

A Comprehensive Survey on Benchmarks and Solutions in Software Engineering of LLM-Empowered Agentic System

Jiale Guo^{1,*}, Suizhi Huang^{2,*}, Mei Li^{2,4,*}, Dong Huang⁵, Xingsheng Chen⁵,
 Regina Zhang^{6,†}, Zhijiang Guo³, Han Yu^{2,†}, Siu-Ming Yiu^{5,†},
 Pietro Lio⁶, Kwok-Yan Lam^{1,†}

¹Digital Trust Centre, Nanyang Technological University, Singapore

²College of Computing and Data Science, Nanyang Technological University, Singapore

³The Hong Kong University of Science and Technology, Hong Kong

⁴School of Computer Science, Shanghai Jiao Tong University, Shanghai, China

⁵School of Computing and Data Science, The University of Hong Kong, Hong Kong

⁶School of Computer Science and Technology, The University of Cambridge, UK

arXiv:2510.09721v3 [cs.SE] 23 Oct 2025

Abstract—The integration of Large Language Models (LLMs) into software engineering has driven a transition from traditional rule-based systems to autonomous agentic systems capable of solving complex problems. However, systematic progress is hindered by a lack of comprehensive understanding of how benchmarks and solutions interconnect. This survey addresses this gap by providing the first holistic analysis of LLM-powered software engineering, offering insights into evaluation methodologies and solution paradigms. We review over 150 recent papers and propose a taxonomy along two key dimensions: (1) Solutions, categorized into prompt-based, fine-tuning-based, and agent-based paradigms, and (2) Benchmarks, including tasks such as code generation, translation, and repair. Our analysis highlights the evolution from simple prompt engineering to sophisticated agentic systems incorporating capabilities like planning, reasoning, memory mechanisms, and tool augmentation. To contextualize this progress, we present a unified pipeline illustrating the workflow from task specification to deliverables, detailing how different solution paradigms address various complexity levels. Unlike prior surveys that focus narrowly on specific aspects, this work connects 50+ benchmarks to their corresponding solution strategies, enabling researchers to identify optimal approaches for diverse evaluation criteria. We also identify critical research gaps and propose future directions, including multi-agent collaboration, self-evolving systems, and formal verification integration. This survey serves as a foundational guide for advancing LLM-driven software engineering. We maintain a GitHub repository that continuously updates the reviewed and related papers at <https://github.com/lisaGuoJ/LLM-Agent-SE-Survey>.

Index Terms—Software Engineering, Benchmarks, Solutions, LLMs, Agents, Prompts

I. INTRODUCTION

SOFTWARE Engineering aims to systematically develop high-quality, reliable, and maintainable software systems through disciplined methodologies, tools, and best practices [1], [2]. Key objectives include effectiveness, efficiency, maintainability and scalability [3], [4]. And the main tasks

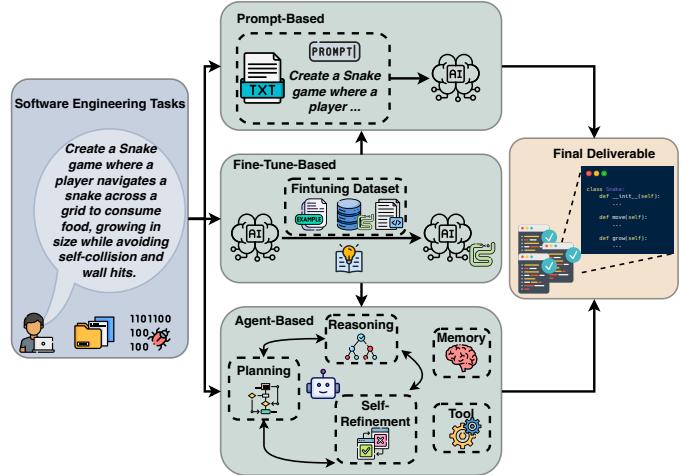


Fig. 1: Illustrations of process of LLM-empowered software engineering

of software engineering are code generation [5], code translation [6] and program repair [7]. Earlier studies for software engineering mainly adopt generating codes via rule-based systems and template-driven approaches, which rely on predefined patterns and heuristics. However, these traditional methods [8] often struggle with complex programming scenarios and lack the flexibility to handle diverse coding styles and requirements. Recent advances [3], [4] have shifted towards learning-based approaches, particularly leveraging deep learning models and large language models to better capture programming patterns and semantics.

The emergence of Large Language Models (LLMs) has fundamentally transformed the landscape of software engineering. As illustrated in Figure 1, modern LLM-empowered software engineering follows a sophisticated multi-stage process that begins with diverse software engineering tasks - from creating simple games to complex enterprise applications. Unlike traditional approaches, this paradigm employs three distinct

* These authors contributed equally to this work

† Corresponding author

TABLE I: Comparison of related surveys on software engineering (PR denotes program repair. CG denotes code generation. SE denotes software engineering.)

Features	Year	Taxonomy	Pipeline	Prompt	Agent	Benchmarks	Solutions	Scope
Zhang et al. [9]	2023	✓	✓	✗	✗	✓	✓	PR
Jiang et al. [10]	2024	✓	✓	✓	✓	✗	✓	CG
Wang et al. [11]	2025	✓	✓	✓	✓	✗	✓	CG
Dong et al. [12]	2025	✓	✓	✓	✓	✗	✓	CG
Sapkota et al. [13]	2025	✗	✗	✓	✓	✗	✓	CG
Ours	2025	✓	✓	✓	✓	✓	✓	PR, CG, SE

methodological frameworks: Prompt-Based approaches that directly query LLMs with carefully crafted instructions, Fine-Tune-Based methods that adapt models to specific domains through supervised learning, and Agent-Based systems that orchestrate multiple components including planning, reasoning, memory, and tool integration. These systems demonstrate remarkable capabilities in understanding natural language specifications and generating contextually appropriate code, ultimately producing final deliverables that range from functional applications to comprehensive software solutions. The integration of these approaches enables LLM-empowered systems to autonomously handle complex programming logic across diverse languages and frameworks, making them increasingly valuable in modern software development workflows where they can operate with minimal human intervention while maintaining high-quality output standards.

Existing surveys: With the success of LLM-driven methods for software engineering, many researchers have begun to summarize various aspects of this rapidly evolving field. As shown in Table I, existing surveys exhibit distinct limitations in their scope and coverage. Zhang et al. [9] (2023) provide an early comprehensive study but focus exclusively on program repair without covering agent-based approaches or prompt engineering techniques. While recent surveys from 2024-2025 [10]–[12] have made significant progress in documenting agent capabilities - including planning and decomposition, reasoning and self-refinement, tool augmentation, and memory mechanisms - they notably lack coverage of benchmarks, making it difficult for researchers to evaluate and compare different approaches systematically. Sapkota et al. [13] (2025) provide insights into agent components and prompting strategies but omit both taxonomy and pipeline perspectives, limiting the structural understanding of the field. Most critically, all existing surveys focus on narrow scopes: they either address only code generation [10]–[13] or program repair [9] in isolation, without providing a holistic view of software engineering tasks. None of these surveys successfully bridge the gap between benchmarks and solutions—a crucial connection needed for advancing the field.

Motivations of This Survey: The limitations identified in existing surveys reveal critical gaps that motivate our comprehensive study. **First**, while recent surveys [10]–[12] document individual agent capabilities, they lack a systematic taxonomy that captures how these components integrate into complete agentic systems, from architectural design to problem-solving strategies. **Second**, the absence of benchmark coverage in most recent surveys means researchers cannot effectively evaluate the agent capabilities being proposed, creating a disconnect

between theoretical advances and practical validation. **Third**, the narrow focus on either code generation or program repair prevents understanding of how LLM-empowered systems can address the full spectrum of software engineering challenges, including code translation, testing, and documentation. **Fourth**, without connecting benchmarks to solutions, researchers struggle to identify which approaches work best for specific tasks and where improvements are needed. **Finally**, the rapid evolution of this field, particularly in areas such as multi-agent collaboration, self-refinement mechanisms, and advanced tool integration strategies, necessitates an up-to-date survey that captures these recent breakthroughs. Our survey addresses all these gaps by providing comprehensive coverage across both benchmarks and solutions, spanning all major software engineering tasks, and offering systematic organization through taxonomy and pipeline perspectives that enable researchers to understand not only what solutions exist, but also how to evaluate them effectively.

Selection Papers of This Survey: To ensure comprehensive coverage and high quality, we systematically collected recent papers from multiple sources including top-tier conferences (NeurIPS, ICML, ICLR, ACL, EMNLP, ICSE, FSE, ASE), journals (TSE, TOSEM, TPAMI), and a small amount of preprint servers (arXiv, OpenReview). Our selection criteria include: (1) Papers published between 2023 and 2025 that explicitly address LLM-based approaches for software engineering; (2) Works that contribute novel benchmarks, datasets, or evaluation metrics; (3) Studies proposing new methods, architectures, or techniques for LLM-empowered agentic systems; (4) Papers with significant empirical evaluation demonstrating practical impact.

To provide comprehensive understanding on software engineering, we provide a taxonomy on LLM-powered agentic system, shown in Figure 3. Our taxonomy is organized into two major dimensions: Solutions and Benchmarks. We also provide a pipeline to illustrate the whole process of LLM-empowered software from raw data to applications. To summarize, we provide our contributions as follows:

- **Comprehensive taxonomy bridging benchmarks and solutions:** We provide the first comprehensive taxonomy that systematically organizes 150+ recent papers in LLM-empowered software engineering, uniquely connecting evaluation benchmarks with their corresponding solution approaches. Unlike existing surveys that treat these aspects separately, our framework categorizes solutions into prompt-based, fine-tuning-based, and agent-based paradigms while simultaneously mapping them to relevant benchmarks across code generation, translation,

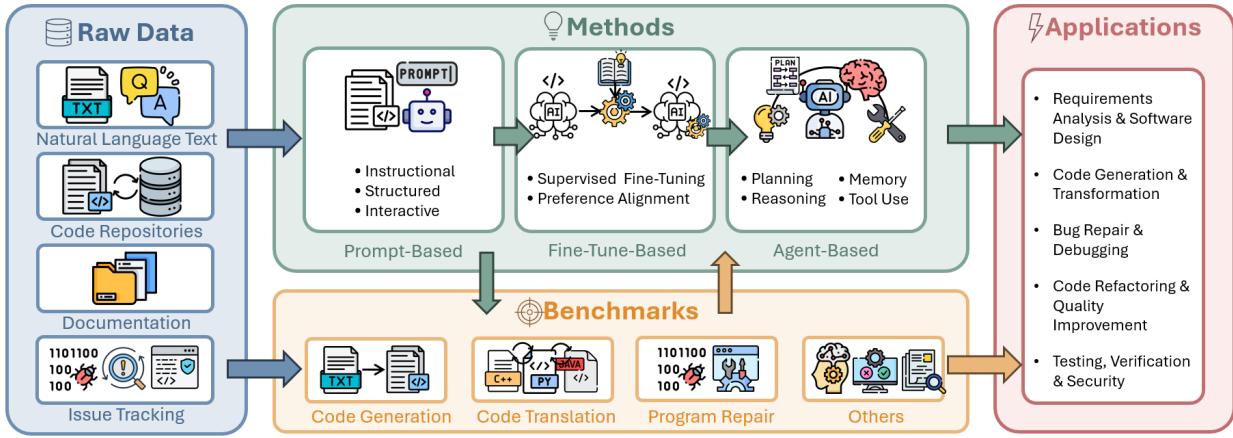


Fig. 2: Pipeline of this survey.

repair, and other tasks, enabling researchers to identify which methods work best for specific evaluation criteria.

- **Unified pipeline with full-spectrum coverage:** We present a unified pipeline that illustrates the complete workflow of LLM-empowered software engineering from initial task specification to final deliverables. Our analysis is the first to provide holistic coverage of all major software engineering tasks rather than focusing on isolated aspects like code generation or program repair. We detail how agent capabilities (planning and decomposition, reasoning and self-refinement, memory mechanisms, and tool augmentation) work synergistically throughout this pipeline to solve complex engineering challenges.
- **Actionable insights and emerging research directions:** Based on our systematic analysis of the field's current state, we identify critical research gaps and provide concrete future directions including multi-agent collaboration strategies for complex software projects, self-evolving code generation systems with continuous learning capabilities, cross-domain knowledge transfer mechanisms, and integration of formal verification methods with LLM-based approaches. These insights offer researchers clear pathways to advance beyond current limitations and develop next-generation software engineering systems.

II. PRELIMINARIES AND PROBLEM DEFINITION

In this section, we provide several key benchmark task definitions of software engineering, namely code generation, code translation, and program repair. The detailed illustrations are shown as follows:

Definition 1 (Code Generation). *We transform high-level concepts into executable codes. We provide formula definitions, which is shown as $\Gamma = \mathcal{S} \times \mathcal{C} \rightarrow \mathcal{P}$, where \mathcal{S} encompasses natural language descriptions, pseudocode, and input-output examples, while \mathcal{C} represents contextual constraints including available APIs, libraries, and environmental factors, and \mathcal{P} denotes the space of executable programs. Traditional approaches to this mapping include rule-based templates (Γ_{rule}) and statistical methods (Γ_{stat}).*

Definition 2 (Code Translation). *We convert source code from a source language to a target language while preserv-*

ing functionality. We define this process with the mapping $\Gamma = \mathcal{S} \times \mathcal{T} \times \mathcal{C} \rightarrow \mathcal{PT}$, where \mathcal{S} represents the source code in the original language, \mathcal{T} denotes the target programming language, and \mathcal{C} captures contextual constraints such as libraries, APIs, and platform requirements. The output \mathcal{PT} is the translated code in the target language that is functionally equivalent to the source. Traditional approaches include rule-based systems (Γ_{rule}) and intermediate representation-based methods (Γ_{IR}).

Definition 3 (Program Repair). *We transform a faulty program into a repaired version that meets given specifications. The repair process is defined by $\Gamma = \mathcal{P}_f \times \mathcal{S} \times \mathcal{C} \rightarrow \mathcal{P}_r$, where \mathcal{P}_f is the faulty program, \mathcal{S} represents specifications (e.g., test suites, assertions), \mathcal{C} captures contextual constraints (e.g., time, resource, and code quality requirements), and \mathcal{P}_r is the repaired program that satisfies \mathcal{S} . Traditional approaches include generate-and-validate methods ($\Gamma_{gen-val}$) and constraint-based repair (Γ_{constr}).*

III. PIPELINE AND TAXONOMY

The application of large language models in software engineering has developed rapidly in recent years, with the emergence of various researches on advanced LLM technologies, as well as numerous benchmarks for evaluating model performance in different tasks. To better understand this rapidly evolving landscape, a systematic framework is needed to analyze and compare existing approaches. Therefore, we propose a taxonomy as a core tool for systematic reviews.

Distinguishing itself from existing surveys, this work reviews LLMs researches in software engineering from two dimensions: solutions and benchmarks.

- **Solutions:** Based on the core technologies employed, the existing approaches can be categorized into three categories: (1) **Prompt-based solutions**, including instructional, structured, and interactive prompting, which are the most straightforward use of LLMs; (2) **Fine-tune-based** solutions, which enhance model performance through supervised fine-tuning as well as preference alignment via reinforcement learning (RL) or RL-free method; (3) **agent-based** solutions, which integrate

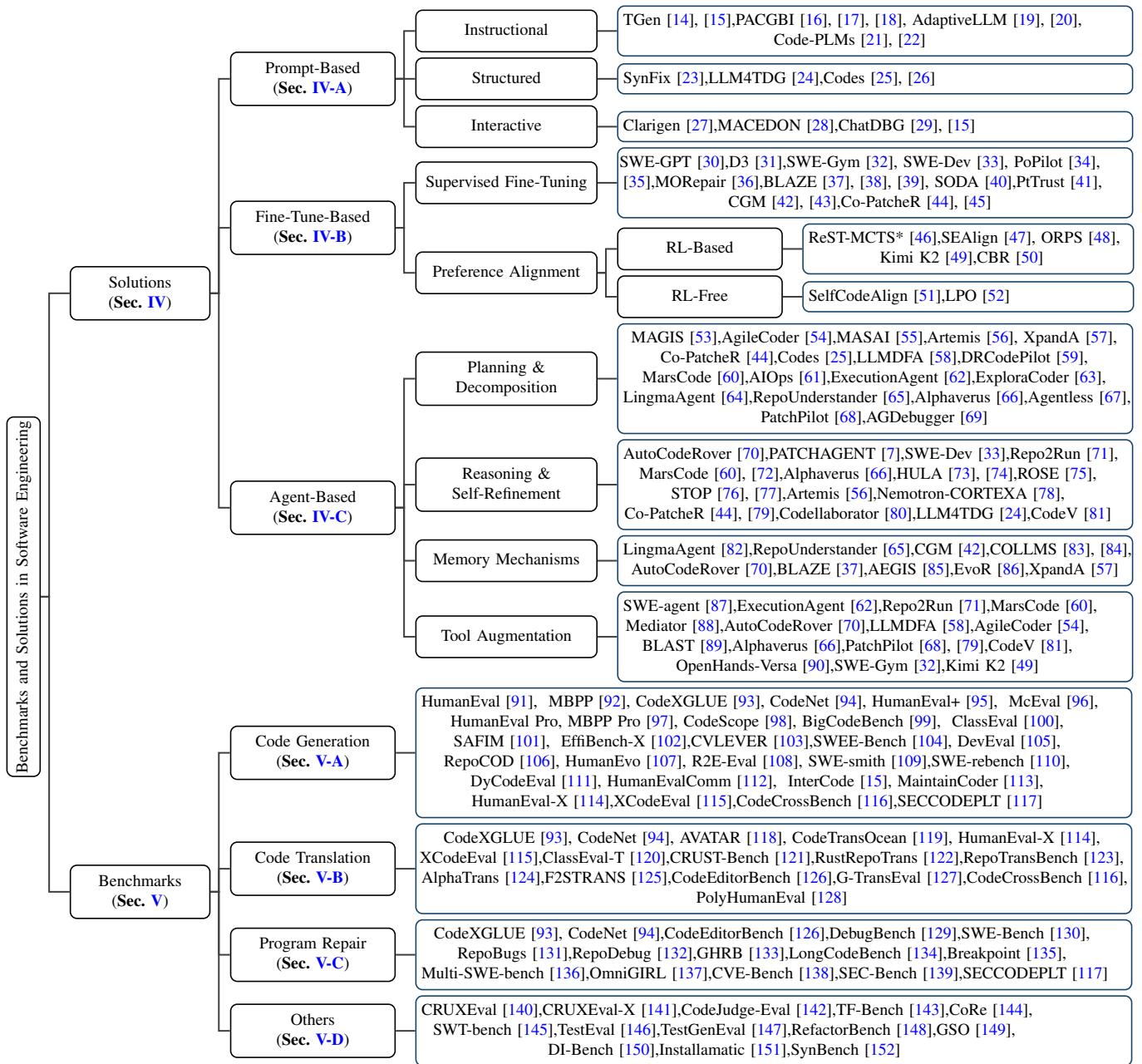


Fig. 3: Taxonomy of existing studies for Software Engineering

task decomposition and planning, reasoning and self-improvement, memory mechanisms, and tool usage, enabling higher levels of automation and collaboration.

- **Benchmarks:** Based on the targeted software engineering tasks, exist benchmarks are categorized into four types: benchmarks for (1) **code generation**; (2) **code translation**; (3) **program repair**; (4) **Others**, such as code reasoning, test generation, code refactoring, and so on.

As shown in Fig 4, we reviewed 140 distinct papers categorized into **Solutions** (94 papers) and **Benchmarks** (72 papers), with the distribution shown for each subcategory. Papers addressing multiple aspects may appear in more than one category. Figure 3 illustrates the details of this taxonomy. Leveraging this framework, we trace the paradigm shift from

prompt-based methods to agent-based systems and position benchmarks according to their coverage of different software engineering scenarios, providing insights into real-world application development.

The overall pipeline of this survey is presented in Figure 2. Specifically, Section IV reviews LLM-based solutions for software engineering, Section V examines benchmark studies, and Section VI discusses real-world applications in detail.

IV. SOLUTIONS

LLMs have catalyzed a paradigm shift in automated software engineering, solving challenges in code generation, maintenance, and analysis. This section reviews emerging methodologies, categorized by increasing autonomy: **Prompt-**



Fig. 4: Overview of reviewed studies.

based solutions, relying on human instruction; **Fine-tune-based** solutions, adapting models to the software engineering domain; and **Agent-based** solutions, autonomous systems that plan, act, and learn. This progression shows the evolution from LLMs as tools to autonomous partners in the software development lifecycle.

A. Prompt-based Solutions

Prompt-based methods are a direct way to interact with LLMs. The model’s behavior is guided by the context and instructions in a single input, or “prompt,” without modifying its parameters. The effectiveness of these methods hinges on prompt engineering, which crafts inputs to elicit the desired output. This section explores software engineering strategies, from natural language instructions to structured and interactive prompting techniques that enhance performance and reliability, as shown in Fig. 5.

1) *Instructional*: Instructional prompting is the most common interaction, where a user provides a natural language task description. This approach leverages the LLM’s ability to understand and follow commands, making it versatile for tasks like code generation, documentation, and bug explanation. This subsection reviews studies on instructional prompts, exploring how instruction composition influences the quality and correctness of generated code, including using examples (few-shot), test cases, or specific formatting.

A key finding is that the precision and verifiability of instructions are paramount. A powerful demonstration is using Test-Driven Development (TDD) as a prompting paradigm. Mathews *et al.* found that providing an LLM with human-written unit tests alongside a natural language description serves as a detailed, executable specification, leading to a higher success rate in generating correct code [14]. This approach provides a formal “contract” the generated code must fulfill, reducing ambiguity.

In contrast, instructions based on less formal specifications can be less effective. Fu *et al.* evaluated LLMs on iterative code generation from input-output (I/O) examples, finding model performance decreases significantly compared to when requirements are described in natural language [15]. This highlights a challenge for LLMs in abstracting complex logic from raw examples without semantic guidance from natural language.

Instructional prompts are explored across software engineering activities. For low-complexity tasks, zero-shot prompting shows promise. Sarschar *et al.*’s PACGBI generates code from product backlog items, demonstrating feasibility but noting challenges with less detailed descriptions [16]. Prompts are

also used for maintenance tasks like refactoring architectural smells [17] and improving requirements engineering by analyzing feature requests [18].

Prompt structure is actively researched. Liu *et al.* found that with Chain-of-Thought (CoT) prompting, it is more effective to generate code first, then reasoning [20]. Cheng *et al.*’s AdaptiveLLM framework uses CoT length as a proxy for problem difficulty to select a cost-efficient LLM [19].

Research has highlighted limitations of current models on specialized or less-common programming languages. Zhao *et al.* evaluated Code-PLMs on the R programming language and found performance degrades due to R’s unique characteristics, indicating general-purpose instructional prompting techniques do not always transfer effectively across language paradigms [21]. To improve context for downstream tasks, Kondo *et al.* propose using an LLM to convert code snippets into natural language, allowing more semantically rich searches over textual representations rather than raw code [22].

2) *Structured*: To overcome natural language ambiguity, structured prompting imposes a formal, machine-parsable format on the LLM’s input. Providing information in a predictable structure, like a dependency graph, constrains the model’s generation, leading to more reliable outputs. This subsection surveys approaches leveraging structural information from the codebase or task to guide the LLM, enhancing its reasoning about complex dependencies.

A prominent strategy uses graph-based representations to capture codebase relationships. Tang *et al.*’s SynFix uses a “RelationGraph” to map dependencies between code elements [23]. This constrains the LLM’s repair task, ensuring a fix is propagated to dependent components. Similarly, Liu *et al.*’s LLM4TDG defines a “constraint dependency graph” from testing objectives, converted into prompt constraints to improve test driver generation [24].

Another approach imposes a hierarchical structure on generation. Zan *et al.*’s Codes framework generates code repositories from a natural language description (NL2Repo) [25]. It employs a multi-layer sketch, generating a repository sketch (directories, dependencies), then a file sketch (imports, signatures), and finally the implementation, ensuring architectural coherence.

Even for models with long context windows, the input structure remains critical. Peng *et al.* investigated long-context models for repository-level code generation and found that concatenating all relevant files is suboptimal [26]. Their study reveals the ordering of code snippets in the prompt significantly impacts performance, suggesting a well-structured prompt reflecting the logical and dependency-based ordering of the code is crucial.

3) *Interactive*: Interactive prompting extends static prompts into a multi-turn dialogue between the user and the LLM. This conversational approach allows for progressive refinement of solutions. The system can ask clarifying questions to resolve ambiguity, and the user can provide iterative feedback to steer the model toward a correct output. This subsection examines frameworks that facilitate this human-AI collaboration, transforming code generation into a dynamic partnership.

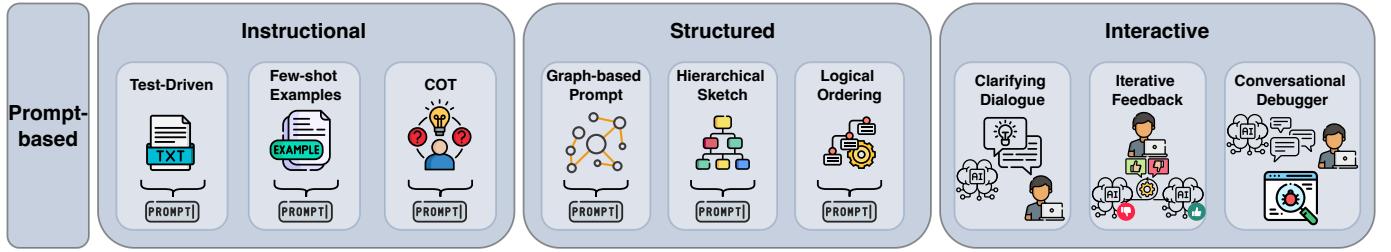


Fig. 5: The components of prompt-based solutions.

A key motivation for interaction is bridging the semantic gap between user intent and the details needed for code generation. Miao *et al.*'s Clarigen framework identifies ambiguities in a user's request and engages in a clarifying dialogue [27]. The question asked to the targeted audience enriches the initial prompt with user feedback prior to generation, improving the alignment with the intent of the user.

Interaction can be user-driven within the IDE. Liu *et al.*'s MACEDON provides real-time code evaluation and optimization suggestions [28]. A more advanced example is ChatDBG by Levin *et al.*, which augments debuggers with an LLM [29]. This makes the debugger a conversational partner, allowing programmers to ask high-level questions (e.g., “why is this variable null?”) to find a bug’s root cause.

However, interaction’s effectiveness is not guaranteed. The study by Fu *et al.* employed an iterative framework where new I/O examples could be provided if the code was incorrect [15]. They found most successes occurred in the first round, suggesting current LLMs struggle to use iterative feedback when presented as I/O examples. This indicates feedback modality and clarity are critical for successful interaction.

B. Fine-tune-based Solutions

Fine-tuning is a crucial step beyond prompt engineering, specializing a general-purpose LLM for software engineering. This process involves training a pre-trained model on a curated dataset of domain-specific examples, adapting its parameters to better understand code syntax, semantics, and software development patterns. This section covers primary fine-tuning methodologies, from supervised learning on code-related tasks to preference alignment techniques that teach models to discern high-quality from suboptimal solutions as illustrated in Fig. 6.

1) Supervised Fine-Tuning (SFT): Supervised Fine-Tuning (SFT) is the cornerstone of model specialization. It involves training an LLM on a dataset of high-quality input-output pairs, like bug reports and their patches, or natural language descriptions and their code. This process directly teaches the model desired behaviors for specific software engineering tasks. This subsection explores diverse SFT applications, focusing on innovations in dataset creation, curriculum learning strategies, and process-centric training capturing the software development workflow.

A significant SFT trend is shifting from static code snapshots to process-oriented data. Ma *et al.* developed SWE-GPT, fine-tuned on data simulating the software improvement process, including repository understanding, fault localization,

and patch generation [30]. Similarly, Piterbarg *et al.* created the D3 dataset, framing code generation as a sequence of file diffs, fine-tuning the model to make iterative changes rather than generating entire files from scratch [31]. Pan *et al.* introduced SWE-Gym, an environment to collect agent’s sequences of actions for solving real-world tasks for SFT [32].

Data-centric SFT, focusing on training data quality and scale, is highly effective. Chowdhury *et al.* present SWE-Dev, a methodology creating powerful open-source agents by generating a massive dataset of “fail-to-pass” test cases from GitHub issues, then used to collect agent trajectories for SFT [33]. This scaling allows smaller models to close the performance gap with larger ones. In formal verification, where data is scarce, Zhang *et al.* developed PoPilot using a synthetic data generation process to teach a model the complex reasoning for proof-oriented programming, enabling a 14B model to outperform GPT-4o [34]. Data curation’s importance is highlighted by Cassano *et al.*, who show finetuning open Code LLMs on their dataset of code edits with natural language instructions significantly improves their editing capabilities [35].

Targeted SFT strategies address specific software engineering goals. Yang *et al.* propose MORRepair, a multi-objective approach training a model on both the syntactic code transformation (the “what”) and the reasoning behind the fix (the “why”), for higher-quality repairs [36]. For bug localization, Chakraborty *et al.* use “hard example learning” in their BLAZE framework, focusing the SFT process on complex bugs to improve model generalizability [37]. Sagtani *et al.* improve Fill-in-the-Middle (FIM) code completion by using curriculum learning, creating a dataset of “hard-to-complete” examples to focus the training where models most often fail [38]. For performance optimization, Shypula *et al.* finetune a model in their ECO system on a curated dataset of performance anti-patterns and their corresponding optimized solutions mined from historical commits [39].

Other SFT approaches include knowledge distillation and leveraging model internals. Chen *et al.* introduce SODA, a framework using a self-paced knowledge distillation cycle to train smaller “student” models to mimic larger “teacher” models [40]. Huang *et al.* developed PtTrust, a risk assessment framework pre-training a risk predictor on a code LLM’s internal states to identify incorrect or insecure generations [41]. Architectural innovations also combine with SFT. Yu *et al.* propose the Code Graph Model (CGM), integrating a repository graph representation into the LLM’s attention mechanism and using a two-phase SFT process to teach the model structural and semantic dependencies [42]. Finally,

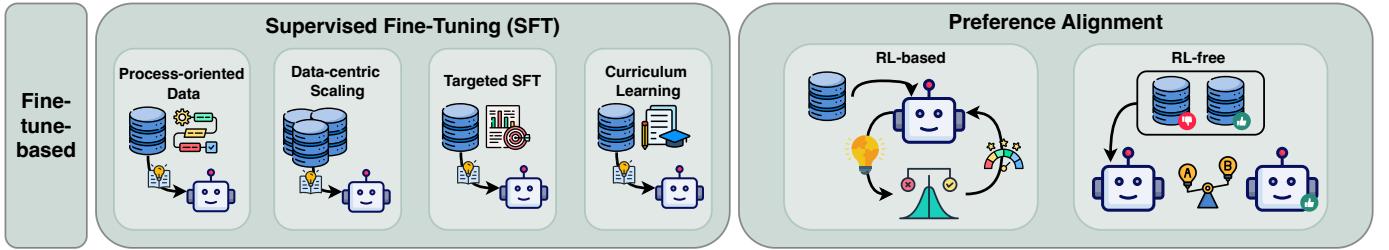


Fig. 6: The components of fine-tune-based solutions.

studies explore SFT's limits. Wang *et al.* found incorporating semantic information from execution traces provided limited SFT benefit, challenging a common assumption [43]. Wei *et al.* introduce Co-PatcheR, using an efficient SFT strategy with data distillation and filtering to train small, specialized models for different software patching sub-tasks [44]. Project-level context's importance is confirmed by Sapronov *et al.*, who show continued pre-training (a form of SFT) on repository-level data significantly improves code completion [45].

2) *Preference Alignment: RL-based:* Reinforcement Learning (RL)-based alignment is a powerful paradigm for optimizing LLMs against complex, non-differentiable reward signals, like code correctness from passing a test suite. The LLM acts as an agent whose policy (generation strategy) is updated by rewards from an environment or reward model. This allows the model to learn complex reasoning and problem-solving. This subsection examines frameworks using RL, often guided by techniques like Monte Carlo Tree Search, for tasks requiring sophisticated reasoning and planning.

Reward function design is central to RL-based alignment. Several works learn from the reasoning process itself. Zhang *et al.* introduce ReST-MCTS*, a reinforced self-training method using a process-reward model (PRM) to guide an MCTS for generating quality reasoning traces [46]. The system infers per-step rewards without manual annotation, creating a self-improvement loop fine-tuning the PRM and policy model. Similarly, Zhang *et al.* use SEAlign, using MCTS to evaluate action trajectories and identify “critical actions” that impact success [47]. The model is then fine-tuned on these critical actions with preference optimization, teaching it to favor effective decision-making paths.

Other approaches generate the reward signal dynamically from execution outcomes. Yu *et al.* propose Outcome Refining Process Supervision (ORPS), where the “reward” is generated at inference time from execution feedback (*e.g.*, correctness, runtime, memory usage) and the LLM’s own self-critique [48]. The agent explores a tree of thoughts, and this dynamically generated reward signal allows it to learn a preference for more efficient and correct implementation strategies without a pre-trained reward model. This unification of process and outcome rewards represents a significant step towards more autonomous learning.

RL also aligns models with production scenarios. Guo *et al.* use an RL fine-tuning stage to optimize a system for generating functional test scripts [50]. After an SFT phase, the model is trained with RL where the reward is based on the similarity between the generated and ground-truth scripts, improving accuracy and reliability. The Kimi K2 model undergoes a joint

RL stage, interacting with real and synthetic environments to enhance its agentic capabilities, particularly in tool use [49].

3) *Preference Alignment: RL-free:* Preference alignment methods fine-tune models based on judgments of which outputs are better, moving beyond right-or-wrong supervision. RL-free techniques, such as Direct Preference Optimization (DPO), achieve this without reinforcement learning’s complexity and instability. They operate on “chosen” and “rejected” responses, directly optimizing the model to increase the likelihood of preferred outputs. This subsection reviews methods using RL-free alignment to instill qualities like code security, efficiency, or adherence to coding styles.

A key challenge is sourcing a quality preference dataset. Cassano *et al.* address this with SelfCodeAlign, a self-alignment pipeline that bootstraps preference data [51]. The base model generates diverse coding tasks and multiple responses for each. It then generates test cases to validate responses in a sandbox. Only instruction-response pairs passing validation are used for the preference dataset, demonstrating a transparent, self-sufficient alignment method.

Innovations are also occurring in the optimization algorithm. Saqib *et al.* introduce Localized Preference Optimization (LPO) to teach LLMs secure coding [52]. After distilling a preference dataset of secure vs. insecure code from a teacher model, LPO applies the preference loss only to tokens where the secure and insecure versions diverge. These methods show RL-free alignment is a promising direction for instilling complex attributes into code generation models stably.

C. Agent-based Solutions

Agent-based systems are the frontier of LLM-driven software engineering, shifting the paradigm from single-shot generation to autonomous, multi-step problem-solving. An “agent” is a system that can perceive its environment, create and decompose plans, utilize external tools, and learn from feedback to achieve a high-level goal. This approach mimics the iterative and interactive workflow of a human developer, enabling LLMs to tackle complex, repository-level tasks intractable with simple prompting or fine-tuning alone. This section explores the core components of modern software engineering agents, as shown in Fig. 7

1) *Planning & Decomposition:* Complex software tasks require breaking down a high-level goal into a sequence of manageable sub-tasks. The Planning and Decomposition component of an agent creates this strategy. Approaches range from static, predefined workflows that guide the agent through fixed stages to dynamic, LLM-driven planning where the agent reasons about its next best action at each step.

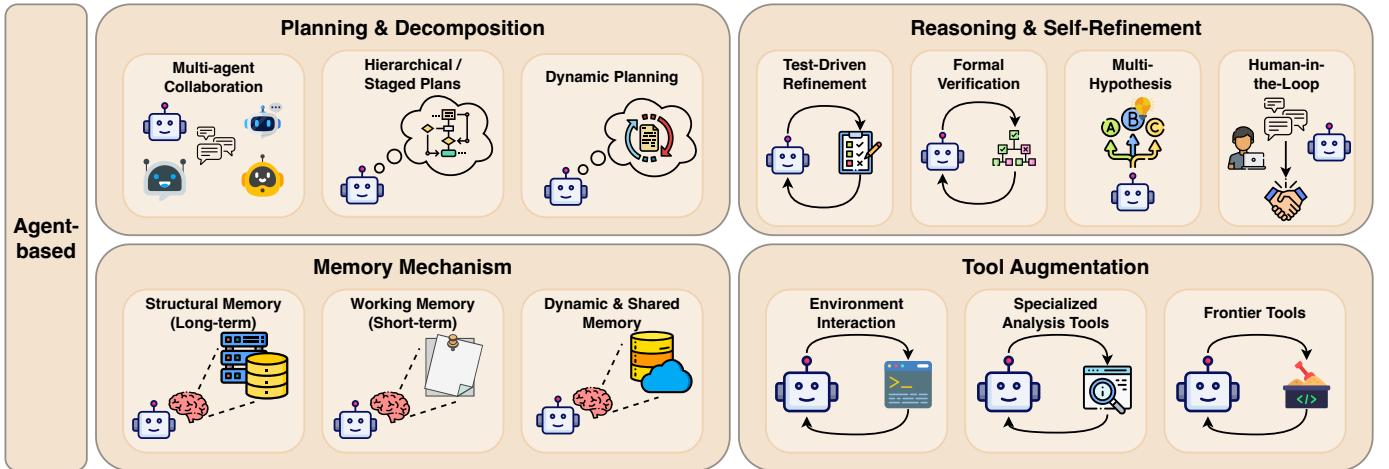


Fig. 7: The components of agent-based solutions.

A popular approach is multi-agent collaboration, which decomposes tasks by assigning specialized roles. Tao *et al.* introduce MAGIS, a framework with Manager, Developer, and QA Engineer agents mimicking a software team [53]. Nguyen *et al.* model the Agile methodology in AgileCoder, with agents for roles like Product Manager and Tester [54]. Wadhwa *et al.* propose MASAI, a modular architecture that decomposes issue resolution into a static pipeline of sub-agents: Test Template Generator, Issue Reproducer, Edit Localizer, Fixer, and Ranker [55]. The Artemis AI framework decomposes code optimization by identifying bottlenecks and distributing these snippets to specialized LLM agents [56]. Similarly, Xiao *et al.* use a “divide and conquer” strategy in XpandA, where a long text is partitioned and assigned to multiple “Explorer” agents [57]. The Co-PatcheR framework uses a collaborative system of smaller, specialized models for localization, generation, and validation [44].

Other approaches use a hierarchical or staged plan. Zan *et al.*’s Codes framework decomposes repository generation into outlining the repository, sketching each file, and then filling the implementation [25]. Wang *et al.* developed LLMDFA, which decomposes dataflow analysis into a three-phase pipeline: source/sink extraction, dataflow summarization, and path feasibility validation [58]. Zhao *et al.* introduce DRCodePilot, which formalizes a “design first, then code” plan by leveraging design rationales from issue logs before generating the patch [59]. The MarsCode Agent employs a systematic, multi-phase process including planning, reproduction, localization, generation, and validation [60]. In a vision paper, Shetty *et al.* outline a framework for AIOps agents to handle tasks like root cause analysis through a planned sequence of observing, hypothesizing, and acting [61].

More dynamic planning mechanisms allow the agent to adapt its strategy. The ExecutionAgent by Bouzenia *et al.* uses a two-phase plan for running tests: a preparation phase to gather information, followed by a feedback loop to refine its plan based on command outputs [62]. Wang *et al.* propose ExploraCoder, which decomposes using unseen APIs into subtasks and uses a “Chain of API Exploration” to rectify its plan based on executability feedback [63]. Ma *et al.* introduce LingmaAgent, which uses Monte Carlo Tree Search

(MCTS) to explore a repository knowledge graph, allowing it to plan its information gathering before attempting a fix [64]. This “understand-then-act” approach is further detailed in RepoUnderstander [65]. The Alphaverus framework decomposes verified code generation into three stages: Exploration, Treefinement, and Critique [66]. In contrast, Xia *et al.* challenge complex planning with “Agentless,” a framework using a fixed, three-phase process (localization, repair, validation), suggesting a simple plan can be sufficient [67]. PatchPilot employs a stable, rule-based workflow of reproduction, localization, generation, validation, and refinement [68]. Finally, some works focus on observing the planning process itself. Epperson *et al.* developed AGDebugger, a tool for developers to visualize, inspect, and edit messages between agents, allowing modification of the distributed plan [69].

2) *Reasoning & Self-Refinement:* Reasoning and self-refinement are the cognitive core of an agent, enabling it to analyze feedback and improve its solutions. This is often implemented as a “generate-test-revise” loop, where the agent proposes a solution, evaluates it with feedback, and reasons about the outcome to generate a better solution. This subsection explores this feedback loop, from human-in-the-loop systems to autonomous self-critique.

Refinement is most commonly driven by execution and testing feedback. AutoCodeRover by Zhang *et al.* uses an iterative code search and patch generation loop; if a patch fails, the agent retries [70]. Yu *et al.* designed PATCHAGENT with an integrated verifier that tests patches against security and functional tests; failures trigger a new reasoning cycle [7]. Chowdhury *et al.* introduce “iteration scaling” in SWE-Dev, giving the agent more interaction rounds to analyze test failures and refine its code [33]. Hu *et al.* developed Repo2Run, an agent that creates a Dockerfile by iteratively building an image, analyzing errors, and refining it [71]. The MarsCode Agent uses feedback from a sandboxed environment, like exception stacks, to drive its refinement process [60]. Wang *et al.* found many “solved” issues in the SWE-bench benchmark are incorrect, calling for more sophisticated validation in agent refinement loops [72].

A more rigorous form of feedback comes from formal verification. Aggarwal *et al.* created Alphaverus, a self-improving

agent using feedback from a formal verifier to correct errors [66]. Its “Treefinement” mechanism treats refinement as a tree search, allowing it to explore different correction paths.

Human-in-the-loop refinement leverages developer expertise. Takerngsiri *et al.* propose the HULA framework, where an AI generates outputs for a Human Agent to review and correct [73]. Omidvar Tehrani *et al.* evaluated a human-AI partnership for code migration, showing human oversight is critical for tasks requiring contextual understanding [74]. Reiss *et al.* developed ROSE, an IDE-based framework where the developer initiates repair from a debugger, then selects the fix from the agent’s suggestions [75].

Some agents employ self-critique or explore multiple hypotheses. Zelikman *et al.* introduced Self-Taught Optimizer (STOP), where an “improver” program recursively optimizes itself by proposing and evaluating modifications to its own code [76]. Dou *et al.* proposed an iterative self-critique method where an LLM corrects its code based on a bug taxonomy and compiler feedback [77]. The Artemis AI framework generates candidate optimizations from multiple LLMs and uses a search and filtering process to select the best solution [56]. Sohrabizadeh *et al.* use a similar strategy in Nemotron-CORTEXA, generating diverse candidate patches and using self-generated tests and majority voting to select the best one [78]. The reasoning in Co-PatcheR is a multi-agent affair where models generate and critique solutions, with a majority vote for the final decision [44].

Other notable reasoning approaches include using simulation for feedback, as demonstrated by Nouri *et al.*, whose agent generates code for autonomous driving systems, which is tested in a simulation environment, with results fed back for refinement [79]. Pu *et al.* explore proactive agent reasoning in “Codellaborator,” which analyzes developer context to decide when to offer assistance [80]. The reasoning process can also be embedded, as in LLM4TDG, which uses constraint reasoning and backtracking to refine generated test drivers [24]. Finally, the reasoning in CodeV is enhanced by processing visual information, allowing it to use visual feedback to verify a patch, especially for GUI bugs [81].

3) *Memory Mechanism:* To solve repository-level tasks, an agent must overcome LLM context window limitations by managing information. Memory mechanisms provide agents the ability to store, retrieve, and synthesize information from codebases and interaction histories. These mechanisms can be categorized into short-term “working memory” for context retrieval and long-term “structural memory” that captures an understanding of the repository. This subsection investigates memory architectures like vector databases, knowledge graphs, and evolving knowledge bases.

Several agents focus on building a repository-scale structural memory for a global understanding of the codebase. Ma *et al.* introduce LingmaAgent, which condenses repository information into a queryable knowledge graph, allowing more informed decisions [82]. This concept is further developed in RepoUnderstander, which organizes the repository into a hierarchical tree of files, classes, and functions as the agent’s memory [65]. Yu *et al.* take this further with the Code Graph Model (CGM), which integrates a code graph with hierar-

chical and reference dependencies into the LLM’s attention mechanism as a structured memory system [42]. Truong *et al.* propose the COLLMS framework, which integrates LLMs with a “Platform Knowledge” base of software catalogs and code patterns, acting as external memory to compensate for lack of domain-specific information [83].

Other agents rely on on-demand context retrieval to create a “working memory”. Zhang *et al.*’s AutoCodeRover interacts with an AST, using APIs to retrieve code snippets and build contextual memory before patch generation [70]. Chakraborty *et al.* use dynamic chunking in BLAZE to break source code into indexed segments, allowing the agent to query for code chunks relevant to a bug report [37]. The AEGIS framework for bug reproduction uses a context construction module that extracts information from the issue description and relevant code, creating a focused memory [85]. Ehsani *et al.* propose a method to identify knowledge gaps in a prompt, triggering an agent to retrieve information from a knowledge base to fill gaps in its context [84].

The most advanced memory mechanisms are dynamic. Su *et al.* developed EvoR, a retrieval-augmented generation pipeline with a dynamic “knowledge soup” memory. This memory is updated with information from web searches, documentation, and execution feedback, allowing the agent to learn and adapt [86]. For multi-agent systems, memory must be coordinated. Xiao *et al.* use a centralized shared memory in their XpandA framework, updated via a “question-guided protocol” to maintain consistent understanding across collaborating agents [57].

4) *Tool Augmentation:* Tool augmentation grounds an agent’s capabilities by giving it the ability to interact with the software development environment. By equipping agents with tools—such as terminal access, file editors, and compilers—they can perform actions and gather information impossible with text generation alone. This subsection overviews tool-use strategies, from Agent-Computer Interfaces to integrating static analysis, formal verification, and multimodal inputs.

A fundamental tool use is interacting with a development environment like a terminal or IDE. Yang *et al.* introduced SWE-agent, centered on an Agent-Computer Interface (ACI) with commands tailored for software engineering, like a file viewer and editor, proving more effective than general terminal access [87]. Bouzenia *et al.*’s ExecutionAgent uses terminal commands to explore and run build systems, and “meta-prompting,” where an LLM generates guidelines for using specific technologies [62]. Hu *et al.*’s Repo2Run agent depends on the Docker engine and testing frameworks; its workflow is a loop of sending commands and parsing output to refine a Dockerfile [71]. The MarsCode Agent utilizes code knowledge graphs and the Language Server Protocol to navigate and analyze code like a developer using an IDE [60]. Li *et al.* propose mediator agents in IDEs to enhance human-developer interaction with LLMs [88].

Many agents are augmented with specialized analysis tools. Zhang *et al.*’s AutoCodeRover integrates Spectrum-Based Fault Localization (SBFL) to analyze test outcomes and assign suspicion scores, directing the bug search [70]. Wang *et al.*’s LLMDFA uses the LLM as a “tool-builder,” generating scripts

that use parsing libraries to identify sources/sinks and the Z3 theorem prover to validate dataflow paths [58]. Nguyen *et al.*'s AgileCoder uses a “Dynamic Code Graph Generator” to provide agents an evolving representation of the software’s structure [54]. Kitsios *et al.* developed BLAST, a tool that combines LLM with Search-Based Software Testing (SBST), where the LLM generates a “seed” test that the SBST tool evolves [89].

The frontier of tool use involves formal verification, simulation, and multimodal inputs. Aggarwal *et al.*'s Alphaverus depends on an external formal verifier; its error messages are the primary tool governing the agent’s action loop and “Treefinement” search [66]. PatchPilot also uses formal verification tools for higher assurance [68]. Nouri *et al.* integrate their agent with a simulation model for autonomous driving to evaluate generated code in realistic scenarios [79]. Expanding sensory input, Zhang *et al.* created CodeV, an agent that processes screenshots and videos from issue reports to resolve GUI bugs difficult to diagnose from text [81]. Soni *et al.* introduced OpenHands-Versa, a generalist agent that uses a minimal set of tools, including multimodal web browsing and file access [90].

Finally, the design of tools and the environment is a research area. Pan *et al.*'s SWE-Gym provides an interactive environment with thousands of Python tasks for training and evaluating tool-using agents [32]. The Kimi K2 model is optimized for agentic tool use, capable of orchestrating commands to render, test, and debug code [49].

V. BENCHMARKS FOR REAL-WORLD SOFTWARE ENGINEERING

Benchmarks have long been a core tool for advancing LLM research for software engineering. From early function-level code generation datasets, such as HumanEval [91] and MBPP [92], to large-scale multilingual benchmarks, such as CodeNet [94], these efforts have significantly advanced LLM development. However, most of these efforts focus on fragmented tasks, which only partially capture the complexity of real-world software engineering. In practice, software systems involve complex processes such as repository-level generation, bug reporting, testing, and integration. This gap has motivated the creation of real-world code benchmarks, such as SWE-bench [130] and its successors, which incorporate real GitHub issues into their testing environments, allowing them to evaluate the performance of models and agents in scenarios more closely resembling those of real software development. This chapter reviews these benchmarks based on their focused tasks, including code generation, code translation, program repair, and other tasks.

A. Code Generation

Code generation refers to automatically generating program code according to natural language descriptions. Pioneering benchmarks in code generation, such as HumanEval [91], MBPP [92], and CodeXGLUE [93], typically focused on function-level tasks, evaluating the ability of language models to generate syntactically correct and functionally equivalent

TABLE II: Representative Benchmarks for three SE tasks (1. Code Generation, 2. Code Translation, and 3. Program Repair) at Function- and Repository-Level. *SWE-Bench reports 2,294 instances; each instance may generate multiple test cases.

Benchmark	Tasks	#PLs	Scope	#Tests
HumanEval	1	1 (Python)	Function	164
DevEval	1	1 (Python)	Repository	1,874
CodeXGLUE	1,2,3	10 (Python, Java, C++, C, etc.)	Function	> 100K
AlphaTrans	2	2 (Java, Python)	Repository	2,719
SWE-Bench	3	1 (Python)	Repository	2,294*

code implementations. To support testing on more programming languages, Puri *et al.* [94] proposed CodeNet covering 55 programming languages. However, these benchmarks often rely on relatively simple tasks and limited testings, which fail to reflect real-world software engineering situations. To mitigate this issue, Liu *et al.* [95] proposed HumanEval+, extending HumanEval [91] by 80x with synthetic test cases.

Subsequent work has enriched evaluation along multiple dimensions. Chai *et al.* [96] proposed McEval, covering 40 PLs with 16,000 samples and supporting tasks beyond code generation, including code review, and error detection. Similarly, Yan *et al.* [98] proposed CodeScope, covering 43 programming languages and 8 coding tasks. While, Yu *et al.* [97] extended the HumanEval and MBPP benchmarks by introducing a self-invoking code generation task. This task requires models to first solve a base problem and then utilize its solution to address a related but more complex problem. Meanwhile, Zhuo *et al.* [99] developed BigCodeBench, focusing on evaluating LLM on handling complex instructions and diverse function call. In terms of evaluation scope, Du *et al.* [100] introduced class-level generation tasks in ClassEval. Gong *et al.* [101] constructed a benchmark for the Fill-in-the-Middle (FIM) completion. Beyond functional correctness, Qing *et al.* [102] proposed EffiBench-X, highlighting efficiency gaps between LLMs and human experts Thakur *et al.* [103] introduced formal verifiability in CLEVER, which requires generated code to pass Lean’s type checker.

To better align with realistic software development processes, Inspired by SWE-Bench, Vergopoulos *et al.* [104] proposed an automated system SetUpAgent that can construct repository-level code benchmarks from GitHub repositories and released an extended SWE-bench. Li *et al.* [105] proposed DevEval including 117 repositories with manual annotations, aiming to preserve realistic code distributions and dependency distributions. Similarly, Liang *et al.* [106] created RepoCOD sourced from 11 real-world projects. Recognizing the evolving nature of software systems, Zheng *et al.* [107] developed an evolving repository-level code generation benchmark, named HumanEvo. It evaluates the LLM performance across project evolution by treating multiple project versions as contextual sources. In addition, Jain *et al.* [108] proposed a framework that can transform GitHub repositories into interactive environment and released R2E-Eval benchmark to evaluate LLM-based agents on code generation. Recently, Yang *et al.* [109] and Badertdinov *et al.* [110] proposed SWE-smith

and SWE-rebench, respectively, both significantly extending SWE-Bench [130] to benchmark LLM agents under more diverse and realistic settings.

Beyond functionality-oriented evaluation, Chen *et al.* [111] focused on data contamination issues and proposed DyCodeEval, which benchmarks LLMs under varying contamination levels. Considering the ambiguity, inconsistency, and incompleteness often present in real-world software engineering problem descriptions, Wu *et al.* [112] designed HumanEval-Comm to evaluate the communication skills of LLMs in code generation tasks. Similarly, Fu *et al.* [15] proposed InterCode, which evaluates code generation using only I/O examples, thereby emphasizing the need for LLMs to understand and implement incomplete and diverse code requirements. In addition, Wang *et al.* [113] proposed MaintainCoder, which emphasizes that generated code should not only be functionally correct but also meet maintainability standards.

B. Code Translation

Code translation, which aims to convert code from one programming language (PL) to another while preserving its functionality, is a critical task in modern software engineering. It enables the migration of legacy systems, facilitates interoperability across heterogeneous platforms, and supports cross-language software development. Leveraging their ability to capture complex code semantics and contextual dependencies, LLMs have emerged as powerful tools for advancing the automation of code translation.

To systematically evaluate the capability of LLMs in this task, various benchmarks have been established. CodeTrans in CodeXGLUE [93] provided parallel Java–C# code from open-source projects. CodeNet [94] further extended coverage to 55 PLs. Ahmad *et al.* [118] presented AVATAR for Java–Python translation, providing unit tests for 250 evaluation examples to measure functional accuracy. In general, these works mainly focusing on function-level tasks.

Building on these datasets, subsequent benchmarks expanded both language coverage and evaluation methodology. Yan *et al.* [119] constructed CodeTransOcean supporting code translation across 8 PLs and including a dataset for translating deep learning code across different frameworks. They also introduced a execution-based evaluation metric Debugging Success Rate @K to evaluate the performance of LLMs. Similarly, Zheng *et al.* [114] and Khan *et al.* [115] released HumanEval-X and xCodeEval, respectively, and adopted execution-based evaluation to address the shortcomings of similarity metrics such as BLEU [153] and CodeBLEU [93]. Scaling up, Nie *et al.* [116] provided 216 tasks across 18 PLs in CodeCrossBench to assess generalization across diverse software engineering tasks. While Tao *et al.* [128] extended HumanEval [91] into PolyHumanEval, covering 14 PLs for large-scale multilingual evaluation.

Driven by real-world migration needs, benchmarks then advanced to class-level and repository-level translation. Xue *et al.* [120] extended ClassEval into ClassEval-T, supporting Python to Java and C++ class-level tasks. At the repository level, several large-scale benchmarks have emerged, such

as CRUST-Bench [121], RustRepoTrans [122], and RepoTransBench [123], each introduced extensive translation tasks from real-world GitHub repositories. Further advancing this direction, Ibrahimzada *et al.* [124] proposed AlphaTrans, a neuro-symbolic framework to automate repository-level code translation and applied it to translate ten real-world open-source projects.

Beyond functional correctness, research has also expanded to assess translation quality and robustness. Zhang *et al.* [125] proposed F2STRANS, which includes manual annotations to support stylistic readability evaluation. While Guo *et al.* [126] presented CodeEditorBench to evaluate broader editing and transformation capabilities. Jiao *et al.* [127] established a four-level taxonomy of translation tasks, including token, syntactic, library, and algorithm levels, and built G-TransEval for integrated evaluation.

C. Program Repair

Automated Program Repair(APR) aims to software reliability by automatically locating and fixing software bugs. The development of LLMs has transformed the APR research paradigm and evaluation methodologies, promoting the development of diverse benchmarks that assess the capabilities of APR technologies at different levels and in different scenarios.

To test LLM debugging capabilities, Tian *et al.* [129] collected 4253 instances from LeetCode across C++, Java, and Python and constructed DebugBench, covering syntax, reference, logic, and multiple error cases. As focus shifted to more complex settings, Jimenez *et al.* [130] introduced SWE-Bench, which is directly derived from GitHub issues and their corresponding pull requests. It requires LLMs to generate patches to modify the codebase, with correctness verified against real test cases. Chen *et al.* [131] and Liu *et al.* [132] introduced RepoBugs and RepoDebug, with realistic repository-level errors and multi-task debugging data, highlighting the importance of cross-file dependencies and context extraction methods. Lee *et al.* [133] proposed GitHub Recent Bugs (GHRB), which collects real-world Java bugs from most stars Java repositories in GitHub. Rando *et al.* [134] introduced LongCodeBench, targeting code comprehension and repair with a context window of up to 1M tokens, revealing the bottlenecks of long-context models in real-world scenarios. Similarly, Hariharan *et al.* [135] injected adversarial errors into real-world GitHub repositories, constructing multi-dimensional tasks ranging from local to system-level reasoning to stress-test the repair capabilities of LLM agents. These benchmarks highlight the necessity of agentic approaches that integrate planning, decomposition, and tool use.

As the software ecosystem diversifies, benchmarks are also increasingly targeting multilingual and multimodal tasks. Zan *et al.* [136] extended SWE-bench [130] to 8 PLs, including Python, Java, TypeScript, JavaScript, Go, Rust, C and C++, enabling cross-ecosystem comparisons. Their study revealed that current LLMs perform well on Python-related issues but demonstrate limited generalization capability across other PLs. Guo *et al.* [137] introduced a benchmark OmniGIRL including not only textual but also multimodal information such as

images in the issue descriptions, providing a multilingual, multimodal, and multi-domain evaluation setting.

In real-world deployment scenarios, benchmarks increasingly emphasize security. Wang *et al.* [138] proposed an evaluation framework CVE-Bench collecting 509 common vulnerabilities and exposures across four PLs to evaluate LLM-based agents' abilities in a realistic vulnerability-repairing scenarios. Targeting security engineering, Lee *et al.* [139] developed SEC-Bench for proof-of-concept (PoC) generation and vulnerability patching tasks. They found that the current models can only achieve a success rate of at most 18% and 34% in these two tasks, illustrating the complexity of security-related repair. While Nie *et al.* [117] provided over 5.9k samples across 44 CWE-based risk categories in SECCODEPLT, supporting evaluations of secure code generation, vulnerability detection, and patching.

D. Other benchmarks

This section reviews benchmark studies that may not directly target code output, but focus on aspects that are also important for real-world software development, such as code understanding, dependency installation, and collaborative development.

Beyond code generation, code understanding and reasoning are also important tasks in software development. Gu *et al.* [140] and Xu *et al.* [141] proposed CRUXEval and CRUXEval-X, respectively, assessing the reasoning capability via input/output prediction tasks. While, Zhao *et al.* [142] introduced CodeJudge-Eval, which evaluates whether LLMs can assess code to evaluate their reasoning capability. To address the gap in long-code understanding, Li *et al.* [154] proposed LONGCODEU, which tests four aspects of comprehension, including code unit perception, intra-code unit understanding, intercode unit relation understanding, and long code documentation understanding. Recently, He *et al.* [143] introduced TF-Bench, which focused on program semantics reasoning tasks leveraging type inference in a formal natural deduction system, System F. Xie *et al.* [144] with over 12k verified instances, targets static analysis capabilities.

Targeting test generation, Mundler *et al.* [145] developed SWT-bench, which collects popular Python repositories requiring test generation aligned with user issues. Wang *et al.* [146] proposed TestEval, with a focus on generating test cases that cover specific program lines, branches, or paths. Building upon SWE-Bench [130], Jain *et al.* [147] developed TestGenEval, a large-scale dataset containing over 60,000 test cases from GitHub repositories.

Targeting code refactoring, Gautam *et al.* [148] built RefactorBench, a benchmark with multi-file refactoring tasks that requires LLMs to manage cross-files dependencies and follow natural language instructions. Shetty *et al.* [149] introduced optimization tasks into evaluation by constructing GSO, a benchmark comprising 102 instances from 10 codebases. The benchmark requires models to generate patches improving runtime efficiency while maintaining functional correctness, with results compared against expert developer optimizations. For dependency analysis, Zhang *et al.* [150] introduced DI-Bench to evaluate dependency inference ability of LLMs,

while Milliken *et al.* [151] proposed Installamic to evaluate repository-level environment setup. Addressing real-world collaborative scenarios, Guo *et al.* [152] created SynBench to evaluate ability of agents to resolve collaboration conflicts and maintain consistency in asynchronous multi-contributor environments.

Overall, benchmarks for software engineering have evolved from function-level evaluations to repository-level and real-world settings, progressively emphasizing robustness, multilinguality, and practical usability. Table II provides a comparison of representative datasets for three SE tasks at function- and repository-level.

VI. APPLICATIONS

Artificial intelligence, particularly the advent of Large Language Models (LLMs), has catalyzed a paradigm shift across the software engineering landscape, introducing powerful new capabilities for automation and assistance. These AI-driven applications are not confined to a single activity but span the entire software development lifecycle (SDLC), fundamentally reshaping how developers design, write, debug, and maintain code. To systematically present these diverse applications, this section is structured along the sequential phases of the SDLC, illustrating the application of artificial intelligence from the initial stages of requirements analysis and architectural design, through code generation and quality improvement, and extending to testing, verification, and security. The taxonomy of applications is detailed in Table III.

A. Requirements Analysis & Software Design

At the inception of the software lifecycle, AI models are being employed to bridge the gap between human intent and technical specification, refining ambiguous requirements and shaping high-level architectural strategy. In the domain of *Requirement Understanding & Refinement*, research focuses on transforming abstract inputs, such as user stories or feature requests from a product backlog, directly into functional code, thereby minimizing manual translation [16]. To achieve this, some systems can automatically “purify” and structure vague user requests into complete, actionable descriptions for developers [18]. Other approaches create interactive agents that can proactively ask clarifying questions to resolve ambiguities in user instructions before proceeding with implementation [27]. This capability can be embedded within larger multi-agent frameworks that simulate agile development roles, such as a product owner, to autonomously handle the entire cycle from requirement grooming to final delivery [54].

Following requirements gathering, AI is also being applied to *High-level Solution & Architectural Design*. A key insight in this area is that generating a high-level plan or design before writing code significantly improves outcomes. This “solution design” phase allows models to first outline a repair strategy in natural language or pseudo-code, mirroring human expert behavior [59]. This planning stage can be interactive, allowing human engineers to guide and correct an agent’s high-level plan before code is generated [73]. For new projects, models can first generate the entire repository structure and

TABLE III: Taxonomy of Applications for Software Engineering

Category	Target	Methods	Related Works
Analysis & Design	Refine ambiguous requirements and formulate high-level architectural strategies	Transform abstract inputs into functional code, utilize interactive agents to clarify user instructions, and generate high-level plans before coding	PACGBI [16], [18], DRCodePilot [59], Clarigen [27], AgileCoder [54], [17], HULA [73], Codes [25], PatcheR [44]
Code Generation & Transformation	Automate the creation and modification of source code	Enhance IDE tools with context-aware suggestions, generate complete code blocks from instructions, and automate large-scale code migrations and modernizations	TGen [14], [26], Codes [25], [45], COLLMS [83], MuSL [6], Codellaborator [80], ExploraCoder [63], [74] ReLoc [155], AutoCodeRover [70], OpenHands-Versa [90], MAGIS [53], AgileCoder [54], EG-CFG [156], Acerason [157], ORPS [48], HULA [73], MaintainCoder [113], Alphaverus [66], [79], [38]
Bug Repair & Debugging	Pinpoint, reproduce, and automatically fix bugs in the codebase	Localize faults based on bug reports, automatically generate test cases to reproduce errors, and employ autonomous agents for bug resolution	BLAZE [37], AEGIS [85], CodeV [81], SWE-agent [87], PATCHAGENT [7], SynFix [23], ChatDBG [29], CORTEXA [78], AEGIS [85], LingmaAgent [64], MORRepair [36], AutoCodeRover [70], MAGIS [53], ROSE [75]
Refactoring & Improvement	Proactively enhance code quality and optimize performance	Automatically rewrite code for efficiency, improve code maintainability and readability through dataflow analysis, and address technical debt	Artemis [56], [39], Afterburner [158], LLMDFA [58], MaintainCoder [113], MACEDON [28], SWE-GPT [30], MuSL [6], PATCHAGENT [7]
Testing & Security	Ensure code correctness, robustness, and security against vulnerabilities	Automate test generation and execution, detect and enhance code security through vulnerability analysis, and utilize formal verification for provably correct code in critical systems	ExecutionAgent [62], CBR [50], LLM4TDG [24], [79], Wedge [159], BLAST [89], SAGA [5], PtTrust [41], LPO [52], Alphaverus [66], PoPilot [34], LLMDFA [58], CGM [42], REAL [160], PurpCode [161], PatchPilot [68]

file skeletons as a “multi-layer sketch” before populating the implementation details [25]. Furthermore, AI can analyze existing codebases to identify architectural “smells”—such as cyclic dependencies or excessive coupling—and propose appropriate refactoring strategies [17]. This capability could be integrated into CI/CD pipelines as an “architecture guardian” to prevent design flaws or used by a “commander” agent to coordinate complex, multi-component bug fixes [17], [44].

B. Code Generation & Transformation

Once requirements and designs are established, AI’s role transitions to the automated generation and transformation of source code, one of its most mature application areas. In *Code Completion & Assisted Writing*, models enhance IDE tools with more contextually aware suggestions. This includes “fill-in-the-middle” (FIM) capabilities, where the model predicts missing code segments within an existing block [38]. To improve accuracy, research has focused on pretraining models for project-level completion by using code graph structures to better understand repository-wide context [45]. This enables proactive assistants that not only complete code but also suggest refactorings or warn of potential errors [80].

Beyond mere completion, AI excels at *Functional Code Synthesis*, where complete blocks of code or even entire applications are generated from instructions. A prominent paradigm is test-driven development (TDD), where models generate functional code that satisfies a set of pre-written unit tests [14]. The scope of generation has expanded significantly, with long-context models capable of implementing complex, cross-cutting features at the repository level [26], and some systems aiming to generate a complete, functional code repository from a single natural language sentence [25]. This is often accomplished by autonomous agents that can handle end-to-end software evolution tasks, sometimes organized into multi-agent frameworks that simulate agile development

methodologies to tackle complex requirements [53], [54], [70]. To handle novel problems, these agents can learn to use unseen APIs by exploring documentation [63] or browse the web to find solutions for configuration errors and library usage [90]. The generation process itself is improved through various techniques, such as iterative self-correction based on execution feedback [155], [156], optimizing reasoning with reinforcement learning [48], [157], and interacting with humans to clarify instructions [27], [73]. For specialized domains, research is focused on generating code with specific properties, such as high maintainability to accommodate dynamic requirements [113], or generating provably correct code for safety-critical systems through simulation-guided refinement or by using formal proofs as an intermediate step [66], [79].

Another function is *Code Transformation & Migration*, where AI automates the often tedious process of updating or translating codebases. This includes large-scale migrations, where human-AI partnerships can handle the bulk of repetitive work, such as upgrading a project to a new framework version [74]. This is also applied to software modernization, such as migrating a legacy system to a new cloud platform by coordinating LLMs with platform-specific knowledge [83]. More granularly, AI tools can perform sound static analysis to safely migrate code from deprecated I/O APIs to modern equivalents or automatically parallelize sequential C/C++ code into high-performance CUDA code to leverage modern hardware [6]. The potential for this technology extends to automating library upgrades and other code-base modernizations [70].

C. Bug Repair & Debugging

Beyond initial code creation, AI offers significant support in the critical phase of bug repair and debugging. A crucial first step is *Bug Localization & Reproduction*. Models can pinpoint the exact location of a fault within a large codebase based on a natural language bug report, even across different languages

and projects [37]. This localization can be enhanced by providing agents with better repository exploration capabilities or by improving their ability to identify the precise “pain points” that need modification [78]. Once a location is identified, other systems automatically reproduce the bug by generating an executable test case that triggers the failure [85]. This process can be enhanced by incorporating multimodal data; for instance, models can leverage screenshots from a bug report to better understand and reproduce UI-related issues [81].

With a bug localized, the focus shifts to *Automated Program Repair* (APR). Here, autonomous agents are designed to resolve software engineering tasks end-to-end, fixing real-world issues in GitHub repositories [87]. These agents often mimic human workflows by analyzing the repository for context, examining the code’s dependency graph to ensure patch correctness, and even considering multiple objectives like correctness and code quality [23], [36], [64]. Advanced agents can handle the full lifecycle of a fix, from creating a new code branch to generating the appropriate patch and writing a commit message, demonstrating the potential for fully autonomous software maintenance [53], [70].

Complementing fully automated repair, AI is also enhancing *Interactive Debugging Assistance*. Tools like ChatDBG augment traditional debuggers with a conversational interface, allowing a developer to query the program state in natural language (e.g., “why is x null?”) after a crash [29]. Similarly, IDE-integrated frameworks like ROSE bring the developer “into the loop,” allowing them to interact with the repair tool to provide feedback and guide the generation of solutions, making the debugging process more intuitive and efficient [75].

D. Code Refactoring & Quality Improvement

Maintaining software quality extends beyond fixing bugs, leading to AI applications in proactive code refactoring and optimization. A key area of focus is *Code Performance Optimization*, where AI systems automatically rewrite code to be more efficient. This can be achieved through multi-LLM frameworks that collaborate to improve performance [56] or by systems that specialize in optimizing high-level scripts for data warehouses [39]. More advanced frameworks utilize reinforcement learning, allowing a model to continuously improve its optimization ability by interacting with the execution environment and learning from the performance outcomes of its changes [158]. This also includes translating sequential code to high-performance parallel code, a direct method for performance enhancement [6].

Alongside performance, AI is being used for *Improving Code Maintainability & Readability*. Models can perform complex dataflow analysis to help developers understand code by answering natural language queries like “Where does this variable’s value come from?” [58]. Other approaches focus on generating code that is inherently more maintainable and adaptable to future changes in requirements [113]. This capability can be delivered directly within the IDE, where plugins provide real-time, multi-dimensional feedback on code quality—assessing readability, complexity, and performance—and proactively suggest improvements [28]. Broader,

process-centric models aim for holistic software improvement, capable of handling tasks from refactoring to implementing new features [30]. This includes agents designed to address technical debt by identifying and fixing “bad smells” accumulated in the codebase [7].

E. Testing, Verification, & Security

To ensure correctness and robustness, AI is increasingly applied to software testing, verification, and security. In *Automated Test Generation & Execution*, agents are being developed to execute tests for arbitrary projects without needing manual configuration, simplifying CI/CD setup [62]. These tools can automatically generate functional test scripts, create complex test data that satisfies specific logical constraints, and synthesize performance test cases to evaluate code efficiency [24], [50], [159]. A particularly powerful application is the automated generation of tests that reproduce a specific issue from a bug report, which is invaluable for regression testing [89]. Some frameworks even unify code and test generation, pushing towards a model of self-verification where the LLM is also responsible for generating challenging tests for its own code [5].

For *Code Security Enhancement & Vulnerability Detection*, research is exploring multiple defensive layers. One approach is to build risk assessment frameworks that leverage a model’s internal states to provide an early warning if it is about to generate insecure code [41]. Another is to use program analysis to detect potential vulnerabilities; for instance, dataflow analysis can trace the flow of tainted data [58], and graph-based models can detect defects at the repository level [42]. Proactively, models can be explicitly taught secure coding practices through specialized fine-tuning with feedback from static analysis tools, or prompted to perform explicit security reasoning before generating code to reduce the likelihood of introducing common vulnerabilities [52], [160], [161].

The most stringent level of software quality is addressed by *Formal Verification & Safety Assurance*. Here, AI is being developed to generate code that is provably correct, a critical need in safety-critical domains. Systems like Alphaverus can bootstrap the generation of formally verified code by using formal proofs as an intermediate representation to guide code synthesis, even in domains where training data is scarce [34], [66]. This verification-aware approach is also being explored in automated program repair to increase the reliability of generated patches [68]. For dynamic systems such as autonomous driving software, a simulation-guided approach is used, where AI-generated code is rigorously tested within a high-fidelity simulator so that its safety and reliability could be ensured before deployment [79].

VII. CHALLENGES AND FUTURE DIRECTIONS

While the integration of LLMs has catalyzed significant advances in software engineering, the path toward fully autonomous and reliable agentic systems is fraught with fundamental challenges. These obstacles, spanning scalability, evaluation, and responsible deployment, are not merely incremental hurdles but rather define the critical research frontiers

for the field. This section outlines these principal challenges and, in turn, proposes corresponding future research directions poised to overcome them, shaping the next generation of software engineering AI.

A. Scalability: From Token Limits to Hierarchical Cognition

A primary challenge is one of scale, where the architectural constraints of current LLMs prevent meaningful engagement with real-world software complexity. While context windows have expanded, they remain insufficient for enterprise systems where codebases routinely exceed millions of lines of code. This limitation manifests as project amnesia, where agents lose track of high-level architectural patterns and design conventions during extended tasks. Current mitigation strategies, such as vector database retrieval, provide only partial solutions, as they fetch isolated snippets of context without comprehending the intricate relationships that define a software architecture. At this scale, the computational economics of iterative refinement also become prohibitive, hindering the application of agents to complex, system-wide tasks.

To overcome these scalability limitations, future research must pivot from expanding linear context windows to developing hierarchical cognitive architectures for code understanding. This necessitates a paradigm shift from processing code as a flat sequence of tokens to reasoning over structured, multimodal representations like Code Property Graphs (CPGs). Such architectures must be coupled with advanced tiered memory systems, such as combining a volatile short-term memory for immediate context with a persistent long-term memory storing architectural knowledge and project conventions. By learning to navigate these compressed, structured representations, agents can transcend token limits to reason about system-wide properties, marking a crucial transition from agents that merely write code to agents that truly comprehend software.

B. Evaluation: From Functional Correctness to Production Readiness

The field currently faces an evaluation crisis, where prevailing benchmarks and metrics incentivize superficial correctness over the qualities that define production-ready code. An over-reliance on pass@k on synthetic, function-level datasets like HumanEval creates a significant reality gap, failing to measure critical non-functional requirements such as security, performance, and maintainability. Consequently, agents are optimized to hack the benchmark rather than to produce robust, high-quality software. Recent analyses reveal that a majority of algorithmically successful solutions on benchmarks like SWE-Bench would fail in production due to the introduction of new security flaws, performance regressions, or violations of coding standards. Without evaluation frameworks that mirror real software lifecycle concerns, progress remains illusory.

Addressing this evaluation gap requires a move toward holistic, lifecycle-aware benchmarking. The next generation of evaluation frameworks should be built around dynamic, digital twin environments that simulate the entire software development lifecycle. Within these sandboxed environments,

agents must be assessed on a suite of production-oriented criteria, including the introduction of security vulnerabilities, performance regressions, and the accumulation of technical debt. Furthermore, evaluation must incorporate economic and operational metrics, such as the computational cost of a solution and the human effort required for review. Developing such comprehensive benchmarks will close the gap between academic research and industrial practice, ensuring that progress is measured by the delivery of truly production-ready software.

C. Deployment: From Ethical Quagmires to Responsible Collaboration

The deployment of autonomous agents introduces a host of ethical, legal, and resource-related challenges that create an accountability vacuum and threaten sustainable adoption. Intellectual property frameworks are unprepared to resolve attribution for code co-created by humans, open-source libraries, and AI, creating significant legal uncertainty. Operationally, the exponential scaling of computational costs presents an unsustainable model, while the workforce faces potential deskilling. Most critically, when AI-generated code fails, the question of liability is diffused among the developer who prompted the agent, the organization that deployed it, and the creators of the model, creating a trilemma that obstructs adoption in mission-critical industries.

Navigating these challenges requires a concerted research effort toward responsible, sustainable, and collaborative AI systems. To establish accountability, agents must be built with inherent mechanisms for transparency, such as the ability to generate verifiable audit trails that explain design rationale and trace code to its sources. To ensure sustainability, research must prioritize resource-frugal agent architectures and cost-aware planning algorithms. Finally, the paradigm must shift from pure automation to human-centric augmentation, framing AI as a scaffolding tool that empowers and upskills developers. By designing agents as accountable, sustainable, and collaborative partners, we can build the foundation of trust required for their ethical and widespread adoption in the software engineering ecosystem.

VIII. CHALLENGES AND FUTURE DIRECTIONS

While the integration of LLMs has catalyzed significant advances in software engineering, the path toward fully autonomous and reliable agentic systems is fraught with fundamental challenges. These obstacles are not merely incremental hurdles but rather define the critical research frontiers for the field. This section outlines these principal challenges and proposes corresponding future research directions poised to overcome them, shaping the next generation of software engineering AI.

A. Scalability: From Token Limits to Hierarchical Cognition

A primary challenge is one of scale, where the architectural constraints of current LLMs prevent meaningful engagement with real-world software complexity. While context windows

have expanded to millions of tokens, they remain insufficient for enterprise systems where codebases routinely exceed tens of millions of lines of code distributed across thousands of modules. This limitation manifests as *project amnesia*, where agents lose track of high-level architectural patterns, design conventions, and cross-module dependencies during extended tasks. Current mitigation strategies, such as vector database retrieval and code summarization, provide only partial solutions, as they fetch isolated snippets of context without comprehending the intricate relationships that define a software architecture. At this scale, the computational economics of iterative refinement also become prohibitive, hindering the application of agents to complex, system-wide tasks that may require dozens of edit-compile-test cycles.

To overcome these scalability limitations, future research must pivot from expanding linear context windows to developing *hierarchical cognitive architectures* for code understanding. This necessitates a paradigm shift from processing code as a flat sequence of tokens to reasoning over structured, multi-modal representations such as Abstract Syntax Trees (ASTs), Control Flow Graphs (CFGs), and Code Property Graphs (CPGs). Such architectures must be coupled with advanced tiered memory systems, combining a volatile short-term memory for immediate context with a persistent long-term memory storing architectural knowledge, design patterns, and project conventions. Promising directions include graph neural networks for repository-level code representation, hierarchical attention mechanisms that operate at multiple levels of abstraction, and neuro-symbolic approaches that integrate symbolic reasoning with neural code generation. By learning to navigate these compressed, structured representations, agents can transcend token limits to reason about system-wide properties such as modularity, coupling, and architectural integrity, marking a crucial transition from agents that merely write code to agents that truly comprehend software systems.

B. Evaluation: From Functional Correctness to Production Readiness

The field currently faces an evaluation crisis, where prevailing benchmarks and metrics incentivize superficial correctness over the qualities that define production-ready code. An over-reliance on pass@k metrics on synthetic, function-level datasets like HumanEval creates a significant *reality gap*, failing to measure critical non-functional requirements such as security, performance, maintainability, and adherence to coding standards. Consequently, agents are optimized to maximize benchmark performance rather than to produce robust, high-quality software suitable for deployment in production environments. Recent analyses reveal that a majority of algorithmically correct solutions on benchmarks like SWE-Bench would fail code review in production due to the introduction of security vulnerabilities, performance regressions, violations of coding standards, or accumulation of technical debt. Furthermore, existing benchmarks often use static test suites that can be gamed through memorization or superficial pattern matching, rather than measuring genuine problem-solving capability. Without evaluation frameworks that mirror

real software lifecycle concerns and incorporate adversarial testing, progress remains illusory.

Addressing this evaluation gap requires a move toward *holistic, lifecycle-aware benchmarking* that captures the full spectrum of software quality attributes. The next generation of evaluation frameworks should be built around dynamic, digital twin environments that simulate the entire software development lifecycle, from requirements elicitation through deployment and maintenance. Within these sandboxed environments, agents must be assessed on a comprehensive suite of production-oriented criteria, including the absence of security vulnerabilities (e.g., CWE compliance), performance characteristics (e.g., time and space complexity), code quality metrics (e.g., maintainability index, cyclomatic complexity), and alignment with project-specific coding standards. Furthermore, evaluation must incorporate economic and operational metrics, such as the computational cost of generating a solution, the human effort required for review and integration, and the long-term maintenance burden. Promising directions include adversarial test generation to assess robustness, mutation testing to evaluate test suite adequacy, and longitudinal studies that track code evolution over time. Developing such comprehensive benchmarks, potentially through partnerships with industry to access real production codebases and issue trackers, will close the gap between academic research and industrial practice, ensuring that progress is measured by the delivery of truly production-ready software.

C. Domain Adaptation: From General Patterns to Specialized Expertise

A critical yet underexplored challenge lies in the domain adaptation problem, where agents trained primarily on public repositories struggle to generalize to specialized domains with distinct programming paradigms, legacy systems, or proprietary frameworks. While current LLMs demonstrate impressive performance on mainstream languages like Python and JavaScript, they exhibit significant degradation when confronted with domain-specific languages (DSLs), legacy codebases written in COBOL or Fortran, or highly specialized domains such as embedded systems, high-performance computing, or formal verification. This limitation stems from the imbalanced distribution of training data, where popular languages and frameworks dominate the corpus while niche but critical domains remain severely underrepresented. Furthermore, different software domains embody fundamentally different design philosophies: real-time systems prioritize deterministic timing over code elegance, safety-critical systems demand formal verification over rapid prototyping, and high-performance computing requires deep hardware awareness absent in web development. Agents that fail to internalize these domain-specific constraints and idioms produce code that, while syntactically correct, violates fundamental domain principles and is unsuitable for deployment.

Overcoming the domain adaptation challenge requires developing *domain-aware agent architectures* that can rapidly specialize to new programming contexts with minimal supervision. Future research should explore few-shot and zero-shot

domain adaptation techniques, enabling agents to quickly internalize domain-specific patterns from limited examples, documentation, or expert demonstrations. This may involve meta-learning approaches that train agents to adapt quickly across diverse programming paradigms, transfer learning strategies that leverage knowledge from related domains, or retrieval-augmented generation systems that dynamically incorporate domain-specific knowledge from curated repositories. For legacy system modernization, a particularly promising direction is the development of *hybrid neuro-symbolic agents* that combine neural code generation with symbolic reasoning engines to enforce hard constraints derived from domain specifications. Additionally, building high-quality, domain-specific datasets and benchmarks for underrepresented areas, such as embedded systems programming or scientific computing, will be essential to drive progress. Ultimately, the goal is to develop agents that are not merely polyglot programmers but true domain specialists capable of understanding and respecting the unique constraints, idioms, and quality standards of each software engineering context.

D. Multi-Agent Coordination: From Individual Capability to Collective Intelligence

As software engineering tasks grow in complexity, spanning multiple subsystems, programming languages, and architectural layers, the limitations of single-agent systems become increasingly apparent. While individual agents may excel at localized tasks such as function generation or bug fixing, they struggle with system-level challenges that require coordinated reasoning across multiple components, such as refactoring a distributed system, implementing a cross-cutting feature, or resolving conflicts between concurrent modifications. The emerging paradigm of multi-agent systems, where specialized agents collaborate to tackle complex tasks, introduces a new class of coordination challenges. These include establishing effective communication protocols between agents with different specializations, preventing redundant or conflicting modifications when multiple agents edit the same codebase, managing dependencies and task ordering when subtasks have complex precedence constraints, and allocating computational resources efficiently across the agent collective. Current multi-agent frameworks often rely on simple coordination mechanisms such as sequential pipelines or centralized orchestration, which fail to capture the dynamic, iterative, and often non-linear nature of real software development workflows.

Advancing multi-agent software engineering requires developing *sophisticated coordination mechanisms* that enable true collective intelligence. Future research should explore decentralized coordination protocols inspired by distributed systems and multi-agent reinforcement learning, allowing agents to negotiate task allocation, resolve conflicts, and dynamically reorganize based on task requirements without centralized control. This includes developing shared memory architectures and communication standards that enable agents to maintain consistent views of the codebase and project state, as well as conflict resolution algorithms that can automatically merge or arbitrate between competing code modifications. A particularly

promising direction is the integration of formal methods for coordination, such as using temporal logic to specify and verify coordination protocols or employing distributed consensus algorithms to ensure consistency. Additionally, research should investigate role specialization strategies that go beyond simple task-based division to create agents with complementary cognitive capabilities, such as pairing a creative code generator with a rigorous verifier, or combining a bottom-up implementer with a top-down architect. To evaluate these systems, new benchmarks are needed that measure not just task success but coordination efficiency, communication overhead, and the ability to gracefully handle dynamic task requirements. By developing agents that can truly collaborate rather than merely coexist, we can tackle the scale and complexity of modern software systems that exceed the capabilities of any individual agent.

E. Continuous Learning: From Static Models to Evolving Expertise

A fundamental limitation of current LLM-based agents is their static nature: once deployed, they remain frozen in time, unable to learn from new technologies, evolving best practices, or project-specific feedback. This creates a *knowledge drift problem*, where agents trained on historical data gradually become obsolete as programming languages introduce new features, frameworks release breaking changes, and community best practices evolve. For instance, an agent trained before the introduction of `async/await` syntax in JavaScript or the major architectural changes in React Hooks would produce outdated code patterns. Moreover, agents cannot currently learn from their own mistakes or incorporate human feedback at scale, missing crucial opportunities for improvement through interaction. This static learning paradigm is particularly problematic in rapidly evolving domains such as web development, where new frameworks and best practices emerge constantly, and in specialized enterprise contexts, where agents must adapt to proprietary codebases, internal libraries, and organization-specific coding standards. Without continuous learning capabilities, agents require expensive and frequent retraining cycles, limiting their long-term utility and cost-effectiveness.

Addressing this challenge requires developing *self-evolving agent architectures* with continuous learning capabilities that enable adaptation to new contexts without catastrophic forgetting. Future research should explore online learning techniques that allow agents to incrementally update their knowledge from user corrections, code review feedback, and execution outcomes, while preserving core competencies. This includes developing memory-efficient parameter update strategies such as adapter layers or low-rank adaptation (LoRA), which enable selective fine-tuning without full model retraining. A particularly promising direction is *reinforcement learning from human feedback* (RLHF) in the software engineering loop, where agents continuously improve by learning from developer interactions, code review outcomes, and post-deployment performance metrics. Additionally, agents should maintain versioned knowledge bases that track the evolution of programming language features, library APIs, and best practices over time, enabling them to generate code appropriate

for specific language versions or project configurations. For organization-specific adaptation, research should investigate federated learning approaches that allow agents to learn from proprietary codebases while preserving intellectual property and privacy. Ultimately, the goal is to develop agents that behave more like human developers who continuously learn throughout their careers, rather than static tools that become obsolete and require replacement.

F. Responsible Deployment: From Ethical Quagmires to Trustworthy Collaboration

The deployment of autonomous agents introduces a host of ethical, legal, and societal challenges that create an accountability vacuum and threaten sustainable adoption. Intellectual property frameworks are unprepared to resolve attribution for code co-created by humans, open-source libraries, and AI, creating significant legal uncertainty that has already manifested in high-profile lawsuits. When AI-generated code inadvertently incorporates copyrighted snippets or violates software licenses, determining liability becomes a complex trilemma involving the developer who prompted the agent, the organization that deployed it, and the creators of the underlying model. Operationally, the exponential scaling of computational costs for advanced agents presents an unsustainable model that risks creating a two-tiered ecosystem where only well-resourced organizations can afford cutting-edge AI assistance. Most critically, the workforce faces potential deskilling as developers increasingly rely on AI for tasks that were previously learning opportunities, raising concerns about the long-term health of the software engineering profession. Additionally, the opacity of LLM decision-making creates trust barriers in safety-critical domains where code must be auditable and explainable. These multifaceted challenges, if left unaddressed, risk undermining the transformative potential of AI in software engineering.

Navigating these challenges requires a concerted research effort toward *responsible, sustainable, and trustworthy AI systems* that prioritize transparency, accountability, and human empowerment. To establish accountability, agents must be built with inherent mechanisms for transparency, such as the ability to generate verifiable audit trails that explain design rationale, cite sources for generated code, and trace decisions to specific training data or retrieval sources. This includes developing explainable AI techniques that can provide human-understandable justifications for code suggestions, as well as provenance tracking systems that document the lineage of generated code to facilitate license compliance and attribution. To ensure sustainability, research must prioritize resource-frugal agent architectures, such as mixture-of-experts models that selectively activate subsets of parameters, and cost-aware planning algorithms that balance solution quality against computational expense. For workforce considerations, the paradigm must shift from pure automation to *human-centric augmentation*, framing AI as scaffolding that empowers and upskills developers rather than replaces them. This involves designing agents that provide educational feedback explaining their reasoning, deliberately expose users to underlying concepts rather than hiding complexity, and adapt their level of

assistance based on developer expertise. Furthermore, establishing industry standards and governance frameworks for AI-generated code, including mandatory disclosure requirements, liability frameworks, and ethical guidelines, will be essential for building societal trust. By designing agents as accountable, sustainable, and collaborative partners that enhance rather than diminish human capability, we can build the foundation of trust required for their ethical and widespread adoption in the software engineering ecosystem.

IX. CONCLUSION

This survey has presented the first comprehensive analysis connecting benchmarks and solutions in LLM-empowered software engineering, addressing a critical gap in understanding how evaluation methodologies align with solution approaches. Through systematic analysis of 150+ papers, we have demonstrated the field's evolution from simple prompt engineering to sophisticated agentic systems incorporating planning, reasoning, memory, and tool augmentation. Our unified taxonomy and pipeline provide researchers with a roadmap for selecting optimal approaches across varying task complexities. The identification of 50+ benchmarks and their corresponding solution strategies establishes a foundation for standardized evaluation and comparison. Looking forward, the emergence of multi-agent collaboration, self-evolving systems, and formal verification integration promises to further revolutionize software engineering practices. This work serves as an essential resource for advancing LLM-based software engineering toward more autonomous, reliable, and intelligent systems that can tackle increasingly real-world challenges.

REFERENCES

- [1] A. W. Services, "Multi-agent code generation in amazon q developer," *AWS AI Blog*, January 2025.
- [2] C. Liang, H. Wang, and Y. Zhang, "Opendevin: An open-source environment for evaluating code generation agents," in *NeurIPS*, vol. 37, 2024.
- [3] X. Liu, J. Yang, and R. Martinez, "Efficient resource management for production code generation agents," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 9, no. 1, pp. 1–25, 2025.
- [4] L. Zhang and D. Robertson, "The economics of ai code generation: Total cost of ownership analysis," *TOSEM*, vol. 34, no. 2, pp. 1–30, 2025.
- [5] Z. Ma, T. Zhang, M. Cao, J. Liu, W. Zhang, M. Luo, S. Zhang, and K. Chen, "Rethinking verification for llm code generation: From generation to testing," 2025.
- [6] C. Ke, R. Zhang, S. Wang, L. Ding, G. Li, Y. Wen, S. Zhang, R. Xu, J. Qin, J. Guo *et al.*, "Mutual-supervised learning for sequential-to-parallel code translation," 2025.
- [7] Z. Yu, Z. Guo, Y. Wu, J. Yu, M. Xu, D. Mu, Y. Chen, and X. Xing, "Patchagent: a practical program repair agent mimicking human expertise," in *USENIX Conference on Security Symposium*, ser. SEC '25. USENIX Association, 2025.
- [8] A. L. Correa, C. M. Werner, and G. Zaverucha, "Object oriented design expertise reuse: An approach based on heuristics, design patterns and anti-patterns," in *International Conference on Software Reuse*. Springer, 2000, pp. 336–352.
- [9] Q. Zhang, C. Fang, Y. Ma, W. Sun, and Z. Chen, "A survey of learning-based automated program repair," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 2, pp. 1–69, 2023.
- [10] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, "A survey on large language models for code generation," *arXiv preprint arXiv:2406.00515*, 2024.

- [11] H. Wang, J. Gong, H. Zhang, J. Xu, and Z. Wang, "Ai agentic programming: A survey of techniques, challenges, and opportunities," *arXiv preprint arXiv:2508.11126*, 2025.
- [12] Y. Dong, X. Jiang, J. Qian, T. Wang, K. Zhang, Z. Jin, and G. Li, "A survey on code generation with llm-based agents," *arXiv preprint arXiv:2508.00083*, 2025.
- [13] R. Sapkota, K. I. Roumeliotis, and M. Karkee, "Vibe coding vs. agentic coding: Fundamentals and practical implications of agentic ai," *arXiv preprint arXiv:2505.19443*, 2025.
- [14] N. S. Mathews and M. Nagappan, "Test-driven development and llm-based code generation," in *ASE*, 2024, pp. 1583–1594.
- [15] Y. Fu, B. Li, L. Li, W. Zhang, and T. Xie, "The first prompt counts the most! an evaluation of large language models on iterative example-based code generation," *PACMSE*, vol. 2, no. ISSTA, pp. 1583–1606, 2025.
- [16] M. Sarschar, G. Zhang, and A. Nowak, "Pacgb: A pipeline for automated code generation from backlog items," in *ASE*, 2024, pp. 2338–2341.
- [17] G. Pandini, A. Martini, A. N. Videsjorden, and F. A. Fontana, "An exploratory study on architectural smell refactoring using large languages models," in *IEEE ICSA-C*. IEEE, 2025, pp. 462–471.
- [18] P. KC, R. Ghandiparsi, T. Herron, J. Heaps, and M. B. Hosseini, "Demystifying feature requests: Leveraging llms to refine feature requests in open-source software," in *Requirements Engineering*, 2025.
- [19] J. Cheng, F. Liu, C. Wu, and L. Zhang, "Adaptivelilm: A framework for selecting optimal cost-efficient llm for code-generation based on cot length," in *Internetware*. Association for Computing Machinery, 2025.
- [20] R.-B. Liu, A. Li, C. Yang, H. Sun, and M. Li, "Revisiting chain-of-thought in code generation: Do language models need to learn reasoning before coding?" in *ICML*.
- [21] Z. Zhao and F. Fard, "Do current language models support code intelligence for r programming language?" *TOSEM*, 2024.
- [22] M. Kondo, D. Kawahara, and T. Kurabayashi, "Improving repository-level code search with text conversion," in *NAACL*, 2024, pp. 130–137.
- [23] X. Tang, J. Gao, J. Xu, T. Sun, Y. Song, S. Ezzini, W. C. Ouedraogo, J. Klein, and T. F. Bissyandé, "Synfix: Dependency-aware program repair via relationgraph analysis," in *ACL*, 2025, pp. 4878–4894.
- [24] J. Liu, R. Liang, X. Zhu, Y. Zhang, Y. Liu, and Q. Liu, "Llm4tdg: test-driven generation of large language models based on enhanced constraint reasoning," *Cybersecurity*, vol. 8, no. 1, p. 32, 2025.
- [25] D. Zan, A. Yu, W. Liu, D. Chen, B. Shen, Y. Yao, W. Li, X. Chen, Y. Gong, B. Guan *et al.*, "Codes: Natural language to code repository via multi-layer sketch," *TOSEM*, 2024.
- [26] Y. PENG, Z. Z. Wang, and D. Fried, "Can long-context language models solve repository-level code generation?" in *LTI Student Research Symposium* 2025.
- [27] C. Miao, Y. Wang, L. He, L. Fang, and P. S. Yu, "Clarigen: Bridging instruction gaps via interactive clarification in code generation," in *AAAI 2025 Workshop on PDLM*, 2025.
- [28] X. Liu, Y. You, X. Qiu, T. Ma, and J. Zhao, "Macedon: Supporting programmers with real-time multi-dimensional code evaluation and optimization," in *ACM UIST*, 2025, pp. 1–17.
- [29] K. H. Levin, N. van Kempen, E. D. Berger, and S. N. Freund, "Chatdbg: Augmenting debugging with large language models," *PACMSE*, vol. 2, no. FSE, pp. 1892–1913, 2025.
- [30] Y. Ma, R. Cao, Y. Cao, Y. Zhang, J. Chen, Y. Liu, Y. Liu, B. Li, F. Huang, and Y. Li, "Swe-gpt: A process-centric language model for automated software improvement," *PACMSE*, vol. 2, no. ISSTA, pp. 2362–2383, 2025.
- [31] U. Piterbarg, K. Gandhi, L. Pinto, N. Goodman, and R. Fergus, "D3: A dataset for training code lms to act diff-by-diff," in *Second Conference on Language Modeling*.
- [32] J. Pan, X. Wang, G. Neubig, N. Jaitly, H. Ji, A. Suhr, and Y. Zhang, "Training software engineering agents and verifiers with SWE-gym," in *ICML*, 2025.
- [33] H. Wang, Z. Hou, Y. Wei, J. Tang, and Y. Dong, "Swe-dev: Building software engineering agents with training and inference scaling," pp. 3742–3761, 2025.
- [34] D. Zhang, J. Wang, and T. Sun, "Building A proof-oriented programmer that is 64% better than gpt-4o under data scarcity," pp. 23 101–23 118, 2025.
- [35] F. Cassano, L. Li, A. Sethi, N. Shinn, A. Brennan-Jones, J. Ginesin, E. Berman, G. Chakhnashvili, A. Lozhkov, C. J. Anderson, and A. Guha, "Can it edit? evaluating the ability of large language models to follow code editing instructions," in *First Conference on Language Modeling*, 2024.
- [36] B. Yang, H. Tian, J. Ren, H. Zhang, J. Klein, T. Bissyande, C. Le Goues, and S. Jin, "Morepair: Teaching llms to repair code via multi-objective fine-tuning," *TOSEM*.
- [37] P. Chakraborty, M. Alfaedel, and M. Nagappan, "Blaze: Cross-language and cross-project bug localization via dynamic chunking and hard example learning," *TSE*, 2025.
- [38] H. Sagtani, R. Mehrotra, and B. Liu, "Improving fim code completions via context & curriculum based learning," in *WSDM*, 2025, pp. 801–810.
- [39] A. Shypula, A. Madaan, Y. Zeng, U. Alon, J. Gardner, M. Hashemi, G. Neubig, P. Ranganathan, O. Bastani, and A. Yazdanbakhsh, "Automated high-level code optimization for warehouse performance," *IEEE Micro*, 2025.
- [40] Y. Chen, Y. Ye, Z. Li, Y. Ma, and C. Gao, "Smaller but better: Self-paced knowledge distillation for lightweight yet effective lcms," *PACMSE*, vol. 2, no. FSE, pp. 3057–3080, 2025.
- [41] Y. Huang, L. Ma, K. Nishikino, and T. Akazaki, "Risk assessment framework for code llms via leveraging internal states," in *FSE*, 2025, pp. 432–443.
- [42] H. Tao, Y. Zhang, Z. Tang, H. Peng, X. Zhu, B. Liu, Y. Yang, Z. Zhang, Z. Xu, H. Zhang *et al.*, "Code graph model (cgm): A graph-integrated large language model for repository-level software engineering tasks," 2025.
- [43] J. Wang, X. Xie, Q. Hu, S. Liu, and Y. Li, "Do code semantics help? a comprehensive study on execution trace-based information for code large language models," in *EMNLP*, 2025.
- [44] Y. Tang, H. Li, K. Zhu, M. Yang, Y. Ding, and W. Guo, "Copatcher: Collaborative software patching with component (s)-specific small reasoning models," 2025.
- [45] M. Sapronov and E. Glukhov, "On pretraining for project-level code completion," in *ICLR 2025 Third Workshop on Deep Learning for Code*, 2025.
- [46] D. Zhang, S. Zhoubian, Z. Hu, Y. Yue, Y. Dong, and J. Tang, "Rest-mcts*: Llm self-training via process reward guided tree search," *NeurIPS*, vol. 37, pp. 64 735–64 772, 2024.
- [47] K. Zhang, H. Zhang, G. Li, J. You, J. Li, Y. Zhao, and Z. Jin, "Sealign: Alignment training for software engineering agent," *ICSE*, 2026.
- [48] Z. Yu, W. Gu, Y. Wang, X. Jiang, Z. Zeng, J. Wang, W. Ye, and S. Zhang, "Reasoning through execution: Unifying process and outcome rewards for code generation."
- [49] K. Team, Y. Bai, Y. Bao, G. Chen, J. Chen, N. Chen, R. Chen, Y. Chen, Y. Chen, Y. Chen *et al.*, "Kimi k2: Open agentic intelligence," *arXiv preprint arXiv:2507.20534*, 2025.
- [50] S. Guo, H. Liu, X. Chen, Y. Xie, L. Zhang, T. Han, H. Chen, Y. Chang, and J. Wang, "Optimizing case-based reasoning system for functional test script generation with large language models," in *ACM SIGKDD*, 2025, pp. 4487–4498.
- [51] Y. Wei, F. Cassano, J. Liu, Y. Ding, N. Jain, Z. Mueller, H. de Vries, L. Von Werra, A. Guha, and L. Zhang, "Selfcodealign: Self-alignment for code generation," *NeurIPS*, vol. 37, pp. 62 787–62 874, 2024.
- [52] M. S. Hasan, S. Chakraborty, S. Karmaker, and N. Balasubramanian, "Teaching an old LLM secure coding: Localized preference optimization on distilled preferences," in *ACL*. Association for Computational Linguistics, 2025, pp. 26 039–26 057.
- [53] W. Tao, Y. Zhou, Y. Wang, W. Zhang, H. Zhang, and Y. Cheng, "Magis: Llm-based multi-agent framework for github issue resolution," *NeurIPS*, vol. 37, pp. 51 963–51 993, 2024.
- [54] M. H. Nguyen, T. P. Chau, P. X. Nguyen, and N. D. Bui, "Agilecoder: Dynamic collaborative agents for software development based on agile methodology," in *FORGE*. IEEE, 2025, pp. 156–167.
- [55] N. Wadhwa, A. Sonwane, D. Arora, A. Mehrotra, S. Utpala, R. B. Bairi, A. Kanade, and N. Natarajan, "Masai: Modular architecture for software-engineering ai agents," in *NeurIPS 2024 Workshop on Open-World Agents*, 2024.
- [56] R. Giavrimis, M. Basios, F. Wu, L. Kanthan, and R. Bauer, "Artemis ai: Multi-llm framework for code optimisation," in *CAI*. IEEE, 2025, pp. 1–6.
- [57] S. Xiao, Z. Lin, W. Gao, H. Chen, and Y. Zhang, "Long context scaling: Divide and conquer via multi-agent question-driven collaboration," 2025.
- [58] C. Wang, W. Zhang, Z. Su, X. Xu, X. Xie, and X. Zhang, "Llmdfa: analyzing dataflow in code with large language models," *NeurIPS*, vol. 37, pp. 131 545–131 574, 2024.
- [59] J. Zhao, D. Yang, L. Zhang, X. Lian, Z. Yang, and F. Liu, "Enhancing automated program repair with solution design," in *ASE*, 2024, pp. 1706–1718.

- [60] Y. Liu, P. Gao, X. Wang, J. Liu, Y. Shi, Z. Zhang, and C. Peng, “Marscode agent: Ai-native automated bug fixing,” *arXiv preprint arXiv:2409.00899*, 2024.
- [61] M. Shetty, Y. Chen, G. Somashekhar, M. Ma, Y. Simmhan, X. Zhang, J. Mace, D. Vandevenoede, P. Las-Casas, S. M. Gupta *et al.*, “Building ai agents for autonomous clouds: Challenges and design principles,” in *SoCC*, 2024, pp. 99–110.
- [62] I. Bouzenia and M. Pradel, “You name it, i run it: An llm agent to execute tests of arbitrary projects,” *PACMSE*, vol. 2, no. ISSTA, pp. 1054–1076, 2025.
- [63] Y. Wang, Y. Zhang, Z. Qin, C. Zhi, B. Li, F. Huang, Y. Li, and S. Deng, “Exploracoder: Advancing code generation for multiple unseen apis via planning and chained exploration,” pp. 18 124–18 145, 2025.
- [64] Y. MA and Y. Liu, “Improving automated issue resolution via comprehensive repository exploration,” in *ICLR 2025 Third Workshop on Deep Learning for Code*.
- [65] Y. Ma, Q. Yang, R. Cao, B. Li, F. Huang, and Y. Li, “How to understand whole software repository,” *arXiv preprint arXiv:2406.01422*, 2024.
- [66] P. Aggarwal, B. Parno, and S. Welleck, “Alphaverus: Bootstrapping formally verified code generation through self-improving translation and treefinement,” in *ICML*, 2025.
- [67] C. S. Xia, Y. Deng, S. Dunn, and L. Zhang, “Demystifying llm-based software engineering agents,” *PACMSE*, vol. 2, no. FSE, pp. 801–824, 2025.
- [68] H. Li, Y. Tang, S. Wang, and W. Guo, “Patchpilot: A cost-efficient software engineering agent with early attempts on formal verification,” in *ICML*.
- [69] W. Epperson, G. Bansal, V. C. Dibia, A. Fourney, J. Gerrits, E. Zhu, and S. Amershi, “Interactive debugging and steering of multi-agent ai systems,” in *CHI*, 2025, pp. 1–15.
- [70] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury, “Autocoderover: Autonomous program improvement,” in *ISSTA*, 2024, pp. 1592–1604.
- [71] R. Hu, C. Peng, X. Wang, and C. Gao, “An llm-based agent for reliable docker environment configuration,” *NeurIPS*, 2025.
- [72] Y. Wang, M. Pradel, and Z. Liu, “Are “solved issues” in sweepbench really solved correctly? an empirical study,” *arXiv preprint arXiv:2503.15223*, 2025.
- [73] W. Takerngaksiri, J. Pasuksmit, P. Thongtanunam, C. Tantithamthavorn, R. Zhang, F. Jiang, J. Li, E. Cook, K. Chen, and M. Wu, “Human-in-the-loop software development agents,” in *ICSE-SEIP*. IEEE, 2025, pp. 342–352.
- [74] B. Omidvar Tehrani, I. M., and A. Anubhai, “Evaluating human-ai partnership for llm-based code migration,” in *CHI EA*, 2024, pp. 1–8.
- [75] S. P. Reiss, X. Wei, J. Yuan, and Q. Xin, “Rose: An ide-based interactive repair framework for debugging,” *TOSEM*, vol. 34, no. 4, pp. 1–39, 2025.
- [76] E. Zelikman, E. Lorch, L. Mackey, and A. T. Kalai, “Self-taught optimizer (stop): Recursively self-improving code generation,” in *First Conference on Language Modeling*, 2024.
- [77] S. Dou, H. Jia, S. Wu, H. Zheng, W. Zhou, M. Wu, M. Chai, J. Fan, C. Huang, Y. Tao *et al.*, “What’s wrong with your code generated by large language models? an extensive study,” *arXiv preprint arXiv:2407.06153*, 2024.
- [78] A. Sohrabizadeh, J. Song, M. Liu, R. Roy, C. Lee, J. Raiman, and B. Catanzaro, “Nemotron-cortexa: Enhancing llm agents for software engineering tasks via improved localization and solution diversity,” in *ICML*.
- [79] A. Nouri, J. Andersson, K. D. J. Hornig, Z. Fei, E. Knabe, H. Sivencrea, B. Cabrero-Daniel, and C. Berger, “On simulation-guided llm-based code generation for safe autonomous driving software,” in *EASE*, 2025.
- [80] K. Pu, D. Lazaro, I. Arawjo, H. Xia, Z. Xiao, T. Grossman, and Y. Chen, “Assistance or disruption? exploring and evaluating the design and trade-offs of proactive ai programming support,” in *CHI*, 2025, pp. 1–21.
- [81] L. Zhang, D. Zan, Q. Yang, Z. Huang, D. Chen, B. Shen, T. Liu, Y. Gong, H. Pengjie, X. Lu, G. Liang, L. Cui, and Q. Wang, “Codev: Issue resolving with visual data,” in *ACL*, 2025, pp. 7350–7361.
- [82] Y. Ma, Q. Yang, R. Cao, B. Li, F. Huang, and Y. Li, “Alibaba lingmaagent: Improving automated issue resolution via comprehensive repository exploration,” in *FSE*, 2025, pp. 238–249.
- [83] H.-L. Truong, M. Vukovic, and R. Pavuluri, “On coordinating llms and platform knowledge for software modernization and new developments,” in *SSE*. IEEE, 2024, pp. 188–193.
- [84] R. Ehsani, S. Pathak, and P. Chatterjee, “Towards detecting prompt knowledge gaps for improved llm-guided issue resolution,” in *MSR*. IEEE, 2025, pp. 699–711.
- [85] X. Wang, P. Gao, X. Meng, C. Peng, R. Hu, Y. Lin, and C. Gao, “Aegis: An agent-based framework for bug reproduction from issue descriptions,” in *FSE*, 2025, pp. 331–342.
- [86] H. Su, S. Jiang, Y. Lai, H. Wu, B. Shi, C. Liu, Q. Liu, and T. Yu, “Evor: Evolving retrieval for code generation,” in *EMNLP*, 2024, pp. 2538–2554.
- [87] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press, “Swe-agent: Agent-computer interfaces enable automated software engineering,” vol. 37, 2024, pp. 50 528–50 652.
- [88] Z. Li and M. Izadi, “Enhancing human-ide interaction in the sdlc using llm-based mediator agents,” in *FSE*, 2025, pp. 1363–1367.
- [89] K. Kitsios, M. Castelluccio, and A. Baccelli, “Automated generation of issue-reproducing tests by combining llms and search-based testing,” in *ASE*, 2025.
- [90] A. B. Soni, B. Li, X. Wang, V. Chen, and G. Neubig, “Coding agents with multimodal browsing are generalist problem solvers,” in *ICML 2025 Workshop on Computer Use Agents*.
- [91] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, “Evaluating large language models trained on code,” *arXiv preprint arXiv:2107.03374*, 2021.
- [92] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le *et al.*, “Program synthesis with large language models,” *arXiv preprint arXiv:2108.07732*, 2021.
- [93] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. GONG, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. LIU, “Codexglue: A machine learning benchmark dataset for code understanding and generation,” in *NeurIPS*, J. Vanschoren and S. Yeung, Eds., vol. 1, 2021.
- [94] R. Puri, D. Kung, G. Janssen, W. Zhang, G. Domeniconi, V. Zolotov, J. T. Dolby, J. Chen, M. Choudhury, L. Decker, V. Thost, V. Thost, L. Buratti, S. Pujar, S. Ramji, U. Finkler, S. Malaika, and F. Reiss, “Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks,” in *NeurIPS*, J. Vanschoren and S. Yeung, Eds., vol. 1, 2021.
- [95] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, “Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation,” in *NeurIPS*, ser. NIPS ’23. Curran Associates Inc., 2023.
- [96] L. Chai, S. Liu, J. Yang, Y. Yin, JinKe, J. Liu, T. Sun, G. Zhang, C. Ren, H. Guo, N. Wang, B. Wang, X. Wu, B. Wang, T. Li, L. Yang, S. Duan, Z. Zhang, and Z. Li, “Mceval: Massively multilingual code evaluation,” in *ICLR*, 2025.
- [97] Z. Yu, Y. Zhao, A. Cohan, and X.-P. Zhang, “HumanEval pro and MBPP pro: Evaluating large language models on self-invoking code generation task,” in *Findings of the Association for Computational Linguistics: ACL 2025*, W. Che, J. Nabende, E. Shutova, and M. T. Pilehvar, Eds. Association for Computational Linguistics, Jul. 2025, pp. 13 253–13 279.
- [98] W. Yan, H. Liu, Y. Wang, Y. Li, Q. Chen, W. Wang, T. Lin, W. Zhao, L. Zhu, H. Sundaram, and S. Deng, “CodeScope: An execution-based multilingual multitask multidimensional benchmark for evaluating LLMs on code understanding and generation,” in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, L.-W. Ku, A. Martins, and V. Srikumar, Eds. Association for Computational Linguistics, Aug. 2024, pp. 5511–5558.
- [99] T. Y. Zhuo, V. M. Chien, J. Chim, H. Hu, W. Yu, R. Widayarsi, I. N. B. Yusuf, H. Zhan, J. He, I. Paul, S. Brunner, C. GONG, J. Hoang, A. R. Zebaze, X. Hong, W.-D. Li, J. Kaddour, M. Xu, Z. Zhang, P. Yadav, N. Jain, A. Gu, Z. Cheng, J. Liu, Q. Liu, Z. Wang, D. Lo, B. Hui, N. Muennighoff, D. Fried, X. Du, H. de Vries, and L. V. Werra, “Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions,” in *ICLR*, 2025.
- [100] X. Du, M. Liu, K. Wang, H. Wang, J. Liu, Y. Chen, J. Feng, C. Sha, X. Peng, and Y. Lou, “Evaluating large language models in class-level code generation,” in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE ’24. Association for Computing Machinery, 2024.
- [101] L. Gong, S. Wang, M. Elhoushi, and A. Cheung, “Evaluation of LLMs on syntax-aware code fill-in-the-middle tasks,” in *Forty-first International Conference on Machine Learning*, 2024.
- [102] Y. Qing, B. Zhu, M. Du, Z. Guo, T. Y. Zhuo, Q. Zhang, J. M. Zhang, H. Cui, S.-M. Yu, D. Huang *et al.*, “Effibench-x: A multi-language benchmark for measuring efficiency of llm-generated code,” in *NeurIPS*, 2025.

- [103] A. Thakur, J. Lee, G. Tsoukalas, M. Sistla, M. Zhao, S. Zetsche, G. Durrett, Y. Yue, and S. Chaudhuri, “Clever: A curated benchmark for formally verified code generation,” in *NeurIPS*, 2025.
- [104] K. Vergopoulos, M. N. Mueller, and M. Vechev, “Automated benchmark generation for repository-level coding tasks,” in *Forty-second International Conference on Machine Learning*, 2025.
- [105] J. Li, G. Li, Y. Zhao, Y. Li, H. Liu, H. Zhu, L. Wang, K. Liu, Z. Fang, L. Wang, J. Ding, X. Zhang, Y. Zhu, Y. Dong, Z. Jin, B. Li, F. Huang, Y. Li, B. Gu, and M. Yang, “DevEval: A manually-annotated code generation benchmark aligned with real-world code repositories,” in *Findings of the Association for Computational Linguistics: ACL 2024*, L.-W. Ku, A. Martins, and V. Srikumar, Eds. Association for Computational Linguistics, Aug. 2024, pp. 3603–3614.
- [106] S. Liang, N. Jiang, Y. Hu, and L. Tan, “Can language models replace programmers for coding? REPOCOD says ‘not yet’,” in *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, W. Che, J. Nabende, E. Shutova, and M. T. Pilehvar, Eds. Association for Computational Linguistics, Jul. 2025, pp. 24698–24717.
- [107] D. Zheng, Y. Wang, E. Shi, R. Zhang, Y. Ma, H. Zhang, and Z. Zheng, “Humanevo: An evolution-aware benchmark for more realistic evaluation of repository-level code generation,” in *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*, ser. ICSE ’25. IEEE Press, 2025, p. 1372–1384.
- [108] N. Jain, M. Shetty, T. Zhang, K. Han, K. Sen, and I. Stoica, “R2e: Turning any github repository into a programming agent environment,” in *ICML*, 2024.
- [109] J. Yang, K. Lieret, C. E. Jimenez, A. Wettig, K. Khandpur, Y. Zhang, B. Hui, O. Press, L. Schmidt, and D. Yang, “Swe-smith: Scaling data for software engineering agents,” in *NeurIPS*, 2025.
- [110] I. Badertdinov, A. Golubev, M. Nekrashevich, A. Shevtsov, S. Karasik, A. Andriushchenko, M. Trofimova, D. Litvinseva, and B. Yangel, “SWE-rebench: An automated pipeline for task collection and de-contaminated evaluation of software engineering agents,” in *NeurIPS*, 2025.
- [111] S. Chen, P. Pusarla, and B. Ray, “Dycodeeval: Dynamic benchmarking of reasoning capabilities in code large language models under data contamination,” in *Forty-second International Conference on Machine Learning*, 2025.
- [112] J. J. Wu and F. H. Fard, “Humanevalcomm: Benchmarking the communication competence of code generation for llms and llm agents,” *ACM Trans. Softw. Eng. Methodol.*, vol. 34, no. 7, Aug. 2025.
- [113] Z. Wang, R. Ling, C. Wang, Y. Yu, S. Wang, Z. Li, F. Xiong, and W. Zhang, “Maintaincoder: Maintainable code generation under dynamic requirements,” 2025.
- [114] Q. Zheng, X. Xia, X. Zou, Y. Dong, S. Wang, Y. Xue, L. Shen, Z. Wang, A. Wang, Y. Li, T. Su, Z. Yang, and J. Tang, “Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x,” in *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, ser. KDD ’23. Association for Computing Machinery, 2023, p. 5673–5684.
- [115] M. A. M. Khan, M. S. Bari, X. L. Do, W. Wang, M. R. Parvez, and S. Joty, “XCodeEval: An execution-based large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval,” in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, L.-W. Ku, A. Martins, and V. Srikumar, Eds. Association for Computational Linguistics, Aug. 2024, pp. 6766–6805.
- [116] C. Niu, C. Li, V. Ng, and B. Luo, “Crosscodebench: Benchmarking cross-task generalization of source code models,” in *Proceedings of the 45th International Conference on Software Engineering*, ser. ICSE ’23. IEEE Press, 2023, p. 537–549.
- [117] Y. Yang, Y. Nie, Z. Wang, Y. Tang, W. Guo, B. Li, and D. Song, “Seccodepl: A unified platform for evaluating the security of code genai,” in *NeurIPS*, 2025.
- [118] W. U. Ahmad, M. G. R. Tushar, S. Chakraborty, and K.-W. Chang, “AVATAR: A parallel corpus for Java-python program translation,” in *Findings of the Association for Computational Linguistics: ACL 2023*, A. Rogers, J. Boyd-Graber, and N. Okazaki, Eds. Association for Computational Linguistics, Jul. 2023, pp. 2268–2281.
- [119] W. Yan, Y. Tian, Y. Li, Q. Chen, and W. Wang, “CodeTransOcean: A comprehensive multilingual benchmark for code translation,” in *Findings of the Association for Computational Linguistics: EMNLP 2023*, H. Bouamor, J. Pino, and K. Bali, Eds. Association for Computational Linguistics, Dec. 2023, pp. 5067–5089.
- [120] X. Du, M. Liu, K. Wang, H. Wang, J. Liu, Y. Chen, J. Feng, C. Sha, X. Peng, and Y. Lou, “Evaluating large language models in class-level code generation,” in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE ’24. Association for Computing Machinery, 2024.
- [121] A. Khatri, R. Zhang, J. Pan, Z. Wang, Q. Chen, G. Durrett, and I. Dillig, “CRUST-bench: A comprehensive benchmark for c-to-safe-rust transpilation,” in *Second Conference on Language Modeling*, 2025.
- [122] G. Ou, M. Liu, Y. Chen, X. Peng, and Z. Zheng, “Repository-level code translation benchmark targeting rust,” *arXiv preprint arXiv:2411.13990*, 2024.
- [123] Y. Wang, Y. Wang, S. Wang, D. Guo, J. Chen, J. Grundy, X. Liu, Y. Ma, M. Mao, H. Zhang et al., “Repotransbench: A real-world benchmark for repository-level code translation,” *arXiv preprint arXiv:2412.17744*, 2024.
- [124] A. R. Ibrahimzada, K. Ke, M. Pawagi, M. S. Abid, R. Pan, S. Sinha, and R. Jabbarvand, “Alphatrans: A neuro-symbolic compositional approach for repository-level code translation and validation,” *Proc. ACM Softw. Eng.*, vol. 2, no. FSE, Jun. 2025.
- [125] L. Zhang, B. Wang, J. Wang, X. Zhao, M. Zhang, H. Yang, M. Zhang, Y. Li, J. Li, J. Yu, and M. Zhang, “Function-to-style guidance of LLMs for code translation,” in *ICML*, 2025.
- [126] J. Guo, Z. Li, X. Liu, K. Ma, T. Zheng, Z. Yu, D. Pan, Y. Li, R. Liu, Y. Wang, S. Guo, X. Qu, X. Yue, G. Zhang, W. Chen, and J. Fu, “Codeeditorbench: Evaluating code editing capability of LLMs,” in *ICLR 2025 Third Workshop on Deep Learning for Code*, 2025.
- [127] M. Jiao, T. Yu, X. Li, G. Qiu, X. Gu, and B. Shen, “On the Evaluation of Neural Code Translation: Taxonomy and Benchmark,” in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, Sep. 2023, pp. 1529–1541.
- [128] Q. Tao, T. Yu, X. Gu, and B. Shen, “Unraveling the potential of large language models in code translation: How far are we?” in *2024 31st Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2024, pp. 353–362.
- [129] R. Tian, Y. Ye, Y. Qin, X. Cong, Y. Lin, Y. Pan, Y. Wu, H. Haotian, L. Weichuan, Z. Liu, and M. Sun, “DebugBench: Evaluating debugging capability of large language models,” in *Findings of the Association for Computational Linguistics: ACL 2024*, L.-W. Ku, A. Martins, and V. Srikumar, Eds. Association for Computational Linguistics, Aug. 2024, pp. 4173–4198.
- [130] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. R. Narasimhan, “SWE-bench: Can language models resolve real-world github issues?” in *ILCR*, 2024.
- [131] Y. Chen, J. Wu, X. Ling, C. Li, Z. Rui, T. Luo, and Y. Wu, “When large language models confront repository-level automatic program repair: How well they done?” in *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, ser. ICSE-Companion ’24. Association for Computing Machinery, 2024, p. 459–471.
- [132] J. Liu, Z. Liu, Z. Cheng, M. He, X. Shi, Y. Guo, X. Zhu, Y. Guo, Y. Wang, and H. Wang, “Reposebug: Repository-level multi-task and multi-language debugging evaluation of large language models,” in *EMNLP*, 2025.
- [133] J. Y. Lee, S. Kang, J. Yoon, and S. Yoo, “The github recent bugs dataset for evaluating llm-based debugging applications,” in *ICST*, 2024, pp. 442–444.
- [134] S. Rando, L. Romani, A. Sampieri, L. Franco, J. Yang, Y. Kyuragi, F. Galasso, and T. Hashimoto, “Longcodebench: Evaluating coding LLMs at 1m context windows,” in *Second Conference on Language Modeling*, 2025.
- [135] K. Hariharan, U. Girit, Z. Wang, and J. Andreas, “Breakpoint: Stress-testing systems-level reasoning in LLM agents,” in *Second Conference on Language Modeling*, 2025.
- [136] M.-S. bench: A Multilingual Benchmark for Issue Resolving, “Multi-swe-bench: A multilingual benchmark for issue resolving,” in *NeurIPS*, 2025.
- [137] L. Guo, W. Tao, R. Jiang, Y. Wang, J. Chen, X. Liu, Y. Ma, M. Mao, H. Zhang, and Z. Zheng, “Omnigirl: A multilingual and multimodal benchmark for github issue resolution,” *Proc. ACM Softw. Eng.*, vol. 2, no. ISSTA, 2025.
- [138] P. Wang, X. Liu, and C. Xiao, “Cve-bench: Benchmarking LLM-based software engineering agent’s ability to repair real-world CVE vulnerabilities,” in *NAACL*, 2025, pp. 4207–4224.
- [139] H. Lee, Z. Zhang, H. Lu, and L. Zhang, “Sec-bench: Automated benchmarking of llm agents on real-world software security tasks,” in *NeurIPS*, 2025.
- [140] A. Gu, B. Roziere, H. J. Leather, A. Solar-Lezama, G. Synnaeve, and S. Wang, “CRUXEval: A benchmark for code reasoning, understanding

- and execution,” in *Proceedings of the 41st International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, R. Salakhutdinov, Z. Kolter, K. Heller, A. Weller, N. Oliver, J. Scarlett, and F. Berkenkamp, Eds., vol. 235. PMLR, 21–27 Jul 2024, pp. 16 568–16 621.
- [141] R. Xu, J. Cao, Y. Lu, M. Wen, H. Lin, X. Han, B. He, S.-C. Cheung, and L. Sun, “CRUXEVAL-X: A benchmark for multilingual code reasoning, understanding and execution,” in *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, W. Che, J. Nabende, E. Shutova, and M. T. Pilehvar, Eds. Association for Computational Linguistics, Jul. 2025, pp. 23 762–23 779.
- [142] Y. Zhao, Z. Luo, Y. Tian, H. Lin, W. Yan, A. Li, and J. Ma, “CodeJudge-eval: Can large language models be good judges in code understanding?” in *Proceedings of the 31st International Conference on Computational Linguistics*, O. Rambow, L. Wanner, M. Apidianaki, H. Al-Khalifa, B. D. Eugenio, and S. Schockaert, Eds. Association for Computational Linguistics, Jan. 2025, pp. 73–95.
- [143] Y. He, L. Yang, C. C. G. Gonzalo, and H. Chen, “Evaluating program semantics reasoning with type inference in system f,” in *NeurIPS*, 2025.
- [144] D. Xie, M. Zheng, X. Liu, J. Wang, C. Wang, L. Tan, and X. Zhang, “Core: Benchmarking llms’ code reasoning capabilities through static analysis tasks,” in *NeurIPS*, 2025.
- [145] N. Mündler, M. N. Mueller, J. He, and M. Vechev, “SWT-bench: Testing and validating real-world bug-fixes with code agents,” in *NeurIPS*, 2024.
- [146] W. Wang, C. Yang, Z. Wang, Y. Huang, Z. Chu, D. Song, L. Zhang, A. R. Chen, and L. Ma, “TestEval: Benchmarking large language models for test case generation,” in *Findings of the Association for Computational Linguistics: NAACL 2025*, L. Chiruzzo, A. Ritter, and L. Wang, Eds. Association for Computational Linguistics, Apr. 2025, pp. 3547–3562.
- [147] K. Jain, G. Synnaeve, and B. Roziere, “Testgeneval: A real world unit test generation and test completion benchmark,” in *ILCR*, 2025.
- [148] D. Gautam, S. Garg, J. Jang, N. Sundaresan, and R. Z. Moghadam, “Refactorbench: Evaluating stateful reasoning in language agents through code,” in *ILCR*, 2025.
- [149] M. Shetty, N. Jain, J. Liu, V. Kethanaboyina, K. Sen, and I. Stoica, “Gso: Challenging software optimization tasks for evaluating swimmers,” in *NeurIPS*, 2025.
- [150] L. Zhang, J. Wang, S. He, C. Zhang, Y. Kang, B. Li, J. Wen, C. Xie, M. Wang, Y. Huang, E. Nallipogu, Q. Lin, Y. Dang, S. Rajmohan, D. Zhang, and Q. Zhang, “DI-BENCH: Benchmarking large language models on dependency inference with testable repositories at scale,” in *Findings of the Association for Computational Linguistics: ACL 2025*, W. Che, J. Nabende, E. Shutova, and M. T. Pilehvar, Eds. Association for Computational Linguistics, Jul. 2025, pp. 10 134–10 153.
- [151] L. Milliken, S. Kang, and S. Yoo, “Beyond pip Install: Evaluating LLM Agents for the Automated Installation of Python Projects ,” in *SANER*. IEEE Computer Society, Mar. 2025, pp. 1–11.
- [152] X. Guo, X. Wang, Y. Chen, S. Li, C. Han, M. Li, and H. Ji, “Syncmind: Measuring agent out-of-sync recovery in collaborative software engineering,” in *Forty-second International Conference on Machine Learning*, 2025.
- [153] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, “Bleu: a method for automatic evaluation of machine translation,” in *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, P. Isabelle, E. Charniak, and D. Lin, Eds. Association for Computational Linguistics, Jul. 2002, pp. 311–318.
- [154] J. Li, X. Guo, L. Li, K. Zhang, G. Li, J. Li, Z. Tao, F. Liu, C. Tao, Y. Zhu, and Z. Jin, “Benchmarking long-context language models on long code understanding,” in *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, W. Che, J. Nabende, E. Shutova, and M. T. Pilehvar, Eds. Association for Computational Linguistics, Jul. 2025, pp. 27 309–27 327.
- [155] Z. Lyu, J. Huang, Y. Deng, S. Hoi, and B. An, “Let’s revise step-by-step: A unified local search framework for code generation with llms,” 2025.
- [156] B. Lavon, S. Katz, and L. Wolf, “Execution guided line-by-line code generation,” 2025.
- [157] Y. Chen, Z. Yang, Z. Liu, C. Lee, P. Xu, M. Shoeybi, B. Catanzaro, and W. Ping, “Acereason-nemotron: Advancing math and code reasoning through reinforcement learning,” in *NeurIPS*, 2025.
- [158] M. Du, L. A. Tuan, Y. Liu, Y. Qing, D. Huang, X. He, Q. Liu, Z. Ma, and S.-k. Ng, “Afterburner: Reinforcement learning facilitates self-improving code efficiency optimization,” 2025.
- [159] J. Yang, C.-C. Wang, B. A. Stoica, and K. Pei, “Synthesizing performance constraints for evaluating and improving code efficiency,” in *NeurIPS*, 2025.
- [160] F. Yao, Z. Wang, L. Liu, J. Cui, L. Zhong, X. Fu, H. Mai, V. Krishnan, J. Gao, and J. Shang, “Training language models to generate quality code with program analysis feedback,” 2025.
- [161] J. Liu, N. Diwan, Z. Wang, H. Zhai, X. Zhou, K. A. Nguyen, T. Yu, M. Wahed, Y. Deng, H. Benkraouda *et al.*, “Purpcode: Reasoning for safer code generation,” 2025.