Code Reasoning for Code Tasks: A Survey and A Call to Action

Anonymous authors
Paper under double-blind review

Abstract

The rise of large language models (LLMs) has led to dramatic improvements across a wide range of natural language tasks. Their performance on certain tasks can be further enhanced by incorporating test-time reasoning techniques. These inference-time advances have been adopted into the code domain, enabling complex software engineering(SWE) tasks such as code generation, test generation and issue resolution. However, the impact of different reasoning techniques on code-centric SWE tasks has not been systematically explored. In this work, we survey code reasoning techniques that underpin these capabilities, with a focus on test-time compute and inference-time reasoning paradigms. We examine a variety of code-specific reasoning methods and progressively build up to SWE agents, which combine planning, tool use, and multi-step interaction. We also compare the impact of different techniques on coding tasks, highlighting their relative importance and outlining open challenges and future research directions. Our contributions are: (1) to the best of our knowledge, the first dedicated survey of code reasoning for SWE tasks, highlighting overarching reasoning strategies, hybrid methods, and agentic approaches; (2) a taxonomy of inference-time techniques used to drive code reasoning, accompanied by a curated set of under-explored benchmarks with high potential for SWE evaluation; (3) a comparative analysis of reasoning design patterns across commonly used models and benchmarks; and (4) a synthesis of gaps in current methods and evaluation practices, identifying under-explored areas and concrete opportunities for future research.

1 Introduction

Hindle et al., 2012 show that software is repetitive and predictable like natural language, and hence can be modeled using statistical techniques like LLMs. Subsequently, LLMs have been used effectively for a wide variety of Software Engineering (SWE) tasks[1], including code generation (Chen et al., 2021b), test generation (Mündler et al., 2025), issue resolution (Jimenez et al., 2024b) and others. Many code specific datasets (Puri et al., 2021; Khan et al., 2024), models (Li et al., 2023; Nijkamp et al., 2023) and benchmarks (Hendrycks et al., 2021a; Zhuo et al., 2025) have also been developed. Despite this progress, LLMs have been shown to be limited in their capacity to solve real-world SWE tasks, like GitHub issue resolution Jimenez et al. (2024b). Recent development of large reasoning models (LRMs) Guo et al. (2025); Anthropic (2025); Jaech et al. (2024) and SWE agents have resulted in tremendous improvement on code tasks, including GitHub issue resolution.

In a recent survey, Yang et al., 2025 explore how code and reasoning reinforce each other. They compile works showing how incorporating code data improves reasoning, and how better reasoning leads to improvement on SWE tasks. Reasoning is induced in LLMs with test-time compute techniques that enable models to "think". These underlying techniques that contribute to reasoning models, include Chain-of-thought or CoT (Wei et al., 2022b) which elicits reasoning, learning from environment feedback (Chen et al., 2024c) and exploring multiple execution paths (Yao et al., 2023a). These techniques are primarily inference time, but can include some training with model generated synthetic reasoning data as well. Many recent surveys explore reasoning techniques, SWE LLMs, benchmarks and Agents, and we discuss them in Sec. 2 and summarize the topics covered in Tab. 1. We did not find any survey that explores the impact of reasoning, and specifically code-based reasoning techniques on SWE tasks.

[1]We use SWE tasks, Code tasks and Software engineering tasks interchangeably.

Many reasoning techniques have been adapted for code specific tasks. Most of this work can be categorized into Chain-of-Thought related (Sec. 3.1), self-refinement related (Sec. 3.2) and inference scaling (Sec. 3.3) related. Agents, and in particular SWE agents (Sec. 3.4) provide a scaffolding for LLMs to use test-time compute for reasoning and solving some challenging tasks like GitHub issue resolution. Papers that use of a combination of these techniques are highlighted in Tab. 2. Fig. 1 shows a basic implementation of these test-time compute specific reasoning techniques. Code specific benchmarks and tasks (Sec. 4) are used to evaluate the effectiveness of one or more these reasoning techniques. This allows us to compare between different techniques (Sec. 5) and suggest some future directions (Sec. 7) based on some informative observations.

SWE is one of the most interesting applications areas of Artificial Intelligence (AI) and there is growing research in this space. As different reasoning techniques mature and agents become more robust, it is reasonable to expect more and more SWE tasks will be automated. With our survey on code reasoning for code tasks, we hope to address this gap by making the following contributions:

(1) The first survey specific to reasoning for coding tasks, emphasizing reasoning techniques which borrow ideas from coding principles (Sec. 3). SWE Agents are given a special focus (Sec. 3.4) since they often rely on multiple reasoning techniques.

(2) A Taxonomy covering different reasoning approaches and benchmarks for code Fig. 2. We also highlight approaches employing multiple reasoning techniques for LLMs and SWE agents (Tab. 2).

(3) Showcase benchmarks used to study the impact of reasoning on SWE tasks. We compiled comparison tables (Tab. 3, 5, 6, 7) showing the performance of different code reasoning and agentic approaches (Sec. 4.1). We also highlight promising benchmarks specific to code reasoning (Sec. 4.4), and surface some new agent-specific benchmarks with potential for furthering SWE research.

(4) Comparison and discussion of results from different reasoning techniques examined in the survey (Sec. 5). In Sec. 7, we use this discussion to motivate future work.

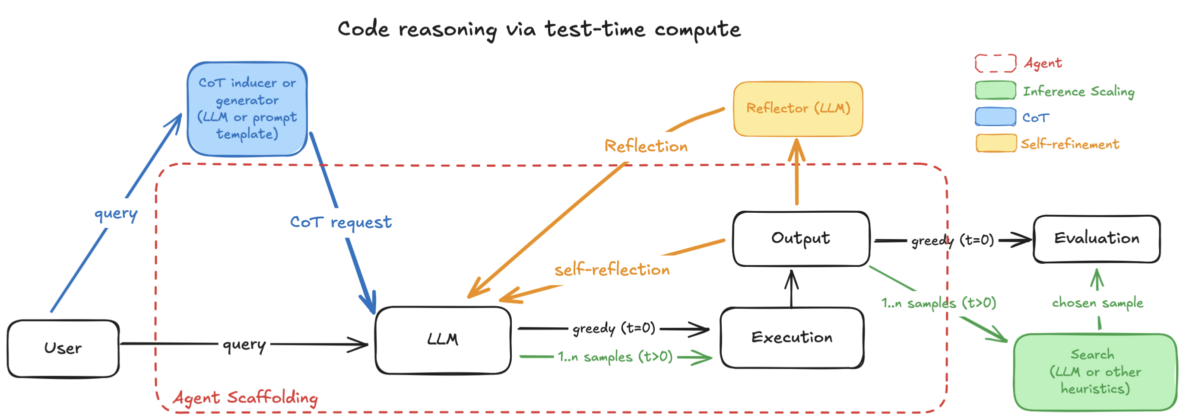

Figure 1: A simplified view of LLM inference for code tasks, illustrating both standard decoding and test-time compute-based reasoning techniques. The different colored regions indicate the core components of the reasoning methods covered in this survey. In standard inference, the user sends a query to an LLM, which produces a single greedy output (typically with temperature $t = 0$); for coding tasks, this output is executed and the resulting behavior is evaluated. CoT (Sec. 3.1) is induced by augmenting the user query with a reasoning-oriented prompt template or an auxiliary LLM. Inference Scaling (Sec. 3.3) generates multiple outputs by sampling with temperature $t > 0$, and uses search over these candidates before evaluation. Self-refine (Sec. 3.2) iteratively improves candidate outputs by having an LLM critique and revise them; the critic may be the same model (self-reflection) or a separate reflector model. Finally, an Agent (Sec. 3.4) orchestrates LLM calls and manages the execution environment, including tool use for interacting with code and external systems.

Survey	Covers Reasoning	Covers SWE Tasks	Covers Agents	Provides Taxonomy	Benchmarks Coverage
Dong et al., 2022	✓	✗	✗	✗	✗
Qiao et al., 2022	✔	✗	✗	✗	✗
Huang & Chang, 2022	✔	✗	✗	✗	✗
Zan et al., 2022	✗	✓	✗	✗	✗
Chu et al., 2023	✓	✓	✗	✗	✗
Jiang et al., 2024a	✓	✓	✗	✗	✗
Sun et al. (2024a)	✔	✔	✗	✗	✓
Plaat et al., 2024	✔	✗	✗	✗	✗
Wang et al. (2024b)	✗	✓	✗	✗	✓
Chen et al. (2024b)	✗	✔	✗	✗	✓
Yehudai et al. (2025)	✗	✔	✔	✗	✔
Xu et al., 2025	✔	✗	✗	✗	✗
Huynh & Lin, 2025	✓	✓	✗	✗	✗
Yang et al., 2025	✓	✓	✗	✗	✗
Mei et al., 2025	✔	✓	✓	✓	✓
Our Survey	✔	✔	✔	✔	✔

Table 1: Comparison of existing surveys along five dimensions: reasoning, SWE tasks, agents, taxonomy, and benchmark coverage. ✗ indicates that the topic is not covered at all. ✓ indicates partial coverage; for example, in the Reasoning column this may mean that only a single technique such as CoT is discussed, and in the SWE tasks column that only code generation is considered. ✔ indicates comprehensive, in-depth coverage, specifically for code and SWE tasks. Our survey provides in-depth coverage across all these dimensions for test-time compute-based code reasoning on SWE tasks.

2 Related Surveys

Wei et al., 2022b introduce CoT as a form of in-context learning which elicits reasoning in LLMs. In the same year, Dong et al., 2022 survey in-context learning techniques and reference CoT reasoning but do not expand on it. Qiao et al., 2022 and Huang & Chang, 2022 survey methods and tasks for reasoning and extensively study CoT and other prompting approaches, but do not include software engineering tasks. Chu et al., 2023 also cover CoT reasoning extensively in a recent work. They define a more general concept of XoT or X-of-Thought, which covers concepts like Program-of-Thought Chen et al. (2022), Tree-of-Thought Yao et al. (2023a) etc. apart from CoT. However, they focus on the impact of these techniques on reasoning benchmarks while we are more interested in how reasoning impacts code specific or software engineering benchmarks. Other recent surveys also cover different types of reasoning techniques for LLMs. Xu et al., 2025 discuss reinforcement learning based reasoning techniques, but they don't discuss code specific reasoning strategies. Plaat et al., 2024 classify the in-context reasoning approaches into prompting, evaluating and control (inference scaling and search) based strategies, but they don't focus on coding tasks.

In their work titled "Code to Think, Think to Code", Yang et al., 2025 highlight the interplay between code properties and reasoning capabilities and how one enhances the other. This survey makes the case that training with code related data improves performance on Math and reasoning benchmarks, while incorporating reasoning improves performance on coding benchmarks because some code properties reinforce reasoning capabilities and vice versa. Compared to this work, we dive deeper into reasoning techniques used for coding tasks and provide a taxonomy covering different strategies. Mei et al. (2025) provide a comprehensive survey on context engineering, which involves context retrieval, processing and management. Because of the board scope of their topic, they cover many aspects of SWE reasoning and tasks. However they have a much more general focus because of which they are unable to cover many aspects of code specific reasoning and different code related benchmarks and tasks.

A lot of surveys do cover impact of LLMs and Agents on Software Engineering tasks but none so far have focused on reasoning based strategies. Zan et al., 2022 survey 27 LLMs for natural language to code generation task. Jiang et al., 2024a undertake an extensive survey for code generation covering not just LLMs but also LLM architectures, many different research topics, benchmarks and datasets, encompassing a total of 235 papers. Sun et al. (2024a) also do a wide ranging survey covering 50 different models and their variants along with 20 different code-related task categories. Huynh & Lin (2025) survey many topics in this space including challenges and applications. Apart from surveys covering multiple topics from the domain of AI for code/software engineering, there are also surveys that are more topic specific. Wang et al., 2024b focus exclusively on reinforcement learning in code generation. Chen et al., 2024b survey different evaluation techniques for coding tasks. Yehudai et al., 2025 also focus on evaluation, but of LLM-agents and including Software Engineering (SWE) Agents.

We did not find any survey specific to code based reasoning techniques for software engineering tasks, covering agents and benchmarks and including a taxonomy.

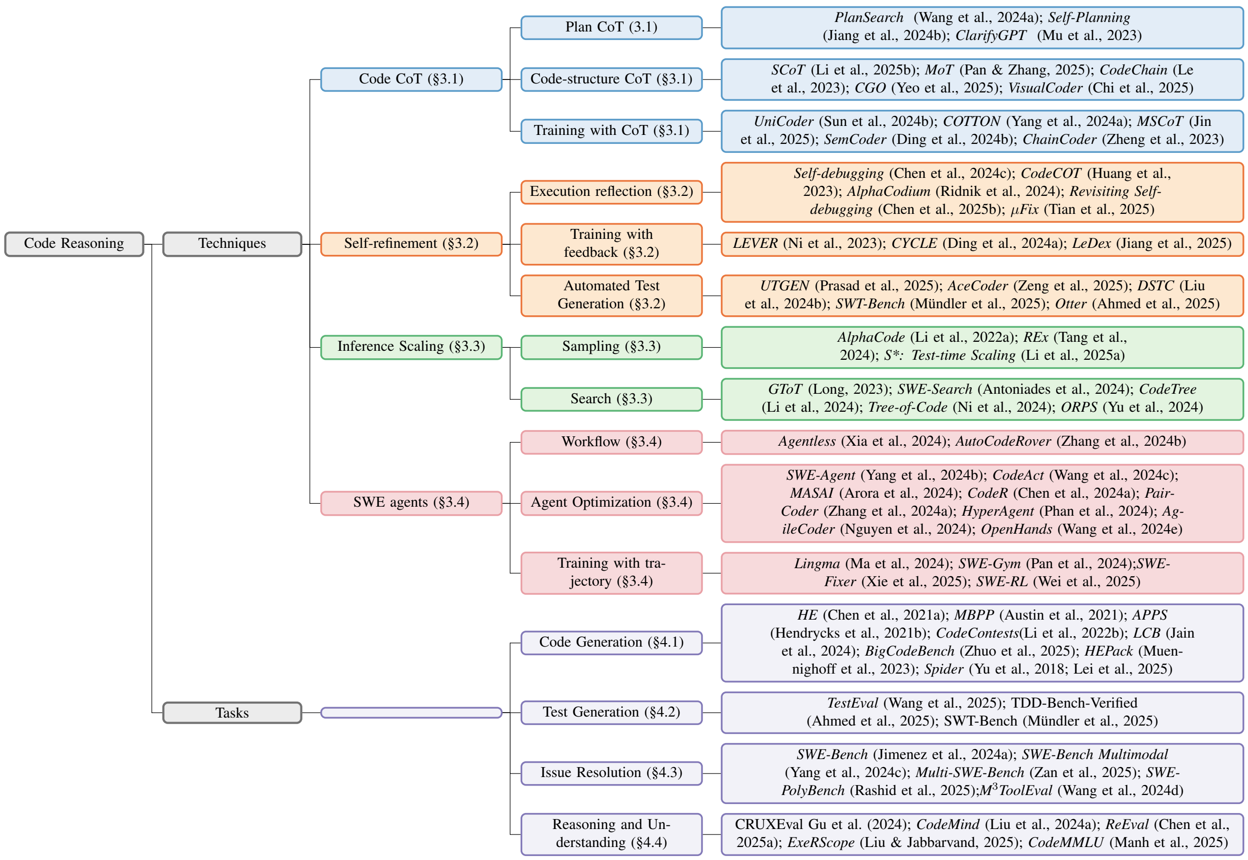

Figure 2: Code Reasoning Taxonomy. We organize prior work on code reasoning along two axes: *techniques* and *tasks*. Techniques are methods that elicit, enhance, or exploit reasoning in LLMs. Tasks (often instantiated as benchmarks) are used to evaluate an LLM's code-reasoning capability. Techniques are frequently combined within a single system; in this taxonomy, we categorize each publication by its dominant technique (i.e., the primary mechanism emphasized by the method). Table 2 highlights works that explicitly integrate multiple techniques.

3 Code Reasoning: Techniques

Brown et al., 2020 show that LLMs are few-shot learners. Performance of LLMs on reasoning tasks is further enhanced by a certain kind of prompting called Chain-of-thought or CoT (Wei et al., 2022b) prompting which elicits LLM reasoning. Wei et al., 2022a suggest that in-context learning ability of LLMs, including CoT reasoning, is an emergent property of LLMs. Code CoT papers (Li et al., 2025b; Jiang et al., 2024b; Pan & Zhang, 2025 and others) suggest that code reasoning is a specific kind of reasoning and CoT can be more impactful when induced with prompts that recognize this difference. We survey such techniques in Sec. 3.1.

One way code output is different from natural language output is that it can be executed and tested to validate its correctness. Self-refinement techniques using code execution as a way to evaluate and improve LLMs' output, such as Self-debugging (Chen et al., 2024c), CodeCoT (Huang et al., 2023), LEVER (Ni et al., 2023) and others, are covered in Sec. 3.2.

Yao et al., 2023a state that "System 2" thinking should involve exploring diverse solution paths rather than greedily picking one. They connect CoT with inference scaling to enable exploration of multiple reasoning paths. Inference scaling involves setting a temperature greater than 0 to generate multiple candidate solutions or samples during inference or test time, and then picking the best candidate. Li et al., 2022a effectively leverage this technique to generate competition level code. Sec. 3.3 covers inference scaling used to explore multiple reasoning paths for software engineering tasks.

Many approaches use a combination of these techniques, although one technique usually dominates. Tab. 2 shows approaches which rely on multiple techniques.

3.1 Code Chain-of-thought

CoT prompts for code can be categorized as plan-based or structure based. Plan-based CoT is a natural language articulation of steps that need to be taken to solve a coding problem. Code-structure-based CoT utilizes some code structure or programming concepts. Besides these two prompting only techniques, another approach used by many is fine-tuning or instruction tuning for software engineering tasks with code CoT data.

Plan CoT. Several recent approaches enhance code generation by explicitly modeling intermediate reasoning or problem understanding steps. For instance, `PlanSearch` Wang et al. (2024a) generates 3–6 problem observations, combines them into natural language plans, and translates these into pseudocode and then code. `Self-Planning` Jiang et al. (2024b) uses few-shot prompting to extract a high-level plan from the problem, which guides code generation. `ClarifyGPT` Mu et al. (2023) employs test generation to construct clarifying questions and answers that are appended to the prompt for code synthesis.

Code-structure CoT. In `SCoT`, Li et al., 2025b use programming structures, like sequence, branch and loop, as steps towards intermediate code, which is used to prompt the model to generate code. `Chain of grounded objectives (CGO)` Yeo et al. (2025) embed appropriately structured functional objectives into the input prompts to enhance code generation. Pan & Zhang, 2025 propose a novel prompting technique, `Modularization-of-thought (MoT)`, which exploits modularization principles to decompose complex programming problems into smaller independent reasoning steps, via a multi-level reasoning graph. Le et al., 2023 also elicit modularized code generation but in a multi-step technique called `CodeChain`, which is a chain of self-revisions applied by picking potentially correct representative sub-modules. Recently, VisualCoder Chi et al. (2025) integrated multimodal CoT reasoning with a visual Control Flow Graph (CFG) to obtain deeper insights into execution flows and improve code execution reasoning and program repair generated solutions.

Training with CoT. Sun et al., 2024b define `UniCoder`; they use an intermediate representation CoT based on PL conventions and use this to instruction-tune a model on a multi-task learning objective. Yang et al., 2024a generate high-quality CoTs based on the `COTTON` framework, which trains light-LMs (< 10B parameters) to generate CoT comparable to those generated by strong teacher LLMs. `ChainCoder` Zheng et al. (2023) generates code iteratively in a "course-to-fine" approach and trains a model using an AST-based vocabulary. `SemCoder` Ding et al. (2024b) uses a monologue reasoning approach to train a model to learn program semantics, which is generated by asking the Code LLM to summarize the program functionalities, key properties and constraints, and reason about code execution step-by-step using a bi-directional monologue reasoning method. `MSCoT` Jin et al. (2025) extends SCoT Li et al. (2025b) to 11 more programming languages beyond Python; a trained MSCoT model generates structured-CoT before producing code in multiple languages.

3.2 Self-refinement

Self-refinement involves executing LLM-generated code in a given environment and having the same or a different LLM reason about the execution environment output. This reasoning can be fed back to the LLM to refine the code.

Execution reflection. These strategies utilize code execution feedback to select the final prediction from a LLM. In Chen et al. (2024c), the `Self-debugging` approach, teaches the model to self-debug i.e., debug the model's predicted code, via few shot prompting and without additional model training. A similar approach was taken in Code Chain-of-Thought (`CodeCoT`) by Huang et al. (2023), where CoT is used as a first step to generate the code, then a LLM generates test cases to validate whether the code has syntax errors during the execution. `AlphaCodium`, proposed by Ridnik et al. (2024), is a flow with two key phases, (a) pre-processing to generate reflection and (b) iterative code generation, to improve code LLM performance that does not require training a model. xIn `revisited self-debugging` Chen et al. (2025b) authors explored both post-execution and in-execution self-debugging, leveraging self-generated tests. More recently, Tian et al. (2025) proposed μFix (Misunderstanding Fixing) where thought-eliciting prompting techniques are combined with feedback-based prompting to improve the code generation performance of LLMs.

Training with feedback. We pinpoint approaches that train an LLM, leveraging execution data, to improve model performance. LEarning to VERify Ni et al. (2023) (LEVER) is an approach where verifiers are trained to check whether the generated code is correct or not based on three sources of information: the natural language input, the program itself, and its execution results. CYCLE Ding et al. (2024a) trains code LLMs to self-refine using natural language specifications, generated code, and execution feedback, while avoiding repeated errors via a Past Generation Mask. Similarly, Jiang et al. (2025) proposed LEDEX, a training framework to improve the self-debugging capability of LLMs using a chain of explanations on the wrong code followed by code refinement.

Automated Test Generation. Unit Tests (UT) are one of the fundamental pieces to assess the correctness of code and give execution-based feedback to code generation models. UTGEN Prasad et al. (2025) is a data creation and training recipe that bootstraps training data for UT generation and works by perturbing code to simulate errors, generating failing tests and augmenting it with CoT rationales.

AceCoder Zeng et al. (2025) leverages automated large-scale test-case synthesis to enhance code model training. They proposed a pipeline that generates extensive *(question, test-cases)* pairs from existing code data. Similarly, Liu et al. (2024b) propose Direct Preference Learning with Only Self-Generated Tests and Code (DSTC), using only self-generated code snippets and tests to construct preference pairs with direct preference learning to improve LM coding accuracy without external annotations. ASTER Pan et al. (2025a) is a multilingual UT-generator built with LLMs guided by lightweight program analysis.

3.3 Inference Scaling

Several approaches to code generation, code repair, and test-case generation use *tree-based* strategies to guide decisions and explore reasoning paths, while others use sampling.

Sampling. In AlphaCode, Li et al. (2022a) filter and cluster samples according to program behavior on model-generated test inputs, selecting one candidate per cluster. The authors of REx (Tang et al., 2024) frame iterative code repair, or *refinement*, as a multi-armed bandit problem which is solved using Thompson sampling. In S*, Li et al. (2025a) take a hybrid sampling approach, first generating N diverse programs in parallel then refining them using iterative debugging (informed by execution). CodeTree (Li et al., 2024) and ToC (Ni et al., 2024) both model reasoning as tree search—CodeTree combines planning, execution-guided reasoning, and heuristics (test-pass rate, LM critique) via multi-agent roles, while ToC uses a binary pass/fail heuristic with reflective, multi-strategy execution for diverse solutions.

Search. Tree-of-Thoughts (ToT) (Yao et al., 2023a) allows LMs to explore multiple reasoning paths over thoughts, where thoughts are language sequences that serve as intermediate steps towards problem solutions and represent the states or nodes of the tree. Similarly, Guided tree-of-thought (GToT) (Long, 2023) uses tree-search guided by an LLM heuristic; it generates intermediate solutions through prompting, employs a checker to validate these solutions, and uses a controller to manage search and backtracking, enabling long-range reasoning. For test generation, Ouédraogo et al. (2024) show that GToT effectively produces syntactically-correct, compilable test suites with superior code coverage. Yu et al. (2024) propose Outcome-Refining Process Supervision (ORPS), a beam-search approach for code generation over a "reasoning tree". SWE-Search (Antoniades et al., 2024) is a moatless-tools (Orwall, 2024) based multi-agent framework which integrates Monte-Carlo Tree Search with self-improvement for bug-fixing.

3.4 SWE agents

Agentic systems use many of the reasoning techniques described in Sec. 3 for different tasks. Software Engineering (SWE) agents take a programming problem and iteratively solve it by self-debugging based on the feedback provided by the environment. The self-debugging is enabled by CoT style natural language reflection (Shinn et al., 2023) on environment feedback. The reasoning is done by an LLM which interacts with the agent execution environment with tool calls (Yao et al., 2023b).

Approach	Code CoT			Self-refinement			Inference Scaling		SWE agents		
	Plan	Struct	Train CoT	Exec. ref.	Train feedback	ATG	Sampling	Search	Workflow	Agent Opt.	Train trajectory
AlphaCode (2022a)							✓	✓			
ClarifyGPT (2023), Self-Planning (2024b)	✓										
CodeChain (2023), SCoT (2025b), CGO (2025), MoT (2025), VisualCoder (2025)		✓									
ChainCoder (2023), UniCoder (2024b), MSCoT (2025)		✓	✓								
LEVER (2023), Self-Debugging (2024c) , ORPS (2024), REx (2024)				✓			✓	✓			
CodeCoT (2023), AlphaCodium (2024)	✓			✓		✓					
GToT (2023)							✓	✓			✓
PlanSearch (2024a)	✓			✓			✓				
COTTON (2024a)	✓		✓								
SemCoder (2024b)		✓	✓			✓					
CYCLE (2024a)				✓	✓		✓				
DSTC (2024b)					✓	✓					
CodeTree (2024), Tree-of-Code (2024), SWE-Search (2024)							✓	✓		✓	
Agentless (2024)				✓					✓		
AutoCodeRover (2024b), PairCoder (2024a)				✓					✓	✓	
CodeAct (2024c)				✓						✓	✓
OpenHands (2024e), MASAI (2024), CodeR (2024a), AgileCoder (2024), HyperAgent (2024), SWE-Agent (2024b)				✓						✓	
Lingma (2024), SWE-Fixer (2025)				✓					✓		✓
SWE-Gym (2024)				✓			✓	✓		✓	✓
Revisiting Self-Debugging (2025b) , S* (2025a)				✓		✓	✓	✓			
μFix (2025)	✓			✓							
LeDex (2025)	✓			✓	✓		✓				
UTGEN (2025), AceCoder (2025)					✓	✓	✓				
SWT-Bench (2025), Otter (2025)	✓					✓	✓			✓	
SWE-RL (2025)											✓

Table 2: This table summarizes the test-time compute-based reasoning techniques used in the papers surveyed. For each work, we assign a dominant strategy for categorization as shown in the taxonomy Fig. 2. Additional techniques used by the same approach are marked with ✓. For example, PlanSearch is categorized under Code CoT as its dominant strategy, but it also incorporates elements of Self-refinement and Inference Scaling.

Workflow. Schluntz & Zhang, 2024 draw a distinction between Agents and LLM-based workflows stating that the latter are simpler, have a fixed path and do not require an LLM to make a decision. Agentless (Xia et al., 2024) is a three step process for Github issue resolution involving localization, repair and patch validation. AutoCodeRover (Zhang et al., 2024b) uses program structure, in the form of an Abstract Syntax Tree (AST), to enhance code search and look at a software project as classes and functions, rather than as a collection of files.

Agent Optimization can often lead to performance gains. There can be many ways to improve an SE agent, including but not limited to, better environment management or agent-environment interface, improved workflow or architecture, and incorporating more tools. SWE-Agent (Yang et al., 2024b) is an agent capable of editing repository-level code by generating a thought and a command, and subsequently incorporating the feedback from the command's execution into the environment. In CodeAct, (Wang et al., 2024c) propose to use executable Python code to consolidate LLM agents' actions into a unified action space. OpenHands (Wang et al., 2024e) is a platform for developing flexible AI agents that interact with the digital world the same way a human would, by writing code, interacting with the command line or browsing the web. This platform allows for integration of other specialist agents, like CodeAct (Wang et al., 2024c) for software engineering. There are other multi-agent techniques like MASAI Arora et al. (2024), CodeR Chen et al. (2024a), PairCoder Zhang et al. (2024a), HyperAgent Phan et al. (2024) and AgileCoder Nguyen et al. (2024).

Training with trajectory. Some agentic systems improve the underlying reasoning model by training on agent trajectories, which include steps like CoT, tool calls, and patches. Ma et al. (2024) note that software evolution spans code, reasoning, tools, and cross-role interactions, and fine-tune their `Lingma` SWE-GPT models (7B, 72B) on repository understanding, bug localization, patching, and rejection sampling from pull requests. Pan et al., 2024 build `SWE-Gym` from 2,438 real-world Python tasks—each with a runnable codebase, unit tests, and an NL spec. Using OpenHands scaffolding (Wang et al., 2024e), they fine-tune Qwen2.5-Coder-32B (Hui et al., 2024) on 491 agent–environment trajectories and train a verifier on the same data for scalable inference. `SWE-Fixer` (Xie et al., 2025) uses a fine-tuned Qwen2.5-7B retriever, boosted with BM25, to identify relevant files, while a fine-tuned Qwen2.5-72B editor generates patches for GitHub issues. In `SWE-RL` (Wei et al., 2025), Llama 3 (Grattafiori et al., 2024) is trained with lightweight rule rewards and GRPO (Shao et al., 2024) on 11M filtered PRs, producing Llama3-SWE-RL-70B, the top medium-sized model on SWE-bench Verified (OpenAI, 2024) upon release.

4 Code Reasoning: Tasks

In this section we discuss the different tasks and benchmarks which are used to evaluate code reasoning techniques described in Sec. 3.

4.1 Code Generation

For code generation, a popular task, most common benchmarks include *HumanEval (HE)* (Chen et al., 2021a), `HumanEvalPack` (Muennighoff et al., 2023), *MBPP* (Austin et al., 2021), *APPS* (Hendrycks et al., 2021b), and *CodeContests* (Li et al., 2022b).

More recently, *LiveCodeBench (LCB)* (Jain et al., 2024) collected new problems for over time from contests platforms including LeetCode, AtCoder, and CodeForces. *BigCodeBench* (Zhuo et al., 2025) challenges LLMs to invoke multiple function calls as tools from multiple libraries and domains for different fine-grained tasks. CRUXEval (Gu et al., 2024) includes both input and output predictions to evaluate code reasoning and code execution, respectively. ConvCodeBench (Han et al., 2025) is a benchmark for interactive code generation, it uses pre-generated feedback logs, avoiding costly LLM calls for verbal feedback while maintaining strong correlation with live results; *Spider* (Yu et al., 2018; Lei et al., 2025) is a benchmark to evaluate the generation of SQL queries from natural language.

4.2 Test Generation

For test generation, benchmarks like *TestEval* (Wang et al., 2025) can help on three different aspects: overall coverage, targeted line/branch coverage, and targeted path coverage. *SWT-Bench* (Mündler et al., 2025) is another github based test-generation benchmark; Otter, too, (Ahmed et al., 2025) proposed an LLM-based solution to generate test cases from issues. [2]

4.3 Issue Resolution

For Github issue resoultion, *SWE-Bench* (Jimenez et al., 2024a) is a popular benchmark. Other variations of SWE-Bench include: *SWE-Bench Multimodal* (Yang et al., 2024c) for visual and user-facing components, and *Multi-SWE-Bench* (Zan et al., 2025) and *SWE-PolyBench* (Rashid et al., 2025) for more programming languages besides Python. *M^3ToolEval* (Wang et al., 2024d) is used for multi-turn, multi-tool complex tasks.

4.4 Reasoning and Understanding

Evaluating the ability of LLMs to both correctly and soundly reason about runtime behavior of code can help understand and verify whether the generated code aligns with the intended goal. *ReEval* (Chen et al., 2025a) helps to analyze how Code LLMs reason about runtime behaviors (e.g., program state, execution paths) of programs. ExeRScope (Liu & Jabbarvand, 2025) analyzes the code execution reasoning output from LLMs for different code reasoning benchmarks and helps understand the impact of code properties (program constructs, complexity, dynamic program properties, and variable types) for such benchmarks. *CodeMMLU* (Manh et al., 2025) is a large benchmark to evaluate both code understanding and code reasoning through a multiple-choice question-answering approach. CodeMind

[2] Appendix A.3 lists metrics that can be used to assess code LLM performance.

(Liu et al., 2024a) is a code reasoning benchmark for LLMs, evaluating Independent Execution Reasoning (IER), Dependent Execution Reasoning (DER), and Specification Reasoning (SR) tasks and metrics.

5 Comparison and Discussion

The best way understand the impact of different code reasoning techniques, is an exhaustive comparative study with common environmental setup. Since this is impractical and in-order to draw fair conclusions across heterogeneous experimental setups, we compare the reported performance on coding benchmarks while considering the same models. All the reported results for the code reasoning techniques described in Sec. 3 on benchmarks and tasks described in Sec. 4 are shown in tables 3, 4, 5, 6, 7. We plot technique performance on the intersection of reported models and benchmarks, which makes cross-model and cross-benchmark trends easier to see. This comparison is not exhaustive; it is restricted to benchmarks and models shared by a subset of techniques.

5.1 Plan CoT vs. Code Structure CoT

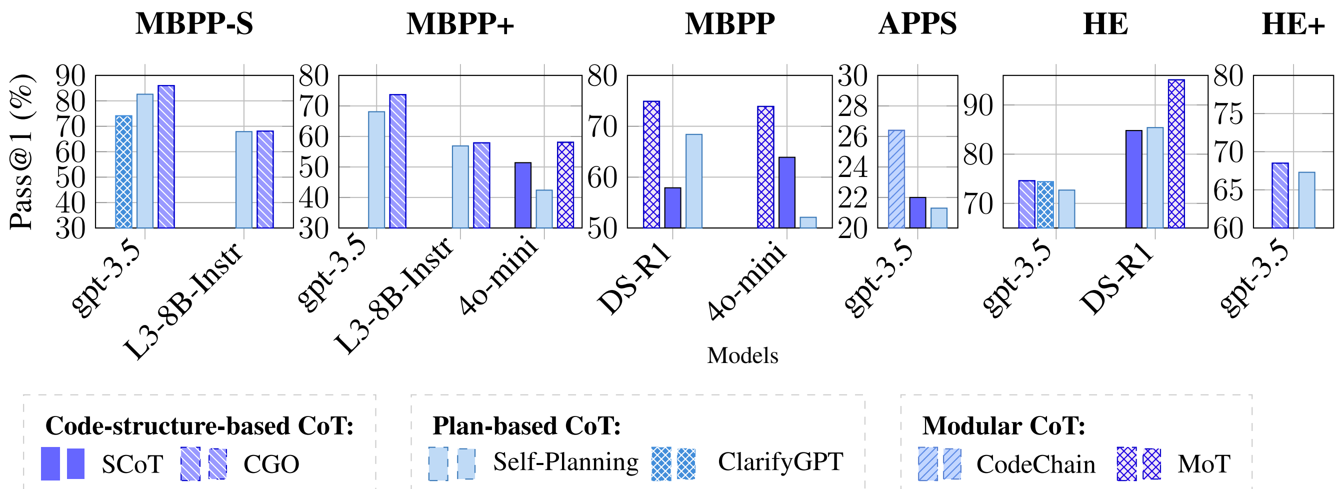

Figure 3: Structure-aware CoT vs. planning-based CoT on benchmarks MBPP-S, MBPP+, MBPP, APPS, HE, and HE+. Modular CoT is a sub-category of Structure-aware CoT. Overall we see that Modular CoT outperforms code structure-based CoT, which outperforms plan-based CoT.

Section 3.1 surveys works that formulate Code CoT prompting into plan-based and structure-based. Fig. 3 shows the performance of different plan and structure based techniques on code generation benchmarks (Sec. 4.1). Chain of Grounded Objectives (CGO) outperforms Self-Planning with gpt-3.5 on MBPP-S, MBPP+, HE and HE+. This also holds true for Llama-3-8B-Instr, where CGO is better than Self-Planning on MBPP-S and MBPP+. CGO is also better than ClarifyGPT with gpt-3.5 on MBPP-S and HE. On MBPP, MBPP+ and APPS with gpt-4o-mini, SCoT is better than Self-Planning. CodeChain outperforms SelfPlanning on APPS with gpt-3.5. CGO, CodeChain and SCoT, which are code structure based techniques, outperform Self-planning and ClarifyGPT, which are plan based techniques. The results suggest that structure-aware strategies outperform plan-based approaches.

Observation 1: Structure-aware CoT strategies are better than planning-based CoT strategies, for code generation task..

5.2 Modular Code Structure CoT

Code structure aware CoT can be sub-categorized into modular approaches, as shown in Fig. 3. Modularity improves upon structure-based CoT by providing ultra-localized scoping; with more clearly defined and specific functionality, modularity eliminates the chance of error propagating to subsequent steps. MoT and CodeChain are modular techniques, a more specific type of structure away CoT. MoT outperforms SCoT and Self-Planning with DS-R1 on MBPP and HE, This is also true for MBPP and MBPP+ with gpt-4o-mini. CodeChain also outperforms SCoT and Self-Planning on APPS with gpt-3.5. Building on Observation 1, we see that modular formats outperform other structure-aware Code CoT prompting techniques on code generation.

Observation 2: Modularity helps in CoT, as is evident when modular techniques dominate other structured and plan-based CoT approaches, for code generation.

5.3 Self-refinement vs. Code CoT

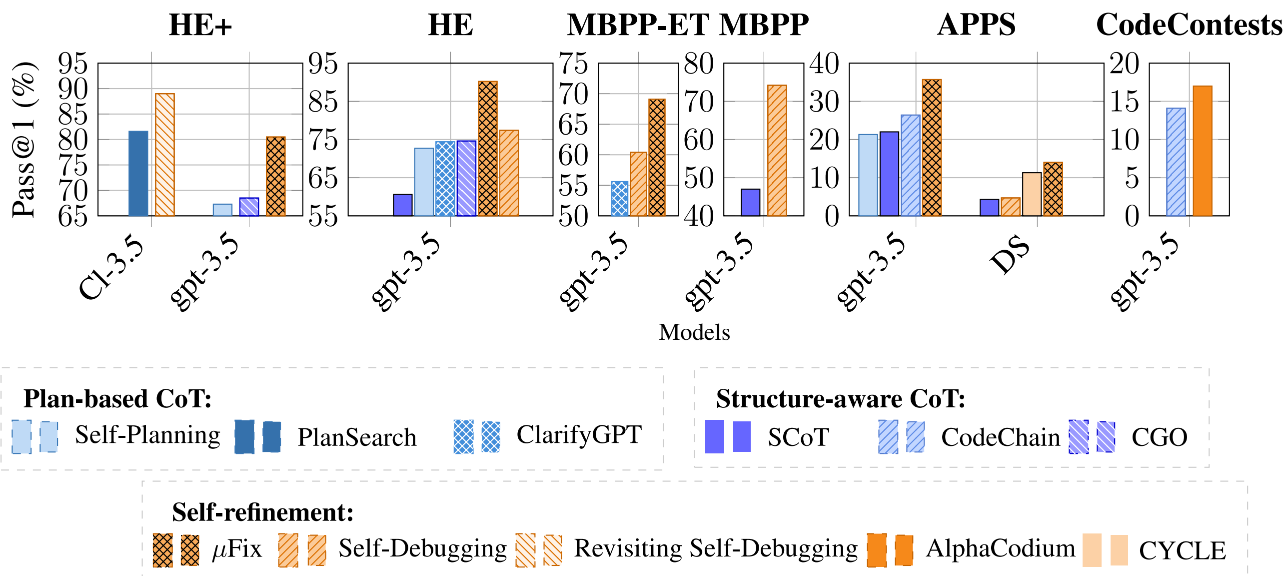

Figure 4: Comparison between code CoT and self-refinement techniques on code generation benchmarks. Code CoT is sub-categorized into plan-based and structure-ware CoT. Self-refinement techniques outperform code CoT based techniques.

Self-refinement (Sec. 4.4) uses the execution output as feedback to iteratively improve the model generated code. Incorrect intermediate code is discarded and the model can improve upon it. As can be seen in Fig. 4, using self-refinement has a bigger impact on code generation task than any code CoT based technique. This is true across different benchmarks and models.

μFix and Self-Debugging surpass other CoT baselines (CGO, SCoT, Self-Plan, Clarify-GPT) on HE (gpt-3.5). Revisiting Self-Debugging beats PlanSearch on HE+ (Claude-3.5). μFix and Self-Debugging outperform ClarifyGPT on MBPP-ET (gpt-3.5). On MBPP with gpt-3.5, Self-Debugging surpasses SCoT by a large margin. μFix and Self-Debugging outperform UniCoder on HE. The findings hold true on the APPS benchmark, where μFix outperforms CodeChain, SCoT, and Self-Planning with gpt-3.5. This is true for DeepSeek-Coder as well, where μFix, Self-Debugging, and CYCLE models, which are smaller-sized parameter models but finetuned, outperform SCoT.

Observation 3: On code-generation task, execution-aware strategies consistently outperform Code CoT based methods.

5.4 Inference Scaling

Inference Scaling (Sec. 3.3) involves generating multiple LLM outputs and then selecting the best candidate among them for evaluation. In Fig. 5, we can see that inference scaling outperforms code CoT on code generation task. We found only one point of comparison between inference scaling and self-refinement, and on which inference scaling does better.

ORPS outperforms MoT and other structure-based and plan-based approaches (like SCoT and Self-Planning) on MBPP with gpt-4o-mini. REx with gpt-4 also claims to achieve the state-of-the art on APPS, with roughly 70%, comfortably beating CodeChain. S* also beats PlanSearch on LCB with o1-mini and 4o-mini. Interesting to note that

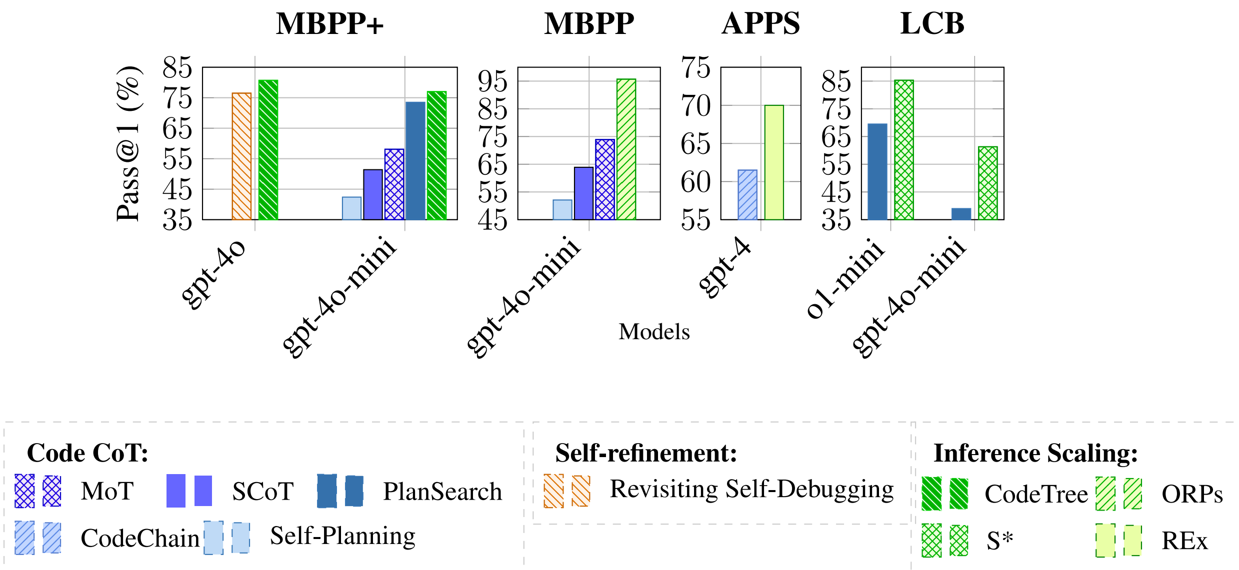

Figure 5: Performance comparison between Inference Scaling, Code CoT and Self-refinement on Code Generation benchmarks. Inference Scaling outperforms code CoT based approaches. Inference scaling also outperforms self-refinement on MBPP+ with gpt-4o.

PlanSearch, which incorporates some inference scaling techniques as highlighted in Tab. 2, outperforms other Code CoT methods by a big margin on MBPP+ with gpt-4o-mini. CodeTree outperforms Revisiting Self-Debugging on MBPP+ with gpt-4o, which is the only point of comparison between inference scaling and self-refinement. CodeTree also does better than multiple Code CoT methods on MBPP+ with gpt-4o-mini.

Observation 4: Approaches that integrate inference scaling outperform CoT-dominant strategies on code generation. Inference scaling can also outperform self-refinement based strategies.

5.5 SWE agents

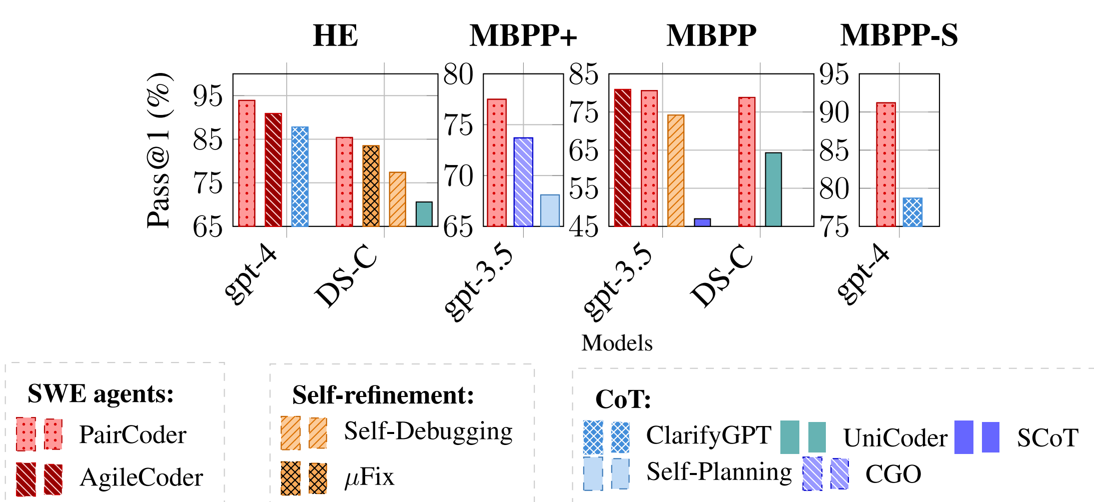

Figure 6: Comparison of SWE agents-based approaches with self-refinement and code CoT-based approaches on code generation benchmarks. SWE agents outperform self-refinement and CoT based techniques.

SWE agents (Sec. 3.4) succeed by integrating chain-of-thought reasoning, execution-based validation, and sampling into a unified framework–thus leveraging code's structured syntax, executable semantics, and error feedback all in one. Fig. 6 shows that agent based approaches perform better on code generation compared to self-refinement and code CoT.

PairCoder and AgileCoder outperform ClarifyGPT with gpt-4 on HE. PairCoder is better than CGO and Self-Planning on MBPP+ with gpt-3.5. Both PairCoder and Agile coder are better than SCoT on MBPP with gpt-3.5; both dominate Self-Debugging as well. With DeepSeek-coder on HE, Paircoder outperforms μFix, Self-Debugging, and UniCoder; also with DeepSeek-Coder, PairCoder outperforms UniCoder on MBPP. This is also true for gpt-4 on MBPP-S, where PairCoder outperforms ClarifyGPT.

Observation 5: Agentic approaches appear to dominate both self-refinement and CoT strategies on code generation

5.6 SWE agents with Inference Scaling

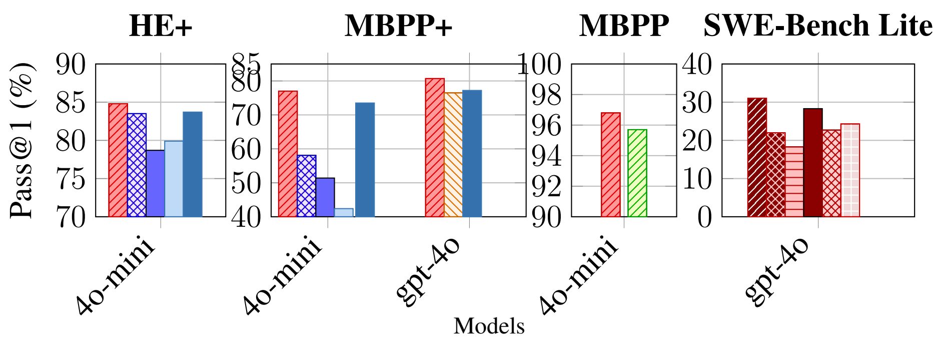

Figure 7: Comparison of agentic + inference scaling approaches with SWE agents, self-refinement, and inference scaling.

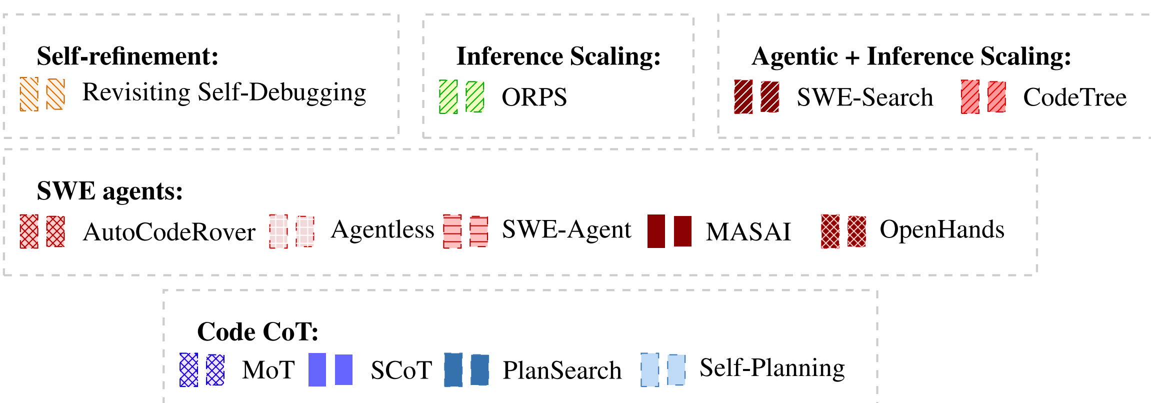

Figure 8: Legend for agentic, self-refinement, and inference-scaling methods.

SWE agents generally have some elements of CoT and self-refinement as part of the scaffolding. Techniques that combine inference scaling with SWE agents have also been proposed. Fig. 7 shows that such techniques which combine inference scaling with SWE agents perform better than SWE agents, self-refinement and inference scaling based approaches.

CodeTree outperforms MoT, SCoT, Self-Planning, and PlanSearch on HE+ and MBPP+ with gpt-4o-mini; CodeTree outperforms these strategies with gpt-4o as well. CodeTree also outperforms ORPS on MBPP with gpt-4o-mini. On M^{3}ToolEval, ToC is better than CodeAct. Moreover, SWE-Search, which combines inference scaling in an agentic approach, dominates the leaderboard on SWE-Bench Lite.

Observation 6: Agentic approaches that scale inference with search are highly competitive and can even outperform other strategies, on both code generation and issue resolution tasks.

6 Gaps and Future Directions

Sec. 3 gave an overview of various inference time code reasoning techniques and in Sec. 5 we saw how this impacted code tasks, primarily code generation but also issue resolution. Despite the vast array of work in this domain covered by our survey, there still appear to be gaps which can be explored by the research community. In this section we try to motivate why some of these gaps must be explored and identify a set of specific Call To Actions (CTA) as specific research directions.

6.1 Structure-based CoT understanding and applications

Section 3.1 surveys works that formulate CoT in plan-based, structure-based, and modular methods. As per Sec. 5.1 and Sec. 5.1, structure-based methods outperform plan-based methods and modular methods outperform other Code CoT prompting methods. To further examine this result, first we must understand why chain-of-thought (CoT) prompting helps over direct prompting. One hypothesis from Prystawski et al. (2023)'s work provides theoretical and experimental evidence that intermediate steps (i.e., chain-of-thought reasoning) reduce bias in transformers. They show that when training data has local structure (as textual data does), intermediate variables (CoT) can outperform direct prediction (no CoT). This suggests that **CoT reasoning helps most when a model is asked to make inferences about concepts that do not co-occur in the training data, but which can be chained together through topics that do.** Section 3.1 surveys works that formulate CoT in plan-based, structure-based, and modular arrangements. The results suggest that structure-aware strategies outperform plan-based approaches, and modular formats outperform structure-aware ones. We posit that because code has properties of *structured syntax*, the primitive structures invoked within the CoT are highly local in the training data. Structures (such as idents, branches, loop invariants, functions, etc) are seen countless times in the training corpus. The model's ability to estimate probabilities (and thus its ability to arrive at a correct solution) become sharper by eliciting these localities. Modular structures may push this same principle further. Based on this thesis we present out first call-to-action below.

CTA-1: Deeper research on why on code structure and it's impact on code tasks will help confirm some of these hypothesis and may lead to more improvement.

Code structure improving CoT result should encourage research in exploiting other code properties. Current Code CoT approaches consider a structured representation of the code and also the reasoning and generation of modular solutions. Code CoT approaches should also consider other mechanisms for generating code such as logic programming or event-driven programming (when it applies). Changing the programming paradigm will modify the way the LLM reason about the solution and might help generate code more aligned to a given specification.

CTA-2: Code CoT that considers other programming paradigms and software design principles may bring further improvements for some code tasks.

Inference scaling approaches (Sec. 3.3) sample multiple solutions from LLMs, selecting the best solutions using different strategies. Table 2 shows many approaches use a combination of inference scaling, self-refinement and SWE agents. However there is little to no exploration of inference scaling and Code CoT approaches. The only technique that explore this is PlanSearch which does better than other code CoT techniques on MBPP+ with gpt-4o-mini (Fig. 5). Works on inference scaling can also leverage other strategies from CoT such as structure of code and modular code generation that have shown to increase the correctness of the generated solutions, compared to plan-based CoT approaches.

CTA-3: Combination of inference scaling and code-structure based CoT approaches can outperform comparable methods.

6.2 Self-refinement

Self-refinement (Sec. 3.2) involves executing the model generated code and then feeding the execution output back to the model for iterative improvement. This technique outperforms code CoT prompting techniques (Sec. 5.3). We posit that execution may help because executing code can be used as a deterministic check. Any chain that violates the check can be discarded. Hence, bad chains are filtered out, so variance may collapse faster. However, even with reduced variance, LLMs can still exhibit issues, such as model rigidity. Because code is inherently deterministic (i.e. under certain assumptions, a given input consistently produces the same output), it can lead models to develop rigid generation patterns in training. For example, Twist et al. (2025) show that LLMs exhibit a strong bias towards certain programming languages, like Python; Liu et al. (2025) document the pervasiveness of repetition in LLM-based code generation, where models often reproduce patterns observed in training. Zhang et al. (2025) demonstrate that LLMs favor certain libraries and APIs by default, reflecting the distribution of their training corpora. Furthermore, Pan et al. (2025b) show that LLMs struggle to generalize to the architectural design principles of given projects, leading to the generation of conflicting code. This phenomenon compels the integration of search in order to explore diverse trajectories, which explains the recent success of inference scaling techniques.

CTA-4: More research is required to confirm if execution is a deterministic check on model rigidity. This could guide better training data construction and overall model quality improvement.

So far self-refinement approaches have considered execution and unit tests as feedback to self-refine the generated code. To increase the robustness of the generated solutions, these approaches could also reason about other characteristics of the generated code such as efficiency, security, and usability, among others. When generating modularized code, besides reasoning about the functionality of the solution, approaches could also consider reasoning about other important aspects such as maintainability and scalability of the proposed code solution. Mechanisms to validate these properties exist in the generated code can also be used as feedback. Besides unit tests, feedback from integration tests should be considered to further evaluate code modules that were tested as separate units (e.g., with unit testing) work correctly also when interacting with each other.

CTA-5: Self-Refinement techniques could also consider software qualities and Integration tests.

6.3 SWE agents Improvements and Benchmarks

SWE agents (Sec. 3.4) orchestrate LLMs, tools and execution environments resulting in state-of-the-art performance (Sec. 5.5 and Sec. 5.6) on code generation and relatively more challenging issue resolution task. Many approaches covered in our survey also involve multi-agent solutions. Although enabling good results, such agent and multi-agent appraoches also increase the degree of complexity of AI systems. Such systems will not be immune to errors, which can propagate through the system and stifle its performance. It is possible that many errors are common across LLMs, agents and tasks. A better understanding and development of techniques to overcome these errors can result in improve performance of multiple systems on multiple tasks.

CTA-6: Incorporating a structured way to analyze errors in agents can help them overcome repetitions of the same error, find alternative methods to solve a given step, avoiding waste of resources and increasing their ability to find a solution.

Overall, patterns of errors made by agents have recently become a topic of growing interest. For instance, Cuadron et al. (2025) examined error patterns in extended internal reasoning chains (i.e., overthinking) in reasoning models, while frameworks such as TRAIL (Deshpande et al., 2025) have focused on categorizing the errors exhibited by agents, and judges the ability of agents to label existing errors in categories. However, these efforts have largely remained at the level of descriptive taxonomies of existing errors. What remains missing is a concrete error benchmark that evaluates agents' ability to recover from their own mistakes or fix injected mistakes. Such a benchmark could be constructed by injecting error instances or categories into a boilerplate agent performing clean tasks, with evaluation metrics including both the raw competence ceiling and the category-wise error recovery rate, alongside latency costs (i.e., the number of steps or seconds required for the agent to recover).

CTA-7: A promising direction for benchmark development would be to design an error benchmark that explicitly measures self-induced error rates versus recovery error rates.

Beyond agents' errors, another critical frontier is benchmarking tool-use, particularly for SWE agents. While works such as GAIA (Mialon et al., 2023) provide benchmarks for general tool use in generic AI agents, there remains a gap in specific tool-use benchmarks tailored to SWE agents which targets the challenge of system orchestration, specifically. Furthermore, recent work built taxonomies of agentic decision-making pathways (Ceka et al., 2025) showing that agents continue to struggle with certain complexities, performing especially poorly on more advanced SWE-bench problems. They also highlight the presence of "hot nodes"(i.e., critical components in these pathways) that strongly influence whether an agent can successfully generate patches. Based on the aforementioned work, two additional benchmark directions could be: (i) benchmarks that explicitly target complex issues that agents consistently fail on, and (ii) benchmarks, and approaches, that emphasize the specific components of agent decision-making that are most predictive of successful outcomes. For example, TDD-Bench (Otter) (Ahmed et al., 2025) isolates reproduction test case generation as one such critical sub-component of SWE Agents.

CTA-8: Despite progress on agents' benchmark development, there remains a gap in specific tool-use benchmarks tailored to SWE agents which targets the challenge of system orchestration; benchmarks that explicitly target complex issues that agents consistently fail on, and benchmarks, and techniques, that emphasize the specific components of agent decision-making that are most predictive of successful outcomes. All those are areas where future research should focus on to advance the understanding and performance of Agentic solutions.

7 Conclusion

Test or inference-time reasoning techniques have driven major recent gains in AI and have been rapidly adopted in software engineering (SWE). In this survey, we focus on code reasoning for SWE tasks, providing, to the best of our knowledge, the first systematic overview of this emerging area. We begin by proposing a taxonomy of reasoning techniques, including SWE agents (Sec. 3), which defines the scope of code reasoning and clarifies the common design patterns that underlie diverse methods.

Our analysis reveals that many state-of-the-art systems combine multiple reasoning strategies, a trend we summarize in Table 2. Despite this diversity of approaches, we observed that evaluation is concentrated on a small set of code-generation benchmarks, leaving other critical SWE tasks comparatively underexplored. To address this imbalance, we catalog a broader set of tasks and benchmarks for code reasoning (Sec. 4).

To asses the impact of existing techniques, we perform a comparative analysis over commonly used models and benchmarks (Sec. 5), highlighting relative strengths of different code reasoning methods. These observations motivate several future research directions, including the design of more realistic and comprehensive SWE benchmarks and exploration of reasoning and agent architectures that better capture the constraints and real-world engineering workflows, while taking advantage of features that formal languages afford (Sec. 7). We hope this this survey serves as a foundation for the growing field of code reasoning for SWE and supports the community in building more capable, reliable, and practical AI tools.

Limitations

This is a survey on Code Reasoning techniques, which is a new and evolving field. We covered reasoning techniques where we found a reasonable volume of work for code tasks. It is possible that we may have missed some reasoning techniques, but if so, it is likely the case that those techniques have not yet been explored by the software engineering community.

Based on our survey methodology (described in A.1) we tried our best to find all relevant code reasoning papers which are applied to code or software engineering tasks. Since it is difficult for any search method to be thorough, we acknowledge that we may have missed some papers. We are happy to take suggestions on what can be included and hope to expand the survey in the future.

Many papers use a combination of reasoning techniques. Our taxonomy and categorization is based on what we considered to be the dominant technique, which can be contested. To ensure there is no misrepresentation, we highlight papers with multiple techniques in Tab. 2.

Since many papers use sophisticated approaches, it was difficult for us to explain every detail given space constraints. For every paper we tried to highlight what we thought were the most relevant, representative and general ideas for the reader.

References

Toufique Ahmed, Jatin Ganhotra, Rangeet Pan, Avraham Shinnar, Saurabh Sinha, and Martin Hirzel. Otter: Generating tests from issues to validate swe patches, 2025. URL `https://arxiv.org/abs/2502.05368`.

Anthropic. Claude 3.7 sonnet and claude code, 2025. URL `https://www.anthropic.com/news/claude-3-7-sonnet`.

Antonis Antoniades, Albert Örwall, Kexun Zhang, Yuxi Xie, Anirudh Goyal, and William Wang. Swe-search: Enhancing software agents with monte carlo tree search and iterative refinement. *arXiv preprint arXiv:2410.20285*, 2024.

Daman Arora, Atharv Sonwane, Nalin Wadhwa, Abhav Mehrotra, Saiteja Utpala, Ramakrishna Bairi, Aditya Kanade, and Nagarajan Natarajan. Masai: Modular architecture for software-engineering ai agents. *arXiv preprint arXiv:2406.11638*, 2024.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models, 2021. URL `https://arxiv.org/abs/2108.07732`.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

Ira Ceka, Saurabh Pujar, Shyam Ramji, Luca Buratti, Gail Kaiser, and Baishakhi Ray. Understanding software engineering agents through the lens of traceability: An empirical study. *arXiv preprint arXiv:2506.08311*, 2025.

Dong Chen, Shaoxin Lin, Muhan Zeng, Daoguang Zan, Jian-Gang Wang, Anton Cheshkov, Jun Sun, Hao Yu, Guoliang Dong, Artem Aliev, et al. Coder: Issue resolving with multi-agent and task graphs. *arXiv preprint arXiv:2406.01304*, 2024a.

Junkai Chen, Zhiyuan Pan, Xing Hu, Zhenhao Li, Ge Li, and Xin Xia. Reasoning Runtime Behavior of a Program with LLM: How Far Are We? . In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pp. 140–152, Los Alamitos, CA, USA, May 2025a. IEEE Computer Society. doi: 10.1109/ICSE55347.2025.00012. URL `https://doi.ieeecomputersociety.org/10.1109/ICSE55347.2025.00012`.

Liguo Chen, Qi Guo, Hongrui Jia, Zhengran Zeng, Xin Wang, Yijiang Xu, Jian Wu, Yidong Wang, Qing Gao, Jindong Wang, et al. A survey on evaluating large language models in code generation tasks. *arXiv preprint arXiv:2408.16498*, 2024b.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021a. URL `https://arxiv.org/abs/2107.03374`.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021b.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W Cohen. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv preprint arXiv:2211.12588*, 2022.

Xiancai Chen, Zhengwei Tao, Kechi Zhang, Changzhi Zhou, Wanli Gu, Yuanpeng He, Mengdi Zhang, Xunliang Cai, Haiyan Zhao, and Zhi Jin. Revisit self-debugging with self-generated tests for code generation. *arXiv preprint arXiv:2501.12793*, 2025b.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. In *The Twelfth International Conference on Learning Representations*, 2024c. URL https://openreview.net/forum?id=KuPixIqPiq.

Cuong Le Chi, Chau Truong Vinh Hoang, Phan Nhat Huy, Dung D. Le, Tien N Nguyen, and Nghi D. Q. Bui. Visual-Coder: Guiding large language models in code execution with fine-grained multimodal chain-of-thought reasoning. In Luis Chiruzzo, Alan Ritter, and Lu Wang (eds.), *Findings of the Association for Computational Linguistics: NAACL 2025*, Albuquerque, New Mexico, April 2025. Association for Computational Linguistics.

Zheng Chu, Jingchang Chen, Qianglong Chen, Weijiang Yu, Tao He, Haotian Wang, Weihua Peng, Ming Liu, Bing Qin, and Ting Liu. Navigate through enigmatic labyrinth a survey of chain of thought reasoning: Advances, frontiers and future. *arXiv preprint arXiv:2309.15402*, 2023.

Alejandro Cuadron, Dacheng Li, Wenjie Ma, Xingyao Wang, Yichuan Wang, Siyuan Zhuang, Shu Liu, Luis Gaspar Schroeder, Tian Xia, Huanzhi Mao, et al. The danger of overthinking: Examining the reasoning-action dilemma in agentic tasks. *arXiv preprint arXiv:2502.08235*, 2025.

Darshan Deshpande, Varun Gangal, Hersh Mehta, Jitin Krishnan, Anand Kannappan, and Rebecca Qian. Trail: Trace reasoning and agentic issue localization. *arXiv preprint arXiv:2505.08638*, 2025.

Yangruibo Ding, Marcus J Min, Gail Kaiser, and Baishakhi Ray. Cycle: Learning to self-refine the code generation. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA1):392–418, 2024a.

Yangruibo Ding, Jinjun Peng, Marcus Min, Gail Kaiser, Junfeng Yang, and Baishakhi Ray. Semcoder: Training code language models with comprehensive semantics reasoning. *Advances in Neural Information Processing Systems*, 37:60275–60308, 2024b.

Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Jingyuan Ma, Rui Li, Heming Xia, Jingjing Xu, Zhiyong Wu, Tianyu Liu, et al. A survey on in-context learning. *arXiv preprint arXiv:2301.00234*, 2022.

Yihong Dong, Jiazheng Ding, Xue Jiang, Ge Li, Zhuo Li, and Zhi Jin. Codescore: Evaluating code generation by learning code execution. *ACM Trans. Softw. Eng. Methodol.*, 34(3), February 2025. ISSN 1049-331X. URL https://doi.org/10.1145/3695991.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I. Wang. Cruxeval: A benchmark for code reasoning, understanding and execution, 2024. URL https://arxiv.org/abs/2401.03065.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Hojae Han, Seung won Hwang, Rajhans Samdani, and Yuxiong He. Convcodeworld: Benchmarking conversational code generation in reproducible feedback environments, 2025. URL https://arxiv.org/abs/2502.19852.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with apps, 2021a. URL https://arxiv.org/abs/2105.09938.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with apps, 2021b. URL `https://arxiv.org/abs/2105.09938`.

Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pp. 837–847. IEEE Press, 2012. ISBN 9781467310673.

Dong Huang, Qingwen Bu, Yuhao Qing, and Heming Cui. Codecot: Tackling code syntax errors in cot reasoning for code generation. *arXiv preprint arXiv:2308.08784*, 2023.

Jie Huang and Kevin Chen-Chuan Chang. Towards reasoning in large language models: A survey. *arXiv preprint arXiv:2212.10403*, 2022.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

Nam Huynh and Beiyu Lin. Large language models for code generation: A comprehensive survey of challenges, techniques, evaluation, and applications. *arXiv preprint arXiv:2503.01245*, 2025.

Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, et al. Openai o1 system card. *arXiv preprint arXiv:2412.16720*, 2024.

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

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*, 2024a.

Nan Jiang, Xiaopeng Li, Shiqi Wang, Qiang Zhou, Soneya Binta Hossain, Baishakhi Ray, Varun Kumar, Xiaofei Ma, and Anoop Deoras. Ledex: Training llms to better self-debug and explain code, 2025. URL `https://arxiv.org/abs/2405.18649`.

Xue Jiang, Yihong Dong, Lecheng Wang, Zheng Fang, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. Self-planning code generation with large language models. *ACM Transactions on Software Engineering and Methodology*, 33(7): 1–30, 2024b.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues?, 2024a. URL `https://arxiv.org/abs/2310.06770`.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024b. URL `https://openreview.net/forum?id=VTF8yNQM66`.

Naizhu Jin, Zhong Li, Tian Zhang, and Qingkai Zeng. Mscot: Structured chain-of-thought generation for multiple programming languages. *arXiv preprint arXiv:2504.10178*, 2025.

Mohammad Abdullah Matin Khan, M Saiful Bari, Xuan Long Do, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. XCodeEval: An execution-based large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 6766–6805, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.367. URL `https://aclanthology.org/2024.acl-long.367/`.

Hung Le, Hailin Chen, Amrita Saha, Akash Gokul, Doyen Sahoo, and Shafiq Joty. Codechain: Towards modular code generation through chain of self-revisions with representative sub-modules. *arXiv preprint arXiv:2310.08992*, 2023.

Fangyu Lei, Jixuan Chen, Yuxiao Ye, Ruisheng Cao, Dongchan Shin, Hongjin Su, Zhaoqing Suo, Hongcheng Gao, Wenjing Hu, Pengcheng Yin, Victor Zhong, Caiming Xiong, Ruoxi Sun, Qian Liu, Sida Wang, and Tao Yu. Spider 2.0: Evaluating language models on real-world enterprise text-to-sql workflows, 2025. URL `https://arxiv.org/abs/2411.07763`.

Dacheng Li, Shiyi Cao, Chengkun Cao, Xiuyu Li, Shangyin Tan, Kurt Keutzer, Jiarong Xing, Joseph E Gonzalez, and Ion Stoica. S*: Test time scaling for code generation. *arXiv preprint arXiv:2502.14382*, 2025a.

Jia Li, Ge Li, Yongmin Li, and Zhi Jin. Structured chain-of-thought prompting for code generation. *ACM Transactions on Software Engineering and Methodology*, 34(2):1–23, 2025b.

Jierui Li, Hung Le, Yingbo Zhou, Caiming Xiong, Silvio Savarese, and Doyen Sahoo. Codetree: Agent-guided tree search for code generation with large language models. *arXiv preprint arXiv:2411.04329*, 2024.

Raymond Li, Loubna Ben allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia LI, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Joel Lamy-Poirier, Joao Monteiro, Nicolas Gontier, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Ben Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason T Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Urvashi Bhattacharyya, Wenhao Yu, Sasha Luccioni, Paulo Villegas, Fedor Zhdanov, Tony Lee, Nadav Timor, Jennifer Ding, Claire S Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro Von Werra, and Harm de Vries. Starcoder: may the source be with you! *Transactions on Machine Learning Research*, 2023. ISSN 2835-8856. URL `https://openreview.net/forum?id=KoFOg41haE`. Reproducibility Certification.

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

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022b. doi: 10.1126/science.abq1158.

Changshu Liu and Reyhaneh Jabbarvand. A tool for in-depth analysis of code execution reasoning of large language models. In *Companion Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, FSE 2025, New York, NY, USA, 2025. Association for Computing Machinery. URL `https://arxiv.org/abs/2501.18482`.

Changshu Liu, Shizhuo Dylan Zhang, Ali Reza Ibrahimzada, and Reyhaneh Jabbarvand. Codemind: A framework to challenge large language models for code reasoning, 2024a. URL `https://arxiv.org/abs/2402.09664`.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL `https://openreview.net/forum?id=1qvx610Cu7`.

Mingwei Liu, Juntao Li, Ying Wang, Xueying Du, Zuoyu Ou, Qiuyuan Chen, Bingxu An, Zhao Wei, Yong Xu, Fangming Zou, et al. Code copycat conundrum: Demystifying repetition in llm-based code generation. *arXiv preprint arXiv:2504.12608*, 2025.

Zhihan Liu, Shenao Zhang, Yongfei Liu, Boyi Liu, Yingxiang Yang, and Zhaoran Wang. Dstc: Direct preference learning with only self-generated tests and code to improve code lms, 2024b. URL `https://arxiv.org/abs/2411.13611`.

Jieyi Long. Large language model guided tree-of-thought. *arXiv preprint arXiv:2305.08291*, 2023.

Yingwei Ma, Rongyu Cao, Yongchang Cao, Yue Zhang, Jue Chen, Yibo Liu, Yuchen Liu, Binhua Li, Fei Huang, and Yongbin Li. Lingma swe-gpt: An open development-process-centric language model for automated software improvement. *arXiv preprint arXiv:2411.00622*, 2024.

Dung Nguyen Manh, Thang Phan Chau, Nam Le Hai, Thong T. Doan, Nam V. Nguyen, Quang Pham, and Nghi D. Q. Bui. Codemmlu: A multi-task benchmark for assessing code understanding & reasoning capabilities of codellms, 2025. URL https://arxiv.org/abs/2410.01999.

Lingrui Mei, Jiayu Yao, Yuyao Ge, Yiwei Wang, Baolong Bi, Yujun Cai, Jiazhi Liu, Mingyu Li, Zhong-Zhi Li, Duzhen Zhang, et al. A survey of context engineering for large language models. *arXiv preprint arXiv:2507.13334*, 2025.

Grégoire Mialon, Clémentine Fourrier, Thomas Wolf, Yann LeCun, and Thomas Scialom. Gaia: a benchmark for general ai assistants. In *The Twelfth International Conference on Learning Representations*, 2023.

Fangwen Mu, Lin Shi, Song Wang, Zhuohao Yu, Binquan Zhang, Chenxue Wang, Shichao Liu, and Qing Wang. Clarifygpt: Empowering llm-based code generation with intention clarification. *arXiv preprint arXiv:2310.10996*, 2023.

Niklas Muennighoff, Qian Liu, Qi Liu, Armel Randy Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and S. Longpre. Octopack: Instruction tuning code large language models. *ArXiv*, abs/2308.07124, 2023. URL https://api.semanticscholar.org/CorpusID:260886874.

Niels Mündler, Mark Niklas Müller, Jingxuan He, and Martin Vechev. Swt-bench: Testing and validating real-world bug-fixes with code agents, 2025. URL https://arxiv.org/abs/2406.12952.

Minh Huynh Nguyen, Thang Phan Chau, Phong X Nguyen, and Nghi DQ Bui. Agilecoder: Dynamic collaborative agents for software development based on agile methodology. *arXiv preprint arXiv:2406.11912*, 2024.

Ansong Ni, Srini Iyer, Dragomir Radev, Ves Stoyanov, Wen-tau Yih, Sida I. Wang, and Xi Victoria Lin. Lever: learning to verify language-to-code generation with execution. In *Proceedings of the 40th International Conference on Machine Learning*, ICML'23. JMLR.org, 2023.

Ziyi Ni, Yifan Li, and Daxiang Dong. Tree-of-code: A hybrid approach for robust complex task planning and execution. *arXiv preprint arXiv:2412.14212*, 2024.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=iaYcJKpY2B_.

OpenAI. Introducing swe-bench verified, 2024. URL https://openai.com/index/introducing-swe-bench-verified.

Albert Orwall. Moatless tools, 2024. URL https://github.com/aorwall/moatless-tools.

Wendkûuni C Ouédraogo, Kader Kaboré, Haoye Tian, Yewei Song, Anil Koyuncu, Jacques Klein, David Lo, and Tegawendé F Bissyandé. Large-scale, independent and comprehensive study of the power of llms for test case generation. *arXiv preprint arXiv:2407.00225*, 2024.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with swe-gym. *arXiv preprint arXiv:2412.21139*, 2024.

Rangeet Pan, Myeongsoo Kim, Rahul Krishna, Raju Pavuluri, and Saurabh Sinha. Aster: Natural and multi-language unit test generation with llms, 2025a. URL https://arxiv.org/abs/2409.03093.

Ruwei Pan and Hongyu Zhang. Modularization is better: Effective code generation with modular prompting. *arXiv preprint arXiv:2503.12483*, 2025.

Zhenyu Pan, Xuefeng Song, Yunkun Wang, Rongyu Cao, Binhua Li, Yongbin Li, and Han Liu. Do code llms understand design patterns? *arXiv preprint arXiv:2501.04835*, 2025b.

Qiwei Peng, Yekun Chai, and Xuhong Li. Humaneval-xl: A multilingual code generation benchmark for cross-lingual natural language generalization, 2024. URL `https://arxiv.org/abs/2402.16694`.

Huy Nhat Phan, Tien N Nguyen, Phong X Nguyen, and Nghi DQ Bui. Hyperagent: Generalist software engineering agents to solve coding tasks at scale. *arXiv preprint arXiv:2409.16299*, 2024.

Aske Plaat, Annie Wong, Suzan Verberne, Joost Broekens, Niki van Stein, and Thomas Back. Reasoning with large language models, a survey. *arXiv preprint arXiv:2407.11511*, 2024.

Archiki Prasad, Elias Stengel-Eskin, Justin Chih-Yao Chen, Zaid Khan, and Mohit Bansal. Learning to generate unit tests for automated debugging, 2025. URL `https://arxiv.org/abs/2502.01619`.

Ben Prystawski, Michael Li, and Noah Goodman. Why think step by step? reasoning emerges from the locality of experience. *Advances in Neural Information Processing Systems*, 36:70926–70947, 2023.

Ruchir Puri, David Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian T Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, Veronika Thost, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. In J. Vanschoren and S. Yeung (eds.), *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, volume 1, 2021. URL `https://datasets-benchmarks-proceedings.neurips.cc/paper_files/paper/2021/file/a5bfc9e07964f8dddeb95fc584cd965d-Paper-round2.pdf`.

Shuofei Qiao, Yixin Ou, Ningyu Zhang, Xiang Chen, Yunzhi Yao, Shumin Deng, Chuanqi Tan, Fei Huang, and Huajun Chen. Reasoning with language model prompting: A survey. *arXiv preprint arXiv:2212.09597*, 2022.

Muhammad Shihab Rashid, Christian Bock, Yuan Zhuang, Alexander Buchholz, Tim Esler, Simon Valentin, Luca Franceschi, Martin Wistuba, Prabhu Teja Sivaprasad, Woo Jung Kim, et al. Swe-polybench: A multi-language benchmark for repository level evaluation of coding agents. *arXiv preprint arXiv:2504.08703*, 2025.

Tal Ridnik, Dedy Kredo, and Itamar Friedman. Code generation with alphacodium: From prompt engineering to flow engineering. *arXiv preprint arXiv:2401.08500*, 2024.

Erik Schluntz and Barry Zhang. Building effective agents, 2024. URL `https://www.anthropic.com/engineering/building-effective-agents`.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Y Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36:8634–8652, 2023.

Qiushi Sun, Zhirui Chen, Fangzhi Xu, Kanzhi Cheng, Chang Ma, Zhangyue Yin, Jianing Wang, Chengcheng Han, Renyu Zhu, Shuai Yuan, et al. A survey of neural code intelligence: Paradigms, advances and beyond. *arXiv preprint arXiv:2403.14734*, 2024a.

Tao Sun, Linzheng Chai, Jian Yang, Yuwei Yin, Hongcheng Guo, Jiaheng Liu, Bing Wang, Liqun Yang, and Zhoujun Li. Unicoder: Scaling code large language model via universal code. *arXiv preprint arXiv:2406.16441*, 2024b.

Hao Tang, Keya Hu, Jin Zhou, Si Cheng Zhong, Wei-Long Zheng, Xujie Si, and Kevin Ellis. Code repair with llms gives an exploration-exploitation tradeoff. *Advances in Neural Information Processing Systems*, 37:117954–117996, 2024.

Zhao Tian, Junjie Chen, and Xiangyu Zhang. Fixing Large Language Models' Specification Misunderstanding for Better Code Generation . In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pp. 645–645, Los Alamitos, CA, USA, May 2025. IEEE Computer Society. URL `https://doi.ieeecomputersociety.org/10.1109/ICSE55347.2025.00108`.

Lukas Twist, Jie M Zhang, Mark Harman, Don Syme, Joost Noppen, and Detlef Nauck. Llms love python: A study of llms' bias for programming languages and libraries. *arXiv preprint arXiv:2503.17181*, 2025.

Evan Wang, Federico Cassano, Catherine Wu, Yunfeng Bai, Will Song, Vaskar Nath, Ziwen Han, Sean Hendryx, Summer Yue, and Hugh Zhang. Planning in natural language improves llm search for code generation. *arXiv preprint arXiv:2409.03733*, 2024a.

Junqiao Wang, Zeng Zhang, Yangfan He, Yuyang Song, Tianyu Shi, Yuchen Li, Hengyuan Xu, Kunyu Wu, Guangwu Qian, Qiuwu Chen, et al. Enhancing code llms with reinforcement learning in code generation. *arXiv preprint arXiv:2412.20367*, 2024b.

Wenhan Wang, Chenyuan Yang, Zhijie Wang, Yuheng Huang, Zhaoyang Chu, Da Song, Lingming Zhang, An Ran Chen, and Lei Ma. TESTEVAL: Benchmarking large language models for test case generation. In *Findings of the Association for Computational Linguistics: NAACL 2025*. Association for Computational Linguistics, April 2025. ISBN 979-8-89176-195-7. URL `https://aclanthology.org/2025.findings-naacl.197/`.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better llm agents. In *Forty-first International Conference on Machine Learning*, 2024c.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better llm agents, 2024d. URL `https://arxiv.org/abs/2402.01030`.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations*, 2024e.

Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, et al. Emergent abilities of large language models. *arXiv preprint arXiv:2206.07682*, 2022a.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022b.

Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida I Wang. Swe-rl: Advancing llm reasoning via reinforcement learning on open software evolution. *arXiv preprint arXiv:2502.18449*, 2025.

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

Chengxing Xie, Bowen Li, Chang Gao, He Du, Wai Lam, Difan Zou, and Kai Chen. Swe-fixer: Training open-source llms for effective and efficient github issue resolution. *arXiv preprint arXiv:2501.05040*, 2025.

Fengli Xu, Qianyue Hao, Zefang Zong, Jingwei Wang, Yunke Zhang, Jingyi Wang, Xiaochong Lan, Jiahui Gong, Tianjian Ouyang, Fanjin Meng, et al. Towards large reasoning models: A survey of reinforced reasoning with large language models. *arXiv preprint arXiv:2501.09686*, 2025.

Dayu Yang, Tianyang Liu, Daoan Zhang, Antoine Simoulin, Xiaoyi Liu, Yuwei Cao, Zhaopu Teng, Xin Qian, Grey Yang, Jiebo Luo, et al. Code to think, think to code: A survey on code-enhanced reasoning and reasoning-driven code intelligence in llms. *arXiv preprint arXiv:2502.19411*, 2025.

Guang Yang, Yu Zhou, Xiang Chen, Xiangyu Zhang, Terry Yue Zhuo, and Taolue Chen. Chain-of-thought in neural code generation: From and for lightweight language models. *IEEE Transactions on Software Engineering*, 2024a.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering, 2024b. URL `https://arxiv.org/abs/2405.15793`.

John Yang, Carlos E. Jimenez, Alex L. Zhang, Kilian Lieret, Joyce Yang, Xindi Wu, Ori Press, Niklas Muennighoff, Gabriel Synnaeve, Karthik R. Narasimhan, Diyi Yang, Sida I. Wang, and Ofir Press. Swe-bench multimodal: Do ai systems generalize to visual software domains?, 2024c. URL https://arxiv.org/abs/2410.03859.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems*, 36:11809–11822, 2023a.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023b.

Asaf Yehudai, Lilach Eden, Alan Li, Guy Uziel, Yilun Zhao, Roy Bar-Haim, Arman Cohan, and Michal Shmueli-Scheuer. Survey on evaluation of llm-based agents. *arXiv preprint arXiv:2503.16416*, 2025.

Sangyeop Yeo, Seung-won Hwang, and Yu-Seung Ma. Chain of grounded objectives: Bridging process and goal-oriented prompting for code generation. *arXiv preprint arXiv:2501.13978*, 2025.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In Ellen Riloff, David Chiang, Julia Hockenmaier, and Jun'ichi Tsujii (eds.), *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, October-November 2018. doi: 10.18653/v1/D18-1425. URL https://aclanthology.org/D18-1425/.

Zhuohao Yu, Weizheng Gu, Yidong Wang, Zhengran Zeng, Jindong Wang, Wei Ye, and Shikun Zhang. Outcome-refining process supervision for code generation. *arXiv preprint arXiv:2412.15118*, 2024.

Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Yongji Wang, and Jian-Guang Lou. Large language models meet nl2code: A survey. *arXiv preprint arXiv:2212.09420*, 2022.

Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Linhao Zhang, Shulin Xin, Lu Chen, Qi Liu, Xiaojian Zhong, Aoyan Li, Siyao Liu, Yongsheng Xiao, Liangqiang Chen, Yuyu Zhang, Jing Su, Tianyu Liu, Rui Long, Kai Shen, and Liang Xiang. Multi-swe-bench: A multilingual benchmark for issue resolving, 2025. URL https://arxiv.org/abs/2504.02605.

Huaye Zeng, Dongfu Jiang, Haozhe Wang, Ping Nie, Xiaotong Chen, and Wenhu Chen. Acecoder: Acing coder rl via automated test-case synthesis, 2025. URL https://arxiv.org/abs/2502.01718.

Huan Zhang, Wei Cheng, Yuhan Wu, and Wei Hu. A pair programming framework for code generation via multi-plan exploration and feedback-driven refinement. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1319–1331, 2024a.

Xiaoyu Zhang, Juan Zhai, Shiqing Ma, Qingshuang Bao, Weipeng Jiang, Chao Shen, and Yang Liu. Unveiling provider bias in large language models for code generation. *arXiv preprint arXiv:2501.07849*, 2025.

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

Wenqing Zheng, S P Sharan, Ajay Kumar Jaiswal, Kevin Wang, Yihan Xi, Dejia Xu, and Zhangyang Wang. Outline, then details: Syntactically guided coarse-to-fine code generation, 2023. URL https://arxiv.org/abs/2305.00909.

Terry Yue Zhuo. Ice-score: Instructing large language models to evaluate code. In *Findings of the Association for Computational Linguistics: EACL 2024*, pp. 2232–2242, 2024.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen Gong, Thong Hoang, Armel Randy Zebaze, Xiaoheng Hong, Wen-Ding Li, Jean Kaddour, Ming Xu, Zhihan Zhang, Prateek Yadav, Naman Jain, Alex Gu, Zhoujun Cheng, Jiawei Liu, Qian Liu, Zijian Wang, Binyuan Hui, Niklas Muennighoff, David Lo, Daniel Fried, Xiaoning Du, Harm de Vries, and Leandro Von Werra. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions, 2025. URL `https://arxiv.org/abs/2406.15877`.

A Appendix

A.1 Survey Methodology

We used arXiv and Google Scholar to ensure comprehensive coverage of all relevant works. In particular, we utilized their advanced search functionality, querying combinations of terms such as *"code reasoning"*, *"reasoning"* + *"LLM"*, and *"agents"* + *"software engineering"*.

To rigorously account for recent work building on existing foundations, we also examined citation graphs in Google Scholar—manually inspecting entries that cited foundational papers.

We focused our review on publications in premier venues including ACL, EMNLP, ICLR, NeurIPS, and others, while also incorporating cutting-edge preprints that may not yet have received broad recognition. This emphasis on both established and emerging work allowed us to capture the state of the art as well as frontier directions in the field.

A.2 Benchmarks

HumanEval (HE) Chen et al. (2021a) is a set of 164 hand-written programming problems. Each problem includes a function signature, docstring, body, and several unit tests, with an average of 7.7 tests per problem. A multi-language version of HE is also available in HumanEval-XL Peng et al. (2024).

MBPP Austin et al. (2021) (The Most Basic Programming Problems) benchmark has 1k crowd-sourced Python programming problems and was designed to be solvable by entry level programmers. Each problem consists of a task description, code solution and three automated test cases. *EvalPlus* Liu et al. (2023) augments a given evaluation dataset with large amounts of new test cases created by an automatic test input generator, powered by both LLM- and mutation-based strategies. EvalPlus includes *MBPP+, HumanEval+*, and *EvalPerf*.

APPS Hendrycks et al. (2021b) is another benchmark for code generation with 10k samples that measures the ability of models to take an arbitrary natural language specification and generate satisfactory Python code. More recent extensions of some of the above benchmarks such as *HumanEval-ET, MBPP-ET*, and *APPS-ET* were introduced by Dong et al. (2025), where the amount of correct test cases were extended for each benchmark 100+ on average according to the reference code.

*CodeContests*Li et al. (2022b) is a code generation dataset with problems curated from competitive programming platforms such as Codeforces, requiring solutions to challenging code generation problems. This dataset has solutions to the given problems in Python, Java, and C++, with an English description of the code problems.

A.3 Code Evaluation

To address the poor correlation with human evaluation of exact or fuzzy match metrics, ICE-Score was recently proposed as an evaluation metric that instructs LLMs for code assessments Zhuo (2024). The ICE-Score evaluation showed superior correlations with functional correctness and human preferences, without the need for test oracles or references. The efficacy of ICE-Score was measured w.r.t. human preference and execution success for four programming languages.

Additionally, CodeScore Dong et al. (2025) is another code evaluation metric that was recently proposed to measure the functional correctness of generated codes on three input formats (Ref-only, NL-only, and Ref&NL). CodeScore can be obtained through the UniCE framework that assists models in learning code execution and predicting an estimate of execution PassRatio.

A.4 Metrics

Functional correctness of generated code by LLMs is mainly measured by passing tests. One of the basic metrics to measure the correctness of code is the percentage of tasks in a given benchmark where the generated code successfully

Approach	Model	APPS Introductory	APPS Interview	APPS Competition	APPS-ET	APPS
CodeChain Le et al. (2023)	gpt-4	71.1	55.0	23.3	–	61.5
	gpt-3.5-turbo-16k	54.5	28.1	12.4	–	26.4
	WizardCoder	26.3	7.5	3.8	–	10.5
ChainCoder ♢ Zheng et al. (2023)	ChainCoder-1B	17.5	7.4	5.5	–	–
AlphaCode ♢ Li et al. (2022a)	AlphaCode-1B	14.4	5.6	4.6	–	–
Self-Planning Jiang et al. (2024b)	gpt-3.5-turbo	–	–	–	8.3	21.3
	DeepSeekCoder	–	–	–	1.0	4.0
SCoT (Li et al., 2025b)	gpt-3.5-turbo	–	–	–	7.7	22.0
	DeepSeek-Coder-6.7B-Instr	–	–	–	1.3	4.3
Self-Debugging (Chen et al., 2024c)	gpt-3.5-turbo	–	–	–	6.2	18.7
	DeepSeek-Coder-6.7B-Instr	–	–	–	1.3	4.7
CYCLE (Ding et al., 2024a)	CYCLE-350M	–	–	–	–	8.7
	CYCLE-1B	–	–	–	–	10.9
	CYCLE-2.7B	–	–	–	–	11.6
	CYCLE-3B	–	–	–	–	11.3
μ-Fix (Tian et al., 2025)	gpt-3.5-turbo	–	–	–	10.3	35.7
	DeepSeek-Coder-6.7B-Instr	–	–	–	5.0	14.0
REx Tang et al. (2024)	gpt-4	–	–	–	–	~ 70

Table 3: Performance across the APPS benchmark Hendrycks et al. (2021a), including the **APPS Introductory**, **Interview**, **Competition**, **APPS-ET**, and **APPS** overall sets. Default performance is reported as $pass@1$ (%). Approaches marked with ♢ use the $n@k$ metric, where $n = 5$ and $k = 1{,}000$.

passes all tests. Chen et al. (2021a) shows that exact or fuzzy match metrics (e.g., BLEU) are not adequate or reliable indicators of functional correctness of code, by showing that functionally different programs generated by a model often have higher BLEU scores than functionally equivalent ones.

The metric $pass@k$ is the probability of generating at least one solution passing all test cases successfully in k trials. The *AvgPassRatio* measures the degree of correctness of generated code on evaluation test cases, it considers whether the generated code is completely correct on evaluation test cases or not. Another metric is the percentage of problems solved using n submissions from k samples per problem, denoted as $n@k$.

Approach	Model	LCB	CodeContests	M3ToolEval
S* (Li et al., 2025a)	Qwen-2.5-Coder-Instruct 32B	70.1	21.8	-
	gpt-4o-mini	61.3	23.0	-
	R1-Distill-32B	85.7	-	-
	o1-mini	85.3	48.5	-
PlanSearch Wang et al. (2024a)	DeepSeek-Coder-V2	41.4	–	-
	gpt-4o-mini	39.0	–	-
	gpt-4o	41.3	–	-
	Claude-Sonnet-3.5	40.3	-	-
	o1-mini	69.5	–	-
CodeChain † Le et al. (2023)	gpt-3.5	-	14.1	-
ChainCoder ‡ Zheng et al. (2023)	ChainCoder-1B	-	~ 15	-
AlphaCode ‡ Li et al. (2022a)	AlphaCode-9B	-	14.3	-
	AlphaCode-41B	-	15.6	-
PairCoder Zhang et al. (2024a)	gpt-3.5-turbo	-	15.2	-
	DeepSeek-Coder	-	14.6	-
CodeTree Zhang et al. (2024a)	gpt-4o-mini	-	26.4	-
	gpt-4o	-	43.0	-
	Llama-3.1-8B	-	12.1	-
AlphaCodium † Ridnik et al. (2024)	DeepSeek-33B	-	24.0	-
	gpt-3.5	-	17.0	-
	gpt-4	-	29.0	-
CodeAct Wang et al. (2024c)	gpt-4	–	–	74.4
Tree-of-Code Ni et al. (2024)	Mix-modal	–	–	81.6

Table 4: Performance across the **LiveCodeBench (LCB)**, **CodeContests (test set)**, and **M3ToolEval**. Default results are reported as $pass@1$. Approaches marked with † indicate $pass@5$, while those marked with ‡ use the $n@k$ of $10@1k$ rate. S* results reflect performance on LCB v2.

Approach	Model	SWE-Bench Verified	SWE-Bench Lite	SWE-Bench
Agentless Xia et al. (2024)	gpt-4o	33.2	24.3	-
	o1-preview	41.3	-	-
	DeepSeek-V3	42.0	-	-
	DeepSeek-R1	49.2	-	-
	Claude-3.5-Sonnet	53.0	-	-
AutoCodeRover Zhang et al. (2024b)	Qwen2-72B-Instruct	-	9.3	-
	gpt-4o	28.8	22.7	-
	gpt-4	-	19.0	-
MASAI Arora et al. (2024)	gpt-4o	–	28.3	-
SWE-Agent Yang et al. (2024b)	Claude-3.5-Sonnet	33.6	23.0	-
	gpt-4o	23.2	18.3	-
SWE-Gym Pan et al. (2024)	Qwen-2.5-Coder-Instruct 32B	20.6	15.3	-
	SWE-Gym-32B	32.0	26.0	-
SWE-Search Antoniades et al. (2024)	gpt-4o	-	31.0	-
	gpt-4o-mini	-	17.0	-
	Qwen-2.5-72b-Instruct	-	24.7	-
	Deepseek-V2.5	-	21.0	-
	Llama-3.1-70b-Instruct	-	17.7	-
Lingma Ma et al. (2024)	Lingma SWE-GPT 72B	30.2	22.0	-
	Lingma SWE-GPT 7B	18.2	12.0	-
SWE-Fixer Xie et al. (2025)	SWE-Fixer-72B	32.8	24.7	-
HyperAgent Phan et al. (2024)	-	33.0	26.0	-
SWE-RL Wei et al. (2025)	Llama3-SWE-RL-70B	41.0	-	-
CodeR Chen et al. (2024a)	gpt-4	–	28.3	-
CodeTree Li et al. (2024)	gpt-4o-mini	–	–	27.6
OpenHands Wang et al. (2024e)	gpt-4o-mini	–	7.0	-
	gpt-4o	–	22.0	-
	Claude-3.5-Sonnet	–	26.0	-

Table 5: Performance on **SWE-Bench Verified**, and **SWE-Bench Lite**, and **SWE-Bench**. Performance is measured by resolved rate.

Approach	Model	MBPP+	MBPP	MBPP-ET	MBPP-S
PlanSearch Wang et al. (2024a)	gpt-4o-mini	73.5	–	–	–
	gpt-4o	77.2	–	–	–
	DeepSeekCoder-V2	76.3	–	–	–
	Claude-3.5-sonnet	77.1	–	–	–
ClarifyGPT Mu et al. (2023)	gpt-3.5-turbo	–	–	55.6	74.1
	gpt-4	–	–	58.5	78.7
Self-Planning Jiang et al. (2024b)	Codex	–	–	41.9	55.7
	gpt-4o-mini	42.4	52.1	48.2	–
	DeepSeek-R1	55.4	68.4	65.5	–
	gpt-3.5-turbo	68.1	–	–	82.6
	Llama-3 8B Instr.	56.9	–	–	67.9
SCoT (Li et al., 2025b)	gpt-3.5-turbo	–	47.0	–	–
	Codex	–	38.3	–	–
	gpt-4o-mini	51.4	63.9	55.6	–
	DeepSeek-R1	46.9	57.9	61.3	–
MoT Pan & Zhang (2025)	DeepSeek-R1	60.4	74.9	68.0	–
	gpt-4o-mini	58.1	73.9	58.9	–
CGO Yeo et al. (2025)	gpt-3.5-turbo	73.7	–	–	86.0
	Llama-3 8B Instr.	57.9	–	–	68.1
UniCoder Sun et al. (2024b)	Deepseek-Coder	–	64.3	–	–
	CodeLlama-7B	–	65.2	–	–
Self-Debugging Chen et al. (2024c)	Codex	–	70.8	–	–
	gpt-3.5-turbo	–	74.2	60.4	–
	gpt-4	–	80.6	–	–
	StarCoder	–	53.2	–	–
	DeepSeek-Coder-6.7B-Instruct	–	–	56.9	–
LeDex Jiang et al. (2025)	StarCoder-15B	54.3	58.2	–	–
	CodeLlama-7B	52.9	58.1	–	–
	CodeLlama-13B	57.9	61.9	–	–
Revisiting Self-Debugging Chen et al. (2025b)	gpt-4o	76.5	91.5	–	–
	Claude-3.5-sonnet	77.0	92.6	–	–
	Llama-3-70B-Instr.	71.2	84.4	–	–
	Qwen-2.5-Coder-7B-Instr	70.6	84.7	–	–
ORPS Yu et al. (2024)	Llama-3.1-8B-Instruct	-	90.4	–	–
	DeepSeek-Coder-7B-Instruct-v1.5	-	93.0	–	–
	Qwen-2.5-Coder-7B-Instruct	-	94.9	–	–
	Qwen-2.5-Coder-14B-Instruct	-	95.3	–	–
	gpt-4o-mini	-	95.7	–	–
CodeTree Li et al. (2024)	gpt-4o-mini	77.0	96.8	–	–
	gpt-4o	80.7	98.7	–	–
	Llama-3.1-8B-Instr.	73.3	90.5	–	–
AgileCoder (Nguyen et al., 2024)	gpt-3.5-turbo	–	80.9	–	–
	claude-3-haiku	–	84.3	–	–
PairCoder (Zhang et al., 2024a)	gpt-3.5-turbo	77.7	80.6	–	–
	DeepSeek-Coder	75.7	78.8	–	–
	gpt-4	–	–	–	91.2
CYCLE (Ding et al., 2024a)	CYCLE-350M	–	–	–	32.6
	CYCLE-1B	–	–	–	35.8
	CYCLE-2.7B	–	–	–	48.5
	CYCLE-3B	–	–	–	51.3
μ-Fix (Tian et al., 2025)	gpt-3.5-turbo	–	–	69.1	–
	DeepSeek-Coder-6.7B-Instruct	–	–	63.3	–
SemCoder (Ding et al., 2024b)	SemCoder-S-6.7B	68.5	79.6	–	–
	SemCoder-6.7B	65.3	79.9	–	–

Table 6: Performance on the **MBPP+**, **MBPP**, **MBPP-ET**, and **MBPP-sanitized** benchmarks. All results are re-

Approach	Model	HE+	HE	HE-XL	HE-X	HE-ET
PlanSearch Wang et al. (2024a)	gpt-4o-mini	83.7	–	–	–	–
	gpt-4o	86.4	–	–	–	–
	DeepSeekCoder-V2	82.8	–	–	–	–
	Claude-3.5-sonnet	81.6	–	–	–	–
ClarifyGPT Mu et al. (2023)	gpt-3.5-turbo	–	74.4	–	–	64.8
	gpt-4	–	87.8	–	–	78.1
Self-Planning Jiang et al. (2024b)	Codex	–	60.3	–	60.3	46.2
	gpt-4o-mini	79.9	87.2	–	–	87.1
	DeepSeek-R1	79.3	85.4	–	–	85.3
	gpt-3.5-turbo	67.3	72.7	–	–	–
	LLaMA-3 8B Instr.	52.8	60.1	–	–	–
SCoT (Li et al., 2025b)	gpt-3.5-turbo	–	60.6	–	–	–
	Codex	–	49.8	–	–	–
	gpt-4o-mini	78.7	86.6	–	–	86.0
	DeepSeek-R1	79.3	84.8	–	–	–
	DeepSeekCoder	–	–	69.3	–	–
	Qwen-2.5-Coder	–	–	74.4	–	–
MoT Pan & Zhang (2025)	DeepSeek-R1	88.4	95.1	–	–	94.5
	gpt-4o-mini	83.5	92.1	–	–	91.5
MSCoT Pan & Zhang (2025)	DeepSeek-Coder	–	–	66.0	–	–
	Qwen2.5-Coder	–	–	72.3	–	–
CGO Yeo et al. (2025)	gpt-3.5-turbo	68.5	74.6	–	–	–
	LLaMA-3 8B Instr.	56.2	62.4	–	–	–
UniCoder Sun et al. (2024b)	DeepSeek-Coder	–	70.6	–	–	–
	CodeLlama-7B	–	65.4	–	–	–
COTTON Yang et al. (2024a)	gpt-3.5-turbo	76.2	74.4	–	–	–
	DeepSeekCoder	–	–	61.8	–	–
	Qwen-2.5-Coder	–	–	68.7	–	–
Agile Coder Nguyen et al. (2024)	gpt-3.5-turbo	–	70.5	–	–	–
	claude-3-haiku	–	79.3	–	–	–
	gpt-4	–	90.9	–	–	–
CodeAct Wang et al. (2024c)	CodeActAgent(LLaMA-2-7B)	–	18.1	–	–	–
	CodeActAgent(Mistral-7B)	–	34.7	–	–	–
PairCoder Zhang et al. (2024a)	gpt-3.5-turbo	77.4	87.8	–	–	–
	DeepSeek-Coder	76.2	85.4	–	–	–
	gpt-4	–	93.9	–	–	–
CodeTree Li et al. (2024)	gpt-4o-mini	84.8	94.5	–	–	–
	gpt-4o	86.0	94.5	–	–	–
	Llama-3.1-8B	72.0	82.3	–	–	–
μ-Fix Tian et al. (2025)	gpt-3.5-turbo	80.5	90.2	–	–	79.9
	DeepSeek-Coder-6.7B-Instr	78.7	83.5	–	–	75.0
Self-Debugging Chen et al. (2024c)	gpt-3.5-turbo	71.3	77.4	–	–	–
	DeepSeek-Coder-6.7B-Instr	73.2	77.4	–	–	–
LeDex Jiang et al. (2025)	StarCoder-15B	46.3	52.3	–	–	–
	CodeLlama-7B	50.0	55.8	–	–	–
	CodeLlama-13B	56.7	61.7	–	–	–
CYCLE Ding et al. (2024a)	CYCLE-350M	–	20.7	–	–	–
	CYCLE-1B	–	22.0	–	–	–
	CYCLE-2.7B	–	29.3	–	–	–
	CYCLE-3B	–	29.9	–	–	–
Revisiting Self-Debugging Chen et al. (2025b)	gpt-4o	87.8	92.1	–	–	–
	Claude-3.5-Sonnet	89.0	94.5	–	–	–
	Llama-3-70B Instr.	73.8	79.9	–	–	–
	Qwen-2.5-Coder	81.7	86.0	–	–	–
SemCoder Ding et al. (2024b)	SemCoder-S-6.7B	74.4	79.3	–	–	–