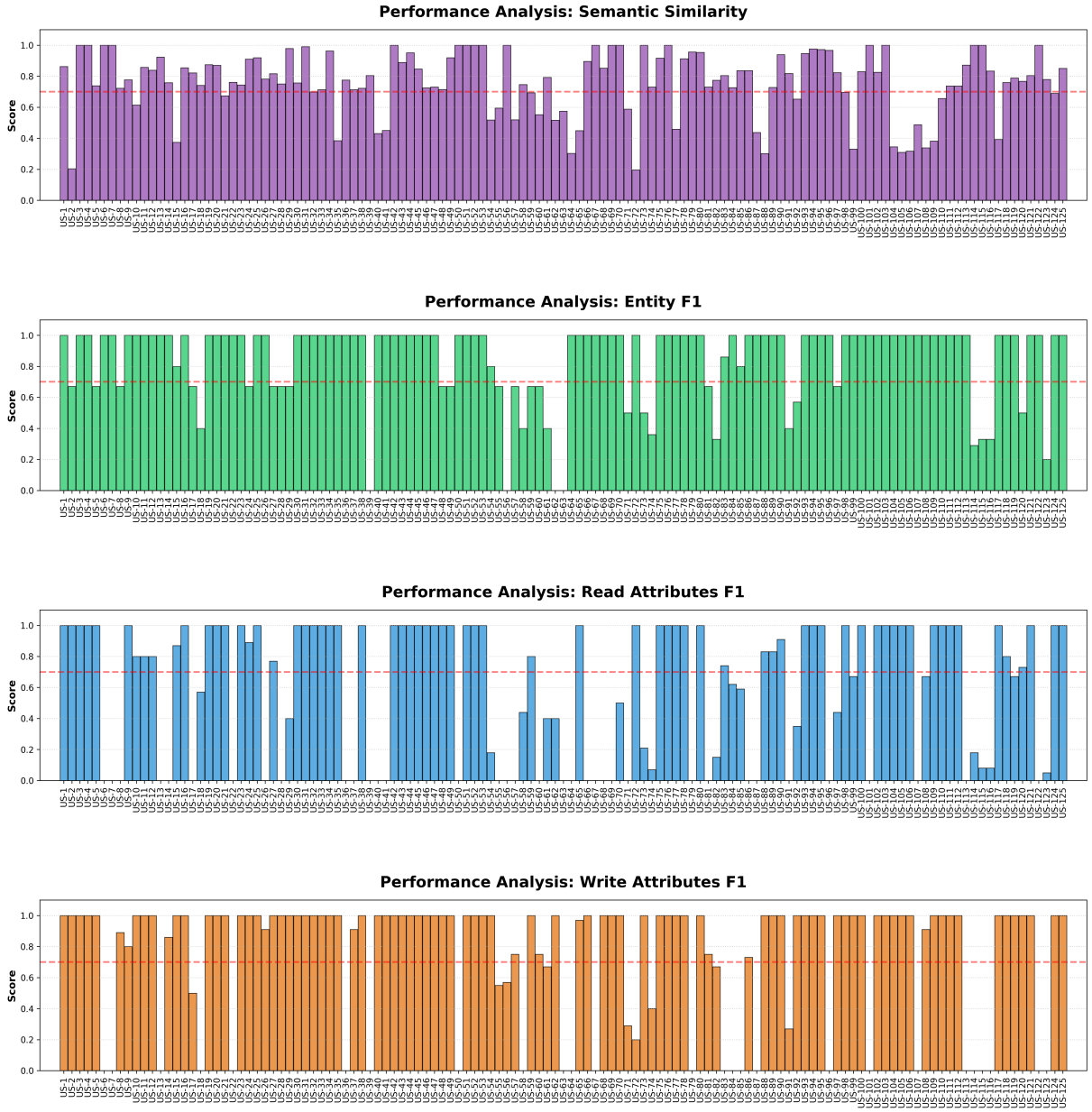

Figure A.4: (Trainticket) Per-instance metrics analysis with Gemini-2.5-Flash.

A.4. Tutored: Per-Instance Metrics Analysis

A.4.1. DeepSeek-Chat

Figure A.5: (Tutored) Per-instance metrics analysis with DeepSeek-Chat.

A.4.2. DeepSeek-Reasoner

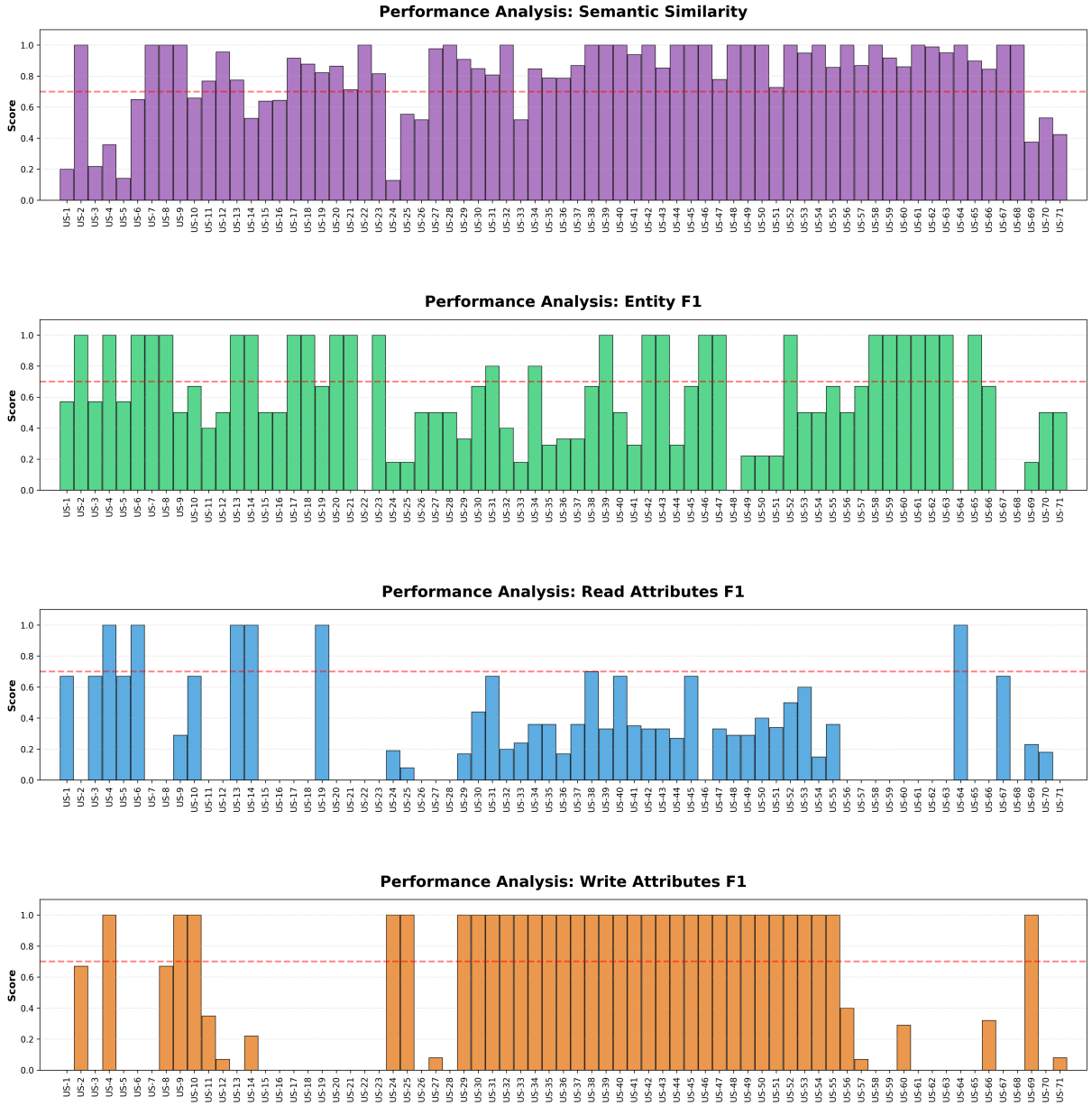

Figure A.6: (Tutored) Per-instance metrics analysis with DeepSeek-Reasoner.

A.4.3. Mistral-Large-2512

Figure A.7: (Tutored) Per-instance metrics analysis with Mistral-Large-2512.

A.4.4. Gemini-2.5-Flash

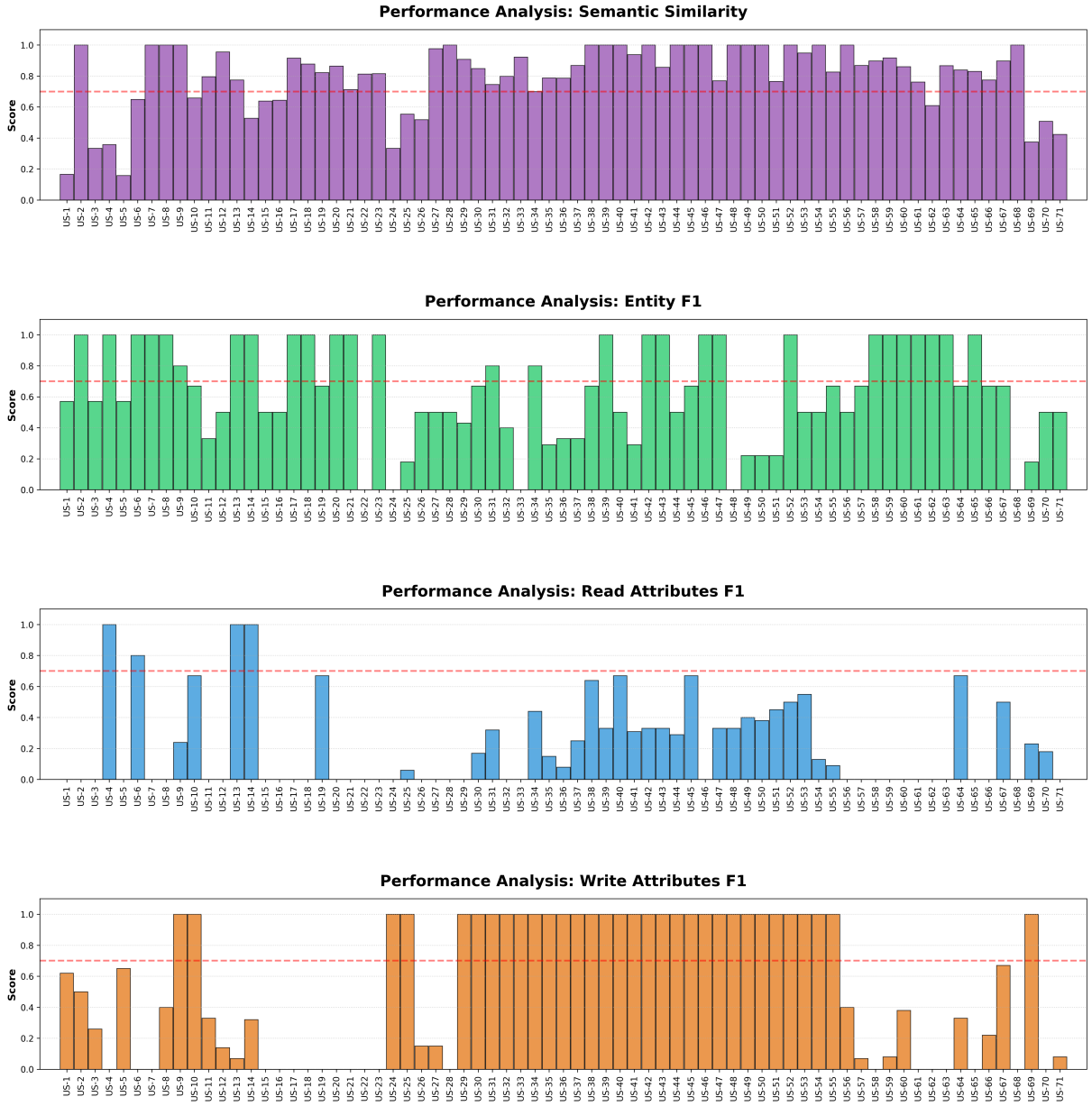

Figure A.8: (Tutored) Per-instance metrics analysis with Gemini-2.5-Flash.

List of Figures

List of Tables