From Code Generation to Code Reasoning: A Survey of Inference-Time Methods in LLM-Based Code Generation

Anonymous ACL submission

Abstract

Large language models (LLMs) have rapidly advanced the state of code generation, evolving from prompt-based function synthesis to iterative, execution-guided, and agentic software engineering systems. While recent progress has led to impressive benchmark results, the growing diversity of methods and evaluation settings has also introduced fragmentation and ambiguity in how capabilities are assessed and compared. In this survey, we present a unified view of LLM-based code generation from the perspective of inference-time reasoning and interaction. We organize existing methods according to how LLMs structure generation, incorporate feedback, and interact with execution environments, covering prompt-conditioned generation, language-based self-refinement, execution-guided revision, and agentic code generation. We further review representative benchmarks across function-level, execution-grounded, repository-level, and long-horizon agentic tasks, and analyze how evaluation assumptions shape reported performance. Our analysis highlights fundamental limitations of test-based correctness metrics, risks of data contamination, and emerging challenges in evaluating iterative and agentic systems. By connecting method design choices with evaluation protocols, this survey aims to clarify current progress, expose open problems, and provide guidance for future research on reliable and scalable LLM-based code generation.

1 Introduction

Large language models (LLMs) have become a central paradigm for automated code generation, demonstrating strong performance on tasks ranging from code completion and program synthesis to bug fixing and repository-level software engineering. Early work largely framed code generation as a single-shot conditional generation problem, where a pretrained model directly maps natural language descriptions or partial code context to executable programs, with progress driven primarily by model scale and data coverage. Representative systems include proprietary API models evaluated on function-level benchmarks such as HumanEval (Chen, 2021), as well as open-source code LMs trained on large-scale corpora, including CodeGen (Nijkamp et al., 2022), StarCoder (Li et al., 2023), and Code Llama (Roziere et al., 2023).

Recent advances have shifted attention from model capacity alone to inference-time strategies that structure reasoning, incorporate feedback, and enable interaction with external environments. Instead of producing code in a single pass, modern systems increasingly rely on iterative refinement, feedback signals, and tool-mediated action loops. Language-based self-refinement methods improve initial generations through critique and revision without executing code, such as Self-Refine (Madaan et al., 2023) and Reflexion (Shinn et al., 2023), while consistency- and rule-based verification strategies, including Self-Consistency (Wang et al., 2022) and Constitutional AI (Bai et al., 2022), enhance robustness through diversified sampling and policy-guided critique.

A parallel line of work grounds feedback in program execution. Execution-guided approaches leverage test results, compiler errors, or runtime traces to guide iterative revision. AlphaCode (Li et al., 2022) illustrates the effectiveness of large-scale test-driven candidate generation and selection, while subsequent methods explore finer-grained execution feedback, including compiler- and runtime-error repair (Chen et al., 2023), execution-guided search (Ding et al., 2023), and execution trace augmentation for debugging and program repair (Wang et al., 2025; Haque et al., 2025; Shi et al., 2024). These approaches move verification beyond purely linguistic signals toward semantic grounding in program behavior.

Beyond iterative generation and debugging, re-

cent work increasingly frames code generation as an agentic process embedded in interactive environments. Agentic systems integrate reasoning, acting, and tool use over extended horizons, exemplified by ReAct-style reason–act loops (Yao et al., 2022), explicit tool-use training (Schick et al., 2023), and open agent platforms operating in realistic development environments (Wang et al., 2024). At the repository level, agents are evaluated on long-horizon software engineering tasks, giving rise to benchmarks such as SWE-bench (Jimenez et al., 2023), SWE-bench Pro (Deng et al., 2025), GitTaskBench (Ni et al., 2025), and unified evaluation frameworks like SWE-Compass (Xu et al., 2025). Multi-agent formulations further extend this paradigm by decomposing software development into role-specialized agents (Huang et al., 2023; Hong et al., 2023).

In contrast, evaluation practices have not evolved at the same pace. Function-level benchmarks with unit-test–based metrics remain dominant due to their simplicity and reproducibility, yet they capture only a narrow view of real-world programming behavior. Execution-grounded benchmarks offer stronger semantic validation but remain limited in scope, while repository-level and agentic benchmarks improve realism at the cost of increased variance, implementation sensitivity, and evaluation complexity (Jimenez et al., 2023; Deng et al., 2025). Consequently, performance gains observed under one evaluation setting often fail to translate to others, complicating comparison across methods.

This survey aims to bring coherence to this field by jointly examining method design and evaluation practice from an inference-time perspective. Rather than focusing on model architectures or pretraining strategies, we analyze how LLM-based code generation systems differ in how they structure reasoning, leverage feedback, and interact with execution environments, and review representative benchmarks spanning function-level, execution-grounded, repository-level, and long-horizon agentic settings.

2 Definition and Scope

This survey focuses on *LLM-based code generation*, a research paradigm in which large language models serve as the primary reasoning and decision-making component for programming tasks. To ensure clarity and avoid scope creep, we provide a concise definition and explicitly delineate the boundaries of our discussion.

Definition. We define **LLM-based code generation** as methods that use a pretrained large language model to generate, revise, or repair source code from textual inputs, such as natural-language specifications, partial code context, or both. The defining characteristic of this paradigm is that the model's behavior at inference time is driven predominantly by language-based reasoning, optionally augmented with external feedback signals including execution results, test outcomes, or tool responses. Under this definition, code generation extends beyond standalone function synthesis to encompass tasks such as code completion, bug fixing, test generation, and repository-level code modification, provided that the LLM remains the central component guiding decisions and actions.

Scope of Methods. Our review emphasizes inference-time strategies that enhance code generation without requiring changes to the underlying model architecture or large-scale retraining. We cover three broad classes of approaches: (i) prompt-conditioned generation that produces code directly from textual context, (ii) iterative refinement methods that leverage language-based or execution-based feedback to improve outputs, and (iii) agentic frameworks in which an LLM interacts with tools and environments over multiple steps to accomplish complex programming objectives.

Evaluation and Exclusions. We consider evaluation settings ranging from unit-test–driven benchmarks to repository-level tasks that require iterative debugging and long-horizon reasoning. At the same time, we exclude classical program synthesis techniques not centered on LLMs, detailed model architecture or pretraining studies, and purely non-LLM software engineering tools. This positioning allows us to present a unified, LLM-centered view of code generation methods while keeping the scope focused and coherent.

3 Taxonomy of LLM-based Code Generation Methods

This section presents a literature-driven taxonomy of LLM-based code generation methods, organized by *how large language models are used at inference time* to structure reasoning, incorporate feedback, and interact with external environments. We group prior work into four major families: (i) prompt-conditioned generation, (ii) language-based feedback and verification, (iii) execution-

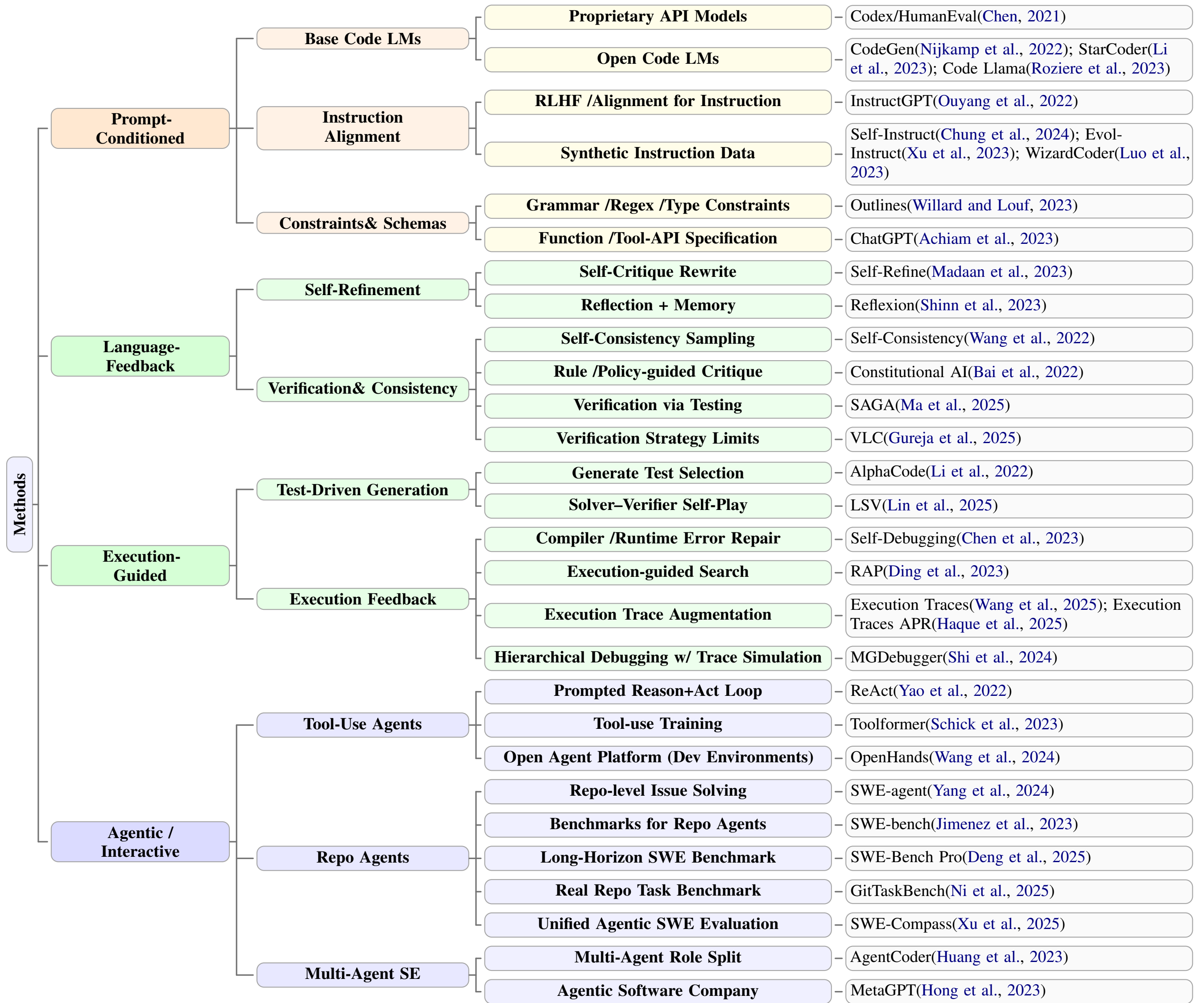

Figure 1: Taxonomy for LLM-based code generation, including verifier-centric verification, execution-trace augmentation, and long-horizon agentic software engineering benchmarks.

guided generation and debugging, and (iv) agentic and environment-interactive systems. For each family, we adopt a unified analytical framework consisting of four components: (1) core idea and paradigm, (2) key design choices, (3) empirical performance and promise, and (4) challenges and limitations. This structure enables systematic comparison across method classes and clarifies the trade-offs underlying different design decisions.

3.1 Prompt-Conditioned Code Generation

Core Idea and Paradigm. Prompt-conditioned methods frame programming as direct conditional generation. Given a natural-language specification, partial code context, or examples, the model produces code in a single forward pass. This family covers both proprietary API models (e.g., Codex-style systems evaluated on HumanEval (Chen, 2021)) and open code language models such as CodeGen (Nijkamp et al., 2022), StarCoder (Li et al., 2023), and Code Llama (Roziere et al., 2023). The underlying assumption is that pretraining and instruction tuning encode sufficient program patterns to map specifications to correct implementations without explicit external feedback.

Key Design Choices. Design primarily centers on (i) prompt construction and context selection, (ii) alignment strategies that improve instruction following, and (iii) constraints that enforce structure. Alignment-based variants incorporate instruction tuning and preference optimization (e.g., InstructGPT-style RLHF (Ouyang et al., 2022)), while data-centric approaches leverage synthetic instruction generation and iterative refinement of

prompts and tasks (Chung et al., 2024; Xu et al., 2023; Luo et al., 2023). To reduce malformed outputs and improve interface adherence, another line of work injects explicit constraints such as grammar/regex/type restrictions (e.g., Outlines (Willard and Louf, 2023)) or tool/function specifications (e.g., ChatGPT-style tool interfaces (Achiam et al., 2023)).

Performance and Promise. Prompt-conditioned generation remains attractive due to its simplicity and low inference cost. It performs strongly on function-level synthesis and completion tasks when benchmarks align with training distributions and when prompts provide adequate scaffolding. In practice, these methods offer good latency-quality trade-offs for interactive assistance, and instruction alignment substantially improves usability and controllability.

Challenges and Limitations. The key limitation is lack of grounded error awareness: the model cannot reliably determine correctness, robustness, or compliance with implicit constraints beyond the prompt. As complexity increases, failures due to underspecified requirements, hidden edge cases, or inconsistent internal decisions become frequent. Structural constraints can improve formatting and reduce syntax errors but do not guarantee semantic correctness, motivating methods that introduce explicit feedback signals.

3.2 Self-Refinement and Language-Based Feedback

Core Idea and Paradigm. Language-based feedback methods extend single-pass generation by introducing an iterative loop where the model critiques and revises its own outputs using natural language feedback. The model plays dual roles: a generator producing candidate code and a critic producing diagnostics (e.g., error hypotheses, missing cases, alternative plans) that guide subsequent revisions. Representative frameworks include Self-Refine (Madaan et al., 2023) and memory-augmented reflection methods such as Reflexion (Shinn et al., 2023).

Key Design Choices. Key choices include (i) what feedback artifact to generate (critique, plan, checklist, counterexamples), (ii) how to incorporate it (inline in the prompt, separated sections, memory buffers), and (iii) the control policy for iteration (fixed rounds vs. stopping heuristics). A complementary direction improves reliability through agreement and verification heuristics, such as sampling-based self-consistency (Wang et al., 2022) or rule/policy-guided critique (Bai et al., 2022). Recent 2025 studies further foreground verification as a central concern and analyze when test-like checking procedures help and when verification strategies exhibit systematic failure modes (Ma et al., 2025; Gureja et al., 2025).

Performance and Promise. Compared to single-pass generation, self-refinement often yields improvements in functional correctness and reasoning quality without additional training. It provides a lightweight form of inference-time scaffolding that can adapt to new tasks via prompting alone, and it can be deployed even when code execution is not available.

Challenges and Limitations. A fundamental limitation is that language-only self-assessment can be unreliable: critiques may be fluent yet incorrect, and refinement can converge to polished but wrong solutions. Iterative loops also exhibit diminishing returns and can be sensitive to prompt format and iteration budgets. Moreover, without grounding in execution, the model may reinforce incorrect assumptions, and verification procedures can be brittle under distribution shift or adversarial edge cases (Gureja et al., 2025).

3.3 Execution-Guided Code Generation

Core Idea and Paradigm. Execution-guided methods incorporate feedback from running the generated program, treating compilation results, runtime errors, and test outcomes as objective signals for revision. Instead of relying solely on linguistic critique, the model interprets observed failures and proposes targeted code changes. This paradigm aligns naturally with test-driven development and debugging workflows.

Key Design Choices. Typical systems alternate between generation and execution in a sandbox. After an initial candidate is produced, the system runs it, collects structured feedback (error messages, failing tests), and conditions the next generation step on this feedback. Some methods emphasize test-centric search and selection (e.g., AlphaCode's generate-and-filter strategy (Li et al., 2022)), while others focus on using error messages for self-debugging (Chen et al., 2023) or integrating execution-guided search procedures (e.g., RAP-style iterative repair/search (Ding et al., 2023)). A notable 2025 trend is to expose richer execution signals such as execution traces or intermediate state summaries, which can improve fault localization

and reduce trial-and-error edits (Wang et al., 2025; Haque et al., 2025; Shi et al., 2024).

Performance and Promise. By grounding revisions in program behavior, execution-guided methods often achieve substantial gains on test-based benchmarks and can correct concrete implementation errors that language-only refinement misses. Richer feedback (e.g., traces) can narrow the gap between natural-language reasoning and program semantics, improving diagnosis for non-trivial logical bugs.

Challenges and Limitations. Execution feedback can still be sparse: a failing test indicates incorrectness but may not reveal root cause. Models may therefore overfit to error messages or patch symptoms. Repeated optimization against a fixed test suite risks overfitting, yielding solutions that pass tests without capturing intended semantics. In addition, repeated execution increases computational overhead and complicates scaling to large code scopes or evolving repositories, while trace-based feedback raises practical questions about summarization, context limits, and evaluation fairness.

3.4 Agentic and Environment-Interactive Code Generation

Core Idea and Paradigm. Agentic methods model code generation as a long-horizon, interactive process. Rather than generating code in isolation, the system decomposes goals into sequences of actions (e.g., searching a repository, editing files, running tests, inspecting outputs) and iteratively updates its plan based on observations. Language functions both as the reasoning substrate and as the interface for tool invocation.

Key Design Choices. Agentic systems differ in (i) the tool/action space, (ii) how state is maintained (conversation history, explicit memory, workspace snapshots), and (iii) how control flow is implemented (scripted controllers vs. model-driven decisions). Classic reason-and-act prompting patterns such as ReAct (Yao et al., 2022) operationalize tool use, while training-based approaches such as Toolformer (Schick et al., 2023) aim to internalize tool invocation behavior. More recent systems provide standardized environments for software agents (e.g., OpenHands (Wang et al., 2024)), and repository-level agents target real issue resolution (e.g., SWE-agent (Yang et al., 2024)). Benchmarking has rapidly evolved from SWE-bench (Jimenez et al., 2023) toward more challenging 2025 evaluations that emphasize long-horizon, realistic repository tasks (Deng et al., 2025; Ni et al., 2025; Xu et al., 2025). Multi-agent frameworks further decompose software development into roles such as planner, implementer, and reviewer (e.g., AgentCoder (Huang et al., 2023), MetaGPT (Hong et al., 2023)).

Performance and Promise. By integrating planning, tool use, execution, and verification, agentic systems can tackle open-ended tasks that exceed the scope of function-level synthesis, including multi-file changes and repository navigation. They represent a practical pathway toward more autonomous development assistants and offer a flexible umbrella for combining prior ideas (self-refinement, execution feedback, verification) within a single interactive loop.

Challenges and Limitations. Greater autonomy introduces compounding errors, difficult credit assignment, and high variance across runs. Performance is sensitive to prompts, tool interfaces, and environment configuration, raising reproducibility concerns. Evaluation is also harder: benchmarks must avoid leakage and measure semantic correctness beyond superficial test passing. Finally, agentic systems incur substantial computational cost and safety risks due to tool access, requiring careful sandboxing, permissioning, and monitoring for deployment.

4 Evaluation of LLM-based Code Generation

Evaluation is a critical factor in understanding the progress and limitations of LLM-based code generation systems. As summarized by the benchmarks in Table 1, existing evaluations span a wide range of task settings, from unit-test–driven function synthesis to repository-level and agentic software engineering tasks. These settings differ not only in scale and realism, but also in the assumptions they make about specification completeness, execution feedback, and interaction horizon. In this section, we review common evaluation settings, examine the metrics they employ, and discuss challenges that arise when assessing modern LLM-based code generation methods.

4.1 Evaluation Settings and Task Granularity

Most widely used benchmarks, such as HumanEval, MBPP, MultiPL-E, and BigCodeBench, evaluate code generation at the function level. In this setting,

Benchmark	Size	Code Language(s)	Primary Evaluation Focus
HumanEval(Chen, 2021)	164	Python	Unit-test correctness for short functions (Pass@1 / Pass@k)
MBPP(Austin et al., 2021)	974	Python	Introductory Python tasks; basic logic and unit-test correctness
MultiPL-E(Boruch-Gruszecki et al., 2025)	∼18K	18 languages	Cross-language unit-test correctness and semantic equivalence
BigCodeBench(Zhuo et al., 2024)	∼1,100	Python	Complex function synthesis; multi-step reasoning and API usage
APPS(Hendrycks et al., 2021)	∼10K	Python	Algorithmic programming tasks with test-based evaluation
DS-1000(Lai et al., 2023)	1,000	Python	Data-science code generation with real-world library usage
LiveCodeBench(Jain et al., 2024)	∼300	Python	Execution-grounded code generation with dynamic tests
CodeXGLUE(Lu et al., 2021)	14	Multiple	Multi-task evaluation: generation, repair, translation, understanding
CodeScope(Yan et al., 2024)	∼1,000	40+ languages	Execution-based correctness across diverse languages
SWE-bench Verified(Jimenez et al., 2023)	500	Python	Repository bug fixing evaluated by CI tests
SWE-bench (Full)(Jimenez et al., 2023)	2,294	Python	Multi-file repository issues evaluated via CI
SWE-Bench Pro(Deng et al., 2025)	∼1,000	Python	Hard repository fixes with reduced benchmark leakage
GitTaskBench(Ni et al., 2025)	∼1,500	Python	Agentic repository tasks with iterative explore–edit–test loops
SWE-Compass(Xu et al., 2025)	∼2,000	Python	Unified evaluation for agentic software engineering systems
Long Code Arena(Bogomolov et al., 2024)	6 suites	Project-dependent	Long-context reasoning over multi-file codebases

Table 1: Representative benchmarks for LLM-based code generation.

models are given a well-defined specification and correctness is assessed via unit tests. Function-level evaluation is attractive due to its simplicity, low computational cost, and reproducibility, and it aligns naturally with prompt-conditioned and self-refinement methods. As a result, it remains the dominant evaluation paradigm for measuring basic functional correctness and reasoning ability.

However, function-level benchmarks abstract away many aspects of real-world programming. They assume that specifications are complete, relevant context fits within a single prompt, and correctness can be captured by a fixed set of tests. Benchmarks such as APPS and DS-1000 partially increase task complexity by emphasizing algorithmic reasoning or real-world library usage, yet still operate within a single-file or single-function abstraction.

Execution-grounded benchmarks, including LiveCodeBench and CodeScope, relax some of these assumptions by evaluating generated code through actual execution, often across dynamic or multi-language settings. These benchmarks better reflect the interaction between natural language reasoning and program semantics, but they remain limited in scope and typically do not require persistent state or long-horizon planning.

Repository-level benchmarks, such as SWE-bench and its variants, further expand task granularity by evaluating whether a system can resolve real software issues through multi-file edits and continuous integration tests. While these benchmarks improve ecological validity, they also introduce additional complexity related to environment setup, dependency management, and test nondeterminism. Agentic benchmarks, including GitTaskBench, SWE-Compass, and Long Code Arena, extend evaluation to long-horizon, tool-using scenarios, where success depends on planning, exploration, and iterative interaction rather than a single generation step.

4.2 Correctness Metrics and Their Limitations

Across most benchmarks in Table 1, correctness is primarily measured using test-based metrics, such as pass@k or task-level success. These metrics provide a clear and objective signal, particularly for function-level and execution-grounded benchmarks where tests are carefully curated. They also align well with execution-guided methods that explicitly optimize against runtime feedback.

Nevertheless, test-based metrics have well-known limitations. Passing a finite test suite does not guarantee semantic correctness, robustness to unseen inputs, or adherence to implicit requirements. In iterative or execution-guided settings, models may overfit to available tests, producing solutions that satisfy the evaluation harness without implementing the intended logic. This risk increases when public or partially overlapping tests are reused across attempts.

Moreover, binary success metrics collapse rich behavioral differences into a single outcome. For repository-level and agentic benchmarks, partial progress—such as correctly identifying relevant files, reducing failing tests, or converging toward a fix—is often informative but typically not reflected in final scores. This lack of granularity complicates failure analysis and obscures differences between systems that vary in reasoning quality rather than final success alone.

4.3 Data Contamination and Temporal Generalization

Data contamination poses a significant challenge for evaluating LLM-based code generation. Many benchmarks, especially those based on publicly

available problems or repositories, risk overlap with model pretraining data. This issue undermines the interpretation of reported results and makes it difficult to distinguish genuine generalization from memorization.

Recent benchmarks have attempted to address this concern by emphasizing temporal generalization, drawing tasks from time periods that postdate the training data of evaluated models. While this approach improves evaluation fidelity, it introduces practical challenges, including maintaining up-to-date benchmarks and verifying training cutoffs. Furthermore, temporal separation alone does not eliminate all forms of leakage, particularly when tasks resemble common programming patterns.

These concerns are especially pronounced for large, closed-source models with undisclosed training data. In such cases, benchmark results should be interpreted cautiously, and complementary analyses—such as sensitivity to prompt perturbations or robustness to minor task variations—can provide additional evidence of true reasoning capability.

4.4 Evaluation of Iterative and Agentic Systems

Evaluating iterative and agentic code generation systems introduces challenges that are not captured by traditional benchmarks. Agentic evaluations often involve stochastic decision-making, variable-length trajectories, and interaction with external tools or environments. Consequently, performance can be highly sensitive to retry budgets, stopping criteria, tool availability, and sandbox configuration.

A central difficulty is credit assignment. When success or failure occurs after many steps, it is unclear which decisions or components were responsible. This complicates scientific analysis and hinders fair comparison between methods. Additionally, differences in system implementation details—often under-specified in published work—can substantially affect results, reducing reproducibility across studies.

Designing agentic benchmarks therefore requires balancing realism and controllability. Highly realistic environments improve ecological validity but introduce noise and nondeterminism, while overly simplified settings may fail to stress the capabilities that agentic methods are intended to demonstrate. Identifying evaluation protocols that strike this balance remains an open research problem.

4.5 Cost, Efficiency, and Budget-Aware Reporting

As code generation methods become more iterative and agentic, inference cost and efficiency emerge as essential evaluation dimensions. Reporting only success rates without accounting for computational budget can be misleading, particularly when gains are achieved through extensive sampling, repeated execution, or long agent trajectories.

Budget-aware evaluation, which reports performance as a function of inference cost or constrains systems to a fixed budget, provides a more realistic assessment of practical utility. Such reporting is especially important for repository-level and agentic benchmarks, where modest improvements in success rate may require disproportionately large increases in compute. Transparent cost accounting enables meaningful comparison between methods and highlights trade-offs between accuracy, latency, and scalability.

5 Open Problems and Future Directions

Despite rapid progress in LLM-based code generation, existing methods and evaluation protocols expose a set of persistent challenges that are unlikely to be resolved through model scaling alone. In this section, we highlight several open problems and outline promising directions for future research, with an emphasis on inference-time reasoning, agentic behavior, and evaluation fidelity.

5.1 Specification Ambiguity and Alignment

A fundamental open problem in LLM-based code generation is the ambiguity of natural language specifications. Many programming tasks are underspecified, omit corner cases, or rely on implicit assumptions that are obvious to human developers but not explicitly stated. Current methods typically assume that the given prompt fully captures the intended behavior, leading models to make unverified assumptions that propagate through generation and refinement loops.

Future research may explore mechanisms for interactive specification clarification, where models actively identify ambiguities and query users or auxiliary systems before committing to an implementation. Another promising direction is the integration of lightweight formal or semi-formal constraints that can coexist with natural language, providing anchors for alignment without requiring full formal specifications.

5.2 Robustness Beyond Test Suites

While execution-guided and agentic methods have improved test pass rates, robustness beyond the available test suite remains largely unaddressed. Passing tests does not guarantee correct behavior under distribution shift, adversarial inputs, or evolving requirements. Moreover, iterative optimization against a fixed set of tests can encourage brittle solutions that exploit test artifacts. Addressing this challenge may require richer forms of behavioral evaluation, such as randomized or adversarial test generation, metamorphic testing, or specification-based fuzzing. Incorporating uncertainty estimation into code generation, where models explicitly signal low-confidence regions of the implementation, could also guide targeted verification and reduce overconfidence in fragile solutions.

5.3 Inference-Time Learning and Adaptation

Most current methods treat inference as a static process: the model reasons, receives feedback, and revises outputs, but does not retain lasting knowledge beyond the current task. This limits the ability of systems to adapt to recurring patterns, project-specific conventions, or repeated failure modes encountered across tasks. A key future direction is inference-time learning, where systems accumulate and reuse experience without full retraining. This may involve persistent memory of past fixes, reusable debugging strategies, or learned heuristics for tool selection. Designing such mechanisms raises questions about stability, generalization, and forgetting, and challenges existing distinctions between training and inference.

5.4 Credit Assignment in Long-Horizon Agentic Systems

As agentic code generation systems operate over longer horizons, understanding why a system succeeds or fails becomes increasingly difficult. Credit assignment across multi-step trajectories remains an open problem, hindering both method improvement and scientific analysis. Without clear attribution, it is challenging to diagnose failure modes, compare systems fairly, or provide meaningful explanations to users.

Future work may draw inspiration from reinforcement learning and causal analysis to develop interpretable abstractions of agent behavior. Intermediate metrics that capture progress toward a goal, rather than binary success, could offer more informative signals. Additionally, structured decomposition of tasks into verifiable subgoals may improve both controllability and interpretability.

5.5 Safety, Governance, and Responsible Deployment

Finally, as LLM-based code generation systems gain autonomy and access to execution environments, safety and governance considerations become increasingly important. Risks include unintended execution of harmful commands, leakage of sensitive information through prompts or repositories, and over-reliance on automated fixes without adequate human oversight. Future research should explore sandboxing, permission systems, and human-in-the-loop designs that balance autonomy with control. Transparent reporting of system capabilities and limitations, along with clear failure documentation, will be essential for responsible deployment. Addressing these concerns is not only a technical challenge but also a prerequisite for widespread adoption of LLM-based code generation in safety-critical domains.

6 Conclusion

This survey reviewed LLM-based code generation from an inference-time perspective, highlighting how recent progress increasingly arises from structured reasoning, feedback integration, and environment interaction rather than model scaling alone. By organizing existing methods into prompt-conditioned, self-refinement, execution-guided, and agentic approaches, we clarified their relationships and the trade-offs underlying their empirical performance.

We argued that evaluation practices play a central role in shaping perceived progress. While function-level benchmarks remain useful, they are insufficient for assessing iterative and agentic systems. Issues such as test overfitting, data contamination, cost sensitivity, and reproducibility underscore the need for evaluation protocols that better reflect task complexity and inference dynamics.

Looking forward, advancing LLM-based code generation will require more faithful evaluation, improved robustness beyond fixed test suites, and principled handling of long-horizon interaction. We hope this survey provides a coherent framework for understanding current methods and informs future work toward reliable and scalable code generation systems.

Limitations

This survey has several limitations. First, our taxonomy and analysis focus on inference-time strategies for LLM-based code generation, intentionally deemphasizing model architecture design, pretraining data curation, and optimization techniques. While this perspective highlights an important and rapidly evolving dimension of the field, it does not fully capture advances driven by large-scale training or model-specific innovations, which may interact with inference-time methods in nontrivial ways.

Second, our coverage of benchmarks and evaluation protocols reflects the current state of publicly available datasets and reported results. As many benchmarks evolve rapidly and new evaluation settings continue to emerge, some recently released datasets or unpublished systems may not be fully represented. In addition, reported performance comparisons across benchmarks are subject to differences in experimental setups, inference budgets, and implementation details, which limits the extent to which conclusions can be directly generalized.

Third, the analysis relies on benchmark-driven evidence, which may not fully reflect real-world deployment scenarios. Benchmarks often simplify task specifications, constrain environments, or assume well-defined success criteria, whereas practical software development involves ambiguity, evolving requirements, and human-in-the-loop decision making. Consequently, conclusions drawn from benchmark performance should be interpreted as indicative rather than definitive measures of practical capability.

Finally, as a survey, this work synthesizes and interprets existing literature rather than proposing new methods or evaluation protocols. While we aim to provide a coherent framework and identify open challenges, the effectiveness of future approaches will ultimately depend on empirical validation in diverse and realistic settings.

AI Disclosure

This work was completed with the assistance of large language models (LLMs) used as a writing and editing aid. Specifically, LLMs were employed to help improve clarity, organization, and conciseness of the text, as well as to refine grammar and presentation. All technical content, interpretations, analyses, and conclusions were conceived, verified, and finalized by the authors.

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, and 1 others. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and 1 others. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, and 1 others. 2022. Constitutional ai: Harmlessness from ai feedback. *arXiv preprint arXiv:2212.08073*.

Egor Bogomolov, Aleksandra Eliseeva, Timur Galimzyanov, Evgeniy Glukhov, Anton Shapkin, Maria Tigina, Yaroslav Golubev, Alexander Kovrigin, Arie Van Deursen, Maliheh Izadi, and 1 others. 2024. Long code arena: a set of benchmarks for long-context code models. *arXiv preprint arXiv:2406.11612*.

Aleksander Boruch-Gruszecki, Yangtian Zi, Zixuan Wu, Tejas Oberoi, Carolyn Jane Anderson, Joydeep Biswas, and Arjun Guha. 2025. Agnostics: Learning to code in any programming language via reinforcement with a universal learning environment. *arXiv preprint arXiv:2508.04865*.

Mark Chen. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*.

Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Yunxuan Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, and 1 others. 2024. Scaling instruction-finetuned language models. *Journal of Machine Learning Research*, 25(70):1–53.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, and 1 others. 2025. Swe-bench pro: Can ai agents solve long-horizon software engineering tasks? *arXiv preprint arXiv:2509.16941*.

Yuxuan Ding, Chunna Tian, Haoxuan Ding, and Lingqiao Liu. 2023. The clip model is secretly an image-to-prompt converter. *Advances in Neural Information Processing Systems*, 36:56298–56309.

Srishti Gureja, Elena Tommasone, Jingyi He, Sara Hooker, Matthias Gallé, and Marzieh Fadaee. 2025. Verification limits code llm training. *arXiv preprint arXiv:2509.20837*.

Mirazul Haque, Petr Babkin, Farima Farmahinifarahani, and Manuela Veloso. 2025. Towards effectively leveraging execution traces for program repair with code llms. *arXiv preprint arXiv:2505.04441*.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with apps. *NeurIPS*.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, and 1 others. 2023. Metagpt: Meta programming for a multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations*.

Dong Huang, Jie M Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. 2023. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010*.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2023. Ds-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning*, pages 18319–18345. PMLR.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, and 1 others. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, and 1 others. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Zi Lin, Sheng Shen, Jingbo Shang, Jason Weston, and Yixin Nie. 2025. Learning to solve and verify: A self-play framework for code and test generation. *arXiv preprint arXiv:2502.14948*.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, and 3 others. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *CoRR*, abs/2102.04664.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*.

Zihan Ma, Taolin Zhang, Maosong Cao, Junnan Liu, Wenwei Zhang, Minnan Luo, Songyang Zhang, and Kai Chen. 2025. Rethinking verification for llm code generation: From generation to testing. *arXiv preprint arXiv:2507.06920*.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, and 1 others. 2023. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36:46534–46594.

Ziyi Ni, Huacan Wang, Shuo Zhang, Shuo Lu, Ziyang He, Wang You, Zhenheng Tang, Yuntao Du, Bill Sun, Hongzhang Liu, and 1 others. 2025. Gittaskbench: A benchmark for code agents solving real-world tasks through code repository leveraging. *arXiv preprint arXiv:2508.18993*.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, and 1 others. 2022. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, and 1 others. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36:68539–68551.

Yuling Shi, Songsong Wang, Chengcheng Wan, Min Wang, and Xiaodong Gu. 2024. From code to correctness: Closing the last mile of code generation with hierarchical debugging. *arXiv preprint arXiv:2410.01215*.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement

learning. *Advances in Neural Information Processing Systems*, 36:8634–8652.

Jian Wang, Xiaofei Xie, Qiang Hu, Shangqing Liu, and Yi Li. 2025. Do code semantics help? a comprehensive study on execution trace-based information for code large language models. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pages 10367–10385.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, and 1 others. 2024. Openhands: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741*.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2022. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*.

Brandon T Willard and Rémi Louf. 2023. Efficient guided generation for large language models. *arXiv preprint arXiv:2307.09702*.

Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, and Daxin Jiang. 2023. Wizardlm: Empowering large language models to follow complex instructions. *arXiv preprint arXiv:2304.12244*.

Jingxuan Xu, Ken Deng, Weihao Li, Songwei Yu, Huaixi Tang, Haoyang Huang, Zhiyi Lai, Zizheng Zhan, Yanan Wu, Chenchen Zhang, and 1 others. 2025. Swe-compass: Towards unified evaluation of agentic coding abilities for large language models. *arXiv preprint arXiv:2511.05459*.

Weixiang Yan, Haitian Liu, Yunkun Wang, Yunzhe Li, Qian Chen, Wen Wang, Tingyu Lin, Weishan Zhao, Li Zhu, Hari Sundaram, and 1 others. 2024. 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)*, pages 5511–5558.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, and 1 others. 2024. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*.