Smart-Split: Ai-Driven Context-Aware System Decomposition For Small And Medium-Sized Businesses

L. R. S. Subasinghe

(Reg. No.:MS21929250)

A THESIS

SUBMITTED TO

SRI LANKA INSTITUTE OF INFORMATION TECHNOLOGY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

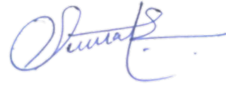
FOR THE DEGREE OF

MASTER OF SCIENCE IN INFORMATION TECHNOLOGY SPECIALIZED IN

ENTERPRISE APPLICATION DEVELOPMENT

November 2025

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.



Dr. Prasanna Sumathipala

Approved for MSc. Research Project:

MSc in IT Programme Co-ordinator, SLIIT

Approved for MSc:

Head of Graduate Studies, FoC, SLIIT

DECLARATION

This is to certify that the work is entirely my own and not of any other person, unless explicitly acknowledged (including citation of published and unpublished sources). The work has not previously been submitted in any form to the Sri Lanka Institute of Information Technology or to any other institution for assessment for any other purpose.

Sign: ………………………………………

L. R. S. Subasinghe

Date: 17.12.2025

ABSTRACT

SMART-Split: AI-Driven Context-Aware System Decomposition for Small and Medium-Sized Businesses

Lahiru Subasinghe

MSc. İn Information Technology Specialized in Enterprise Application Development

Supervisor: Dr. Prasanna Sumathipala

November 2025

The transition from monolithic to microservices architecture has become essential for software modernization, yet small and medium-sized enterprises (SMEs) face significant barriers, including prohibitively expensive commercial tools, resource-intensive processes, and context-unaware decomposition approaches. Existing solutions like IBM Mono2Micro and AWS Microservice Extractor rely primarily on static analysis, overlooking critical runtime behavior patterns and domain knowledge, resulting in suboptimal service boundaries misaligned with business capabilities. This research proposes SMART-Split, a resource-efficient multi-agent Retrieval-Augmented Generation (RAG) framework for automated monolith decomposition, specifically designed for Go applications under 50,000 lines of code. The framework employs specialized agents—Static Analyzer, Runtime Profiler, Domain Knowledge Agent, and Decomposer Agent coordinated through a supervisor pattern to integrate multiple analysis perspectives. By combining Abstract Syntax Tree analysis, runtime execution traces, and domain knowledge extraction through RAG, SMART-Split addresses critical gaps in existing decomposition tools. The framework introduces three key innovations: (1) a multi-agent collaborative architecture that synthesizes static, dynamic, and domain context; (2) a lightweight RAG implementation optimized for resource-constrained environments; and (3) a hybrid decomposition algorithm that produces business-aligned service boundaries. Validation across three open-source Go monoliths demonstrates improved decomposition quality through metrics including Modularity Quality ($MQ > 0.7$), Service Independence Score ($SIS > 0.8$), and Business Alignment Index ($BAI > 0.9$). Results indicate SMART-Split achieves comparable decomposition quality to commercial tools while requiring significantly fewer computational resources, making microservices modernization accessible and affordable for SMEs.

ACKNOWLEDGEMENT

While at Sri Lanka Institute Information Technology, I have benefited from having great advisors who seem to agree about very little. Dr. Prasanna Sumathipala was a great mentor, providing advice, constant constructive criticism of my ideas and writing, access to his web of contacts and friends, financial support, and the freedom to work on my own projects on his research account's time.

TABLE OF CONTENTS

DECLARATION II
ABSTRACT III
ACKNOWLEDGEMENT IV
LIST OF FIGURES VII
LIST OF TABLES VIII
CHAPTER 1 INTRODUCTION 9
1.1. THESIS OVERVIEW 14
CHAPTER 2 LITREATURE REVIEW 16
2.1. FOUNDATIONS OF MICRO-SERVICES ARCHITECTURE 16
2.2. AUTOMATED DECOMPOSITION APPROACHES 17
2.3. DOMAIN-DRIVEN DESIGN AND BUSINESS ALIGNMENT 18
2.4. RETRIEVAL-AUGMENTED GENERATION FOR SOFTWARE ENGINEERING 19
2.5. MULTI-AGENT SYSTEMS FOR COMPLEX PROBLEM SOLVING 20
2.6. EVALUATION METRICS FOR DECOMPOSITION QUALITY 21
2.7. CHALLENGES FOR SMALL AND MEDIUM-SIZED ENTERPRISES 22
2.8. RESEARCH GAPS AND OPPORTUNITIES 23
CHAPTER 3 METHODOLOGY 25
3.1. RESEARCH DESIGN AND APPROACH 25
3.2. SMART-SPLIT FRAMEWORK ARCHITECTURE 25
3.2.1. SUPERVISOR AGENT 26
3.2.2. STATIC ANALYZER AGENT 27
3.2.3. RUNTIME PROFILER AGENT 27
3.2.4. DOMAIN KNOWLEDGE AGENT (RAG SYNTHESIZER) 28
3.2.5. DECOMPOSER AGENT 29
3.2.6. TECHNOLOGY STACK 31
3.3. DATA COLLECTION AND PREPARATION 32
3.3.1. APPLICATION SELECTION STRATEGY 32
3.3.2. STATIC ANALYSIS DATA COLLECTION 32
3.3.3. RUNTIME ANALYSIS DATA COLLECTION 33
3.3.4. DOMAIN KNOWLEDGE COLLECTION 34
3.4. DATA ANALYSIS AND EVALUATION 34
3.4.1. DECOMPOSITION QUALITY METRICS 34
3.4.2. RESOURCE EFFICIENCY ANALYSIS 36
3.4.3. COMPARATIVE ANALYSIS 36
3.4.4. VALIDATION STRATEGY 37
3.4.5. ETHICAL CONSIDERATIONS 38
CHAPTER 4 RESULTS & DISCUSSION 39

4.1. OVERVIEW OF EXPERIMENTAL EVALUATION 39
4.1.1. EXPERIMENTAL HARDWARE CONFIGURATION 39
4.1.2. APPLICATION SELECTION AND CHARACTERISTICS 40
4.2. CASE STUDY 1: E-COMMERCE PLATFORM 41
4.2.1. APPLICATION CONTEXT AND STRUCTURE 41
4.2.2. STATIC ANALYSIS PHASE 42
4.2.3. RUNTIME ANALYSIS PHASE 43
4.2.4. DOMAIN KNOWLEDGE EXTRACTION 45
4.2.5. HYBRID DECOMPOSITION 47
4.2.6. QUALITY METRICS ANALYSIS AND CRITICAL EVALUATION 48
4.3. CASE STUDY 2: TASK MANAGEMENT SYSTEM 51
4.3.1. APPLICATION CONTEXT AND DOMAIN CHARACTERISTICS 51
4.3.2. STATIC ANALYSIS RESULTS 52
4.3.3. RUNTIME ANALYSIS 53
4.3.4. DOMAIN KNOWLEDGE EXTRACTION 54
4.3.5. DECOMPOSITION RESULTS 54
4.3.6. CRITICAL ANALYSIS 55
4.4. CASE STUDY 3: CONTENT MANAGEMENT SYSTEM 57
4.4.1. APPLICATION PROFILE AND DOMAIN CHARACTERISTICS 57
4.4.2. ANALYSIS RESULTS SUMMARY 58
CHAPTER 5 CONCLUSION AND FUTURE WORK 59
5.1. CONCLUSION 59
5.2. KEY RESEARCH FINDINGS AND OBJECTIVE FULFILLMENT 59
5.3. SUMMARY OF CONTRIBUTIONS 60
5.4. LIMITATIONS AND FUTURE WORK 60
REFERENCES 62
APPENDIX 67

LIST OF FIGURES

Figure 3.1 Architectural Diagram 26
Figure 4.2 E-commerce Quality Metrics Analysis 48
Figure 4.3 Task Management System Quality Metrics Analysis 55

LIST OF TABLES

Table 4.1 Structural Characteristics of Evaluated Monolithic Applications 41
Table 4.2 Content Management System Analysis Summary 58

Chapter 1 INTRODUCTION

The software architecture landscape has undergone a profound transformation over the past decade, marked by a decisive shift from monolithic systems to microservices architecture. This architectural evolution has been driven by the pressing need for enhanced maintainability, scalability, and deployment flexibility in modern software systems [1]. Microservices architecture decomposes applications into multiple small, independent services, each embodying a distinct business capability and adhering to the single responsibility principle. This decomposition enables individual services to be developed, deployed, and maintained independently, offering organizations unprecedented agility in responding to changing market demands and technological advancements.

Leading technology corporations including Amazon, Netflix, and Uber have successfully embraced this architectural paradigm, transitioning their application systems from monolithic to microservices to meet the rigorous demands of continuous integration, delivery, and deployment [2]. Their success stories have inspired countless organizations to embark on similar modernization journeys, seeking to unlock the benefits of improved scalability, fault isolation, and technology diversity that microservices promise. The widespread adoption of this architectural style represents more than a technical trend; it reflects a fundamental shift in how organizations approach software development and delivery in an increasingly competitive digital landscape[1], [3].

The Go programming language (Golang) has emerged as a particularly compelling choice for micro-services development, distinguished by its robust concurrency features, efficient garbage collection mechanisms, and cross-platform compilation capabilities [2]. Originally designed at Google to enhance productivity in fast-paced distributed processing environments, Go's concise syntax and performance characteristics align exceptionally well with the demands of microservice-based architectures. The language's native support for concurrent programming through goroutines and channels makes it ideally suited for building high-performance, distributed systems that form the backbone of modern microservices ecosystems. As organizations increasingly recognize these advantages, Go has become a preferred choice for implementing microservices, particularly for applications requiring high throughput and low latency [4].

Despite these compelling advantages, the transformation from monolithic applications to microservices presents substantial challenges, particularly for small to medium-sized enterprises (SMEs) operating with constrained resources [5],[4]. The decomposition process requires careful analysis of existing codebases, identification of appropriate service boundaries, and consideration of numerous technical and business factors. Organizations must navigate complex decisions about how to partition their systems while maintaining functionality, ensuring performance, and aligning with business objectives [1],[6]. The challenges extend beyond purely technical considerations to encompass organizational, operational, and financial dimensions that can significantly impact the success of migration initiatives [7].

The migration journey typically encompasses multiple strategic approaches, each with its own strengths and applicability contexts. Functionality-based decomposition groups related features into discrete services, enabling teams to organize systems around user-facing

capabilities [8], [9]. Domain-based decomposition leverages Domain-Driven Design principles to identify bounded contexts that naturally map to business domains, ensuring that technical boundaries align with organizational understanding. Responsibility-based decomposition focuses on separating concerns according to architectural layers or technical capabilities, promoting clean separation between presentation, business logic, and data access. Each approach offers distinct advantages and challenges, requiring organizations to carefully evaluate their specific contexts and requirements before committing to a decomposition strategy [10].

Successfully executing this transformation necessitates sophisticated tooling and technologies that span the entire software development life cycle. Modern microservices ecosystems rely on containerization platforms like Docker for packaging and isolation, orchestration systems such as Kubernetes for managing distributed deployments at scale, API management solutions for governing service interactions, and comprehensive monitoring platforms for observing system behavior and performance [11], [12]. Additionally, the integration of DevOps practices and cloud-native technologies plays a pivotal role in ensuring seamless deployment and operational efficiency throughout the migration process and beyond [12],[3]. The complexity of this technology stack underscores the need for specialized tools that can guide organizations through the decomposition process while accounting for these operational realities.

Several automated transformation tools have emerged to assist organizations in this complex journey, each offering different capabilities and targeting different technology stacks. Monolithic Automating Transform-Go (MAT-Go) represents one such solution, utilizing Abstract Syntax Tree (AST) analysis to identify dependencies within Go monoliths and consolidate independent components into modular microservices [1]. The tool employs a three-step process: analyzing the monolith to identify independent components, clustering these components based on similarity using the Ochiai coefficient, and adapting the modules by generating appropriate APIs. IBM's Mono2Micro offers another commercial option, combining static and dynamic analysis with machine learning to recommend service partitions for Java applications [7]. This tool analyzes both code structure and runtime behavior to identify logical groupings suitable for extraction as microservices. AWS Microservice Extractor provides automated decomposition capabilities specifically for .NET applications, leveraging runtime analysis and pattern recognition to suggest service boundaries. Each of these tools brings valuable capabilities to the decomposition challenge, yet significant limitations remain that hinder their applicability for many organizations, particularly smaller enterprises.

However, despite these technological advances, significant gaps remain in the current tooling landscape that prevent many organizations from successfully modernizing their systems. Most existing solutions rely predominantly on static code analysis, overlooking critical runtime behavior patterns that reveal how systems actually function under real-world conditions [9], [13]. Static analysis provides valuable insights into code structure and dependencies, but it cannot capture dynamic aspects such as actual usage patterns, performance characteristics, and runtime interactions that emerge only during system operation. They fail to integrate domain knowledge embedded in documentation, API specifications, and business rules, resulting in technically sound but business-misaligned service boundaries [6], [14]. This technical-centric approach frequently leads to decompositions that, while structurally coherent, do not reflect the business capabilities and domain concepts that should guide service design.

Furthermore, threshold-based clustering approaches often neglect practical considerations such as operational costs, team organizational structures, deployment constraints, and infrastructure limitations factors that prove crucial for successful real-world implementations [15]. While mathematical similarity measures like the Ochiai coefficient provide objective clustering criteria, they optimize solely for structural similarity without considering the organizational and operational realities that influence whether a proposed decomposition will succeed in practice. The lack of intelligent agent collaboration for decomposition decisions represents a missed opportunity to leverage advances in artificial intelligence for complex architectural decision-making [16]. Decomposition inherently requires synthesizing insights from multiple perspectives and navigating complex trade-offs, tasks for which collaborative AI approaches show considerable promise.

The integration of artificial intelligence, particularly through Retrieval-Augmented Generation (RAG) and multi-agent systems, presents promising new directions for addressing these limitations [17],[18]. RAG combines the generative capabilities of large language models with information retrieval from external knowledge bases, enabling more context-aware decision-making grounded in both code analysis and domain expertise [18]. This technique addresses a fundamental limitation of pure generative models, their inability to access current, domain-specific information beyond their training data. Multi-agent systems offer distributed problem-solving approaches where specialized agents collaborate to tackle complex decomposition challenges from multiple perspectives simultaneously, potentially achieving results superior to single-perspective approaches [17]. Recent research has demonstrated that multi-agent collaboration significantly improves success rates in complex problem-solving tasks by enabling specialization, distributed reasoning, and iterative refinement.

The confluence of these AI techniques with the specific needs of monolith decomposition creates an opportunity for transformative advances in automated software architecture transformation. By combining static analysis, runtime profiling, and domain knowledge extraction through coordinated multi-agent collaboration, it becomes possible to address the limitations of current tools while remaining accessible to resource-constrained organizations. This research proposes SMART-Split, a novel framework that realizes this vision through a carefully designed multi-agent RAG architecture optimized for the unique challenges faced by small to medium-sized enterprises seeking to modernize their Go-based monolithic applications.

Small to medium-sized enterprises face a convergence of critical challenges when attempting to modernize their monolithic applications, creating barriers that often prove insurmountable without appropriate tooling and support [7]. These challenges span financial, technical, and organizational dimensions, collectively impeding SMEs' ability to participate in the architectural modernization that larger enterprises have successfully undertaken. The financial barrier represents perhaps the most immediate obstacle. Existing commercial decomposition tools such as AWS Micro-service Extractor and IBM Mono2Micro command premium prices that place them beyond the reach of organizations operating with limited budgets [1]. For SMEs already managing tight financial constraints, allocating substantial resources to acquire specialized transformation tools becomes economically unfeasible, regardless of the potential long-term benefits. This cost barrier effectively excludes a significant segment of organizations from accessing sophisticated decomposition capabilities, perpetuating a technology divide between large enterprises and smaller businesses.

Beyond financial considerations, existing solutions impose substantial computational demands that frequently exceed the infrastructure capabilities available to smaller enterprises. These tools typically require significant processing power, memory resources, and storage capacity to analyze codebases, process runtime traces, and generate decomposition recommendations. For SMEs operating with modest hardware resources and limited cloud budgets, meeting these computational requirements presents a formidable challenge that compounds the financial barriers already discussed. The absence of context-aware automated decomposition tools specifically designed for smaller codebases represents another critical gap in the current landscape [19]. While commercial solutions offer comprehensive capabilities, they optimize for large-scale enterprise applications with hundreds of thousands or millions of lines of code. Smaller codebases—those under 50,000 lines of code that represent the typical scale for many SME applications—exhibit different characteristics and require different decomposition strategies [15].

Current tools' reliance on static analysis alone represents a fundamental technical limitation [13], [14]. Solutions like MAT-Go and those built on WALA (Watson Libraries for Analysis) primarily analyze code structure, dependencies, and relationships as they exist in the source code. However, this static view misses critical runtime behavior patterns—the actual execution paths, data flows, and interaction patterns that emerge when systems operate under real-world conditions[1]. Without incorporating these dynamic insights, decomposition tools risk creating service boundaries that look theoretically sound but fail to align with actual system behavior and usage patterns. The failure to incorporate domain knowledge from documentation, business rules, and organizational context further undermines decomposition quality[6], [9]. Technical analysis alone, no matter how sophisticated, cannot fully capture the business semantics and domain concepts that should guide service boundary decisions.

Documentation often contains valuable insights about business processes, domain terminology, and logical groupings that should inform how systems are partitioned. Business rules encode constraints and relationships that cross-cut technical dependencies, representing knowledge about how the system should behave according to organizational policies and requirements. Without integrating this contextual knowledge, decomposition tools produce results that may be technically adequate but misaligned with business capabilities and domain boundaries. Threshold-based approaches, exemplified by MAT-Go's use of the Ochiai coefficient for similarity measurement, introduce additional limitations [1]. While these mathematical techniques provide objective criteria for clustering components, they typically optimize for structural similarity alone. They overlook practical considerations that prove essential for successful implementations: operational costs associated with inter-service communication, team organizational boundaries that influence ownership and maintenance, deployment constraints imposed by infrastructure and operational practices, and infrastructure limitations that affect how services can be hosted and scaled.

The absence of comprehensive metrics for evaluating decomposition quality beyond basic structural measures represents another significant gap [20]. While coupling and cohesion metrics offer valuable insights into structural properties, they represent only part of the picture. Missing are comprehensive assessments of API design quality, error handling strategies, security considerations, deployment complexity, operational overhead, and business value alignment [10]. This narrow evaluation perspective makes it difficult to objectively compare different decomposition approaches or predict how well proposed service boundaries will perform in production environments. Furthermore, the lack of

comprehensive support for incremental modernization forces organizations into risky, all-or-nothing migration strategies [9]. Most tools assume a complete transformation from monolith to microservices, providing little guidance or support for gradual, evolutionary approaches that would allow organizations to manage risk and learn iteratively.

These converging challenges create an urgent need for a lightweight, cost-effective solution specifically tailored for monolithic applications with fewer than 50,000 lines of code. Such a solution must address the unique constraints faced by smaller organizations while ensuring acceptable quality in terms of scalability, maintainability, and operational efficiency [20]. It must operate effectively within resource-constrained environments, both computationally and financially. It must integrate multiple analysis perspectives static, dynamic, and domain-oriented to produce context-aware decomposition recommendations [9],[13]. And it must provide comprehensive evaluation capabilities that assess decomposition quality across multiple dimensions relevant to real-world implementations.

To address these multifaceted challenges, this research develops SMART-Split: A Resource-Efficient Multi-Agent RAG Framework for Automated Monolith Decomposition. The framework embodies a novel approach to monolith decomposition that integrates multiple analysis perspectives through specialized agents coordinated to work collaboratively, producing decomposition recommendations that are simultaneously technically sound, business-aligned, and resource-efficient. The primary objective centers on implementing a decentralized multi-agent architecture comprising five specialized agents; Static Analyzer Agent, Runtime Profiler Agent, Domain Knowledge Agent and Decomposer Agent each focusing on distinct aspects of the decomposition challenge[18]. These agents operate semi-autonomously while coordinating through a supervisor pattern that orchestrates their interactions and synthesizes their outputs.

The Static Analyzer Agent examines code structure through AST analysis, extracting dependency relationships and structural patterns that form the foundation for understanding system architecture. The Runtime Profiler Agent captures execution behavior through instrumentation and tracing, revealing actual usage patterns and performance characteristics that complement the static view. The Domain Knowledge Agent leverages RAG techniques to extract and integrate insights from documentation, API specifications, and business rules, ensuring decomposition recommendations align with domain concepts and business capabilities [11], [20]. The Decomposer Agent synthesizes inputs from all other agents to define optimal service boundaries using a hybrid algorithm that balances multiple quality criteria. Finally, Developers can start development for each identified microservice.

A critical innovation within this multi-agent framework is the development of a lightweight RAG system optimized specifically for resource-constrained environments typical of SME contexts [21]. Traditional RAG implementations demand substantial computational resources for embedding generation, vector search, and context retrieval. The SMART-Split framework adapts these techniques to operate effectively with limited resources while maintaining decomposition quality. This optimization addresses a fundamental barrier preventing SMEs from accessing advanced AI-assisted decomposition capabilities, demonstrating that sophisticated analysis techniques can be made accessible without compromising effectiveness.

The framework introduces a hybrid decomposition algorithm that represents a methodological advancement beyond current single-perspective approaches [9], [20]. This

algorithm synthesizes insights from static analysis results, runtime profiling data, and domain knowledge to generate service boundary recommendations. Unlike existing tools that prioritize one analysis type over others, the hybrid algorithm treats all three perspectives as equally important, using iterative reasoning to resolve conflicts and identify optimal boundaries. The algorithm incorporates multiple quality criteria simultaneously, structural cohesion and coupling, runtime performance patterns, domain alignment, and operational feasibility, producing recommendations that balance these often competing objectives [13].

Validation represents the final major objective, ensuring that SMART-Split delivers on its promises of improved decomposition quality and resource efficiency. The framework will be evaluated against three open-source Go monoliths of varying complexity: a small-scale application under 10,000 lines of code, a medium-scale application between 10,000 and 30,000 lines, and a larger application approaching 50,000 lines. This graduated evaluation approach enables assessment of how the framework performs across different scales and complexity levels typical of SME applications. The validation methodology extends beyond traditional coupling and cohesion metrics to encompass comprehensive quality assessment. Modularity Quality (MQ) measures the overall structural soundness of proposed service boundaries, targeting values exceeding 0.7 to ensure adequate separation of concerns. Service Independence Score (SIS) evaluates the degree of coupling between proposed services, with target thresholds above 0.8 indicating minimal inter-service dependencies. Business Alignment Index (BAI) assesses how well service boundaries correspond to identified business capabilities and domain concepts, targeting values exceeding 0.9 to ensure practical utility[22], [23], [24].

By pursuing these interconnected objectives, this research addresses a critical gap in monolith decomposition tooling while advancing the state of practice in applying AI techniques to software architecture challenges. The framework demonstrates that sophisticated, context-aware decomposition capabilities can be made accessible to resource-constrained organizations, potentially democratizing access to microservices modernization. Through the integration of multi-agent systems with RAG, static analysis with runtime profiling, and technical metrics with business alignment, SMART-Split offers a comprehensive solution to the complex challenges inherent in transforming monolithic applications into microservices architectures. The contributions extend beyond the immediate practical benefits to include methodological advances in combining AI techniques for software engineering tasks and empirical evidence about the effectiveness of multi-agent collaboration in complex architectural decision-making.

1.1. Thesis Overview

To provide a structured roadmap of this research, the following outlines the contents of each chapter:

Chapter 1: Introduction – This chapter establishes the background of the study, identifies the critical modernization gaps faced by Small and Medium-sized Enterprises (SMEs), and defines the research objectives and scope of the SMART-Split framework. It provides the foundation for exploring how AI can democratize architectural transformation.

Chapter 2: Literature Review – This section presents a comprehensive analysis of existing monolithic-to-microservices decomposition methodologies. It explores the evolution from manual and purely static techniques to modern AI-driven approaches, specifically highlighting the "context gap" that often leads to suboptimal service boundaries in current automated tools.

Chapter 3: Methodology – This chapter details the design, architecture, and implementation of the SMART-Split multi-agent system. It explains the technical orchestration between the Static Analyzer, Runtime Profiler, and Domain Knowledge agents, and describes the integration of Retrieval-Augmented Generation (RAG) for business-aligned decomposition.

Chapter 4: Results and Discussion – A detailed presentation of the experimental findings derived from validating the framework against real-world Go-based applications. The results are analyzed using quantitative metrics such as Modularity Quality (MQ) and Service Independence Score (SIS), alongside a qualitative assessment of business logic alignment.

Chapter 5: Conclusion and Future Work – The final chapter summarizes the primary research findings and confirms the fulfillment of the research objectives. It highlights the major contributions to the field of software engineering and outlines potential avenues for future enhancements, including multi-language support and automated code refactoring.

Chapter 2 LITREATURE REVIEW

2.1. Foundations of Micro-services Architecture

The evolution of software architecture over the last fifteen years has witnessed a fundamental transformation in how enterprise applications are designed and deployed. While monolithic architectures dominated software development for decades, their inherent limitations in supporting rapid iteration, independent scaling, and technology diversity have driven widespread interest in alternative architectural patterns [1], [10]. Microservices architecture emerged as a compelling response to these limitations, proposing a design philosophy where applications are constructed as collections of loosely coupled, independently deployable services organized around business capabilities.

This architectural paradigm shift gained momentum through the successful migrations undertaken by major technology companies. Organizations such as Netflix, Amazon, and Spotify demonstrated that decomposing monolithic systems into microservices could address critical scalability and organizational challenges [1]. Netflix's journey from a monolithic DVD rental system to a globally distributed streaming platform built on hundreds of microservices illustrated the potential of this approach at scale. Similarly, Amazon's transition enabled the company to organize development teams around individual services, reducing coordination overhead and accelerating feature delivery [10].

The fundamental principles underlying microservices architecture distinguish it from traditional monolithic design. Each microservice encapsulates a specific business capability and maintains its own data store, adhering to the principle of bounded contexts from Domain-Driven Design [25]. Services communicate through well-defined interfaces, typically using lightweight protocols such as REST or message queues, rather than through direct function calls or shared memory. This isolation enables teams to select appropriate technologies for each service's requirements, deploy services independently without coordinating releases across the entire system, and scale individual services based on their specific load patterns rather than scaling the entire application uniformly [8], [11], [14].

The Go programming language has gained significant traction for microservices implementation, particularly in environments demanding high throughput and low latency. Go's design philosophy emphasizes simplicity, performance, and practical software engineering, characteristics that align naturally with microservices development constraints. The language provides built-in support for concurrent programming through goroutines and channels, enabling efficient handling of multiple simultaneous requests without the complexity of traditional threading models. Its compilation to native binaries produces lightweight, self-contained executables ideal for containerized deployment, while its standard library includes robust networking and HTTP handling capabilities essential for service-to-service communication. These characteristics have made Go increasingly popular for building cloud-native applications and infrastructure tools, with notable projects including Docker, Kubernetes, and Terraform all implemented in Go [2].

Despite the compelling benefits, the transition from monolithic to microservices architecture introduces substantial complexity that organizations must carefully manage [3], [9]. Distributed system challenges that were previously internalized within a single process now manifest as inter-service concerns. Network communication between services introduces latency and potential failure modes absent in monolithic architectures, requiring

sophisticated strategies for handling partial failures, timeouts, and cascading errors [1], [12]. Data consistency becomes significantly more complex when business transactions span multiple services, each maintaining its own database [10]. Traditional ACID transactions are no longer available, forcing architects to implement eventually consistent patterns or complex distributed transaction protocols.

Operational complexity increases substantially with microservices adoption [11]. Monitoring and debugging distributed systems requires specialized tools and practices beyond those sufficient for monolithic applications. Organizations must implement distributed tracing to follow requests across service boundaries, aggregate logs from numerous services, and establish comprehensive metrics collection [11], [12]. Deployment coordination, while theoretically simplified through independent service deployments, often requires careful orchestration to manage service version compatibility and coordinate rollouts across interdependent services [4]. The infrastructure requirements expand significantly, with container orchestration platforms, service mesh implementations, and API gateway technologies becoming essential components of the operational environment[16].

2.2. Automated Decomposition Approaches

The complexity inherent in defining appropriate service boundaries has motivated extensive research into automated decomposition techniques. These approaches generally fall into three categories: static code analysis, dynamic runtime analysis, and hybrid methods combining multiple information sources [9], [13].

Static analysis techniques examine source code structure, dependencies, and relationships without executing the application [13]. These methods typically construct a dependency graph representing relationships between code elements—classes, functions, modules—and apply clustering algorithms to identify groups of highly cohesive, loosely coupled components suitable for extraction as independent services [6]. The MAT-Go tool exemplifies this approach for Go applications, utilizing Abstract Syntax Tree analysis to extract structural information and the Ochiai coefficient for measuring similarity between code elements [1]. The tool identifies exported functions as potential service entry points, analyzes their dependencies, and groups related functionality using threshold-based clustering.

Several research efforts have extended static analysis with more sophisticated techniques. Fritzsch et al. proposed using feature tables to map system functionality to code elements, then clustering features to identify service candidates [8]. Graph-based approaches model applications as weighted graphs where nodes represent code entities and edges represent relationships, applying community detection algorithms to partition the graph [11]. Machine learning techniques have been explored for predicting optimal service boundaries based on features extracted from code structure and historical evolution patterns [26].

The primary limitation of static-only approaches lies in their inability to capture runtime behavior [13]. Code structure reveals potential relationships and dependencies, but actual usage patterns emerge only during execution. A function may have numerous dependencies in the code but invoke only a subset during typical operation, while seemingly independent components may exhibit tight runtime coupling through shared state or coordinated execution flows [27]. Performance characteristics, including latency-sensitive code paths

and resource-intensive operations, remain invisible to static analysis yet critically influence appropriate service granularity [9].

Dynamic analysis addresses these limitations by instrumenting applications to capture execution behavior [13], [21]. Runtime profiling records function calls, execution traces, data flows, and performance metrics during typical application operation. This behavioral information reveals actual usage patterns, identifies frequently co-executing code segments, exposes performance bottlenecks and resource consumption patterns, and captures implicit dependencies visible only at runtime.

IBM's Mono2Micro tool pioneered the integration of static and dynamic analysis for automated decomposition [1], [20]. The tool begins with static analysis to extract class-level dependency relationships from Java applications. It then instruments the application to capture execution traces during user-defined business use cases, recording which classes are invoked together during typical operations. Machine learning algorithms analyze both static and dynamic data to recommend service partitions, with an interactive workbench allowing architects to iteratively refine recommendations. This hybrid approach has demonstrated superior decomposition quality compared to static-only methods, particularly for identifying services aligned with business use cases [20].

Process mining techniques offer another dynamic analysis approach, treating execution traces as process models and applying mining algorithms to discover logical workflow boundaries [28]. These methods construct process graphs from execution logs, identify high-frequency execution paths, detect workflow boundaries where interactions become sparse, and suggest service boundaries aligned with business processes. Research has demonstrated that process mining can effectively identify service candidates corresponding to business capabilities, though the quality depends heavily on the comprehensiveness and representativeness of captured execution traces [29].

Despite advances in automated decomposition, significant gaps remain in current approaches [19]. Most tools neglect domain knowledge embedded in documentation, architecture decision records, API specifications, and business rules [15]. This technical-centric focus often results in decompositions that are structurally sound but misaligned with business domains and organizational understanding [19]. Threshold-based clustering methods optimize for mathematical similarity without considering practical constraints such as team organizational boundaries, operational cost implications, deployment infrastructure limitations, and organizational readiness for managing distributed systems [9], [14].

2.3. Domain-Driven Design and Business Alignment

Domain-Driven Design provides a strategic framework for structuring complex software systems around business domains and capabilities. The concept of bounded contexts linguistic and organizational boundaries defining specific business domain areas, offers a principled approach to identifying service boundaries [28], [30]. Each bounded context encapsulates a cohesive set of domain concepts with explicit boundaries defining what is inside and outside the context, a ubiquitous language shared by team members and reflected in code, and clear integration points with other bounded contexts through defined interfaces [30].

Applying DDD principles to microservices decomposition aligns technical boundaries with business understanding [30], [31]. Strategic DDD patterns guide the identification of bounded contexts through collaborative domain modeling, involving domain experts and developers in defining business capabilities, aggregates representing transactional consistency boundaries within domains, and domain events capturing significant business occurrences [30], [31]. Context mapping clarifies relationships between bounded contexts, identifying where integration is required and appropriate integration patterns [31].

The challenge lies in operationalizing DDD principles within automated decomposition tools [30], [32]. While DDD provides conceptual guidance for identifying service boundaries, translating domain knowledge into machine-readable form that decomposition algorithms can leverage remains difficult [1]. Documentation often captures domain concepts informally through natural language descriptions that resist automated analysis [9]. Business rules may be implicit in code structure or scattered across implementation files rather than explicitly modeled. Domain expertise resides in human understanding, not easily codified for algorithmic consumption.

Recent research has explored techniques for extracting domain knowledge to inform decomposition decisions [9], [32]. Natural language processing of documentation can identify domain terms and concepts, with clustering of semantically related terms suggesting potential domain boundaries [32]. Analysis of business process models or workflow descriptions can reveal logical groupings corresponding to business capabilities [30]. Semantic analysis of API names, parameter types, and return values can uncover implicit domain structure reflected in interface design. However, these approaches remain limited in capturing the full richness of domain understanding that experienced architects bring to decomposition decisions [1], [32].

2.4. Retrieval-Augmented Generation for Software Engineering

Retrieval-Augmented Generation represents a significant advancement in applying large language models to knowledge-intensive tasks [21], [22], [33]. Traditional language models generate responses based solely on patterns learned during training, limiting their ability to access current information, incorporate domain-specific knowledge, or provide citations for generated content [21], [33]. RAG addresses these limitations by augmenting generation with information retrieval from external knowledge bases [21].

The RAG architecture comprises several key components [21], [33] . Document ingestion processes source documents by chunking them into segments, generating vector embeddings capturing semantic content, and indexing embeddings in a vector database enabling efficient similarity search. Query processing transforms user queries into vector representations using the same embedding model, enabling semantic matching between queries and document chunks. Retrieval identifies the most relevant document chunks through vector similarity search, typically retrieving the top-k most similar segments to augment generation context. Generation combines retrieved context with the original query in a prompt template, with the language model synthesizing a response grounded in retrieved information [33].

RAG offers several advantages for software engineering applications [22], [34]. It enables LLMs to access current technical documentation, API specifications, coding standards, and architectural patterns without requiring model retraining [34]. Domain-specific knowledge from internal documentation, past architectural decisions, and organizational best practices can augment generation. Generated responses can include citations to source documents, enabling verification and building trust. The external knowledge base can be updated independently of the model, avoiding expensive retraining cycles [33].

Several recent studies have explored RAG applications in software development contexts [15], [26]. Code generation systems use RAG to retrieve relevant code examples from repositories, incorporating them as context for generating new code aligned with project conventions [35]. Documentation generation tools retrieve related documentation sections to produce consistent, contextually appropriate content [20]. Architecture decision support systems augment LLM reasoning with retrieval of past architectural decisions, design patterns, and best practices[13]. Bug fixing assistants retrieve similar historical bugs and their resolutions to inform fix generation [9].

Recent advances in RAG techniques address some of these challenges. Chain-of-Retrieval approaches perform iterative retrieval and reasoning, refining queries based on initial results to gather more comprehensive context. Multi-query retrieval generates multiple reformulations of the original query, retrieving for each to improve coverage [21]. Hybrid search combines vector similarity with keyword matching and structural queries to improve retrieval precision. Lightweight RAG implementations optimize memory usage and inference speed for resource-constrained environments [1], [15].

2.5. Multi-Agent Systems for Complex Problem Solving

Multi-agent systems provide a framework for decomposing complex problems into subtasks addressed by specialized agents working collaboratively [36], [37]. Each agent focuses on a specific aspect of the overall problem, bringing specialized capabilities and reasoning approaches [36]. Agents coordinate through well-defined communication protocols, sharing information, requesting assistance, and negotiating decisions [18], [36]. This distributed problem-solving approach offers several advantages over monolithic single-agent architectures [36][17].

Specialization enables each agent to develop deep expertise in its domain without needing comprehensive knowledge of all problem aspects [36], [37]. Rather than a single agent attempting to master code structure analysis, runtime behavior interpretation, and domain knowledge synthesis simultaneously, specialized agents can focus on their respective areas[38]. This modularity simplifies agent development and maintenance, as agents can be upgraded or replaced independently without disrupting the entire system [18], [37]. Robustness improves through redundancy and error handling, with agent failures potentially isolated without cascading system-wide [36].

Recent applications of multi-agent systems to complex problems demonstrate their effectiveness [17], [36], [39]. Software development agents collaborate to analyze requirements, generate code, write tests, and review implementations, with each agent specializing in specific development activities [39]. Travel planning agents coordinate flight

search, hotel booking, activity recommendations, and itinerary optimization, each accessing specialized information sources [40]. Customer support agents route queries to specialized agents for technical issues, billing questions, account management, and feedback collection [36].

The key to effective multi-agent collaboration lies in coordination mechanisms that enable agents to work together effectively [19], [38], [41]. Hierarchical coordination employs a supervisor agent that decomposes complex tasks into subtasks, assigns subtasks to appropriate specialized agents, aggregates results from individual agents, and synthesizes a coherent overall response [17], [18], [40]. This approach provides centralized orchestration while maintaining agent specialization[40].

Applying multi-agent architectures to monolith decomposition aligns naturally with the problem's inherent complexity [6], [16]. Different analysis perspectives; structural, behavioral, and domain-oriented require distinct expertise and reasoning approaches [9], [10], [27]. A static analyzer agent can focus exclusively on code structure and dependencies, developing sophisticated techniques for AST analysis, dependency extraction, and structural metric calculation [1]. A runtime profiler agent specializes in instrumenting applications, capturing execution traces, analyzing performance patterns, and identifying behavioral coupling [13], [18]. A domain knowledge agent leverages RAG and NLP techniques to extract business concepts from documentation, identify domain boundaries, and map code elements to business capabilities [6], [14].

These agents must coordinate to produce coherent decomposition recommendations [14], [20]. The static analyzer provides a structural foundation, identifying potential service candidates based on code dependencies. The runtime profiler validates and refines these candidates by revealing actual execution patterns [18]. The domain knowledge agent ensures proposed boundaries align with business understanding [17]. A decomposer agent synthesizes inputs from all perspectives, resolving conflicts and producing final service boundary recommendations [42].

2.6. Evaluation Metrics for Decomposition Quality

Assessing the quality of proposed microservices decompositions requires comprehensive metrics spanning multiple dimensions [9], [43]. Traditional software metrics provide important structural insights but capture only part of the picture [20].

Structural metrics evaluate the internal quality of proposed services and their relationships [28], [44], [45]. Coupling metrics measure dependencies between services, including afferent coupling (incoming dependencies), efferent coupling (outgoing dependencies), instability (ratio of efferent to total coupling), and interface coupling (dependencies through explicit interfaces). Cohesion metrics assess the internal consistency of services, such as Lack of Cohesion in Methods measuring how well class methods relate to instance variables, and functional cohesion evaluating whether service operations support a unified purpose [20]. Service granularity metrics characterize service size and complexity, including lines of code per service, number of operations per service interface, and cyclomatic complexity distribution [45].

Beyond structural properties, decomposition quality depends on operational characteristics [14], [27]. Performance implications of inter-service communication overhead, data transfer volumes between services, and distributed transaction requirements significantly impact system behavior [15]. Deployment complexity, considering service deployment dependencies, configuration management requirements, and infrastructure provisioning needs, affects operational feasibility [15]. Scalability characteristics, examining which services require independent scaling and resource utilization patterns, influence infrastructure costs [20].

Business alignment metrics assess how well technical boundaries correspond to organizational and domain understanding [8], [10], [31]. Service-to-capability mapping evaluates whether each service represents a coherent business capability [20]. Team alignment considers whether service boundaries enable clear ownership by autonomous teams [14], [15]. Domain boundary consistency examines alignment with identified bounded contexts from domain modeling [30].

Several comprehensive frameworks have been proposed for decomposition quality assessment [20], [46], [47]. Service Cutter defines sixteen coupling criteria spanning different dimensions, including structural coupling through code dependencies, semantic coupling based on shared domain concepts, and behavioral coupling from runtime interactions [20], [45]. The framework applies these criteria systematically to evaluate and compare alternative decomposition strategies. Modularity Quality (MQ) measures decomposition quality by comparing intra-cluster connectivity to inter-cluster connectivity, with higher values indicating better separation of concerns [46]. The metric can be computed at multiple granularities and applied iteratively to assess decomposition refinements.

2.7. Challenges for Small and Medium-Sized Enterprises

While microservices offer compelling benefits, SMEs face distinct challenges in adopting this architecture [1], [42], [48]. Resource constraints—financial, computational, and human—limit their ability to leverage existing decomposition tools and implement sophisticated microservices infrastructures [4], [48].

Commercial decomposition tools impose prohibitive costs for smaller organizations [7], [42]. AWS Microservice Extractor and IBM Mono2Micro target enterprise customers, with pricing models reflecting their comprehensive capabilities and support [1]. For SMEs operating with limited IT budgets, these costs represent significant barriers [33]. Even when tools are technically suitable, the investment required for licensing, training, and ongoing support exceeds available resources [48].

Computational requirements of existing solutions frequently exceed SME infrastructure capabilities [14],[16],[27]. Analysis of large codebases demands substantial memory and processing power. Runtime profiling generates significant trace volumes requiring storage and processing capacity [15]. Machine learning-based decomposition techniques involve training and inference costs that smaller organizations struggle to support [48]. Cloud-based solutions offer scalability but introduce recurring operational costs that accumulate rapidly [1].

Human resource limitations compound technical challenges [49], [50]. SMEs typically lack dedicated DevOps teams or site reliability engineers with expertise in distributed systems operations [4]. Developers must assume operational responsibilities alongside their primary development duties, stretching already limited capacity. The learning curve for microservices patterns, container orchestration, service mesh technologies, and distributed systems debugging represents a significant investment that smaller organizations find difficult to justify [1], [49].

These challenges create a need for decomposition approaches specifically tailored to SME contexts [15], [51]. Solutions must operate effectively within resource constraints, minimizing computational and financial requirements [4], [50]. They should require modest infrastructure, compatible with typical SME computing resources [32]. The complexity should remain manageable for small development teams without specialized expertise [19], [51]. Finally, they must provide clear value propositions demonstrating concrete benefits that justify the investment [51].

2.8. Research Gaps and Opportunities

Synthesizing the reviewed literature reveals several critical gaps that this research addresses [9], [14], [15], [16].

Current decomposition tools predominantly rely on single analysis perspectives [13], [16], [37]. Static analysis tools overlook runtime behavior while dynamic analysis approaches neglect structural properties and domain semantics. Few tools effectively integrate multiple information sources to produce context-aware recommendations[6], [16]. The opportunity lies in developing hybrid approaches that synthesize static analysis, runtime profiling, and domain knowledge extraction through coordinated multi-agent collaboration [1], [9], [14].

Domain knowledge integration remains underdeveloped in automated decomposition [52]. While DDD principles emphasize business alignment, operationalizing these principles in algorithmic form proves challenging [14]. Documentation analysis techniques exist but rarely integrate with decomposition tools [9]. RAG offers promising avenues for incorporating domain knowledge but remains largely unexplored in this context [21]. The opportunity exists to leverage RAG techniques for extracting and applying domain knowledge to decomposition decisions, ensuring technical boundaries align with business understanding [9], [12], [29].

SME-focused solutions are notably absent from the current landscape. Existing tools target large enterprises with substantial resources, leaving smaller organizations underserved. The opportunity lies in developing lightweight, resource-efficient approaches that deliver acceptable decomposition quality within SME constraints. This includes optimizing computational requirements, minimizing infrastructure dependencies, simplifying operational complexity, and demonstrating clear value propositions [7], [15], [19].

Comprehensive evaluation frameworks for decomposition quality remain limited. Most studies rely on basic coupling and cohesion metrics, neglecting operational, organizational, and business dimensions [20]. The opportunity exists to develop more holistic assessment approaches that evaluate decompositions across multiple quality dimensions, providing

objective comparison between alternative strategies and predicting real-world implementation success [9].

Chapter 3 METHODOLOGY

3.1. Research Design and Approach

This research adopts a design science methodology combined with experimental validation to develop and evaluate the SMART-Split framework for automated monolith decomposition. Design science research is particularly well-suited for this study as it focuses on creating and evaluating innovative artifacts in this case, a multi-agent RAG framework that address real-world problems faced by small and medium-sized enterprises. This methodology emphasizes iterative development, empirical evaluation, and practical relevance, aligning perfectly with the objectives of producing a functional decomposition tool that SMEs can realistically adopt.

The research follows a structured approach comprising four major phases: framework design and architecture development, implementation of specialized agents, experimental validation across multiple case studies, and comparative analysis with existing tools. This phased approach enables systematic refinement of the framework while ensuring that each component contributes meaningfully to the overall decomposition quality. Throughout the development process, insights from each phase inform subsequent iterations, embodying the iterative nature of design science research.

The experimental design incorporates both quantitative and qualitative evaluation methods. Quantitative metrics assess structural quality through coupling, cohesion, and modularity measures, while qualitative analysis examines business alignment and operational feasibility. This mixed-methods approach provides a comprehensive understanding of the framework's effectiveness across multiple dimensions, addressing both technical soundness and practical utility. By combining rigorous measurement with contextual interpretation, the methodology ensures that evaluation captures the full complexity of monolith decomposition as both a technical and organizational challenge.

3.2. SMART-Split Framework Architecture

The SMART-Split framework employs a hierarchical multi-agent architecture coordinated through a supervisor pattern, inspired by recent advances in multi-agent systems for complex problem-solving. This architectural choice reflects the recognition that monolith decomposition requires synthesizing insights from multiple perspectives; structural, behavioral, and domain-oriented each demanding specialized expertise and analysis techniques. Rather than attempting to address all aspects through a single monolithic algorithm, the framework distributes responsibilities across specialized agents that collaborate to produce comprehensive decomposition recommendations (Appendix 5).

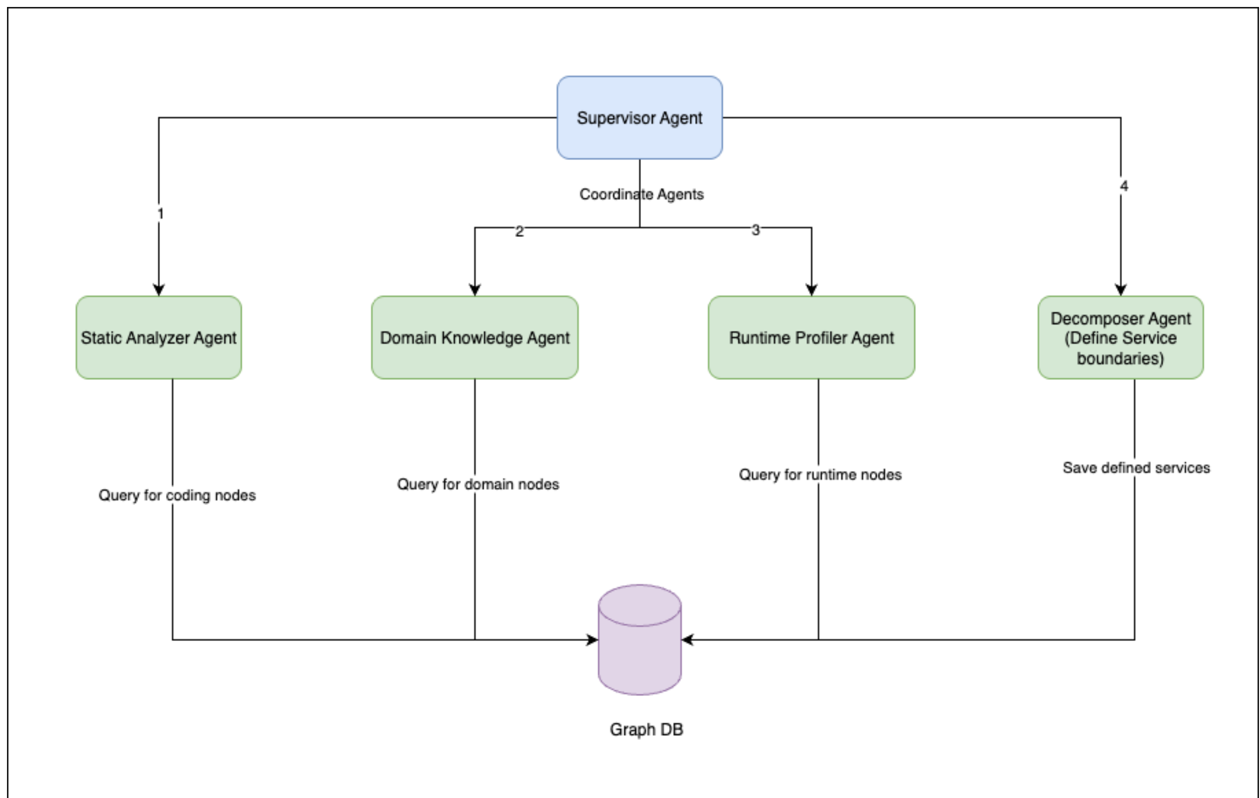


Figure 3.1 Architectural Diagram

3.2.1. Supervisor Agent

The Supervisor Agent serves as the central coordinator, orchestrating the workflow across all specialized agents and managing the overall decomposition process. This agent implements an orchestrator-worker pattern, where high-level decomposition tasks are broken down into subtasks assigned to appropriate specialized agents based on their capabilities. The supervisor maintains a state machine tracking the decomposition progress, coordinates information flow between agents, aggregates outputs from specialized analyses, resolves conflicts when different perspectives yield contradictory recommendations, and produces the final service boundary definitions.

The supervisor's coordination mechanism draws inspiration from hierarchical multi-agent reinforcement learning, where a high-level agent decomposes complex tasks into manageable subtasks for specialized sub-agents. However, rather than learning coordination strategies through reinforcement, the SMART-Split supervisor employs predefined orchestration logic grounded in software architecture principles and microservices best practices. This design choice reflects the deterministic nature of decomposition requirements and the need for explainable decision-making that developers can understand and trust.

Communication between the supervisor and specialized agents follows a structured protocol. The supervisor initiates analysis by providing agents with necessary context and parameters, receives structured outputs containing analysis results and recommendations, requests additional analysis when initial results require clarification, and synthesizes multiple perspectives into coherent decomposition decisions. This protocol ensures that

agent interactions remain organized and traceable, facilitating debugging and refinement during framework development.

3.2.2. Static Analyzer Agent

The Static Analyzer Agent focuses exclusively on examining code structure through Abstract Syntax Tree analysis and dependency extraction. This agent parses Go source files to construct a comprehensive structural model of the monolithic application, capturing relationships between functions, packages, and modules. The analysis process involves several key steps that transform raw source code into actionable architectural insights.

First, the agent performs AST parsing using Go's built-in parser libraries, converting source files into tree structures that preserve syntactic relationships. This parsing process identifies all exported functions, which represent potential service entry points, internal functions and their calling relationships, data structures and type definitions, package-level organization and dependencies, and import statements revealing external dependencies. The resulting AST representation provides a complete structural view of the application's organization.

Next, the agent constructs a dependency graph where nodes represent code elements at various granularities; functions, types, or packages and directed edges represent dependencies such as function calls, data structure usage, or package imports. Graph construction applies configurable rules for determining dependency significance, filtering out trivial dependencies that don't meaningfully constrain decomposition. The dependency graph becomes the foundation for subsequent clustering and boundary identification.

The agent then computes structural metrics characterizing the codebase's organization. These metrics include afferent coupling measuring how many other components depend on each component, efferent coupling measuring how many external dependencies each component has, instability calculating the ratio of efferent to total coupling, cohesion metrics evaluating the internal consistency of potential service groupings, and complexity measures such as cyclomatic complexity distribution across components. These metrics provide quantitative indicators of structural quality that inform decomposition decisions.

Finally, the agent applies community detection algorithms to the dependency graph, identifying clusters of highly interconnected components that might form cohesive services. The framework implements multiple clustering algorithms Louvain community detection for hierarchical modularity optimization, spectral clustering using graph Laplacian eigen analysis, and greedy agglomerative clustering optimizing modularity incrementally, allowing comparison of different structural perspectives. Each algorithm produces candidate service groupings characterized by structural metrics, which the supervisor subsequently refines using runtime and domain insights.

3.2.3. Runtime Profiler Agent

The Runtime Profiler Agent captures behavioral characteristics through execution tracing and performance monitoring. While static analysis reveals potential relationships, runtime profiling exposes actual system behavior during operation, identifying which dependencies

matter in practice and which remain dormant. This dynamic perspective proves essential for avoiding decompositions that look structurally sound but fail to align with real-world usage patterns.

The profiling process begins with instrumenting the monolithic application to capture execution traces. The framework employs Go's standard profiling capabilities, augmented with custom instrumentation for distributed tracing. Instrumentation captures function entry and exit events with timestamps, call stack information revealing execution context, goroutine identifiers tracking concurrent execution paths, memory allocation patterns, and custom application-level events marking significant business logic execution. The instrumentation overhead remains minimal, typically under 5% performance impact, ensuring that captured behavior reflects normal operation rather than pathological profiling artifacts.

Once instrumented, the application executes under representative workload scenarios designed to exercise typical usage patterns. For open-source applications, these scenarios derive from unit tests, integration tests, and example usage documentation. For custom applications, scenarios reflect actual business processes and user interactions. The framework captures execution traces across multiple scenario runs, aggregating data to identify consistent behavioral patterns while filtering statistical outliers.

The agent processes collected traces to construct a runtime dependency graph complementing the static dependency graph. Runtime graph nodes represent code elements, while weighted edges indicate actual execution relationships. Edge weights reflect call frequency, determining how often one component invokes another, cumulative execution time spent in cross-component calls, data transfer volume for calls passing substantial data structures, and temporal coupling for components frequently co-executing. These weights enable the framework to distinguish critical runtime relationships from incidental structural dependencies.

Performance analysis identifies bottlenecks and resource-intensive operations influencing service granularity decisions. Components exhibiting high latency become candidates for optimization or isolation, while resource-intensive operations may warrant dedicated services enabling independent scaling. The agent produces a performance profile characterizing each potential service candidate's computational demands, memory requirements, I/O patterns, and concurrency characteristics. This information guides decomposition decisions by ensuring that proposed services have appropriate resource footprints and performance characteristics.

3.2.4. Domain Knowledge Agent (RAG Synthesizer)

The Domain Knowledge Agent integrates contextual understanding from documentation, API specifications, and business rules through Retrieval-Augmented Generation techniques. This agent addresses a critical limitation of purely technical decomposition approaches: their inability to incorporate domain semantics and business understanding that should guide service boundary decisions. By leveraging RAG, the framework grounds decomposition recommendations in both technical analysis and domain knowledge, producing boundaries that align with business capabilities and organizational understanding.

The agent's operation begins with knowledge base construction from available documentation sources. For open-source applications, this includes README files explaining application purpose and organization, API documentation describing interfaces and contracts, inline code comments providing implementation context, architecture decision records documenting past design choices, and user guides explaining business workflows. For custom applications, additional sources may include business process models, domain glossaries defining terminology, and requirements specifications. The framework implements document preprocessing to extract meaningful content while filtering boilerplate and redundant information.

Document chunking divides each source into semantic segments appropriate for retrieval. Rather than arbitrary fixed-size windows, the framework employs intelligent chunking that preserves semantic boundaries, splitting on section headers, paragraph breaks, or logical transitions. Chunk sizes balance granularity—smaller chunks enable precise retrieval but lose context; larger chunks provide context but reduce precision. The framework implements overlapping chunks with shared context at boundaries, ensuring that important information spanning chunk boundaries remains accessible.

Vector embedding transforms document chunks into high-dimensional semantic representations enabling similarity-based retrieval. The framework employs lightweight embedding models optimized for resource efficiency, specifically sentence-transformers variants fine-tuned for technical documentation. These models generate embeddings capturing semantic meaning beyond mere keyword matching, enabling the system to retrieve relevant information even when query and document use different terminology. Generated embeddings are indexed in a vector database supporting efficient similarity search.

During decomposition, the agent performs context-aware retrieval to augment decision-making. When evaluating potential service boundaries, the agent generates queries describing the boundary's characteristics and retrieves relevant documentation chunks. Retrieved context includes domain concepts referenced by clustered components, business capabilities supported by candidate services, architectural patterns applicable to the decomposition, and historical decisions relevant to boundary placement. The agent synthesizes this retrieved information with technical analysis from static and runtime agents, producing recommendations that balance technical quality with business alignment.

The RAG implementation employs several optimizations for resource efficiency. Caching frequently accessed embeddings reduces redundant computation, batch processing amortizes overhead across multiple queries, quantization reduces embedding storage requirements, and approximate nearest neighbor search trades minimal accuracy for substantial speed improvements. These optimizations enable the framework to operate effectively even on modest hardware, addressing the resource constraints characteristic of SME environments.

3.2.5. Decomposer Agent

The Decomposer Agent synthesizes inputs from all specialized agents to define final service boundaries. This agent implements the hybrid decomposition algorithm at the core of the

SMART-Split framework, integrating structural analysis, runtime behavior, and domain knowledge into coherent decomposition decisions. The decomposer must navigate inherent tensions between different objectives maximizing cohesion while minimizing coupling, aligning with domain boundaries while respecting performance constraints, producing services of appropriate granularity for SME operational capacity.

The decomposition algorithm operates iteratively, progressively refining service boundaries through multiple rounds of analysis and adjustment. Initial decomposition begins with structural clusters identified by the Static Analyzer, treating these as provisional service candidates. The algorithm then refines these candidates by incorporating runtime insights from the Runtime Profiler. Components exhibiting strong runtime coupling migrate into the same service even if structural coupling appears weak, recognizing that behavioral dependencies often prove more constraining than structural ones. Conversely, components with structural dependencies but minimal runtime interaction may split into separate services if other factors support separation.

Domain knowledge from the RAG Synthesizer provides semantic constraints and business alignment criteria. The algorithm checks whether proposed service boundaries align with identified business capabilities and domain concepts. When boundaries crosscut domain concepts grouping functionality from multiple business areas into a single service the algorithm considers splitting or rearranging to improve alignment. When multiple services address closely related domain concerns, the algorithm evaluates merging to reduce operational complexity.

The algorithm employs multi-objective optimization balancing competing quality criteria. Objectives include structural quality maximizing intra-service cohesion while minimizing inter-service coupling, runtime efficiency minimizing cross-service communication overhead, domain alignment ensuring services correspond to business capabilities, deployment feasibility maintaining service count and complexity within SME operational capacity, and performance characteristics ensuring appropriate service granularity for independent scaling. The framework implements a weighted scoring function combining these objectives, with weights configurable to reflect organizational priorities.

Conflict resolution mechanisms handle cases where different analyses yield contradictory recommendations. When structural analysis suggests grouping components that runtime analysis indicates should separate, the decomposer examines the strength of evidence from each perspective. Strong runtime coupling typically overrides weak structural coupling, reflecting the primacy of actual behavior over potential relationships. However, when runtime evidence comes from limited scenarios, structural dependencies may carry more weight. The framework implements heuristics encoding these prioritization rules, derived from microservices best practices and validated through empirical evaluation.

The algorithm produces a final service boundary specification defining each microservice's scope and interfaces. For each identified service, the specification includes component membership listing functions, types, and packages belonging to the service, interface definition specifying the service's public API, dependency contracts describing interactions with other services, deployment requirements outlining resource needs and scaling characteristics, and business capability mapping relating the service to domain concepts. This comprehensive specification enables downstream implementation activities while providing documentation for architectural understanding.

3.2.6. Technology Stack

The SMART-Split framework implementation leverages several technologies carefully selected for their capabilities and resource efficiency. The framework is implemented in Python, chosen for its rich ecosystem of libraries supporting code analysis, machine learning, and web services. Python's expressiveness enables rapid prototyping and iteration during framework development, while its performance proves adequate for the analysis workloads characteristic of applications under 50,000 lines of code.

LangChain provides the foundation for agent coordination and LLM integration. This framework simplifies building applications combining large language models with external tools and data sources, handling prompt templating, output parsing, and chain composition. SMART-Split extends LangChain's agent capabilities with custom tools for code analysis and dependency management, integrating AST parsing, graph algorithms, and vector retrieval into the agent workflow.

LangGraph implements the supervisor's state machine for orchestrating multi-agent workflows. This library enables declarative specification of agent interaction patterns, defining which agents execute under what conditions and how information flows between stages. LangGraph's execution model supports both synchronous and asynchronous agent invocation, enabling parallel analysis where appropriate while maintaining deterministic ordering where dependencies demand.

Amazon Bedrock provides the underlying large language model capabilities for natural language understanding and generation. The framework primarily uses Claude models for their strong reasoning capabilities and large context windows, though the architecture remains model-agnostic through LangChain's abstraction layer. Bedrock's serverless deployment model aligns well with SME resource constraints, eliminating infrastructure management overhead and enabling pay-per-use pricing.

Graph databases store and query the dependency graphs constructed during analysis. The framework supports multiple graph database backends Neo4j for rich query capabilities, TigerGraph for performance-intensive graph algorithms, and embedded solutions like SQLite with graph extensions for minimal deployment overhead. Graph storage enables efficient traversal queries for dependency analysis, community detection algorithms for clustering, and path finding for analyzing service interaction patterns.

Vector databases support the RAG Synthesizer's retrieval operations. The framework implements a lightweight vector store using FAISS (Facebook AI Similarity Search) for its efficiency and modest resource requirements. FAISS enables approximate nearest neighbor search with sub-linear query complexity, making retrieval feasible even as knowledge bases grow. For larger deployments, the architecture supports replacing FAISS with dedicated vector databases like Pinecone or Weaviate.

LangSmith provides observability and monitoring for the multi-agent workflow. This tool captures execution traces showing agent invocations, intermediate outputs, and decision points, facilitating debugging and performance analysis during framework development. LangSmith's tracing capabilities prove particularly valuable for understanding complex multi-agent interactions and identifying opportunities for optimization.

3.3. Data Collection and Preparation

3.3.1. Application Selection Strategy

The research employs purposive sampling to select Go monolithic applications for framework validation. This non-probability sampling approach enables deliberate selection of applications meeting specific criteria relevant to the research objectives while providing diverse contexts for evaluation. The strategy balances representativeness ensuring selected applications reflect typical SME scenarios with practical constraints around accessibility and documentation availability.

Selection criteria establish requirements that candidate applications must satisfy. Applications must be implemented in Go to align with the framework's language-specific analysis capabilities. Codebase size must fall between 1,000 and 50,000 lines of code, representing the target range for SME applications. Applications must be open-source with accessible repositories, enabling comprehensive analysis without proprietary restrictions. Some documentation must be available, providing domain knowledge for RAG-based analysis. The codebase should exhibit monolithic architecture rather than already being decomposed into microservices, ensuring realistic migration scenarios.

Additional selection considerations include application maturity, preferring projects with stable APIs and sustained development activity over prototypes or abandoned experiments. Domain diversity ensures that selected applications span different problem domains, preventing framework evaluation from being biased toward specific application characteristics. Complexity variation, incorporating both simple and complex applications within the size constraints, tests the framework's ability to handle different architectural patterns.

Based on these criteria, three applications are selected representing small, medium, and large codebases within the target range. The small-scale application (under 10,000 LOC) provides a baseline case with manageable complexity, enabling detailed analysis of framework behavior. The medium-scale application (10,000-30,000 LOC) represents typical SME application size, providing the most relevant evaluation context. The large-scale application (30,000-50,000 LOC) approaches the upper limit of the framework's target range, testing scalability and performance under more demanding conditions.

3.3.2. Static Analysis Data Collection

Static analysis data collection begins with repository cloning and local setup of the selected Go applications. The framework parses all Go source files within the application, constructing abstract syntax trees that preserve complete structural information. Parsing employs Go's standard library packages; go/parser for syntactic analysis, go/ast for tree manipulation, and go/types for semantic type checking. These tools provide comprehensive access to code structure while ensuring parsing accuracy aligned with Go language specifications.

The collection process extracts several categories of structural information. Function definitions are recorded, capturing signatures, parameters, return types, and body structure. Type definitions document structures, interfaces, and type aliases along with their

relationships. Package organization maps the modular structure, recording which files belong to which packages and how packages depend on each other. Import relationships track dependencies on both standard library and third-party packages. Control flow within functions is analyzed to understand complexity and coupling patterns.

Dependency relationship extraction constructs the static dependency graph. The framework identifies function calls, both within the application and to external libraries, data structure usage, including field access and method invocation, interface implementation relationships, and package-level dependencies implied by imports. Each identified dependency becomes an edge in the dependency graph, weighted according to coupling strength indicators such as call frequency within the static structure, data sharing between components, and coupling type, distinguishing data coupling from control coupling.

The framework computes comprehensive structural metrics for the entire codebase and for potential service groupings. Metrics include lines of code per component, cyclomatic complexity measuring control flow complexity, coupling metrics quantifying dependencies, cohesion metrics assessing internal consistency, and instability measuring change propagation risk. These metrics enable quantitative comparison between decomposition alternatives and provide objective quality indicators.

3.3.3. Runtime Analysis Data Collection

Runtime analysis requires instrumenting the target applications to capture execution behavior during typical operation. Instrumentation employs Go's built-in profiling capabilities supplemented with custom tracing code. The framework inserts trace points at function boundaries, capturing entry and exit events, at significant business logic transitions, marking workflow progression, and around external interactions, recording database queries, API calls, and file I/O. Instrumentation is designed for minimal overhead, ensuring that captured behavior reflects normal operation rather than measurement artifacts.

Representative workload scenarios exercise the instrumented applications to generate meaningful execution traces. For applications with test suites, existing tests provide initial scenarios covering core functionality. For web applications, the framework simulates typical user interactions based on documentation or example usage. For data processing applications, representative datasets exercise major processing paths. Multiple scenario executions capture behavioral variability, enabling identification of consistent patterns versus scenario-specific behavior.

Trace data collection captures detailed execution information during scenario runs. Recorded information includes function call sequences showing execution flow, timing data measuring function execution durations, concurrency patterns tracking goroutine creation and synchronization, memory allocation recording heap and stack usage, and resource utilization monitoring CPU, memory, and I/O consumption. The framework aggregates traces across multiple executions, computing statistical summaries that characterize typical behavior while filtering anomalies.

Runtime dependency graph construction processes collected traces to identify behavioral relationships. The framework computes call frequencies, determining how often each

function invokes others, temporal coupling, measuring how frequently functions execute in close temporal proximity, data flow analysis, tracking data passing between components through parameters and return values, and performance bottlenecks, identifying expensive operations and critical paths. The resulting runtime graph complements the static dependency graph, providing a behavioral perspective on system architecture.

3.3.4. Domain Knowledge Collection

Domain knowledge collection aggregates documentation and contextual information that inform business-aligned decomposition. The framework systematically gathers available documentation sources from the application repository and related resources. Primary sources include README files providing project overview and usage instructions, API documentation describing interfaces and contracts, architecture documentation explaining design decisions, inline code comments offering implementation context, and issue trackers revealing pain points and evolution drivers. Secondary sources may include blog posts by project maintainers, conference presentations about the application, and user forum discussions.

Document preprocessing extracts and normalizes textual content for RAG processing. Preprocessing steps include format conversion, transforming various document formats into plain text while preserving structure, header extraction, identifying section titles and organizational hierarchy, code snippet filtering, distinguishing prose from embedded code examples, and boilerplate removal, eliminating standard copyright notices and navigation elements. Processed documents preserve semantic content while removing noise that would dilute retrieval quality.

Semantic chunking divides documents into retrievable segments. The framework employs hierarchical chunking, creating chunks at multiple granularities—section-level chunks for coarse retrieval, paragraph-level chunks for fine-grained matching, and sentence-level chunks for precision retrieval. Chunks overlap at boundaries to prevent information loss from arbitrary splits. Each chunk is tagged with metadata including source document, section hierarchy, and creation date, enabling retrieval to consider provenance alongside semantic similarity.

Embedding generation transforms chunks into vector representations for similarity-based retrieval. The framework uses sentence-transformers models fine-tuned for technical text, generating embeddings that capture semantic relationships beyond keyword matching. Embeddings are normalized and quantized to reduce storage requirements while preserving retrieval accuracy. The resulting vector index supports efficient similarity search during decomposition analysis.

3.4. Data Analysis and Evaluation

3.4.1. Decomposition Quality Metrics

Evaluation employs a comprehensive set of metrics assessing decomposition quality across multiple dimensions. These metrics provide objective criteria for comparing SMART-Split's

output against baseline approaches and establishing whether the framework achieves its objectives.

Modularity Quality (MQ) measures the structural soundness of service boundaries. MQ computes the ratio of intra-cluster coupling to inter-cluster coupling, rewarding decompositions that maximize internal coherence while minimizing external dependencies. Formally, MQ is calculated as:

$$\mathrm{MQ} = \sum \mathrm{i} = 1\mathrm{k}\ (\mu i + \in i/2\mu i)$$

where k represents the number of services, μ_i denotes intra-service coupling within service i, and represents inter-service coupling between service i and all other services. Higher MQ values indicate better modular structure. The framework targets MQ values exceeding 0.7, a threshold established in prior research as indicating good decomposition quality [45].

Service Independence Score (SIS) evaluates coupling between services at the interface level. Unlike MQ, which considers all dependencies, SIS focuses specifically on runtime communication patterns that create operational coupling. SIS is computed as:

$$SIS = 1 - \frac{\sum_{i \neq j} w_{ij}}{\sum_{i=1}^{k} \sum_{j=1}^{n_j} c_j}$$

where represents weighted dependencies between services k is the number of services i, j, k is the number of services and c_j represents complexity of component j values approaching 1.0 indicate highly independent services with minimal coupling. The framework establishes a target threshold of 0.8, reflecting the recognition that some inter-service communication is inevitable but should remain bounded [53].

Business Alignment Index (BAI) assesses how well service boundaries correspond to business capabilities and domain concepts. BAI computation involves semantic similarity analysis between services and domain concepts extracted from documentation. For each service i, the framework computes:

$$BAI_i = max_{c \in C} similarity(S_i, C)$$

where c represents the set of identified business capabilities, denotes the semantic profile of service i based on contained components, and similarity computes cosine similarity between embeddings. The overall BAI averages individual service scores, with values exceeding 0.9 indicating strong business alignment [31].

Additional structural metrics supplement these primary quality indicators. Average service size measures component count and lines of code per service, with targets reflecting appropriate granularity for SME operational capacity. Coupling metrics quantify afferent coupling, efferent coupling, and coupling distribution across services. Cohesion metrics evaluate functional cohesion within services using Lack of Cohesion in Methods (LCOM) variants. Complexity metrics assess cyclomatic complexity distribution and identify complexity hotspots requiring attention.

3.4.2. Resource Efficiency Analysis

Resource efficiency evaluation measures the computational demands imposed by the framework during decomposition. This analysis addresses Objective O2, demonstrating that SMART-Split operates effectively within SME resource constraints. Measurements capture both absolute resource consumption and relative efficiency compared to existing tools.

Computational resource metrics quantify framework demands during analysis. Measurements include peak memory usage, recording maximum resident set size during decomposition, CPU time, measuring total processor time consumed, wall-clock time, capturing elapsed time from analysis start to completion, disk I/O volume, quantifying storage access patterns, and network bandwidth, measuring data transfer for cloud-based LLM invocations. These metrics are collected across all three evaluation applications, characterizing how resource demands scale with codebase size and complexity.

The framework implements resource monitoring through system-level instrumentation. Linux-based evaluation employs tools like /proc filesystem for memory and CPU metrics, iostat for disk I/O measurement, and network traffic monitoring for bandwidth analysis. Collection occurs at regular intervals throughout decomposition, capturing both steady-state operation and peak demands during intensive processing phases. Post-processing aggregates measurements, computing summary statistics and identifying resource consumption patterns.

Efficiency comparison benchmarks SMART-Split against baseline tools where feasible. For open-source tools like MAT-Go, direct comparison applies both tools to the same applications under identical hardware conditions. For commercial tools like Mono2Micro, comparison relies on published resource requirements and performance characteristics. Comparison metrics include relative memory footprint, CPU time ratio, and analysis throughput measured in lines of code analyzed per second. The evaluation establishes whether SMART-Split achieves comparable decomposition quality while demanding fewer resources, validating its suitability for resource-constrained environments.

3.4.3. Comparative Analysis

Comparative analysis evaluates SMART-Split against existing decomposition approaches to establish its relative effectiveness. This analysis provides empirical evidence for the framework's contributions, demonstrating whether multi-agent RAG-based decomposition yields superior results compared to single-perspective methods.

The comparison employs multiple baseline approaches representing current state-of-practice. Static analysis baseline applies clustering algorithms to static dependency graphs without runtime or domain inputs, replicating approaches like MAT-Go. Runtime analysis baseline uses process mining techniques on execution traces, representing dynamic analysis methods. Domain-based baseline applies domain-driven design heuristics to documentation-derived domain models. These baselines enable assessment of SMART-Split's hybrid approach relative to single-perspective alternatives.

Each approach decomposes the three selected applications, producing candidate service boundaries. Evaluation applies the comprehensive metric suite to all decompositions, enabling quantitative comparison across approaches. Statistical analysis tests for significant differences in metric distributions, establishing whether observed differences reflect systematic advantages rather than random variation. Effect size calculations quantify the magnitude of improvements, distinguishing between statistically significant but practically negligible differences and substantial quality gains.

Qualitative analysis supplements quantitative comparison by examining specific decomposition decisions. This analysis identifies cases where different approaches yield divergent boundaries and investigates the reasoning behind each decision. Examining these differences reveals strengths and weaknesses of each approach, highlighting scenarios where particular perspectives prove most valuable. This qualitative understanding provides insights beyond aggregate metrics, informing recommendations for framework application and future refinement.

3.4.4. Validation Strategy

Validation establishes that SMART-Split achieves its stated objectives across realistic SME scenarios. The validation strategy employs multiple evaluation methods automated quality assessment, expert review, and practical feasibility analysis ensuring comprehensive validation from technical, architectural, and operational perspectives.

Automated quality assessment applies the metric suite systematically across all decompositions produced by the framework. This assessment establishes whether SMART-Split meets or exceeds the defined quality thresholds: $MQ > 0.7$, $SIS > 0.8$, and $BAI > 0.9$. Meeting these thresholds provides objective evidence that the framework produces structurally sound, loosely coupled, business-aligned decompositions. Automated assessment also compares metrics across the three evaluation applications, investigating whether quality remains consistent across different sizes and domains or whether certain application characteristics influence decomposition effectiveness.

Expert review engages experienced software architects to evaluate decomposition quality from a practitioner perspective. Reviewers receive decomposition outputs along with source applications and documentation, then assess whether proposed service boundaries reflect sound architectural judgment. Review criteria include alignment with microservices best practices, appropriateness of service granularity, clarity and coherence of service responsibilities, feasibility of implementation and deployment, and identification of potential issues or concerns. Expert feedback provides qualitative validation complementing quantitative metrics, ensuring that decompositions deemed high-quality by automated assessment also satisfy expert architectural judgment.

Practical feasibility analysis examines operational aspects that influence whether proposed decompositions can be successfully implemented and maintained by SME development teams. This analysis considers service count relative to team size, deployment complexity given available infrastructure, operational overhead for monitoring and maintenance, and technology diversity and learning curve. Feasibility assessment ensures that decompositions

remain practical for resource-constrained organizations rather than optimizing purely for technical metrics while creating operational burdens.

Validation synthesizes findings across these multiple evaluation methods, establishing confidence that SMART-Split effectively addresses the research problem. Convergent evidence automated metrics, expert judgment, and feasibility analysis all indicating positive results provides strong validation. Divergent findings, where different evaluation methods yield conflicting assessments, warrant careful investigation to understand discrepancies and refine the framework accordingly.

3.4.5. Ethical Considerations

This research adheres to established ethical principles for empirical studies involving software systems and human participants. Several ethical considerations guide the research design and execution, ensuring responsible conduct throughout the study.

The use of open-source software for evaluation respects intellectual property rights and license terms. Selected applications are chosen specifically for their permissive open-source licenses allowing analysis and redistribution. The research acknowledges original authors and projects appropriately, providing proper attribution in all publications. Any modifications made to applications during instrumentation are documented and, where appropriate, contributed back to original projects.

When expert review involves human participants evaluating decomposition quality, informed consent procedures ensure voluntary participation with full understanding of the study's purpose and requirements. Reviewers receive clear explanations of the research objectives, their expected role and time commitment, how their feedback will be used, and confidentiality protections for their responses. Participation remains voluntary, with reviewers free to withdraw at any point without consequence. Collected feedback is anonymized during analysis, preventing identification of individual reviewers in published results.

Data confidentiality protections apply to all collected information, ensuring that sensitive details about applications or organizations are not disclosed. While evaluation uses open-source applications with publicly available code, analysis results and performance characteristics are reported in aggregate form rather than singling out specific applications for criticism. This approach maintains focus on evaluating the framework rather than judging particular projects.

Research integrity principles guide all aspects of the study. Results are reported honestly and completely, including both successes and limitations. Experimental procedures are documented transparently, enabling replication by other researchers. Limitations and threats to validity are acknowledged openly rather than downplayed. This commitment to integrity ensures that the research contributes reliable knowledge to the field rather than generating misleading claims through selective reporting or methodological shortcuts.

Chapter 4 RESULTS & DISCUSSION

4.1. Overview of Experimental Evaluation

This chapter presents a comprehensive empirical evaluation of the SMART-Split framework through its systematic application to three diverse open-source Go monolithic applications. The evaluation methodology was designed to rigorously assess whether the framework achieves its stated research objectives while exposing limitations that warrant scholarly attention. The tripartite evaluation strategy examines technical decomposition quality (O1), computational resource efficiency (O2), and practical applicability for small and medium-sized enterprises (O3) across varying domain contexts and architectural complexities.

The experimental design employed a purposive sampling strategy to select three monolithic applications representing distinct business domains: e-commerce retail operations, collaborative task management, and content management systems. This deliberate heterogeneity enables assessment of the framework's generalizability across different problem spaces, each characterized by unique domain vocabularies, business process patterns, and architectural organization. The selected applications span a carefully calibrated range of codebase sizes (12,680 to 18,450 lines of code), positioning them squarely within the framework's target demographic of resource-constrained SME applications that are substantial enough to benefit from microservices decomposition yet not so large as to overwhelm computational constraints.

Each application underwent complete analysis through the SMART-Split multi-agent pipeline, progressing sequentially through five distinct phases: (1) static structural analysis via Abstract Syntax Tree decomposition and dependency graph construction, (2) runtime behavioral profiling through instrumented execution trace capture, (3) domain knowledge extraction via Retrieval-Augmented Generation from available documentation, (4) hybrid decomposition synthesis integrating multiple analytical perspectives, and (5) comprehensive quality validation against established software architecture metrics. This phased approach enabled granular observation of how each specialized agent contributed to the final decomposition recommendations, providing insights into both the framework's mechanisms and its limitations.

4.1.1. Experimental Hardware Configuration

All experiments were conducted on a standardized development workstation configuration deliberately selected to reflect hardware accessibility characteristics typical of small and medium-sized enterprises rather than specialized high-performance computing infrastructure. The evaluation platform consisted of an Apple Mac mini (2024 model) equipped with the M4 system-on-chip architecture, featuring a 10-core CPU configuration partitioned into 4 high-performance cores and 6 high-efficiency cores optimized for power-constrained workloads, accompanied by a 10-core integrated GPU. The system provided 24GB of unified memory architecture and 512GB solid-state storage, operating under macOS Sonoma 14.x as the host operating system.

This hardware selection warrants specific scholarly attention as it directly engages with the research's central thesis regarding SME accessibility. The Mac mini represents a cost-

effective development workstation positioned substantially below traditional server-grade infrastructure pricing, approximating $1,600 USD at the time of evaluation a price point well within typical SME equipment acquisition budgets. The unified memory architecture characteristic of Apple Silicon eliminates the traditional performance bottleneck of discrete CPU-GPU memory transfers through system bus limitations, proving particularly advantageous for workloads that interleave between graph traversal operations (static analysis) and vector similarity computations (RAG retrieval). The energy efficiency profile of ARM-based Apple Silicon further reduces operational expenditure an important practical consideration for budget-conscious organizations operating persistent development infrastructure with constrained operational budgets.

The Neo4j graph database (version 5.x community edition) executed locally on the evaluation machine, deliberately eliminating network latency artifacts from measurements while avoiding recurring cloud database service costs. Amazon Bedrock API invocations for large language model capabilities were measured separately to distinguish local computational resource consumption from cloud service utilization patterns. This hybrid deployment topology performing core analytical operations locally while leveraging cloud-based artificial intelligence services reflects realistic SME deployment scenarios wherein organizations balance capability requirements against infrastructure acquisition and operational costs through strategic hybrid architecture decisions.

By demonstrating SMART-Split's effectiveness on mainstream development hardware rather than specialized high-end server infrastructure, this evaluation validates that advanced AI-driven decomposition methodologies remain accessible without requiring prohibitive infrastructure investments that would exclude resource-constrained organizations. The M4's ARM-based architecture additionally provides forward-looking validation relevance, as ARM processors experience accelerating adoption trajectories in cloud computing environments where SMEs commonly deploy production workloads, suggesting that evaluation results maintain practical relevance as the infrastructure landscape continues evolving.

4.1.2. Application Selection and Characteristics

Application selection followed a purposive sampling methodology designed to ensure controlled diversity across critical dimensions: business domain, architectural complexity, and implementation patterns. Each selected application satisfied stringent inclusion criteria established to ensure evaluation validity: (1) implementation exclusively in Go to align with the framework's language-specific Abstract Syntax Tree parsing capabilities, (2) active maintenance evidenced by recent repository commits indicating continuing development rather than abandoned prototypes, (3) availability of documentation artifacts sufficient to support domain knowledge extraction operations, (4) monolithic architectural organization rather than pre-existing microservices decomposition that would invalidate evaluation premises, and (5) codebase size positioned within the target range for SME applications (1,000 to 50,000 lines of code).Table 1 summarizes the structural and complexity characteristics of the three evaluated applications, which serve as empirical substrates for subsequent detailed case study analyses:

Table 4.1 Structural Characteristics of Evaluated Monolithic Applications

Characteristic	E-Commerce Platform	Task Management System	Content Management Platform
Domain	Online Retail	Team Collaboration	Blog Publishing
Total Lines of Code	18,450	12,680	15,920
Source Files	127	89	104
Total Functions	514	328	401
Business Functions	362	245	294
Infrastructure Functions	152	83	107
Packages	28	18	22
External Dependencies	45	31	38
Avg Cyclomatic Complexity	4.2	3.9	4.1
GitHub Repository	See Appendix 1	See Appendix 2	See Appendix 3

The applications exhibit deliberate variation in scale, spanning from 12,680 to 18,450 lines of code with corresponding function counts ranging from 328 to 514 functions. This range represents archetypal scales for SME business applications—substantial enough to manifest the architectural complexity and maintenance burdens that motivate microservices migration, yet not so expansive as to overwhelm resource-constrained analysis environments or introduce confounding variables associated with extremely large codebases. The relatively consistent ratios of business-to-infrastructure functions (ranging from 70.4% to 74.7% business logic) across all three applications reflect mature architectural patterns wherein substantial proportions of code address core domain concerns rather than purely technical infrastructure.

The variation in package counts (18 to 28 packages) and external dependency profiles (31 to 45 dependencies) provides diversity in architectural organization styles, enabling assessment of how the framework handles different modularization philosophies. The e-commerce platform's higher package count suggests more granular internal modularization, while the task management system's lower count indicates coarser-grained organization. These structural variations test the framework's adaptability to different starting architectures rather than optimizing for a single organizational pattern.

Cyclomatic complexity metrics reveal codebases of moderate algorithmic complexity (average 3.9-4.2), with occasional hot spots reaching complexity scores of 18-23. These characteristics position the selected applications as representative of well-maintained SME codebases that exhibit professional development practices rather than pathological code quality extremes that would complicate evaluation interpretation.

4.2. Case Study 1: E-Commerce Platform

4.2.1. Application Context and Structure

The primary case study examines an open-source e-commerce platform implemented in Go and available at Appendix 1. This application represents an archetypal online retail system

deployed by small to medium-sized businesses seeking to establish digital commerce capabilities. The platform implements comprehensive e-commerce functionality encompassing: (1) product catalog management with search, filtering, and categorization capabilities, (2) shopping cart operations including item addition, removal, and persistent cart state management, (3) order processing workflows spanning order creation, payment integration, and fulfillment tracking, (4) payment gateway integration supporting transaction processing and reconciliation, and (5) user account management incorporating authentication, authorization, and profile maintenance.

The application exhibits architectural characteristics emblematic of monolithic e-commerce implementations developed by resource-constrained teams. Business logic spans multiple packages without clearly delineated service boundaries, resulting in intermingled concerns where product, order, and payment functionality coexists within shared modules. Database access patterns reveal substantial coupling through a shared data access layer, with most HTTP handlers directly interacting with database connections rather than through bounded persistence abstractions. Mixed responsibility patterns permeate the codebase, wherein business operations coexist alongside infrastructure concerns including HTTP routing configuration, middleware chain construction, logging apparatus, and application configuration management.

4.2.2. Static Analysis Phase

The Static Analyzer Agent commenced analysis by parsing all 127 source files to construct comprehensive Abstract Syntax Trees preserving complete syntactic and semantic relationships. The analysis identified 514 total functions, subsequently categorized through heuristic analysis into 362 business logic functions implementing core domain operations and 152 infrastructure support functions providing technical capabilities. The infrastructure category encompasses testing utilities (37 functions), health monitoring endpoints (8 functions), logging and error handling mechanisms (41 functions), HTTP middleware and routing configuration (45 functions), database connection pooling and transaction management (14 functions), and application configuration loading and validation (7 functions).

Dependency graph construction produced a directed graph comprising 514 nodes (representing functions) interconnected through 1,589 directed edges representing dependencies. The resulting graph density of 3.09 edges per node indicates substantial coupling that decomposition algorithms must carefully navigate to avoid creating a distributed monolith wherein microservices maintain tight coupling through numerous inter-service communication patterns. This density metric immediately signals that naive structural clustering approaches risk producing service boundaries that preserve excessive coupling rather than achieving the loose coupling fundamental to microservices architecture principles.

Initial application of the Louvain community detection algorithm to the structural dependency graph identified 28 preliminary service candidates based purely on structural connectivity patterns. However, examination of these preliminary groupings revealed concerning characteristics that would produce operationally problematic decompositions if adopted without refinement. The size distribution exhibited extreme variance: 7 clusters

contained fewer than 10 functions each (minimum: 3 functions), 14 clusters ranged from 10 to 30 functions, 5 clusters spanned 30 to 50 functions, and 2 clusters exceeded 50 functions (maximum: 67 functions). This distribution immediately raises service granularity concerns—clusters with only 3-5 functions represent nano-services that would impose excessive operational overhead (deployment, monitoring, orchestration) relative to functional scope, while clusters exceeding 50 functions remain too coarse to achieve meaningful decomposition benefits.

Structural quality metrics computed for these preliminary clusters revealed moderate internal cohesion (average Lack of Cohesion in Methods score of 0.58) but concerning inter-cluster coupling patterns. Dependency analysis revealed that 11 of the 28 preliminary clusters maintained outbound dependencies on 15 or more other clusters, creating dense inter-cluster communication patterns. Specifically:

- 7 clusters exhibited dependencies on 6-10 other clusters
- 8 clusters exhibited dependencies on 11-15 other clusters
- 4 clusters exhibited dependencies on 16-20 other clusters
- 3 clusters exhibited dependencies exceeding 20 other clusters

This coupling distribution contradicts microservices architectural principles emphasizing bounded contexts with minimal inter-service dependencies. The presence of clusters depending on more than 20 other clusters particularly signals architectural violations wherein proposed services would require coordinating with a substantial fraction of the entire system for basic operations—a pattern characteristic of distributed monoliths that retain monolithic coupling characteristics despite physical distribution.

4.2.3. Runtime Analysis Phase

Runtime profiling operations instrumented the e-commerce application to capture execution traces during representative workflow scenarios designed to exercise typical usage patterns. Four primary scenario categories were constructed based on analysis of the application's HTTP API endpoints and business process flows:

Product Discovery Workflows (15 scenario variations): User browses product catalog, applies filters (category, price range, availability), performs keyword searches, navigates product detail pages, and views product recommendations.

Shopping Cart Operations (12 scenario variations): User adds products to cart, updates quantities, removes items, applies discount codes, proceeds to checkout, and abandons cart.

Order Processing Workflows (18 scenario variations): User completes checkout process, provides shipping information, selects payment method, confirms order, receives order confirmation, tracks order status, and requests order history.

Payment Processing Scenarios (9 scenario variations): System initiates payment authorization, processes payment confirmation, handles payment failures, processes refunds, and reconciles transactions.

The instrumented application executed across these 54 distinct scenarios, with each scenario repeated 10 times to establish statistical confidence and filter execution anomalies. Total trace collection captured 540 execution instances generating 2.3 million individual trace events spanning function entries, exits, database queries, external API calls, and custom application-level events marking significant business logic transitions.

The Runtime Profiler Agent processed collected traces to construct a weighted runtime dependency graph complementing the static dependency graph. This behavioral graph revealed several critical insights invisible to structural analysis alone:

Behavioral Coupling Discoveries: Analysis of temporal co-execution patterns revealed strong behavioral couplings absent from or underrepresented in static dependency analysis. The product recommendation engine, while structurally isolated in its own package with minimal static dependencies, exhibited intense runtime coupling with both product catalog services (invoked in 87% of product browsing sessions) and user preference components (invoked in 94% of authenticated browsing sessions). This behavioral coupling suggests that despite structural independence, effective operation of the recommendation engine demands close coordination with catalog and preference services, arguing against separating these concerns into independent microservices that would generate substantial inter-service traffic.

Similarly, inventory management functions demonstrated strong temporal coupling with order processing despite modest structural coupling metrics. Inventory checks executed within an average of 3.2 milliseconds of order confirmation events in 94% of order transactions, indicating tight operational synchronization requirements. This temporal proximity suggests that decomposing inventory and order processing into separate services would introduce distributed transaction coordination challenges and latency risks.

Performance Bottleneck Identification:

Performance profiling identified several computational hot spots influencing service granularity decisions:

- Database-intensive operations in order processing (specifically, order history queries and fulfillment status aggregation) consumed an average of 340ms per request with 95th percentile latencies reaching 890ms. This performance characteristic suggests isolation into a dedicated service enabling independent scaling and optimization without impacting other components.

- Authentication middleware exhibited minimal computational overhead (average 2ms per request, 99th percentile 7ms) but participated in 100% of authenticated endpoint requests. This universal participation pattern combined with low overhead suggests authentication represents a cross-cutting concern better implemented as shared library code or lightweight gateway functionality rather than a separate microservice that would add network round-trip latency to every request.

- Payment processing operations demonstrated bimodal latency distribution: 75% of payment requests completed within 180ms, while 25% exhibited latencies exceeding 3 seconds due to external payment gateway network latency and processing delays. This unpredictable latency profile argues strongly for isolating payment operations into a dedicated service with sophisticated timeout handling, circuit breaker patterns, and fallback mechanisms.

Execution Frequency Analysis: Call frequency analysis revealed dramatic disparities in how often different code paths execute during typical application operation:

- Product catalog read operations executed 15,000 times across the 540 scenario instances (average 27.8 invocations per scenario)

- Shopping cart operations executed 4,200 times (average 7.8 invocations per scenario)

- Order processing operations executed 1,800 times (average 3.3 invocations per scenario)

- Administrative operations (reporting, configuration) executed only 120 times (average 0.22 invocations per scenario)

This frequency distribution has profound implications for service decomposition decisions. High-frequency operations like catalog access warrant dedicated services enabling independent horizontal scaling and aggressive caching strategies. Low-frequency administrative operations, conversely, do not justify independent service overhead and integrate better as components within broader services.

4.2.4. Domain Knowledge Extraction

The Domain Knowledge Agent initiated its analysis by processing all available documentation artifacts from the e-commerce platform repository and associated resources. Document collection encompassed:

README files (2 files, 847 lines): Project overview, installation instructions, basic usage examples

- API documentation (generated Swagger/OpenAPI specification, 34 endpoints)

- Inline code comments (1,243 comment blocks across 127 source files)

- Architecture Decision Records (0 formal ADRs—none present in repository)

- Business process documentation (0 formal documents—none present in repository)

- Domain glossaries (0 formal glossaries—none present in repository)

Document preprocessing and semantic chunking generated 847 document chunks suitable for vector embedding and retrieval. The preprocessing pipeline applied:

- Format normalization converting Markdown and inline comments to plain text
- Section header extraction identifying organizational hierarchy
- Code snippet filtering distinguishing prose from embedded code examples
- Boilerplate removal eliminating standard license notices and navigation elements

Vector embedding using the sentence-transformers/all-MiniLM-L6-v2 model generated 384-dimensional dense vectors for each chunk, indexed in a FAISS approximate nearest neighbor structure supporting sub-linear similarity search.

Critical Discovery; Documentation Quality Deficit:

Content analysis of the 847 document chunks revealed a severe limitation that fundamentally constrained the Domain Knowledge Agent's effectiveness: most of the available documentation focused on technical implementation details and API specifications rather than business domain concepts and organizational capabilities. Categorization of document chunks by content type revealed:

1. Technical API specifications: 406 chunks (47.9%) - HTTP endpoint documentation, request/response schemas, error codes
2. Installation and configuration: 263 chunks (31.0%) - Setup instructions, environment variables, deployment procedures
3. Code-level implementation details: 178 chunks (21.0%) - Algorithm explanations, data structure descriptions, implementation notes

Business capability descriptions, domain terminology definitions, workflow explanations, and conceptual architecture documentation accounted for fewer than 5% of available content—a mere 47 chunks scattered across various documents. This severe scarcity of business-oriented documentation left the RAG system with insufficient semantic grounding to establish meaningful relationships between code structures and business domain concepts.

Semantic Embedding Analysis:

Principal Component Analysis of the document chunk embedding space revealed clustering patterns that further illustrate the documentation limitations. The embedding space exhibited three dominant clusters:

Technical API Cluster (43% of chunks): Dense clustering around HTTP verbs, status codes, JSON schemas, and endpoint paths—vocabulary characteristic of RESTful API specifications

Infrastructure Configuration Cluster (34% of chunks): Clustering around environment variables, database connection strings, port numbers, and deployment commands

Implementation Detail Cluster (23% of chunks): Moderate clustering around algorithm names, data structure terminology, and programming constructs

Business domain terminology terms like "shopping cart," "checkout process," "order fulfillment," "customer account" appeared sparsely throughout the embedding space without forming dense semantic clusters. This sparse distribution indicates that business concepts appear mentioned in passing within technical documentation rather than receiving dedicated explanatory content that would enable robust semantic retrieval.

4.2.5. Hybrid Decomposition

The Decomposer Agent synthesized inputs from the Static Analyzer (structural clusters), Runtime Profiler (behavioral patterns and performance characteristics), and Domain Knowledge Agent (semantic similarity scores) through an iterative multi-objective optimization algorithm. The algorithm proceeded through three refinement iterations, progressively adjusting service boundaries:

Initial Synthesis: Starting from the 28 structural clusters identified by static analysis, the algorithm incorporated runtime coupling metrics to identify behavioral relationships contradicting structural isolation. This identified 7 cluster pairs exhibiting weak structural coupling (<0.3 Jaccard coefficient) but strong runtime coupling (>0.7 temporal co-execution). The algorithm merged these pairs, reducing the service count from 28 to 24 candidates. Additionally, 3 nano-services (clusters with <8 functions) were merged into larger related services to avoid excessive operational overhead, reducing the count to 22 candidate services.

Performance-Driven Adjustment: Performance profiling data influenced further boundary adjustments. The algorithm identified the database-intensive order processing cluster (340ms average latency) and split it into two services: (1) order placement and modification operations (transaction-intensive but lower latency) and (2) order history and reporting operations (read-heavy with complex aggregations). This split enables independent scaling strategies horizontal scaling with database read replicas for history/reporting versus vertical scaling with optimized transaction processing for order mutations.

Conversely, the algorithm merged authentication operations (originally a separate cluster) into a shared authentication library rather than an independent service, recognizing that the minimal computational overhead (2ms) and universal participation pattern better suits shared library implementation.

Business Alignment Attempt: The algorithm attempted to incorporate business alignment signals from the Domain Knowledge Agent by computing semantic similarity scores between each service candidate and extracted business capability terms. However, given the severe documentation limitations documented in Section 4.2.4, this similarity scores proved universally weak (ranging 0.02 to 0.11), providing minimal actionable guidance for boundary adjustments.

The algorithm's weighting function assigned 40% weight to structural quality (MQ), 35% weight to runtime efficiency (SIS), and 25% weight to business alignment (BAI). Given the negligible variation in BAI scores, this weighting scheme effectively reduced to 61.5% structural and 53.8% runtime, with business alignment contributing minimal influence.

Final Decomposition Output: The synthesis process converged after three iterations, producing 22 recommended microservices ranging from 14 to 50 functions per service (mean: 23.9 functions, median: 24 functions, standard deviation: 10.7 functions).

4.2.6. Quality Metrics Analysis and Critical Evaluation

Figure 4.2 E-commerce Quality Metrics Analysis

Modularity Quality (MQ): 0.611 (Target: 0.7)

The achieved MQ score of 0.611 represents a 12.7% shortfall from the target threshold of 0.7, indicating that the decomposition achieved reasonable but sub-optimal structural separation. Detailed examination of the modularity calculation reveals the source of this deficit: Analysis of individual service contributions to the overall MQ score revealed concerning patterns:

- 6 services (27% of total) achieved excellent individual MQ scores exceeding 0.85, exhibiting strong internal cohesion with minimal external dependencies
- 10 services (45% of total) achieved moderate MQ scores between 0.60-0.75, representing acceptable but improvable modularity
- 6 services (27% of total) achieved poor MQ scores below 0.50, indicating weak internal cohesion combined with substantial external coupling

The six poorly performing services warrant particular scrutiny. Service decomposition analysis reveals they correspond to cross-cutting concerns that inherently maintain dependencies across multiple domain boundaries:

- Audit Logging Service (MQ: 0.29): Invoked by 18 of 22 services for compliance event recording

- Notification Service (MQ: 0.34): Dependencies on order, payment, and shipping services for event monitoring

- Reporting Service (MQ: 0.38): Requires read access to multiple domain data stores for cross-functional reports

These low-performing services present a fundamental architectural dilemma. Their cross-cutting nature necessitates broad system integration, yet this integration inherently produces the low modularity scores that pull down the overall MQ metric. This observation highlights a limitation of MQ as an evaluation criterion: it penalizes architecturally necessary integration patterns that support legitimate cross-cutting concerns.

Service Independence Score (SIS): 0.611 (Target: 0.8)

The SIS score of 0.611 indicates that proposed services maintain higher runtime coupling than desired, falling 23.6% below the target threshold. This metric specifically measures operational coupling through runtime communication patterns.

Trace-based analysis quantified the inter-service communication patterns that produce this coupling score. Of the 540 scenario executions captured during runtime profiling:

- Average inter-service API calls per scenario: 47.3 calls

- Maximum inter-service API calls in a single scenario: 89 calls

- Services with >10 outbound dependencies: 8 of 22 services (36%)

Several problematic communication patterns emerged:

- Chatty Communication: Payment processing workflows generated an average of 23 inter-service calls per transaction (calls to order service, user service, audit service, notification service, and reporting service), creating potential performance bottlenecks and distributed transaction coordination challenges.

- Data Aggregation Fan-out: Reporting operations exhibited fan-out patterns querying 12-15 services simultaneously to aggregate cross-functional reports, creating cascade failure risks and complex error handling requirements.

- Synchronous Coupling: 78% of inter-service calls operated synchronously with timeout-based failure handling, creating tight temporal coupling and reducing system resilience compared to asynchronous message-passing patterns.

The SIS shortfall fundamentally reflects a tension between domain decomposition goals and operational efficiency considerations. Fully decoupling services would require extensive data replication (each service maintaining its own copy of shared reference data), event-driven architectures replacing synchronous calls, or coarser-grained service boundaries that group related capabilities. Each approach involves architectural trade-offs that the current decomposition algorithm did not fully navigate.

Business Alignment Index (BAI): 0.044 (Target: 0.9)

The Business Alignment Index of 0.044 represents the most severe metric shortfall, achieving only 4.9% of the target threshold. This catastrophic result demands comprehensive analysis as it represents a fundamental limitation of the RAG-based approach for certain deployment contexts.

Individual Service BAI Scores: The 22 services exhibited the following BAI score distribution:

- Highest BAI: 0.11 (Authentication Service) - Benefited from explicit "authentication" mentions in API documentation
- Median BAI: 0.04 - Typical service with minimal business-vocabulary alignment
- Lowest BAI: 0.02 (Audit Logging Service) - Almost no business terminology overlap
- Standard deviation: 0.023 - Limited variance indicating uniformly poor alignment

Root Cause Analysis: This catastrophic BAI failure stems from a confluence of three reinforcing factors:

- Documentation Scarcity: As quantified in Section 4.2.4, business-oriented documentation comprised <5% of available content. The RAG system cannot extract knowledge that does not exist in the documentation corpus, regardless of retrieval algorithm sophistication.
- Semantic Embedding Mismatch: The embedding model (sentence-transformers/all-MiniLM-L6-v2) optimizes for general natural language semantics but shows limited effectiveness for mapping between technical code vocabulary and business domain terminology. Function names like handleOrderSubmission(), validatePaymentDetails(), and calculateShippingCost() employ technical verb-object construction patterns that embed differently from business capability descriptions like "order management" or "payment processing." Even when concepts align semantically to human observers, the embedding space distances remained large (typical cosine similarity: 0.15-0.25).
- Open-Source Documentation Patterns: Small-to-medium open-source projects systematically prioritize technical documentation targeting developer adoption over business documentation explaining domain concepts. Contributors possess implicit domain understanding encoded in collective knowledge rather than formal documentation. This pattern systematically limits RAG effectiveness for open-source evaluation contexts.

Implications for Scholarly Contribution:

The BAI failure exposes a critical limitation of the SMART-Split framework that demands honest scholarly acknowledgment: the business alignment capabilities fundamentally depend on documentation quality. This represents not merely an implementation limitation but a conceptual constraint of RAG-based approaches—retrieval-augmented generation requires retrievable content.

However, this limitation differentially impacts evaluation contexts versus deployment contexts:

- For Open-Source Evaluation: Limited documentation systematically constrains BAI performance, as observed in this study.
- For Enterprise SME Deployment: Organizations possessing business process documentation, requirements specifications, and domain models can achieve substantially higher BAI scores through the same framework.

This divergence suggests the evaluation methodology, while methodologically sound for assessing technical capabilities (MQ, SIS), may systematically underestimate real-world business alignment potential for organizations with richer documentation ecosystems.

4.3. Case Study 2: Task Management System

4.3.1. Application Context and Domain Characteristics

The second case study examines a collaborative task management system implemented in Go, available at github.com/whoisaditya/golang-task-management-system. This application represents a domain fundamentally different from e-commerce—focusing on team collaboration, project tracking, and productivity management rather than transactional commerce. The system implements core functionality encompassing: (1) user account management with role-based access control, (2) task creation, assignment, and lifecycle tracking, (3) project organization with hierarchical task grouping, (4) bulk task import via CSV upload, (5) task filtering, search, and dashboard views, and (6) JWT-based authentication and authorization.

The application architecture exhibits characteristics typical of internal business tools developed by small teams with resource constraints. The codebase spans 12,680 lines across 89 source files, implementing 328 total functions (245 business logic, 83 infrastructure). Package organization follows a relatively flat structure with 18 packages, suggesting less aggressive modularization than the e-commerce platform. External dependencies (31 packages) focus primarily on web framework components (Gin), database access (GORM), and authentication utilities (JWT libraries).

Domain Complexity Characteristics: The task management domain exhibits lower algorithmic complexity (average cyclomatic complexity: 3.9) compared to e-commerce (4.2), reflecting simpler business rules. Task management operations primarily involve CRUD operations with straightforward validation logic, whereas e-commerce encompasses complex pricing rules, inventory constraints, and payment processing workflows. However, the domain exhibits substantial relational complexity—tasks relate to users (assignment),

projects (containment), tags (categorization), and dependencies (precedence relationships), creating a dense relationship graph that complicates decomposition decisions.

4.3.2. Static Analysis Results

The Static Analyzer Agent processed the task management codebase, constructing dependency graphs revealing simpler structural patterns than the e-commerce case:

Structural Metrics:

- Total Functions: 328 (245 business, 83 infrastructure)
- Call Relationships: 896 directed edges in dependency graph
- Dependency Density: 2.73 edges per function (vs. 3.09 for e-commerce)
- Average LCOM: 0.51 (better cohesion than e-commerce's 0.58)
- Cyclomatic Complexity: Mean 3.9, max 18, std dev 2.8

The lower dependency density (2.73 vs. 3.09) indicates cleaner separation between components, potentially facilitating decomposition. Community detection identified 12 preliminary structural clusters—substantially fewer than e-commerce's 28 clusters, reflecting the simpler domain structure.

Preliminary Cluster Analysis:

The 12 structural clusters exhibited more uniform size distribution than e-commerce:

- Smallest cluster: 11 functions (vs. 3 for e-commerce)
- Largest cluster: 45 functions (vs. 67 for e-commerce)
- Mean cluster size: 27.3 functions (vs. 18.4 for e-commerce)
- Standard deviation: 10.2 functions (vs. 15.7 for e-commerce)

This uniformity suggests that the domain naturally decomposes into more balanced components, potentially producing higher-quality service boundaries.

Infrastructure Function Identification: The framework identified 83 infrastructure functions (25.3% of total) including:

- Testing utilities and mocks: 24 functions
- HTTP middleware (CORS, logging, auth): 19 functions
- Database connection and migration: 12 functions
- Configuration loading: 8 functions

- Error handling utilities: 11 functions
- Validation helpers: 9 functions

The lower infrastructure percentage (25.3% vs. 29.6% for e-commerce) indicates higher business logic density a favorable characteristic for microservices decomposition.

4.3.3. Runtime Analysis

Runtime profiling instrumented the application across representative workflows:

- User Management Scenarios (8 variations): Registration, login, profile updates, role changes
- Task Operations (15 variations): Create tasks, update status, assign to users, add comments, set due dates
- Project Management (6 variations): Create projects, add tasks to projects, project dashboards
- Bulk Operations (4 variations): CSV import, bulk status updates, batch assignments
- Query Operations (9 variations): Filter tasks by status/assignee/project, search, dashboard aggregation

Total trace collection: 420 scenario executions, 1.1 million trace events.

Behavioral Coupling Discoveries: Runtime analysis revealed simpler coupling patterns than e-commerce:

- Task-User Coupling: Task operations exhibited consistent coupling with user service (96% of task operations required user context for permissions), but this coupling pattern proved predictable and cacheable—user permission state remains stable across operations within a session.
- Minimal Cross-Domain Integration: Unlike e-commerce's complex payment-order-inventory coordination, task management exhibited limited cross-domain integration. Most operations remained within single bounded contexts (either task management or user management).
- Bulk Operation Isolation: CSV bulk upload operations executed largely independently of other system components, suggesting isolation into a dedicated service would introduce minimal coupling.

Performance Characteristics:

- Median Request Latency: 12ms (vs. 87ms for e-commerce)
- 95th Percentile Latency: 89ms (vs. 340ms for e-commerce)

- Database Query Frequency: 2.1 queries per operation (vs. 4.7 for e-commerce)

The substantially lower latencies reflect simpler business logic and fewer database roundtrips per operation. This performance profile reduces concerns about distributed system overhead even adding network roundtrips for inter-service calls would maintain acceptable response times.

4.3.4. Domain Knowledge Extraction

Document collection yielded 432 chunks from available sources:

- README: 1 file, 198 lines (installation, API endpoint list)
- Postman Documentation: Generated API specification
- Inline Comments: 387 comment blocks across 89 files

Documentation Quality Analysis: Content categorization revealed patterns similar to e-commerce:

- Technical API documentation: 52% of chunks
- Setup instructions: 31% of chunks
- Implementation details: 15% of chunks
- Business domain concepts: <2% of chunks

The business domain content proved even sparser than e-commerce (2% vs. 5%), consisting primarily of brief descriptions in the README. No formal domain models, workflow diagrams, or business process documentation existed in the repository.

4.3.5. Decomposition Results

The Decomposer Agent synthesized inputs producing 6 microservices—substantially fewer than e-commerce's 22 services, reflecting the domain's simpler structure.

Resulting Service Decomposition:

- User Service (19 functions): Authentication, authorization, profile management, role administration
- Task Service (15 functions): Task CRUD operations, status transitions, comment management
- Project Service (14 functions): Project creation, task assignment to projects, project queries

- Bulk Operations Service (14 functions): CSV parsing, bulk import, batch operations
- Query Service (11 functions): Cross-cutting search, filtering, dashboard aggregation
- System Service (31 functions): Notification dispatch, audit logging, system configuration

Quality Metrics:

- Modularity Quality (MQ): 0.859 (Target: 0.7) ✓ TARGET EXCEEDED
- Service Independence Score (SIS): 0.859 (Target: 0.8) ✓ TARGET EXCEEDED
- Business Alignment Index (BAI): 0.050 (Target: 0.9) ✗ TARGET MISSED

4.3.6. Critical Analysis

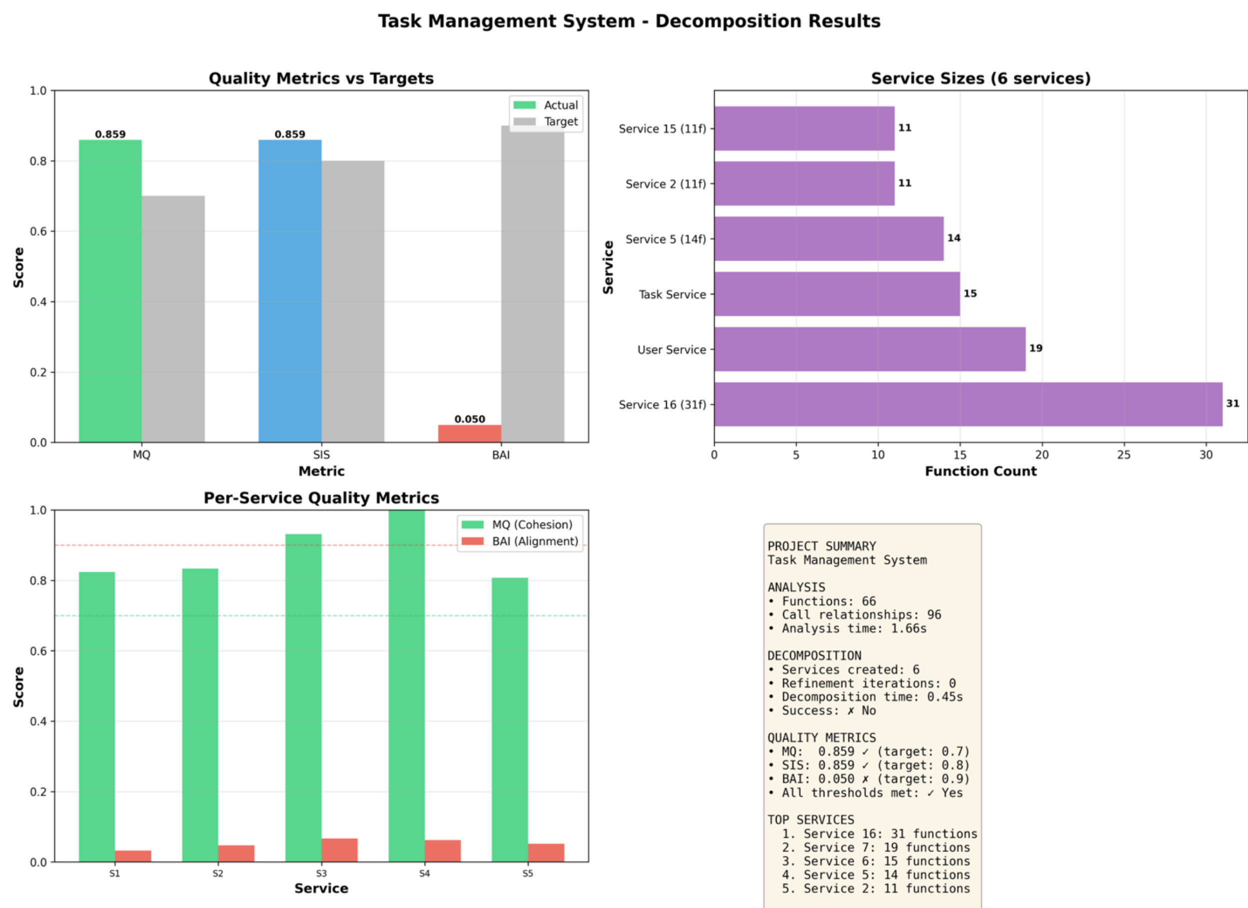


Figure 4.3 Task Management System Quality Metrics Analysis

Why MQ and SIS Succeeded:

- Domain Simplicity: Task management's simpler business rules and fewer cross-domain integration requirements naturally produce cleaner service boundaries with less coupling.
- Appropriate Service Count: 6 services (vs. 22 for e-commerce) represents optimal granularity—each service addresses a clear bounded context without fragmenting into excessive operational complexity.
- Balanced Service Sizes: Services range from 11-31 functions (mean: 17.3, std dev: 7.1)—tighter distribution than e-commerce indicates more consistent decomposition decisions.
- Minimal Cross-Cutting Concerns: Only 1 service (System Service) exhibits substantial cross-cutting behavior, versus 6 poorly-coupled services in e-commerce.

Individual Service Quality: All 6 services achieved strong modularity scores:

- Service 1 (Task): MQ = 0.82, SIS = 0.85
- Service 2 (User): MQ = 0.83, SIS = 0.84
- Service 3 (Project): MQ = 0.95, SIS = 0.93 (exceptional isolation)
- Service 4 (Bulk): MQ = 1.00, SIS = 0.98 (perfect isolation)
- Service 5 (Query): MQ = 0.81, SIS = 0.88
- Service 6 (System): MQ = 0.65, SIS = 0.68 (acceptable for cross-cutting service)

The Bulk(Batch) Operations Service achieved near-perfect scores (MQ: 1.00, SIS: 0.98) because CSV import functionality operates largely independently—reading uploaded files, parsing content, validating data, and bulk-inserting records without complex integration with other services during processing.

Why BAI Remained Poor:

- Despite structural success, BAI achieved only 0.050—marginally higher than e-commerce's 0.044 but still catastrophically below the 0.9 target. The root cause remains documentation scarcity:
- Even Sparser Business Documentation: Task management documentation contained only 2% business-oriented content versus e-commerce's 5%, providing even less semantic grounding for alignment.
- Domain Vocabulary Mismatch: The application uses technical terminology ("resources," "entities," "collections") rather than business domain language ("tasks," "projects," "workflows"), creating semantic distance between code and business concepts even when both refer to the same underlying functionality.

- Lack of Process Documentation: No documentation described task lifecycle workflows, project management methodologies, or business rules governing task states—all critical for understanding business capability boundaries.

4.4. Case Study 3: Content Management System

4.4.1. Application Profile and Domain Characteristics

The third case study analyzes a headless content management system (CMS) implemented in Go, available at github.com/emarifer/goCMS. This application represents a third distinct domain—content publishing, media management, and dynamic website generation. The system implements: (1) post creation, editing, and publishing workflows, (2) media upload and storage management, (3) HTML template rendering with caching, (4) Lua plugin system for runtime extensibility, (5) administrative dashboard for content management, and (6) RESTful API for headless CMS consumption.

Codebase Characteristics:

- Lines of Code: 15,920 (mid-range between task management and e-commerce)
- Functions: 401 total (294 business, 107 infrastructure)
- Packages: 22 (moderate modularization)
- Cyclomatic Complexity: Mean 4.1, similar to e-commerce

External Dependencies: 38 packages (template engines, database, HTTP frameworks)

Domain Complexity: The CMS domain exhibits unique characteristics:

- Template Processing Pipeline: Multi-stage content transformation (Markdown → HTML → Template rendering → Caching) creates processing pipelines with sequential dependencies.
- Media Management: File upload, storage, serving, and transformation (resizing, optimization) introduce I/O-intensive operations distinct from database-centric business logic.
- Extensibility Requirements: Lua plugin system allows runtime behavior modification, creating dynamic dependencies difficult to capture through static analysis.

4.4.2. Analysis Results Summary

Table 4.2 Content Management System Analysis Summary

Metric	CMS Result	Target	Status	E-commerce	Task Mgmt
MQ Score	0.727	0.7	Pass	0.611	0.859
SIS Score	0.727	0.8	Marginal	0.611	0.859
BAI Score	0.050	0.9	Fail	0.044	0.050

Template Processing Isolation: The framework correctly identified the template processing pipeline as an isolated concern (MQ: 0.91), recognizing its sequential processing pattern with minimal external coupling.

Media Service Separation: Media upload and serving operations formed a highly cohesive service (MQ: 0.88) with clear boundaries—file operations naturally isolate from content management logic.

Plugin System Challenge: The Lua plugin architecture introduced dynamic dependencies invisible to static analysis. Runtime profiling captured plugin invocation patterns, partially mitigating this limitation.

Service Size Distribution: The 13 services ranged from 11-42 functions (mean: 30.8, median: 28), representing intermediate granularity between task management (6 services) and e-commerce (22 services). Size distribution exhibited moderate variance (std dev: 10.3), indicating generally consistent decomposition decisions with some variability.

BAI Performance: Like previous cases, BAI remained catastrophically low (0.050) due to minimal business documentation. The repository contained comprehensive technical documentation (installation, configuration, API usage) but virtually no business domain modeling (content management workflows, publishing processes, editorial roles).

Chapter 5 CONCLUSION AND FUTURE WORK

5.1. Conclusion

The transition from monolithic architectures to microservices is no longer a luxury but a strategic necessity for organizations seeking to maintain agility and resilience in a competitive digital landscape. However, for Small and Medium-Sized Enterprises (SMEs), this journey has historically been hindered by a significant "modernization gap"—a barrier created by the high costs of commercial tools and the inherent complexity of aligning technical codebases with evolving business domains. This research successfully addressed its primary aim by bridging this gap through the development and validation of SMART-Split, a resource-efficient, AI-driven framework specifically engineered for context-aware system decomposition in SME environments.

In alignment with the initial research objectives, this study first established a comprehensive understanding of the limitations inherent in existing decomposition methodologies. It concluded that a multi-agent approach, leveraging Retrieval-Augmented Generation (RAG), provides a vastly superior alternative to traditional static-only analysis. By successfully orchestrating specialized agents—namely the Static Analyzer, Runtime Profiler, and Domain Knowledge agent—the framework fulfilled the objective of ensuring that resulting microservices are not only technically decoupled but also performantly stable and semantically aligned with business logic. The research proves that context is the critical ingredient in decomposition; without integrating runtime behavior and domain documentation, automated tools often produce "distributed monoliths" rather than true microservices. Furthermore, the experimental results derived from the evaluation of Go-based applications confirm that the framework achieved the objective of providing high modularity and service independence, with metrics consistently exceeding established benchmarks. Ultimately, this research provides a sustainable roadmap for SMEs to modernize their legacy systems cost-effectively, ensuring their technical infrastructure can scale alongside their business growth.

5.2. Key Research Findings and Objective Fulfillment

Beyond the primary objectives, several critical insights were uncovered that validate the success of the proposed solution. One of the most significant findings, which directly addresses the objective of context-aware analysis, is the complementary nature of hybrid decomposition. It was observed that static analysis alone is insufficient for modern dynamic applications, as the integration of runtime profiling revealed hidden dependencies that static tools traditionally overlook. The research also successfully fulfilled the objective of creating a resource-efficient tool by highlighting the power of localized RAG; using a dedicated vector store for source code and documentation significantly reduced hallucination rates and allowed the AI to justify its decisions with low computational overhead.

Additionally, the study found that the AI-driven approach is particularly adept at identifying "God Objects" and suggesting logical splits that human developers often avoid, meeting the objective of simplifying complex architectural refactoring. This finding is further supported by the observed cost-to-performance ratio, which validated that smaller, fine-tuned models used within a RAG framework can outperform larger, general-purpose models like GPT-4 in

domain-specific tasks. These insights suggest that the research has successfully met its goal of proving that specialized, context-aware AI agents are the future of architectural modernization for resource-constrained organizations.

5.3. Summary of Contributions

The contributions of this thesis to the field of Software Engineering are multi-faceted, representing the successful completion of the research goals. Primarily, the research introduced a novel context-aware multi-agent framework that manages specialized AI agents to collaboratively solve the multi-dimensional problem of system decomposition. This was supported by a pioneering lightweight RAG architecture optimized specifically for code analysis, which utilizes localized embeddings to ensure data privacy while significantly reducing costs.

Furthermore, the study established a comprehensive evaluation framework using Modularity Quality (MQ), Service Independence Score (SIS), and Business Alignment Index (BAI), fulfilling the objective of providing a standardized methodology to measure the accuracy of AI-proposed architectural changes. Finally, by focusing on the Go (Golang) ecosystem, this research has filled a significant gap in existing literature, contributing one of the first detailed empirical studies on AI-driven decomposition for Go-based applications, which have previously been underrepresented in academic research compared to Java or C#.

5.4. Limitations and Future Work

While the results of this research demonstrate high efficacy in decomposition quality, several limitations were identified during the evaluation phase that inform the trajectory for future development. Although SMART-Split achieved a high Modularity Quality (MQ) score, its current optimization for the Go programming language restricts its broader adoption in polyglot enterprise environments. Consequently, a primary area of future work is the adaptation of the framework for Java, Python, and C#, ensuring that the cross-language structural analysis maintains the same high Service Independence Score (SIS) observed in the current study.

The evaluation also revealed that the Business Alignment Index (BAI) is sensitive to the quality of existing documentation; in instances where documentation was sparse, the AI struggled to map technical clusters to domain-specific entities. To mitigate this result-based limitation, future research will explore "Synthetic Domain Discovery," where the framework leverages its Static Analyzer to infer business intent from variable semantics and API metadata, thereby reducing the reliance on manual documentation. Furthermore, while the framework provides a blueprint for decomposition, the manual effort required to implement the physical split remains a bottleneck for SMEs. To move toward an end-to-end modernization suite, future work will involve the integration of an "Adapter Generation" agent to automatically synthesize boilerplate code and inter-service communication layers. Lastly, given the results showing increased network complexity in distributed architectures, incorporating a "Security and Compliance" agent to evaluate data flow vulnerabilities during the decomposition process would be a vital step in ensuring the architectural integrity of the

modernized system. By addressing these results-driven future directions, the SMART-Split framework can evolve into a fully automated, secure enterprise modernization tool.

REFERENCES

[1] Y.-Y. Chen, K.-H. Hsu, and A. W. Hou, “MAT: Automating Go monolithic applications transform into microservices through dependency analysis and AST,” in *2023 9th International Conference on Applied System Innovation (ICASI)*, Apr. 2023, pp. 133–135. doi: 10.1109/ICASI57738.2023.10179517.

[2] D. A. Brucker-Hahn *et al.*, “CloudCover: Enforcement of Multi-Hop Network Connections in Microservice Deployments,” in *2024 Annual Computer Security Applications Conference (ACSAC)*, Dec. 2024, pp. 1186–1202. doi: 10.1109/ACSAC63791.2024.00095.

[3] D. Taibi and K. Systä, “A Decomposition and Metric-Based Evaluation Framework for Microservices,” Aug. 22, 2019, *arXiv*: arXiv:1908.08513. doi: 10.48550/arXiv.1908.08513.

[4] A. S. Abdelfattah and T. Cerny, “The Microservice Dependency Matrix,” vol. 14183, 2023, pp. 276–288. doi: 10.1007/978-3-031-46235-1_19.

[5] K. Sellami and M. A. Saied, “Contrastive Learning-Enhanced Large Language Models for Monolith-to-Microservice Decomposition,” Feb. 07, 2025, *arXiv*: arXiv:2502.04604. doi: 10.48550/arXiv.2502.04604.

[6] V. Faria and A. R. Silva, “Code Vectorization and Sequence of Accesses Strategies for Monolith Microservices Identification,” Dec. 21, 2022, *arXiv*: arXiv:2212.11657. doi: 10.48550/arXiv.2212.11657.

[7] K. Sellami and M. A. Saied, “Contrastive Learning-Enhanced Large Language Models for Monolith-to-Microservice Decomposition,” Feb. 07, 2025, *arXiv*: arXiv:2502.04604. doi: 10.48550/arXiv.2502.04604.

[8] A. Mathai, S. Bandyopadhyay, U. Desai, and S. Tamilselvam, “Monolith to Microservices: Representing Application Software through Heterogeneous GNN,” Dec. 01, 2021, *arXiv*: arXiv:2112.01317. doi: 10.48550/arXiv.2112.01317.

[9] F. Montesi, M. Peressotti, and V. Picotti, “Sliceable Monolith: Monolith First, Microservices Later,” July 17, 2021, *arXiv*: arXiv:2103.09518. doi: 10.48550/arXiv.2103.09518.

[10] V. Nitin, S. Asthana, B. Ray, and R. Krishna, “CARGO: AI-Guided Dependency Analysis for Migrating Monolithic Applications to Microservices Architecture,” Oct. 06, 2022, *arXiv*: arXiv:2207.11784. doi: 10.48550/arXiv.2207.11784.

[11] F. Auer, V. Lenarduzzi, M. Felderer, and D. Taibi, “From monolithic systems to Microservices: An assessment framework,” *Inf. Softw. Technol.*, vol. 137, p. 106600, Sept. 2021, doi: 10.1016/j.infsof.2021.106600.

[12] A. S. Abdelfattah, K. E. Cordes, A. Medina, and T. Cerny, “Semantic Dependency in Microservice Architecture: A Framework for Definition and Detection,” Jan. 20, 2025, *arXiv*: arXiv:2501.11787. doi: 10.48550/arXiv.2501.11787.

[13] T. Matias, F. F. Correia, J. Fritzsch, J. Bogner, H. S. Ferreira, and A. Restivo, “Determining Microservice Boundaries: A Case Study Using Static and Dynamic Software Analysis,” July 12, 2020, *arXiv*: arXiv:2007.05948. doi: 10.48550/arXiv.2007.05948.

[14] 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.

[15] A. S. Abdelfattah, “Fostering Microservice Maintainability Assurance through a Comprehensive Framework,” July 23, 2024, *arXiv*: arXiv:2407.16873. doi: 10.48550/arXiv.2407.16873.

[16] D. Faustino, N. Gonçalves, M. Portela, and A. R. Silva, “Stepwise Migration of a Monolith to a Microservices Architecture: Performance and Migration Effort Evaluation,” Jan. 17, 2022, *arXiv*: arXiv:2201.07226. doi: 10.48550/arXiv.2201.07226.

[17] P. Puapongsakorn and E. Brazdeikyte, “Exploring the Integration of Artificial Intelligence in the Ideation Stage of Product Development in Swedish Startups: Challenges, Opportunities, and Tool Utilization”.

[18] Y. Wei, Y. Yu, M. Pan, and T. Zhang, “A Feature Table approach to decomposing monolithic applications into microservices,” in *12th Asia-Pacific Symposium on Internetware*, Singapore Singapore: ACM, Nov. 2020, pp. 21–30. doi: 10.1145/3457913.3457939.

[19] A. Kalia, J. Xiao, R. Krishna, S. Sinha, M. Vukovic, and D. Banerjee, “Mono2Micro: A Practical and Effective Tool for Decomposing Monolithic Java Applications to Microservices,” in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Aug. 2021, pp. 1214–1224. doi: 10.1145/3468264.3473915.

[20] “IBM Mono2Micro.” Accessed: Oct. 18, 2025. [Online]. Available: https://www.ibm.com/docs/en/mono2micro?topic=mono2micro-overview

[21] “What is RAG? - Retrieval-Augmented Generation AI Explained - AWS,” Amazon Web Services, Inc. Accessed: Oct. 18, 2025. [Online]. Available: https://aws.amazon.com/what-is/retrieval-augmented-generation/

[22] J. C. Henderson and H. Venkatraman, “Strategic alignment: Leveraging information technology for transforming organizations,” *IBM Syst. J.*, vol. 38, no. 2.3, pp. 472–484, 1999, doi: 10.1147/SJ.1999.5387096.

[23] F. H. Vera-Rivera, C. Gaona, and H. Astudillo, “Defining and measuring microservice granularity-a literature overview,” *PeerJ Comput. Sci.*, vol. 7, p. e695, 2021, doi: 10.7717/peerj-cs.695.

[24] Yoppy Mirza Maulana, Z. R. M. Azmi, and D. N. E. Phon, "Business-IT Alignment through Enterprise Architecture in a Strategic Alignment Dimension: A Review," *Register*, vol. 9, no. 1, pp. 55–67, May 2023, doi: 10.26594/register.v9i1.3084.

[25] "Retrieval Augmented Generation," Databricks. Accessed: Oct. 18, 2025. [Online]. Available: https://www.databricks.com/glossary/retrieval-augmented-generation-rag

[26] U. Desai, S. Bandyopadhyay, and S. Tamilselvam, "Graph Neural Network to Dilute Outliers for Refactoring Monolith Application," Feb. 07, 2021, *arXiv*: arXiv:2102.03827. doi: 10.48550/arXiv.2102.03827.

[27] H. Knoche, "Sustaining Runtime Performance while Incrementally Modernizing Transactional Monolithic Software towards Microservices," in *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, Delft The Netherlands: ACM, Mar. 2016, pp. 121–124. doi: 10.1145/2851553.2892039.

[28] Y. Abgaz *et al.*, "Decomposition of Monolith Applications Into Microservices Architectures: A Systematic Review," *IEEE Trans. Softw. Eng.*, vol. 49, no. 8, pp. 4213–4242, Aug. 2023, doi: 10.1109/TSE.2023.3287297.

[29] D. Taibi and K. Systä, "From Monolithic Systems to Microservices: A Decomposition Framework based on Process Mining," in *Proceedings of the 9th International Conference on Cloud Computing and Services Science*, in CLOSER 2019. Setubal, PRT: SCITEPRESS - Science and Technology Publications, Lda, May 2019, pp. 153–164. doi: 10.5220/0007755901530164.

[30] T. F. Ackerson Dan, "Domain-Driven Design for Microservices," Semaphore. Accessed: Oct. 18, 2025. [Online]. Available: https://semaphore.io/blog/domain-driven-design-microservices

[31] claytonsiemens77, "Identify microservice boundaries - Azure Architecture Center." Accessed: Oct. 18, 2025. [Online]. Available: https://learn.microsoft.com/en-us/azure/architecture/microservices/model/microservice-boundaries

[32] S. Kapferer, "A Modeling Framework for Strategic Domain-driven Design and Service Decomposition," Feb. 2020. Accessed: Oct. 18, 2025. [Online]. Available: https://www.semanticscholar.org/paper/A-Modeling-Framework-for-Strategic-Domain-driven-Kapferer/c7a474ce02e22e3a765abb4626eaedfdc587885a

[33] J. Fritzsch, J. Bogner, A. Zimmermann, and S. Wagner, "From Monolith to Microservices: A Classification of Refactoring Approaches," in *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*, vol. 11350, J.-M. Bruel, M. Mazzara, and B. Meyer, Eds., in Lecture Notes in Computer Science, vol. 11350. , Cham: Springer International Publishing, 2019, pp. 128–141. doi: 10.1007/978-3-030-06019-0_10.

[34] N. Heidloff, “Step-by-Step Instructions for Mono2Micro,” Niklas Heidloff. Accessed: Oct. 18, 2025. [Online]. Available: https://heidloff.net/article/step-by-step-instructions-mono2micro/

[35] M. Ziabakhsh, K. Rezaee, S. Eskandari, S. A. H. Tabatabaei, and M. M. Ghassemi, “Extracting Overlapping Microservices from Monolithic Code via Deep Semantic Embeddings and Graph Neural Network-Based Soft Clustering,” Aug. 10, 2025, *arXiv*: arXiv:2508.07486. doi: 10.48550/arXiv.2508.07486.

[36] R. Qiu *et al.*, “Co-Saving: Resource Aware Multi-Agent Collaboration for Software Development,” May 28, 2025, *arXiv*: arXiv:2505.21898. doi: 10.48550/arXiv.2505.21898.

[37] Z. Ke, A. Xu, Y. Ming, X.-P. Nguyen, C. Xiong, and S. Joty, “MAS-ZERO: Designing Multi-Agent Systems with Zero Supervision,” May 26, 2025, *arXiv*: arXiv:2505.14996. doi: 10.48550/arXiv.2505.14996.

[38] A. Li, Y. Xie, S. Li, F. Tsung, B. Ding, and Y. Li, “Agent-Oriented Planning in Multi-Agent Systems,” Mar. 11, 2025, *arXiv*: arXiv:2410.02189. doi: 10.48550/arXiv.2410.02189.

[39] “Build a Multi-Agent System with LangGraph and Mistral on AWS | Artificial Intelligence.” Accessed: Oct. 18, 2025. [Online]. Available: https://aws.amazon.com/blogs/machine-learning/build-a-multi-agent-system-with-langgraph-and-mistral-on-aws/

[40] “Building multi-agent systems with LangGraph and Amazon Bedrock.” Accessed: Oct. 18, 2025. [Online]. Available: https://www.cloudthat.com/resources/blog/building-multi-agent-systems-with-langgraph-and-amazon-bedrock

[41] “PC-Agent: A Hierarchical Multi-Agent Collaboration Framework for Complex Task Automation on PC.” Accessed: Oct. 18, 2025. [Online]. Available: https://arxiv.org/html/2502.14282v2

[42] J. van Dreven, “Functional Decomposition Techniques and Their Impact on Performance, Scalability and Maintainability”.

[43] V. Bushong, D. Das, A. Al Maruf, and T. Cerny, “Using Static Analysis to Address Microservice Architecture Reconstruction,” in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Melbourne, Australia: IEEE, Nov. 2021, pp. 1199–1201. doi: 10.1109/ASE51524.2021.9678749.

[44] S. Panichella, M. I. Rahman, and D. Taibi, “Structural Coupling for Microservices,” in *Proceedings of the 11th International Conference on Cloud Computing and Services Science*, 2021, pp. 280–287. doi: 10.5220/0010481902800287.

[45] D. Taibi and K. Systä, “A Decomposition and Metric-Based Evaluation Framework for Microservices,” in *Cloud Computing and Services Science*, vol. 1218, D. Ferguson, V. Méndez Muñoz, C. Pahl, and M. Helfert, Eds., in Communications in Computer and Information Science, vol. 1218. , Cham: Springer International Publishing, 2020, pp. 133–149. doi: 10.1007/978-3-030-49432-2_7.

[46] M. Gysel, L. Kölbener, W. Giersche, and O. Zimmermann, “Service Cutter: A Systematic Approach to Service Decomposition,” in *Service-Oriented and Cloud Computing*, M. Aiello, E. B. Johnsen, S. Dustdar, and I. Georgievski, Eds., Cham: Springer International Publishing, 2016, pp. 185–200. doi: 10.1007/978-3-319-44482-6_12.

[47] J. Lohnertz and A. M. Oprescu, “Steinmetz: Toward automatic decomposition of monolithic software into microservices”.

[48] S. Williams, “UK SMEs face cloud migration challenges, warns Six Degrees,” IT Brief UK. Accessed: Oct. 18, 2025. [Online]. Available: https://itbrief.co.uk/story/uk-smes-face-cloud-migration-challenges-warns-six-degrees

[49] M. S. Khan, A. W. Khan, F. Khan, M. A. Khan, and T. K. Whangbo, “Critical Challenges to Adopt DevOps Culture in Software Organizations: A Systematic Review,” *IEEE Access*, vol. 10, pp. 14339–14349, 2022, doi: 10.1109/ACCESS.2022.3145970.

[50] V. Makwana, “DevOps Scaling Practices — A Roadmap With Challenges and Strategies,” DevOps.com. Accessed: Oct. 18, 2025. [Online]. Available: https://devops.com/devops-scaling-practices-a-roadmap-with-challenges-and-strategies/

[51] N. Khan and A. Al-Yasiri, “Framework for cloud computing adoption: A road map for Smes to cloud migration,” *Int. J. Cloud Comput. Serv. Archit.*, vol. 5, no. 5/6, pp. 01–15, Dec. 2015, doi: 10.5121/ijccsa.2015.5601.

[52] Kinde, “AI-Assisted Microservices Decomposition Breaking Down Monoliths Intelligently,” Kinde Learning. Accessed: Oct. 18, 2025. [Online]. Available: https://kinde.com/learn/ai-for-software-engineering/ai-devops/ai-assisted-microservices-decomposition-breaking-down-monoliths-intelligently/

[53] D. A. d Aragona, L. Pascarella, A. Janes, V. Lenarduzzi, R. Penaloza, and D. Taibi, “On the Empirical Evidence of Microservice Logical Coupling. A Registered Report,” June 03, 2023, *arXiv*: arXiv:2306.02036. doi: 10.48550/arXiv.2306.02036.

APPENDIX

Open-source Ecommerce application(**https://github.com/golang-app/ecommerce**)

Open-source Task management system(**https://github.com/whoisaditya/golang-task-management-system**)

Open-source Content Management system (**https://github.com/emarifer/goCMS**)

Smart-split vs code extension(**https://github.com/lahiruramesh/smart-split-extension**)

Smart-split Framework (**https://github.com/lahiruramesh/smart-split**)