BACHELOR OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING

QualCoder: A Quality-Driven Multi-Agent Framework for Automated Code Generation

Zaara Zabeen Arpa
200042101
Sadnam Sakib Apurbo
200042135
Nazia Karim Khan Oishee
200042137

Department of Computer Science and Engineering
Islamic University of Technology
September, 2025

QualCoder: A Quality-Driven Multi-Agent Framework for Automated Code Generation

Zaara Zabeen Arpa
200042101
Sadnam Sakib Apurbo
200042135
Nazia Karim Khan Oishee
200042137

Department of Computer Science and Engineering
Islamic University of Technology
September, 2025

Declaration of Candidate

This is to certify that the work presented in this thesis is the outcome of the analysis and experiments carried out by **Zaara Zabeen Arpa**, **Sadnam Sakib Apurbo**, and **Nazia Karim Khan Oishee** under the supervision of **Dr. Md. Azam Hossain** , Associate Professor, Department of Computer Science and Engineering, Islamic University of Technology, Dhaka, Bangladesh. It is also declared that neither this thesis nor any part of it has been submitted anywhere else for any degree or diploma. Information derived from the published and unpublished work of others have been acknowledged in the text and a list of references is given.

Dr. Md. Azam Hossain
Associate Professor
Department of Computer Science and Engineering
Islamic University of Technology (IUT)
Date: September 28, 2025

Zaara Zabeen Arpa
Student ID: 200042101
Date: September 28, 2025

Sadnam Sakib Apurbo
Student ID: 200042135
Date: September 28, 2025

Nazia Karim Khan Oishee
Student ID: 200042137
Date: September 28, 2025

Dedicated to Our supervisor, whose guidance, patience, and mentorship have greatly shaped the journey of this thesis.

Contents

1 Introduction **1**

1.1 Introductory Information 1
1.2 Motivations and Scope 2
1.3 Problem Statement 3
1.4 Research Challenges 5
1.5 Contributions 7
1.6 Organization 8

2 Related Works **10**

2.1 Background and Evolution 10
2.1.1 Historical Development of Code Generation 10
2.1.2 Single-Agent Era 12
2.1.3 Limitations Driving Multi-Agent Approaches 15
2.1.4 A Generalized Multi-Agent Code Generation Pipeline 16
2.2 Taxonomy of Multi-Agent Code Generation Systems 17
2.2.1 Dimension 1: Agent Architecture 19
2.2.2 Dimension 2: Role Specialization 21
2.2.3 Dimension 3: Task Scope 23
2.2.4 Dimension 4: Interaction Mechanism 25
2.2.5 Dimension 5: Learning and Adaptation 27
2.3 Framework Analysis 30
2.3.1 Pipeline-Based Frameworks 30
2.3.2 Hierarchical Team-Based Frameworks 34
2.3.3 Adaptive and Evolutionary Frameworks 37
2.3.4 Specialized Domain Frameworks 39
2.3.5 Emerging Frameworks and Novel Approaches 41
2.3.6 Comprehensive Framework Comparison and Analysis 42
2.4 Performance Metrics and Benchmarking 44

2.4.1 Standard Benchmarks for Code Generation 44
2.4.2 Evaluation Metrics 47
2.4.3 Evaluation Methodologies 49
2.5 Challenges and Limitations 50
2.5.1 Coordination Overhead and Complexity 51
2.5.2 Error Propagation and Cascading Failures 52
2.5.3 Scalability to Real-World Complexity 52
2.5.4 Homogeneous LLM Limitations 53
2.5.5 Evaluation and Benchmarking Gaps 53
2.5.6 Resource Efficiency and Deployment 54

3 Proposed Methodology **55**
3.1 Overview of the Methodology 55
3.2 Chronological Breakdown of Components 56
3.2.1 High-Level Planning Agent (HLP) 56
3.2.2 Sub-Planning Agent (SP) 58
3.2.3 Coding Agent (CA) 58
3.2.4 Debugging Agent (DA) 58
3.3 Integration and Interconnections 58
3.4 Visual Representation Clarity and Completeness 59
3.5 Summary of the Methodology 60

4 Results and Discussion **61**
4.1 Datasets and Experimental Setup 61
4.2 Presenting the Results 62
4.2.1 Performance on HumanEval 62
4.2.2 Competitive Programming Performance 63
4.2.3 Problem Analysis 63
4.2.4 Heterogeneous Configuration 64
4.2.5 Ablation Studies and Analyses 64
4.3 Interpreting the Results 66
4.4 Challenges and Limitations 68
4.5 Implications and Significance of Findings 68
4.6 Decision to Split or Combine Results and Discussion 69
4.7 Conclusion of Results and Discussion 69

5 Conclusion **71**
5.1 Research Overview 71

5.2 Key Contributions 71
5.3 Limitations and Future Studies 72
5.4 Final Reflections 72

References **73**

List of Figures

2.1 Phases of Multi Agent Code Generation 15
2.2 Distribution plot visualization of frameworks across five dimensions . 29
2.3 Performance vs. token-efficiency scatter plot. 43
2.4 Bar chart of frameworks by year. 44

3.1 Overview of **QualCoder** framework. The process begins with the High-Level Planner[(i)] Agent generating multiple candidate plans. The Quality Assurance Agent[(iii)] evaluates and selects the best plan, which is then passed to the Sub Planning Agent[(ii)] for detailed breakdown. The Coding Agent[(iv)] implements the detailed plan into code, which is tested on sample cases. If the code fails, it is reviewed by the Quality Assurance Agent[(iii)] and then passed to the Debugging Agent[(v)] for error analysis. The Debugging Agent[(v)] refines the code based on feedback from the Quality Assurance Agent[(iii)]. This feedback-driven traversal improves the code and facilitates error correction. 56

4.1 CodeContest results showing QualCoder's **24.10%** improvement over the MapCoder approach on complex algorithmic tasks. The results for MapCoder [25] are taken from their paper, while the results for other methods were obtained by running each with GPT-4o. 63
4.2 Comparison of QualCoder and Direct approach across difficulty levels, showing strongest performance on moderate problems. 64
4.3 Tag-wise accuracy comparison between QualCoder and Direct approach, highlighting QualCoder's algorithmic strengths and weaknesses. . . . 65
4.4 Accuracy vs. token usage comparison between QualCoder and MapCoder across different LLMs on HumanEval dataset. 67

List of Tables

2.1 Major Frameworks and Models of the Single-Agent Era 14
2.2 Overview of the proposed five-dimensional taxonomy for multi-agent code generation systems. The table categorizes systems based on their agent architecture, role specialization, task scope, interaction mechanisms and learning and adaptation capabilities, providing a structured framework to analyze and compare the design and behavior of diverse multi-agent frameworks. 18
2.3 Classification of existing multi-agent code generation frameworks according to the proposed five-dimensional taxonomy. The table categorizes each framework based on its agent architecture, role specialization, task scope, interaction mechanism and learning and adaptation capabilities, providing a structured overview of how different systems are designed and how they relate to one another across these key dimensions. 19
2.4 Comprehensive Framework Comparison 42
2.5 Comprehensive Benchmark Characteristics 47
2.6 Major Challenges in Multi-Agent Code Generation 51

4.1 Accuracy (%) comparison across LLMs on HumanEval with accuracy gain percentage over MapCoder[25]. QualCoder shows consistent improvements, with gains inversely correlated to base model capability. . 62
4.2 QualCoder demonstrates better robustness to model size reduction on Gemma compared to MapCoder[25]. 63
4.3 Heterogeneous model configuration achieving competitive performance with resource optimization. 66
4.4 Resource consumption analysis for QualCoder across different language models on HumanEval. 66
4.5 HumanEval efficiency comparison on GPT-4o showing QualCoder's lower token consumption with higher accuracy. 66

List of Abbreviations

LLM	Large Language Model
SOP	Standard Operating Procedure
MoE	Mixture of Experts
HLP	High-Level Planner
QA	Quality Assurance
SP	Sub-Planner
CA	Coding Agent
TD	Test Designer
DA	Debugging Agent
APPS	Automated Programming Progress Standard
MBPP	Mostly Basic Programming Problems
Pass@1	Percentage of problems solved correctly in one attempt
CodeCoR	Code Collaboration and Repair
SoA	Self-Organized Agents
RNN	Reccurrent Neural Network
MetaGPT	Meta Programming for Multi-Agent Collaboration

Acknowledgement

We would like to express our deepest gratitude to our supervisor, Dr. Md. Azam Hossain, Associate Professor, Department of Computer Science and Engineering, for his unwavering support, insightful feedback, and patient guidance throughout the course of this research. His expertise and encouragement were instrumental in the successful completion of this thesis.

We are also sincerely thankful to Md. Farhan Ishmam, Lecturer, Department of Computer Science and Engineering whose valuable advice, constructive suggestions, and continuous motivation greatly enriched our work. Their belief in our abilities was a constant source of inspiration, especially during challenging periods.

We extend our heartfelt appreciation to our families, whose unconditional love, encouragement, and patience have been the foundation of our academic journey. Their support, even when faced with our long and often technical explanations, has been invaluable.

Finally, we are grateful to our friends and colleagues for their moral support, engaging discussions, and much-needed moments of respite throughout this endeavor.

To all of you, we extend our sincere thanks.

Abstract

Generating high-quality code for problem-solving remains a major challenge as it requires both linguistic understanding and algorithmic reasoning. Recent multi-agent frameworks have shown great potential for code generation through collaborative planning, implementation, and debugging, yet their performance is hindered by redundant low-quality plans, the inability to decompose high-level plans into simpler subtasks, and the lack of evaluation against given test cases. We introduce **QualCoder**, a novel multi-agent framework that enhances code quality and generation efficiency by using (i) a confidence-aware planner that filters low-quality plans, (ii) a sub-task planning agent that decomposes high-level plans into smaller, executable tasks, and (iii) a quality assurance agent that validates plans and code against test cases. Our experiments across problem-solving benchmarks reveal that QualCoder improves performance over current best approaches by roughly 2% on HumanEval and 24.10% on CodeContest, while reducing the number of API calls and token count. Further analysis reveals the efficacy of QualCoder in reducing error propagation by filtering plans with lower confidence and using test cases as preliminary evaluations.

Chapter 1

Introduction

1.1 Introductory Information

Large language models (LLMs) like GPT-4, CodeLlama and others have shown impressive automatic coding capabilities from natural-language input in recent years. These models have been very accurate on basic programming assignments and have been used extensively as part of developer tools. Still, when these models are used on more complicated, competition-type problems, like those in assessments like HumanEval, CodeContest, MBPP and APPS [5], [17], they sharply degrade. These types of problems involve multi-step reasoning, logical breakdown, test-driven tuning, and strong debugging—capabilities that single-pass LLMs struggle to deliver consistently.

For closing this gap, new research has proposed multi-agent LLM frameworks that emulate how professional programmers work collaboratively. Such systems divide the coding process among specialized agents like a planner, coder, tester and debugger [22], [25], [47]. Each semi-autonomous agent enhances overall robustness and correctness in the final product. Such frameworks like **AgentCoder**, **MapCoder**, **CodeCoR** and **AdaCoder** have shown better pass rates on competitive benchmark datasets with this divide-and-conquer paradigm.

We present QualCoder, a multi-agent code generation framework. Our key contribution is the integration of a Quality Assurance (QA) Agent at multiple stages of the coding pipeline. Unlike existing frameworks, which typically perform testing or verification only at the end, the QA Agent in QualCoder interacts continuously with planning, coding, and debugging agents. This enables early detection and correction of errors, iterative refinement of generated plans and code, and overall improvement in code quality, reducing downstream failures.

In addition, we focus on smaller, resource-efficient open-source LLMs such as Gemma and QwenCoder, rather than large, resource-intensive models, making our approach more accessible and cost-effective. We evaluated our framework on both basic programming tasks (HumanEval) and complex competition-style problems (CodeContests dataset) using multiple LLMs ranging from Gemma 3 (4B parameters) to Kimi K2 (1T parameters), demonstrating strong performance and potential improvements over existing baselines such as MapCoder [26].

Until now, all multi-agent code generation frameworks have relied on the same LLM for all agents (a homogeneous setup), making the system dependent on a single model's strengths and weaknesses. Heterogeneous agent specialization has been explored in frameworks like X-MAS [39], and our framework extends this idea by supporting both homogeneous and heterogeneous configurations. We also evaluate the performance of these heterogeneous setups using our framework to assess whether task-specific LLM allocation provides advantages over homogeneous configurations.

1.2 Motivations and Scope

The impetus for such research arises from some key observations in the current literature. For instance, although systems like AgentCoder [22] and CodeCoR [47] have been seen to achieve remarkable results with massive in-house models such as GPT-4, it has not been explored yet whether they generalize efficiently to small, open-source LLMs [67]. This shortcoming calls into question the practicability and affordability of such systems, considering that in-house models usually have immense computational requirements and limited accessibility.

Second, the dominant homogeneous agent architectures, where every agent utilizes the same LLM, also inherently curtail the promise of multi-agent systems by not tapping into the varied strengths and specialized capabilities of alternative models. Some LLMs, for example, demonstrate superior capabilities in abstract reasoning and planning activities, while others may be superior in exact syntax generation, fine-grained debugging, or efficient test-case construction. Neglecting such specialized capabilities hinders the optimization potential of multi-agent pipelines.

Of the notable approaches, **MapCoder** [25] presents a novel planning-aided paradigm called "Program-of-Thought", methodically reducing complicated issues to systematic intermediary steps. The clear planning heavily increases the accuracy of code generation, as every sub-step can be individually optimized and verified through iterative feedback cycles [25]. MapCoder's systematic breakdown process matches the pro-

fessional software development processes tightly, indicating the possible advantages of role specialization in multi-agent systems. Although MapCoder's accuracy gains through the use of high-powered in-house models have been remarkable, the question of whether MapCoder is also effective when using smaller and more specialized open-source models has not been closed and remains a fascinating question to investigate.

Further, there has been a definitive trend in the recent literature towards adaptive and cost-effective architectures. AdaCoder proposed an adaptive two-phase pipeline using conditional agent invocation and very token-efficient LLMs. According to their results, stunning savings were achieved with a 12 times lower token usage while at the same time enhancing the performance by about 27.7% in the metric of pass@1 accuracy over the prior state-of-the-art methods [67]. Related areas like multi-agent question-answering [12], multi-agent planning [6], and generative simulations [48] also provide supporting insights into the effectiveness of heterogeneous configurations and demonstrate that allocating specialized models or agents to specific roles results in significant improvements in accuracy, robustness, and system stability.

Having seen these gaps and noticed these trends, our research strives to systematically analyze the efficiency of heterogeneous LLM architectures. We want to examine role-model matching, training each agent role—planning, coding, testing, and debugging—to the best-fitting LLM based on its capabilities.Central to our framework is the integration of a Quality Assurance (QA) Agent at multiple stages of the pipeline. Unlike prior frameworks, which typically perform testing or verification only at the end, the QA Agent interacts continuously with planning, coding, and debugging agents. This enables early detection and correction of errors, refinement of generated plans and code, and overall improvement in code quality while reducing downstream failures. We then evaluate how such strategic role-model matching, combined with the QA Agent, influences key factors such as accuracy, inference cost, token usage, and error propagation.

1.3 Problem Statement

Why current multi-agent systems stall

Multi-agent LLM architectures—*MapCoder*, *AgentCoder*, *CodeCoR*, *AdaCoder* and others—have achieved spectacular accuracy improvements on functional coding benchmarks by splitting tasks into clearly defined roles like planner, coder, tester and debugger. However, a critical limitation of all existing frameworks is the absence of a

dedicated Quality Assurance (QA) Agent that continuously monitors and refines the outputs at multiple stages of the pipeline. In these systems, testing or verification is typically performed only at the end, which delays error detection and allows mistakes to propagate downstream. Another key limitation is the reliance on a *homogeneous-model assumption*: every agent in the pipeline uses the *same* underlying foundation model, typically GPT-4 or GPT-3.5. While this uniformity simplifies implementation, it introduces three major strategic limitations:

1. **Mis-matched proficiency.**
   Different roles in the pipeline demand different strengths. A planner agent needs long-context reasoning. A coder benefits from precise syntax control. A tester mainly requires speed. Forcing all roles to rely on the same model means we miss out on the specialized capabilities of modern models—like the code-focused CODELLAMA, the reasoning-optimized GEMMA, or the ultra-efficient PHI.

2. **Cost asymmetry.**
   Reusing GPT-4 for every task, even the trivial ones like generating boilerplate or compiling, incurs a high and constant cost. For instance, a single run on APPS using MapCoder can exceed **20,000 tokens**, which is prohibitively expensive for real-time or large-scale usage.

3. **Systemic failure modes.**
   When all agents are powered by the same model family, they share the same blind spots. If GPT-4 misses an edge case during planning, it's likely the coder and debugger will miss it too—leading to uncorrected errors that cascade through the system.

Opportunity: Role-Aware and QA-Enhanced Agents

A critical opportunity lies in integrating a dedicated Quality Assurance (QA) Agent that continuously monitors and refines outputs at multiple stages of the coding pipeline. Unlike current frameworks, which typically perform testing or verification only at the end, a QA Agent can detect errors early, improve plan and code quality iteratively, and prevent mistakes from propagating downstream.

Alongside QA integration, there's growing anecdotal evidence that *heterogeneous* configurations—such as using GPT-4 for planning, GPT-3.5 for coding, a 7B model for drafting, and a Python sandbox for execution—can match or even outperform their homogeneous counterparts, all while being significantly more cost-efficient. How-

ever, the field lacks a thorough, benchmark-level evaluation of this role-specific alignment strategy.

Additionally, current systems don't include *adaptive controllers* that can dynamically choose the best model for each task, taking into account the complexity, cost, and latency involved.

Research Objectives

O1. Integrate and evaluate a Quality Assurance (QA) Agent across multiple stages of the code generation pipeline. We aim to measure its impact on early error detection, code quality, and downstream failure reduction, comparing performance against baseline methods such as MapCoder and other homogeneous frameworks.

O2. Quantify the actual performance gap when replacing the single GPT-4 backbone in current leading pipelines with open-source LLMs ranging from 1B to 14B parameters. We will assess this in terms of pass@1 accuracy, hidden test robustness, and categorized errors.

O3. Design a *role-aware heterogeneous architecture* which assigns each role in the pipeline to the model or external tool best suited for that task. For example, GEMMA-9B for high-level planning, QWEN2-CODER-7B for generating code, PHI-4-MINI for bulk test generation, and a sandbox executor for validating outputs.

O3. Evaluate this proposed system across four standard benchmarks—HUMANEVAL, MBPP, APPS, and CODECONTESTS—and compare its accuracy, token cost, and runtime performance against the top-performing homogeneous baselines (including GPT-4-only and GPT-3.5-only setups from MapCoder, CodeCoR, and AdaCoder).

1.4 Research Challenges

Developing a *heterogeneous* multi-agent pipeline goes far beyond simply swapping out models. It introduces critical questions around model-role alignment, reliability, and consistent evaluation standards. Based on recent literature, we identify six key, interrelated challenges:

- **Model–role alignment.**
  Large language models vary significantly in reasoning ability, syntax precision,

and context length. For instance, MapCoder's four-stage design relies heavily on a reasoning-intensive PLANNER—replacing it with a smaller model leads to substantial performance drops, whereas the RETRIEVER can be scaled down with minimal impact [25], [67]. This suggests the need for *role-specific profiling* of models rather than relying on overall benchmarks.

- **Error propagation and containment.**
  In sequential pipelines, mistakes made early in the process can cascade. CodeCoR illustrates how a flawed prompt agent can mislead both the coder and the tester, resulting in compounded errors [47]. Some mitigation strategies—like internal simulations [25], n-best result pruning [47], and teacher–learner feedback loops [22]—have been proposed, but they haven't been stress-tested in heterogeneous setups, where diagnosis and correction may involve different model families.

- **Dynamic routing and load balancing.**
  AdaCoder uses a two-phase controller to cut token usage by 16×, invoking costly planners only for complex tasks [67]. Scaling this approach to handle *multiple* models of varying sizes (e.g., 7B, 13B, 33B) requires additional components—like task difficulty estimators, quota managers, and latency-aware heuristics—that are missing in current frameworks.

- **Evaluation fairness.**
  Cost evaluation in heterogeneous systems is complicated by system prompts, retries, external tool latency, and communication between agents. Existing studies use inconsistent accounting: MetaGPT counts only model tokens [20], while AgentVerse includes tool usage but ignores retries [6]. There's a pressing need for standardized benchmark protocols that normalize: *(i)* cost per successfully solved instance, *(ii)* wall-clock latency under API rate constraints, and *(iii)* resilience to hidden tests.

- **Token efficiency in inter-agent dialogue.**
  Detailed hand-offs—like PRDs, UML diagrams, or verbose reasoning traces—often dwarf the final code output in size. On average, MapCoder logs over 12 k tokens per task, with more than 60% spent on agent communication [25]. Figuring out how to compress or cache this chatter without losing essential alignment signals remains a complex engineering issue.

- **Tool–model interoperability.**
  Systems that integrate compilers, linters, or search engines (e.g., CodeAgent) outperform purely LLM-based approaches on complex codebases. However,

this introduces challenges around interface compatibility and handling tool-generated errors [48]. In a heterogeneous team, it's crucial that a lightweight coding model can accurately interpret diagnostics from an external tool originally optimized for GPT-4 input.

- **QA Agent response parsing and overhead.**
  Integrating a QA Agent at multiple stages introduces additional communication overhead and complexity. The QA Agent must parse outputs from planning, coding, and debugging agents, generate actionable feedback, and relay it effectively without significantly increasing latency or token usage. Designing robust parsing strategies and efficient feedback mechanisms is essential to ensure the QA Agent improves code quality without becoming a bottleneck.

Tackling these challenges is critical for realizing the full accuracy and cost-efficiency potential of mixed-model, multi-agent code generation systems.

1.5 Contributions

This thesis presents four interconnected contributions to the field of multi-agent reasoning and automatic code generation, with a particular emphasis on scalability, robustness, and efficiency under resource-constrained settings.

1. **Design of an Efficient Role-Aware Multi-Agent Framework.**
   We introduce a novel multi-agent architecture that assigns lightweight, open-source models to specialized roles, including high-level planning, sub-task decomposition, quality assurance, code generation, test design, and debugging. Unlike traditional homogeneous frameworks that rely on a single large model, our approach ensures that only the necessary parameter footprint is active at any given time, enabling efficient deployment even on limited computational resources.

2. **Introduction of a Quality Assurance Agent.**
   Building upon prior frameworks such as MapCoder [25] and AdaCoder [67], we incorporate a dedicated Quality Assurance (QA) agent tasked with evaluating both high-level and detailed plans through confidence scoring and targeted feedback. Plans that fall below a predefined confidence threshold are dynamically refined before execution, implementing a lightweight self-reflection loop inspired by CodeCoR [47]. This mechanism improves plan reliability without imposing significant computational overhead.

3. **Systematic Evaluation of Heterogeneous Architectures on Existing Frameworks.**
   To explore the broader applicability of our heterogeneous, role-specialized design, we retrofit small-model configurations into leading pipelines such as MapCoder, CodeCoR, and AdaCoder. Our evaluation systematically investigates how model heterogeneity affects key performance metrics, including pass@1 accuracy, token efficiency, and runtime on benchmarks like HUMANEVAL, MBPP, and APPS.

4. **Development of an Open-Source Toolkit for Reproducible Research.**
   We provide an open-source release of all code, prompt templates, configuration files, and evaluation scripts under an MIT license. This enables future researchers to reproduce, extend, and rigorously evaluate heterogeneous multi-agent architectures in the context of code generation tasks.

Taken together, these contributions lay a foundation for systematically exploring the design space of lightweight, role-specialized multi-agent systems and their potential advantages over monolithic large-model pipelines.

1.6 Organization

This thesis follows the canonical structure recommended by the *IUT CSE BSc Thesis Template*, with each chapter answering a specific research question and feeding the next stage of the argument.

Chapter 1 – Introduction. Establishes the context of multi-agent code generation, motivates the need for heterogeneous teams, states the research objectives (§1.3), enumerates the technical challenges, and previews our main contributions.

Chapter 2 – Related Works. Develops a thematic narrative of prior art in three concentric circles: (i) single-model code LLMs, (ii) homogeneous multi-agent frameworks such as *MapCoder* and *CodeCoR*, and (iii) early evidence for heterogeneous or tool-augmented agents. Gaps identified here directly shape the design requirements in Chapter 3.

Chapter 3 – Proposed Methodology. Presents HETEROCODER, our role-aware heterogeneous pipeline. After an architectural overview, the chapter walks chronologically through agent roles, model assignments, the adaptive routing policy, and sandbox execution, concluding with a summary of how these components jointly tackle the challenges outlined in Chapter 1.

Chapter 4 – Conclusion. Recaps how each objective has been met, distils the key findings, articulates the theoretical and practical contributions, acknowledges limitations, and charts avenues for future research (e.g., retrieval-augmented planning and live contest deployment).

Chapter 5 – Citations. Demonstrates citation management with `biblatex`; not part of the technical narrative.

References. Consolidates all cited works in IEEE style.

Appendices. Provide full prompt templates, hyper-parameter grids, additional ablation tables, and the reproducibility checklist required by the department.

Chapter 2

Related Works

2.1 Background and Evolution

2.1.1 Historical Development of Code Generation

The search for automated programming goes back to the very first days of computers, with visionary computer scientists imagining machines that could write their own programs. The journey from these early dreams to today's sophisticated multi-agent systems reflects decades of technological evolution and paradigm shifts.

This search began with foundational ideas in 1949, when Maurice Wilkes proposed a "library of subroutines" and later evolved through the formal program synthesis methods pioneered by Manna and Waldinger in the 1960s and 70s.

The 1980s and 1990s saw a shift towards more practical applications with the emergence of rule-based code generators, template systems, and CASE tools designed for automatic code generation. A subsequent neural revolution, ignited by sequence-to-sequence architectures and attention mechanisms, dramatically accelerated progress. Early neural models like DeepCoder and RobustFill showcased the significant potential of this approach.

A critical turning point was the creation of pre-trained models specifically designed for coding tasks, including: CodeBERT, GraphCodeBERT, CodeT5, PLBART. This historical progression has culminated in the current era, where the scaling up to large language models has fundamentally transformed code generation capabilities.

Early Foundations (1950s-1980s)

The concept of automatic programming emerged alongside the first computers. In 1949, Maurice Wilkes envisioned a “library of subroutines" that could be automatically composed, laying conceptual groundwork for modular programming [59]. The 1960s and 1970s saw the development of formal program synthesis methods, with pioneering work by Manna and Waldinger (1971) on deriving provably correct programs from logical specifications [42]. These approaches, even though theoretically elegant, struggled with the scale and ambiguity inherent in real-world programming tasks.

Rule-Based and Template Systems (1980s-2000s)

The 1980s and 1990s witnessed the rise of rule-based code generators and template systems. CASE tools (Computer-Aided Software Engineering) promised to revolutionize software development through visual modeling and automatic code generation. Systems like Rational Rose and Together/J could generate fundamental code from UML diagrams, though the generated code often required substantial manual refinement. Domain-specific languages (DSLs) and code generators for specific tasks (e.g., parser generators like YACC) achieved practical success within limited domains.

Statistical and Machine Learning Approaches (2000s-2015)

A paradigmatic shift occurred with the recognition of the “naturalness of software" [18], which observed that source code exhibits statistical regularities similar to natural language. This insight enabled the application of statistical language modeling techniques to code, including n-gram models, probabilistic context-free grammars and early neural approaches. Researchers began exploring code completion, bug prediction and simple code generation tasks using machine learning techniques.

Neural Revolution and Pre-trained Models (2015-2020)

The introduction of sequence-to-sequence architectures and attention mechanisms revolutionized code generation. Early neural models like DeepCoder [3] and RobustFill [9] demonstrated the potential of neural approaches for program synthesis. The emergence of pre-trained models specifically designed for code marked a turning point.These included CodeBERT [13], a bimodal pre-trained model for natural language and code that laid the groundwork for code understanding tasks. This was followed by GraphCodeBERT [15], which improved code comprehension by incorporating data flow information into its pre-training. Capabilities were further expanded

by CodeT5 [56], an encoder-decoder model supporting both code understanding and generation, and PLBART [1], which unified pre-training across multiple programming languages.

The LLM Era and Towards Multi-Agent Systems (2020-Present)

The scale-up to large language models fundamentally transformed code generation capabilities. OpenAI Codex [5] demonstrated that models trained on vast code repositories could generate functional code from natural language descriptions. Subsequent models like GPT-4, Code Llama [51], StarCoder [33] and DeepSeek-Coder [16] pushed the boundaries further. However, as these models approached their architectural limits, researchers recognized that further progress required moving beyond single-agent paradigms to collaborative multi-agent systems that could leverage specialized expertise and structured workflows.

2.1.2 Single-Agent Era

The single-agent era of code generation was revolutionized by the introduction of the Transformer architecture [55] and the advent of large-scale pre-training. This progress began with key milestones in early neural models, where sequence-to-sequence architectures first demonstrated success in code generation tasks [37], [63]. This was followed by the creation of foundational pre-trained code models such as CodeBERT [13], GraphCodeBERT [15], and CodeT5 [56], which were instrumental in establishing robust methods for both code understanding and generation. The era culminated in the development of powerful Large Language Models, including OpenAI Codex, a foundational 12B parameter model that achieved 28.8% pass@1 on HumanEval [5]; the current state-of-the-art GPT-4, which achieves 67-80% pass@1 on the same benchmark [45]; the open-source Code Llama family, with its 34B variant reaching 53.7% pass@1 [51]; StarCoder, a 15.5B parameter model supporting over 80 languages [33]; DeepSeek-Coder, which provides competitive performance with improved efficiency [16]; and AlphaCode, which demonstrated strong competitive programming capabilities [34].

The success of these models is rooted in their function as general-purpose task processors, a capability driven by the convergence of three key developments: advanced reasoning, sophisticated memory architectures, and tool manipulation [41]. The internal planning abilities of these agents evolved from simple linear methods like Chain-of-Thought[58] to more robust multi-path techniques like Tree-of-Thought[62], which allow an agent to explore different solutions and backtrack from mistakes. To man-

age context over long interactions, these agents developed memory mechanisms, with frameworks like Reflexion pioneering the use of experience to learn from past failures. Furthermore, a crucial capability is their use of external tools to overcome inherent limitations which provides real-time, context-aware code suggestions directly within IDEs [28].

However, despite these advanced capabilities, single agents face critical limitations that motivate the shift to multi-agent systems. They are highly resource-intensive to train and deploy and can generate code with significant syntactic errors, semantic flaws, security vulnerabilities, and biases inherited from their training data. More fundamentally, their core training objective is next-token prediction, a process not optimized for complex reasoning which is required for many real-world software engineering tasks. This leads to a performance ceiling that collaborative agent systems aim to overcome [11].

As shown in Table 2.1, the single-agent era produced many influential code generation frameworks, from early neural-guided methods like DeepCoder [3] to large-scale models such as AlphaCode[34] and GPT-4[45]. The table summarizes their core architecture and notable evaluation results on benchmarks like HumanEval[5] and CodeXGLUE[40].

Table 2.1: Major Frameworks and Models of the Single-Agent Era

Framework/Model	Architecture	Evaluation
DeepCoder[3]	Uses a search-based method guided by a neural model.	Demonstrated the feasibility of using neural networks to guide program synthesis.
CodeBERT[13]	A bimodal pre-trained model based on the Transformer architecture.	Achieved state-of-the-art results on the CodeXGLUE[40] . [40] benchmark.
GraphCodeBERT[15]	Enhanced pre-training by incorporating data flow graph.	Outperformed previous models on semantic understanding on CodeXGLUE[40].
CodeT5[56]	A unified encoder-decoder framework.	Achieved state-of-the-art results on CodeXGLUE[40].
OpenAI Codex[5]	A large-scale (12B parameters) GPT based model.	Achieved 28.8% pass@1 on the HumanEval[5] benchmark.
AlphaCode[34]	A complete LLM-based system.	Achieved an average ranking in simulated Codeforces competitions, comparable to human competitors.
GPT-4[45]	A large-scale, multimodal model.	Demonstrated significant improvements over academic exams, including programming tasks.
StarCoder[33]	A 15.5B parameter open-source model.	Outperformed other open models of similar size at the time.
Code Llama[51]	A family of open-source LLMs (7B, 13B, 34B).	Achieved 53% pass@1 on HumanEval[5].
DeepSeek-Coder[16]	A series of models trained on 2 trillion tokens of data.	Achieved a state-of-the-art 73.8% pass@1 on HumanEval[5].

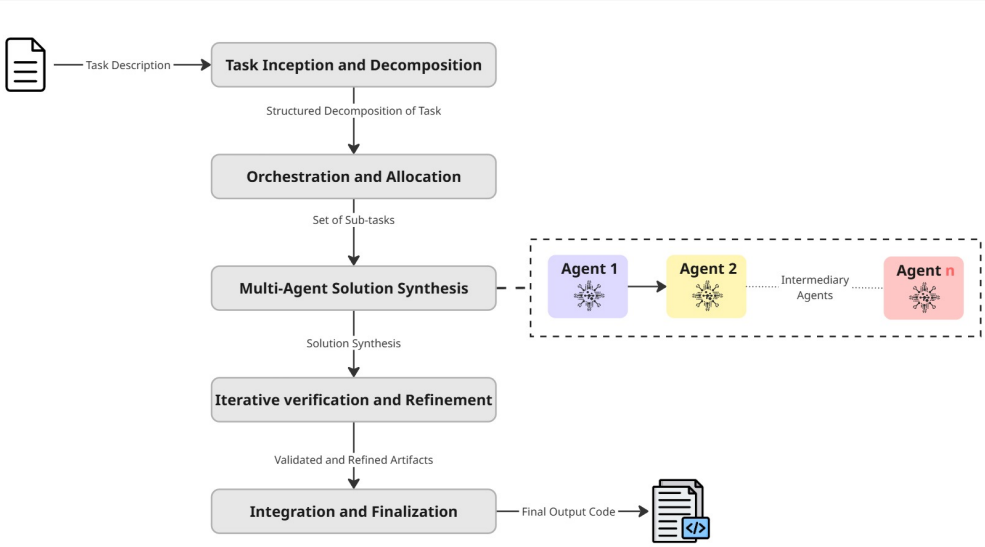

Figure 2.1: Phases of Multi Agent Code Generation

2.1.3 Limitations Driving Multi-Agent Approaches

Despite the remarkable progress in automated code generation, single-agent Large Language Models (LLMs) exhibit critical limitations that motivate the shift toward multi-agent approaches. The severity of these issues is highlighted by their performance on complex, real-world benchmarks like SWE-bench, where current systems achieve only a 1-4% success rate [29]. This significant performance gap highlights the need for more sophisticated, collaborative frameworks.These core deficits manifest in several key areas.

Single agents lack systematic methods for task decomposition, often attempting to solve complex software projects monolithically rather than breaking them into manageable sub-problems. They also cannot replicate the specialized roles common in human software development, such as architects, developers, and testers. While techniques like self-debugging exist [7], most models are constrained by limited iterative refinement, generating code in single passes with minimal feedback loops. Furthermore, they face technical constraints related to context and scalability, struggling with the long-term dependency management required for very large codebases [38]. These models can also produce syntactically correct but semantically flawed code due to hallucination, which is considered a theoretically inevitable issue [60].

Finally, single agents suffer from internal resource competition when trying to optimize multiple objectives simultaneously, leading to compromised performance in areas like code generation and test case creation.

2.1.4 A Generalized Multi-Agent Code Generation Pipeline

To overcome the limitations of single-agent systems, multi-agent frameworks introduce structured, collaborative workflows that mimic human software engineering teams. As illustrated in Figure 2.1, the pipeline for multi-agent code generation can be decomposed into five interconnected stages, each involving specialized agents coordinated through an orchestration mechanism.

The process begins with the **Task Inception and Decomposition** stage, where a high-level user prompt is translated into a structured, actionable task representation. This phase involves decomposing complex objectives into manageable subtasks that provide clarity and reduce ambiguity, thereby constraining the problem space early in the process.

Next, in the **Orchestration and Allocation** stage, these subtasks are assigned to appropriate agents based on their roles and capabilities. A central orchestrator manages the flow of information and ensures effective task distribution.

During the **Multi-Agent Solution Synthesis** stage, execution agents collaboratively generate the core solution artifacts such as source code, documentation, and configuration files. This phase reflects the parallel or sequential collaboration between multiple agents, shown in Figure 2.1 as Agent 1 through Agent n, where intermediary agents facilitate communication and iterative improvement.

The **Iterative Verification and Refinement** stage implements quality assurance loops. Generated artifacts are evaluated, tested, and debugged to ensure correctness and robustness. Feedback flows between agents (e.g., from validation to synthesis) to iteratively enhance the solution quality, capturing the generate–test–refine loop.

Finally, in the **Integration and Finalization** stage, all validated artifacts are assembled into a cohesive software system that fulfills the original user request. This step ensures component compatibility, consistent interfaces, and overall system coherence.

Figure 2.1 illustrates this end-to-end pipeline, showing how structured decomposition, coordinated agent collaboration, and continuous refinement converge to produce high-quality, executable software products in a modular and scalable fashion.

2.2 Taxonomy of Multi-Agent Code Generation Systems

To systematically understand and compare the diverse landscape of multi-agent code generation systems, we propose a comprehensive 5-dimensional taxonomy that captures the essential characteristics of these frameworks while minimizing dimensional overlap. This classification framework, presented in Table 2.2, provides a structured lens through which we can deconstruct and analyze these complex systems across their core attributes. The five dimensions we examine are: **Agent Architecture**, which defines the structural organization of agents; **Role Specialization**, which characterizes the functional differentiation among agents; **Task Scope**, which determines the scale of problems addressed; **Interaction Mechanism**, which describes communication patterns; and **Learning and Adaptation**, which assesses the system's capacity for behavioral modification.

The proposed five-dimensional taxonomy was carefully designed to provide a clear, systematic and comprehensive framework for analyzing multi-agent code generation systems. Each dimension was selected based on its significance in capturing the core characteristics that differentiate these frameworks, while avoiding redundancy across categories.

Agent Architecture's structural organization of agents fundamentally shapes how tasks are decomposed, coordinated, and executed within the system. **Role Specialization** highlights the functional differentiation among agents, which is crucial for understanding how systems leverage division of labor and expertise. **Task Scope** allows for distinguishing frameworks according to the complexity and scale of problems they can address, which is essential for assessing applicability and limitations. **Interaction Mechanism** captures the communication and coordination patterns among agents, a key determinant of system efficiency, scalability and adaptability. **Learning and Adaptation** evaluates the framework's ability to improve over time or adjust to changing environments, reflecting the system's long-term robustness and intelligence.

Building upon this taxonomy, we have systematically classified the surveyed frameworks as shown in Table 2.3. This classification reveals interesting patterns in how different architectural choices correlate with specific task scopes and interaction mechanisms. Furthermore, the distribution of frameworks across these dimensions, visualized in Figure 2.2, provides valuable insights into current research trends and potential areas for future development.

Table 2.2: Overview of the proposed five-dimensional taxonomy for multi-agent code generation systems. The table categorizes systems based on their agent architecture, role specialization, task scope, interaction mechanisms and learning and adaptation capabilities, providing a structured framework to analyze and compare the design and behavior of diverse multi-agent frameworks.

Dimension	Categories	Description
Agent Architecture	Sequential Pipeline	Agents work in a fixed, linear order, often with feedback loops.
	Hierarchical	A main agent directs specialized sub-agents in a top-down structure.
	Peer-to-Peer	Equal agents collaborate flexibly without a central authority.
	Self-Organizing	Agents create new agents or change their structure as needed.
Role Specialization	Homogeneous	All agents have similar skills and are assigned different sub-tasks.
	Functional	Agents specialize in key development tasks like coding, testing, or debugging.
	Full Specialization	Agents mirror a real-world team with diverse roles like management and design.
Task Scope	Function-Level	Generates small, specific code snippets or functions.
	Module-Level	Creates cohesive classes, packages, or related groups of functions.
	Project-Level	Develops complete, deployable software applications from requirements.
	Repository-Level	Analyzes, modifies, or enhances large, existing codebases.
Interaction Mechanism	Direct Messaging	Agents send information or instructions directly to each other.
	Shared Memory	Agents collaborate asynchronously through a common workspace or storage.
	Structured Dialogue	Agents use formatted natural language conversations to collaborate.
	Tool-Mediated	Agents interact by using external tools like version control systems.
Learning & Adaptation	Static	Agent roles and workflows are fixed and do not change.
	Reactive	The system adjusts its behavior within a session based on immediate feedback.
	Adaptive	The system can dynamically change its structure, roles, or collaboration patterns.

Table 2.3: Classification of existing multi-agent code generation frameworks according to the proposed five-dimensional taxonomy. The table categorizes each framework based on its agent architecture, role specialization, task scope, interaction mechanism and learning and adaptation capabilities, providing a structured overview of how different systems are designed and how they relate to one another across these key dimensions.

Framework	Architecture	Role Spec.	Task Scope	Interaction	Learning
AgentCoder [23]	Sequential	Functional	Function	Direct Msg	Reactive
MapCoder [26]	Sequential	Functional	Function	Direct Msg	Reactive
CodeCoR [47]	Sequential	Functional	Function	Direct Msg	Reactive
CodeSIM [57]	Sequential	Functional	Function	Direct Msg	Reactive
Self-Collaboration [10]	Sequential	Functional	Function	Direct Msg	Reactive
AutoSafeCoder [43]	Sequential	Functional	Function	Direct Msg	Reactive
MUARF [65]	Sequential	Functional	Module	Tool-Med	Reactive
Reflexion [52]	Sequential	Homogeneous	Function	Direct Msg	Reactive
AutoP2C [36]	Sequential	Functional	Repository	Tool-Med	Static
MetaGPT [19]	Hierarchical	Full Special.	Project	Shared Mem	Static
CodePori [32]	Hierarchical	Functional	Project	Structured Dialogue	Static
Intervenor [14]	Hierarchical	Functional	Function	Direct Msg	Reactive
L2MAC [35]	Hierarchical	Functional	Repository	Shared Mem	Reactive
MAGIS [53]	Hierarchical	Functional	Repository	Direct Msg	Reactive
DocAgent [61]	Hierarchical	Functional	Repository	Shared Mem	Static
ChatDev [49]	Peer-to-Peer	Full Special.	Project	Structured Dialogue	Static
AgileCoder [8]	Peer-to-Peer	Full Special.	Project	Structured Dialogue	Reactive
EvoMAC [21]	Self-Organizing	Functional	Project	Tool-Med	Adaptive
SoA [24]	Self-Organizing	Homogeneous	Repository	Shared Mem	Adaptive

2.2.1 Dimension 1: Agent Architecture

The first dimension of our taxonomy characterizes the structural organization and coordination patterns among agents in the system. As evident from Table 2.3, this architectural foundation significantly influences how agents collaborate and the complexity of tasks they can address.

Sequential Pipeline

The most straightforward approach is the sequential pipeline architecture, where agents operate in a linear sequence resembling an assembly line. This structured approach has proven particularly effective for function-level tasks, as demonstrated by numerous frameworks.

AgentCoder [23] exemplifies this pattern with its Programmer-Test Designer-Test Executor pipeline. This framework's distinct roles and linear flow highlight the efficiency of a specialized division of labor: the Programmer generates the code, the Test Designer creates verification tests, and the Test Executor runs them, providing a clear

feedback loop for iterative refinement.

The multi-agent version of **Self-Collaboration** [10] similarly employs Analyst, Coder, and Tester agents working in sequence. Here, the Analyst agent's role is to first interpret and break down the problem, a crucial step that guides the subsequent Coder and Tester agents in a structured, cooperative process. The prevalence of this architecture, particularly among function-level systems, reflects its simplicity and effectiveness for well-defined, decomposable tasks.

The prevalence of this architecture, particularly among function-level systems, reflects its simplicity and effectiveness for well-defined, decomposable tasks.

Hierarchical

Building upon the sequential foundation, hierarchical architectures introduce a central coordinator that manages subordinate agents in a top-down structure. This organizational pattern excels at complex planning and coordination tasks.

MetaGPT [19] pioneered this approach with its CEO agent managing specialized roles. The CEO agent acts as a project leader, breaking down a request into a comprehensive plan before delegating specific tasks to agents like the Product Manager, Architect, and Programmer, thereby orchestrating the entire software development life cycle.

Newer frameworks have refined concept. **CodePori** [32] employs a Manager agent to coordinate development activities. This Manager agent is responsible for dynamically decomposing a project into sub-tasks and assigning them to specialized developer agents, ensuring seamless integration of their outputs.

Intervenor [14] creates a clear instructional hierarchy with its "Code Teacher" agent guiding a "Code Learner" agent. This unique structure demonstrates a top-down control mechanism where the Teacher agent observes and intervenes to correct mistakes, providing a continuous feedback loop that guides the subordinate Learner agent toward a correct solution.

Peer-to-Peer

In contrast to hierarchical control, peer-to-peer architectures enable agents to interact as equals without fixed authority structures. This design fosters flexible, emergent collaboration that can adapt to varying problem requirements.

ChatDev [49] demonstrates this approach by simulating a software company where

agents collaborate through dialogue. This framework allows agents with different roles, such as Programmers and Testers, to engage in multi-turn, unstructured conversations, with the final solution emerging organically from their collective communication rather than from a top-down command.

AgileCoder [8] adapts this concept by having agents participate in Agile ceremonies like stand-ups and retrospectives. By mimicking these real-world team rituals, AgileCoder enables agents to provide mutual updates and collectively refine their workflow, highlighting a more structured form of peer-to-peer collaboration that promotes continuous improvement. This approach is particularly effective for tasks requiring dynamic communication and flexible problem-solving, as the system can reconfigure its interactions on the fly without needing to go through a central coordinator.

Self-Organizing Agents

The most sophisticated architectural approach is self-organizing systems, where agents can dynamically create sub-agents or alter their structural connections during execution.

SoA (Self-Organized Agents) [24] allows a "Mother" agent to spawn "Child" agents to handle emerging sub-tasks. In this model, the system's architecture is not fixed; the Mother agent can dynamically generate specialized Child agents at runtime to address novel sub-problems and can terminate them upon completion, allowing for a highly adaptive and resource-efficient structure.

EvoMAC [21] advances this concept further by using textual backpropagation to evolve the entire agent network during execution. EvoMAC's innovative approach enables the system to analyze its failures and use a process akin to backpropagation to dynamically modify agent roles and communication pathways, demonstrating a continuous self-optimization of the collaboration network itself. These systems represent a paradigm shift towards truly autonomous agent collectives that can learn, adapt, and restructure themselves in real-time.

2.2.2 Dimension 2: Role Specialization

The second dimension captures the degree and nature of functional differentiation among agents, revealing how systems balance generalization with specialization to address diverse coding challenges.

Homogeneous

At the most basic level, some systems employ homogeneous structures where all agents possess similar capabilities but are assigned different sub-tasks.

Reflexion [52] illustrates this approach with a single worker agent that reflects on its own output with assistance from a reviewer agent, both sharing similar core abilities. The system's strength lies in a single agent's ability to learn from its past failures, using an internal memory to critically evaluate its performance and iteratively improve, without needing a separate, specialized agent for the review process.

Similarly, **SoA** [24] utilizes identical agents that self-organize to tackle different aspects of a problem, demonstrating how homogeneous systems can still achieve complex task decomposition. SoA's architecture is best classified as self-organizing due to its ability to dynamically adapt its structure in real time, demonstrating a more advanced form of collaboration than a purely homogeneous system.

Functional

More commonly, frameworks adopt functional specialization, where agents are assigned distinct roles based on specific development functions. This represents the most prevalent approach in current systems, as evidenced by the distribution in Figure 2.2.

In **AgentCoder** [23], the framework assigns distinct roles: a Programmer agent is responsible solely for code generation, a Test Designer agent focuses on creating test cases and a Test Executor agent runs those tests and provides feedback. This separation ensures each task is handled by a dedicated specialist, improving the overall reliability of the code generation process.

Intervenor [14] uses a unique hierarchical teacher-student model for code generation and debugging. Its two main functional agents are - Code Teacher Agent, a high-level agent that monitors the Learner's progress and Code Learner Agent, the primary code generator that attempts to solve the problem and writes the initial code.

CodePori [32] uses a central Manager agent to oversee and coordinate development activities. Its core agents are Manager Agent and Developer Agent.

Full Specialization

At the highest level of specialization, some frameworks implement full specialization that mirrors complete, real-world software teams, including management and plan-

ning functions.

MetaGPT [19] framework implements full specialization by assigning a wide range of roles from a software company, including a CEO, Product Manager and Architect. Its function is to handle comprehensive project-level tasks by using a top-down structure to orchestrate diverse expertise and generate full project artifacts from a single prompt.

ChatDev [49] represent prime examples of this approach, incorporating roles such as CEO, Product Manager, Architect, and QA Engineer. This comprehensive role differentiation enables these systems to handle complex, project-level tasks that require diverse expertise and sophisticated coordination.

2.2.3 Dimension 3: Task Scope

The third dimension defines the scale and complexity of coding tasks that systems are designed to address, ranging from individual functions to entire repository-level operations. The distribution shown in Figure 2.2 reveals that function-level tasks dominate current research, though significant work exists across all scales.

Function-Level

At the most granular level, function-level systems focus on generating individual functions, commonly evaluated on competitive programming benchmarks. This scope represents the majority of current frameworks.

AgentCoder [23] uses a straightforward sequential pipeline with a Programmer agent, a Test Designer and a Test Executor, demonstrating a highly effective division of labor for generating and validating individual functions.

CodeSIM [57] takes this a step further by using a novel multi-agent framework that uses a human-like perception approach with simulation-driven planning and debugging to generate code.

Self-Collaboration [10] enables a team of peer agents (Analyst, Coder, Tester) to work together on function-level tasks, with the solution emerging from their collaborative dialogue and iterative refinement.

Intervenor [14] employs a hierarchical structure where a Code Teacher agent guides a Code Learner agent, providing a clear example of how specialized roles can be applied to debug and refine functions dynamically.

These systems achieve high success rates on benchmarks like HumanEval, demon-

strating the effectiveness of multi-agent approaches for well-defined programming tasks.

Module-Level

Advancing to module-level scope, systems handle the creation of cohesive classes, packages, or related groups of functions that work together.

MUARF [65] operates effectively at this level, focusing on code refactoring tasks that require maintaining consistency and relationships across related components within a module. It functions by coordinating a team of agents to modify and improve the structure of existing code while maintaining consistency and relationships across related components.

Project-Level

Project-level frameworks represent a significant leap in complexity, designed to develop complete, deployable applications from high-level requirements.

MetaGPT [19] mirrors a complete software company, assigning roles like CEO, Product Manager, and Architect to specialized agents. The system takes a high-level requirement and, through its orchestrated workflow, generates not just code, but an entire suite of project artifacts, including competitive analysis, design documents, and APIs. Its ambitious scope is defined by its ability to manage the entire software development lifecycle from a single prompt.

ChatDev [49] takes a similar ambitious approach but uses a peer-to-peer model of collaboration through dialogue to simulate a virtual software company. Agents with roles such as CEO and Programmer work together in a conversational manner to handle the full scope of a project, from designing and coding to testing and documenting.

CodePori [32] also operates at the project level, utilizing a hierarchical multi-agent system to automate large and complex software development. It employs a central Manager agent to coordinate development activities and a team of developer agents to handle the implementation.

These frameworks operate at project level scope, generating entire projects with frontend interfaces, backend logic and comprehensive documentation.

Repository-Level

At the largest scale, repository-level systems tackle the challenge of understanding and modifying existing large-scale codebases. **MAGIS** [53] addresses this challenge by resolving GitHub issues through comprehensive repository analysis. It addresses the limitations of single LLMs by using a team of specialized agents, including a Manager, a Repository Custodian, a Developer and a Quality Assurance Engineer.

DocAgent [61] generates extensive documentation by understanding the full context and architecture of existing codebases. It is particularly innovative because it doesn't just process code in isolation. Instead, it uses a topological code processing approach to build an incremental understanding of the codebase's architecture and dependencies.

2.2.4 Dimension 4: Interaction Mechanism

The fourth dimension describes how agents communicate and coordinate their activities, with different mechanisms suited to different architectural patterns and task complexities. As illustrated in Figure 2.2, direct messaging represents the most common approach, though other mechanisms offer unique advantages for specific scenarios.

Direct Messaging

Direct messaging facilitates straightforward point-to-point communication between agents, making it particularly suitable for sequential and hierarchical architectures.

The "Code Teacher" in **Intervenor** [14] exemplifies this approach by sending targeted feedback directly to the "Code Learner".

The Programmer in **AgentCoder** [23] receives specific error messages directly from the Test Executor.

Shared Memory

Shared memory systems enable asynchronous collaboration through common workspaces, allowing agents to work independently while maintaining coordination. **MetaGPT** [19] demonstrates this approach effectively, with agents sharing information through a structured repository of documents and designs. In MetaGPT, agents operate in a shared environment where they publish their outputs as standardized documents,

such as requirements, APIs, and design artifacts. Other agents can then actively observe this shared workspace and retrieve the information they need to begin their tasks, eliminating the need for direct, synchronous communication. This publish-subscribe model is crucial for enabling a complex, assembly-line-style workflow where agents work on different parts of a project concurrently. Similarly,

L2MAC [35] employs a shared file store to overcome context limitations, enabling agents to access and build upon each other's work without direct communication. In this framework, each agent is given a specific instruction and the ability to read from and write to a central file store. This external memory acts as a persistent knowledge base, allowing agents to access and build upon the work of others without needing the entire project history in their own context window.

Structured Dialogue

Structured dialogue represents a more sophisticated communication pattern where agents engage in natural language conversations following defined protocols.

This mechanism is the hallmark of **ChatDev** [49] where agents collaborate through a "chat chain" to collectively produce software. Each phase of the software development process, such as designing, coding, and testing, is broken down into a series of multi-turn dialogues between two agents. These conversations follow predefined formats and protocols to ensure productive communication and prevent issues like hallucinations, with the final solution emerging from the collaborative consensus of the chat.

AgileCoder [8], advances this concept by embedding agent communication within the structured ceremonies of Agile methodology. Instead of a free-form chat chain, agents participate in formalized dialogue patterns like daily stand-ups, sprint planning, and retrospectives. These protocols guide the conversation, ensuring that agents share critical updates on their progress, collectively plan upcoming tasks, and provide structured feedback to improve their workflow. This approach simulates the disciplined communication of a human Agile team, enabling the system to adapt and refine its process iteratively in a highly organized manner.

Tool-Mediated

Tool-mediated communication involves agents interacting through external development tools and environments, enabling more complex and realistic development workflows.

EvoMAC [21] achieves this through a novel self-evolving network structure. While agents communicate through internal dialogue, their actions are governed by an external, tool-like process called “textual backpropagation." This mechanism allows a higher-level agent to analyze the collective performance and failures, and then generate instructions in a natural language format to dynamically restructure the network itself. This process, which acts as a meta-tool, governs the agent's collaborative patterns and communication pathways to optimize for future success.

MUARF [65] coordinates refactoring tasks through specialized tools like RefactoringMiner, integrating seamlessly with existing development ecosystems. This framework's agents do not simply generate code; they interact with tools like RefactoringMiner, which are designed to analyze code and identify refactoring opportunities. Agents use these tools to perform tasks that would be difficult to accomplish with just a large language model, such as detecting specific code smells or applying complex refactoring patterns. By seamlessly integrating with existing development ecosystems through these tools, MUARF demonstrates how multi-agent systems can perform highly specialized and practical software engineering tasks.

2.2.5 Dimension 5: Learning and Adaptation

The final dimension assesses systems' capacity for behavioral modification during operation, representing a crucial factor in determining their flexibility and potential for improvement. The distribution in Figure 2.2 shows that reactive systems dominate current research, with only a few frameworks achieving true adaptability.

Static

At the foundational level, static systems maintain fixed roles and workflows throughout their operation. **MetaGPT** [19] exemplifies this approach by following a strict, predefined set of Standard Operating Procedures (SOPs). The framework's core process mirrors a waterfall model, where a CEO agent delegates tasks to a Product Manager, who then passes a requirements document to an Architect, and so on. The roles of these agents and the sequence in which they interact are fixed from the start, providing a predictable and highly structured workflow that is effective for managing the complexity of a software project.

ChatDev [49] also operates with a static, predefined workflow that is broken down into four distinct phases: designing, coding, testing, and documenting. While the communication within each phase is dynamic and dialogue-based, the overall se-

quence of these phases is fixed. Agents are recruited for each phase in a set order, and the project moves from one stage to the next only after the previous one is completed. This structured, step-by-step approach ensures that the project follows a clear path from conception to completion.

CodePori [32] likewise employs a static, predefined workflow centered around a hierarchical structure. It starts with a Manager agent that has a clear, fixed plan for task decomposition and delegation. While the Manager agent assigns tasks to other agents, the overall process and the roles of each agent are determined at the outset. This demonstrates that even with a fixed, static structure, sophisticated predefined processes can still achieve impressive results in large-scale software development.

Reactive

Reactive systems represent a significant advancement, adapting their behavior based on immediate feedback within individual sessions. Most function-level frameworks, including **AgentCoder** [23], **Reflexion** [52] and **Intervenor** [14], employ this approach.

AgentCoder [23] provides a clear example of this reactive behavior through its sequential pipeline. When the Test Executor agent runs a test and it fails, the system provides an error message and other feedback directly back to the Programmer agent. The Programmer then uses this immediate, within-session feedback to refine the code and try again, creating a direct feedback loop that drives the system toward a correct solution.

Reflexion [52] embodies a more nuanced form of reactive behavior. The system's core strength lies in its ability to reflect on its own past failures. After a failed attempt, a single worker agent reviews its own output and the resulting error messages from a previous trial. It then uses this immediate feedback to generate a revised plan or a new piece of code for the next attempt. This process of internal, self-correcting reflection is a powerful form of reactive learning.

Intervenor [14] uses a hierarchical, reactive model. When the primary Code Learner agent produces incorrect code or gets stuck, the higher-level Code Teacher agent immediately intervenes. This intervention is a direct response to a failure or a roadblock, with the Teacher providing guidance, hints, or corrections based on the immediate context.

These systems iteratively refine their code based on test results, error messages, and performance feedback, demonstrating the effectiveness of within-session learning.

Adaptive

The most advanced form of learning appears in adaptive systems, which can dynamically modify their own structure, roles, or collaboration patterns during execution. This represents the cutting edge of current research, with only a few frameworks achieving true adaptability. **EvoMAC** [21] evolves its agent network structure through a unique process called textual backpropagation. Inspired by the training of neural networks, EvoMAC obtains text-based feedback from the environment on its performance and then uses a higher-level agent to generate natural language instructions to dynamically update the network. This can include modifying agent roles or altering communication pathways, allowing the system to learn and optimize its own architecture with each iteration.

SoA [24] scales its agent hierarchy based on problem complexity, showcasing genuine structural adaptability. Its Mother agent can dynamically spawn Child agents at runtime to handle new, emerging sub-tasks. The system's architecture is not fixed but can grow and shrink in real-time, allowing it to adapt its structural complexity to meet the demands of the problem. For instance, if a coding task requires a new, specialized function, the Mother agent can create a new Child agent to handle it, demonstrating a level of structural fluidity that goes beyond static or reactive systems.

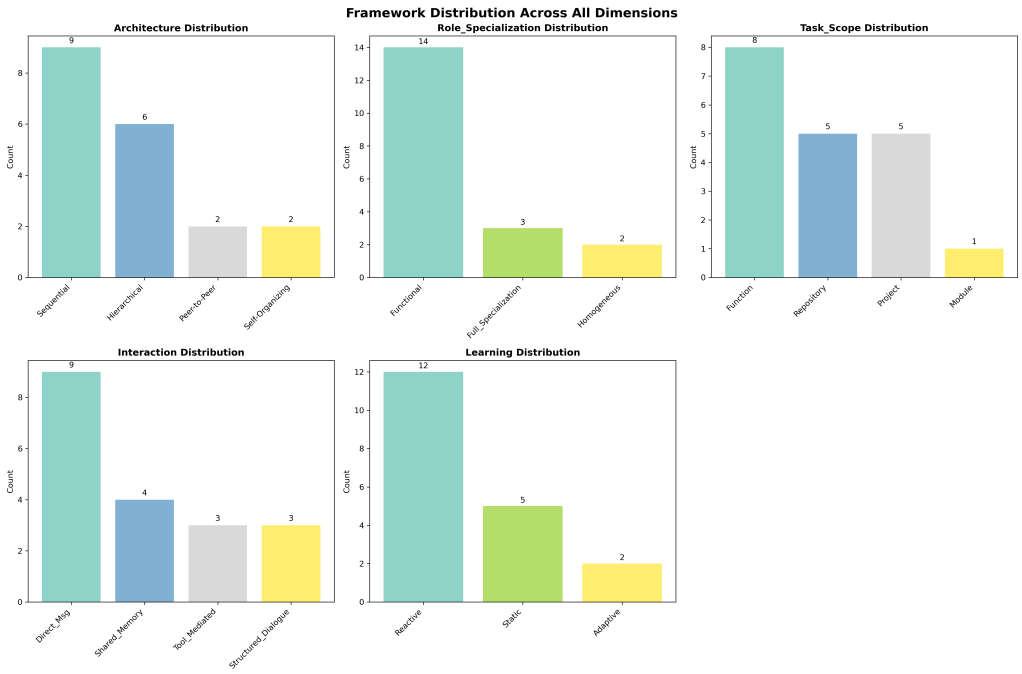

Figure 2.2: Distribution plot visualization of frameworks across five dimensions

The analysis presented in Figure 2.2 reveals several important trends in the current research landscape. The dominance of sequential architectures and functional spe-

cialization suggests that researchers are focusing on proven, effective approaches for immediate practical applications. However, the limited number of adaptive systems indicates significant opportunities for future research in developing more flexible and learning-capable multi-agent frameworks. These distribution patterns provide valuable insights for identifying both current strengths and future research directions in multi-agent code generation systems.

2.3 Framework Analysis

The landscape of multi-agent code generation frameworks has evolved rapidly, with researchers exploring diverse architectural paradigms to address the challenges of automated programming. This comprehensive analysis examines the most prominent frameworks, organizing them by their fundamental design principles and architectural innovations. As illustrated in Figure 2.4, the field has experienced significant growth, particularly in 2024, which saw the emergence of eleven major frameworks compared to six in 2023 and five in 2025, indicating the intense research interest and rapid development in this domain.

Our analysis begins with pipeline-based frameworks, which have demonstrated exceptional success in function-level code generation through their sequential, specialized agent workflows. These systems, exemplified by AgentCoder [23] and MapCoder [26], have consistently achieved state-of-the-art performance on established benchmarks. We then examine hierarchical team-based frameworks such as MetaGPT [19] and ChatDev [49], which simulate entire software development organizations to tackle complex, project-level tasks. The discussion progresses to cutting-edge adaptive and evolutionary frameworks like AdaCoder [66] and EvoMAC [21], which introduce dynamic resource management and self-improving capabilities. Finally, we explore specialized domain frameworks designed for specific applications, including security-focused systems like AutoSafeCoder [43] and process-oriented approaches like AgileCoder [8], along with emerging frameworks like MAGIS [53] that integrate directly with real-world development environments.

2.3.1 Pipeline-Based Frameworks

Pipeline-based architectures have emerged as the predominant pattern in multi-agent code generation, organizing specialized agents in sequential, assembly-line workflows where each agent performs a distinct task before passing output to the next stage. This structured approach has proven highly effective for function-level code genera-

tion, with frameworks in this category consistently achieving superior performance on benchmarks like HumanEval [5] and MBPP [2]. While sharing structural similarities, these frameworks employ diverse quality assurance strategies that distinguish their approaches and performance characteristics.

AgentCoder

Leading the charge in minimalist pipeline design, AgentCoder [23] implements a streamlined three-agent architecture focused on test-driven development with iterative refinement. The framework's efficiency stems from its deliberate simplicity and the innovative approach of generating test cases independently of the code implementation, which ensures objectivity and avoids biases that could arise from potentially flawed initial implementations.

The system employs a sequential pipeline comprising three specialized agents: a Programmer Agent that generates code using Chain-of-Thought reasoning, a Test Designer Agent responsible for independent test case generation, and a Test Executor Agent that handles script-based execution and provides feedback. This architecture creates a tight feedback loop that enables iterative code refinement based on concrete test results, leading to improved quality and correctness.

AgentCoder's key innovation lies in its Test Designer Agent, which generates comprehensive test cases without knowledge of the initial code implementation. This independence ensures that the testing phase remains objective and can effectively identify implementation flaws that might otherwise go undetected in more tightly coupled systems.

The framework has demonstrated exceptional performance, achieving 96.3% pass@1 on HumanEval and 91.8% on MBPP with GPT-4 [45], while significantly reducing token usage compared to more complex multi-agent systems. Its primary strengths include the minimal agent architecture that promotes efficiency, objective test generation that enhances reliability, and consistent improvements across various base language models. However, the system faces limitations including dependence on local execution environments and iteration budget constraints that may leave some bugs unresolved.

MapCoder

Building upon the pipeline concept with greater complexity, MapCoder [26] implements a comprehensive four-stage framework designed to explicitly model the sophis-

ticated workflows and problem-solving strategies employed by human competitive programmers. This approach represents a significant evolution from simpler pipeline designs, incorporating deeper understanding of human programming methodologies.

The framework's architecture consists of four sequential agents, each handling a distinct stage of the development process: retrieval of relevant examples from existing codebases, strategic planning of the solution design, implementation of the planned solution, and systematic debugging for error correction. This comprehensive coverage mirrors the complete programming lifecycle that experienced developers follow.

MapCoder's key innovation is its multi-plan coding approach, which involves generating multiple solution interpretations before committing to implementation. A confidence-guided mechanism evaluates these alternative plans and selects the most promising approach to pursue, reducing the likelihood of pursuing suboptimal solution paths from the outset.

The system has achieved strong performance with 93.9% pass@1 on HumanEval and 83.1% on MBPP, demonstrating effectiveness across various competitive programming benchmarks. The framework's primary strengths lie in its complete emulation of the human programming cycle and its effective use of multi-plan exploration for solution optimization. However, limitations include unexplored generalizability across different base language models beyond GPT-4, and the computational overhead introduced by the multi-plan generation strategy, which requires evaluation of multiple solution approaches and potential re-invocation of agents during debugging phases.

CodeCoR

Advancing the pipeline paradigm through redundancy and self-reflection, CodeCoR [47] introduces a framework built on principles of self-reflective multi-agent collaboration and built-in redundancy mechanisms. This approach addresses reliability concerns that can affect simpler pipeline designs by incorporating multiple validation layers throughout the development process.

The architecture employs a four-agent pipeline consisting of Prompt, Coding, Test, and Repair agents. A defining characteristic of this structure is that each agent generates multiple candidate outputs, which are subsequently evaluated and pruned based on quality metrics. This redundancy ensures that the system can recover from individual agent failures and select optimal solutions from multiple alternatives.

The framework's key innovation lies in its use of multi-candidate generation combined with self-reflection capabilities at every stage of the pipeline. This approach

is coupled with dedicated repair mechanisms that facilitate iterative improvement, creating a robust system capable of self-correction and enhancement.

CodeCoR's focus on robustness has yielded impressive results, achieving 94.5% pass@1 on the HumanEval benchmark. This performance is directly attributed to the framework's use of redundancy and self-reflection mechanisms. The system's primary strengths include robust error handling capabilities enabled by the combination of redundancy and self-reflection, and comprehensive quality assurance through multiple validation layers. However, the approach incurs computational overhead from generating multiple candidates and introduces complexity in defining and implementing criteria for pruning candidate solutions.

CodeSIM

Representing a novel evolution within pipeline-based systems, CodeSIM [57] introduces simulation-driven planning and internal debugging capabilities. The framework's core innovation lies in modeling the step-by-step input/output behavior of algorithms to validate their logic before writing the final implementation, effectively creating a mental model similar to how experienced programmers trace through code execution.

The system employs a three-agent architecture consisting of Planning, Coding, and Debugging agents, with the distinctive feature of simulation-based plan verification occurring prior to actual implementation. This pre-implementation validation phase helps identify logical flaws early in the development process, reducing the need for extensive post-implementation debugging.

CodeSIM's key innovation is its internal simulation of algorithm behavior, which serves dual purposes: plan validation and initial debugging. This process mimics the mental models that human programmers use when they mentally trace through code execution, providing a more intuitive and effective validation mechanism.

This unique methodology has led to state-of-the-art performance, with the framework achieving 95.1% pass@1 on HumanEval and 90.7% on MBPP. The primary strengths of CodeSIM include its human-like plan verification process and reduced dependency on external debugging tools during initial development phases. However, the framework faces challenges related to the complexity of implementing the simulation mechanism itself and potential limitations when applied to non-algorithmic code generation tasks that may not benefit from step-by-step simulation.

2.3.2 Hierarchical Team-Based Frameworks

Moving beyond the linear constraints of pipeline architectures, hierarchical and team-based frameworks represent a significant evolutionary step in multi-agent system design. These sophisticated systems model the complex, collaborative structures characteristic of human software development teams, typically employing top-down organizational approaches where central coordinators, planners, or manager agents orchestrate the work of multiple specialized subordinate agents. A distinguishing feature of this category is the simulation of authentic organizational roles—with agents assigned titles reflecting real-world positions such as Product Manager, Engineer, or CEO—enabling them to tackle ambitious, project-level tasks that exceed the scope of individual function generation.

MetaGPT

As a pioneering effort in hierarchical framework design, MetaGPT [19] represents a groundbreaking move beyond simple pipelines to simulate the comprehensive structure and workflow of a complete software company. The framework achieves this ambitious goal through a team of specialized LLM agents governed by carefully encoded Standard Operating Procedures (SOPs) that mirror real-world software development practices.

The framework employs a hierarchical architecture where agents embody diverse real-world roles including Product Manager, Architect, Project Manager, Engineer, and QA Engineer. Their collaborative efforts are structured and coordinated through adherence to the encoded SOPs, which provide clear guidelines for interaction and workflow management.

MetaGPT's key innovation was being the first framework to explicitly model a complete software development workflow using structured SOPs. This approach enables highly organized collaboration and facilitates the generation of comprehensive project artifacts that extend far beyond simple code generation to include documentation, specifications, and project management deliverables.

In terms of performance evaluation, MetaGPT achieved an 85.9% pass@1 score on HumanEval and 87.7% on MBPP using GPT-4. The framework's primary strengths include comprehensive role modeling that reflects real-world development teams, highly structured workflow management through SOPs, and the capability to produce coherent, well-documented solutions that encompass entire project ecosystems. However, the system encounters limitations including high token consumption re-

sulting from extensive document generation requirements and potential rigidity due to the fixed nature of its SOPs, which may not adapt well to unconventional or rapidly changing project requirements.

ChatDev

While also simulating a software company structure, ChatDev [49] distinguishes itself through a unique focus on natural language dialogue as the primary collaboration medium between role-playing agents. To maintain conversational focus and accuracy, the framework incorporates specialized communicative dehallucination mechanisms that prevent drift and ensure productive exchanges.

The architecture implements a peer-to-peer network of agents including roles such as CEO, CTO, Programmer, and Tester, who collaborate through structured "chat chains" that guide conversations through different development phases. This conversational approach creates a more natural and flexible interaction model compared to rigid procedural frameworks.

ChatDev's key innovation lies in its strong emphasis on natural language as a unifying medium for collaboration across the entire development lifecycle. The framework employs sophisticated techniques including chat chains for conversation structure and communicative dehallucination mechanisms to ensure conversational coherence and prevent the common problem of dialogue drift that can occur in multi-agent conversational systems.

Performance evaluation demonstrates ChatDev's effectiveness, achieving 56.00% completeness, 88.00% executability, and 80.21% consistency, resulting in an overall quality score of 0.3953. Notably, this performance exceeded both GPT-Engineer and MetaGPT across all evaluated metrics, demonstrating the effectiveness of the dialogue-based approach.

The framework's main strengths include its natural, dialogue-based collaboration model that mirrors human team interactions and its unified communication paradigm that simplifies agent coordination. However, the system faces limitations including potential for role confusion among agents, susceptibility to unproductive exchanges that can waste computational resources, and vulnerability to hallucination despite implemented mitigation efforts.

L2MAC

Addressing critical challenges in context management for large, complex software projects, L2MAC [35] introduces a hierarchical multi-agent framework centered on explicit memory management capabilities. This focus on memory and context preservation becomes increasingly important as projects scale beyond simple function-level tasks to encompass multi-file, interconnected software systems.

The architecture implements a multi-agent system where a central Control Unit (CU) orchestrates the overall workflow, strategically assigning each instruction to appropriate LLM agents while maintaining context and memory consistency. The agents interact through a sophisticated shared memory system composed of an instruction registry and a comprehensive file store that preserves project state across multiple development iterations.

L2MAC's key innovation is its sophisticated memory management system, specifically designed to enable agents to maintain context and continuity across complex, multi-file projects. This capability addresses a critical limitation of many existing frameworks that struggle with context preservation in large-scale development scenarios.

The framework has demonstrated strong performance, achieving 90.2% pass@1 on HumanEval and showing improved performance on specific multi-file benchmarks that better reflect real-world development challenges. L2MAC's primary strengths include excellent context management capabilities that enable work on larger projects and resulting scalability to more complex, interconnected software systems. The main limitations include memory overhead created by the comprehensive context management system and the inherent complexity of managing shared state across multiple agents, which can introduce coordination challenges.

Self-Organized Agents (SoA)

Pushing the boundaries of hierarchical systems toward ultimate scalability, Self-Organized Agents [24] provides a framework specifically designed for ultra large-scale code generation through dynamic agent organization. This approach represents a significant departure from fixed hierarchical structures toward adaptive organizational models.

The architecture implements a hierarchical system where a central planner dynamically assesses project complexity and instantiates the appropriate number and types of coding agents based on specific task requirements. This dynamic allocation enables the system to scale efficiently from simple tasks requiring minimal resources to complex projects demanding extensive agent collaboration.

SoA's core innovation is this dynamic allocation of agents based on specific task requirements, creating a truly scalable architecture suitable for projects of widely varying sizes and complexity levels. This adaptability addresses the resource inefficiency that can occur when fixed-size agent teams are applied to tasks that don't match their designed capacity.

Evaluation results show SoA achieved a Pass@1 accuracy of 71.4% on the HumanEval benchmark, representing a 5 percentage point improvement over the Reflexion baseline [52]. The framework's most significant strengths include its exceptional scalability to large projects and efficient resource allocation through dynamic agent management. However, performance can be affected by the choice of base language model, and the system has limited exposure to diverse task types, while agent communication mechanisms could benefit from further optimization.

2.3.3 Adaptive and Evolutionary Frameworks

Representing the cutting edge of multi-agent system design, adaptive and evolutionary frameworks move beyond static architectures to incorporate dynamic mechanisms for self-modification and improvement during operation. These advanced systems exhibit sophisticated forms of adaptation that distinguish them from their predecessors. Some frameworks, like AdaCoder [66], focus on resource efficiency through adaptive planning, dynamically adjusting their strategies based on immediate task complexity assessments. Others, such as EvoMAC [21], employ evolutionary principles that enable the entire agent network to self-improve by optimizing agent behaviors and collaboration patterns over time based on performance feedback. Through these mechanisms, these advanced frameworks aim to achieve not only optimal resource utilization but also higher performance ceilings through their inherent ability to learn and evolve.

AdaCoder

Leading the adaptive framework category, AdaCoder [66] introduces a highly efficient approach centered on adaptive planning mechanisms designed to optimize resource usage by intelligently switching between simpler single-agent and more complex multi-agent modes based on dynamic assessments of task difficulty and initial performance indicators.

The framework employs a two-phase approach that begins with an initial direct generation phase using a single agent. This is followed by a planning-based multi-agent

refinement phase that is activated only when needed, incorporating both rule-based debugging for common issues and LLM-based planning for more complex problems. This selective activation prevents unnecessary computational overhead for tasks that can be adequately addressed by simpler approaches.

AdaCoder's key innovation lies in its resource-efficient adaptation based on real-time problem complexity assessment and error type classification. This intelligent switching mechanism allows the system to avoid the unnecessary planning overhead that other systems might apply uniformly to simple tasks, while still providing sophisticated multi-agent capabilities when the situation demands them.

This adaptive strategy has demonstrated significant performance gains, showing a remarkable 27.69% improvement over MapCoder while using 12 times fewer tokens and achieving 16 times faster inference. These efficiency gains highlight the effectiveness of the adaptive approach in optimizing both performance and resource utilization.

The primary strengths of AdaCoder include excellent generalizability across a diverse range of language models and optimal resource utilization through intelligent adaptation. However, the system's effectiveness depends on receiving quality error feedback for accurate complexity assessment, and its rule-based debugging component may have limitations when encountering novel error types not covered by predefined rules.

EvoMAC

Pushing the boundaries of adaptive systems into evolutionary territory, EvoMAC [21] introduces groundbreaking evolutionary mechanisms that enable multi-agent collaboration networks to be genuinely self-improving. The framework achieves this through a novel form of textual backpropagation that continuously refines agent behaviors and interactions over time based on accumulated performance feedback.

The architecture features dynamic agent spawning capabilities, where the system can create new agents as needed to address emerging requirements. This is combined with evolutionary optimization of both individual agent prompts and their collective collaboration patterns, guided by comprehensive performance feedback from previous iterations.

EvoMAC's key innovation is being the first framework to incorporate evolutionary optimization of both individual agent behaviors and the overall system structure simultaneously. This dual optimization is accomplished through unique textual backpropagation mechanisms that treat agent prompts and interaction patterns as evolvable

parameters in a continuous improvement process.

Performance evaluation demonstrates that EvoMAC has outperformed static multi-agent systems on complex software requirements benchmarks, including the novel rSDE-Bench designed specifically for evaluating sophisticated development scenarios. EvoMAC's most significant strengths include its inherent self-improvement capabilities that enable continuous enhancement and its ability to jointly optimize both individual agent roles and collective interactions. However, the approach requires significant computational overhead for the continuous evolution process and involves considerable technical complexity in implementing effective textual backpropagation mechanisms.

2.3.4 Specialized Domain Frameworks

As the field of multi-agent code generation has matured, a growing category of frameworks has emerged that focuses on specific domains, processes, or quality attributes rather than pursuing general-purpose code generation. This specialization manifests in various forms, reflecting the diverse needs of real-world software development. Some frameworks, like AutoSafeCoder [43], are tailored to specific technical domains such as security, integrating specialized analysis agents directly into their workflows to enhance particular quality aspects of generated code. Others, such as AgileCoder [8], specialize in established development methodologies by structuring their entire multi-agent collaboration around proven human development practices like the Agile framework. These specialized systems demonstrate the field's evolution from generating merely functional code toward creating solutions that adhere to specific, real-world development constraints and quality standards.

AutoSafeCoder

Addressing the critical need for secure code generation, AutoSafeCoder [43] represents the first multi-agent framework specifically designed to enhance security in generated code through specialized analysis agents that incorporate both static analysis and dynamic fuzzing techniques.

The framework implements a three-agent pipeline composed of a Coding agent for initial implementation, a Static Analyzer agent for comprehensive code analysis, and a Fuzzing agent for dynamic testing. This structure operates in iterative, security-focused refinement loops that systematically identify and address potential vulnerabilities.

AutoSafeCoder's key innovation lies in being the first multi-agent system specifically designed to enhance security in generated code, achieving this through a synergistic combination of static and dynamic analysis techniques. This dual-analysis approach provides comprehensive vulnerability detection that addresses both obvious security flaws and subtle vulnerabilities that might only manifest under specific execution conditions.

Evaluation results demonstrate the framework's effectiveness, achieving a 13% reduction in vulnerabilities on the SecurityEval benchmark [54] while maintaining minimal impact on code functionality. The primary strengths of AutoSafeCoder include its specialized focus on security enhancement and its comprehensive approach to vulnerability detection through multiple analysis techniques. However, limitations include its domain-specific applicability, which may not transfer well to general-purpose development tasks, and potential trade-offs between security enhancement and full functionality preservation.

AgileCoder

Bridging the gap between established software development methodologies and multi-agent code generation, AgileCoder [8] integrates Agile methodology principles into a collaborative agent framework designed to mirror the iterative, team-based approach that has proven successful in human software development.

The framework's architecture is built around sprint-based development cycles, where different agents assume specific Scrum roles including Product Owner, Scrum Master, Developers, and QA. This role-based organization ensures that all aspects of the Agile development process are represented and properly coordinated throughout the code generation process.

AgileCoder's key innovation is being the first framework to explicitly model established Agile development practices within a multi-agent system, complete with sprint planning, daily standups, and retrospectives adapted for AI agent collaboration. This approach brings the proven benefits of Agile methodology to automated code generation.

The framework has proven effective, achieving 90.85% pass@1 on HumanEval with GPT-4 and demonstrating particularly improved performance on iterative development tasks that benefit from the Agile approach. The main strengths of AgileCoder include its natural integration with existing human-centric Agile practices, making it easier for development teams to adopt, and its inherent capabilities for iterative im-

provement through sprint-based cycles.

However, the framework faces limitations including management overhead that comes with implementing a full sprint-based process, and potential for over-engineering when applied to simple, non-iterative tasks that might not benefit from the full Agile methodology.

2.3.5 Emerging Frameworks and Novel Approaches

Beyond frameworks designed for specific domains or established architectures, an emerging category of novel approaches focuses on bridging the gap between theoretical code generation and practical, real-world software engineering applications. These cutting-edge systems are characterized by their direct integration with existing development platforms and workflows, tackling complex challenges like bug fixing and software maintenance rather than focusing solely on greenfield code creation. By operating within authentic development contexts, such as resolving actual GitHub issues, these frameworks represent a crucial step toward creating autonomous agents capable of seamlessly assisting human developers in their day-to-day activities.

MAGIS

Addressing real-world software maintenance challenges, MAGIS [53] focuses on GitHub issue resolution through sophisticated multi-agent collaboration, representing a significant shift from theoretical code generation to practical software engineering support.

The framework employs a sequential pipeline architecture with specialized agents dedicated to comprehensive issue analysis, strategic solution planning, and systematic final implementation. Each agent is optimized for its specific role in the issue resolution process, from understanding problem context to delivering working solutions.

MAGIS's key innovation lies in its direct integration with real-world development workflows, achieved through comprehensive utilization of the GitHub API for seamless interaction with existing repositories and issue tracking systems. This integration enables the framework to work with authentic development contexts rather than artificial benchmarks.

During comprehensive evaluation, MAGIS successfully resolved 13.94% of issues on the challenging SWE-bench benchmark [29], which consists of real-world GitHub issues that require understanding of complex codebases and existing development con-

texts. The framework's primary strength lies in its direct applicability to existing developer workflows and its ability to handle authentic software engineering challenges.

A notable limitation is that performance can vary significantly depending on issue complexity and repository context, with simpler issues seeing higher resolution rates than those requiring deep architectural understanding or extensive codebase modifications.

2.3.6 Comprehensive Framework Comparison and Analysis

To provide a holistic view of the multi-agent code generation landscape, Table 2.4 presents a comprehensive comparison of the frameworks discussed, highlighting their architectural characteristics and performance metrics. This comparison reveals several important insights about the current state of the field and the trade-offs inherent in different approaches.

Table 2.4: Comprehensive Framework Comparison

Framework	Agents	Architecture	HumanEval	MBPP	Year
AgentCoder [23]	3	Sequential	96.3%	91.8%	2024
MapCoder [26]	4	Sequential	93.9%	83.1%	2024
CodeCoR [47]	4	Sequential	94.5%	-	2025
CodeSIM [57]	3	Sequential	94.5%	89.7%	2025
MetaGPT [19]	5	Hierarchical	85.9%	87.7%	2023
AdaCoder [66]	4	Adaptive	96.95%	90.40%	2025
AutoSafeCoder [43]	3	Sequential	87.8%	-	2024
L2MAC [35]	Multiple	Hierarchical	90.2%	-	2024
SoA [24]	Dynamic	Hierarchical	71.4%	-	2024
AgileCoder [8]	5	Sprint-based	90.85%	80.92%	2024
MAGIS [53]	4	Sequential	-	-	2024
Parsel [64]	N/A	Hierarchical	85.0%	-	2023
Self-Collab [10]	1 (3 roles)	Sequential	90.2%	78.9%	2023
Reflexion [52]	1	Sequential	91.0%	77.1%	2023

The comparison reveals that pipeline-based frameworks, particularly AgentCoder [23] and AdaCoder [66], currently achieve the highest performance on standard benchmarks, with both exceeding 96% pass@1 on HumanEval [5]. Notably, many frameworks in our analysis did not disclose their token consumption metrics, which could be indicative of frameworks requiring substantial computational resources. This lack of transparency regarding resource usage makes it difficult to assess the true efficiency of many systems. Despite this general trend, frameworks like AdaCoder [66] have

demonstrated exceptional efficiency, requiring significantly fewer tokens compared to other frameworks while maintaining superior performance.

The performance versus token efficiency relationship is further illustrated in Figure 2.3, which presents a scatter plot analysis of frameworks for which both performance and token usage data are available. This visualization reveals that AdaCoder [66] achieves an optimal position in the performance-efficiency space, offering high accuracy with low computational cost. MapCoder [26] and CodeSIM [57] occupy middle positions, providing good performance with moderate resource requirements, while some frameworks show high performance but with correspondingly high resource consumption.

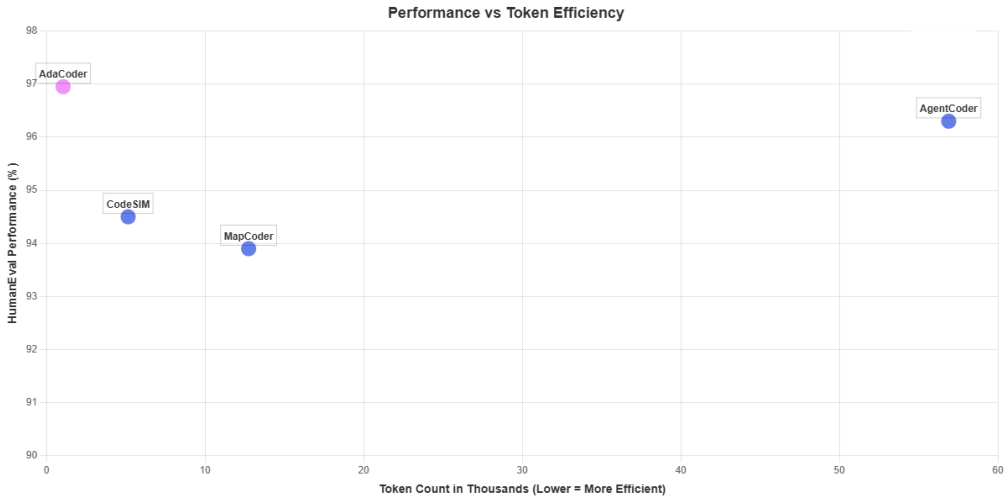

Figure 2.3: Performance vs. token-efficiency scatter plot.

The temporal evolution of framework development is depicted in Figure 2.4, which shows the number of frameworks introduced each year. The dramatic increase in 2024, with eleven new frameworks compared to six in 2023, demonstrates the accelerating pace of research in this field. This surge reflects both the growing recognition of multi-agent approaches' potential and the rapid advancement of underlying language model capabilities that make sophisticated agent collaboration increasingly feasible.

Several key insights emerge from this comprehensive analysis. First, the field shows a clear trend toward specialization, with newer frameworks targeting specific domains or development methodologies rather than pursuing general-purpose solutions. Second, there is an increasing emphasis on resource efficiency, as demonstrated by frameworks like AdaCoder [66] that achieve superior performance with reduced computational costs. Third, the integration with real-world development environments, exemplified by frameworks like MAGIS [53], represents a crucial evolution toward practical applicability beyond benchmark performance.

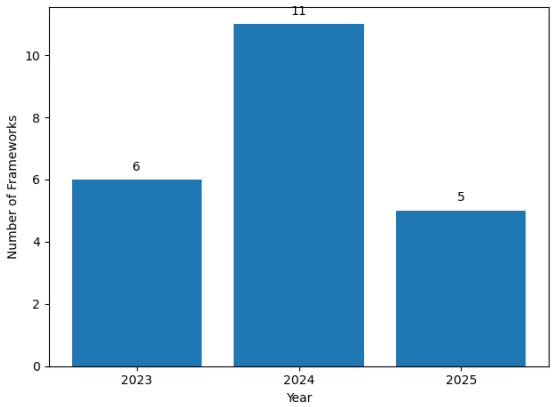

Figure 2.4: Bar chart of frameworks by year.

The diversity of architectural approaches—from simple pipelines to complex hierarchical systems to adaptive frameworks—indicates that the field has not yet converged on a single optimal design pattern. Instead, different architectures appear to excel in different contexts, suggesting that future development may focus on hybrid approaches that combine the strengths of multiple paradigms or adaptive systems that can dynamically select appropriate strategies based on task characteristics.

2.4 Performance Metrics and Benchmarking

The evaluation of multi-agent code generation systems is crucial for understanding their capabilities, comparing different approaches, and tracking progress in the field. This comprehensive evaluation landscape rests on three core pillars. First, it utilizes a diverse and evolving suite of Benchmarks, ranging from traditional function-level tests like HumanEval to complex, real-world challenges like SWE-bench. Second, assessment relies on a multi-faceted set of Metrics, which go beyond simple functional correctness (Pass@k) to include code quality (CodeBLEU), efficiency (Token Efficiency), and multi-agent specific measures like communication overhead. Finally, the field employs a hybrid Methodology, combining rigorous quantitative analysis with insightful qualitative assessments like expert human evaluation and user studies. Together, these components provide a robust framework for understanding the true capabilities of these complex systems.

2.4.1 Standard Benchmarks for Code Generation

The evaluation of multi-agent code generation systems relies on a diverse and evolving landscape of benchmarks, which are crucial for measuring capabilities, comparing different approaches, and tracking progress in the field. The assessment began with

traditional, function-level benchmarks like HumanEval and MBPP, which established standards for algorithmic reasoning and functional correctness on discrete problems. To better reflect the challenges of practical software engineering, the focus has expanded to include complex, repository-level benchmarks such as SWE-bench, which test an agent's ability to resolve real-world issues in large codebases. Complementing these are a variety of emerging and specialized benchmarks designed to evaluate nuanced capabilities like security, data science proficiency, and cross-language generalization, reflecting the growing complexity and ambition of the field.

Traditional Function-Level Benchmarks

The foundation of code generation evaluation rests upon several well-established function-level benchmarks that have become standard references in the field. HumanEval [5] represents the gold standard with its collection of 164 hand-written Python problems designed to check for functional correctness. Despite its widespread adoption across research studies, recent analysis has revealed potential data contamination issues in newer models, raising concerns about the reliability of comparative results. Building upon similar principles, MBPP [2] offers a larger dataset of 974 crowd-sourced Python problems that target basic programming concepts and are generally considered easier than HumanEval, providing a complementary perspective on fundamental coding abilities.

For more challenging algorithmic assessment, APPS [17] presents an extensive collection of 10,000 competitive programming problems categorized by difficulty levels, specifically designed to test advanced algorithmic reasoning capabilities that go beyond simple function implementation. At the apex of difficulty, CodeContests [34] features 165 problems sourced from Codeforces competitions, demanding sophisticated algorithms and optimizations that challenge even the most advanced systems.

Repository-Level and Real-World Benchmarks

The evolution toward more realistic evaluation scenarios has led to the development of repository-level benchmarks that better capture the complexity of real-world software development. SWE-bench [29] stands as a landmark achievement in this direction, featuring 2,294 real GitHub issues from popular Python repositories. The benchmark's significance is underscored by the fact that current AI systems achieve only 1-4% success rates, highlighting the substantial complexity gap between function-level tasks and real-world software engineering challenges.

To address quality concerns and ensure reliable evaluation, SWE-bench Verified presents

a carefully curated subset of 500 human-validated problems from the original SWE-bench dataset, providing researchers with high-quality evaluation tasks that have been thoroughly vetted. The benchmark family has further expanded with SWE-bench Multimodal, which introduces 617 JavaScript tasks requiring visual understanding, thereby testing multi-modal capabilities that are increasingly important in modern web development. Additionally, SWE-Lancer provides an economic perspective through its economic valuation benchmark containing $1M worth of freelance tasks, enabling practical value assessment that connects technical performance to real-world economic impact.

Emerging and Specialized Benchmarks

The field continues to evolve with specialized benchmarks targeting specific domains and capabilities. BigCodeBench [68] addresses the critical need for evaluating function-level tasks that require diverse library usage, spanning 139 libraries across 7 domains to test real-world integration capabilities that are essential for practical applications. To combat the persistent issue of data contamination, LiveCodeBench [27] provides a contamination-free benchmark with continuously updated problems sourced from recent programming competitions, ensuring that evaluation remains fair and representative.

Cross-language capabilities are assessed through MultiPL-E [4], which extends the HumanEval concept to 18+ programming languages, enabling comprehensive cross-language generalization assessment. For domain-specific evaluation, DS-1000 [31] focuses on data science with 1,000 problems testing proficiency with pandas, numpy, scipy and other scientific libraries that are fundamental to modern data analysis workflows. Security considerations are addressed by SecurityEval [54], a specialized benchmark for evaluating security aspects of generated code, while RSD-Bench focuses on requirements-to-software-design capabilities for evaluating high-level design skills. The comprehensive CodeXGLUE [40] benchmark suite rounds out the landscape by covering multiple code intelligence tasks in a unified framework.

Table 2.5 provides a comprehensive overview of the benchmark characteristics, illustrating the diversity in problem counts, language coverage, difficulty levels, and evaluation focus areas that collectively form the current evaluation ecosystem.

Table 2.5: Comprehensive Benchmark Characteristics

Benchmark	Problems	Languages	Difficulty	Type	Focus
HumanEval [5]	164	Python	Medium	Function	Basic correctness
MBPP	974	Python	Easy-Medium	Function	Basic programming
APPS [17]	10,000	Python	Hard	Algorithm	Competitive programming
CodeContests	165	Multiple	Very Hard	Algorithm	Competition-level
SWE-bench [29]	2,294	Python	Very Hard	Repository	Real issues
SWE-bench Verified [29]	500	Python	Very Hard	Repository	Curated issues
BigCodeBench [68]	1,140	Python	Hard	Function	Library usage
LiveCodeBench [27]	Dynamic	Multiple	Medium-Hard	Function	Contamination-free
MultiPL-E [4]	164×18	18 Languages	Medium	Function	Cross-language
DS-1000 [31]	1,000	Python	Medium	Function	Data science
SecurityEval [54]	150	Multiple	Medium	Function	Security
RSD-Bench	50	Multiple	Hard	Design	Requirements

2.4.2 Evaluation Metrics

Beyond the benchmarks themselves, a comprehensive evaluation of multi-agent code generation systems requires a multi-faceted set of metrics that assess not only the final output but also the quality and efficiency of the generation process. This assessment is anchored by primary metrics like Pass@k and functional correctness, which measure a system's core ability to produce correct solutions, supplemented by efficiency metrics like token usage and inference time. To move beyond simple correctness, code quality metrics such as CodeBLEU [50], cyclomatic complexity, and style adherence are used to evaluate the engineering soundness and maintainability of the generated artifact. Finally, unique to these collaborative systems, multi-agent specific metrics are employed to analyze the internal dynamics of the framework itself, including communication overhead and error propagation, providing a holistic view of a system's overall performance.

Primary Metrics

The fundamental assessment of multi-agent code generation systems begins with Pass@k, which measures the percentage of problems solved correctly within k attempts. Pass@1 represents single-attempt accuracy and provides the most stringent measure of system reliability, while Pass@10 and Pass@100 measure success with multiple sampling attempts, offering insights into the system's ability to generate correct solutions through repeated trials. Functional correctness serves as the binary assessment foundation, based on passing all provided test cases and representing the most objective measure of code quality available to researchers.

Efficiency considerations are captured through token efficiency metrics, which track the total tokens consumed including both prompt and generation phases per problem, making this measurement critical for practical cost assessment. Current lead-

ing frameworks demonstrate significant variation in this regard, ranging from 56.9K tokens per problem for AgentCoder to 138.2K tokens for MetaGPT, as illustrated in Figure 2.3. Complementing token efficiency, inference time measures the wall-clock time required for solution generation, which proves particularly important for interactive applications where user experience depends on rapid response times. Adaptive frameworks like AdaCoder have demonstrated remarkable improvements in this area, achieving up to 16 times speedup over traditional approaches.

Code Quality Metrics

Beyond basic correctness, the evaluation of generated code quality requires sophisticated metrics that capture semantic and structural properties. CodeBLEU [50] represents a significant advancement in this area, combining n-gram matching with weighted subtree matching and data-flow matching to provide comprehensive semantic similarity assessment that goes beyond surface-level text comparison. Structural complexity is measured through cyclomatic complexity, which quantifies code complexity by counting independent paths in the control flow graph, providing insights into code maintainability and testing requirements.

The maintainability index offers a composite perspective by considering multiple factors including cyclomatic complexity, lines of code, and Halstead volume to provide a holistic view of code quality. Style adherence represents another crucial dimension, measuring compliance with language-specific style guides such as PEP8 for Python or ESLint for JavaScript, ensuring that generated code meets professional standards and integrates well with existing codebases.

Multi-Agent Specific Metrics

The unique collaborative nature of multi-agent systems necessitates specialized metrics that capture the dynamics of agent interaction and coordination. Agent utilization measures the percentage of time each agent spends actively contributing versus waiting, providing insights into coordination efficiency and identifying potential bottlenecks in the collaborative process. Communication overhead, calculated as the ratio of coordination tokens to task-solving tokens, serves as a critical indicator of framework efficiency and helps identify systems that achieve effective collaboration without excessive coordination costs.

Error propagation rate tracks the frequency with which errors cascade through sequential agents, measuring the robustness of the overall system and its ability to recover from individual agent failures. Convergence speed quantifies the number of

iterations required to reach a correct solution, indicating the efficiency of the refinement process and the system's ability to iteratively improve solutions through agent collaboration.

2.4.3 Evaluation Methodologies

A comprehensive evaluation methodology for multi-agent code generation systems requires a hybrid approach, combining rigorous quantitative analysis with insightful qualitative assessment to provide a complete picture of a framework's capabilities. Quantitative evaluation forms the foundation, involving the execution of frameworks on standard benchmark suites to gather objective performance metrics, with key considerations for ensuring fair comparisons and statistical significance. Complementing this, qualitative analysis provides crucial insights that go beyond raw numbers. Through methods like expert human evaluation of code quality, deep-dive case studies, and user studies, this approach assesses practical aspects like maintainability, design elegance, and real-world usability. By integrating both data-driven metrics and human-centric assessment, the field can develop a more holistic understanding of a system's true strengths and weaknesses.

Quantitative Analysis

Standard quantitative evaluation involves running frameworks on benchmark suites and measuring performance metrics, but requires careful attention to methodological rigor to ensure meaningful results. Statistical significance must be established through multiple runs and appropriate statistical tests to ensure reliable comparisons between systems, particularly given the inherent variability in large language model outputs. Fair comparison demands strict control over experimental conditions including base LLM selection, temperature settings, and computational resources to isolate the impact of framework design choices rather than confounding factors.

Ablation studies represent a critical component of quantitative analysis, involving the systematic removal of individual agents or components to assess their specific contributions to overall system performance. This approach helps researchers understand which aspects of multi-agent collaboration provide the most significant benefits and identify potential redundancies in system design. Cross-benchmark validation extends this rigor by testing frameworks across multiple benchmarks to avoid overfitting to specific evaluation scenarios and ensure that performance gains generalize across different problem domains and difficulty levels.

Qualitative Analysis

Beyond quantitative metrics, qualitative assessment provides crucial insights through several complementary methodological approaches that capture aspects of system performance not easily quantified. Human evaluation involves expert assessment of generated code across multiple dimensions including readability, design quality, and maintainability, providing insights into code characteristics that automated metrics may miss. This human-centered evaluation proves particularly valuable for understanding how generated code would perform in real-world development scenarios where human developers must read, modify, and maintain the output.

Error analysis represents another essential qualitative methodology, involving the systematic categorization of failure modes to identify patterns of systematic weaknesses that point toward specific areas for improvement. Through detailed examination of failed test cases and incorrect solutions, researchers can understand not just what went wrong, but why certain types of problems prove particularly challenging for specific frameworks. Case studies provide complementary depth by conducting detailed analysis of framework performance on carefully selected representative problems, offering insights into the decision-making processes and collaboration patterns that lead to successful or unsuccessful outcomes.

User studies round out the qualitative evaluation approach by assessing system usability and integration with existing developer workflows, providing essential feedback on the practical deployment challenges and opportunities that determine real-world adoption. These studies help bridge the gap between laboratory performance and practical utility, ensuring that evaluation considers not just technical capabilities but also human factors that influence system effectiveness in production environments.

2.5 Challenges and Limitations

Despite the significant progress and promising results of multi-agent code generation, the field faces several substantial challenges that currently limit the capabilities, adaptability, and real-world deployment of these systems. These hurdles are not trivial; they span the entire lifecycle of the agent-based workflow, from internal system dynamics to practical application and foundational design. The key challenges include the inherent complexity and overhead of coordinating multiple agents, the risk of error propagation that can undermine performance, and the significant gap in scalability from academic benchmarks to real-world software projects. Furthermore, the field must contend with fundamental limitations in the underlying models, such as

the constraints of homogeneous LLM configurations, inefficiencies in resource utilization, and gaps in current evaluation methodologies. Overcoming these obstacles is critical for the field to mature from experimental research to robust, production-ready technology.

Table 2.6 presents key challenges faced by multi agent frameworks. It organizes problems into categories, detailing the specific issues, their negative impacts and proposed mitigation strategies for each.

Table 2.6: Major Challenges in Multi-Agent Code Generation

Challenge Category	Specific Issues	Impact	Mitigation Strategies
Coordination Complexity	Communication overhead	5-10× token increase	Efficient protocols
	Synchronization bottlenecks	Reduced parallelism	Asynchronous designs
	Conflict resolution	Inconsistent outputs	Consensus mechanisms
Error Propagation	Cascading failures	System-wide errors	Validation checkpoints
	Hallucination amplification	Compounded mistakes	Independent verification
	Debug difficulty	Complex root cause analysis	Error tracking systems
Scalability Issues	Context window limits	Cannot handle large codebases	Hierarchical processing
	Memory management	State synchronization failures	Distributed memory
	Agent proliferation	Exponential complexity growth	Dynamic agent allocation
Resource Utilization	High token consumption	2-3× cost increase	Adaptive frameworks
	Computational overhead	Slower inference	Efficient architectures
	Infrastructure complexity	Deployment challenges	Containerization
Quality Control	Response inconsistency	Unpredictable outputs	Standardized protocols
	Verification gaps	Undetected errors	Multi-stage validation
	Homogeneous limitations	Suboptimal specialization	Heterogeneous LLMs

2.5.1 Coordination Overhead and Complexity

A primary challenge in multi-agent systems is the significant overhead that arises from managing coordination among the agents, which includes complexities in communication, synchronization, and conflict resolution.

This coordination challenge manifests in several ways. These include an exponential growth in communication complexity as the number of agents increases, the emergence of coordination bottlenecks (particularly in hierarchical systems), and the generation of conflicting outputs that require resolution mechanisms. The impact is starkly evident in token consumption, where coordination can require up to nine times more tokens than the core task itself.

To address this overhead, current approaches rely on methods like structured communication protocols and centralized coordination. While these mechanisms can be effective, they often do not fully solve the underlying scalability concerns.

2.5.2 Error Propagation and Cascading Failures

Another significant challenge facing multi-agent systems is the risk of error propagation. Errors introduced by one agent can cascade through the entire system, potentially degrading overall performance to a level below even single-agent baselines.

This problem of error propagation manifests in several critical ways. These include vulnerabilities within sequential pipelines where early mistakes go unnoticed , the amplification of hallucinations as they are passed between agents , and the inherent difficulty in identifying and isolating the original source of an error. To better understand these issues, the Multi-Agent System Failure Taxonomy (MASFT) has been developed to identify such systematic failure patterns.

To mitigate this risk, current systems employ several strategies. Methods such as multi-candidate generation, the use of independent verification agents, and checkpoint-based recovery mechanisms all help to reduce the impact of errors, but they do not eliminate the risk entirely.

2.5.3 Scalability to Real-World Complexity

Current systems demonstrate impressive performance on function-level benchmarks; however, they often fall short when applied to repository-level and real-world software engineering tasks. These more complex scenarios demand a deeper understanding of software context, structure that current models are not yet fully equipped to handle.

Recent performance results, such as the 1–4% success rates observed on the SWE-bench benchmark, highlight the substantial gap between academic benchmarks and practical, real-world applications. These findings indicate that while models can solve isolated problems effectively, they struggle significantly when required to understand and modify integrated, large-scale codebases.

Several factors contribute to this difficulty. Limitations include the restricted context window of large language models when processing extensive codebases, challenges in handling long-term dependencies across files, insufficient grasp of common architectural and design patterns, and a lack of consistency when generating or modifying code across multiple interrelated files. These issues collectively hinder the models' ability to perform robust, end-to-end software engineering tasks.

2.5.4 Homogeneous LLM Limitations

Most current frameworks are built around a homogeneous configuration of large language models, which prevents them from fully specializing on the diverse strengths of specialized models. This uniform approach limits the adaptability and effectiveness of such systems across a wide range of tasks.

A compelling demonstration of the advantages of a heterogeneous setup is seen in the X-MAS framework, which achieved a 47% improvement in performance by integrating multiple LLMs with different capabilities. This result underscores the untapped potential in designing systems that strategically combine models based on their individual strengths.

The limitations of relying solely on a single type of model are significant. They include shared weaknesses across all agents, missed opportunities to optimize cost versus performance, and an inability to assign specific tasks—such as mathematical reasoning—to models best suited for them. Additionally, this approach leads to resource inefficiency, as large and computationally expensive models are often used even for simple tasks that could be handled by lighter-weight alternatives.

2.5.5 Evaluation and Benchmarking Gaps

Current evaluation methodologies often fall short in capturing the full complexity of multi-agent interactions and that of real-world software development. As a result, they may provide an incomplete picture of a system's effectiveness in practical, collaborative coding environments.

A major concern is the heavy dependence on function-level benchmarks, which, while useful for assessing isolated capabilities, do not reflect the challenges of long-term software engineering tasks. Moreover, current evaluations rarely consider the quality of collaboration between agents, overlooking how well they coordinate, share context, and make joint decisions.

These shortcomings are compounded by the lack of robust assessment criteria for long-term code maintainability and evolution. Additionally, existing benchmarks are not explicitly designed to test multi-agent capabilities, leaving a critical gap in evaluating systems intended to operate in collaborative, complex development scenarios.

2.5.6 Resource Efficiency and Deployment

Many multi-agent systems demand substantial computational resources, which poses significant obstacles to their real-world deployment and scalability. These systems, while promising in capability, often come with practical trade-offs that restricts their broader adoption.

One major consideration is the high token consumption, which can be five to ten times greater than that of single-agent approaches. This not only increases processing time but also amplifies cost, especially in large-scale or time-sensitive applications. Sequential processing among agents further contributes to increased latency, making the systems less responsive and efficient.

In addition to computational overhead, the infrastructure required to coordinate multiple agents adds additional complexity. Managing communication, state sharing and synchronization between agents necessitates robust and often custom-built orchestration mechanisms. These factors collectively raise the cost and introduce technical hindrance for deploying multi-agent systems in production environments.

Chapter 3

Proposed Methodology

This chapter introduces our suggested multi-agent modular architecture, designed to enable efficient, reliable, and scalable competitive programming code development. With regards to the limitations of previous pipelines, such as MapCoder [25], CodeCoR [47], and AdaCoder [67], we introduce a hybrid pipeline that combines thorough planning, fine-grained sub-task definition, adaptable coding methods, automatic testing procedures, and iterative debugging methods, particularly for smaller, open-source models.

The approach used in this research integrates a mixture-of-experts (MoE) model, dynamic orchestration of agents, and strict quality control mechanisms to improve code correctness while optimizing resource allocation.

3.1 Overview of the Methodology

The input to our system is a problem statement expressed in natural language. Through sequential and feedback-driven multi-agent interactions, we generate functionally correct code that passes both public and generated test cases.

Basic elements of our strategic framework include:

- **High-Level Planner Agent (HLP)** – develops multiple strategic alternatives for the presented problem.
- **Quality Assurance Agent (QA)** – identifies and suggests the most practical plan(s) with the use of a confidence scoring method.
- **Sub-Planning Agent (SP)** – translates the selected overall plan(s) into detailed coding directives.

- **Coding Agent (CA)** – generates code directly according to the sub-plans.
- **Debugging Agent (DA)** – corrects faulty code blocks up to a particular number of attempts.

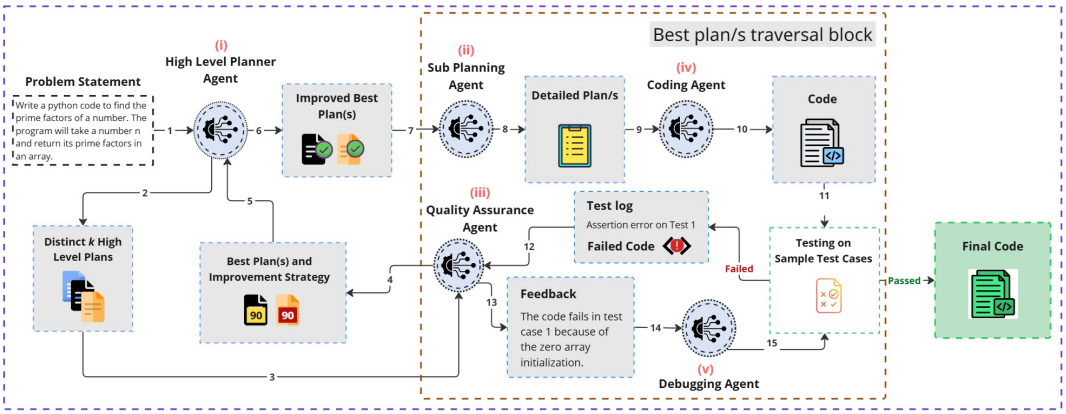

Figure 3.1: Overview of **QualCoder** framework. The process begins with the High-Level Planner[(i)] Agent generating multiple candidate plans. The Quality Assurance Agent[(iii)] evaluates and selects the best plan, which is then passed to the Sub Planning Agent[(ii)] for detailed breakdown. The Coding Agent[(iv)] implements the detailed plan into code, which is tested on sample cases. If the code fails, it is reviewed by the Quality Assurance Agent[(iii)] and then passed to the Debugging Agent[(v)] for error analysis. The Debugging Agent[(v)] refines the code based on feedback from the Quality Assurance Agent[(iii)]. This feedback-driven traversal improves the code and facilitates error correction.

This system is modular, allowing heterogeneity in model selection across agents. At a given time, there is only a single agent involved:

$$\theta_{\text{active}} = \max_{i=1,\ldots,N} ||\theta_i||$$

where the set of parameters for the i–th agent model is represented as θ_i, ensuring the total number of active parameters is kept within the individual model's maximal constraints. This ensures a lower level of GPU memory demand while operating the framework. This setup helps fortify its resource utilization efficiency.

3.2 Chronological Breakdown of Components

3.2.1 High-Level Planning Agent (HLP)

Role: The HLP receives the problem statement $\mathcal{P}$ and generates $k = 3$ distinct high-level strategies $\{\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3\}$ using controlled prompting techniques to ensure diversity (inspired by MapCoder [25]).

Why Multiple Plans? Previous research, including MapCoder, have indicated that LLMs often produce a range of solution methods with varying accuracy and performance. Formulating different approaches ahead of time increases the chances of finding a strong pathway.

Input: Problem statement $\mathcal{P}$

Output: k overall strategies $\mathcal{S}_i$

Quality Assurance Agent (QA)

The **Function:** evaluates each plan $\mathcal{S}_i$ using a scoring rubric that includes

- Completeness of solution
- Its soundness and practicability.
- Adhering to set problem constraints
- Expected computational efficiency

Each plan is assigned a confidence score $c_i \in [0, 100]$.

Selection Criteria:

$$\mathcal{S}_{\text{best}} = \arg\max_i c_i$$

If multiple plans have the highest confidence levels, then all such top-ranked plans are kept for further analysis. Unlike previous research that strives to execute all generated plans, as for instance with MapCoder [25], our approach purposefully limits this selection process. Empirical observation showed that in cases where Plan A receives a confidence score x and Plan B receives a lower score $y < x$, Plan B always fails in later assessments.

Therefore, for efficient utilization of computing resources to avoid wasted execution, only the highest-ranking plans are executed. Additionally, if the best confidence level c_{best} drops below a specified acceptance level of 70%, the Quality Assurance Agent provides accurate feedback to the High-Level Planner Agent. Thereafter, feedback comes into action to start a regeneration loop where the planner adjusts its strategies. The self-enhancement framework is guided by the ideas of remediation and self-reflection as presented in CodeCoR [47].

3.2.2 Sub-Planning Agent (SP)

Function: Converts the selected $\mathcal{S}_{\text{best}}$ into detailed, step-by-step coding instructions (sub-plans).

Why Sub-Planning? AdaCoder showed that LLMs benefit from breaking large instructions into manageable, error-resistant sub-tasks, especially with smaller models [67].

Input: High-level plan(s)

Output: Detailed plan(s) $\mathcal{D}$

Each instruction is atomic, natural-language, and ready for direct translation into code.

3.2.3 Coding Agent (CA)

Functionality: The Coding Agent receives detailed sub-task plans and generates implementation code.

3.2.4 Debugging Agent (DA)

Purpose: Upon failure on $\mathcal{T}_{\text{eval}}$, the DA receives the error logs and fixes the code iteratively.

Methodology: Maximum of $k = 3$ debugging attempts

Every iteration involves:

$$\mathcal{C}_{i+1} = \text{DA\_fix}(\mathcal{C}_i, \text{ErrorLog}_i)$$

If after fixing, $\mathcal{C}$ passes all tests, it is finalized and outputted.

3.3 Integration and Interconnections

The components of our framework are highly interdependent, yielding a pipeline that is structured yet flexible.

1. The **High-Level Planner Agent** initiates the exploratory process by proposing different strategic options.

2. The **Quality Assurance Agent** makes a prompt evaluation of these plans, thus ensuring only the most viable plan(s) proceed downstream.
3. The chosen plans are designed by the **Sub-Planning Agent** so that the programming instructions are comprehensive as well as context-specific, reducing the burden on the Coding Agent.
4. The **Coding Agent** then follows these successive directions strictly in order to produce the initial executable code.
5. Upon failures, the **Debugging Agent** performs trace-directed repair cycles to enable accurate and systematic changes to the code.

Key integration principles:

1. **Feedback Loops:** Between QA and HLP (for weak plans), and between CA, DA, and EC (for failed code).
2. **Budget Control:** Debugging is bounded at 3 attempts. Test generation is validated to prevent test-set pollution.
3. **Model Heterogeneity:** Different LLMs optimized for specific agent roles — e.g., larger model for planning, smaller model for QA, coder, and debugger.

The agent modular architecture provides scalability in the system with the ability to integrate improvements directly into the individual subcomponents without retraining or reconfiguration of the entire pipeline.

3.4 Visual Representation Clarity and Completeness

To enable understanding of the overall system, a diagrammatic representation with annotations is provided:

- **Overall Architectural Diagram** (Figure 3.1): Illustrates the flow of data, interactions between agents, and feedback mechanisms integral to the multi-agent system.
- One example of a competitive programming problem.
- The particular directives given to all of these agents, including the High-Level Planner, QA Agent, Sub-Planner, Coding Agent, Test Designer, and Debugging Agent, while working to solve this problem.

The complete diagram outlines the execution process within the framework, describing the role of each agent, the architecture for communication among agents, and decision mechanisms at each step.

All illustrations and visual aids are carefully arranged in a way that:

- The labels are concise and readily understandable.
- These data and visualizations are consistent with the protocols set down by the methodology.
- Every occurrence or demonstration is independent and can be understood separately with a minimum of dependence upon background information.

It helps provide insight to those with little knowledge of multi-agent large language model systems.

3.5 Summary of the Methodology

Here we propose a modular approach that consists of a number of agents who efficiently produce working programming code. Our methodology involves the steps of coding, testing, evaluation, planning, and debugging within a tightly interlinked, feedback-enriched workstream.

Main characteristics:

- **Strategic planning and evaluation** reduces cases of hallucination and incoherence.
- **Sub-planning decomposition** adapts the problem to the capabilities of smaller LLMs.
- **Mixture-of-Experts model selection** improves specialization without inflating active memory footprints.
- **Test augmentation and validation** fortifies evaluation against hidden errors.
- **Constrained debugging iterations** manage costs while improving successful code execution rates.

Overall, our methodological approach is designed to maximize open-source model utilization for comprehensive language processing, reduce inference cost, and maintain high standards of robustness and accuracy—moving multi-agent code generation closer to real-world application.

Chapter 4

Results and Discussion

QualCoder is evaluated on the HumanEval[44] and CodeContest[34] benchmarks using diverse language models from Gemma3 4B to GPT-4o, which allows for the analysis of both homogeneous and heterogeneous multi-agent configurations.

4.1 Datasets and Experimental Setup

QualCoder has been evaluated on two benchmarks of varying complexity. **HumanEval** [44] contains 164 basic Python problems with prompts and test cases focusing on core algorithmic skills. **CodeContest** [34] includes 165 real-world competitive programming tasks requiring advanced reasoning and implementation across difficulty levels.

To evaluate the proposed framework, direct comparisons between QualCoder and three primary baselines have been conducted across language models: the **Direct approach**, where code is generated in a single step from the prompt, **Chain of Thought Prompting (CoT)** [58] and **MapCoder**[25], a prominent homogeneous multi-agent framework that employs the same LLM across planning, coding, and debugging agents. To demonstrate the generalizability of the approach has been evaluated across multiple open-source language models of varying sizes, including Gemma3-4B, Gemma3-12B, Qwen2.5-Coder, Kimi-K2. The purpose of this diverse model selection is to analyze both the resource-efficiency of QualCoder and its effectiveness across different model capabilities and architectures. For the heterogeneous QualCoder variant, specialized models have been assigned for each agent's role: Qwen2.5 Instruct 7B for coding and debugging tasks, Gemma 3 4B for quality assurance, and Llama 3.2 3B for planning and sub-planning operations. This configuration allows to leverage the distinct strengths of different models within the proposed multi-agent framework.

Table 4.1: Accuracy (%) comparison across LLMs on HumanEval with accuracy gain percentage over MapCoder[25]. QualCoder shows consistent improvements, with gains inversely correlated to base model capability.

Approach	LLMs				
	Gemma3 4B	**Gemma3 12B**	**Kimi K2**	**Qwen 2.5**	**GPT-4o**
Direct	68.90	85.06	94.67	87.80	73.80
CoT	62.80	75.00	92.07	82.92	90.85
MapCoder	73.78	85.06	96.67	91.46	92.07
QualCoder	**78.26** (↑ 6.07%)	**88.96** (↑ 4.58%)	**98.00** (↑ 1.37%)	**95.73** (↑ 4.66%)	**93.90** (↑ 2.00%)

All experiments use OpenRouter[46] APIs with the same providers , quantization and temperature value set to 0 for consistent comparison across approaches. CodeContest evaluation employs xCodeEval [30] for solution validation.
The parameter k is set to 3, meaning three distinct plans are generated for each problem. For the debugging process, a maximum of t=5 attempts is imposed. The primary evaluation metric is Pass@1, where a problem is considered correctly solved if the first generated solution passes all provided test cases.

4.2 Presenting the Results

This section presents the experimental outcomes of QualCoder on the HumanEval [44] and CodeContest [34] benchmarks. The evaluations cover a wide range of language models, from smaller models such as Gemma3 4B to larger ones like GPT-4o, enabling analysis across both homogeneous and heterogeneous multi-agent configurations.

4.2.1 Performance on HumanEval

Table 4.1 reports the performance comparison of QualCoder against the Direct and MapCoder baselines across different LLMs. Results show that QualCoder consistently surpasses both baselines. The performance gains are most notable on smaller models, such as Gemma3 4B (+6.07%), while diminishing slightly with larger models such as Kimi K2 (+1.37%). This trend suggests that QualCoder's structured reasoning particularly benefits resource-constrained models.

Further analysis on robustness to model size reduction is provided in Table 4.2. When scaling down from Gemma 12B to 4B, QualCoder exhibits a smaller performance degradation (12.03%) compared to MapCoder (13.26%), indicating stronger resilience under resource-constrained conditions.

Table 4.2: QualCoder demonstrates better robustness to model size reduction on Gemma compared to MapCoder[25].

Approach	Accuracy (%)		Performance Drop	
	12B	4B	Absolute	Relative
MapCoder	85.06	73.78	11.28	13.26%
QualCoder	88.96	78.26	**10.70**	**12.03%**

4.2.2 Competitive Programming Performance

On the CodeContest benchmark, QualCoder achieves **35.37%** accuracy with GPT-4o, as shown in Figure 4.1. This corresponds to a substantial **264.64%** relative improvement over the Direct approach and a **24.10%** improvement compared to MapCoder (28.5%).

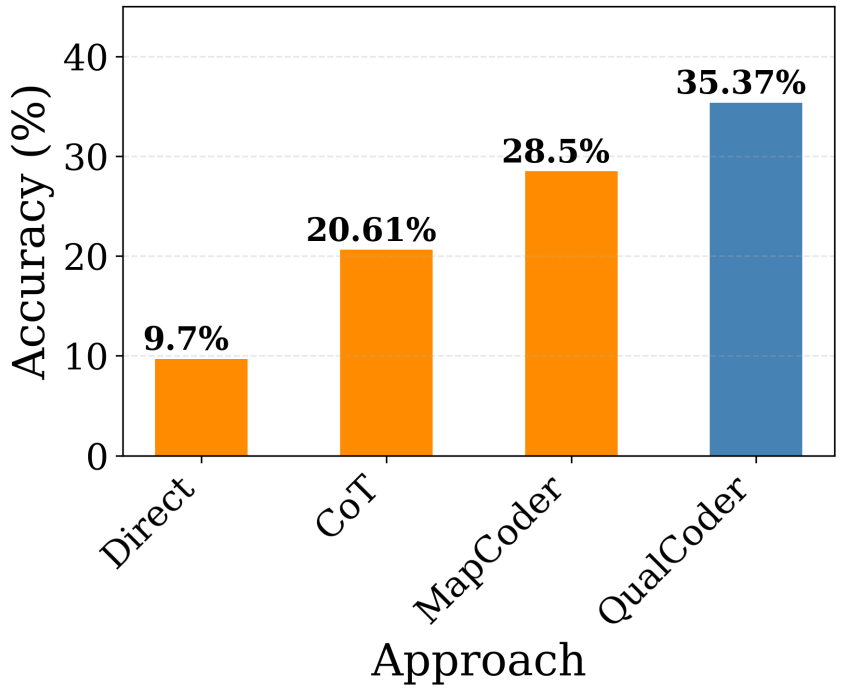

Figure 4.1: CodeContest results showing QualCoder's **24.10%** improvement over the MapCoder approach on complex algorithmic tasks. The results for MapCoder [25] are taken from their paper, while the results for other methods were obtained by running each with GPT-4o.

4.2.3 Problem Analysis

QualCoder's performance breakdown across problem difficulty levels is illustrated in Figure 4.2. Results show that QualCoder performs best on moderately difficult problems (levels 7–10), while accuracy declines for the most challenging tasks.

Figure 4.3 further analyzes performance by problem tags. QualCoder demonstrates notable strengths in **geometry (66.7%)**, **implementation (57.9%)**, **string problems (50.0%)**, and **divide-and-conquer (50.0%)**, but struggles with specialized areas such as FFT (0.0%), combinatorics (5.3%), and graph algorithms. The Direct approach shows overall lower accuracy but follows similar performance trends.

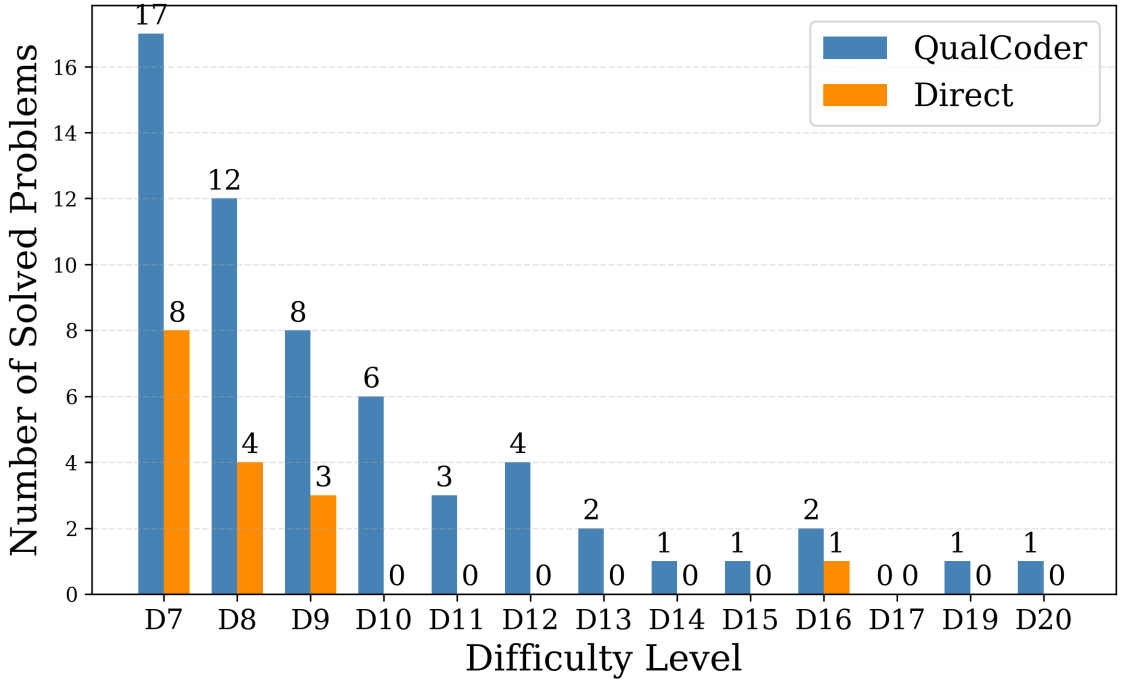

Figure 4.2: Comparison of QualCoder and Direct approach across difficulty levels, showing strongest performance on moderate problems.

4.2.4 Heterogeneous Configuration

The evaluation of heterogeneous multi-agent configurations is reported in Table 4.3. A resource-efficient setup combining smaller models (within 7B parameters) achieves a strong **64.63%** accuracy. Notably, the combination of LLaMA 3.2 (3B, 8.53% alone) with Qwen Coder 2.5 (7B, 90.85% alone) demonstrates that integrating weaker and stronger models can yield practical performance gains.

4.2.5 Ablation Studies and Analyses

To better understand the computational trade-offs of QualCoder's multi-agent architecture, we conducted a series of ablation studies focusing on resource utilization and efficiency. These experiments highlight how model size and coordination overhead influence overall performance.

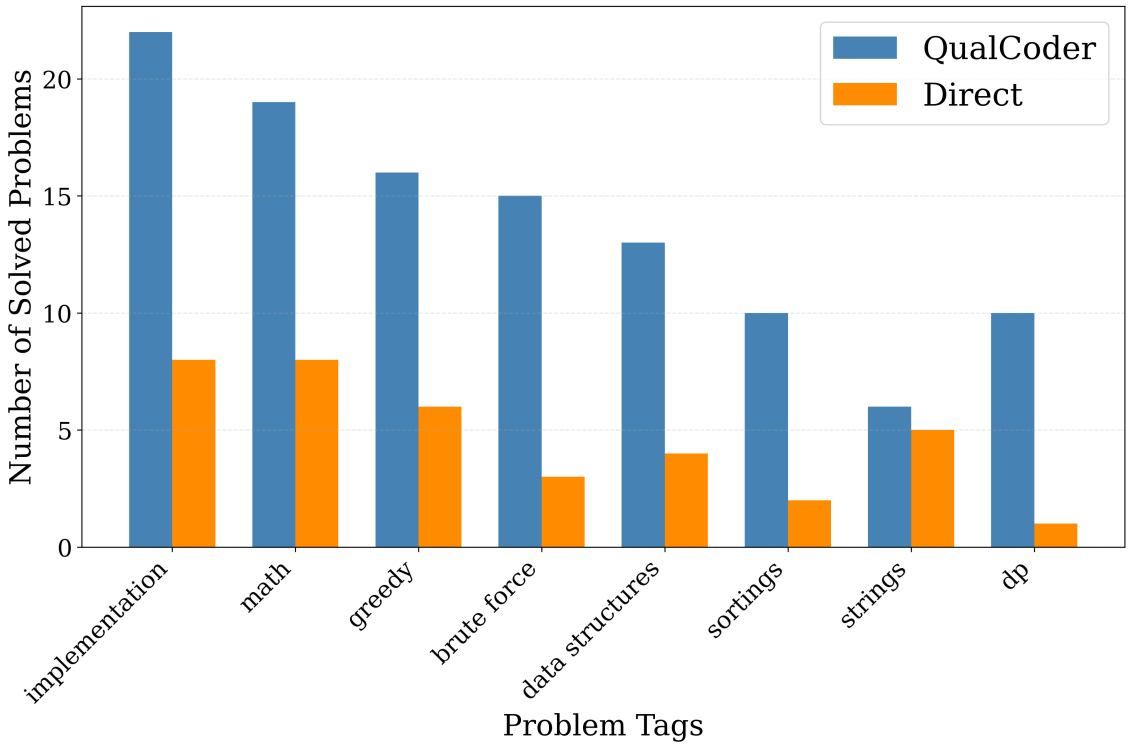

Figure 4.3: Tag-wise accuracy comparison between QualCoder and Direct approach, highlighting QualCoder's algorithmic strengths and weaknesses.

Table 4.4 reports the average input tokens, output tokens, and API calls required by QualCoder on the HumanEval benchmark [44]. While QualCoder generally requires more tokens and calls compared to direct prompting approaches, this additional cost translates into notable gains in accuracy. An interesting trend is that smaller models such as Gemma3 4B demand substantially more tokens (8,142 on average) compared to larger models like GPT-4o (3,178 tokens). This suggests that resource-constrained models need more intensive agent interaction and coordination to achieve competitive performance.

We further compared QualCoder against MapCoder [25] across the HumanEval [44] and CodeContest [34] benchmarks. As shown in Figure 4.4, QualCoder achieves higher accuracy with reduced token usage, particularly on more challenging datasets. On the CodeContest benchmark, QualCoder requires 17.5% fewer tokens while achieving a 2.0% accuracy gain, underscoring the benefits of structured quality assurance mechanisms in complex reasoning scenarios.

We also conducted ablation studies to isolate the contribution of the QA Agent in QualCoder's pipeline. Specifically, we compared the performance of the system with and without the QA Agent across different model configurations and benchmarks. The results consistently demonstrate that incorporating the QA Agent leads to im-

Table 4.3: Heterogeneous model configuration achieving competitive performance with resource optimization.

Agent	Model	Parameters	Accuracy
HL Planner ST Planner	LLaMA 3.2	3B	**64.63%**
Coder Debugger	Qwen 2.5	7B	
QA Agent	Gemma 3	4B	

Table 4.4: Resource consumption analysis for QualCoder across different language models on HumanEval.

Model	Avg Input Tokens	Avg Output Tokens	Avg API Calls
Gemma3 4B	8,142	4,695	9.81
Gemma3 12B	6,229	3,487	8.12
Qwen 2.5	5,614	3,526	8.17
GPT-4o	3,178	1,655	7.27
Kimi K2	2,853	2,292	6.95

Table 4.5: HumanEval efficiency comparison on GPT-4o showing QualCoder's lower token consumption with higher accuracy.

Approach	Accuracy (%)	Total Tokens (K)
MapCoder	92.07	56.79
QualCoder	93.90	48.32
Efficiency Gain	**2.00%↑**	**17.50%↑**

proved accuracy, confirming its effectiveness in early error detection, iterative code refinement, and overall enhancement of pipeline robustness. This highlights the critical role of the QA Agent in maintaining high-quality outputs, particularly in complex coding tasks.

4.3 Interpreting the Results

The results indicate that QualCoder delivers consistent improvements over existing baselines across multiple benchmarks and configurations. Several key insights can be drawn:

First, the superior gains on smaller models suggest that QualCoder's structured rea-

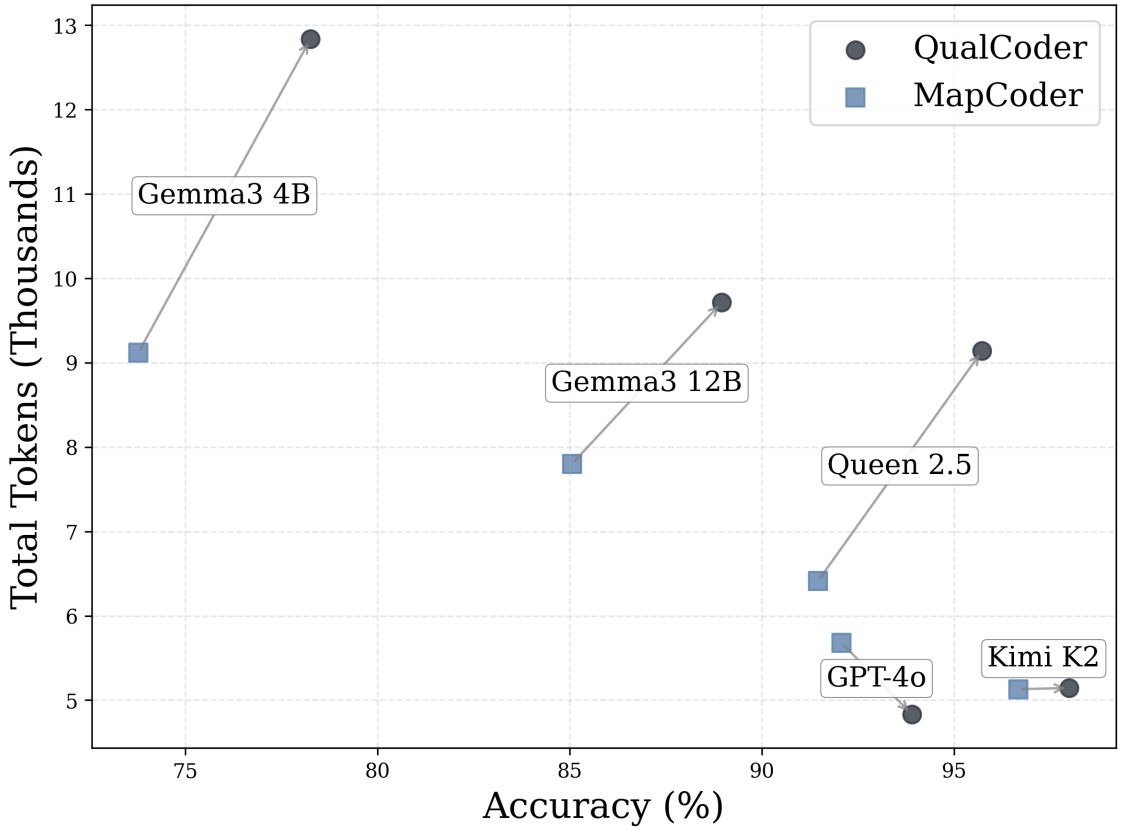

Figure 4.4: Accuracy vs. token usage comparison between QualCoder and MapCoder across different LLMs on HumanEval dataset.

soning provides greater benefits when computational capacity is limited. This aligns with prior findings that multi-agent reasoning frameworks are particularly effective for resource-constrained LLMs.

Second, the robustness analysis demonstrates that QualCoder maintains higher relative performance when scaling down model size, highlighting its practicality for efficiency-driven deployments.

Third, the competitive programming experiments reveal that QualCoder significantly outperforms Direct and MapCoder on the CodeContest benchmark, particularly in structured and moderately complex tasks. While performance drops on highly specialized categories such as FFT and combinatorics, the consistent advantage in geometry, implementation, and divide-and-conquer tasks shows that QualCoder is well-suited for decomposable algorithmic problems.

Finally, the heterogeneous configuration analysis underscores the trade-offs of combining weaker and stronger models. While integrating weaker agents reduces the upper-bound performance of stronger ones, the results demonstrate that careful model-role alignment can achieve strong accuracy at a fraction of the computational cost.

This finding is particularly relevant for real-world applications where resources are constrained but robust performance is required.

In summary, QualCoder proves effective across diverse LLMs, problem categories, and deployment configurations, validating its design as a versatile and resource-aware framework for code generation.

4.4 Challenges and Limitations

The research encountered several overarching challenges that shaped both the design and evaluation of QualCoder. A key difficulty lies in balancing computational cost with accuracy improvements, as more capable models such as GPT-4o and Kimi K2 achieve high performance with modest overhead, whereas smaller models require disproportionately larger token usage and coordination steps to obtain meaningful gains. This trade-off highlights scalability concerns when deploying multi-agent systems in resource-constrained or cost-sensitive environments. Another major challenge is ensuring robustness in structured output generation: while larger models adhere reasonably well to XML or JSON schemas, smaller models frequently struggle, necessitating additional parsing mechanisms or simplified output formats. Beyond model-specific issues, the complexity of coordinating multiple agents introduces synchronization and error-propagation risks, where failures in one agent's reasoning can cascade across the pipeline. These challenges underscore the inherent tension between accuracy, efficiency, and reliability in multi-agent code generation frameworks, pointing to the need for adaptive strategies that optimize coordination depth based on model capacity and task complexity.

4.5 Implications and Significance of Findings

The findings of this research carry important implications for both the academic study of large language models and their practical deployment in real-world systems. By demonstrating that QualCoder consistently improves accuracy across diverse LLMs on HumanEval and CodeContest, the study establishes the value of multi-agent orchestration as a mechanism for enhancing reliability in code generation tasks. This contributes to the growing body of evidence that quality assurance and structured collaboration between agents can compensate for the weaknesses of smaller models, thus lowering the barrier to adopting advanced AI solutions in resource-constrained environments. In practical terms, these results suggest that industry applications such

as software development tools, automated testing frameworks, and educational platforms could benefit from incorporating orchestration-based systems to achieve more consistent outputs while optimizing costs. Furthermore, the efficiency gains observed in ablation studies highlight the potential of adaptive coordination strategies, which can dynamically balance token usage and accuracy depending on the model's capacity and task complexity. Taken together, these contributions extend the current understanding of multi-agent systems and position QualCoder as a promising foundation for future research and applications in reliable, efficient AI-assisted programming.

4.6 Decision to Split or Combine Results and Discussion

In structuring this chapter, a decision was made to separate the presentation of results from their interpretation. This choice reflects the multi-phase nature of the research, which involved evaluation across multiple benchmarks, heterogeneous language models, and detailed ablation studies. Presenting the raw data first ensured clarity and allowed readers to follow the empirical evidence systematically, while a separate discussion section enabled deeper exploration of the underlying trends, trade-offs, and theoretical implications. This separation was particularly valuable given the complexity of the experiments, where accuracy improvements were often intertwined with variations in token efficiency, resource requirements, and robustness across different models. By splitting these sections, the chapter maintains a logical flow that enhances readability and provides space for nuanced interpretation of results, ensuring that technical details are conveyed transparently while broader insights are contextualized within the field.

4.7 Conclusion of Results and Discussion

This chapter presented and analyzed the results of evaluating QualCoder against existing approaches, with experiments conducted on HumanEval and CodeContest benchmarks. The findings confirmed that QualCoder consistently outperforms MapCoder and direct prompting approaches, delivering improvements in both accuracy and efficiency. Ablation studies further revealed the computational trade-offs inherent in multi-agent coordination, highlighting both the benefits of structured quality assurance and the challenges of scaling to smaller, resource-constrained models. The discussion also identified limitations such as increased token usage, output robustness

issues, and synchronization complexity, which provide important considerations for future work. Overall, the results demonstrate the effectiveness of QualCoder's architecture in advancing reliable and efficient code generation, laying the foundation for the broader conclusions and future directions discussed in the final chapter.

Chapter 5

Conclusion

5.1 Research Overview

This thesis discussed the limitations of single-agent and homogeneous multi-agent systems in the area of automatic code generation. We proposed a modular, role-specialized multi-agent architecture using lightweight, open-source models for the purpose of enhanced efficiency, scalability, and flexibility. Our system design includes dynamic task decomposition, adaptive quality control, and modular agent communication to enable greater cooperation between multiple agents.

5.2 Key Contributions

This research contributes to four main areas:

- **Framework Design:** A heterogeneous, role-specialized multi-agent architecture aligning model strengths with specific agent functions.
- **Quality Assurance Mechanism:** An expert QA agent that works to increase the trustworthiness of strategies through confidence scoring and iterative refinement.
- **Adaptation Strategies:** A systematic integration of various minor models into multi-agent systems to improve adaptability.
- **Open Science Commitment:** The release of codebases, templates and evaluation protocols aims to promote reproducibility and accessibility.

5.3 Limitations and Future Studies

Although the model has significant theoretical potential, extensive benchmarking and careful empirical testing are ongoing. Potential research areas include:

- Implementing comprehensive experimental verifications over a range of benchmarks.
- Enhancing dynamic cost-sensitive agent orchestration and memory sharing.
- Including symbolic or formal verification tools.
- Extending the current framework to incorporate repository-level code generation with human-in-the-loop supervision.

5.4 Final Reflections

Our study underscores the critical importance of both coordination and role-specific specialization in multi-agent LLM systems, showing that their contribution can be as significant as model size when it comes to achieving high performance. By integrating a Quality Assurance (QA) Agent and exploring heterogeneous, role-aware configurations, we demonstrate that structured collaboration between agents not only improves accuracy but also enhances efficiency, reduces error propagation, and optimizes resource usage.

This work lays a solid foundation for the scalable, efficient, and resilient generation of code using AI. It illustrates that intelligent orchestration of multiple models, each specialized for specific tasks, can bridge the gap between raw model capabilities and practical coding performance. Beyond immediate performance gains, our approach points toward the broader vision of autonomous software generation, where AI systems can handle complex programming challenges reliably with minimal human oversight.

Looking forward, these insights open several avenues for future research, including more adaptive controllers for dynamic task allocation, deeper integration of QA and verification mechanisms, and exploration of even more resource-efficient model combinations. Overall, this research represents a significant step toward realizing AI-driven software development that is both robust and accessible, paving the way for a new era of intelligent code generation.

References

[1] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics*, pp. 2655–2668, 2021. [Online]. Available: https://aclanthology.org/2021.naacl-main.211/

[2] J. Austin et al., "Program synthesis with large language models," *arXiv preprint arXiv:2108.07732*, 2021. [Online]. Available: https://arxiv.org/abs/2108.07732

[3] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow, "Deepcoder: Learning to write programs," *International Conference on Learning Representations*, 2016. [Online]. Available: https://arxiv.org/abs/1611.01989

[4] F. Cassano et al., "Multipl-e: A scalable and polyglot approach to benchmarking neural code generation," *IEEE Transactions on Software Engineering*, 2023. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/TSE.2023.3267446

[5] M. Chen et al., "Evaluating large language models trained on code," *arXiv*, 2021. DOI: 10.48550/arXiv.2107.03374 arXiv: 2107.03374 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2107.03374

[6] W. Chen et al., *Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors*, 2023. arXiv: 2308.10848 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2308.10848

[7] X. Chen, M. Lin, N. Schärli, and D. Zhou, "Teaching large language models to self-debug," *arXiv preprint arXiv:2304.05128*, 2023. [Online]. Available: https://arxiv.org/abs/2304.05128

[8] Z. Chen et al., "Agilecoder: Dynamic collaborative agents for software development based on agile methodology," *arXiv preprint arXiv:2406.11912*, 2024. [Online]. Available: https://arxiv.org/abs/2406.11912

[9] J. Devlin, J. Uesato, S. Bhupatiraju, R. Singh, A.-r. Mohamed, and P. Kohli, *Robustfill: Neural program learning under noisy i/o*, 2017. arXiv: 1703.07469 [cs.AI]. [Online]. Available: https://arxiv.org/abs/1703.07469

[10] Y. Dong, X. Jiang, Z. Jin, and G. Li, "Self-collaboration code generation via chatgpt," *arXiv preprint arXiv:2304.07590*, 2023. [Online]. Available: https://arxiv.org/abs/2304.07590

[11] S. Du et al., *A survey on the optimization of large language model-based agents*, 2025. arXiv: 2503.12434 [cs.AI]. [Online]. Available: https://arxiv.org/abs/2503.12434

[12] Y. Du, S. Li, A. Torralba, J. B. Tenenbaum, and I. Mordatch, *Improving factuality and reasoning in language models through multiagent debate*, 2023. arXiv: 2305.14325 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2305.14325

[13] Z. Feng et al., "Codebert: A pre-trained model for programming and natural languages," *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: Findings*, 2020. [Online]. Available: https://aclanthology.org/2020.findings-emnlp.139/

[14] Y. Gao et al., "Intervenor: A multi-agent interactive framework for software bug localization and repair," *arXiv preprint arXiv:2311.09868*, 2023. [Online]. Available: https://arxiv.org/abs/2311.09868

[15] D. Guo et al., "Graphcodebert: Pre-training code representations with data flow," *International Conference on Learning Representations*, 2021. [Online]. Available: https://arxiv.org/abs/2009.08366

[16] D. Guo et al., "Deepseek-coder: When the large language model meets programming–the rise of code intelligence," *arXiv preprint arXiv:2401.14196*, 2024. [Online]. Available: https://arxiv.org/abs/2401.14196

[17] D. Hendrycks et al., "Measuring coding challenge competence with apps," *arXiv*, 2021. DOI: 10.48550/arXiv.2105.09938 arXiv: 2105.09938 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2105.09938

[18] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," *Communications of the ACM*, vol. 55, no. 3, pp. 103–113, 2012. [Online]. Available: https://dl.acm.org/doi/10.5555/2337223.2337322

[19] S. Hong et al., "Metagpt: Meta programming for a multi-agent collaborative framework," *arXiv preprint arXiv:2308.00352*, 2023. [Online]. Available: https://arxiv.org/abs/2308.00352

[20] S. Hong et al., *Metagpt: Meta programming for a multi-agent collaborative framework*, 2024. arXiv: 2308.00352 [cs.AI]. [Online]. Available: https://arxiv.org/abs/2308.00352

[21] Y. Hu et al., *Self-evolving multi-agent collaboration networks for software development*, 2024. arXiv: 2410.16946 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2410.16946

[22] D. Huang, Q. Bu, J. M. Zhang, M. Luck, and H. Cui, "Agentcoder: Multi-agent-based code generation with iterative testing and optimisation," *arXiv*, 2024. DOI: 10.48550/arXiv.2312.13010 arXiv: 2312.13010 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2312.13010

[23] D. Huang, J. M. Zhang, M. Luck, Q. Bu, Y. Qing, and H. Cui, *Agentcoder: Multi-agent-based code generation with iterative testing and optimisation*, 2024. arXiv: 2312.13010 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2312.13010

[24] Y. Ishibashi and Y. Nishimura, "Self-organized agents: A llm multi-agent framework toward ultra large-scale code generation and optimization," *arXiv preprint arXiv:2404.02183*, 2024. [Online]. Available: https://arxiv.org/abs/2404.02183

[25] M. A. Islam, M. E. Ali, and M. R. Parvez, *Mapcoder: Multi-agent code generation for competitive problem solving*, 2024. arXiv: 2405.11403 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2405.11403

[26] P. Islam Ali, "Mapcoder: Multi-agent code generation for competitive problem solving," *arXiv preprint arXiv:2405.11403*, 2024. [Online]. Available: https://arxiv.org/abs/2405.11403

[27] N. Jain, K. H. Gadepalli, T. Kallarackal, P. Shi, L. He, et al., "Livecodebench: Holistic and contamination free evaluation of large language models for code," *arXiv preprint arXiv:2403.07974*, 2024. [Online]. Available: https://arxiv.org/abs/2403.07974

[28] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, *A survey on large language models for code generation*, 2024. arXiv: 2406.00515 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2406.00515

[29] C. E. Jiménez et al., "Swe-bench: Can language models resolve real-world github issues?" *arXiv preprint arXiv:2310.06770*, 2024. [Online]. Available: https://arxiv.org/abs/2310.06770

[30] M. A. M. Khan, M. S. Bari, X. L. Do, W. Wang, M. R. Parvez, and S. Joty, *Xcodeeval: A large scale multilingual multitask benchmark for code understanding, gen-*

eration, translation and retrieval, 2023. arXiv: 2303.03004 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2303.03004

[31] Y. Lai et al., "Ds-1000: A natural and reliable benchmark for data science code generation," *International Conference on Machine Learning*, pp. 18 319–18 345, 2023. [Online]. Available: https://ds1000-code-gen.github.io/

[32] J. Li, G. Zhang, Y. Wang, W. Chen, and M. Liu, "Codepori: A multi-agent system for solving large and complex programming problems," *arXiv preprint arXiv:2402.01411*, 2024. [Online]. Available: https://arxiv.org/abs/2402.01411

[33] R. Li et al., "Starcoder: May the source be with you!" *arXiv preprint arXiv:2305.06161*, 2023. [Online]. Available: https://arxiv.org/abs/2305.06161

[34] Y. Li, K. Ganchev, Z. Fan, M. D. Hoffman, M. Ranzato, and O. Vinyals, *Competition-level code generation with alphacode*, 2022. arXiv: 2203.07814 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2203.07814

[35] Z. Li, X. Zheng, Y. Chen, H. Liu, X. Chen, J. Yang, et al., "L2mac: Large language model automatic computer for unbounded code generation," *arXiv preprint arXiv:2409.02141*, 2024. [Online]. Available: https://arxiv.org/abs/2310.02003

[36] Z. Lin, Y. Shen, Q. Cai, H. Sun, J. Zhou, and M. Xiao, "Autop2c: An llm-based agent framework for code repository generation from multimodal content in academic papers," 2025. arXiv: 2504.20115 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2504.20115

[37] W. Ling et al., "Latent predictor networks for code generation," *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, pp. 599–609, 2016. [Online]. Available: https://aclanthology.org/P16-1057/

[38] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, Y. Belinkov, and P. Liang, "Lost in the middle: How language models use long contexts," *arXiv preprint arXiv:2307.03172*, 2023. [Online]. Available: https://arxiv.org/abs/2307.03172

[39] X. Liu, Y. Zhang, W. Chen, and L. Wang, "X-mas: Towards building multi-agent systems with heterogeneous llms," *arXiv preprint arXiv:2505.16997*, 2025. [Online]. Available: https://arxiv.org/abs/2505.16997

[40] S. Lu et al., "Codexglue: A machine learning benchmark dataset for code understanding and generation," *arXiv preprint arXiv:2102.04664*, 2021. [Online]. Available: https://arxiv.org/abs/2102.04664

[41] J. Luo et al., *Large language model agent: A survey on methodology, applications and challenges*, 2025. arXiv: 2503.21460 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2503.21460

[42] Z. Manna and R. J. Waldinger, "Toward automatic program synthesis," *Communications of the ACM*, vol. 14, no. 3, pp. 151–165, 1971. [Online]. Available: https://doi.org/10.1145/362566.362568

[43] A. Nunez, N. T. Islam, S. K. Jha, and P. Najafirad, *Autosafecoder: A multi-agent framework for securing llm code generation through static analysis and fuzz testing*, 2024. arXiv: 2409.10737 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2409.10737

[44] OpenAI, *Humaneval*, GitHub repository, 2021. [Online]. Available: https://github.com/openai/human-eval

[45] OpenAI, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023. [Online]. Available: https://arxiv.org/abs/2303.08774

[46] OpenRouter, *OpenRouter*, Accessed: 2025-07-29, 2023. [Online]. Available: https://openrouter.ai/

[47] R. Pan, H. Zhang, and C. Liu, *Codecor: An llm-based self-reflective multi-agent framework for code generation*, 2025. arXiv: 2501.07811 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2501.07811

[48] J. S. Park, J. C. O'Brien, C. J. Cai, M. R. Morris, P. Liang, and M. S. Bernstein, *Generative agents: Interactive simulacra of human behavior*, 2023. arXiv: 2304.03442 [cs.HC]. [Online]. Available: https://arxiv.org/abs/2304.03442

[49] C. Qian et al., *Chatdev: Communicative agents for software development*, 2024. arXiv: 2307.07924 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2307.07924

[50] S. Ren et al., "Codebleu: A method for automatic evaluation of code synthesis," *arXiv preprint arXiv:2009.10297*, 2020. [Online]. Available: https://arxiv.org/abs/2009.10297

[51] B. Roziere et al., "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023. [Online]. Available: https://arxiv.org/abs/2308.12950

[52] N. Shinn, E. Labash, S. Yao, et al., *Reflexion: Language agents with verbal reinforcement learning*, 2023. arXiv: 2303.11366 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2303.11366

[53] W. Tao, Y. Wang, W. Jia, W. Che, and E. Wang, "Magis: Llm-based multi-agent framework for github issue resolution," *arXiv preprint arXiv:2403.17927*, 2024. [Online]. Available: https://arxiv.org/abs/2403.17927

[54] C. Tony, M. Mutas, N. E. D. Ferreyra, and R. Scandariato, "Securityeval dataset: Mining vulnerability examples to evaluate machine learning-based code generation techniques," *Proceedings of the 1st International Workshop on Mining Software Repositories Applications for Privacy and Security*, pp. 29–38, 2023. [Online]. Available: https://doi.org/10.1145/3549035.3561184

[55] A. Vaswani et al., "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017. [Online]. Available: https://arxiv.org/abs/1706.03762

[56] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, M.-F. Moens, X. Huang, L. Specia, and S. W.-t. Yih, Eds., Online and Punta Cana, Dominican Republic: Association for Computational Linguistics, Nov. 2021, pp. 8696–8708. DOI: 10.18653/v1/2021.emnlp-main.685 [Online]. Available: https://aclanthology.org/2021.emnlp-main.685/

[57] Z. Wang, Y. Zhang, X. Liu, and Z. Chen, "Codesim: Multi-agent code generation and problem solving through simulation-driven planning and debugging," *arXiv preprint arXiv:2502.05664*, 2025. [Online]. Available: https://arxiv.org/abs/2502.05664

[58] J. Wei et al., "Chain-of-thought prompting elicits reasoning in large language models," in *Proceedings of the 36th International Conference on Neural Information Processing Systems*, ser. NIPS '22, New Orleans, LA, USA: Curran Associates Inc., 2022, ISBN: 9781713871088. [Online]. Available: https://dl.acm.org/doi/10.5555/3600270.3602070

[59] M. V. Wilkes, D. J. Wheeler, and S. Gill, "The preparation of programs for an electronic digital computer," *Addison-Wesley*, 1949.

[60] Z. Xu, S. Jain, and M. Kankanhalli, "Hallucination is inevitable: An innate limitation of large language models," *arXiv preprint arXiv:2401.11817*, 2024. [Online]. Available: https://arxiv.org/abs/2401.11817

[61] D. Yang et al., "Docagent: A multi-agent system for automated code documentation generation," 2025. arXiv: 2504.08725 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2504.08725

[62] S. Yao et al., *Tree of thoughts: Deliberate problem solving with large language models*, 2023. arXiv: 2305.10601 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2305.10601

[63] P. Yin and G. Neubig, "A syntactic neural model for general-purpose code generation," *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*, 2017. [Online]. Available: https://aclanthology.org/P17-1041/

[64] E. Zelikman, Q. Huang, G. Poesia, N. D. Goodman, and N. Haber, "Parsel: A unified natural language framework for algorithmic reasoning," *arXiv preprint arXiv:2212.10561*, 2023. [Online]. Available: https://arxiv.org/abs/2212.10561

[65] W. Zhang, M. Liu, Y. Chen, and L. Wang, "Muarf: Multi-agent workflows for automated code refactoring," *arXiv preprint arXiv:2411.08742*, 2024. [Online]. Available: https://doi.org/10.1109/ICSE-Companion66252.2025.00071

[66] Y. Zhang, J. Wang, M. Li, and W. Chen, "Adacoder: An adaptive planning and multi-agent framework for function-level code generation," *arXiv preprint arXiv:2504.04220*, 2024. [Online]. Available: https://arxiv.org/abs/2504.04220

[67] Y. Zhu et al., *Adacoder: An adaptive planning and multi-agent framework for function-level code generation*, 2025. arXiv: 2504.04220 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2504.04220

[68] T. Y. Zhuo, M. Huang, L. Xiong, et al., "Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions," *arXiv preprint arXiv:2406.15877*, 2024. [Online]. Available: https://arxiv.org/abs/2406.15877