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 code generation, evolving from prompt-based function synthesis to iterative, execution-guided, tool-integrated, and agentic software engineering systems. While recent progress has produced impressive benchmark results, the diversity of methods and evaluation settings has also introduced fragmentation 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 by how LLMs constrain generation, search among candidates, incorporate feedback, and interact with execution environments, covering grammar- and schema-constrained generation, prompt-conditioned generation, language-based self-refinement, execution-guided revision, test-time ranking, tool-integrated generation, and agentic code generation. We further review benchmarks across function-level, execution-grounded, repository-level, and long-horizon agentic tasks, and analyze how evaluation assumptions shape reported performance. Our analysis highlights limitations of test-based correctness metrics, contamination risks, latency–quality trade-offs, and challenges in evaluating iterative and agentic systems. By connecting method design with evaluation protocols, this survey aims to clarify current progress and guide 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, covering tasks from code completion and program synthesis to bug fixing and repository-level software engineering. Early LLM-based work framed code generation as conditional generation, where a model maps natural-language descriptions, partial code context, or examples to executable programs. Representative systems include proprietary API models evaluated on HumanEval (Chen et al., 2021), as well as open-source code LMs such as CodeGen (Nijkamp et al., 2022), StarCoder (Li et al., 2023), and Code Llama (Roziere et al., 2023).

Code generation, however, predates modern LLMs. Before the current LLM era, grammar- and syntax-aware methods constrained outputs to valid programs or formal meaning representations. TRANX, for example, introduced transition-based abstract syntax generation for semantic parsing, showing that explicit grammar constraints can reduce invalid generations and improve structural correctness (Yin and Neubig, 2018). Although modern LLMs emphasize pretraining, prompting, and instruction following, grammar-constrained decoding remains important because it separates syntactic validity from semantic correctness.

Recent advances have shifted attention from model capacity alone to inference-time strategies that structure reasoning, incorporate feedback, and interact with external environments. Modern systems increasingly rely on iterative refinement, feedback signals, and tool-mediated action loops rather than single-pass generation. Language-based self-refinement methods improve outputs through critique and revision without necessarily 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), improve robustness through sampling and policy-guided critique.

With large code LMs, research also shifted from single-sample generation to test-time selection, ranking, and revision. AlphaCode (Li et al., 2022) showed that large-scale sampling with filtering improves competitive programming performance. Fault-aware neural code rankers rank candidates without execution by predicting correctness and failure types (Inala et al., 2022). Self-Edit

uses execution results in a generate-and-edit framework (Zhang et al., 2023a), while CodeT uses generated tests and dual execution agreement for candidate selection (Chen et al., 2022). These methods mark a transition from one-shot generation toward inference-time scaling, where candidates, tests, and feedback improve final outputs.

Another major transition is tool integration. ToolCoder introduced API search tools to help models select APIs and reduce hallucinated library usage (Zhang et al., 2023b). CodeAgent extended this idea to repository-level generation by integrating tools for retrieval, implementation, and testing (Zhang et al., 2024a). This line connects to broader agentic systems, including SWE-agent (Yang et al., 2024), OpenHands (Wang et al., 2024), Agentless (Xia et al., 2024), and AutoCodeRover (Zhang et al., 2024b). Together, these systems show that repository-level software engineering requires code synthesis, localization, context retrieval, test execution, patch validation, and workflow design.

The practical landscape has also evolved through commercial coding agents such as Claude Code, OpenAI Codex, and Cursor Agent. Although not always fully documented in peer-reviewed literature, they are important for understanding deployment constraints and user-facing workflows. They emphasize codebase understanding, multi-file editing, terminal/tool use, sandboxed execution, and pull-request-oriented workflows. Their emergence broadens the research question: beyond whether LLMs generate correct functions, the field must evaluate when agentic assistance is reliable, fast, safe, and cost-effective for real developer workflows.

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

This survey brings coherence to the field by jointly examining method design and evaluation practice from an inference-time perspective. Rather than focusing on architectures or pretraining, we analyze how LLM-based code generation systems structure reasoning, leverage feedback, interact with tools, and allocate inference-time budget. We further review benchmarks spanning function-level, execution-grounded, repository-level, and long-horizon agentic settings, emphasizing evaluation reliability, contamination risk, and latency–quality trade-offs.

2 Definition and Scope

This survey focuses on *LLM-based code generation*, where large language models serve as the primary reasoning and decision-making component for programming tasks. To ensure clarity and avoid scope creep, we define the paradigm and 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, including natural-language specifications, partial code context, or both. The defining characteristic is that inference-time behavior is driven mainly by language-based reasoning, optionally augmented with external feedback such as execution results, test outcomes, or tool responses. Under this definition, code generation extends beyond standalone function synthesis to include code completion, bug fixing, test generation, and repository-level modification, as long as the LLM remains the central component guiding decisions and actions.

Scope of Methods. Our review emphasizes inference-time strategies that improve code generation without modifying the underlying model architecture or requiring large-scale retraining. We cover three broad classes: (i) prompt-conditioned generation from textual context, (ii) iterative refinement using language-based or execution-based feedback, and (iii) agentic frameworks where an LLM interacts with tools and environments over multiple steps.

Evaluation and Exclusions. We consider evaluation settings from unit-test–driven benchmarks to repository-level tasks requiring iterative debugging and long-horizon reasoning. We exclude classical program synthesis not centered on LLMs, detailed architecture or pretraining studies, and purely non-LLM software engineering tools. This positioning provides a unified, LLM-centered view while keep-

ing the survey focused.

3 Taxonomy of LLM-based Code Generation Methods

This section presents a literature-driven taxonomy of LLM-based code generation, organized by *how LLMs are used at inference time* to structure reasoning, incorporate feedback, allocate computation, and interact with environments. We group prior work into four families: (i) prompt- and constraint-conditioned generation, (ii) language-based feedback and verification, (iii) execution-guided generation and test-time selection, and (iv) agentic and environment-interactive systems.

Unlike surveys organized by model family or task type, our taxonomy emphasizes *inference-time control*: how a system constrains generation, searches among candidates, obtains feedback, invokes tools, and decides when to stop. This view is useful because many gains come not only from larger models, but from embedding LLMs in reasoning loops, execution environments, or agent controllers. It also links methods to evaluation: final unit-test success may hide whether improvement comes from initial generation, ranking, debugging, sampling, or tool interaction.

For each family, we summarize core ideas, design choices, empirical promise, and limitations, enabling comparison across method classes and clarifying their trade-offs.

3.1 Prompt- and Constraint-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 one pass. This family includes proprietary API models evaluated on HumanEval (Chen et al., 2021) and open code LMs such as CodeGen (Nijkamp et al., 2022), StarCoder (Li et al., 2023), and Code Llama (Roziere et al., 2023). The assumption is that pretraining and instruction tuning encode enough program patterns to produce plausible implementations without explicit feedback.

Key Design Choices. Design centers on prompt construction, context selection, instruction alignment, and structural constraints. Alignment variants use instruction tuning and preference optimization, such as InstructGPT-style RLHF (Ouyang et al., 2022), while data-centric methods use synthetic instruction generation and task refinement (Chung et al., 2024; Xu et al., 2023; Luo et al., 2023). Structural constraints trace back to grammar-based systems such as TRANX (Yin and Neubig, 2018); modern constrained decoding applies grammar-, regex-, type-, or schema-based constraints to LLM outputs and tool/function interfaces (Willard and Louf, 2023; Achiam et al., 2023). These constraints help with APIs, JSON schemas, and DSLs, but mainly ensure structural validity rather than semantic correctness.

Performance and Promise. Prompt-conditioned generation is attractive due to simplicity and low inference cost. It performs well on function-level synthesis and completion when benchmarks align with training distributions and prompts provide scaffolding. These methods offer favorable latency–quality trade-offs for interactive assistance, while instruction alignment improves usability.

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 tasks become harder, failures from underspecification, hidden edge cases, hallucinated APIs, or inconsistent decisions become frequent. Structural constraints reduce formatting and syntax errors but do not guarantee semantic correctness.

3.2 Self-Refinement and Language-Based Feedback

Core Idea and Paradigm. Language-based feedback methods extend single-pass generation with an iterative critique–revision loop. The model acts as both generator and critic, producing diagnostics such as error hypotheses, missing cases, alternative plans, or review comments to guide revisions. Representative methods include Self-Refine (Madaan et al., 2023) and Reflexion (Shinn et al., 2023).

Key Design Choices. Key choices include what feedback artifact to generate, how to incorporate it, and how to control iteration. Feedback may take the form of critiques, plans, checklists, or counterexamples, and may be inserted inline, separated into structured sections, or stored in memory. Reliability can be improved through agreement and verification heuristics, including self-consistency (Wang et al., 2022) and rule/policy-guided critique (Bai et al., 2022). Recent work further analyzes when test-like checking helps and when verification fails (Ma et al., 2025; Gureja

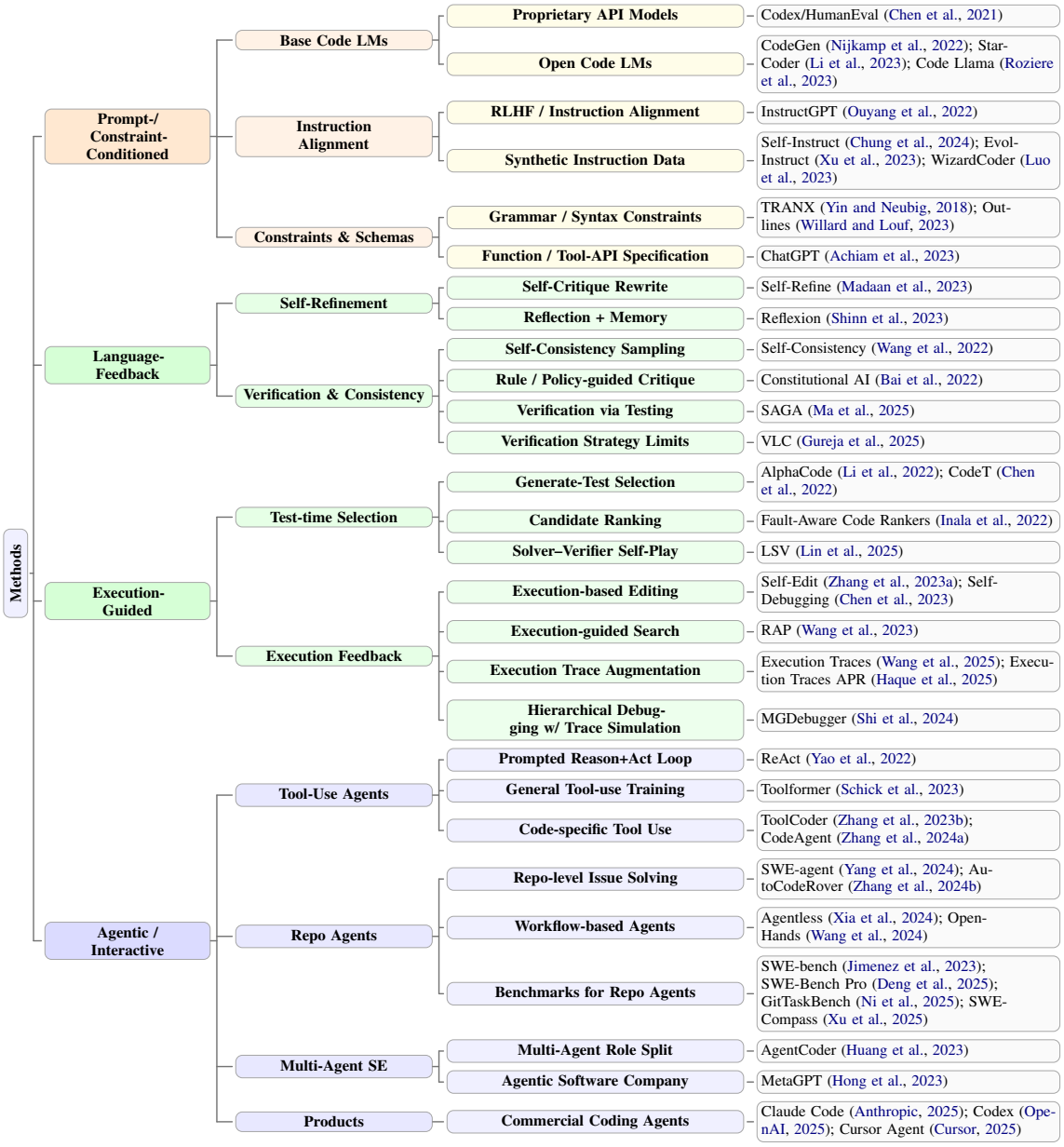

Figure 1: Taxonomy of LLM-based code generation from an inference-time perspective. The taxonomy covers grammar and schema constraints, prompt-conditioned generation, self-refinement, test-time ranking and selection, execution-guided revision, tool-integrated agents, repository-level software engineering agents, multi-agent systems, and commercial coding agents.

et al., 2025).

Performance and Promise. Compared to single-pass generation, self-refinement often improves correctness and reasoning quality without extra training. It provides lightweight inference-time scaffolding, adapts via prompting, and can be used when execution is unavailable.

Challenges and Limitations. Language-only self-assessment can be unreliable: critiques may be fluent but wrong, and refinement can converge to polished yet incorrect solutions. Iterative loops show diminishing returns and sensitivity to prompt format and budget. Without execution grounding, models may reinforce false assumptions, and verification can remain brittle under distribution shift (Gureja et al., 2025).

3.3 Execution-Guided Code Generation and Test-Time Selection

Core Idea and Paradigm. Execution-guided methods use compilation results, runtime errors, test outcomes, and traces as behavioral signals for selection or revision. Instead of relying only on linguistic critique, the model or system interprets failures to revise code or select candidates, aligning this

paradigm with debugging, test-driven development, and test-time scaling.

Key Design Choices. Systems differ in how they spend test-time computation. One line emphasizes *candidate ranking and selection*: AlphaCode samples and filters many programs (Li et al., 2022); fault-aware rankers predict correctness and failure types without execution (Inala et al., 2022); CodeT selects programs through dual execution agreement between generated tests and candidate outputs (Chen et al., 2022). Another line emphasizes *iterative repair*: Self-Edit edits faulty programs using execution results (Zhang et al., 2023a), while self-debugging and execution-guided search condition revision on compiler errors, runtime exceptions, failing tests, or traces (Chen et al., 2023; Wang et al., 2025; Haque et al., 2025; Shi et al., 2024). These approaches trade off sampling, ranking, test generation, and repair rounds.

Performance and Promise. Execution-guided methods improve pass rates by adding behavioral evidence unavailable to language-only generation. Ranking helps when many candidates exist; editing helps when initial solutions are close but flawed; generated tests reduce dependence on manual tests. Thus, execution-guided generation is a clear form of test-time scaling.

Challenges and Limitations. The same mechanism introduces risks. Generated tests may be weak or wrong; public tests may encourage overfitting; and repeated execution can optimize for the visible harness rather than the intended specification. Ranking methods may inherit label biases, while editing methods may patch symptoms without fixing root causes. Gains should therefore be interpreted with sample count, generated tests, repair iterations, execution budget, and hidden or temporal test separation.

3.4 Agentic and Environment-Interactive Code Generation

Core Idea and Paradigm. Agentic methods model code generation as long-horizon interaction. Rather than generating code in isolation, the system decomposes goals into actions such as repository search, file inspection, code editing, test execution, and output interpretation. Language serves as both reasoning substrate and tool interface.

Key Design Choices. Agentic systems differ in tool/action space, control policy, state representation, and autonomy. ReAct (Yao et al., 2022) operationalizes reason-and-act prompting, while Toolformer (Schick et al., 2023) aims to internalize tool use. ToolCoder (Zhang et al., 2023b) integrates API search to reduce inappropriate API calls, and CodeAgent (Zhang et al., 2024a) extends tool integration to repository-level retrieval, implementation, and testing.

SWE-agent (Yang et al., 2024) emphasizes agent-computer interfaces for repository navigation and command execution, while OpenHands (Wang et al., 2024) provides an open platform for sandboxed software agents. Agentless (Xia et al., 2024) shows that localization–repair–validation workflows can be effective without full autonomy, and AutoCodeRover (Zhang et al., 2024b) combines structured code search, fault localization, and patch validation. Commercial systems such as Claude Code, OpenAI Codex, and Cursor Agent reflect these ideas in products that combine codebase indexing, multi-file editing, terminal/tool use, sandboxing, and human review; we treat them as deployment trends rather than reproducible baselines.

Performance and Promise. Agentic systems expand code generation from isolated functions to realistic software engineering workflows. They can search repositories, inspect dependencies, run tests, revise patches, and manage multi-step tasks. Results from Agentless and AutoCodeRover also suggest that full autonomy is not always necessary; constrained workflows can improve reliability, interpretability, and cost.

Challenges and Limitations. Agentic code generation introduces high variance, long trajectories, and sensitivity to prompts, tools, retry budgets, sandboxes, and stopping rules. Deployment also involves a latency–quality trade-off: real-time IDE assistants must respond within seconds, while offline agents may spend minutes exploring, testing, and validating. Thus, high-success long-trajectory agents may be unsuitable for interactive completion, while low-latency assistants remain valuable even without full autonomy.

4 Evaluation of LLM-based Code Generation

Evaluation is central to understanding progress in LLM-based code generation. As shown in Table 2, existing benchmarks range from unit-test–driven function synthesis to repository-level and agentic software engineering tasks. They differ in scale, realism, specification completeness, execution feedback, interaction horizon, and inference budget.

Method Family	Representative Works	Feedback / Signal	Suitable Evaluation Setting	Common Failure Modes
Grammar- / schema-constrained generation	TRANX; constrained decoding; tool/function schemas	Grammar, type, regex, API schema	DSLs, structured outputs, function calls, syntax-sensitive tasks	Ensures form but not semantics; may reject valid but unexpected solutions
Prompt-conditioned generation	Codex, CodeGen, StarCoder, Code Llama	Prompt context only	Function-level synthesis, completion, low-latency IDE assistance	Hallucinated APIs, missing edge cases, weak handling of implicit requirements
Candidate ranking / test-time selection	AlphaCode, CodeRanker, CodeT	Samples, generated tests, ranker scores, execution agreement	Competitive programming, HumanEval/MBPP/APPS-style tasks	High sampling cost; weak generated tests; overfitting to visible tests
Iterative editing and debugging	Self-Edit, Self-Debugging, execution-guided repair	Compiler errors, runtime errors, failing tests, traces	Tasks where initial solution is close but flawed	Symptom patching; error-message overfitting; diminishing returns
Tool-integrated agents	ToolCoder, CodeAgent, SWE-agent, OpenHands	Repository search, file editing, terminal commands, test execution	Repo-level issue fixing, long-horizon software engineering	Trajectory variance; tool/interface sensitivity; reproducibility and safety risks
Constrained / workflow-based agents	Agentless, AutoCodeRover	Localization, structured code search, patch validation	SWE-bench-style repository repair	May be less flexible for open-ended tasks; depends on retrieval and fault localization quality
Commercial coding agents	Claude Code, Codex, Cursor Agent	Codebase context, tools, sandbox execution, human review	Practical developer workflows	Limited reproducibility; fast-changing capabilities; unclear training/evaluation boundaries

Table 1: Comparison of major families in LLM-based code generation. The table emphasizes feedback signals, suitable evaluation settings, and common failure modes, helping connect method design with benchmark choice.

This section reviews common settings, metrics, and reliability challenges in evaluating modern code generation systems.

4.1 Evaluation Settings and Task Granularity

Function-level benchmarks such as HumanEval, MBPP, MultiPL-E, and BigCodeBench evaluate code generation using well-defined specifications and unit tests. They are simple, low-cost, reproducible, and naturally aligned with prompt-conditioned or self-refinement methods, making them the dominant setting for measuring basic functional correctness.

However, function-level evaluation abstracts away many aspects of real programming: specifications are assumed complete, relevant context fits in one prompt, and correctness is approximated by fixed tests. APPS and DS-1000 increase difficulty through algorithmic reasoning or library usage, but still largely remain within single-file or single-function settings. Execution-grounded benchmarks such as LiveCodeBench and CodeScope better reflect program behavior through actual execution, often across dynamic or multi-language settings, but still rarely require persistent state or long-horizon planning.

Repository-level benchmarks, including SWE-bench and its variants, evaluate real software issues through multi-file edits and CI tests, improving ecological validity while introducing environment setup, dependency management, and nondeterminism. Agentic benchmarks such as GitTaskBench, SWE-Compass, and Long Code Arena further test planning, repository exploration, tool use, and iterative interaction.

4.2 Correctness Metrics and Limitations

Most benchmarks use test-based metrics such as pass@k or task-level success. These metrics are objective, reproducible, and well aligned with execution-guided methods. Yet passing finite tests does not guarantee semantic correctness, robustness to unseen inputs, or satisfaction of implicit requirements. In iterative settings, models may overfit to public, generated, or repeatedly reused tests, adapting to the evaluation signal rather than the intended specification.

Binary success metrics also hide useful behavioral differences. For repository-level and agentic benchmarks, intermediate progress—such as locating relevant files, reducing failing tests, or producing a partially correct patch—can be informative but is rarely captured by final scores. This limits failure analysis and makes it difficult to compare systems whose reasoning quality differs despite similar final pass rates.

4.3 Data Contamination and Temporal Generalization

Data contamination is a spectrum: models may have seen exact solutions, similar problems, repository histories, issue discussions, or public tests. Since code benchmarks often rely on public problems and open-source repositories, high scores on static public benchmarks should not be interpreted as conclusive evidence of generalization without contamination controls.

Useful mitigation strategies include temporally separated tasks, hidden tests, regenerated tests, perturbation-based variants, and disclosure of whether tests are public, private, generated, or man-

Benchmark	Size	Code Language(s)	Primary Evaluation Focus
HumanEval(Chen et al., 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(**?**)	∼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 2: Representative benchmarks for LLM-based code generation.

ually validated. For closed-source models with undisclosed training data, results should be framed as empirical performance under a specific protocol rather than definitive evidence of reasoning ability.

4.4 Iterative and Agentic Systems

Iterative and agentic systems introduce challenges beyond traditional benchmarks. Their performance depends on stochastic trajectories, retry budgets, stopping rules, tool availability, sandbox configuration, and implementation details. Credit assignment is also difficult: when success or failure occurs after many steps, it is unclear which decisions or components were responsible.

Agentic benchmark design must therefore balance realism and controllability. Realistic environments improve ecological validity but increase noise and nondeterminism, while simplified environments may fail to test the capabilities agentic systems are designed for. More informative protocols should report intermediate progress, repeated trials, tool access, and trajectory-level behavior in addition to final success.

4.5 Cost, Efficiency, and Latency–Quality Trade-off

As methods become more iterative and agentic, cost and latency become essential evaluation dimensions. Reporting success rates alone can be misleading when gains rely on extensive sampling, repeated execution, or long trajectories. Budget-aware evaluation, which reports performance as a function of inference cost or constrains systems to a fixed budget, better reflects practical utility.

This is especially important because different systems target different interaction regimes. Real-time IDE completion and chat-based assistance require low latency, often within seconds, where prompt-conditioned generation or lightweight self-refinement may be preferable. Offline or semi-autonomous agents can spend more time searching repositories, running tests, generating patches, and validating changes, but their higher quality must justify added latency, cost, and supervision. Evaluations should therefore report wall-clock latency, model calls, samples, test executions, tool/action count, retry budget, and monetary cost when applicable.

5 Open Problems and Future Directions

Despite rapid progress in LLM-based code generation, existing methods and evaluation protocols still expose persistent challenges unlikely to be solved by scaling alone. We highlight several future directions centered on inference-time reasoning, agentic behavior, and evaluation fidelity.

5.1 Specification Ambiguity and Alignment

Natural-language programming tasks often omit corner cases, project conventions, or implicit assumptions. Recent coding agents and commercial systems can inspect repositories and infer some local context, but these capabilities remain uneven: models may still silently choose one plausible interpretation, especially when tests are incomplete or user intent is underspecified.

Future research should move beyond assuming either fully specified prompts or fully autonomous intent inference. Promising directions include interactive clarification, uncertainty reporting, intent checking before implementation, and lightweight formal or semi-formal constraints that complement natural language. Evaluation should also include ambiguous tasks where desired behavior cannot be recovered from unit tests alone.

Evaluation Setting	Typical Benchmarks	Reliability Strengths	Main Risks	Recommended Reporting Practice
Function-level unit tests	HumanEval, MBPP, MultiPL-E, BigCodeBench	Simple, reproducible, low-cost; supports pass@k comparison	Small task scope; incomplete tests; possible contamination	Report pass@1/pass@k, sampling temperature, sample count, and hidden or augmented tests when possible
Algorithmic / contest tasks	APPS, CodeContests, LiveCodeBench	Harder reasoning; stronger execution grounding	Prompt sensitivity; public problem leakage; hidden-test dependence	Use temporally separated tasks; report test-generation or repair budget; separate public and hidden-test performance
Library / data-science tasks	DS-1000, API-heavy benchmarks	Tests realistic API usage and dependency reasoning	Environment/version sensitivity; API hallucination; brittle setup	Report package versions, environment details, and dependency/API failure categories
Repository-level repair	SWE-bench, SWE-bench Verified, SWE-bench Pro	Closer to real software maintenance; multi-file context	CI nondeterminism; issue ambiguity; patch overfitting; high cost	Report resolved rate, cost, trajectory length, retry budget, and exact environment; include secondary validation when possible
Agentic long-horizon tasks	GitTaskBench, SWE-Compass, Long Code Arena	Evaluates planning, navigation, tool use, and interaction	High variance; tool-interface sensitivity; difficult credit assignment	Run repeated trials; report tool access, stopping rules, action count, latency, and intermediate progress metrics
Commercial/product evaluations	Claude Code, Codex, Cursor Agent	Reflects practical developer workflows and adoption constraints	Closed models, changing systems, limited reproducibility	Treat as deployment case studies; distinguish product observations from controlled academic evidence

Table 3: Evaluation reliability for LLM-based code generation. Robust comparison requires reporting correctness together with benchmark scope, contamination controls, execution budget, tool access, and reproducibility conditions.

5.2 Robustness Beyond Test Suites

Although execution-guided and agentic methods improve test pass rates, robustness beyond available tests remains unresolved. Passing tests does not ensure correctness under distribution shift, adversarial inputs, or evolving requirements, and optimization against fixed tests can produce brittle solutions that exploit test artifacts. Addressing this requires richer behavioral evaluation, including randomized or adversarial test generation, metamorphic testing, and specification-based fuzzing. Uncertainty estimation may also help identify low-confidence code regions for targeted verification.

5.3 Inference-Time Learning and Adaptation

Most current methods treat inference as task-local: the model reasons, receives feedback, and revises outputs, but does not retain lasting knowledge across tasks. This limits adaptation to recurring patterns, project conventions, and repeated failure modes. A key direction is inference-time learning, where systems accumulate reusable experience without full retraining, such as memories of past fixes, debugging strategies, or tool-selection heuristics. The challenge is ensuring stability, generalization, and resistance to erroneous patterns.

5.4 Credit Assignment in Long-Horizon Agentic Systems

As agents operate over longer horizons, it becomes difficult to determine why they succeed or fail. Credit assignment across multi-step trajectories remains an open problem, limiting failure diagnosis, method improvement, and fair comparison. Future work may use reinforcement learning, causal analysis, or trajectory-level diagnostics to build interpretable abstractions of agent behavior. Intermediate progress metrics and verifiable subgoals could provide richer signals than binary success, improving controllability and interpretability.

5.5 Safety, Governance, and Deployment

As code agents gain access to execution environments, safety and governance become increasingly important. Risks include harmful commands, sensitive information leakage, insecure patches, and over-reliance on automated fixes. Future systems should incorporate sandboxing, permission control, and human-in-the-loop review. Transparent reporting of capabilities, limitations, and failure modes is essential for responsible deployment, especially in safety-critical software domains.

6 Conclusion

This survey reviewed LLM-based code generation from an inference-time perspective, showing that recent progress increasingly depends on structured reasoning, feedback, test-time selection, tool use, and environment interaction rather than model scaling alone. By organizing methods into prompt-/constraint-conditioned, self-refinement, execution-guided, and agentic approaches, we clarified their relationships and trade-offs. We also emphasized that evaluation shapes perceived progress. Function-level benchmarks remain useful but are insufficient for iterative and agentic systems, where test overfitting, contamination, cost, latency, and reproducibility become critical concerns. Future progress will require faithful evaluation, robustness beyond fixed tests, and principled treatment of long-horizon interaction.

Limitations

This survey has several limitations. First, our taxonomy and analysis focus on inference-time strategies for LLM-based code generation, intentionally de-emphasizing 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*.

Anthropic. 2025. Claude code. https://docs.anthropic.com/en/docs/claude-code/overview. Official documentation for Claude Code, an agentic coding tool for terminal, IDE, and GitHub workflows. Accessed: 2026-05-26.

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*.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 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.

Cursor. 2025. Cursor agent. https://cursor.com/docs/agent/overview. Official documentation for Cursor Agent, an assistant for autonomous coding tasks, terminal commands, and code editing. Accessed: 2026-05-26.

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*.

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*.

Jeevana Priya Inala, Chenglong Wang, Mei Yang, Andres Codas, Mark Encarnación, Shuvendu Lahiri, Madanlal Musuvathi, and Jianfeng Gao. 2022. Fault-aware neural code rankers. *Advances in Neural Information Processing Systems*, 35:13419–13432.

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*.

OpenAI. 2025. Codex: Openai's coding agent. https://developers.openai.com/codex/cloud. Official documentation for Codex, OpenAI's coding agent that can read, edit, and run code. Accessed: 2026-05-26.

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.

Weishi Wang, Yue Wang, Shafiq Joty, and Steven CH Hoi. 2023. Rap-gen: Retrieval-augmented patch generation with codet5 for automatic program repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 146–158.

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*.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489*.

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*.

Pengcheng Yin and Graham Neubig. 2018. Tranx: A transition-based neural abstract syntax parser for semantic parsing and code generation. In *Proceedings of the 2018 conference on empirical methods in natural language processing: System demonstrations*, pages 7–12.

Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. 2024a. Codeagent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13643–13658.

Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. 2023a. Self-edit: Fault-aware code editor for code generation. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 769–787.

Kechi Zhang, Huangzhao Zhang, Ge Li, Jia Li, Zhuo Li, and Zhi Jin. 2023b. Toolcoder: Teach code generation models to use api search tools. *arXiv preprint arXiv:2305.04032*.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024b. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1592–1604.

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*.