A Survey of Large Language Models in Low-Resource Code Generation and Software Development Automation

www.surveyx.cn

Abstract

This survey explores the transformative role of Large Language Models (LLMs) in software development, focusing on their impact on code generation, automation, and AI-assisted programming. LLMs have emerged as pivotal technologies, automating complex coding tasks and enhancing workflow efficiency. Key advancements include their integration with multi-agent systems and the development of Parameter-Efficient Fine-Tuning (PEFT) techniques, which optimize performance in low-resource environments. Despite these advancements, challenges remain, such as managing hallucinations in LLM outputs and ensuring security in generated code. The survey highlights innovative strategies like the Multi-Programming Language Ensemble (MPLE) and the importance of structured prompting to improve LLM performance. Additionally, it underscores the need for robust evaluation frameworks to address the limitations of existing benchmarks and enhance LLM applicability across diverse programming environments. Future research should focus on expanding LLM capabilities, improving model efficiency, and addressing ethical implications to fully realize their potential in revolutionizing software development.

1 Introduction

1.1 Scope and Significance

This survey investigates the transformative potential of Large Language Models (LLMs) in software development, particularly their automation capabilities in complex coding tasks such as code generation and Infrastructure as Code (IaC) across various platforms. LLMs have emerged as essential technologies, enhancing software development efficiency and decision-making across sectors. The survey emphasizes LLMs' ability to generate specifications for static verification, especially in business contracts and formal specifications, thereby automating intricate coding processes [1].

A significant focus of this survey is the evaluation of LLMs for code generation from natural language inputs, intentionally excluding tasks like code summarization or comment generation to maintain a targeted analysis [2]. This focus highlights LLMs' potential to tackle software testing challenges by improving techniques and bridging knowledge gaps [3]. The integration of LLMs into Business Process Management (BPM) is also examined, showcasing their ability to enhance and automate organizational activities [4].

Additionally, the survey explores LLMs' impact on automating machine learning tasks, such as AutoML, through natural language interfaces that increase usability for non-expert users. Their role in customer service automation further illustrates their broader applicability in enhancing agent efficiency via natural language processing (NLP) technologies [5].

By addressing these aspects, the survey aims to provide a comprehensive overview of the current landscape and future directions of LLMs in software development, emphasizing their significance in

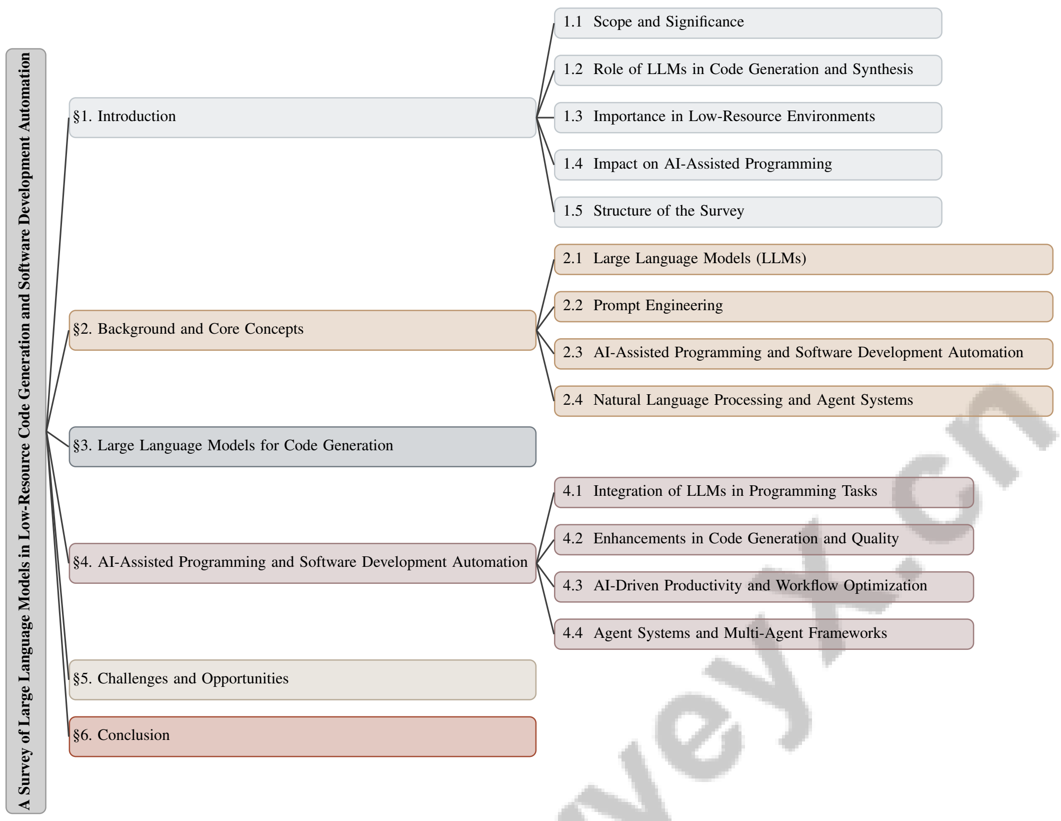

A Survey of Large Language Models in Low-Resource Code Generation and Software Development Automation
§1. Introduction
1.1 Scope and Significance
1.2 Role of LLMs in Code Generation and Synthesis
1.3 Importance in Low-Resource Environments
1.4 Impact on AI-Assisted Programming
1.5 Structure of the Survey
§2. Background and Core Concepts
2.1 Large Language Models (LLMs)
2.2 Prompt Engineering
2.3 AI-Assisted Programming and Software Development Automation
2.4 Natural Language Processing and Agent Systems
§3. Large Language Models for Code Generation
§4. AI-Assisted Programming and Software Development Automation
4.1 Integration of LLMs in Programming Tasks
4.2 Enhancements in Code Generation and Quality
4.3 AI-Driven Productivity and Workflow Optimization
4.4 Agent Systems and Multi-Agent Frameworks
§5. Challenges and Opportunities
§6. Conclusion

Figure 1: chapter structure

revolutionizing the field. It also discusses the challenges and opportunities in leveraging LLMs for code generation in Low-Resource Programming Languages (LRPLs) and Domain-Specific Languages (DSLs), filling a notable gap in existing literature [6]. Furthermore, the survey examines the synergies between LLMs and Evolutionary Algorithms (EAs), seeking to establish a complementary relationship in solving complex problems [2].

Understanding legacy code in large repositories poses a significant challenge, with only 15.4% of Java GitHub codes documented, complicating developers' understanding of functionality [7]. This survey aims to bridge such knowledge gaps by providing a thorough overview of advancements in LLMs, addressing their capabilities, architectures, and applications [8]. It also explores the intersection of LLMs and optimization algorithms, covering their development, application scenarios, and future research directions [9].

1.2 Role of LLMs in Code Generation and Synthesis

Large Language Models (LLMs) such as ChatGPT and GitHub Copilot are pivotal in transforming software development through the automation of code generation and synthesis. These advanced models convert natural language inputs into executable code, leveraging their understanding of semantic facts and patterns in source code to enhance developer productivity and reduce cognitive load, thus enabling more efficient code documentation, generation, and summarization [10, 11, 7, 12]. By integrating LLMs into development environments, developers can translate conceptual ideas into functional software more efficiently, streamlining workflows and improving the software development lifecycle.

Beyond basic code generation, LLMs automate critical software testing tasks, including unit test case generation, test oracle generation, and program repair, addressing essential aspects of software quality assurance [3]. They also facilitate BPM task automation, such as extracting information from unstructured textual documents, highlighting their transformative impact on process management [4].

Despite their capabilities, LLMs encounter challenges in independently generating reliable code, often producing insecure outputs that may lead to vulnerabilities [13]. The lack of diversity in search methods for code generation and the risk of generating license-protected code without proper licensing information underscore the necessity for innovative approaches to enhance the robustness and reliability of LLM-generated code.

To address these challenges, methodologies like the Multi-Programming Language Ensemble (MPLE) have been developed, which integrate outputs from various programming languages to mitigate language-specific errors and biases, thereby improving the robustness of generated code [14]. Additionally, decomposing large programs into smaller, independent fragments allows for more efficient processing by LLMs, enhancing the accuracy and reliability of code generation [15].

LLMs also aid in synthesizing comprehensive natural language explanations, which enhance code comprehension and documentation. This capability enriches the educational dimension of programming, enabling learners to better grasp complex codebases and underlying logic [16]. Moreover, causality analysis-based approaches provide a systematic framework for evaluating and explaining LLMs' code generation capabilities, offering insights into the causal relationships between input prompts and generated code [17].

Benchmarks like WebApp1K facilitate comparisons among frontier LLMs, providing insights into their code generation capabilities [18]. Tools like DEBUGEVAL evaluate LLMs' debugging capabilities across tasks such as bug localization, identification, code review, and repair, further underscoring their transformative impact on software development [19].

LLMs have fundamentally altered the landscape of code generation and synthesis, offering innovative solutions that automate complex coding tasks and enhance software development processes. As LLMs continue to evolve, they are expected to further transform software development by improving efficiency, increasing accessibility, and aligning more closely with human cognitive processes. Increasingly utilized as AI Pair Programming Assistants, LLMs automate various tasks across the software development lifecycle—from requirements elicitation to code generation and testing. While this shift presents significant benefits, it also introduces challenges such as data privacy concerns, biases, and the need for effective prompt engineering to ensure accurate outputs. Addressing these challenges is crucial for maximizing LLMs' potential in software engineering and ensuring their seamless integration into the development process [20, 21, 22, 10, 23].

1.3 Importance in Low-Resource Environments

In low-resource environments, deploying Large Language Models (LLMs) is vital for democratizing access to advanced programming capabilities. These regions often face significant challenges due to the high computational and memory demands of LLMs, which limit accessibility for users with constrained resources [9]. The scarcity of high-quality, domain-specific training data further complicates the generation of syntactically and semantically accurate code for Low-Resource Programming Languages (LRPLs) and Domain-Specific Languages (DSLs) [24].

To mitigate these challenges, parameter-efficient fine-tuning techniques have been developed to reduce computational requirements while maintaining performance in code generation tasks [25]. The AutoML-Agent framework exemplifies this approach by providing an accessible platform for users with limited AI expertise, enhancing LLM applicability in low-resource environments [24]. Additionally, the PairCoder system, which utilizes two collaborative LLM agents—a Navigator for high-level planning and a Driver for implementation—demonstrates LLMs' potential to generate and refine code efficiently in resource-constrained settings [26].

Despite these advancements, LLMs still face challenges, such as limited context window lengths that restrict input text processing, resulting in incomplete analyses [15]. The operational costs and time consumption associated with existing methods highlight the impracticality of traditional approaches, emphasizing LLMs' importance in optimizing code generation in low-resource environments [18]. Moreover, developing benchmarks to generate domain-specific data through LLMs can improve pre-trained language models' performance, enhancing their applicability in these contexts [5].

Furthermore, the LPW approach addresses challenges in text-to-code generation in low-resource settings by enhancing LLMs' efficiency and reasoning capacity through structured methodologies [27]. By automating traditionally manual and error-prone processes, LLMs offer significant benefits in constrained environments, facilitating efficient and accurate code generation [28]. However, current benchmarks often fail to simulate the complexities of multi-turn interactions and do not adequately capture the nuances of algorithmic reasoning required for high-stakes programming challenges [29].

The need for effective human evaluation and prompt engineering remains a primary challenge in leveraging the full capabilities of generative LLMs [22]. Existing methods often yield incremental improvements based on a single programmer's thought patterns, limiting the scope for significant algorithmic advancements [30]. As LLMs evolve, their ability to optimize code generation and enhance software development in low-resource environments will be crucial in driving innovation and expanding the reach of programming tools to underserved communities.

1.4 Impact on AI-Assisted Programming

The integration of Large Language Models (LLMs) into AI-assisted programming has significantly reshaped the software development landscape by enhancing productivity and efficiency. LLMs facilitate dynamic prompt refinement and contextual adaptability, essential for improving productivity in document generation and coding tasks [31]. By allowing developers to begin with informal descriptions and iteratively refine their requirements, tools like ASCUS streamline the specification process, enhancing development workflows [32].

Despite these advancements, deploying LLMs in AI-assisted programming presents challenges. A notable concern is the 'hallucination' phenomenon, where LLMs generate plausible yet incorrect outputs, raising questions about the integrity and correctness of the produced code. This necessitates robust evaluation mechanisms to ensure safety and reliability in AI-assisted programming environments [33]. The need for improved training on effective prompting strategies is emphasized to mitigate overreliance on LLMs and enhance their utility [34].

LLMs offer numerous advantages, including improved automation, reduced reliance on human labor, and enhanced accuracy in network operations through intelligent control [6]. Parameter-Efficient Fine-Tuning (PEFT) techniques have been crucial in democratizing access to powerful LLMs, enabling effective fine-tuning in resource-constrained environments. This approach ensures that even users with limited computational resources can leverage LLMs' full potential, broadening the reach of AI-assisted programming tools.

Current research in AI-assisted programming has made significant strides, particularly in automating programming tasks, enhancing code quality, and boosting developer productivity. Notable advancements include integrating LLMs in automated program repair, which improves bug detection and context-aware fixes, thereby reducing manual debugging efforts. Additionally, innovative techniques in code generation, such as identifier-aware training and semantic code structure incorporation, have emerged, facilitating more accurate and efficient coding. Furthermore, intelligent assistants that mine existing software repositories are aiding developers in creating checkable specifications for code, streamlining subsystem development. Collectively, these advancements enhance the programming experience, making it more efficient and effective for developers [35, 32, 2]. As LLM technologies continue to evolve, their role in AI-assisted programming is expected to expand, driving further innovation and efficiency in software development.

1.5 Structure of the Survey

This survey is meticulously organized to provide readers with an in-depth exploration of the intricate role of Large Language Models (LLMs) in software development automation and low-resource code generation, highlighting recent advancements, performance comparisons between general and specialized Code LLMs, and practical insights for effectively deploying these models in real-world applications [36, 37]. The introduction sets the stage by elucidating the scope and significance of LLMs in transforming software development processes, particularly in low-resource settings. It also emphasizes LLMs' role in code generation and synthesis, underscoring their impact on AI-assisted programming.

Following the introduction, the survey delves into background and core concepts, providing an overview of key topics such as LLMs, prompt engineering, AI-assisted programming, and the integration of Natural Language Processing (NLP) and agent systems in software development automation. This section establishes a foundational understanding necessary for comprehending subsequent discussions.

The third section explores LLMs' capabilities in code generation, examining their strengths and limitations while discussing advancements in prompt engineering that enhance performance. Strategies for optimizing LLMs in low-resource environments are also addressed, highlighting innovative approaches to mitigate resource constraints.

In the fourth section, the survey discusses AI integration in programming and its role in automating software development tasks. It highlights how LLMs enhance developer productivity and improve code quality, while also exploring the impact of agent systems and AI-assisted tools in streamlining development workflows.

The penultimate section identifies challenges and opportunities in deploying LLMs for code generation and software automation, particularly in low-resource settings. It discusses potential strategies for improving model efficiency and expanding LLM applicability across diverse programming environments.

The conclusion integrates the key findings of the research, emphasizing LLMs' transformative potential in software development. It highlights their ability to enhance early-stage coding tasks, such as generating foundational code structures and debugging, while also addressing challenges faced in low-resource environments. Furthermore, the conclusion outlines future directions for LLM integration in educational curricula and real-world applications, advocating for strategies that promote effective human-AI collaboration and mitigate overreliance on these tools in academic settings [38, 34, 39, 40, 41]. This structured approach ensures a comprehensive understanding of the current landscape and future prospects of LLMs in software development automation.The following sections are organized as shown in Figure 1.

2 Background and Core Concepts

2.1 Large Language Models (LLMs)

Large Language Models (LLMs) are central to artificial intelligence, adept at comprehending and generating human-like text through extensive datasets and sophisticated neural architectures. In software development, they translate natural language prompts into executable code, streamlining processes and easing developers' cognitive load [24]. LLMs are instrumental in generating syntactically and functionally correct code, vital for applications such as blockchain smart contracts [9].

Beyond code generation, LLMs facilitate automated program repair (APR) and software testing, enhancing customer service through auto-generated responses. However, the economic implications of their deployment warrant consideration [5]. Their effectiveness in identifying and rectifying bugs is demonstrated through benchmarks focused on Java code repair [7], while they also improve software testing by automating test case generation and bug localization [3].

Despite their potential, LLMs face challenges such as producing incorrect outputs in simple tasks, necessitating improved model robustness and internal understanding [33]. Their educational integration showcases versatility, offering personalized programming instruction and adapting to diverse prompts for accurate code generation [42]. The development of Multilingual Large Language Models (MLLMs) enhances linguistic inclusivity, enabling LLM deployment in multilingual settings [8].

LLMs have evolved significantly, with architectures like GPT broadening their capabilities [9]. Frameworks such as AutoML-Agent leverage LLMs to automate the AutoML pipeline, enhancing accessibility and usability [24]. As LLMs continue to advance, their role in software development automation is set to expand, driven by innovations in prompt engineering and new frameworks for general and domain-specific applications [38].

2.2 Prompt Engineering

Prompt engineering is crucial for optimizing LLMs in code generation and reasoning tasks, involving the crafting of precise prompts that guide LLMs to produce accurate outputs. This enhances efficiency and reliability, particularly in emulating human language styles and integrating content into systems.

However, recent benchmarks suggest that prompt engineering's benefits may be less significant for advanced LLMs, necessitating a reevaluation of its effectiveness [21].

Structured frameworks like the Prompt Pattern Catalog (PPC) standardize reusable prompt patterns, improving LLM interactions [43]. This reduces cognitive load, especially for novices, facilitating intuitive AI tool interactions [10]. In education, effective prompt engineering maximizes LLM potential, enabling personalized learning experiences [42].

Prompt engineering strategies, including prompt template engineering, prompt answer engineering, and multi-prompting, enhance LLM performance across contexts [44]. Techniques like Prompt Engineering for Tool Calling (PETC) expand LLM functionality by enabling interaction with external tools without fine-tuning [45].

Understanding user behaviors, prompt modifications, and contextual applications is vital for improving LLM interactions and aligning prompts with human reasoning [46]. The choice of metrics for evaluating prompt engineering significantly influences insights into model behavior, focusing on stability and reliability beyond traditional accuracy metrics [47].

2.3 AI-Assisted Programming and Software Development Automation

AI integration marks a transformative shift in software development automation, primarily through LLMs that enhance productivity and streamline tasks. Methodologies optimizing LLM capabilities improve response integrity and reflect progress in robotic capabilities [48]. AI-assisted programming transforms workflows, addressing system limitations by ensuring accuracy and comprehensiveness in evaluation methods [49].

LLM-based automation agents execute complex tasks autonomously, raising trustworthiness concerns about their accuracy and reliability without human oversight [50]. LLMs' integration into software testing exemplifies their potential, facilitating stages of the testing lifecycle, including test case preparation, bug analysis, debugging, and repair [3].

Evaluating LLM agents' performance in multi-hop reasoning and decision-making highlights AI-assisted programming complexities. Benchmarks by [51] assess LLM agents' effectiveness in intricate reasoning processes. Additionally, LLMs generate natural language explanations for code snippets, enhancing code comprehension and documentation [7].

As AI permeates programming, software development task automation is expected to expand, driven by advancements in LLM architectures and methodologies. Generative LLMs are set to reshape software development by improving efficiency, accessibility, and alignment with industry practices. These models automate complex tasks throughout the development lifecycle, enhancing prompt engineering precision and fostering better developer-AI interactions. AI-assisted programming's evolution is poised to redefine software development, offering unprecedented opportunities for innovation and efficiency [38, 32, 52, 22, 10].

2.4 Natural Language Processing and Agent Systems

Natural Language Processing (NLP) and agent systems are pivotal in developing LLM-driven software systems, enhancing automation and efficiency. NLP enables LLMs to interpret and generate human language, crucial for translating programming requirements into executable code, streamlining tasks, and reducing developers' cognitive burdens [53].

Multi-Agent Systems (MAS) enhance LLM capabilities by automating tasks through collaborative efforts among agents. These systems operate autonomously, sharing information to achieve common goals within the software lifecycle. Integrating LLMs with MAS improves task automation, process efficiency, and decision-making [54].

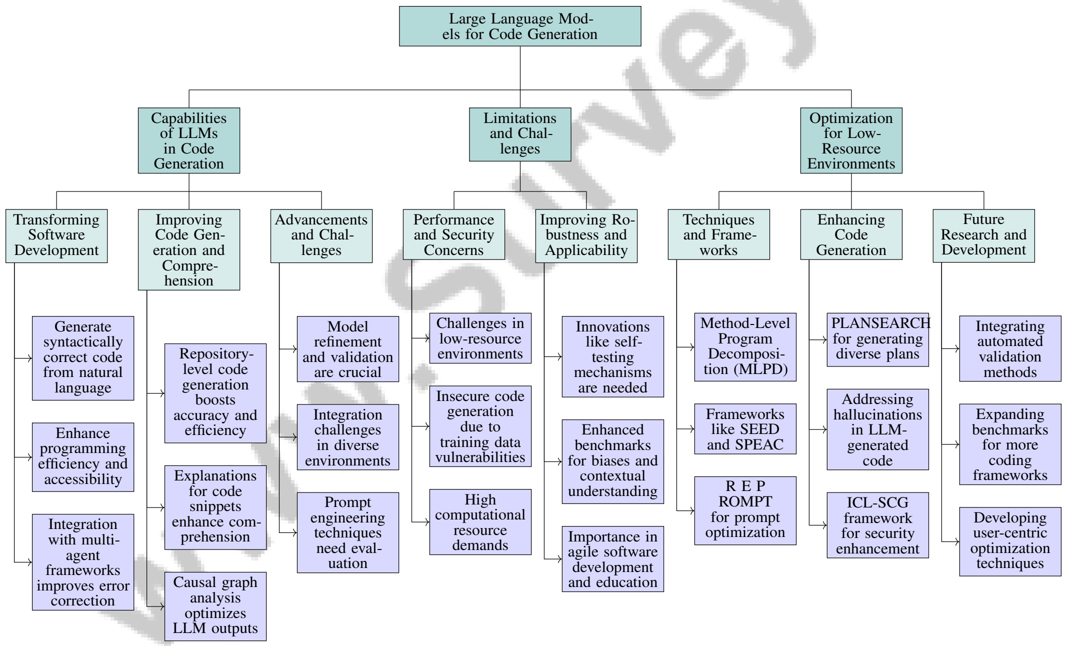
Architectural frameworks for Multi-LLM-Agent Systems (MLAS) include centralized, decentralized, and fully decentralized architectures, each offering distinct advantages regarding control and data access. Centralized architectures provide a single control point, while decentralized architectures enhance robustness and scalability. Fully decentralized architectures promote autonomy and resilience [54].

AI-generated, for reference only.

In multilingual contexts, developing Multilingual Large Language Models (MLLMs) is crucial for extending LLM applicability across diverse linguistic environments. A comprehensive MLLM framework ensures LLMs handle linguistic diversity in global software projects [55]. This broadens LLM-driven systems' reach, enhancing their ability to process and generate content in multiple languages, fostering inclusive software solutions.

The synergy between NLP, agent systems, and LLMs revolutionizes software development automation. Leveraging each component's strengths, these systems enhance efficiency, accuracy, and scalability. Research indicates LLMs improve the software development lifecycle, generating foundational code structures, assisting with syntax and error debugging, and enhancing user story quality in agile frameworks. These findings highlight the importance of integrating LLMs into development toolchains, boosting productivity and facilitating effective human-AI collaboration. The future of software design, development, and deployment is poised for transformation through these technologies, shaping a new industry paradigm [56, 37, 39].

In recent years, Large Language Models (LLMs) have significantly transformed the landscape of software development. Their capabilities and limitations, alongside various optimization strategies, are critical for understanding their role in code generation. Figure 2 illustrates the hierarchical structure of these aspects, providing a visual representation that delineates the transformative impact of LLMs on the field. This figure not only highlights the challenges and security concerns associated with LLMs but also presents techniques for optimizing their performance in low-resource environments. By integrating this visual framework, we can better appreciate the complexities involved in leveraging LLMs for effective software development.



Large Language Models for Code Generation
Capabilities of LLMs in Code Generation
Limitations and Challenges
Optimization for Low-Resource Environments
Transforming Software Development
Improving Code Generation and Comprehension
Advancements and Challenges
Performance and Security Concerns
Improving Robustness and Applicability
Techniques and Frameworks
Enhancing Code Generation
Future Research and Development
Generate syntactically correct code from natural language
Enhance programming efficiency and accessibility
Integration with multi-agent frameworks improves error correction
Repository-level code generation boosts accuracy and efficiency
Explanations for code snippets enhance comprehension
Causal graph analysis optimizes LLM outputs
Model refinement and validation are crucial
Integration challenges in diverse environments
Prompt engineering techniques need evaluation
Challenges in low-resource environments
Insecure code generation due to training data vulnerabilities
High computational resource demands
Innovations like self-testing mechanisms are needed
Enhanced benchmarks for biases and contextual understanding
Importance in agile software development and education
Method-Level Program Decomposition (MLPD)
Frameworks like SEED and SPEAC
R E P ROMPT for prompt optimization
PLANSEARCH for generating diverse plans
Addressing hallucinations in LLM-generated code
ICL-SCG framework for security enhancement
Integrating automated validation methods
Expanding benchmarks for more coding frameworks
Developing user-centric optimization techniques

Figure 2: This figure illustrates the hierarchical structure of the capabilities, limitations, and optimization strategies for Large Language Models (LLMs) in code generation. It delineates the transformative impact of LLMs on software development, highlights challenges and security concerns, and presents techniques for optimizing LLMs in low-resource environments.

3 Large Language Models for Code Generation

3.1 Capabilities of LLMs in Code Generation

Large Language Models (LLMs) like GPT-4, Claude 3.5 Haiku, and Mistral 7B are transforming software development by generating syntactically correct code from natural language inputs, thereby enhancing programming efficiency and accessibility. As illustrated in Figure 3, the capabilities of

LLMs in code generation can be categorized into three main areas: code generation, repository-level generation, and optimization and integration. Each of these areas encompasses specific tasks and methodologies that collectively demonstrate the transformative impact of LLMs on software development.

Despite their capabilities, these models sometimes produce logically flawed outputs, highlighting the need for enhanced logical reasoning [57]. LLMs facilitate complex coding tasks, such as generating formal specifications and test cases, and their integration with multi-agent frameworks like TRANSAGENT enhances error correction and code quality [32, 58]. The AutoML-Agent framework exemplifies LLMs' potential to automate the AutoML process, achieving high success rates in downstream tasks [24].

LLMs significantly improve repository-level code generation, boosting code completion accuracy and efficiency [38]. They excel in providing explanations for code snippets, even in zero-shot scenarios, enhancing code comprehension [7]. Innovative methodologies like causal graph analysis optimize LLM outputs by exploring the relationship between prompt features and code quality [17]. Furthermore, LLMs improve model performance and efficiency in optimization processes, underscoring their transformative impact on software development [9].

To fully exploit LLM capabilities, ongoing advancements in model refinement and validation are crucial. Evaluating prompt engineering techniques is essential, as traditional methods may not benefit sophisticated reasoning models. Addressing integration challenges in diverse programming environments can enhance productivity and reduce costs [20, 21].

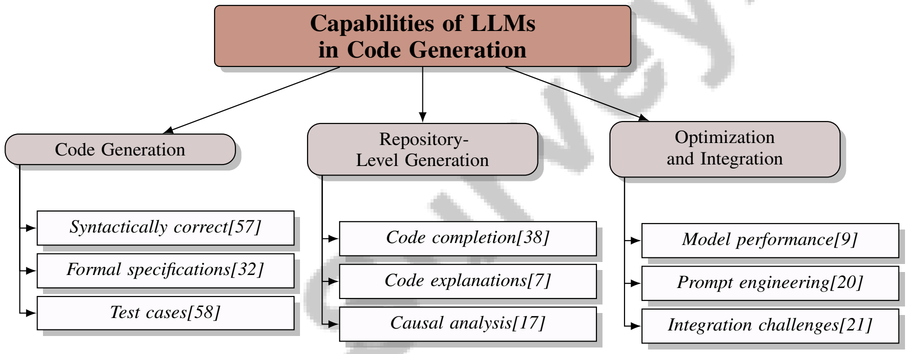

Capabilities of LLMs in Code Generation
Code Generation
Repository-Level Generation
Optimization and Integration
Syntactically correct[57]
Formal specifications[32]
Test cases[58]
Code completion[38]
Code explanations[7]
Causal analysis[17]
Model performance[9]
Prompt engineering[20]
Integration challenges[21]

Figure 3: This figure illustrates the capabilities of Large Language Models (LLMs) in code generation, highlighting three main areas: code generation, repository-level generation, and optimization and integration. Each area includes specific tasks and methodologies that demonstrate the transformative impact of LLMs on software development.

3.2 Limitations and Challenges

LLMs face challenges in code generation, especially in low-resource environments. Existing benchmarks often fail to capture coding tasks' complexity, leading to unreliable performance assessments [18]. Smaller models exacerbate this issue due to limited reasoning abilities, affecting code generation accuracy [59]. Large-scale industrial projects pose challenges due to intricate dependencies and extensive codebases, causing context window limitations and hindering comprehensive software understanding [9]. Benchmarks focusing on code repair often overlook debugging capabilities, limiting the understanding of LLMs' effectiveness in software development [19].

LLMs may generate insecure code due to reliance on training data with vulnerabilities, raising concerns about the functional correctness and security of outputs. Robust evaluation frameworks are needed to ensure compliance with licensing requirements and assess whether outputs are independently created or copied [60]. High computational resource demands and inherent complexities impede LLM integration into traditional optimization frameworks, limiting applicability in diverse contexts [9]. Economic implications of deploying LLMs present challenges, particularly in balancing operational costs with generating valuable responses in applications like customer service [61].

Addressing these limitations is crucial for enhancing LLM robustness and applicability across diverse environments. Innovations like self-testing mechanisms and improved benchmarks are vital for tackling challenges such as biases and contextual understanding, ensuring LLMs' practical applicability in low-resource settings and supporting business functions like strategic planning and data-driven decision-making. Recent studies emphasize LLMs' importance in agile software development and advanced computing education, highlighting innovations that improve user story quality and enhance learning outcomes [34, 56, 37, 41].

3.3 Optimization for Low-Resource Environments

Optimizing LLMs for low-resource environments is crucial for accessibility and efficiency. Techniques like Method-Level Program Decomposition (MLPD) address out-of-context issues, enabling LLMs to process extensive datasets [15]. Frameworks such as SEED and SPEAC customize LLMs with minimal training samples, revising errors during code generation to enhance performance in low-resource environments. These methodologies emphasize generating syntactically correct code using intermediate languages and advanced compiler techniques, improving LLM performance and facilitating essential code repairs [62, 15].

Frameworks like R E P ROMPT, utilizing interaction history and feedback for prompt optimization, enhance reasoning performance in low-resource settings [16]. PLANSEARCH, which generates diverse natural language plans to explore solutions before code generation, further improves LLM adaptability [63]. Addressing hallucinations in LLM-generated code is vital for enhancing capabilities. A comprehensive taxonomy of hallucinations provides insights into their distribution and causes, guiding research to improve LLM reliability [64]. Understanding code generation errors informs strategies to mitigate these errors, optimizing LLM performance in constrained environments [65].

The ICL-SCG framework employs In-Context Learning to enhance the security of LLM-generated code, particularly in low-resource settings where security concerns may be heightened [13]. Goal-oriented prompting methods, involving stages like goal decomposition and action execution, provide a structured framework for enhancing LLM outputs, ensuring contextually relevant and accurate code generation with limited resources [46].

Test-driven frameworks like TGen, combining problem statements with corresponding tests, ensure LLM-generated code meets specified requirements, improving reliability and functionality in low-resource environments [57]. Future research should focus on integrating automated validation methods and expanding benchmarks to encompass more coding frameworks and languages, enhancing effectiveness in these settings [18].

These strategies ensure LLMs remain effective and efficient in low-resource environments, driving innovation and expanding programming tools' reach to underserved communities. As research progresses, developing sophisticated and user-centric optimization techniques will be essential for fully leveraging LLM capabilities across various environments. Integrating LLMs with traditional optimization algorithms can enhance decision-making and problem-solving, refining architectures and output quality for effective real-world applications. Studies indicate effective prompting strategies can significantly improve LLM performance in educational settings, emphasizing tailored approaches to maximize potential in programming tasks [34, 9, 37, 66, 42].

4 AI-Assisted Programming and Software Development Automation

The convergence of artificial intelligence and software development has revolutionized the landscape by integrating advanced tools and methodologies. Among these, Large Language Models (LLMs) have become pivotal, reshaping programming paradigms and enhancing the software development lifecycle. This section examines the multifaceted role of LLMs in programming and automation, focusing on their integration into programming tasks as a foundation for understanding their broader impact on software development practices.

4.1 Integration of LLMs in Programming Tasks

Integrating Large Language Models (LLMs) into programming tasks has dramatically improved software development efficiency and productivity. LLMs automate code generation, reducing manual effort and enhancing output accuracy, which streamlines workflows. The Self-Organized multi-Agent framework (SoA) exemplifies this by enabling independent yet collaborative code generation and

AI-generated, for reference only.

modification, addressing scalability issues seen in single-agent methods [67]. This collaboration boosts developers' creative output and fosters innovative solutions.

As illustrated in Figure 4, the integration of LLMs in programming tasks can be categorized into three main frameworks: code generation, test case generation, and multi-agent systems. Each category highlights specific frameworks or methods that exemplify advancements in software development, automation, and efficiency.

LLMs facilitate task decomposition in programming environments, allowing tasks to be treated as entire programs or broken into subtasks. Frameworks like PairCoder, which combines high-level planning with specific implementation through a Navigator and Driver agent, exemplify this flexibility [26]. Such frameworks enhance collaborative programming, boosting the capabilities of developers and artists.

Frameworks like TestChain, employing Designer and Calculator agents for test case generation, enhance the reliability and functionality of LLM-generated code [68]. LEMAS further integrates multiple agents to enhance LLMs, enabling data retrieval, task planning, and solution evaluation from natural language inputs [69].

In low-resource programming languages (VLPLs), the SPEAC method creates an intermediate language aligning with LLMs' expectations, facilitating integration into programming tasks [70]. This enhances LLM applicability across diverse contexts, overcoming resource constraints.

BudgetMLAgent demonstrates the potential of multi-agent systems in automating machine learning tasks with cost-effective LLMs [25]. This highlights the cost-effectiveness and efficiency of LLM integration, making advanced AI capabilities more accessible.

Overall, integrating LLMs into programming tasks marks a paradigm shift in software development, characterized by enhanced efficiency, productivity, and innovation. As LLMs evolve, their integration with evolutionary algorithms and automated machine learning is expected to significantly enhance programming process automation and optimization, driving advancements in code generation, software engineering, and business process management [71, 72, 4].

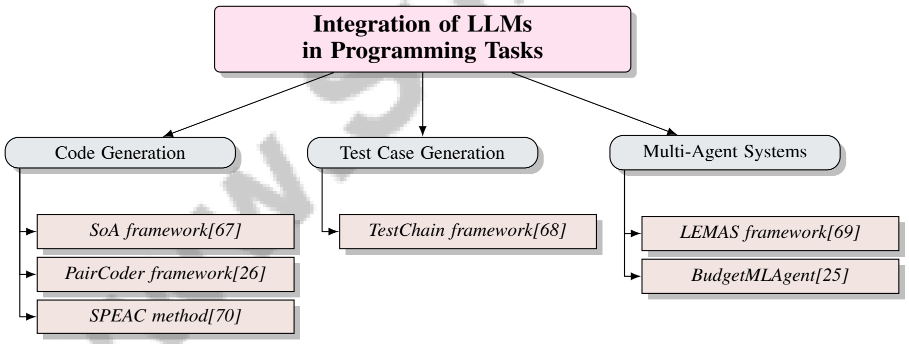


Figure 4: This figure illustrates the integration of Large Language Models (LLMs) in programming tasks, categorizing the frameworks and methods into code generation, test case generation, and multi-agent systems. Each category highlights specific frameworks or methods that exemplify advancements in software development, automation, and efficiency.

4.2 Enhancements in Code Generation and Quality

Significant enhancements in code generation and quality assurance through Large Language Models (LLMs) are driven by innovative methodologies leveraging their capabilities. Frameworks that allow LLMs to self-evaluate and repair code, using feedback from testing and static analysis, greatly improve code quality by autonomously identifying and rectifying errors [73]. These self-evaluation processes are vital for maintaining high code quality standards.

Concept-guided LLM agents further improve code quality by enhancing accuracy in safety analysis and maintaining logical coherence [74]. This ensures adherence to safety standards and logical structures, crucial in precision-demanding domains.

Automation of Infrastructure as Code (IaC) generation has improved efficiency and reduced errors, facilitating rapid deployments [75]. This minimizes manual intervention, reducing human error potential and accelerating deployment timelines.

The PLANSEARCH framework exemplifies enhancements by generating diverse outputs to increase the likelihood of correct solutions [63]. This enhances the robustness and reliability of generated code, ensuring optimal performance.

LLMs' potential to transform telecom operations through automation and improved decision-making underscores their versatility across industries [76]. These advancements represent a paradigm shift in software development practices, enhancing code generation efficiency and reliability across programming environments. Empirical studies show LLMs assist in early software development stages, improving productivity and fostering human-AI collaboration in academic settings. Challenges remain regarding output integrity, a critical area for further exploration [40, 39].

4.3 AI-Driven Productivity and Workflow Optimization

AI-driven tools, particularly LLMs, have markedly optimized productivity and workflow in software development, revolutionizing traditional paradigms. These models automate repetitive tasks, freeing developers for more complex and creative design and implementation aspects [24]. LLM deployment in development environments improves efficiency by streamlining code generation, debugging, and documentation processes [7].

Key advancements include LLMs automating Infrastructure as Code (IaC) generation, accelerating software system deployment and reducing human error [75]. LLMs enhance workflows with intelligent code suggestions and real-time feedback, improving code quality and reducing manual review times [73].

AI-driven tools optimize project management workflows, automating task allocation and progress tracking to ensure efficient resource use and project timeline adherence [76]. LLMs facilitate diverse language and framework integration, enabling seamless team collaboration and productivity [70].

Methodologies like PLANSEARCH exemplify LLM workflow optimization by generating diverse programming solutions, increasing optimal solution identification likelihood [63]. This enhances software system robustness and fosters innovation through multiple development pathways exploration.

4.4 Agent Systems and Multi-Agent Frameworks

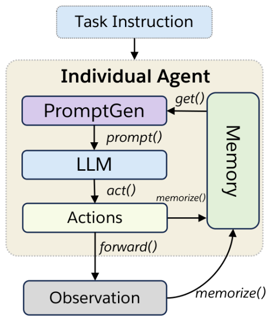
Agent systems and multi-agent frameworks enhance software development automation by leveraging collaborative capabilities of multiple autonomous agents. These systems perform complex tasks through coordinated efforts, with each agent functioning independently while contributing to overall objectives. Integrating LLMs within these frameworks amplifies their potential, automating intricate programming tasks and optimizing workflows [77].

The conceptualization of LLM agents as graph nodes, with specific functions and communication pathways, exemplifies sophisticated agent interaction orchestration [77]. This facilitates efficient information exchange and task allocation, ensuring seamless and effective software development processes.

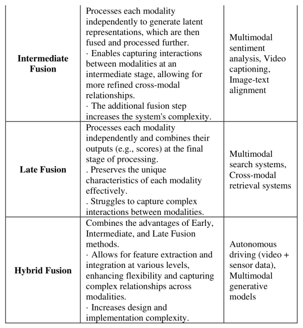
Multi-agent systems (MAS) are beneficial in integrating diverse programming skills and knowledge domains. By distributing tasks among specialized agents, MAS effectively address intricate software development challenges, leveraging LLMs to automate complex processes, enhance collaboration, and improve task efficiency [53, 78, 79]. This decentralized architecture enhances system robustness, scalability, and resilience in dynamic, resource-constrained environments.

Agent systems enhance collaborative capabilities among intelligent components, offering a versatile framework for adaptive strategies in software development. They automate requirements elicitation, improve user story quality, and address communication complexities and early-stage uncertainties through LLM integration [56, 80, 79]. Agents' ability to learn from interactions and adapt behavior based on feedback is crucial for optimizing performance and achieving desired outcomes. LLM integration enhances this adaptability, providing advanced natural language processing capabilities for high accuracy and contextual relevance in human language interpretation and response.

Overall, agent systems and multi-agent frameworks represent transformative advancements in software development automation. By integrating autonomous agents and LLM capabilities, these systems enhance efficiency, accuracy, and scalability of development processes, improve user story quality in agile teams, automate knowledge discovery, and facilitate multi-agent collaboration for complex tasks. For instance, a reference model for LLM-based agents at the Austrian Post Group improved user story quality, while a dual-agent system enhanced knowledge extraction from extensive research articles. A collaborative LLM framework effectively addresses challenges in various domains, leveraging multiple intelligent agents' strengths [56, 81, 80]. As research evolves, the potential for further LLM-driven software development advancements is immense, promising to revolutionize future software design, development, and deployment.

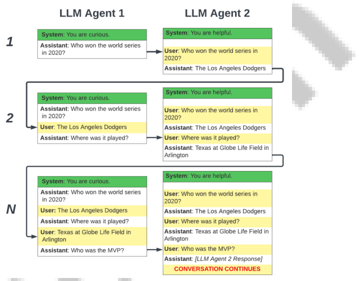


(a) A flowchart illustrating the interaction between an individual agent and its memory[51]



Intermediate Fusion	Processes each modality independently to generate latent representations, which are then fused and processed further. · Enables capturing interactions between modalities at an intermediate stage, allowing for more refined cross-modal relationships. · The additional fusion step increases the system's complexity.	Multimodal sentiment analysis, Video captioning, Image-text alignment
Late Fusion	Processes each modality independently and combines their outputs (e.g., scores) at the final stage of processing. . Preserves the unique characteristics of each modality effectively. . Struggles to capture complex interactions between modalities.	Multimodal search systems, Cross-modal retrieval systems
Hybrid Fusion	Combines the advantages of Early, Intermediate, and Late Fusion methods. · Allows for feature extraction and integration at various levels, enhancing flexibility and capturing complex relationships across modalities. · Increases design and implementation complexity.	Autonomous driving (video + sensor data), Multimodal generative models

(b) Types of Fusion Methods in Multimodal Systems[53]



1

LLM Agent 1:
System: You are curious.
Assistant: Who won the world series in 2020?

LLM Agent 2:
System: You are helpful.
User: Who won the world series in 2020?
Assistant: The Los Angeles Dodgers

2

LLM Agent 1:
System: You are curious.
Assistant: Who won the world series in 2020?
User: The Los Angeles Dodgers
Assistant: Where was it played?

LLM Agent 2:
System: You are helpful.
User: Who won the world series in 2020?
Assistant: The Los Angeles Dodgers
User: Where was it played?
Assistant: Texas at Globe Life Field in Arlington

N

LLM Agent 1:
System: You are curious.
Assistant: Who won the world series in 2020?
User: The Los Angeles Dodgers
Assistant: Where was it played?
User: Texas at Globe Life Field in Arlington
Assistant: Who was the MVP?

LLM Agent 2:
System: You are helpful.
User: Who won the world series in 2020?
Assistant: The Los Angeles Dodgers
User: Where was it played?
Assistant: Texas at Globe Life Field in Arlington
User: Who was the MVP?
Assistant: *[LLM Agent 2 Response]*
CONVERSATION CONTINUES

(c) The image shows a conversation between two LLM (Large Language Model) agents, LLM Agent 1 and LLM Agent 2, discussing the World Series in 2020.[82]

Figure 5: Examples of Agent Systems and Multi-Agent Frameworks

In AI-assisted programming and software development automation, agent systems and multi-agent frameworks are pivotal, driving innovation and efficiency. The examples in Figure 5 showcase these systems' diverse applications and interactions. The flowchart illustrates an agent's interaction with its memory, highlighting components like PromptGen in autonomous decision-making and task execution. The comparative table of fusion methods in multimodal systems underscores framework versatility and adaptability, detailing fusion strategies for varied data modalities and enhanced performance. The conversation between two LLM agents exemplifies AI systems' potential for natural language dialogue, reflecting on historical events like the 2020 World Series. These examples demonstrate agent systems and multi-agent frameworks' transformative impact on modern software development, streamlining processes, enhancing interoperability, and fostering intelligent interactions [51, 53, 82].

5 Challenges and Opportunities

Category	Feature	Method
Opportunities for Improving Model Efficiency	Multi-Task Approaches	CPL[59]
Expanding LLM Applicability	Dynamic Adjustments Automated Adaptation Collaborative Frameworks	MCC[83], DPPA[84] PP[85], ILF[86], IPA[87] LLM-Duo[81], CTS[88], LAI[82]

Table 1: This table provides a comprehensive summary of methods aimed at enhancing the efficiency and applicability of Large Language Models (LLMs) in software development. It categorizes the methods into opportunities for improving model efficiency and expanding LLM applicability, highlighting key features and corresponding methodologies. The table is based on recent studies that explore innovative approaches to optimize LLM performance and broaden their application scope in diverse programming environments.

The integration of Large Language Models (LLMs) into software development presents both challenges and opportunities, particularly in automated program repair and code generation. Key areas of

AI-generated, for reference only.

focus include optimizing prompt engineering techniques to enhance model performance, reducing manual debugging efforts, and deriving insights from contextualized AI coding assistants. Table 1 presents a detailed summary of recent methodologies designed to improve the efficiency and expand the applicability of Large Language Models (LLMs) in software development. Addressing these facets provides a framework for leveraging AI in software development while considering potential drawbacks and security concerns [35, 89, 32, 90, 2]. This exploration aids in understanding LLM implementation complexities, identifying obstacles, and uncovering innovation avenues.

5.1 Challenges and Opportunities in AI-Assisted Programming

Integrating LLMs into AI-assisted programming introduces a complex landscape of challenges and opportunities. A significant challenge is 'hallucination,' where LLMs produce plausible but incorrect outputs, affecting code reliability and security. Existing benchmarks often emphasize model ranking without adequately measuring trade-offs between response quality and inference costs, complicating the assessment of LLMs' practical utility [61].

Another challenge is the bias towards high-resource languages, limiting LLM effectiveness in low-resource contexts. This bias, compounded by tokenization complexities for morphologically rich languages, poses substantial obstacles. Fine-tuning methods often perform poorly with limited data, degrading model performance. Challenges in implementing essential security measures in smart contract generation highlight significant limitations in LLM capabilities. While LLMs can create common contract types, they struggle to incorporate rigorous security details, raising concerns about vulnerabilities and the trustworthiness of generated code [91, 41, 13, 92, 73].

Despite these challenges, opportunities exist to enhance AI-assisted programming through innovative approaches. Optimizing prompts for complex reasoning tasks is crucial for improving LLM adaptability, leading to significant performance enhancements across programming environments. This approach empowers LLMs to tackle intricate problems and allows for systematic categorization of prompt engineering strategies tailored to educational needs. Studies show specific prompt strategies, such as the "multi-step" approach, can enhance LLM performance, especially in educational contexts like programming [93, 42]. Test-driven methodologies also present opportunities to improve LLM-generated code accuracy and reliability, ensuring outputs meet specified requirements.

The SPEAC method highlights challenges in generating syntactically valid code for very low-level programming languages (VLPLs), often underrepresented in pre-training datasets. It identifies key areas for future research, such as applying SPEAC to additional VLPLs and enhancing automated repair techniques. This method uses an intermediate language aligning with LLM capabilities, facilitating automatic compilation of generated code into target VLPLs. Empirical evaluations with the UCLID5 formal verification language show SPEAC increases syntactically correct code output frequency while maintaining semantic accuracy [70, 49]. Additionally, reliance on text rather than direct file generation in Business Process Management (BPM) tasks presents challenges requiring further exploration.

LLMs enhance automation and productivity in software development—particularly in code generation, user story improvement, and debugging—but challenges persist, including generalization, data privacy, bias, and security issues. These challenges necessitate focused research efforts [39, 21, 56, 41, 10]. Addressing these challenges while exploiting LLMs' potential to enhance instruction adherence and reduce resource consumption can drive advancements in software development, improving programming tools' quality and accessibility.

5.2 Opportunities for Improving Model Efficiency

Enhancing LLM efficiency in code generation is crucial, with promising strategies emerging to optimize performance across diverse programming environments. Refining prompt engineering techniques to tackle complex programming challenges can improve educational outcomes and model performance. This includes innovative prompting strategies and assessing the usability of LLM-generated code, particularly the security aspects of smart contracts [60].

Developing hierarchical decomposition methods and exploring synergies between prompt engineering stages represent further opportunities to enhance LLM performance [59]. Expanding benchmarks to include more programming languages and recent LLMs can aid in estimating task complexity and

Benchmark	Size	Domain	Task Format	Metric
LICOEVAL[60]	4,187	Software Engineering	Code Generation	LICO
EXACT[94]	220	Software Security	Code Evaluation	Security Issues, Code Quality
DPE[95]	48	Software Engineering	Design Pattern Classification	Accuracy, CS
MT-LLM[96]	1,000	Literary Translation	Translation	BLEU, chrF++
LCLM[97]	60	Python Programming	Code Generation	Pass@1
LLM-Enhance[11]	112,709	Fault Localization	Fault Localization	F1-score, MAP@R
LLM-Robustness[98]	40	Programming	Code Generation	Solved Rate, Average Solved Rate
MATH[99]	12,500	Mathematics	Formalization	BLEU

Table 2: This table presents a comprehensive overview of various benchmarks utilized to evaluate large language models (LLMs) in different domains. It includes details on benchmark size, domain, task format, and evaluation metrics, providing a broad perspective on the diverse applications and assessment criteria for LLMs in fields such as software engineering, literary translation, and mathematics.

model capabilities, guiding targeted improvements [60]. Table 2 provides a detailed overview of representative benchmarks that are instrumental in assessing the efficiency and performance of large language models across various domains.

Improving tokenization strategies and reducing computational costs are critical for LLM efficiency, especially in multilingual contexts [8]. Exploring alternative agent interaction modes and developing faster inference methods for real-time applications can improve model efficiency [8]. Integrating LLMs with traditional testing methodologies, particularly in early testing phases, is crucial for enhancing prompt engineering techniques [9].

Addressing hallucinations in LLM outputs, which could lead to incorrect decisions, underscores the importance of robust monitoring and evaluation frameworks [9]. Future research should focus on improving the MergeBack procedure and refining the code reduction algorithm to enhance bug-fixing effectiveness and reliability.

Sample-efficient methods like SEED offer avenues for adapting LLMs with fewer training samples, improving performance while minimizing resource consumption. Investigating the scalability of test-driven development (TDD) across programming environments can enhance test case quality and facilitate robust code generation by leveraging LLM capabilities. Studies indicate integrating TDD principles into AI-assisted code generation can improve code accuracy and correctness, streamline automated unit test generation, and address code reliability and quality assurance challenges [68, 57, 100, 73].

5.3 Expanding LLM Applicability

Method Name	Scalability and Adaptability	Integration and Automation	Diversity and Robustness
PP[85]	Diverse Educational Contexts	Minimal Human Intervention	Broader Applicability
CTS[88]	Complex And Diverse	Minimal Human Intervention	Wider Range Languages
MCC[83]	Varying Sizes	Self-testing Mechanism	Various Model Families
ILF[86]	Model's Adaptability	Automating The Generation	Wider Range Scenarios
LLM-Duo[81]	Other Domains	Dual-agent System	Additional Llm Capabilities
IPA[87]	Complex And Diverse	Automated Multi-agent	Various Llms
LAI[82]	Complex Programming Tasks	Minimal Human Intervention	Wider Range Languages
DPPA[84]	Adaptive Processing	Minimal Human Intervention	Diverse Datasets

Table 3: Comparison of various methods for expanding the applicability of large language models (LLMs) in software development. The table highlights their scalability, integration, and robustness across diverse programming contexts, emphasizing minimal human intervention and adaptability.

Expanding LLM applicability across diverse programming environments is crucial for realizing their full potential in software development. Future research should prioritize exploring LLM scalability in handling complex creative programming tasks, broadening their scope in various contexts [85]. This involves refining outline generation techniques and exploring additional applications of natural language outlines in software development scenarios to enhance LLM adaptability [62]. Table 3 presents a comparative analysis of different methodologies aimed at enhancing the applicability of large language models (LLMs) across various programming environments, focusing on scalability, integration, and robustness.

To improve decision-making processes, particularly in determining when to use code or text, developing sophisticated multi-agent frameworks is necessary [88]. These frameworks could facilitate better LLM integration into programming tasks, allowing more dynamic and contextually relevant outputs. Exploring dynamic threshold adjustments within model cascading approaches could improve performance across programming languages, extending beyond prevalent languages like Python [83].

Automating prompt optimization frameworks is a promising avenue for expanding LLM applicability. Extending metrics to broader applications beyond classification can enhance LLM adaptability and efficiency in diverse environments [47]. This approach complements the need to explore additional prompting strategies and expand datasets to include more diverse codebases, improving LLM robustness and versatility [101].

Future research could explore automating natural language feedback generation, enhancing model adaptability in code generation tasks [86]. This could lead to more user-friendly interactions and improve LLM effectiveness in software development. Integrating additional LLM capabilities, such as those in ontology design processes, could further enhance LLM performance in knowledge discovery tasks [81].

Research should also explore fully automated multi-agent interactions to minimize human overhead while maintaining high adaptation performance, expanding LLM applicability [87]. Expanding datasets to include more languages and exploring advanced training methods to mitigate multilingual bias are crucial steps in broadening LLM reach [102]. Opportunities for expanding LLM applicability in simulation scenarios, such as negotiations and collaborative problem-solving, should be explored [82].

Expanding benchmarks to include more programming languages and error types, and enhancing code review comment quality, can significantly improve LLM robustness in diverse environments. Research could focus on enhancing algorithm robustness in unpredictable environments and exploring applications across domains like real-time data analytics [84].

6 Conclusion

Large Language Models (LLMs) have demonstrated a profound impact on the landscape of software development, particularly in automating code generation and streamlining complex coding processes. Their integration into development workflows significantly enhances efficiency and reduces cognitive load for developers. The deployment of LLMs, especially when combined with multi-agent systems like the Self-Organized multi-Agent framework (SoA), exemplifies their capability to optimize large-scale programming tasks through collaborative frameworks. Despite these advancements, challenges remain, particularly in low-resource settings where computational limitations impede LLM accessibility and performance. Techniques such as Parameter-Efficient Fine-Tuning (PEFT) are pivotal in addressing these constraints, ensuring that LLMs remain effective even in resource-constrained environments. Furthermore, approaches like the Multi-Programming Language Ensemble (MPLE) continue to set benchmarks in improving LLM performance across diverse coding contexts.

The survey underscores the critical role of structured prompting in enhancing LLM interactions, which involves refining prompt designs to yield more accurate and contextually appropriate outputs. This practice, alongside the strategic use of code reviews, enhances the capabilities of automated program repair models, thereby improving overall software quality. Future research directions should focus on increasing the interpretability of AI-driven solutions, developing new benchmarks, and addressing the ethical considerations associated with AI in software development. The integration of security knowledge into the code generation process is also essential, as highlighted by benchmarks like LICOEVAL, which reveal gaps in LLMs' ability to provide precise license information. As LLMs continue to evolve, they are poised to further transform software development, making it more efficient and accessible while aligning with the growing demands of the global landscape. Continued innovation and research are crucial to fully leverage the potential of LLMs across various domains, ensuring their sustained prominence in the realm of software development automation.

References

[1] Mounira Nihad Zitouni, Amal Ahmed Anda, Sahil Rajpal, Daniel Amyot, and John Mylopoulos. Towards the llm-based generation of formal specifications from natural-language contracts: Early experiments with symboleo, 2024.

[2] Avinash Anand, Akshit Gupta, Nishchay Yadav, and Shaurya Bajaj. A comprehensive survey of ai-driven advancements and techniques in automated program repair and code generation, 2024.

[3] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. Software testing with large language models: Survey, landscape, and vision, 2024.

[4] Michael Grohs, Luka Abb, Nourhan Elsayed, and Jana-Rebecca Rehse. Large language models can accomplish business process management tasks, 2023.

[5] Kristen Howell, Gwen Christian, Pavel Fomitchov, Gitit Kehat, Julianne Marzulla, Leanne Rolston, Jadin Tredup, Ilana Zimmerman, Ethan Selfridge, and Joseph Bradley. The economic trade-offs of large language models: A case study, 2023.

[6] Danshi Wang, Yidi Wang, Xiaotian Jiang, Yao Zhang, Yue Pang, and Min Zhang. When large language models meet optical networks: Paving the way for automation, 2024.

[7] Paheli Bhattacharya, Manojit Chakraborty, Kartheek N S N Palepu, Vikas Pandey, Ishan Dindorkar, Rakesh Rajpurohit, and Rishabh Gupta. Exploring large language models for code explanation, 2023.

[8] Humza Naveed, Asad Ullah Khan, Shi Qiu, Muhammad Saqib, Saeed Anwar, Muhammad Usman, Naveed Akhtar, Nick Barnes, and Ajmal Mian. A comprehensive overview of large language models. *arXiv preprint arXiv:2307.06435*, 2023.

[9] Sen Huang, Kaixiang Yang, Sheng Qi, and Rui Wang. When large language model meets optimization, 2024.

[10] Sajed Jalil. The transformative influence of large language models on software development, 2023.

[11] Xin Yin, Chao Ni, Xiaodan Xu, Xinrui Li, and Xiaohu Yang. Improving the ability of pre-trained language model by imparting large language model's experience, 2025.

[12] Toufique Ahmed, Kunal Suresh Pai, Premkumar Devanbu, and Earl T. Barr. Automatic semantic augmentation of language model prompts (for code summarization), 2024.

[13] Ahmad Mohsin, Helge Janicke, Adrian Wood, Iqbal H. Sarker, Leandros Maglaras, and Naeem Janjua. Can we trust large language models generated code? a framework for in-context learning, security patterns, and code evaluations across diverse llms, 2024.

[14] Tengfei Xue, Xuefeng Li, Tahir Azim, Roman Smirnov, Jianhui Yu, Arash Sadrieh, and Babak Pahlavan. Multi-programming language ensemble for code generation in large language model, 2024.

[15] Ali Reza Ibrahimzada. Program decomposition and translation with static analysis, 2024.

[16] Weizhe Chen, Sven Koenig, and Bistra Dilkina. Reprompt: Planning by automatic prompt engineering for large language models agents, 2024.

[17] Zhenlan Ji, Pingchuan Ma, Zongjie Li, and Shuai Wang. Benchmarking and explaining large language model-based code generation: A causality-centric approach, 2023.

[18] Yi Cui. Insights from benchmarking frontier language models on web app code generation, 2024.

[19] Weiqing Yang, Hanbin Wang, Zhenghao Liu, Xinze Li, Yukun Yan, Shuo Wang, Yu Gu, Minghe Yu, Zhiyuan Liu, and Ge Yu. Coast: Enhancing the code debugging ability of llms through communicative agent based data synthesis, 2025.

[20] Cuiyun Gao, Xing Hu, Shan Gao, Xin Xia, and Zhi Jin. The current challenges of software engineering in the era of large language models, 2024.

[21] Guoqing Wang, Zeyu Sun, Zhihao Gong, Sixiang Ye, Yizhou Chen, Yifan Zhao, Qingyuan Liang, and Dan Hao. Do advanced language models eliminate the need for prompt engineering in software engineering?, 2024.

[22] Andreas Vogelsang. From specifications to prompts: On the future of generative llms in requirements engineering, 2024.

[23] Dae-Kyoo Kim. Prompted software engineering in the era of ai models, 2023.

[24] Patara Trirat, Wonyong Jeong, and Sung Ju Hwang. Automl-agent: A multi-agent llm framework for full-pipeline automl, 2024.

[25] Shubham Gandhi, Manasi Patwardhan, Lovekesh Vig, and Gautam Shroff. Budgetmlagent: A cost-effective llm multi-agent system for automating machine learning tasks, 2025.

[26] Huan Zhang, Wei Cheng, Yuhan Wu, and Wei Hu. A pair programming framework for code generation via multi-plan exploration and feedback-driven refinement, 2024.

[27] Chao Lei, Yanchuan Chang, Nir Lipovetzky, and Krista A. Ehinger. Planning-driven programming: A large language model programming workflow, 2025.

[28] Heiko Koziolek and Anne Koziolek. Llm-based control code generation using image recognition, 2023.

[29] Kunhao Zheng, Juliette Decugis, Jonas Gehring, Taco Cohen, Benjamin Negrevergne, and Gabriel Synnaeve. What makes large language models reason in (multi-turn) code generation?, 2024.

[30] Tong Ye, Tengfei Ma, Xuhong Zhang, Hang Yu, Jianwei Yin, and Wenhai Wang. A problem-oriented perspective and anchor verification for code optimization, 2025.

[31] Emanuele Musumeci, Michele Brienza, Vincenzo Suriani, Daniele Nardi, and Domenico Daniele Bloisi. Llm based multi-agent generation of semi-structured documents from semantic templates in the public administration domain, 2024.

[32] Steven P. Reiss. Assisted specification of code using search, 2022.

[33] Sean Williams and James Huckle. Easy problems that llms get wrong, 2024.

[34] Anupam Garg, Aryaman Raina, Aryan Gupta, Jaskaran Singh, Manav Saini, Prachi Iiitd, Ronit Mehta, Rupin Oberoi, Sachin Sharma, Samyak Jain, Sarthak Tyagi, Utkarsh Arora, and Dhruv Kumar. Analyzing llm usage in an advanced computing class in india, 2024.

[35] Man Fai Wong, Shangxin Guo, Ching Nam Hang, Siu Wai Ho, and Chee Wei Tan. Natural language generation and understanding of big code for ai-assisted programming: A review, 2023.

[36] Zibin Zheng, Kaiwen Ning, Yanlin Wang, Jingwen Zhang, Dewu Zheng, Mingxi Ye, and Jiachi Chen. A survey of large language models for code: Evolution, benchmarking, and future trends, 2024.

[37] Baolin Li, Yankai Jiang, Vijay Gadepally, and Devesh Tiwari. Llm inference serving: Survey of recent advances and opportunities, 2024.

[38] Douglas Schonholtz. A review of repository level prompting for llms, 2023.

[39] Sanka Rasnayaka, Guanlin Wang, Ridwan Shariffdeen, and Ganesh Neelakanta Iyer. An empirical study on usage and perceptions of llms in a software engineering project, 2024.

[40] Mohamed Nejjar, Luca Zacharias, Fabian Stiehle, and Ingo Weber. Llms for science: Usage for code generation and data analysis, 2024.

[41] Ming Cheung. A reality check of the benefits of llm in business, 2024.

[42] Tianyu Wang, Nianjun Zhou, and Zhixiong Chen. Enhancing computer programming education with llms: A study on effective prompt engineering for python code generation, 2024.

[43] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C. Schmidt. A prompt pattern catalog to enhance prompt engineering with chatgpt, 2023.

[44] Yuanfeng Song, Yuanqin He, Xuefang Zhao, Hanlin Gu, Di Jiang, Haijun Yang, Lixin Fan, and Qiang Yang. A communication theory perspective on prompting engineering methods for large language models, 2023.

[45] Shengtao He. Achieving tool calling functionality in llms using only prompt engineering without fine-tuning, 2024.

[46] Haochen Li, Jonathan Leung, and Zhiqi Shen. Towards goal-oriented prompt engineering for large language models: A survey, 2024.

[47] Federico Errica, Giuseppe Siracusano, Davide Sanvito, and Roberto Bifulco. What did i do wrong? quantifying llms' sensitivity and consistency to prompt engineering, 2025.

[48] Yeseung Kim, Dohyun Kim, Jieun Choi, Jisang Park, Nayoung Oh, and Daehyung Park. A survey on integration of large language models with intelligent robots, 2024.

[49] Liguo Chen, Qi Guo, Hongrui Jia, Zhengran Zeng, Xin Wang, Yijiang Xu, Jian Wu, Yidong Wang, Qing Gao, Jindong Wang, Wei Ye, and Shikun Zhang. A survey on evaluating large language models in code generation tasks, 2024.

[50] Sivan Schwartz, Avi Yaeli, and Segev Shlomov. Enhancing trust in llm-based ai automation agents: New considerations and future challenges, 2023.

[51] Zhiwei Liu, Weiran Yao, Jianguo Zhang, Liangwei Yang, Zuxin Liu, Juntao Tan, Prafulla K. Choubey, Tian Lan, Jason Wu, Huan Wang, Shelby Heinecke, Caiming Xiong, and Silvio Savarese. Agentlite: A lightweight library for building and advancing task-oriented llm agent system, 2024.

[52] Hans-Alexander Kruse, Tim Puhlfürß, and Walid Maalej. Can developers prompt? a controlled experiment for code documentation generation, 2024.

[53] Cheonsu Jeong. Beyond text: Implementing multimodal large language model-powered multi-agent systems using a no-code platform, 2025.

[54] Yingxuan Yang, Qiuying Peng, Jun Wang, Ying Wen, and Weinan Zhang. Llm-based multi-agent systems: Techniques and business perspectives, 2024.

[55] Junhua Liu and Bin Fu. Responsible multilingual large language models: A survey of development, applications, and societal impact, 2024.

[56] Zheying Zhang, Maruf Rayhan, Tomas Herda, Manuel Goisauf, and Pekka Abrahamsson. Llm-based agents for automating the enhancement of user story quality: An early report, 2024.

[57] Noble Saji Mathews and Meiyappan Nagappan. Test-driven development for code generation, 2024.

[58] Zhiqiang Yuan, Weitong Chen, Hanlin Wang, Kai Yu, Xin Peng, and Yiling Lou. Transagent: An llm-based multi-agent system for code translation, 2024.

[59] Zhihong Sun, Chen Lyu, Bolun Li, Yao Wan, Hongyu Zhang, Ge Li, and Zhi Jin. Enhancing code generation performance of smaller models by distilling the reasoning ability of llms, 2024.

[60] Weiwei Xu, Kai Gao, Hao He, and Minghui Zhou. Licoeval: Evaluating llms on license compliance in code generation, 2025.

[61] Dewu Zheng, Yanlin Wang, Ensheng Shi, Ruikai Zhang, Yuchi Ma, Hongyu Zhang, and Zibin Zheng. Towards more realistic evaluation of llm-based code generation: an experimental study and beyond, 2024.

[62] Kensen Shi, Deniz Altınbüken, Saswat Anand, Mihai Christodorescu, Katja Grünwedel, Alexa Koenings, Sai Naidu, Anurag Pathak, Marc Rasi, Fredde Ribeiro, Brandon Ruffin, Siddhant Sanyam, Maxim Tabachnyk, Sara Toth, Roy Tu, Tobias Welp, Pengcheng Yin, Manzil Zaheer, Satish Chandra, and Charles Sutton. Natural language outlines for code: Literate programming in the llm era, 2024.

[63] Evan Wang, Federico Cassano, Catherine Wu, Yunfeng Bai, Will Song, Vaskar Nath, Ziwen Han, Sean Hendryx, Summer Yue, and Hugh Zhang. Planning in natural language improves llm search for code generation, 2024.

[64] Ziyao Zhang, Yanlin Wang, Chong Wang, Jiachi Chen, and Zibin Zheng. Llm hallucinations in practical code generation: Phenomena, mechanism, and mitigation, 2024.

[65] Zhijie Wang, Zijie Zhou, Da Song, Yuheng Huang, Shengmai Chen, Lei Ma, and Tianyi Zhang. Towards understanding the characteristics of code generation errors made by large language models, 2025.

[66] Jierui Li, Szymon Tworkowski, Yingying Wu, and Raymond Mooney. Explaining competitive-level programming solutions using llms, 2023.

[67] Yoichi Ishibashi and Yoshimasa Nishimura. Self-organized agents: A llm multi-agent framework toward ultra large-scale code generation and optimization, 2024.

[68] Kefan Li and Yuan Yuan. Large language models as test case generators: Performance evaluation and enhancement, 2024.

[69] Feibo Jiang, Li Dong, Yubo Peng, Kezhi Wang, Kun Yang, Cunhua Pan, Dusit Niyato, and Octavia A. Dobre. Large language model enhanced multi-agent systems for 6g communications, 2023.

[70] Federico Mora, Justin Wong, Haley Lepe, Sahil Bhatia, Karim Elmaaroufi, George Varghese, Joseph E. Gonzalez, Elizabeth Polgreen, and Sanjit A. Seshia. Synthetic programming elicitation for text-to-code in very low-resource programming and formal languages, 2024.

[71] Jinglue Xu, Jialong Li, Zhen Liu, Nagar Anthel Venkatesh Suryanarayanan, Guoyuan Zhou, Jia Guo, Hitoshi Iba, and Kenji Tei. Large language models synergize with automated machine learning. *arXiv preprint arXiv:2405.03727*, 2024.

[72] Xingyu Wu, Sheng hao Wu, Jibin Wu, Liang Feng, and Kay Chen Tan. Evolutionary computation in the era of large language model: Survey and roadmap, 2024.

[73] Greta Dolcetti, Vincenzo Arceri, Eleonora Iotti, Sergio Maffeis, Agostino Cortesi, and Enea Zaffanella. Helping llms improve code generation using feedback from testing and static analysis, 2025.

[74] Florian Geissler, Karsten Roscher, and Mario Trapp. Concept-guided llm agents for human-ai safety codesign, 2024.

[75] Kalahasti Ganesh Srivatsa, Sabyasachi Mukhopadhyay, Ganesh Katrapati, and Manish Shrivastava. A survey of using large language models for generating infrastructure as code, 2024.

[76] Hao Zhou, Chengming Hu, Ye Yuan, Yufei Cui, Yili Jin, Can Chen, Haolun Wu, Dun Yuan, Li Jiang, Di Wu, Xue Liu, Charlie Zhang, Xianbin Wang, and Jiangchuan Liu. Large language model (llm) for telecommunications: A comprehensive survey on principles, key techniques, and opportunities, 2024.

[77] Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. Language agents as optimizable graphs, 2024.

[78] Jinyang Li, Jack Williams, Nick McKenna, Arian Askari, Nicholas Wilson, and Reynold Cheng. Agents help agents: Exploring training-free knowledge distillation for small language models in data science code generation.

[79] Malik Abdul Sami, Muhammad Waseem, Zheying Zhang, Zeeshan Rasheed, Kari Systä, and Pekka Abrahamsson. Ai based multiagent approach for requirements elicitation and analysis, 2024.

[80] Yashar Talebirad and Amirhossein Nadiri. Multi-agent collaboration: Harnessing the power of intelligent llm agents, 2023.

[81] Yuting Hu, Dancheng Liu, Qingyun Wang, Charles Yu, Heng Ji, and Jinjun Xiong. Automating knowledge discovery from scientific literature via llms: A dual-agent approach with progressive ontology prompting, 2024.

[82] Edward Junprung. Exploring the intersection of large language models and agent-based modeling via prompt engineering, 2023.

[83] Boyuan Chen, Mingzhi Zhu, Brendan Dolan-Gavitt, Muhammad Shafique, and Siddharth Garg. Model cascading for code: Reducing inference costs with model cascading for llm based code generation, 2024.

[84] Biyang Guo, He Wang, Wenyilin Xiao, Hong Chen, Zhuxin Lee, Songqiao Han, and Hailiang Huang. Sample design engineering: An empirical study of what makes good downstream fine-tuning samples for llms, 2024.

[85] James Prather, Paul Denny, Juho Leinonen, David H. Smith IV au2, Brent N. Reeves, Stephen MacNeil, Brett A. Becker, Andrew Luxton-Reilly, Thezyrie Amarouche, and Bailey Kimmel. Interactions with prompt problems: A new way to teach programming with large language models, 2024.

[86] Angelica Chen, Jérémy Scheurer, Tomasz Korbak, Jon Ander Campos, Jun Shern Chan, Samuel R. Bowman, Kyunghyun Cho, and Ethan Perez. Improving code generation by training with natural language feedback, 2024.

[87] Tanghaoran Zhang, Yue Yu, Xinjun Mao, Shangwen Wang, Kang Yang, Yao Lu, Zhang Zhang, and Yuxin Zhao. Instruct or interact? exploring and eliciting llms' capability in code snippet adaptation through prompt engineering, 2024.

[88] Yongchao Chen, Harsh Jhamtani, Srinagesh Sharma, Chuchu Fan, and Chi Wang. Steering large language models between code execution and textual reasoning, 2024.

[89] Banghao Chen, Zhaofeng Zhang, Nicolas Langrené, and Shengxin Zhu. Unleashing the potential of prompt engineering in large language models: a comprehensive review, 2024.

[90] Gustavo Pinto, Cleidson de Souza, João Batista Neto, Alberto de Souza, Tarcísio Gotto, and Edward Monteiro. Lessons from building stackspot ai: A contextualized ai coding assistant, 2024.

[91] Siddhartha Chatterjee and Bina Ramamurthy. Efficacy of various large language models in generating smart contracts, 2024.

[92] Rabimba Karanjai, Lei Xu, and Weidong Shi. Teaching machines to code: Smart contract translation with llms. *arXiv preprint arXiv:2403.09740*, 2024.

[93] Tuo Zhang, Jinyue Yuan, and Salman Avestimehr. Revisiting opro: The limitations of small-scale llms as optimizers, 2024.

[94] Chun Jie Chong, Zhihao Yao, and Iulian Neamtiu. Artificial-intelligence generated code considered harmful: A road map for secure and high-quality code generation, 2024.

[95] Zhenyu Pan, Xuefeng Song, Yunkun Wang, Rongyu Cao, Binhua Li, Yongbin Li, and Han Liu. Do code llms understand design patterns?, 2025.

AI-generated, for reference only.

[96] Nooshin Pourkamali and Shler Ebrahim Sharifi. Machine translation with large language models: Prompt engineering for persian, english, and russian directions, 2024.

[97] Jessica López Espejel, Mahaman Sanoussi Yahaya Alassan, Merieme Bouhandi, Walid Dahhane, and El Hassane Ettifouri. Low-cost language models: Survey and performance evaluation on python code generation, 2024.

[98] Atsushi Shirafuji, Yutaka Watanobe, Takumi Ito, Makoto Morishita, Yuki Nakamura, Yusuke Oda, and Jun Suzuki. Exploring the robustness of large language models for solving programming problems, 2023.

[99] Yuhuai Wu, Albert Q. Jiang, Wenda Li, Markus N. Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. Autoformalization with large language models, 2022.

[100] Wendkûuni C. Ouédraogo, Kader Kaboré, Haoye Tian, Yewei Song, Anil Koyuncu, Jacques Klein, David Lo, and Tegawendé F. Bissyandé. Large-scale, independent and comprehensive study of the power of llms for test case generation, 2024.

[101] Chanathip Pornprasit and Chakkrit Tantithamthavorn. Fine-tuning and prompt engineering for large language models-based code review automation, 2024.

[102] Chaozheng Wang, Zongjie Li, Cuiyun Gao, Wenxuan Wang, Ting Peng, Hailiang Huang, Yuetang Deng, Shuai Wang, and Michael R. Lyu. Exploring multi-lingual bias of large code models in code generation, 2024.