AI-Driven Refactoring: Semantic Reconstruction of Domain Models Using LLM Reasoning

Mohamed El BOUKHARI[1], Nassim KHARMOUM[2], Soumia ZITI[3]
IPSS Research Team-Faculty of Sciences, Mohammed V University in Rabat, Rabat, Morocco[1,3]
IPSS Research Team-Faculty of Sciences, National Center for Scientific and Technical Research (CNRST), Rabat, Morocco[2]

***Abstract*—This study examines the application of large language models (LLMs) for automating domain layer reconstruction in legacy systems, with a specific focus on a case study involving water consumption management. The process begins with a deliberately disordered JSON representation that conflates domain, application, and infrastructure issues. An LLM, specifically GPT-5.2, was employed to identify misplaced methods, inconsistent naming, DTO misuse, incoherent aggregates, and unrelated modules, and subsequently reorganize the model into a structure aligned with Domain-Driven Design (DDD). The structure includes entities, value objects, aggregates, domain services, domain events, and repositories. The methodology involves encoding the legacy model as JSON, applying an LLM-based diagnosis and reconstruction pipeline, and producing both a refined domain model and a categorized catalogue of corrections. A comparative analysis of candidate LLMs, informed by recent code-centric benchmarks, such as SWE-bench and LiveCodeBench, supports the selection of GPT-5.2 as the primary model for this study. The findings indicate that the LLM can swiftly recover key domain concepts and achieve semantically consistent refactoring, a task that typically requires extensive manual effort. This suggests that LLM-assisted domain reconstruction is a promising adjunct to traditional refactoring practices and can facilitate continuous architectural improvements in organizations.**

Keywords*—Domain-Driven Design; large language models; AI-driven software refactoring legacy systems modernization; semantic code analysis; architecture reconstruction; GPT; LLM; domain layer reconstruction; AI-assisted software engineering

I. Introduction

Legacy software systems represent a significant challenge in modern software engineering, where aging codebases continue to serve critical business functions despite outdated architectures and technologies. These systems often lack proper documentation, use obsolete programming languages or frameworks, and contain years of accumulated technical debt, making them difficult and risky to modify. Domain extraction is a crucial refactoring technique that involves identifying and separating distinct business domains or functional areas within monolithic systems to improve maintainability and enable modernization. However, extracting domains from legacy systems presents unique obstacles compared to working with newer codebases, including unclear business logic boundaries, tight coupling between components, and the absence of clear architectural patterns in the code. This process requires careful analysis to avoid introducing bugs or breaking existing functionality, while successfully isolating cohesive business domains that can be independently developed and deployed.

A. Domain Concepts in Modern Software Architecture

The concept of a domain in software engineering pertains to the comprehensive body of knowledge relevant to the problem at hand, encompassing the entire business area or specific support areas where particular categories of software systems are employed [1][2]. More specifically, a domain can be characterized as an area of expertise abstracted from the implementation details of the software, representing the minimal set of properties that accurately delineate a family of problems for which computer applications are necessary [1][5].

The domain layer is central to the domain-centric software architecture, encompassing the core logic of the developed software and serving as a pivotal reference throughout the application [6]. The domain model offers a representation of real-world entities and their interrelationships, which define the problem space of a system. It functions as a communication tool for all stakeholders, facilitating the emergence of concepts, such as ubiquitous language in Domain-Driven Design [3][2].

Domain engineering involves domain analysis, focusing on developing and maintaining reusable infrastructure within specific domains, and domain implementation, which includes application and component engineering [1]. The objective is to enhance domain information accessibility and extract relevant components from repositories rather than building new components. This methodology prioritizes the reusability of the analysis and design, not solely the code.

B. Challenges in Domain Extraction for Refactoring

The task of extracting domains to refactor legacy systems involves unique challenges that hinder the modernization efforts. A key challenge is determining service boundaries within monolithic architectures, which leads to inadequate service decomposition. This affects the system performance, scalability, and maintainability [4]. Legacy applications are not built with service-oriented principles but contain reusable functions that can become services [7].

A significant challenge is the structural disparity between the code organization and domain concepts. Source code rarely indicates the system's high-level components, and syntactical analysis can produce clusters that do not correspond to domain concepts owing to differing organizational patterns. This gap complicates the identification of refactoring opportunities where the benefits are evident [9].

Migration remains largely manual and labor-intensive, with methodologies providing guidance at the architectural level, often overlooking the code-level challenges [8]. When domain models supplement static code analysis, their misalignment with the source code complicates the extraction process [7].

C. AI Domain Knowledge Extraction Capabilities

Modern AI systems effectively derive domain-specific insights from source codes using complementary techniques. Abstract Syntax Trees (ASTs) represent code's structural characteristics, enabling machine learning algorithms to discern syntactic relationships and semantic elements [10][19]. These structures allow AI models to leverage the organized nature of programming languages rather than processing code as linear text [20].

The construction of knowledge graphs is a pivotal capability in the extraction of domain knowledge, where AI systems develop graph representations with nodes that denote code entities and directed edges that define various relationships among program components [14][15]. These knowledge graphs empower AI to model complex inter-function dependencies and cross-file relationships, which are vital for a comprehensive understanding of program behavior [12].

AI systems can derive extensive semantic insights by integrating various analytical methodologies. Through static program analysis, these systems can identify essential elements, such as method overriding relationships, method calls, interface implementations, and type usage patterns [18]. Additionally, more sophisticated systems employ semantic dependency analysis to create semantic networks from imported libraries and public application programming interfaces (APIs). This process aids in the disambiguation of identifier senses, similar to the disambiguation of word senses in natural language processing [11].

The integration of ontological representations establishes a formal framework for modeling software system architecture and semantics, thereby enhancing the clarity and precision of the depiction of system relationships. This approach also facilitates dependency tracking, which is crucial for maintenance and refactoring tasks [17]. These capabilities extend across various applications, including vulnerability prediction, semantic-based code search, code summarization, classification, clone detection, and semantic error identification [13][16].

D. Semantic Analysis

In contemporary AI, advanced neural architectures conduct in-depth semantic analysis of source code beyond basic syntactic parsing. Transformer models, such as CodeBERT and CodeT5, have emerged as tools for semantic understanding, using self-supervised objectives to capture semantic and syntactic details [22][25]. These models interpret both programming and natural language contexts, while operating on raw source codes [22].

Attention mechanisms are a key technique in semantic analysis, enabling models to focus on relevant code segments during the output generation [29]. These mechanisms help AI systems better comprehend the code context in tasks such as variable naming and code summarization. Advanced methods use contrastive learning, comparing positive and negative code samples to improve representation learning and understanding of the target variables [29][23].

AI systems utilize multimodal analysis engines capable of concurrently processing various types of software artifacts, thereby discerning the semantic relationships among code implementations, requirement specifications, test cases, and documentation [21]. This holistic analysis approach considers the entire context of software development activities rather than examining the code in isolation.

Program analysis methods, such as abstract syntax trees (ASTs), control flow graphs (CFGs), and static analysis, help AI systems identify code patterns and suggest improvements [26]. Large language models (LLMs) have been effective in software engineering tasks, including code generation, summarization, clone detection, and automated repair, demonstrating their ability to understand code structure and semantics [28] [24] [27].

E. Planned Contributions and Paper Structure

This study presents three principal contributions: 1) the development of an end-to-end LLM pipeline for reconstructing domain layers from legacy artifacts, which automates the generation of entities/DTOs and the organization of subdomains; 2) the establishment of a semantic classification framework that employs function analysis and schema-driven prompts to ensure precise class placement; and 3) the initiation of an empirical evaluation framework with benchmarks for assessing fidelity, modularity, and scalability within enterprise contexts.

Section I introduces the research problem, outlines the motivation for automating domain reconstruction in legacy systems, and highlights the value of LLM-assisted refactoring. Section II reviews the related work on AI-driven refactoring and Domain-Driven Design extraction. Section III details the methodology, covering the artifact extraction, large language model prompting, and validation processes. Section IV presents a case study of a legacy monolithic system. Section V reports the evaluation results and analysis. Section VI discusses the findings. Section VII presents the limitations and future research directions, and Section VIII concludes the study.

II. Related Work

Cao et al. enhanced early neural methodologies by transitioning from basic networks for concept assignment to the application of LSTMs and Siamese RNNs for entity normalization and the development of knowledge graphs from codes [30]. Kabore et al. and Zhang et al. further refined AST-based approaches, such as ASTNN, by segmenting large trees into statement sequences [31][32], which were subsequently encoded using B-RNNs.

Keller et al. integrated multimodal representations, including lexical tokens, abstract syntax trees (ASTs), program trees and code visuals. They demonstrated the application of WySiWiM in utilizing code screenshots for image classification, achieving a performance comparable to syntax-based methods [31][34].

Zhang et al. (2024), Sachdev et al. (2018), and Ling et al. (2020) advanced the field of code search by transitioning from TF-IDF to the utilization of graph neural networks and attention mechanisms [33][35][37]. Furthermore, Rahman et al. (2023) and Ling et al. (2020) facilitated pattern transfer

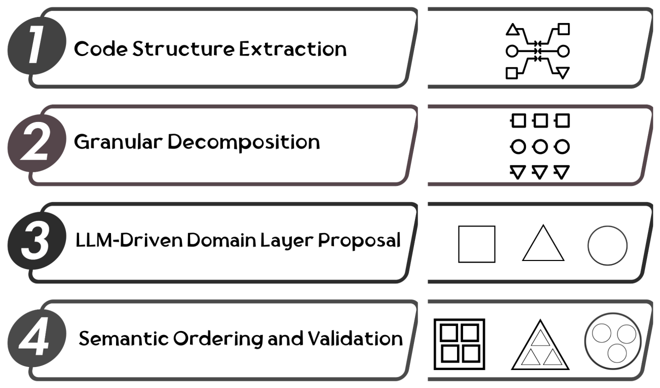
1 Code Structure Extraction
2 Granular Decomposition
3 LLM-Driven Domain Layer Proposal
4 Semantic Ordering and Validation

Fig. 1. Reconstruction pipeline.

in AdaCS by implementing domain embeddings and syntax matching [36][37].

Chowdhury et al. (2025), Chimalakonda et al. (2023), Chaieb et al. (2024), and Al-Debagy (2021) have integrated Abstract Syntax Trees (ASTs), Control Flow Graphs (CFGs), and Data Flow Graphs (DFGs) to capture behavior, thereby extending code2vec embeddings for the extraction of microservices from monolithic architectures [38][44][42][43]. This study directly informs the reconstruction of domain layers from legacy artifacts using large language models (LLMs). These studies underscore the semantic gaps that our LLM schema pipeline addresses in the generation of entities and Data Transfer Objects (DTOs).

A growing body of research explores how AI, and more recently, LLM-based techniques can automate refactoring and testing within software development cycles. Boukhlif et al. showed that intelligent testing approaches can reduce manual validation efforts in safety-critical medical systems [39], while El Boukhari et al. demonstrated AI-assisted refactoring for CQRS-based architectures and healthcare systems using NLP and design patterns to guide structural improvements and enhance testability [41][40]. These contributions support the feasibility of automating substantial portions of refactoring and testing, motivating the use of LLMs as a more semantically capable foundation for domain-driven reconstruction, as pursued in this study.

III. Methodology

The methodology delineates a systematic LLM-driven refactoring pipeline for legacy codebases, as shown in Fig. 1. This process transforms monolithic structures or incomplete domains into semantically coherent, lightweight domain layers.

A. LLM Choice

An advanced large language model serves as the primary reasoning engine, which is chosen for processing structured JSON, deducing domain concepts, and performing semantic transformations on software models. The model is applied throughout the pipeline, ensuring that the analysis, restructuring, and explanation are based on a cohesive semantic understanding. A comparative study of state-of-the-art language models is needed to evaluate their performance on code and architecture tasks to identify the most suitable model.

B. Reconstruction Pipeline

The methodology is delineated into four principal stages that outline the utilization of the LLM in reconstructing the domain layer as follows:

1) Code structure extraction: The initial step involves analyzing a JSON representation of the structure of the legacy codebase, with the assumption that method names convey meaningful semantic information (e.g., “calculateUserDiscount” suggests discount-related logic). This JSON encapsulates hierarchical components, including classes, methods, parameters, dependencies, and call graphs, within a standardized schema: {“classes”: [{“name”: str, “methods”: [{“name”: str, “params”: list, “body_summary”: str, “dependencies”: list}]}]}. Parsing tools such as tree-sitter or AST analyzers are employed to extract this information without modifying the source code, thereby facilitating the subsequent semantic analysis.

2) Granular decomposition: The extracted structure was decomposed into the smallest viable functions, adhering to the single-responsibility principle. Each method is recursively divided by identifying cohesive blocks, such as loops and conditionals, through the semantic clustering of method names and body summaries. Fragments are renamed to preserve context, for example, “processDiscount” is divided into “validateEligibleDiscount” and “computeDiscountRate”. Atomicity is ensured by limiting functions to a single semantic intent, which is verified through validation prompts such as “Is this function doing one thing only?” The output is a refined JSON that contains nested microfunctions.

3) LLM-driven domain layer proposal: By utilizing the decomposed JSON, a large language model (LLM) proposes an innovative domain layer structure that is semantically aligned with the intended purpose of the codebase. An LLM identifies domain concepts and adheres to the principles of domain-driven design (DDD).

4) Semantic ordering and validation: The proposed structure was enhanced using LLM-ordered topological sorting. The prompt validation process involves iterating through simulated refactoring paths, verifying semantic preservation, such as the cosine similarity of name embeddings before and after refactoring, and refining the process using few-shot examples from refactoring datasets. The final output is an executable JSON/YAML scaffold prepared for code generation and maintaining traceability to the original legacy elements. This pipeline improves automation in empirical refactoring studies, thereby reducing the cognitive load on engineers.

IV. Case Study

The case study examined a legacy water management system, as presented in Fig. 2, wherein the domain model was restructured into a JSON representation of the DomainLayer. This JSON format encapsulates the domain concepts in Fig. 3, including clients, water meters, and consumption records, as well as services, value objects, aggregates, and events, while highlighting design and layering issues. The WaterModule delineates the primary domain entities: a client characterized by identification and categorization attributes with a one-to-many relationship with the WaterMeter; a WaterMeter associated with location and linked consumption data; and a

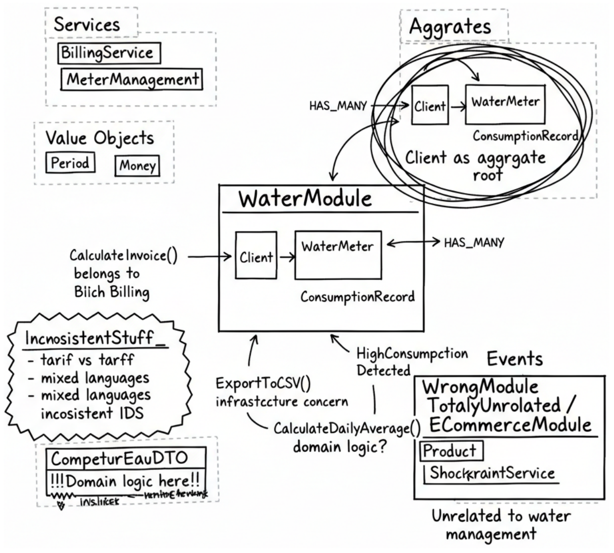

Fig. 2. Water meter system domain.

ConsumptionRecord that details metered usage over billing periods. The representation also enumerates domain services (e.g., BillingService and MeterManagement), value objects (e.g., Period and Money), aggregates rooted in Client, and domain events such as HighConsumptionDetected, offering a structural overview of the system.

The JSON format encodes layering violations and misplaced responsibilities through wrongMethods annotations across entities, services, value objects, aggregates, and events. The Client entity contains invoicing and notification functionalities (CalculateInvoice, SendEmailNotification) that are better suited for billing or infrastructure layers, whereas the WaterMeter entity includes export functionality (ExportToCSV) that is specific to application concerns rather than domain logic. Value objects such as Period and Money contain inappropriate persistence methods, and aggregates and events include messaging behaviors. A mixed layer, InconsistentStuff_MixedLayer, documents naming and language inconsistencies, whereas WrongModule_TotallyUnrelated shows an e-commerce module erroneously integrated with the water domain. This imperfect model serves as an input for the LLM-based approach to identify and realign responsibilities for an improved domain layer architecture.

V. Results

GPT-5.2 was identified as the primary large language model based on the comparative analysis presented in Table I. Among the models evaluated [45][46][47], GPT-5.2 exhibited superior performance on code-centric benchmarks, such as SWE-bench and LiveCodeBench, and excelled in repository-scale reasoning, semantic refactoring and domain-oriented architecture synthesis. These capabilities are directly aligned with the requirements of the proposed method. Although alternative models, including Claude 4.5 Sonnet and Kimi-Dev-72B, were retained as secondary references to contextualize the results, the detailed case study focused on GPT-5.2 to ensure a comprehensive and consistent evaluation.

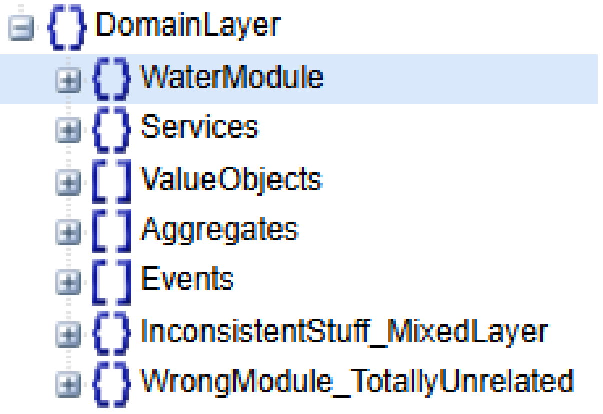

Fig. 3. Initial domain structure.

The LLM initially performs a thorough semantic analysis of the existing JSON domain model, identifying misplaced methods, incorrect or unrelated modules, inconsistent naming conventions, mixed layers, and anti-patterns, such as Data Transfer Objects (DTOs) within the domain, entities containing service logic, value objects exhibiting persistence behavior, incoherent aggregates, and behavioral events. Subsequently, the model is transformed into a Domain-Driven Design (DDD)-aligned Domain Layer by reorganizing the content into canonical building blocks, including entities, value objects, aggregates, domain services, domain events, and repositories. This process involves relocating misplaced methods to the appropriate layers, standardizing naming conventions, and isolating unrelated modules as out-of-scope while maintaining traceability.

The LLM implements a series of targeted corrections that can be categorized into nine distinct areas. Initially, it removes **misplaced methods** from entities, value objects, aggregates, events, and DTOs (e.g., notifications, exports, persistence, webhooks), relocating authentic domain logic into domain services, and documenting technical concerns as out-of-scope infrastructure hints. Subsequently, it isolates **wrong or unrelated modules**, notably extracting the e-commerce module into a separate "UnrelatedModules" section with a clear rationale. **Naming conventions** are standardized by aligning identifiers, field names, value-object attributes, categories, and types to consistent English, PascalCase, and coherent ID conventions. To **address mixed layers**, infrastructure responsibilities (email, SMS, webhooks, CSV/PDF, persistence) are removed from the domain, with repository interfaces introduced to replace the entity-level data access methods. **DTOs are relocated** to a dedicated DTO section and stripped of domain logic, while **entities are cleaned** of calculation, notification, and retrieval behavior, which is centralized into dedicated domain services such as billing and meter management. **Value objects** are restored to immutability and behavioral purity by eliminating persistence-related operations. Aggregates are clarified so that a Client aggregate coherently contains WaterMeter, ConsumptionRecord, and Invoice, a WaterMeter aggregate encapsulates ConsumptionRecord, and all aggregate behaviors are removed. Finally, **domain events** are restricted to payload and description only, with behavioral aspects (e.g., webhooks) relocated to the appropriate application or integration layers.

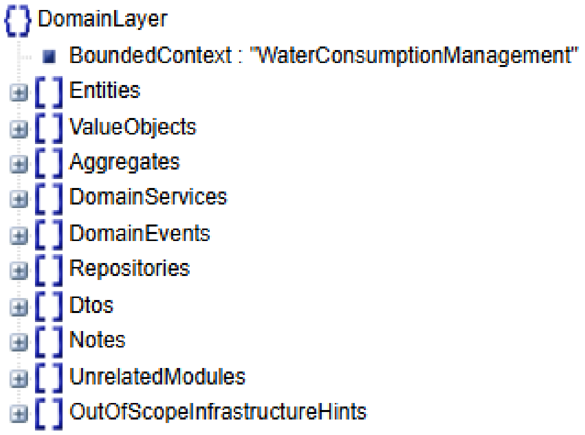
DomainLayer
BoundedContext : "WaterConsumptionManagement"
[ ] Entities
[ ] ValueObjects
[ ] Aggregates
[ ] DomainServices
[ ] DomainEvents
[ ] Repositories
[ ] Dtos
[ ] Notes
[ ] UnrelatedModules
[ ] OutOfScopeInfrastructureHints

Fig. 4. Refactored domain tree view.

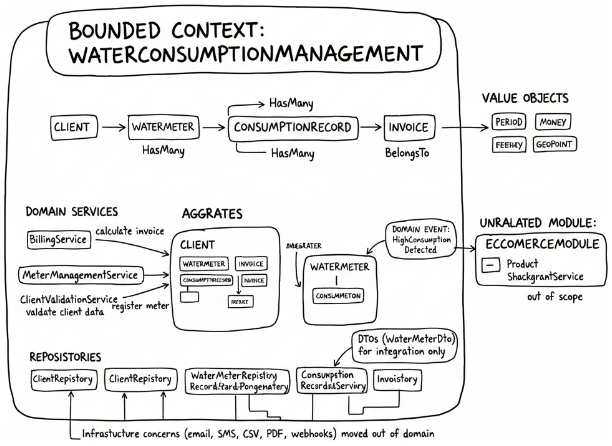

Fig. 5. Water meter system: new refactored domain.

The two figures present the LLM-driven reconstruction of the domain layer. Fig. 4 shows a refactored JSON tree view of the DomainLayer, with a cleaned bounded context and separated core domain elements (entities, value objects, aggregates, services, events, and repositories). Fig. 5 shows the domain architecture as a conceptual diagram, depicting the WaterConsumptionManagement bounded context with its main entities (Client, WaterMeter, ConsumptionRecord, Invoice), aggregates, value objects, and domain services, illustrating the final DDD-aligned structure.

The reconstructed model effectively illustrates the capability of large language models (LLMs) to rapidly identify key domain components with minimal guidance from humans. From the unstructured JSON data, the model consistently identified the central entities—Client, WaterMeter, ConsumptionRecord, and Invoice—along with their associated value objects, such as Period, Money, and GeoPoint. Furthermore, it recognized primary behavioral anchors, including billing and meter management services, even when these concepts were dispersed across various layers and inconsistent naming conventions. This automatic detection and clustering of core domain elements provide a robust foundation for architects, enabling them to refine aggregates and responsibilities rather than manually rediscovering the fundamental structure of a domain.

In a single execution, the LLM adeptly generated both the refined domain model and a list of necessary corrections. In contrast, the manual process of refactoring a domain layer of equivalent complexity generally requires a considerably greater investment of time from architects and domain experts.

VI. Discussion

The selection of GPT-5.2 as the primary model influenced the validity and generalizability of this study. Models in the GPT-5 class achieve state-of-the-art results on code benchmarks, such as SWE-bench and LiveCodeBench [45][46][47], which assess repository-scale reasoning and multi-file modification. These capabilities align with the requirements of this study, where the LLM must reason over JSON representations and identify architectural issues to reconstruct domain models. Utilizing this tool allows for the observation of the maximum potential of current LLM technology for domain-driven refactoring. Claude 4.5 Sonnet and Kimi-Dev-72B provide context within the broader model landscape, although the absence of a full parallel analysis limits cross-model conclusions. Future research should test multiple LLMs to evaluate how the choice of model affects the reconstruction quality.

The LLM exhibits two-phase behavior, aligned with the evidence of LLM-based refactoring capabilities [48][49]. Initially, it performs semantic analysis of the JSON domain model, detecting misplaced methods, unrelated modules, inconsistent naming, mixed layers, and anti-patterns, including embedded DTOs, entities with service logic, and events containing behavior. Studies have confirmed that LLMs can identify code smells and refactoring needs across code bases, particularly for recurring patterns. In the second phase, the LLM reorganizes the model into a DDD-aligned Domain Layer by mapping elements to canonical building blocks, relocating methods, normalizing names, and isolating unrelated modules while maintaining traceability. This aligns with research showing that LLMs effectively enforce architectural rules and layered separations when explicitly prompted.

The findings suggest that the LLM can execute a substantial, semantically coherent restructuring of the domain model in a significantly reduced timeframe compared to that typically required by human architects. Empirical research on LLM-based refactoring and automated tools demonstrates that these systems are particularly adept at addressing systematic, repetitive issues and can produce extensive refactoring suggestions at a much faster rate than developers [48], who may require several hours or days to implement comparable changes in nontrivial systems.

The reconstructed model effectively illustrates the capability of large language models to rapidly surface key domain components with limited human supervision. From the unstructured and inconsistent JSON, the LLM consistently recovered the central entities (Client, WaterMeter, ConsumptionRecord, Invoice), their associated value objects (Period, Money, GeoPoint), and the main behavioral anchors related to billing and meter management, despite these concepts being scattered across layers and expressed with heterogeneous naming conventions. This aligns with recent empirical work

TABLE I. Comparison of LLMs for Legacy Code Refactoring and Domain Modeling

Aspect	GPT-5.2 (OpenAI)	Claude 4.5 Sonnet (Anthropic)	Kimi-Dev-72B (Open-source)
Code benchmarks (e.g., SWE-bench, LiveCodeBench)	Among the top scores on SWE-bench and LiveCodeBench, repository-scale reasoning and transformation were very strong.	Competitive on coding and reasoning; slightly below frontier models on the hardest coding tasks.	Best open-source scores on SWE-bench Verified (around 60%); below frontier closed models but strong for OSS.[1]
Large-repository understanding	128k–200k+ context window; robust handling of mixed code and documentation for architecture tasks.	Very strong long-context reasoning (up to approximately 200k tokens); effective for narrative architecture analysis.	They have good context and stability but are more limited; large codebases typically require chunking strategies.
Refactoring and decomposition	It excels at stepwise refactoring, function extraction, and semantic renaming using chain-of-thought prompting.	Strong in explaining and refactoring code but slightly less aggressive by default on structural transformations.	Good at classic refactorings; more difficulty with deep semantic decomposition without careful prompting.
Domain and architecture modeling	It is very good at proposing DDD-like aggregates, domain layers, and JSON/YAML blueprints from the code structure.	Excellent at high-level conceptual modeling and textual rationale and strong for architecture narratives and trade-offs.	It can infer domains, but abstractions are less consistent and require more prompt engineering.
Tooling and ecosystem	Mature APIs and assistants for coding, IDE integration, CI, and code-review workflows.	Solid integration of conversational and analysis-centric workflows, including documentation and reviews.	Full local control and good OSS tooling integration, but higher DevOps and infrastructure overheads.
Openness and reproducibility	Closed weights: experiments reproducible via documented prompts and configs, but not locally hostable.	Closed model with reproducibility constraints similar to those of other proprietary systems.	Open weights; well-suited for fully reproducible, self-hosted experimental pipelines in research.

[51][50][5][48], which shows that LLMs can reliably extract domain concepts and modular structures from existing code and documentation, often matching or approaching human performance in tasks such as entity discovery, feature localization, and architectural view reconstruction, while requiring significantly less time. In practice, this automatic detection and clustering of core domain elements provides a strong starting point for architects, who can concentrate on refining aggregates, invariants, and responsibility boundaries instead of manually rediscovering the fundamental structure of the domain—a role for LLMs that is also emphasized in studies on AI-augmented software architecture and DDD refactoring in industrial contexts.

Although this study does not explicitly quantify cost or time, the capability of the LLM to generate a coherent domain reconstruction in a single iteration implies that it can potentially supplant numerous hours of detailed refactoring work typically undertaken by multiple developers. In practical applications, developers' efforts can be predominantly concentrated on validating and refining the LLM's proposal rather than executing the restructuring itself [51][48][52]. This shift in focus reduces the manual workload and, indirectly, the staffing and time costs that are generally associated with large-scale refactoring projects.

VII. Limitations and Future Directions

A. Limitations

Despite promising outcomes, several challenges persist in the utilization of LLMs for domain-layer reconstruction. The model's decision-making process remains partially opaque: while it identifies recurring anti-patterns (e.g., misplaced methods), its rationale for nuanced design decisions, such as determining aggregate boundaries or allocating responsibilities between services, can be difficult to audit. The performance of LLMs is sensitive to prompt design and input representation; changes in the JSON structure or constraint phrasing may yield different reconstructions, raising concerns about stability compared to traditional refactoring approaches. Current models struggle with implicit domain knowledge and non-obvious invariants that are not explicitly encoded in the inputs, potentially oversimplifying business rules, as noted in studies of AI-augmented software architecture and DDD refactoring.

B. Future Direction

AI-driven refactoring is evolving towards more autonomous and intelligent systems that function as collaborative partners rather than suggestion tools.

Agentic coding tools represent a significant advancement in the role of artificial intelligence in refactoring processes, acting as collaborators that autonomously plan, execute, test, and refine complex development tasks with minimal supervision. Despite this potential, current agentic refactoring is largely limited to low-level, consistency-driven modifications, such as renaming and type adjustments, with agents performing fewer high-level design transformations than human efforts. To realize the potential of agents as "software architects", substantial progress is required to enable autonomous architecturally aware restructuring that addresses higher-level design challenges.

Future advancements are anticipated to focus on developing advanced AI-driven tools that effectively assist in both development and maintenance processes. These tools include automated debugging assistants, refactoring advisors, code summarizers, and documentation generators. The design of these tools must consider complex factors, such as historical usage patterns, module dependencies, backward compatibility requirements, and runtime constraints. Therefore, it is crucial for these tools to integrate seamlessly with developer workflows, including version control systems, CI/CD pipelines and issue trackers.

The field is advancing towards the development of more sophisticated automated refactoring tools capable of proposing specific refactoring strategies. These tools may, for instance, suggest the division of large modules into smaller, interconnected units with enhanced cohesion, with AI systems potentially executing these recommendations autonomously (Khanzadeh et al., 2023). Future developments are expected to include the advent of domain-specific AI models, increased transparency of AI-driven solutions, and collaborative tools designed to integrate refined and updated development methodologies.

VIII. Conclusion

This study illustrates the effective application of advanced large language models (LLMs) in automating the semantic reconstruction of domain models from legacy software artifacts. Through the implementation of a structured LLM-driven pipeline, this approach adeptly identifies and reorganizes key domain elements, rectifies architectural inconsistencies, and aligns legacy codebases with Domain-Driven Design principles. This automation substantially diminishes the manual effort traditionally associated with domain extraction and refactoring, facilitating a more rapid, consistent, and semantically coherent modernization of legacy systems. A case study on a sample water management system demonstrates the method's capability to detect misplaced responsibilities, standardize naming conventions, and isolate unrelated modules while maintaining traceability. Future advancements in AI-driven refactoring hold the promise of increasingly autonomous and intelligent tools that will further transform software refactoring into an adaptive, explainable, and efficient process, fostering collaboration between human architects and AI agents.

References

[1] F. Losavio, A. Matteo, and I. Pacilli Camejo, "Unified Process for Domain Analysis integrating Quality, Aspects and Goals," CLEIej, vol. 17, no. 2, Aug. 2014, doi: 10.19153/cleiej.17.2.1.

[2] H. S. Da Silva, G. Carneiro, and M. Monteiro, "Towards a Roadmap for the Migration of Legacy Software Systems to a Microservice based Architecture:," in Proceedings of the 9th International Conference on Cloud Computing and Services Science, Heraklion, Crete, Greece: SCITEPRESS - Science and Technology Publications, 2019, pp. 37–47. doi: 10.5220/0007618400370047.

[3] S. Meliá, R. Reyes, and C. Cachero, "The Effect of Developers' General Intelligence on the Understandability of Domain Models: An Empirical Study," IEEE Access, vol. 11, pp. 70153–70167, 2023, doi: 10.1109/ACCESS.2023.3293199.

[4] I. Trabelsi, B. Mahmoudi, J. B. Minani, N. Moha, and Y.-G. Guéhéneuc, "A Systematic Literature Review of Machine Learning Approaches for Migrating Monolithic Systems to Microservices," Aug. 21, 2025, arXiv: arXiv:2508.15941. doi: 10.48550/arXiv.2508.15941.

[5] O. Özkan, Ö. Babur, and M. Van Den Brand, "Refactoring with domain-driven design in an industrial context: An action research report," Empir Software Eng, vol. 28, no. 4, p. 94, July 2023, doi: 10.1007/s10664-023-10310-1.

[6] S. Flaga and K. Pacholczak, "Demonstrator of a Digital Twin for Education and Training Purposes as a Web Application," Adv. Sci. Technol. Res. J., vol. 16, no. 5, pp. 110–119, Nov. 2022, doi: 10.12913/22998624/152927.

[7] H. Mili et al., "Service-Oriented Re-engineering of Legacy JEE Applications: Issues and Research Directions," June 03, 2019, arXiv: arXiv:1906.00937. doi: 10.48550/arXiv.1906.00937.

[8] R. Peixoto, F. F. Correia, T. Rosa, E. Guerra, and A. Goldman, "Refactoring Towards Microservices: Preparing the Ground for Service Extraction," Oct. 03, 2025, arXiv: arXiv:2510.03050. doi: 10.48550/arXiv.2510.03050.

[9] D. van der Leij, J. Binda, R. van Dalen, P. Vallen, Y. Luo, and M. Aniche, "Data-Driven Extract Method Recommendations: A Study at ING," July 22, 2021, arXiv: arXiv:2107.05396. doi: 10.48550/arXiv.2107.05396.

[10] E. Spirin, E. Bogomolov, V. Kovalenko, and T. Bryksin, "PSIMiner: A Tool for Mining Rich Abstract Syntax Trees from Code," Mar. 23, 2021, arXiv: arXiv:2103.12778. doi: 10.48550/arXiv.2103.12778.

[11] A. Saeidi, J. Hage, R. Khadka, and S. Jansen, "On the Effect of Semantically Enriched Context Models on Software Modularization," Programming, vol. 2, no. 1, p. 2, Aug. 2017, doi: 10.22152/programming-journal.org/2018/2/2.

[12] S. Khemka and M. Architect, "Legacy Modernization with AI - Mainframe modernization".

[13] L. Buch and A. Andrzejak, "Learning-Based Recursive Aggregation of Abstract Syntax Trees for Code Clone Detection," in 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), Hangzhou, China: IEEE, Feb. 2019, pp. 95–104. doi: 10.1109/SANER.2019.8668039.

[14] A. Qayum, S. U. R. Khan, Inayat-Ur-Rehman, and A. Akhunzada, "FineCodeAnalyzer: Multi-Perspective Source Code Analysis Support for Software Developer Through Fine-Granular Level Interactive Code Visualization," IEEE Access, vol. 10, pp. 20496–20513, 2022, doi: 10.1109/ACCESS.2022.3151395.

[15] J. Zhang, R. Xie, W. Ye, Y. Zhang, and S. Zhang, "Exploiting Code Knowledge Graph for Bug Localization via Bi-directional Attention," in Proceedings of the 28th International Conference on Program Comprehension, Seoul Republic of Korea: ACM, July 2020, pp. 219–229. doi: 10.1145/3387904.3389281.

[16] Z. Bilgin, "Code2Image: Intelligent Code Analysis by Computer Vision Techniques and Application to Vulnerability Prediction," May 07, 2021, arXiv: arXiv:2105.03131. doi: 10.48550/arXiv.2105.03131.

[17] A. S. Da Silva, R. E. Garcia, and L. C. Botega, "Bug Localization Model in Source Code Using Ontologies," IEEE Access, vol. 11, pp. 98542–98557, 2023, doi: 10.1109/ACCESS.2023.3313598.

[18] University of Oviedo, Computer Science Department, c/Calvo Sotelo s/n, 33007, Oviedo, Spain, F. Ortin, J. Escalada, and O. Rodriguez-Prieto, "Big Code: New Opportunities for Improving Software Construction," JSW, vol. 11, no. 11, pp. 1083–1008, Nov. 2016, doi: 10.17706/jsw.11.11.1083-1088.

[19] Y. Shido, Y. Kobayashi, A. Yamamoto, A. Miyamoto, and T. Matsumura, "Automatic Source Code Summarization with Extended Tree-LSTM," June 20, 2019, arXiv: arXiv:1906.08094. doi: 10.48550/arXiv.1906.08094.

[20] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "A General Path-Based Representation for Predicting Program Properties," Apr. 22, 2018, arXiv: arXiv:1803.09544. doi: 10.48550/arXiv.1803.09544.

[21] M. Hariharan, "Semantic mastery- Enhancing LLMs with Advanced Natural Language Understanding".

[22] S. Nursapa, A. Samuilova, A. Bucaioni, and P. T. Nguyen, "ROSE: Transformer-Based Refactoring Recommendation for Architectural Smells," July 20, 2025, arXiv: arXiv:2507.12561. doi: 10.48550/arXiv.2507.12561.

[23] H. Liu et al., "RefBERT: A Two-Stage Pre-trained Framework for Automatic Rename Refactoring," May 28, 2023, arXiv: arXiv:2305.17708. doi: 10.48550/arXiv.2305.17708.

[24] N. Jiang, T. Lutellier, and L. Tan, "CURE: Code-Aware Neural Machine Translation for Automatic Program Repair," in 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), May 2021, pp. 1161–1173. doi: 10.1109/ICSE43902.2021.00107.

[25] Z. Feng et al., "CodeBERT: A Pre-Trained Model for Programming and Natural Languages," Sept. 18, 2020, arXiv: arXiv:2002.08155. doi: 10.48550/arXiv.2002.08155.

[26] M. R. Tapader, M. M. Rahman, A. I. Shiplu, M. F. I. Amin, and Y. Watanobe, "Code Refactoring with LLM: A Comprehensive Evaluation With Few-Shot Settings," Nov. 26, 2025, arXiv: arXiv:2511.21788. doi: 10.48550/arXiv.2511.21788.

[27] E. Mashhadi and H. Hemmati, "Applying CodeBERT for Automated Program Repair of Java Simple Bugs," Mar. 30, 2021, arXiv: arXiv:2103.11626. doi: 10.48550/arXiv.2103.11626.

[28] S. Afrin, M. Z. Haque, and A. Mastropaolo, "A Systematic Literature Review of Parameter-Efficient Fine-Tuning for Large Code Models," Aug. 15, 2025, arXiv: arXiv:2504.21569. doi: 10.48550/arXiv.2504.21569.

[29] B. Nyirongo, Y. Jiang, H. Jiang, and H. Liu, "A Survey of Deep Learning Based Software Refactoring".

[30] K. Cao and J. Fairbanks, "Unsupervised Construction of Knowledge Graphs From Text and Code," Aug. 25, 2019, arXiv: arXiv:1908.09354. doi: 10.48550/arXiv.1908.09354.

[31] A. K. Kaboré, E. T. Barr, J. Klein, and T. F. Bissyandé, "CodeGrid: A Grid Representation of Code," in Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, Seattle WA USA: ACM, July 2023, pp. 1357–1369. doi: 10.1145/3597926.3598141.

[32] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A Novel Neural Source Code Representation Based on Abstract Syntax Tree," in 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), Montreal, QC, Canada: IEEE, May 2019, pp. 783–794. doi: 10.1109/ICSE.2019.00086.

[33] F. Zhang, M. Li, H. Wu, and T. Wu, "Intelligent code search aids edge software development," J Cloud Comp, vol. 13, no. 1, p. 78, Apr. 2024, doi: 10.1186/s13677-024-00629-5.

[34] P. Keller, L. Plein, T. F. Bissyandé, J. Klein, and Y. L. Traon, "What You See is What it Means! Semantic Representation Learning of Code based on Visualization and Transfer Learning," Feb. 07, 2020, arXiv: arXiv:2002.02650. doi: 10.48550/arXiv.2002.02650.

[35] S. Sachdev, H. Li, S. Luan, S. Kim, K. Sen, and S. Chandra, "Retrieval on source code: a neural code search," in Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, Philadelphia PA USA: ACM, June 2018, pp. 31–41. doi: 10.1145/3211346.3211353.

[36] M. M. Rahman, Y. Watanobe, A. Shirafuji, and M. Hamada, "Exploring Automated Code Evaluation Systems and Resources for Code Analysis: A Comprehensive Survey," July 08, 2023, arXiv: arXiv:2307.08705. doi: 10.48550/arXiv.2307.08705.

[37] X. Ling et al., "Deep Graph Matching and Searching for Semantic Code Retrieval," ACM Trans. Knowl. Discov. Data, vol. 15, no. 5, pp. 1–21, Oct. 2021, doi: 10.1145/3447571.

[38] J. M. Chowdhury, A. F. S. Chowdhury, H. B. Monwar, and M. Naznin, "ATLAS: Automated Tree-based Language Analysis System for C and C++ source programs," Dec. 14, 2025, arXiv: arXiv:2512.12507. doi: 10.48550/arXiv.2512.12507.

[39] Boukhlif, M., Kharmoum, N., Hanine, M., Elasri, C., Rhalem, W., & Ezziyyani, M. (2024). Exploring the application of classical and intelligent software testing in medicine: A literature review. In the Proceedings of the International Conference on Advanced Intelligent Systems for Sustainable Development (pp. 37-46). Springer, Cham. DOI: 10.1007/978-3-031-52388-5-4

[40] M. EL Boukhari, N. Kharmoum, S. N. Lagmiri, and S. Ziti, "Enhancing CQRS Implementation Through AI-Driven Refactoring: Query and Command Classification," in Smart Business and Technologies, S. N. Lagmiri, M. Lazaar, and F. M. Amine, Eds., Cham: Springer Nature Switzerland, 2025, pp. 384–390.

[41] M. El Boukhari, S. Retal, N. Kharmoum, F. Saoiabi, S. Ziti, and W. Rhalem, "An Approach for Refactoring System Healthcare Using CQRS, GoF, and Natural Language Processing," in International Conference on Advanced Intelligent Systems for Sustainable Development (AI2SD'2023), M. Ezziyyani, J. Kacprzyk, and V. E. Balas, Eds., Cham: Springer Nature Switzerland, 2024, pp. 47–55.

[42] M. chaieb and M. A. Saied, "Migration to Microservices: A Comparative Study of Decomposition Strategies and Analysis Metrics," Feb. 13, 2024, arXiv: arXiv:2402.08481. doi: 10.48550/arXiv.2402.08481.

[43] O. Al-Debagy and P. Martinek, "A Microservice Decomposition Method Through Using Distributed Representation of Source Code," SCPE, vol. 22, no. 1, pp. 39–52, Feb. 2021, doi: 10.12694/scpe.v22i1.1836.

[44] S. Chimalakonda, D. Das, A. Mathai, S. Tamilselvam and A. Kumar, "The Landscape of Source Code Representation Learning in AI-Driven Software Engineering Tasks," 2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), Melbourne, Australia, 2023, pp. 342-343, doi: 10.1109/ICSE-Companion58688.2023.00098. keywords: Surveys;Codes;Source coding;Pipelines;Semantics;Tutorials;Syntactics;Code Representation;Machine Learning,

[45] R. Aleithan, H. Xue, M. M. Mohajer, E. Nnorom, G. Uddin, and S. Wang, "SWE-Bench+: Enhanced Coding Benchmark for LLMs," Oct. 10, 2024, arXiv: arXiv:2410.06992. doi: 10.48550/arXiv.2410.06992.

[46] N. Jain et al., "LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code," June 06, 2024, arXiv: arXiv:2403.07974. doi: 10.48550/arXiv.2403.07974.

[47] Z. Yang et al., "Kimi-Dev: Agentless Training as Skill Prior for SWE-Agents," Dec. 08, 2025, arXiv: arXiv:2509.23045. doi: 10.48550/arXiv.2509.23045.

[48] B. Liu, Y. Jiang, Y. Zhang, N. Niu, G. Li, and H. Liu, "An Empirical Study on the Potential of LLMs in Automated Software Refactoring," Nov. 07, 2024, arXiv: arXiv:2411.04444. doi: 10.48550/arXiv.2411.04444.

[49] R. Morales, F. Khomh, and G. Antoniol, "RePOR: Mimicking humans on refactoring tasks. Are we there yet?," May 17, 2019, arXiv: arXiv:1808.04352. doi: 10.48550/arXiv.1808.04352.

[50] M. Robredo, M. Esposito, F. Palomba, R. Peñaloza, and V. Lenarduzzi, "What Were You Thinking? An LLM-Driven Large-Scale Study of Refactoring Motivations in Open-Source Projects," Sept. 09, 2025, arXiv: arXiv:2509.07763. doi: 10.48550/arXiv.2509.07763.

[51] "AI-Augmented Software Architecture: Autonomous Refactoring with Design Pattern Awareness," IJETCSIT, vol. 6, 2025, doi: 10.63282/3050-9246.IJETCSIT-V6I3P113.

[52] S. Abrahão, J. Grundy, M. Pezzè, M.-A. Storey, and D. A. Tamburri, "Software Engineering by and for Humans in an AI Era," ACM Trans. Softw. Eng. Methodol., vol. 34, no. 5, pp. 1–46, June 2025, doi: 10.1145/3715111.