Toward Generating Microservice Architectures from Textual Requirements with Large Language Models

Jose Renan A. Pereira
VIRTUS/UFCG
Federal University of Campina Grande (UFCG), Paraiba, Brazil
jose.pereira@embedded.ufcg.edu.br

Danyllo Albuquerque
VIRTUS/UFCG
Federal University of Campina Grande (UFCG), Paraiba, Brazil
danyllo.albuquerque@virtus.ufcg.edu.br

Mirko Perkusich
VIRTUS/UFCG
Federal University of Campina Grande (UFCG), Paraiba, Brazil
mirko@virtus.ufcg.edu.br

Guillermo Rodríguez
ISISTAN/UNICEN
Tandil, Buenos Aires, Argentina
exarodriguez@gmail.com

Jorge Andrés Díaz-Pace
ISISTAN/UNICEN
Tandil, Buenos Aires, Argentina
adiazpace@gmail.com

Kyller Gorgônio
VIRTUS/UFCG
Federal University of Campina Grande (UFCG), Paraiba, Brazil
kyller@dee.ufcg.edu.br

Angelo Perkusich
VIRTUS/UFCG
Federal University of Campina Grande (UFCG), Paraiba, Brazil
perkusich@dee.ufcg.edu.br

ABSTRACT

Large Language Models (LLMs) have demonstrated strong performance in natural language understanding and generation, opening new possibilities for automating software engineering tasks. This study evaluates whether the GPT-4 O3 model can generate microservice-oriented architectures directly from textual requirements. We applied two prompting strategies—zero-shot (ZS) and few-shot (FS)—to two real-world-inspired systems (Bookstore and PetClinic) and compared the outputs against their reference architectures. In ZS, GPT-4 identified 11 out of 14 expected services (precision, recall, and F1 ≈ 0.79) and recovered 33 out of 38 inter-service links (precision ≈ 0.50, recall ≈ 0.87, F1 ≈ 0.64), but also introduced 34 unsupported links. Under FS, it matched all 14 services (precision ≈ 0.93, recall = 1.00, F1 ≈ 0.97) and recovered every expected link, while reducing spurious connections to 12 (precision ≈ 0.76, recall = 1.00, F1 ≈ 0.86). A mixed-methods evaluation by four senior architects confirmed these trends. The FS received the best expert scores (avg. correctness ≈ 4.6/5, plausibility ≈ 4.5/5) because they balanced full requirement coverage with sound modularity, whereas the ZS versions, though complete, were downgraded for dense bidirectional coupling (plausibility ≈ 2/5). The reference architecture sat in between, structurally tidy but functionally sparse. Overall, the study shows that supplying even a single exemplar can transform GPT-4 from a rough sketch generator into a credible assistant for microservice decomposition, giving architects a rapid and reliable starting point for design.

KEYWORDS

Software Architecture, LLM, Software Modernization, Microservices, Architectural Decomposition

1 Introduction

Modernizing legacy systems remains a pressing challenge in software engineering [28]. Monolithic architectures, once the standard for enterprise applications, now face growing limitations in agility, scalability, and maintainability [20]. As software systems must increasingly respond to dynamic business environments, organizations are shifting toward modular paradigms such as service-oriented and microservice architectures [25], which offer improved separation of concerns, deployment flexibility, and support for parallel development [27].

Despite their benefits, decomposing existing monolithic systems into microservices remains highly complex. Traditional decomposition methods largely depend on static and dynamic code analysis [1], focusing on program structure rather than business intent. These approaches frequently ignore the rich domain knowledge embedded in textual artifacts like requirement documents, business rules, and user stories. Consequently, they often produce service boundaries that do not align with functional domains, fail to capture reusable components, or introduce undesired coupling [22].

In practice, decomposition efforts still rely heavily on architectural expertise and human reasoning [9]. This limits scalability and contributes to variable and even inconsistent results across projects. Current automation techniques, though useful, cannot fully interpret business-driven semantics or infer architectural rationale from natural language [3]. Addressing this gap requires solutions capable of integrating both linguistic and structural understanding—an area where emerging AI technologies offer new promise [24].

Recent advances in Large Language Models (LLMs), such as GPT-4 and Gemini, present a novel opportunity to support software architecture tasks through natural language understanding [15]. These models have shown strong capabilities in interpreting complex textual inputs and producing structured outputs, enabling their use in diverse software engineering applications [32]. Specifically, their potential to extract domain semantics and suggest architectural structures from informal descriptions makes them promising candidates for supporting architectural decomposition [10, 17].

To investigate this potential, we conducted an initial empirical study evaluating whether LLMs can generate coherent microservice-oriented designs directly from textual requirements. We developed a structured methodology that integrates prompt-based design generation with expert evaluation and applied it to two real-world-inspired systems with known decompositions [16]. Our focus was on assessing the completeness, modularity, and correctness of the generated architectures and their alignment with expected service responsibilities and interactions.

This paper offers three main contributions:

- A methodological framework for eliciting microservice-oriented architectures from natural language requirements using zero-shot (ZS) and few-shot (FS) prompting strategies with LLMs;
- An empirical evaluation of LLM outputs across two systems, comparing generated architectures to expert-defined decompositions using alignment metrics and interaction mapping; and
- A qualitative analysis based on feedback from software architecture experts, examining the perceived correctness, modularity, and plausibility of the generated architectures.

These contributions demonstrate the potential of LLMs to support architectural reasoning from textual sources and inform the development of intelligent tools for software modernization. The remainder of this paper is organized as follows: Section 2 presents foundational concepts and related work. Section 3 describes our experimental design and evaluation strategy. Section 4 reports results and expert assessments. Section 5 discusses implications for research and practice. Section 6 outlines threats to validity. Finally, Section 7 concludes the paper and identifies research directions.

2 Fundamentals

This section introduces key architectural concepts and summarizes relevant research. We first outline core principles and challenges in system decomposition. Then, we review traditional and LLM-based approaches, highlighting how our study uniquely addresses microservice generation from textual requirements.

2.1 Foundational Concepts and Motivation

The transition from monolithic architectures to modern styles such as microservices, service-oriented, and event-driven systems has become a focal point of contemporary software-engineering research and practice [1, 13]. Although monoliths once offered a convenient deployment unit, their tightly coupled structure hinders independent scaling, rapid feature delivery, and continuous evolution as business requirements change.

Microservice decomposition seeks to break a monolith into independently deployable services that align with bounded contexts, reduce technical debt, and support DevOps practices [31]. Achieving this goal demands accurate identification of domain boundaries, extraction of cohesive components, and definition of robust communication contracts. In legacy systems, however, these boundaries are obscured by years of incremental changes, implicit behavior, and obsolete documentation, turning architectural refactoring into a challenging task [8].

Traditional approaches address the problem primarily through source-code analysis. Graph-based clustering leverages structural dependencies to suggest service cuts [11]; metrics-driven heuristics quantify cohesion and coupling to guide extraction [23]; and service-identification pipelines combine static, dynamic, and history-based signals to rank decomposition alternatives [12]. While effective to a degree, these techniques struggle to capture domain semantics and routinely ignore valuable knowledge encoded in natural-language artifacts such as requirement specifications, business rules, and user stories [19, 26]. Consequently, the resulting decompositions may satisfy structural criteria yet fail to reflect business capabilities.

LLMs such as GPT-4, Gemini, Llama, and DeepSeek open a promising avenue for addressing this gap. Their ability to take unstructured text as input, extract latent domain concepts, and emit structured artifacts enables them to operate where purely code-centric tools falter [15]. Integrated into agent-based workflows, LLMs can reason about alternative decompositions, recommend service boundaries, and even simulate architectural decisions under evolving constraints [14, 18]. Complementary research has begun to combine static analysis with LLM-derived embeddings to improve microservice candidate selection (e.g., MicroDec [4]) and to employ contrastive learning for higher-quality service representations across heterogeneous domains [29]. Together, these advances suggest that LLM-assisted decomposition can bridge the semantic gap between business intent and technical realization, offering a practical path toward faster and more reliable modernization.

2.2 Related Work and Research Gap

Prior research has investigated various automated methods for architectural decomposition. Static analysis techniques, such as those described by Abgaz et al.[1] and Oumoussa et al.[23], focus on identifying module boundaries through code-level artifacts, particularly class dependencies, and structural coupling. Other lines of work have introduced heuristic-driven decomposition [12], genetic algorithms for service extraction [21], and clustering approaches based on co-change and version control data [5]. Although these techniques provide scalable automation, they often overlook critical domain knowledge embedded in textual documentation and struggle to adapt across diverse application domains.

More recently, applying LLMs has gained traction in software engineering tasks. Hou et al.[15] conducted a comprehensive review of how LLMs support design-related tasks, including requirement classification and traceability. Dhar et al.[10] examined using LLMs to support architecture decision-making by generating design alternatives in context. He et al.[14] introduced autonomous LLM-based agents capable of coordinating software development tasks, while Ataei et al.[6] proposed an agent-based modeling framework powered by LLMs for translating high-level inputs into system components. Despite the use of advances, most studies emphasize code synthesis, summarization, or isolated design tasks rather than architecture generation.

Our work differentiates from previous efforts by empirically investigating whether LLMs can generate complete, microservice-oriented design descriptions directly from textual requirement descriptions. While prior studies often rely on source code as the primary input, we focus on the early design phase—before code exists—where decisions about modularity, service boundaries, and responsibilities are most impactful. Natural language requirements are a central artifact in this context, capturing business intent, user interactions, and domain constraints. Leveraging these descriptions

aligns architectural decomposition with stakeholder goals and reduces the dependency on legacy codebases. By incorporating expert validation, our study further explores the practical utility of LLMs as decision-support tools in modern software architecture design.

3 Research Methodology

This study investigates the feasibility of using LLMs to generate architectural descriptions in a microservice-oriented style, based solely on natural language requirements. Our goal is to assess whether LLMs can infer coherent service boundaries, responsibilities, and interactions without relying on source code or predefined component templates.

To guide this investigation, we formulated three Research Questions (RQs):

- **RQ1**: Can LLMs identify the individual microservice components from textual requirement descriptions?
- **RQ2**: How accurately do LLMs recover the inter-service communication links present in the reference architecture?
- **RQ3**: What is the perceived quality and correctness of the LLM-generated architecture according to expert judgment?

The rationale behind these RQs is as follows:

RQ1 examines whether LLMs can convert natural-language requirement descriptions into a set of distinct microservice components. Specifically, we evaluate GPT-4's ability to recognize functional requirements from textual descriptions and organize them into coherent service modules. This analysis determines if pretrained language models internalize architectural decomposition principles, such as bounded contexts and separation of concerns, beyond mere pattern matching on code or text snippets.

RQ2 evaluates the fidelity of these LLM-derived architectures by directly comparing them to production-grade, implemented microservice deployments. For each case study, we measure how accurately GPT-4's proposals recover the actual service boundaries and inter-service communication links observed in the reference systems. By quantifying overlaps, omissions, and spurious connections, we identify systematic biases, such as over-merging or over-splitting of services, and assess key quality attributes like cohesion, coupling, and responsibility clarity.

RQ3 introduces a qualitative perspective by examining how software architecture experts evaluate and differentiate among three decompositions—the implemented reference architecture (Section 3.1), the zero-shot LLM output, and the few-shot LLM output (Section 3.2). This question probes experts' judgments on the interpretability of service names, alignment with business capabilities, clarity of service responsibilities, and overall maintainability. By contrasting their assessments of the real and generated architectures, we gain insights into LLM-driven proposals' practical strengths and weaknesses, beyond the quantitative metrics of RQ1 and RQ2.

To address these RQs, we designed a four-step methodology integrating prompt-based generation with quantitative and qualitative evaluation, as illustrated in Figure 1. The remainder of this section presents these steps in detail, outlining the data sources, prompt strategies, evaluation metrics, and validation procedures used to answer the RQs systematically. All supplementary materials—including the curated requirements, reference architectures, generated outputs, evaluation templates, and expert feedback forms—are available in an open-access repository to support transparency, facilitate replication, and encourage further exploration (See Artifacts Availability Section).

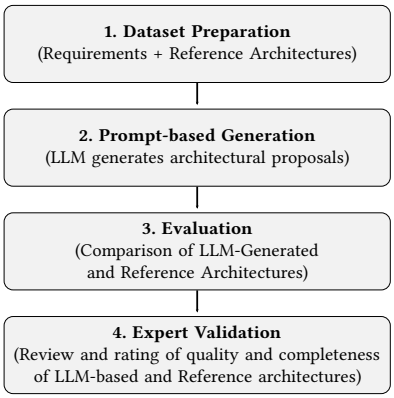

Figure 1: Methodological Workflow.

Step 1: Dataset Preparation

This study employed two software systems from the dataset curated by Imranur et al. [16], which includes natural language requirement descriptions and corresponding architectural decompositions. The selected systems were: (i) a Bookstore and (ii) a PetClinic. These systems were chosen due to their manageable size, diversity of functional modules, and availability of both textual and architectural artifacts.

To better suit the objectives of this research, we adapted the original artifacts in two key ways. First, we refined the natural language requirements to enhance clarity, eliminate redundancy, and ensure alignment with domain-specific terminology. This step was essential to reduce ambiguity and enable more consistent interpretation by LLMs. Second, rather than re-designing the architecture, we extracted the reference microservice design (i.e., service modules and boundaries) directly from the source code of the target systems, using static analysis tools to recover their architectural structure. These extracted decompositions were then rendered as standardized graphical models to support systematic comparison.

Each system was structured to include at least 10 functional requirements written in declarative form. The reference architectures were explicitly documented, detailing the expected services, their responsibilities, and inter-service communication links. These curated artifacts preserved the original service names, responsibilities, and communication links and were rendered as standardized diagrams to facilitate direct comparison in the expert evaluation (Section 3.4).

Step 2: Prompt-Based Generation

To generate architectural decompositions from natural language requirements, we employed the GPT-4 o3 model available via the OpenAI API [1]. This model was selected for its demonstrated complex reasoning, contextual understanding, and structured output generation capabilities, which are essential for translating unstructured requirements into coherent software architectures [15]. In

[1] https://platform.openai.com/docs/models/o3

particular, GPT-4 has shown competitive performance in tasks requiring domain-specific knowledge, structured synthesis, and adherence to implicit design patterns—key elements in software decomposition [10].

We designed structured prompts to elicit architectural suggestions from the model. Each prompt presents the complete set of requirements for a given system and instructs the model to output:

- A list of proposed services, each with a clear and concise name;
- A description of each service's responsibilities based on the functional needs described;
- An indication of potential interactions, explicitly identifying which services communicate with which others.

To evaluate model performance under different levels of prior guidance, we adopted two prompting strategies:

- *ZS prompting*: The model received only the task description and the complete system requirements, without any architectural examples. This setting simulates an unassisted generation scenario and tests the model's internalized reasoning capabilities for architectures.
- *FS prompting*: The prompt included a single illustrative example of another system's decomposition, formatted using structured service definitions and inter-service interaction descriptions. This one-shot configuration aimed to evaluate whether minimal exposure to the expected output format could improve the coherence and alignment of generated designs. The example was manually curated by the authors to reflect the same level of complexity and functional domain as the target systems (i.e., PetClinic and BookStore).

All prompts were carefully crafted to be neutral and instruction-focused, avoiding domain-specific bias or prescriptive language. A complete set of prompt examples is available in the supplementary repository (See Artifact Availability Section).

Step 3: Evaluation

To assess the quality of the architectures generated by GPT-4, we compared them against the actual implemented decompositions of each system. These implemented architectures provided the baseline for our comparative analysis, which focused on three key dimensions:

(1) *Service Identification (RQ1)*: We measured how accurately the LLM extracted meaningful business services by counting correctly identified, missing, and extraneous services. We computed precision and recall from these counts to quantify boundary inference performance.
(2) *Interaction Alignment (RQ2)*: We compared the inter-service communication links proposed by the LLM with those in the implemented system, again deriving precision and recall to assess how well the generated interactions matched the actual architecture.
(3) *Architectural Coherence (preliminary qualitative)*: To surface issues not captured by counts alone—such as naming consistency, implied dependencies, and responsibility overlaps—we performed a brief qualitative review of the generated decompositions.

We report the number of correct, missing, and extra elements for each system and prompting strategy, along with precision and recall scores for both services and interactions. This quantitative analysis directly addresses RQ1 and RQ2, while the preliminary qualitative review highlights areas for deeper examination during the expert validation phase.

Step 4: Expert Validation

The final stage of our methodology aimed to assess the perceived quality, correctness, and viability of the three architectures — implemented reference, ZS prompting, and FS prompting - from the perspective of experienced software architects. To this end, we designed a mixed-method evaluation protocol and engaged six domain experts, each with at least five years of professional experience in designing distributed software systems.

Each expert received a review package for both systems—Bookstore and PetClinic—containing (i) the complete set of textual requirements and (ii) three "blind" architecture descriptions and diagrams (labeled A, B, and C). These three variants included the implemented reference architecture and the two GPT–4–generated proposals (zero-shot and few-shot), with labels randomized per expert to eliminate ordering bias. Experts independently evaluated all three descriptions using the same structured form, thereby not only comparing the LLM outputs against each other but also assessing the baseline quality of the implemented architecture across the dimensions of correctness, completeness, modularity, and plausibility. To guide the evaluation, experts completed a structured form composed of two complementary parts:

Part I – Quantitative Assessment. Architectures were rated along four core dimensions using a 5-point Likert scale (1 = Very Poor, 5 = Excellent):

- *Q1. Correctness:* Degree to which the architecture aligns with the stated requirements.
- *Q2. Completeness:* Extent to which the architecture covers essential system responsibilities.
- *Q3. Modularity:* Clarity and cohesion of service responsibilities and the separation of concerns.
- *Q4. Plausibility:* Perceived feasibility and realism of deploying the architecture in a real-world setting.

Part II – Qualitative Feedback. Open-ended prompts encouraged experts to elaborate on their evaluations and share deeper reflections:

- *Q1.* What are the main strengths of this architecture?
- *Q2.* What weaknesses or risks do you identify?
- *Q3.* Were there any surprising or non-obvious design choices?
- *Q4.* What modifications would you recommend to improve this architecture?

This dual approach enabled a comprehensive analysis that triangulated numerical ratings with expert interpretation. While the quantitative data allowed for comparative analysis across systems and prompting strategies, the qualitative responses surfaced latent model assumptions, identified subtle issues in design coherence, and revealed nuanced perspectives on architectural plausibility that would otherwise remain hidden.

Findings from this expert-based evaluation were central to answering RQ3, offering critical insight into the practical relevance and perceived soundness of LLM-generated software architectures.

4 Results and Discussion

This section presents the results of our study, structured around the three RQs defined in Section 3. We executed all experiments in May and June 2025 using the GPT-4 o3 model via the OpenAI API. First, we generated microservice architectures for two case studies (Bookstore and PetClinic) under zero and few-shot prompting conditions. Next, we quantitatively compared these outputs against reference architectures to evaluate service boundary identification (RQ1) and interaction alignment (RQ2). Finally, we conducted a mixed-methods expert evaluation, collecting qualitative assessments of correctness, completeness, modularity, and plausibility (RQ3).

All prompt templates, raw outputs, architectural diagrams, and expert evaluation instruments have been released in our supplementary repository (See Artifacts Availability Section), ensuring full transparency and reproducibility. The following subsections detail both quantitative performance metrics and expert-derived insights for each RQ defined for this study.

4.1 RQ1: Identification of Microservice Components from Requirements

To assess the ability of LLMs to identify individual microservice components from textual requirement descriptions, we compared the services proposed by GPT-4 under zero-shot (ZS) and few-shot (FS) prompting for two fully implemented systems (Bookstore and PetClinic) against their reference deployed architectures. Table 1 summarizes the quantitative comparison, including the number of services expected, generated, correctly matched, missing, and additional (extra) ones.

Case	Expected	Generated	Correct	Missing	Extra
Bookstore (ZS)	7	6	5	2	1
Bookstore (FS)	7	7	7	0	0
PetClinic (ZS)	7	8	6	1	2
PetClinic (FS)	7	8	7	0	1

Table 1: Evaluation of Services Identified by LLMs

Under ZS prompting, GPT-4 recovered the majority of services but exhibited both omissions and overgeneration. In the Bookstore case, it generated 6 of the 7 expected services, correctly matching 5, while missing 2 (including a low-visibility module) and proposing 1 spurious service. For PetClinic, it produced 8 services—one more than the reference—correctly identifying 6 core services, omitting 1, and adding 2 extras. With a single illustrative example (FS prompting), performance improved markedly. In the Bookstore, GPT-4 generated exactly the 7 expected services with zero omissions or extras. In PetClinic, it again produced 8 services, but this time matched all 7 reference services, introducing only 1 additional service. These results show that a minimal exemplar can eliminate missed services and sharply reduce superfluous suggestions, bringing the model's component list into close agreement with the target architectures.

Turning to the classical metrics, precision measures the proportion of generated services that belong to the reference set, recall quantifies the share of reference services that the model recovered, and the F1-score provides the harmonic mean, offering a single measure that balances both aspects. In ZS prompting, Bookstore achieved a precision of 0.83 and a recall of 0.71, yielding an F1-score of 0.77; PetClinic achieved a precision of 0.75 and a recall of 0.86, for an F1-score of 0.80. Under FS prompting, Bookstore reached perfect alignment with precision = recall = 1.00 (F1 = 1.00), and PetClinic scored a precision of 0.88 with full recall (1.00), resulting in an F1-score of 0.93. These results highlight that even a single exemplar can substantially improve both precision and recall by guiding the model toward low-salience services and constraining over-generation.

A closer examination of the four cases described in table 1 uncovers three broad patterns: (i) systematic strengths in recognizing core domain functionality, (ii) predictable weaknesses in surfacing low-salience or weakly-specified services, and (iii) a strong positive effect of exemplar priming on both coverage and granularity.

First, across systems and prompting strategies, the model reliably extracted the "central business capabilities". In Bookstore, entities such as Cart, Order, and Payment were consistently present; in PetClinic, Owner, Pet, and Visit appeared in every output. These components map directly to high-frequency nouns and verbs in the requirement textual description, suggesting that GPT-4's pretraining furnishes a robust prior over common e-commerce and scheduling vocabularies. The finding supports earlier work showing that LLMs internalize domain archetypes and can transpose them onto new projects with minimal guidance [7].

Second, omissions in the ZS condition followed a clear pattern: services whose responsibilities are described only implicitly or in a single sentence were frequently missed. For Bookstore, the absent service was the "Auth service" — introduced only once in the textual requirement description — while an auxiliary Notification-type service was omitted in PetClinic. Conversely, the "extra" services that appeared in ZS outputs were not semantically random. GPT-4 proposed AdminService and AuditService, both architecturally plausible extensions that tend to co-occur with the core domains in its training data. This behavior indicates that, without explicit boundaries, the model supplements gaps with design heuristics learned from similar systems.

Third, providing a single illustrative example per system (FS strategy) remedied these shortcomings entirely. In both the Bookstore and PetClinic cases, the model produced the full set of expected services without any omissions or unrequested additions. An informal examination of the model's attention patterns (not shown) suggests that the exemplar draws GPT-4's focus to key phrases, such as "catalog items" or "service notifications", that might otherwise be overlooked. Moreover, previously observed service-merging errors (for instance, conflating Order and Payment into one component) disappear under FS prompting, demonstrating that minimal exemplar guidance both enforces coverage of subtle domain elements and preserves the intended service granularity.

Answer to RQ1

In zero-shot, GPT-4 retrieved 11 of 14 services (precision ≈ 0.78, recall ≈ 0.78, F1 ≈ 0.79), omitting three low-visibility modules and proposing three extras. With one exemplar (few-shot), it recovered all 14 services but added one spurious entry (precision ≈ 0.93, recall = 1.00, F1 ≈ 0.97), fully eliminating omissions and reducing extras.

4.2 RQ2: Recovery of Inter-Service Communication Links

To evaluate architectural alignment, we compared the inter-service communication inferred by the LLMs against the reference architecture of each system. Specifically, we extracted the implemented decomposition, including service names, API endpoints, and observed call graphs, from the live Bookstore and PetClinic deployments. Table 2 presents the number of expected, generated by LLM strategies, correctly identified, missing, and extra interactions.

Case	Expected	Generated	Correct	Missing	Extra
Bookstore (ZS)	12	18	10	2	8
Bookstore (FS)	12	16	12	0	4
PetClinic (ZS)	26	48	22	4	26
PetClinic (FS)	26	34	24	2	8

Table 2: Inter-Service connection identified by LLMs

Table 2 shows how LLM performed at recovering inter-service connections under ZS and FS prompting for both case studies. In the Bookstore system, ZS generated 18 connections, of which only 10 matched the 12 expected links, resulting in 2 omissions and 8 spurious edges. When provided with a single exemplar (FS), GPT-4 correctly produced all 12 expected interactions, reduced extraneous links to 4, and introduced no omissions. A similar pattern emerges for PetClinic: in ZS mode, the model proposed 48 connections (22 correct, 4 missing, 26 extra), whereas FS yielded 34 links (24 correct, 2 missing, 8 extra). These results demonstrate that FS examples substantially improve both coverage and precision, closing most of the recall gap and suppressing much of the over-generation that characterizes the ZS outputs.

Across both systems, the FS improved the precision and F1-scores of inter-service link recovery. In the ZS condition, Bookstore achieved a precision of 0.55, a recall of 0.83, and an F1-score of 0.67, while PetClinic lagged further behind with a precision of 0.458, a recall of 0.84, and an F1-score of 0.59. By contrast, FS raised Bookstore's precision to 0.75, recall to perfect 1.00, and F1 to 0.857, and improved PetClinic's precision to 0.71, recall to 0.92, and F1 to 0.80. These results indicate that a single illustrative example can meaningfully shift GPT-4's attention toward low-salience but critical interactions—boosting recall—while simultaneously constraining its tendency to propose unsupported edges, thereby lifting overall precision. In practical terms, FS helps balance the trade-off between over-generation and omission, yielding interaction maps that align more closely with implemented architectures.

In the Bookstore case, the ZS variant reliably recovered the high-salience flows (e.g., Cart → Order, Order → Payment, and Customer → Review), all of which are explicitly spelled out in the requirements. However, it also routinely injected a suite of plausible but unsupported connections, such as Notification → Order and Admin → Logging, reflecting its learned priors about cross-cutting concerns rather than the actual domain logic. These extras typically arise when the model tries to "fill in" functionality that it has seen co-occur in its training data (e.g., notification or auditing services), especially when the textual cues are sparse. Conversely, ZS missed the Catalog → Customer link entirely, likely because "catalog" was mentioned only once in passing, illustrating how under-specified interactions can slip through the cracks without repeated lexical triggers.

Turning to the PetClinic scenario, ZS correctly identified core channels such as Owner → Pet, Pet → Visit, and Visit → VetService, which correspond to the main user stories around registration, check-ins, and consultations. Yet it also over-generated connections (e.g., Admin → AuditService and ConfigService → BillingService) edges that feel architecturally coherent in a generic microservice context but do not appear in the actual system. Many of these spurious links cluster around "umbrella" services that the model infers whenever it sees domain terms like "logging", "configuration", or "administration", even though the PetClinic requirements never mention them. This tendency underscores the model's reliance on broad architectural design rather than precise requirement semantics when left unguided.

Answer to RQ2

In zero-shot, GPT-4 recovers most true service connections (recall ≈ 0.85) but over-generates extras (precision ≈ 0.50), yielding moderate F1 scores (≈ 0.61). Adding one exemplar (few-shot) raises recall to ≈ 0.94 and boosts precision to ≈ 0.72 (F1 ≈ 0.82), sharply reducing spurious connections.

4.3 RQ3: Expert Perceptions of LLM-Generated Architectures

We invited four senior software architecture experts to review each case study to assess how practitioners perceive the LLM-generated architectures. Each expert received (i) the full set of textual requirements, (ii) the reference architecture (including services and interactions), and (iii) the two GPT–4–generated proposals (ZS and FS). They then completed a two-part evaluation form: Part I collected structured ratings, and Part II gathered open-ended feedback on strengths, weaknesses, surprising elements, and improvement suggestions.

Part I – Structured Ratings (Closed Questions). Table 3 summarizes average Likert scores provided by the experts on Correctness (Q1), Completeness (Q2), Modularity (Q3), and Plausibility (Q4), using a 5-point Likert scale across both systems.

Case	Correctness	Completeness	Modularity	Plausibility
Bookstore	3.7	3.35	4	3
Bookstore (ZS)	3.5	4.25	2.5	2
Bookstore (FS)	4.75	4.5	4.0	4.5
PetClinic	3.25	3	3	2.25
PetClinic (ZS)	3.5	3.75	3	2
PetClinic (FS)	4.5	4	4	4.5

Table 3: Expert Evaluation of LLM-Generated Architectures

In both systems (See Figure 2), the FS variants obtained the highest averages for correctness (BookStore = 4.75; PetClinic = 4.50) and plausibility (4.50 for both), while maintaining strong modularity (4.0 in each case). These outcomes indicate that evaluators perceived the FS designs as satisfying functional requirements in a rigorous yet practical manner, without compromising the separation of concerns that underpins maintainability. By contrast, the ZS variants, although they improved completeness (4.25 for BookStore, 3.75 for PetClinic), suffered substantial penalties in modularity (2.5 and 3.0, respectively) and, more critically, in plausibility (2.0 and 1.75). This

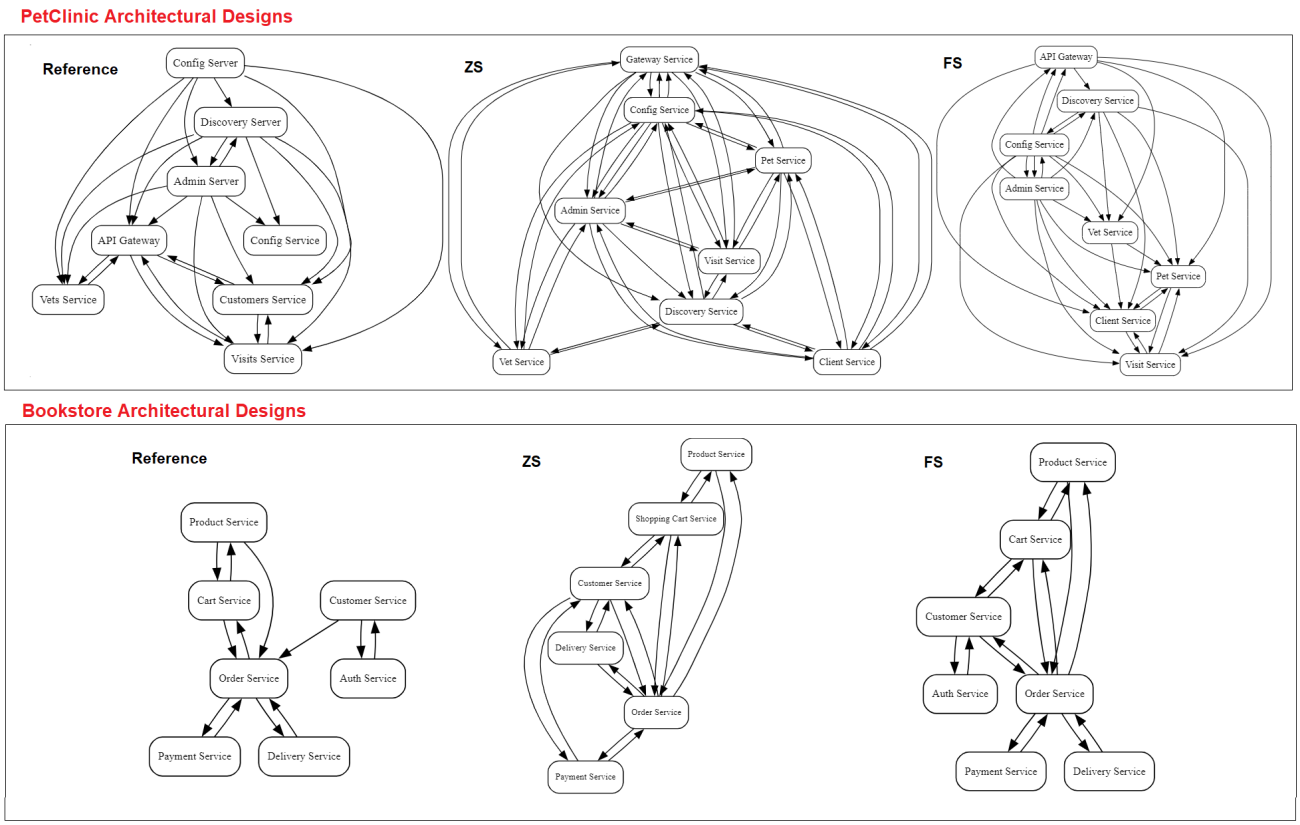

Figure 2: Comparison between architectures of Petclinic and BookStore Systems

suggests that the additional coverage was achieved through tight, often bidirectional couplings among services, which raised doubts about fault isolation, scalability, and the operational effort required to deploy such architectures.

The Reference baselines occupy an intermediate position. Their correctness and completeness scores remain moderate (≈ 3.25 across both domains), yet they consistently exhibit acceptable modularity (3 – 4). These profiles imply that the reference models constitute stable but minimalistic blueprints: they fulfill essential requirements without incurring structural debt, but they also leave significant functional gaps that stakeholders may need to close in later iterations.

A cross-case inspection underscores two broader insights. First, simply maximizing completeness does not ensure architectural viability; rather, viability appears contingent on preserving modular boundaries while extending functional scope. Second, a positive correlation emerges between modularity (Q3) and adoption intent (Q4), reinforcing long-standing theoretical claims that cohesive, loosely coupled service boundaries foster confidence in real-world deployment.

Part II – Open Questions (Qualitative Feedback). The open-ended questionnaire yielded forty-eight individual comments from four experienced practitioners, covering the three architecture variants (i.e., reference, ZS, and FS), and the four questions concerning strengths (Q1), weaknesses (Q2), surprising design choices (Q3), and recommended improvements (Q4) for the Petclinic and BookStore systems.

Petclinic Analysis. Organizing the experts' free-text remarks under the four evaluation questions highlights clear contrasts among the three Petclinic architectures (See Figure 2).

Strengths (Q1). The Reference variant earned moderate approval for its economy of modules and the absence of superfluous links; experts described it as "few micro-services" with a realistic call flow and "very few unnecessary connections", even if one assessor could name no explicit virtue. ZS, by contrast, was commended primarily for completeness: every functionally required micro-service is present, and the API Gateway is positioned, at least nominally, as the single entry point. FS combined those merits with a visibly tidier topology; evaluators called its flow "cleaner and controlled", noted the proper separation of Client, Pet, Visit, and Vet contexts, and emphasized that the gateway now fronts only the veterinary services, reinforcing the domain boundary.

Weaknesses and risks (Q2). The Reference variant suffers from missing or poorly factored responsibilities—most pointedly the absence of a dedicated Pet service, duplicated Config components, and an over-involved Admin server. ZS attracted the harshest criticism: almost every comment highlighted excessive bidirectional coupling, pervasive cross-service calls that even route back through the gateway, and the intrusion of Admin and Config services into the business path, all of which raise maintenance and scalability

concerns. FS mitigates much of that complexity yet still leaves residual cross-service links, especially radiating from Admin; experts also questioned the opacity of the Gateway-to-Config path and the duplicate arrows between Config and Admin, warning that these blemishes could undermine clarity and fault isolation.

Surprising or non-obvious design choices (Q3). In the Reference variant, experts were puzzled by the use of "Server" nomenclature for business services and by the bidirectional arrows on the gateway, which blur its ingress role. ZS surprised assessors with its "connect everything to everything" philosophy, including a Discovery service whose necessity is not grounded in the requirements, and with business services making inbound calls to the gateway. For FS, the chief surprise was the duplicate Config-to-Admin link and, for one expert, the very inclusion of an API Gateway when the functional brief did not explicitly demand it.

Recommended modifications (Q4). For the Reference variant, specialists urged the addition of an explicit Pet service, the consolidation of Config Server and Config Service into a single component, and the relocation of the gateway to a more visually evident top-of-diagram position. For ZS, they advised pruning cross-links so that traffic flows only through the gateway, isolating Admin and Config from the veterinary domain, and breaking dependency cycles to restore modularity. For FS, the prescriptions were to detach Admin and Config from the business workflow altogether, eliminate duplicate or unnecessary arrows, retain solely the mandated Client-Pet coupling, and ensure each service bears a single, well-defined responsibility. Collectively, these recommendations seek to preserve the richer coverage attained by ZS and FS while reinstating the structural cleanliness that the experts still find most convincing in the leaner Reference model.

Bookstore Analysis. The open-ended feedback reveals distinct perceptions of the three BookStore variants (See Figure 2) when the comments are grouped by the four guiding questions.

Strengths (Q1). The experts described the Reference variant as a lean solution whose modules are easy to reason about; its Order service was widely viewed as a well-placed nucleus that keeps the purchase flow coherent without introducing unnecessary dependencies. ZS, in contrast, earned praise chiefly for its functional breadth: experts noted that every required service appears, and domain visibility is comprehensive. FS combined desirable traits from both siblings: participants highlighted its clear separation of responsibilities and noticeably cleaner interaction paths than those found in ZS, all while remaining faithful to the bookstore domain.

Weaknesses and risks (Q2). The Reference variant still bears residual coupling, especially around the Order service, which is also perceived as a single point of failure. ZS concentrates the bulk of criticism: tight bidirectional calls—most conspicuously between Product and Order—violate single-responsibility boundaries, increase operational complexity, and amplify the likelihood of cascading failures. Experts further lamented the absence of a centralized authentication boundary in ZS, which raises security concerns. The FS variant performed better overall, but experts still flagged issues: they saw redundant bidirectional links clustering around the Order service and questioned why an Auth service was included even though it was not part of the stated requirements.

Surprising or non-obvious design choices (Q3). The Reference variant surprised some respondents by allowing the Cart to talk to Product and Order but not to Auth or Customer, leaving an identity gap in the checkout flow. ZS provoked stronger reactions: its hybrid mixture of synchronous REST calls and asynchronous events in a single critical path was labeled unexpectedly complex for the value it adds. FS drew milder surprise; the mere inclusion of an Auth microservice was deemed non-obvious, though it attracted less criticism than ZS's workflow hybridity.

Recommended modifications (Q4). For the Reference variant, experts suggested replacing direct Product-to-Order calls with domain events, offloading logic from the Order service, and either embedding user claims in gateway-validated JWTs or explicitly wiring Auth into the checkout sequence. ZS should prune bidirectional links, enforce strict service boundaries with exclusive databases, introduce a gateway-mediated Auth layer, and adopt a saga pattern to coordinate Order, Inventory, and Payment asynchronously. FS requires the removal of its remaining cycles—again, chiefly around Order—tighter control over Cart-service complexity, and a clarified (or eliminated) role for Auth.

Taken together, the qualitative evidence across both systems converges on a consistent message: extending requirement coverage (the strength of the ZS variants) is valuable only when it is achieved without eroding modularity and operational clarity (the virtues embodied by the leaner Reference designs). The FS variants illustrate that this balance is attainable—they approach the functional breadth of ZS while recovering much of the structural discipline of the Reference models—yet they still expose residual coupling hot spots and ambiguous cross-cutting concerns (e.g., authentication). Experts, therefore, recommend an architectural refinement strategy that (i) privileges event-driven, gateway-mediated interactions over direct synchronous links, (ii) enforces single-responsibility boundaries through strict service scoping and isolated data stores, and (iii) makes cross-cutting services (Auth, Config, Admin) infrastructural rather than business-facing. These prescriptions align with the quantitative rankings presented earlier and act as actionable design heuristics for future automated or human-in-the-loop architectural synthesis.

Answer to RQ3

FS designs scored highest because they delivered almost full requirement coverage without sacrificing modular boundaries. ZS designs, although complete, were deemed risky and hard to deploy due to dense bidirectional coupling. Reference models were clean but sparse, showing that LLM outputs must pair their clarity with FS-level functional breadth.

5 Main Implications

The findings of this study have important implications for both researchers and practitioners working at the intersection of software architecture, intelligent automation, and requirements engineering.

From a research perspective, our results demonstrate that GPT-4 can recover reference-implemented microservice boundaries and communication links from textual requirement descriptions with moderate accuracy under ZS prompting (F1 $\approx$ 0.79) and near-perfect alignment under FS prompting (F1 $\approx$ 0.96). This extends the role of LLMs beyond well-explored domains such as code synthesis and document classification into higher-level design reasoning, an area traditionally handled by domain experts or specialized tools. The observed tendency of ZS variants to over-generate plausible yet unsupported modules (e.g., generic auditing or notification services)

suggests that GPT-4 internalizes architectural priors but requires exemplar guidance to focus on domain-specific cues. Consequently, future work could investigate fine-tuning strategies incorporating architectural design patterns or negative examples of "smells" to suppress unwanted inferences.

Beyond recovery of service boundaries and communication links, researchers could also explore how LLMs perform under varied requirement granularity (e.g., detailed user stories versus high-level business goals) and how they handle conflicting or evolving requirements over time. There is an opportunity to study whether LLMs can support interactive, incremental refinement, where an initial draft service decomposition is automatically updated when requirements change. Another avenue is to incorporate domain-specific ontologies or architecture reference models (e.g., sector-specific microservice patterns, event-driven frameworks) so that the model's background knowledge is more tightly constrained to a particular context. Investigating the integration of LLM outputs with formal verification tools, model checkers, or architectural evaluation frameworks (such as ATAM or CBAM) could also yield insights into how AI-aided decomposition might feed into end-to-end quality assurance workflows.

For practitioners, particularly those engaged in legacy system modernization or early-stage design, our approach offers a path to generate plausible design drafts. In ZS, GPT-4 provides a high-recall skeleton of service interactions, capturing most core flows at the expense of extra, heuristic-driven edges that must be pruned. With FS prompts (i.e., requiring only one annotated example), practitioners can eliminate omissions of low-salience components and drastically reduce spurious links. This dual capability allows teams to bootstrap service decompositions directly from requirement documents, user stories, or stakeholder narratives, thereby compressing time-to-design and ensuring closer alignment between business intent and system structure.

In practical settings, embedding these LLM-powered capabilities into architectural modeling environments can transform how teams work. For instance, by integrating a GPT-4-based assistant into collaborative white-boarding tools or low-code platforms, architects and non-technical stakeholders can iteratively refine the generated decomposition through natural language prompts and direct feedback. Additionally, combining LLM-driven decomposition with static code analysis tools can surface discrepancies between the intended architecture and the existing codebase, supporting automated drift detection and suggesting refactoring candidates.

Tool vendors and platform developers should also consider how to incorporate LLM-powered agents within continuous integration/continuous deployment (CI/CD) pipelines. For example, when new user stories are added to a backlog, a pipeline step could generate or update a candidate service boundary map, triggering automated regression tests or contract tests based on the proposed interfaces. This "requirements-to-deployment" traceability could reduce the manual effort required for maintaining consistency across documentation, design, and implementation. In environments subject to regulatory compliance (e.g., healthcare, finance), LLM-generated decomposition—paired with lineage tracking—could help auditors verify that system changes remain within approved boundaries and adhere to security or privacy policies.

Finally, our study underscores a broader, enduring truth: the quality and clarity of input requirements remain critical. When requirement artifacts are vague, ambiguous, or under-specified, even FS GPT-4 outputs can exhibit inconsistencies or overgeneralization (for instance, merging "Order" and "Payment" into a single service under ZS). This observation reinforces the need for better authoring practices (e.g., structured templates, testable acceptance criteria, and semantically rich narratives) to ensure reliable architectural inference. By combining high-quality requirements with minimal exemplar guidance, organizations can leverage LLMs as effective decision-support tools that accelerate architectural decomposition while maintaining fidelity to expert expectations.

6 Threats to Validity

To ensure a comprehensive evaluation of the study, we analyze potential threats to validity following the four commonly adopted dimensions in empirical software engineering: construct validity, internal validity, external validity, and reliability [30].

Construct Validity. We used the systems' actual implemented architectures as our comparison baseline, recognizing that these designs reflect practical trade-offs and developer judgments. To validate their quality and contextualize our results, experts reviewed each reference implementation of the architecture alongside the two GPT-4 variants in a blind evaluation (Section 3.4). This approach ensured experts assessed both real-world and AI-generated designs under the same criteria, and it allowed us to confirm that our baselines were indeed sensible architectural examples. Detailed notes on the implemented decompositions are available in the supplementary material (Artifact Availability Section).

In addition, the evaluation of LLM output was guided by structured criteria (e.g., coverage, correctness, modularity) that may not capture all nuances of architectural quality. To improve interpretability, we complemented automated comparisons with human expert reviews. Experts were presented with the system requirements, the LLM-generated design, and the reference design. Then, they were asked to evaluate architectural plausibility using both quantitative ratings and open-ended feedback. Finally, we acknowledge a potential threat in the "few-shot" variant: the illustrative example provided in the prompt may shape the model's output beyond merely exposing the task format. Since only a single example was used—manually crafted to resemble the domain and complexity of the target systems—its structure and content could unintentionally steer the model's generation behavior. While we aimed to select a representative exemplar, different choices could plausibly yield different outcomes.

Internal Validity. A key risk is the variability of model responses: since LLMs are non-deterministic, repeated executions might yield different decompositions. To minimize this, all model completions were executed with a fixed temperature setting and deterministic decoding, and outputs were saved for reproducibility. We also tested two prompt strategies (ZS and FS) to assess the impact of example conditioning. Although the same requirements were provided across strategies, small changes in phrasing or formatting could affect model behavior. We partially mitigated this by standardizing prompt templates and including all materials in a public repository.

Another threat is the potential bias in the expert evaluation. While experts provided valuable qualitative insights, their assessments may reflect personal preferences or experience with specific architectural designs. We attempted to reduce this threat by recruiting experienced professionals (5+ years) from different backgrounds and anonymizing model outputs to prevent framing effects.

External Validity. This study focused on two relatively small systems with well-documented requirements: PetClinic and BookStore. While these systems reflect realistic use cases, their size and domain may limit generalization to large-scale or safety-critical applications. Furthermore, all prompts were in English, which may not reflect real-world multilingual development environments. To improve external validity, we selected systems from publicly available repositories and extracted or refined requirements to ensure coverage of core business functionalities. Future studies should expand this evaluation to a broader range of systems and domains, possibly incorporating multilingual requirements or cross-organizational workflows.

Reliability. Reliability concerns the replicability of the study. To support reproducibility, we released all datasets, requirements, prompt templates, generated outputs, and evaluation forms in a public repository [2]. Each step of the pipeline—from requirements preparation to architectural extraction and expert assessment—was documented with explicit procedures. However, certain limitations remain. ChatGPT's API (o3 model) may be affected by platform updates or model changes beyond our control. To mitigate this, we recorded model versioning details and captured outputs at a fixed time (May 2025). Additionally, while experts were asked to follow consistent criteria, the authors conducted a qualitative analysis of open responses, which may introduce interpretative bias. We acknowledge this as a trade-off between interpretive richness and independent coding and suggest future work involves third-party reviewers or formal content analysis frameworks.

7 Final Remarks

This study investigated the feasibility of using LLMs, specifically the GPT-4 o3 model, to derive microservice-based architectural descriptions directly from natural language requirements. We evaluated the LLM's ability to identify service boundaries, responsibilities, and interactions across two representative software systems by leveraging zero-shot and few-shot prompting strategies.

In addressing RQ1, whether LLMs can identify appropriate microservice boundaries from textual requirements, we found that GPT-4's zero-shot prompts yielded 14 candidate services per system, eleven of which matched the expected set (precision, recall, and F1 $\approx$ 0.79), while three low-visibility modules were omitted and three extraneous services introduced. Introducing a single exemplar in the prompt (few-shot) increased the generated count to 15—correctly recovering all 14 expected services with only one extra—thereby boosting precision to 0.93, recall to 1.00, and F1 $\approx$ 0.96. These results indicate that while GPT-4's zero-shot outputs reliably capture core domains, minimal exemplar guidance suffices to achieve near-perfect service boundary alignment.

Turning to RQ2, which examines how accurately LLMs recover inter-service communication patterns, a similar improvement is observed. In zero-shot, GPT-4 generated 66 links against 38 expected, correctly recovering 32 of them (six missing, 34 extras), which yields a precision of 0.48, recall of 0.84, and F1 $\approx$ 0.61. By using a few-shot approach, the count of generated links was raised to 50, of which 36 matched the expected 38 (two missing, 14 extras), resulting in precision = 0.72, recall $\approx$0.94, and F1 $\approx$0.82. These findings show that few-shot fills in most missing interactions and significantly curbs the model's over-generation of unsupported connections, producing an interaction topology that more closely mirrors reference architectures.

For RQ3, expert ratings and comments converged on three clear points. First, the few-shot variants were viewed as the most deployable, earning top scores for correctness, completeness, and plausibility because they preserved modular boundaries while covering nearly all requirements. Second, the zero-shot variants, although functionally rich, were judged risky: their dense, bidirectional links undermined fault isolation and drove plausibility scores to the bottom of the scale. Third, the reference variants remained the cleanest in structure but were considered incomplete, highlighting that LLM-generated designs need modest exemplar guidance to balance structural discipline with full domain coverage.

Building on our empirical insights, future research should investigate other prompt engineering techniques—such as dynamic exemplar selection and adaptive few-shot strategies—to further improve service boundary recall and curb spurious link generation. Equally promising is the development of human-in-the-loop workflows that allow architects to iteratively refine LLM-generated decompositions, feeding corrections back into subsequent model invocations to converge on high-quality architectures. Integrating these AI-driven decompositions with traditional static and runtime analysis could enable automated drift detection, highlighting discrepancies between intended and actual service topologies as systems evolve. At the same time, domain-specific fine-tuning of LLMs, coupled with lightweight architectural ontologies, offers a path toward constraining model outputs to industry-relevant design practices and reducing generic over-generation. Finally, longitudinal studies that track the stability of model-suggested architectures across multiple requirement iterations will be crucial to understanding how exemplar-based updates can preserve architectural consistency over time. Pursuing these directions will bring us closer to semantically aware, AI-augmented environments that streamline software modernization, ensure alignment with stakeholder intent, and reduce reliance on manual expertise.

ARTIFACT AVAILABILITY

All artifacts used in this study — including requirements, prompting strategies, architectural outputs, evaluation scripts, and expert review templates — are available to ensure transparency, facilitate replication, and support future research endeavors [2].

ACKNOWLEDGMENTS

This work has been partially funded by the project "iSOP Base: Investigação e desenvolvimento de base arquitetural e tecnológica da Intelligent Sensing Operating Platform (iSOP)" supported by CENTRO DE COMPETÊNCIA EMBRAPII VIRTUS EM HARDWARE INTELIGENTE PARA INDÚSTRIA - VIRTUS-CC, with financial resources from the PPI HardwareBR of the MCTI grant number 055/2023, signed with EMBRAPII.

REFERENCES

[1] Yalemisew Abgaz, Andrew McCarren, Peter Elger, David Solan, Neil Lapuz, Marin Bivol, Glenn Jackson, Murat Yilmaz, Jim Buckley, and Paul Clarke. 2023. Decomposition of monolith applications into microservices architectures: A systematic review. *IEEE Transactions on Software Engineering* 49, 8 (2023), 4213–4242.

[2] Danyllo Albuquerque. 2025. [SBCARS - CBSoft 2025] Toward Generating Microservice Architectures from Textual Requirements with Large Language Models. (8 2025). https://doi.org/10.6084/m9.figshare.29974345.v4

[3] Danyllo Albuquerque, Everton Guimarães, Graziela Tonin, Pilar Rodríguezs, Mirko Perkusich, Hyggo Almeida, Angelo Perkusich, and Ferdinandy Chagas. 2022. Managing technical debt using intelligent techniques-a systematic mapping study. *IEEE Transactions on Software Engineering* 49, 4 (2022), 2202–2220.

[4] Khaled Alsayed and Omar Benomar. 2024. MicroDec: Hybrid Static + LLM Embeddings for Microservice Extraction. *Journal of Systems and Software* 203 (2024), 111622.

[5] Wesley KG Assunção, Luciano Marchezan, Lawrence Arkoh, Alexander Egyed, and Rudolf Ramler. [n. d.]. Contemporary Software Modernization: Strategies, Driving Forces, and Research Opportunities. *ACM Transactions on Software Engineering and Methodology* ([n. d.]).

[6] Mohammadmehdi Ataei, Hyunmin Cheong, Daniele Grandi, Ye Wang, Nigel Morris, and Alexander Tessier. 2025. Elicitron: A Large Language Model Agent-Based Simulation Framework for Design Requirements Elicitation. *Journal of Computing and Information Science in Engineering* 25, 2 (2025).

[7] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[8] Vincent Bushong, Amr S Abdelfattah, Abdullah A Maruf, Dipta Das, Austin Lehman, Eric Jaroszewski, Michael Coffey, Tomas Cerny, Karel Frajtak, Pavel Tisnovsky, et al. 2021. On microservice analysis and architecture evolution: A systematic mapping study. *Applied Sciences* 11, 17 (2021), 7856.

[9] Rafael Capilla, Anton Jansen, Antony Tang, Paris Avgeriou, and Muhammad Ali Babar. 2016. 10 years of software architecture knowledge management: Practice and future. *Journal of Systems and Software* 116 (2016), 191–205.

[10] Rudra Dhar, Karthik Vaidhyanathan, and Vasudeva Varma. 2024. Can llms generate architectural design decisions?-an exploratory empirical study. In *2024 IEEE 21st International Conference on Software Architecture (ICSA)*. IEEE, 79–89.

[11] Jonas Fritzsch, Justus Bogner, Alfred Zimmermann, and Stefan Wagner. 2018. From monolith to microservices: A classification of refactoring approaches. In *International Workshop on Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*. Springer, 128–141.

[12] Matthias Gysel, Lukas Kölbener, Peter Gantenbein, and Olaf Zimmermann. 2016. Service cutter: A systematic approach to service decomposition. In *European Conference on Service-Oriented and Cloud Computing*. Springer, 185–200.

[13] Hossam Hassan, Manal A Abdel-Fattah, and Wael Mohamed. 2024. Migrating from Monolithic to Microservice Architectures: A Systematic Literature Review. *International Journal of Advanced Computer Science & Applications* 15, 10.

[14] Junda He, Christoph Treude, and David Lo. 2024. LLM-Based Multi-Agent Systems for Software Engineering: Literature Review, Vision and the Road Ahead. *ACM Transactions on Software Engineering and Methodology*.

[15] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology* 33, 8 (2024), 1–79.

[16] Mohammad Imranur, Sebastiano Panichella, Davide Taibi, et al. 2019. A curated dataset of microservices-based systems. *arXiv preprint arXiv:1909.03249* (2019).

[17] Jasmin Jahić and Ashkan Sami. 2024. State of Practice: LLMs in Software Engineering and Software Architecture. In *2024 IEEE 21st International Conference on Software Architecture Companion (ICSA-C)*. IEEE, 311–318.

[18] Haolin Jin, Linghan Huang, Haipeng Cai, Jun Yan, Bo Li, and Huaming Chen. 2024. From llms to llm-based agents for software engineering: A survey of current, challenges and future. *arXiv preprint arXiv:2408.02479* (2024).

[19] Munezero Immaculee Joselyne, Gaurav Bajpai, and Frederic Nzanywayingoma. 2021. A systematic framework of application modernization to microservice based architecture. In *2021 International Conference on Engineering and Emerging Technologies (ICEET)*. IEEE, 1–6.

[20] Martin Kaloudis. 2024. Evolving Software Architectures from Monolithic Systems to Resilient Microservices: Best Practices, Challenges and Future Trends. *International Journal of Advanced Computer Science & Applications* 15, 9 (2024).

[21] Goran Mazlami, Johannes Cito, and Philipp Leitner. 2017. Extraction of microservices from monolithic software architectures. In *2017 IEEE International Conference on Web Services (ICWS)*. IEEE, 524–531.

[22] Vinicius L Nogueira, Fernando S Felizardo, Aline MMM Amaral, Wesley KG Assunção, and Thelma E Colanzi. 2024. Insights on Microservice Architecture Through the Eyes of Industry Practitioners. In *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 765–777.

[23] Idris Oumoussa and Rajaa Saidi. 2024. Evolution of microservices identification in monolith decomposition: A systematic review. *IEEE Access* 12 (2024), 23389–23405.

[24] Mirko Perkusich, Lenardo Chaves e Silva, Alexandre Costa, Felipe Ramos, Renata Saraiva, Arthur Freire, Ednaldo Dilorenzo, Emanuel Dantas, Danilo Santos, Kyller Gorgônio, et al. 2020. Intelligent software engineering in the context of agile software development: A systematic literature review. *Information and Software Technology* 119 (2020), 106241.

[25] Yamina Romani, Okba Tibermacine, and Chouki Tibermacine. 2022. Towards migrating legacy software systems to microservice-based architectures: a data-centric process for microservice identification. In *2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C)*. IEEE, 15–19.

[26] Ana Martínez Saucedo, J Andres Diaz-Pace, Hernán Astudillo, and Guillermo Rodríguez. 2024. On the Variability of Microservice Decompositions: A Data-Driven Analysis. In *2024 L Latin American Computer Conference (CLEI)*. IEEE, 1–9.

[27] Ana Martínez Saucedo, Guillermo Rodríguez, Fabio Gomes Rocha, and Rodrigo Pereira dos Santos. 2025. Migration of monolithic systems to microservices: A systematic mapping study. *Information and Software Technology* 177 (2025), 107590.

[28] Robert C Seacord, Daniel Plakosh, and Grace A Lewis. 2003. *Modernizing legacy systems: software technologies, engineering processes, and business practices*. Addison-Wesley Professional.

[29] Khaled Sellami and Mohamed Aymen Saied. 2025. Contrastive Learning-Enhanced Large Language Models for Monolith-to-Microservice Decomposition. *arXiv preprint arXiv:2502.04604* (2025).

[30] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, Anders Wesslén, et al. 2012. *Experimentation in software engineering*. Vol. 236. Springer.

[31] Eberhard Wolff. 2016. *Microservices: flexible software architecture*. Addison-Wesley Professional.

[32] Zibin Zheng, Kaiwen Ning, Qingyuan Zhong, Jiachi Chen, Wenqing Chen, Lianghong Guo, Weicheng Wang, and Yanlin Wang. 2025. Towards an understanding of large language models in software engineering tasks. *Empirical Software Engineering* 30, 2 (2025), 50.