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

Anonymous ACL submission

Abstract

The rise of large language models (LLMs) has led to dramatic improvements across a wide range of natural language tasks. These advancements have extended into the domain of code, facilitating complex tasks such as code generation, translation, summarization, and repair. However, their utility for real-world deployment in-the-wild has only recently been studied, on Software Engineering (SWE) tasks such as GitHub issue resolution. In this study, we examine the code reasoning techniques that underlie the ability to perform such tasks, and examine the paradigms used to drive their performance. Our contributions in this paper are: (1) the first dedicated survey on code reasoning for code tasks, highlighting overarching strategies, hybrid and agentic approaches; (2) a taxonomy of various techniques used to drive code reasoning; (3) a comprehensive overview of performance on common benchmarks and showcase new, under-explored benchmarks with high potential in SWE; (4) an exploration on how core properties of code can be used to explain different reasoning techniques; and (5) gaps and under-explored areas 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), language translation (Roziere et al., 2020) code summarization (Sun et al., 2025) 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., 2021b; 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 generation, test generation and GitHub issue resolution as well.

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. Many underlying techniques contribute to reasoning models, including Chain-of-Thought or CoT (Wei et al., 2022b) which elicits reasoning, learning from environment feedback (Chen et al., 2024c) and exploring multiple reasoning paths (Yao et al., 2023a). Many recent surveys explore reasoning techniques, SWE LLMs, benchmarks and Agents, and we discuss them in Sec. B. However we did not find any survey that explores the impact of reasoning, and specifically code-based reasoning techniques on SWE tasks. 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. 2). SWE Agents are given a special focus (Sec. 3) given they depend on multiple reasoning techniques.

(2) A Taxonomy covering different reasoning approaches and benchmarks for code Fig. 1. We also highlight approaches employing multiple reasoning techniques for LLMs in general (Tab. 1) and agents in particular (Tab. 2).

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

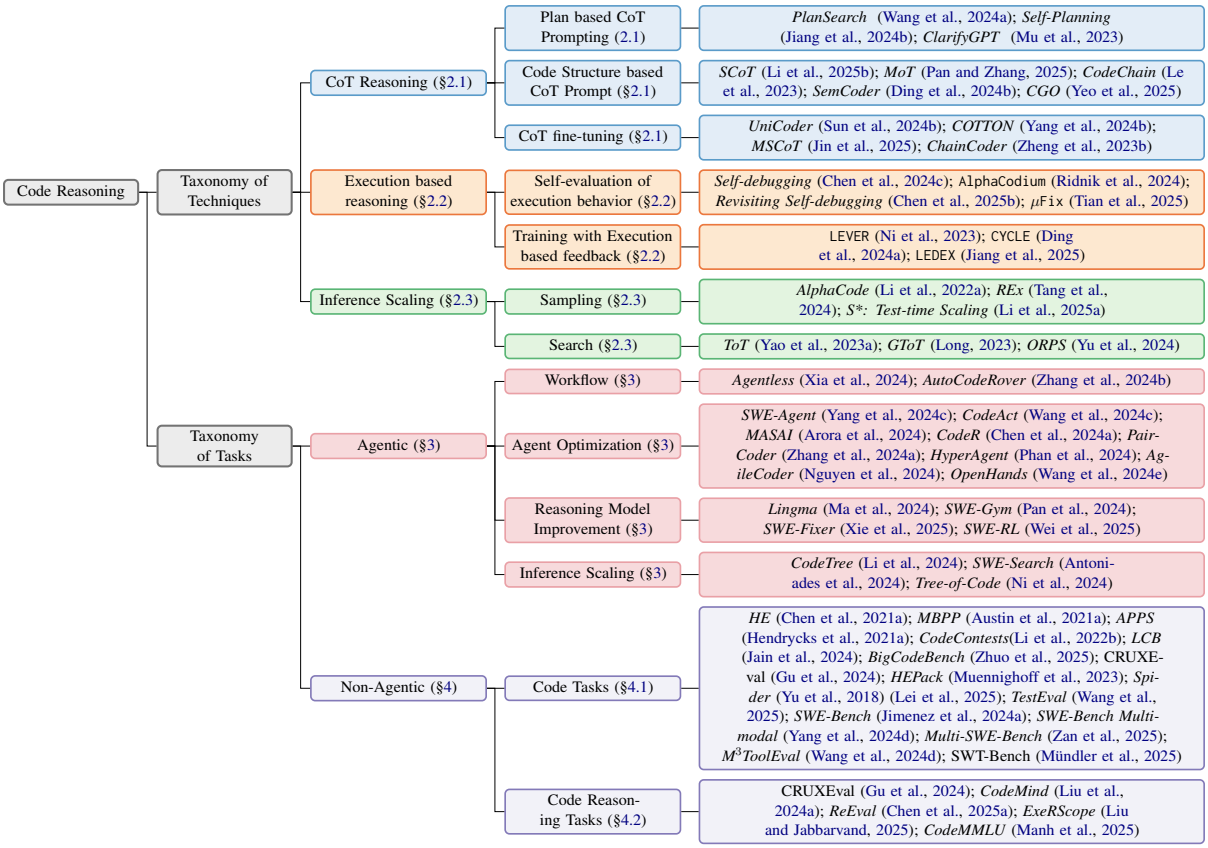

Figure 1: Code Reasoning Taxonomy.

(3) Showcase benchmarks used to study the impact of reasoning on SWE tasks. We compiled comparison tables (Tab. 4, 6, 7, 8) showing the performance of different code reasoning and agentic approaches (Sec. 4.1). We also highlight promising benchmarks specific to code reasoning (Sec. 4.2), and surface some new agent-specific benchmarks with potential for furthering SWE research.

(4) Discussion on how the performance of different code reasoning techniques might be connected to different code properties (Sec. 3). In Sec. 6, we use this discussion to motivate future work.

2 Taxonomy of 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 and 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. 2.1.

Yao et al., 2023a state that "System 2" thinking should involve exploring diverse solution paths rather than greedily picking one. They connect CoT with sampling and search to enable exploration of multiple reasoning paths. Li et al., 2022a effectively leverage sampling and search techniques to generate competition level code. Sec. 2.3 covers sampling and search techniques used to explore multiple reasoning paths for software engineering tasks.

One way code output is different from natural language output is that it can be executed and tested to validate it's correctness. Yao et al., 2023a highlight that execution can be a way to check if the reasoning is correct. Li et al., 2022a use code execution and sample tests as a way to filter generated output. Chen et al., 2024c teach the model to self-debug based on reasoning from execution results. Other such techniques based on code execution are covered in Sec. 2.2.

Agents bring most of these reasoning techniques together. ReAct (Yao et al., 2023b) enables problem-solving through real-time environ-

Approach	CoT	Exe Based	Inf. Scaling	Other
PlanSearch	P	B	S	
Self-Planning, ClarifyGPT	P			
SCoT, CGO, MoT, CodeChain	S			
UniCoder, ChainCoder, MSCoT	S-FT			
COTTON	P-FT			
SemCoder	S-FT	G		
MSCoT	S-FT			
Self-Debug		B		MV
CodeCOT, AlphaCodium	P	G-B		
Revisit Self-Debug		G-B		MV
muFix	P	G		
LEVER		B	S	RR
CYCLE		B	S	
LEDEX	P	B	S	RL
ORPS		G-B		
GToT			LM	
S*		G-B	S	

Table 1: LLM reasoning approaches for code tasks and key components. CoT (Chain-of-Thought, including Plan (P), Structure (S), or Finetuning (FT)); Exe-based (Execution-based feedback using model-generated (G) or benchmark (B) tests); Other includes MV (Majority Vote), RR (Re-Ranking), and RL (Reinforcement Learning). Approaches are categorized by dominant strategy: CoT and Planning, Execution-driven, and sampling or search

Approach	Work Flow	Reasoning Model	Agent Optim.	Inf. Scaling
Agentless	✓			
AutoCodeRover	✓		T	
SWE-Agent			T	
CodeAct		SFT	T	
OpenHands			MA-T	
MASAI			MA-T	
CodeR			MA-T	
AgileCoder			MA-T	
PairCoder	✓		MA	
HyperAgent			MA	
Lingma	✓	SFT		
SWE-Fixer	✓	SFT		
SWE-Gym		SFT-V	T	✓
SWE-RL		RL	MA	
CodeTree			MA	✓
ToC				✓
SWE-Search			MA-T	✓

Table 2: In our taxonomy Agents are classified as employing one of the following techniques (1) Workflow (2) Reasoning Model improvement (3) Agent optimization (4) Inference scaling. However many agents employ multiple techniques. For ex., SWE-Gym is classified in Reasoning model improvement category, but they also train a verifier model for inference scaling. This table highlights such nuances.

mental engagement. Reflexion (Shinn et al., 2023) leverages linguistic reflection to enhance performance. Sec. 3 surveys different software engineering agents which build on reasoning techniques mentioned above.

2.1 Code Chain-of-Thought Reasoning

Chain-of-Thought or CoT (Wei et al., 2022b) is a prompting technique for large language models (LLMs) designed to elicit step-by-step reasoning, making it likelier that the LLMs arrive at the correct answer and in the process making their "thoughts" more transparent. 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 utilize some code structure or programming concepts. Besides prompting only techniques, another approach used by many is fine-tuning or instruction tuning for software engineering tasks with code CoT data.

Plan-based CoT Prompting. 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 based CoT Prompting. 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 and Zhang, 2025 propose a novel prompting technique, `Modularization-of-thought (MoT)`, which exploits modularization principals 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.

CoT fine-tuning. 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., 2024b 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., 2023b) 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.

2.2 Execution-based Reasoning

Execution-based reasoning involves executing LLM-generated code in a given environment and having the LLM reason and learn from the execution environment output.

Self-Evaluation of Execution Behavior. 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. The model is instructed to execute the code and then generate a feedback message based on the code and its execution result from running Unit Tests (UT). 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 to improve code LLM performance that does not require training a model. The two key phases in AlphaCodium's flow are: (a) a pre-processing phase, where it generates problem reflection and test reasoning; and (b) an iterative code generation phase, where code is generated, run, and fixed against both public and AI-generated tests. In revisited self-debugging (Chen et al., 2025b) authors explored both post-execution and in-execution self-debugging, leveraging self-generated tests. In post-execution self-debugging, the process directly validates the correctness of code by checking whether the output after execution matches the test output or not, whereas in-execution self-debugging analyzes the intermediate runtime states during program execution without knowing the results from post-execution. 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. They show that CoT, SCoT, and Self-repair can fail due to specification misunderstandings, which test case analysis helps mitigate to improve both feedback-based prompting and code generation.

Training with Execution-based 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. The generated code is re-ranked based on the verification score and the LLM generation probability, and marginalizing over programs with the same 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. Both Supervised Full Tuning (SFT) and Reinforcement Learning (RL) train the code model using success and failure trajectories.

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. We present different strategies to generate UTs with LLMs. For instance, 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. More recently, ASTER (Pan et al., 2025a) is a multilingual UT-generator built with LLMs guided by lightweight program analysis.

2.3 Search and Sampling for SE Tasks

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. The heuristic reward is the fraction of specifications (test cases) satisfied by the program. 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).

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. The LM's reasoning acts as the heuristic, contrasting traditional approaches that use learned or programmed rules. `ToT` traverses the tree using BFS or DFS. 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." Each tree state includes theoretical reasoning, code implementation, and execution outcomes. ORPS updates states with LM-generated reasoning, executes code for feedback, applies critique and rewards, and retains top solutions via self-refinement.

3 Taxonomy of Tasks: Agentic

Agentic systems use many of the reasoning techniques described in Sec. 2 for different tasks. Software Engineering (SE) 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).

Workflow. Schluntz and 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., 2024c) 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. This is claimed to be better than the existing technique of producing actions by generating JSON or text in a predefined format, which is less flexible and has a constrained 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) described in Appendix. C.

Reasoning Model Improvement. 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 observe that software evolution involves not just code but developers' reasoning, tools, and cross-role interactions. Their `Lingma` SWE-GPT models (7B, 72B) are fine-tuned on repository understanding, bug localization, patching, and rejection sampling using pull-requests from repos. Training starts from Qwen2.5-Coder-7B (Hui et al., 2024) and Qwen2.5-72B-Instruct (Yang et al., 2024a), and inference runs through `SWESynInfer`, an AutoCodeRover-based workflow (Zhang et al., 2024b). 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) is an open-source, two-stage GitHub issue fixer. A fine-tuned Qwen2.5-7B retriever, boosted with BM25, identifies relevant files, while a fine-tuned Qwen2.5-72B editor generates patches. Each model was trained on 110k issue–patch pairs, with the editor further tuned on CoT data synthesized by GPT-4o (Hurst et al., 2024). `SWE-RL` (Wei et al., 2025) is the first scalable RL-based reasoning approach for SE. 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).

Inference Scaling. Agentic systems often involve a component that scales inference time compute and improves agent performance by searching over multiple samples. 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. `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. An LLM-backed hybrid value function combines numeric and qualitative scores from trajectories, file context, and test output to steer node expansion.

4 Taxonomy of Tasks: Non-Agentic

In this section we discuss the different non-agentic reasoning and non-reasoning tasks which are used to evaluate reasoning techniques described in Sec. 2.

4.1 Code Tasks

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., 2021a), *APPS* (Hendrycks et al., 2021a), 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.

For test generation, benchmarks like *TestEval* (Wang et al., 2025) can help w.r.t. three different aspects: overall coverage, targeted line/branch coverage, and targeted path coverage. 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., 2024d) for visual and user-facing components, and *Multi-SWE-Bench* (Zan et al., 2025) for more programming languages besides Python. *M*^{3}*ToolEval* (Wang et al., 2024d) is used for multi-turn, multi-tool complex tasks. *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]

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

4.2 Code Reasoning Tasks

ReEval (Chen et al., 2025a) helps to analyze how Code LLMs reason about runtime behaviors (e.g., program state, execution paths) of programs. The *ExeRScope* (Liu and Jabbarvand, 2025) tool helps to analyze the result of code execution reasoning frameworks and understand the impact of code properties. *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 (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

How can variance in performance of different techniques (planning, structure-aware, execution-based, inference scaling, etc.) on common benchmarks be explained by properties of code? 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.** This may shed light on the variance in performance across different CoT patterns. Section 2.1 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.

Observation G.1: Structure-aware CoT strategies are better than planning-based CoT strategies, especially for self-contained code-contest benchmarks like MBPP and HE benchmarks.

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, which explains the next finding.

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

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. Additionally, text lacks the precision required for computational tasks, whereas structure and modularity are more precise and unambiguous. Still, there are other fascinating properties of code that can be leveraged: code exhibits *deterministic output, executable nature, and error feedback*. These properties can be leveraged to validate or verify a solution, which explains the next observation:

Observation G.3: Execution-aware strategies dominate CoT-based strategies.

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), this 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.

Observation G.4: Approaches that integrate inference scaling outperform execution-dominant or CoT-dominant strategies.

Due to the issues mentioned prior, methods that incorporate both exploration and feedback (as these

Approach	Model	SWE-Bench Verified	SWE-Bench Lite
Agentless (Xia et al., 2024)	gpt-4o	33.2	24.3
	Claude-3.5-Sonnet	53.0	-
AutoCodeRover (Zhang et al., 2024b)	gpt-4o	28.8	22.7
	gpt-4	-	19.0
MASAI (Arora et al., 2024)	gpt-4o	–	28.3
SWE-Agent (Yang et al., 2024c)	Claude-3.5-Sonnet	33.6	23.0
	gpt-4o	23.2	18.3
SWE-Gym (Pan et al., 2024)	SWE-Gym-32B	32.0	26.0
SWE-Search (Antoniades et al., 2024)	gpt-4o	-	31.0
Lingma (Ma et al., 2024)	Lingma SWE-GPT 72B	30.2	22.0
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
OpenHands (Wang et al., 2024e)	gpt-4o	–	22.0
	Claude-3.5-Sonnet	–	26.0

Table 3: Performance on **SWE-Bench Verified** and **SWE-Bench Lite**; Performance is measured by resolved rate.

search-based techniques do) have shown superior performance. These methods can actively counteract model rigidity by encouraging the model to deviate from its default generation paths, resulting in more diverse and contextually appropriate outputs. In fact, several works support the case for re-sampling, or exploring multiple and diverse paths through a combination of models.

Observation G.5: Agentic approaches appear to dominate both execution-based and CoT strategies.

Agentic approaches 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.

Observation G.6: Agentic approaches that scale inference with search are highly competitive and can even outperform other strategies.

Furthering the case for counteracting model rigidity, agents that integrate search to scale their inference achieve state-of-the-art performance. ToC and SWE-Search in particular show that integrating diverse trajectories (either via multiple models or collaborative agents) and incorporating backtracking can lead to major gains. This reinforces the case for exploration. Indeed, SWE-Search tops the leaderboard, achieving 31% on SWE-Bench Lite (Table 3). We leave it to future work to undertake the validation and theoretical substantiation of the premises discussed here.

6 Conclusion and Future Work

Reasoning techniques and agents have driven major AI gains and been successfully adapted for software engineering (SE). AI for SE is a rapidly-evolving field, with reasoning/agents at the cutting edge. We present the first survey focused on code reasoning for SE tasks, with a taxonomy of techniques and special attention to SE agents. We compare methods across benchmarks, highlight hybrid and agent-specific strategies, and analyze how code properties influence performance. Our discussion surfaces promising benchmarks and motivates future directions in code reasoning and SE agents.

We highlight several directions: it's clear that adapting reasoning techniques to code improves performance on code tasks; *this should be explored for diverse programming languages*; we observe that code-oriented plans, combined with other techniques (search, execution feedback) are more robust; however, *incorporating code-specific plans or using structure in CoT for agents is still under-explored.* Most approaches have been applied to Python (and Python-specific benchmarks); multilingual reasoning techniques and benchmarks can lead to the development of a more general code reasoning capability; moreover, different approaches need to be explored for other tasks (e.g., *search-based techniques that leverage modularity, LM and execution-based heuristics for test-case generation*).

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 through, 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. 1 and 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. 2025. Otter: Generating tests from issues to validate swe patches. *Preprint*, arXiv:2502.05368.

Anthropic. 2025. Claude 3.7 sonnet and claude code.

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

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

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

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

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

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, and 1 others. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

Hyungjoo Chae, Yeonghyeon Kim, Seungone Kim, Kai Tzu-iunn Ong, Beong-woo Kwak, Moohyeon Kim, Seonghwan Kim, Taeyoon Kwon, Jiwan Chung, Youngjae Yu, and 1 others. 2024. Language models as compilers: Simulating pseudocode execution improves algorithmic reasoning in language models. *arXiv preprint arXiv:2404.02575*.

Dong Chen, Shaoxin Lin, Muhan Zeng, Daoguang Zan, Jian-Gang Wang, Anton Cheshkov, Jun Sun, Hao Yu, Guoliang Dong, Artem Aliev, and 1 others. 2024a. Coder: Issue resolving with multi-agent and task graphs. *arXiv preprint arXiv:2406.01304*.

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

Liguo Chen, Qi Guo, Hongrui Jia, Zhengran Zeng, Xin Wang, Yijiang Xu, Jian Wu, Yidong Wang, Qing Gao, Jindong Wang, and 1 others. 2024b. A survey on evaluating large language models in code generation tasks. *arXiv preprint arXiv:2408.16498*.

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, and 39 others. 2021a. Evaluating large language models trained on code. *Preprint*, arXiv: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, and 1 others. 2021b. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

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

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

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2024c. Teaching large language models to self-debug. In *The Twelfth International Conference on Learning Representations*.

Yongchao Chen, Harsh Jhamtani, Srinagesh Sharma, Chuchu Fan, and Chi Wang. 2024d. Steering large language models between code execution and textual reasoning. *arXiv preprint arXiv:2410.03524*.

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

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

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

Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Jingyuan Ma, Rui Li, Heming Xia, Jingjing Xu, Zhiyong Wu, Tianyu Liu, and 1 others. 2022. A survey on in-context learning. *arXiv preprint arXiv:2301.00234*.

Yihong Dong, Jiazheng Ding, Xue Jiang, Ge Li, Zhuo Li, and Zhi Jin. 2025a. Codescore: Evaluating code generation by learning code execution. *ACM Trans. Softw. Eng. Methodol.*, 34(3).

Yihong Dong, Jiazheng Ding, Xue Jiang, Ge Li, Zhuo Li, and Zhi Jin. 2025b. Codescore: Evaluating code generation by learning code execution. *ACM Transactions on Software Engineering and Methodology*, 34(3):1–22.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, and 1 others. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I. Wang. 2024. Cruxeval: A benchmark for code reasoning, understanding and execution. *Preprint*, arXiv:2401.03065.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, and 1 others. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

Hojae Han, Seung won Hwang, Rajhans Samdani, and Yuxiong He. 2025. Convcodeworld: Benchmarking conversational code generation in reproducible feedback environments. *Preprint*, arXiv: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. 2021a. Measuring coding challenge competence with apps. *Preprint*, arXiv: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. 2021b. Measuring coding challenge competence with apps. *Preprint*, arXiv:2105.09938.

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

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

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

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, and 1 others. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, and 1 others. 2024. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*.

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

Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, and 1 others. 2024. Openai o1 system card. *arXiv preprint arXiv:2412.16720*.

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

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

Nan Jiang, Xiaopeng Li, Shiqi Wang, Qiang Zhou, Soneya Binta Hossain, Baishakhi Ray, Varun Kumar, Xiaofei Ma, and Anoop Deoras. 2025. Ledex: Training llms to better self-debug and explain code. *Preprint*, arXiv:2405.18649.

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

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024a. Swe-bench: Can language models resolve real-world github issues? *Preprint*, arXiv:2310.06770.

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

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

Mohammad Abdullah Matin Khan, M Saiful Bari, Xuan Long Do, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. 2024. 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)*, pages 6766–6805, Bangkok, Thailand. Association for Computational Linguistics.

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

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. 2025. Spider 2.0: Evaluating language models on real-world enterprise text-to-sql workflows. *Preprint*, arXiv:2411.07763.

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

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

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

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, and 39 others. 2023. Starcoder: may the source be with you! *Transactions on Machine Learning Research*. Reproducibility Certification.

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

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, and 7 others. 2022b. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Changshu Liu and Reyhaneh Jabbarvand. 2025. 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. Association for Computing Machinery.

Changshu Liu, Shizhuo Dylan Zhang, Ali Reza Ibrahimzada, and Reyhaneh Jabbarvand. 2024a. Codemind: A framework to challenge large language models for code reasoning. *Preprint*, arXiv:2402.09664.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023a. 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*.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023b. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36:21558–21572.

Mingwei Liu, Juntao Li, Ying Wang, Xueying Du, Zuoyu Ou, Qiuyuan Chen, Bingxu An, Zhao Wei, Yong Xu, Fangming Zou, and 1 others. 2025. Code copycat conundrum: Demystifying repetition in llm-based code generation. *arXiv preprint arXiv:2504.12608*.

Zhihan Liu, Shenao Zhang, Yongfei Liu, Boyi Liu, Yingxiang Yang, and Zhaoran Wang. 2024b. Dstc: Direct preference learning with only self-generated tests and code to improve code lms. *Preprint*, arXiv:2411.13611.

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

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

Dung Nguyen Manh, Thang Phan Chau, Nam Le Hai, Thong T. Doan, Nam V. Nguyen, Quang Pham, and Nghi D. Q. Bui. 2025. Codemmlu: A multi-task benchmark for assessing code understanding reasoning capabilities of codellms. *Preprint*, arXiv:2410.01999.

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

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. 2023. Octopack: Instruction tuning code large language models. *ArXiv*, abs/2308.07124.

Niels Mündler, Mark Niklas Müller, Jingxuan He, and Martin Vechev. 2025. Swt-bench: Testing and validating real-world bug-fixes with code agents. *Preprint*, arXiv:2406.12952.

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

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

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

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

OpenAI. 2024. Introducing swe-bench verified.

Albert Orwall. 2024. Moatless tools.

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

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

Rangeet Pan, Myeongsoo Kim, Rahul Krishna, Raju Pavuluri, and Saurabh Sinha. 2025a. Aster: Natural and multi-language unit test generation with llms. *Preprint*, arXiv:2409.03093.

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

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

Qiwei Peng, Yekun Chai, and Xuhong Li. 2024a. Humaneval-xl: A multilingual code generation benchmark for cross-lingual natural language generalization. *Preprint*, arXiv:2402.16694.

Qiwei Peng, Yekun Chai, and Xuhong Li. 2024b. HumanEval-XL: A multilingual code generation benchmark for cross-lingual natural language generalization. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pages 8383–8394, Torino, Italia. ELRA and ICCL.

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

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

Archiki Prasad, Elias Stengel-Eskin, Justin Chih-Yao Chen, Zaid Khan, and Mohit Bansal. 2025. Learning to generate unit tests for automated debugging. *Preprint*, arXiv:2502.01619.

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

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. 2021. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, volume 1.

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

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

Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, Red Hook, NY, USA.

Erik Schluntz and Barry Zhang. 2024. Building effective agents.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Y Wu, and 1 others. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*.

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

Qiushi Sun, Zhirui Chen, Fangzhi Xu, Kanzhi Cheng, Chang Ma, Zhangyue Yin, Jianing Wang, Chengcheng Han, Renyu Zhu, Shuai Yuan, and 1 others. 2024a. A survey of neural code intelligence: Paradigms, advances and beyond. *arXiv preprint arXiv:2403.14734*.

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

Weisong Sun, Yun Miao, Yuekang Li, Hongyu Zhang, Chunrong Fang, Yi Liu, Gelei Deng, Yang Liu, and Zhenyu Chen. 2025. Source Code Summarization in the Era of Large Language Models . In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 419–431, Los Alamitos, CA, USA. IEEE Computer Society.

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

Zhao Tian, Junjie Chen, and Xiangyu Zhang. 2025. Fixing Large Language Models' Specification Misunderstanding for Better Code Generation . In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 645–645, Los Alamitos, CA, USA. IEEE Computer Society.

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

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

Junqiao Wang, Zeng Zhang, Yangfan He, Yuyang Song, Tianyu Shi, Yuchen Li, Hengyuan Xu, Kunyu Wu, Guangwu Qian, Qiuwu Chen, and 1 others. 2024b. Enhancing code llms with reinforcement learning in code generation. *arXiv preprint arXiv:2412.20367*.

Wenhan Wang, Chenyuan Yang, Zhijie Wang, Yuheng Huang, Zhaoyang Chu, Da Song, Lingming Zhang, An Ran Chen, and Lei Ma. 2025. TESTEVAL: Benchmarking large language models for test case generation. In *Findings of the Association for Computational Linguistics: NAACL 2025*. Association for Computational Linguistics.

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

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024d. Executable code actions elicit better llm agents. *Preprint*, arXiv:2402.01030.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, and 1 others. 2024e. Openhands: An open platform for ai software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations*.

Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, and 1 others. 2022a. Emergent abilities of large language models. *arXiv preprint arXiv:2206.07682*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, and 1 others. 2022b. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

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

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

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

Fengli Xu, Qianyue Hao, Zefang Zong, Jingwei Wang, Yunke Zhang, Jingyi Wang, Xiaochong Lan, Jiahui Gong, Tianjian Ouyang, Fanjin Meng, and 1 others. 2025. Towards large reasoning models: A survey of reinforced reasoning with large language models. *arXiv preprint arXiv:2501.09686*.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, and 1 others. 2024a. Qwen2. 5 technical report. *arXiv preprint arXiv:2412.15115*.

Dayu Yang, Tianyang Liu, Daoan Zhang, Antoine Simoulin, Xiaoyi Liu, Yuwei Cao, Zhaopu Teng, Xin Qian, Grey Yang, Jiebo Luo, and 1 others. 2025. Code to think, think to code: A survey on code-enhanced reasoning and reasoning-driven code intelligence in llms. *arXiv preprint arXiv:2502.19411*.

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

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024c. Swe-agent: Agent-computer interfaces enable automated software engineering. *Preprint*, arXiv: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. 2024d. Swe-bench multimodal: Do ai systems generalize to visual software domains? *Preprint*, arXiv:2410.03859.

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

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

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

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

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. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics.

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

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

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. 2025. Multi-swe-bench: A multilingual benchmark for issue resolving. *Preprint*, arXiv:2504.02605.

Huaye Zeng, Dongfu Jiang, Haozhe Wang, Ping Nie, Xiaotong Chen, and Wenhu Chen. 2025. Acecoder: Acing coder rl via automated test-case synthesis. *Preprint*, arXiv:2502.01718.

Huan Zhang, Wei Cheng, Yuhan Wu, and Wei Hu. 2024a. 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*, pages 1319–1331.

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

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

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, and 1 others. 2023a. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 5673–5684.

Wenqing Zheng, S P Sharan, Ajay Kumar Jaiswal, Kevin Wang, Yihan Xi, Dejia Xu, and Zhangyang Wang. 2023b. Outline, then details: Syntactically guided coarse-to-fine code generation. *Preprint*, arXiv:2305.00909.

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

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, and 14 others. 2025. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *Preprint*, arXiv: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.

B Related Surveys

Wei et al., 2022b introduce CoT as a form of in-context learning which induces 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 and 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.

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 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 and 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.

B.1 Summaries

Chen et al. (2024d) show that larger models favor textual reasoning over code execution. However, textual reasoning has inherent limitations in solving challenges in math, logic, optimization, searching which is unlikely to be solved by scaling up the model and data size. Inverse Scaling issue: GPT-4o might perform worse than GPT-3.5 on some tasks like Game of 24 because it defaults to using textual reasoning as opposed to code generation (GPT 3.5). Introduce a method to steer the model towards code generation over textual reasoning. Introduce a hybrid approach to mix both textual reasoning and code reasoning. Introduce a mechanism to allow the model to assess whether textual or code-based reasoning are appropriate. Insights from the paper are: textual reasoning alone lacks the precision required for computational tasks. Code is more precise and unambiguous. Text-based reasoning may struggle on tasks that require exact calculations or procedural steps. Suggests there is an open area for developing a method that can intelligently decide when to use code reasoning and when to use textual reasoning. They seem to show that iterative refinement helps for code-based reasoning. Overall, combining code-based and text-based reasoning is an effective strategy. "The methods that rely more on code tend to improve with multi-turn refinement, likely because code execution provides additional feedback for reflection (Gou et al., 2023). In contrast, the degradation of text-based methods suggests that LLMs can worsen answers through self-reflection alone, supporting findings from previous studies (Huang et al., 2023)." They test on several tasks involving arithmetic, logic, puzzles. A recurring task is Game 24. Date-related questions, boolean expressions, math.

Chae et al. (2024) has a think phase and an execution phase. Generates firstly a plan in pseudocode then simulates execution of the pseudocode. Having a model plan in pseudocode seems to significantly boost reasoning. Dominant strategy might be structure-aware?

B.2 Execution-driven Reasoning

B.2.1 Self-Evaluation of Execution Behavior

`Self-debugging` (Chen et al., 2024c) approach, teaches the model to self-debug. The code explanation along with the execution results constitute the feedback message that is used for debugging the generated code. When unit tests are not available, the feedback is purely based on code explanation.

`AlphaCodium`, proposed by (Ridnik et al., 2024), is a flow to improve code LLM performance that does not require training a model. Best practices in AlphaCodium's flow are: using YAML structured output, bullet point analysis for semantic reasoning, modular code generation, soft decision with double validation, encourage exploration and postpone direct decisions, and use of test anchors.

In `revisited self-debugging` (Chen et al., 2025b) authors explored both post-execution and in-execution self-debugging, leveraging self-generated tests. They found that post-execution suffers from bias in self-generated tests, while in-execution self-debugging minimizes the bias by focusing on the intermediate states during the program execution, and consistently outperforms post-execution approach on both basic and competitive tasks.

μFix (Misunderstanding Fixing) (Tian et al., 2025) thought-eliciting prompting techniques are combined with feedback-based prompting to improve the code generation performance of LLMs. Feedback-based prompting focuses on trying to understand the root cause of failure of tests by analyzing the actual understanding implicitly utilized by LLMs for code generation through code summarization.

B.2.2 Training with Execution-based Feedback

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. The generated code is re-ranked based on the verification score and the LLM generation probability, and marginalizing over programs with the same execution results.

(Jiang et al., 2025) proposed **LEDEX**, a training framework to improve the self-debugging capabil-

ity of LLMs using a chain of explanations on the wrong code followed by code refinement. Their automated pipeline collects a high-quality dataset for code explanation and refinement by generating a number of explanations and refinement trajectories from the LLM itself, or a larger teacher model, and filtering via execution verification. Then a combined Supervised Full Tuning (SFT) and Reinforcement Learning (RL) technique is used on both success and failure trajectories to train the code model.

B.2.3 Automated Test Generation

UTGEN (Prasad et al., 2025) is a data creation and training recipe that bootstraps training data for UT generation from existing code generation datasets by perturbing code to simulate errors, generating failing unit test and augmenting it with CoT rationales. UTGen yields a higher number of unit tests that have both attacking inputs and correct outputs. Along with UTGEN, authors presented UTDEBUG, an improved multi- turn debugging method that improves the output accuracy of generated UTs by scaling test-time compute via self-consistency, and regularizes the debugging process by generating multiple UTs and accepting code edits only if the revised code passes more generated UTs, backtracking edits otherwise.

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. In the UT generation process, a LLM is asked to imagine and generate 20 test cases from a refined code problem description (instruction), then another stronger LLM is used as a proxy to validate the quality of the generated UTs. With the aid of test cases they create preference pairs based on pass rates over sampled programs to train reward models with Bradley-Terry loss. Using the preference pairs data, they leverage RL on both reward models and test-case pass rewards, resulting in improvements of the models for several benchmarks.

In a similar way, (Liu et al., 2024b) proposes Direct Preference Learning with Only Self-Generated Tests and Code (DSTC), a framework that uses only self-generated code snippets and tests to construct preference pairs with direct preference learning to improve LM coding accuracy without external annotations. The UT generation process is joint with the code generation process, where the LLM is prompted to generate multiple code snippets and tests for each given instruction. ASTER (Pan et al., 2025a) is a multilingual unit test generator built with LLMs guided by lightweight program analysis. ASTER is a generic pipeline that incorporates static analysis to guide LLMs in generating compilable and high-coverage test cases for Python and Java. They showed that LLM-based test generation, guided by static analysis, can be competitive sometimes outperform, state-of-the-art test- generation techniques in coverage while also producing considerably more natural test cases that developers find easy to understand.

More recently, SWT-Bench (Mündler et al., 2025) is a benchmark based on GitHub repositories, containing real-world issues, ground-truth bug-fixes, and golden tests for Python. This benchmarks has over $1,900$ samples that were created by transforming SWE-BENCH (Jimenez et al., 2024a) from code repair to test generation. Authors of SWT-Bench performed a study and found out that LLMs perform well at generating relevant test cases, where Code Agents designed for code repair have better performance than systems designed specifically for test generation.

B.3 Inference Scaling

B.3.1 Sampling

AlphaCode. (Li et al., 2022a) solve competitive programming problems using large-scale sampling followed by filtering and clustering. AlphaCode diversifies the generation process by generating half of its samples in Python and half in C++, randomizing problem tags and ratings in the prompt, and using a high sampling temperature. With these techniques, they are able to generate millions of sample solutions per programming problem. This generation phase is then followed by filtering and clustering. Test cases are integral to these phases and are either (1) provided along with the problem statement or (2) generated using another model. After filtering samples by (1), thousands of candidate solutions remain. Thus, in (2) a trained model is used to generate new test inputs, which are then used to assess the remaining generated samples. The samples are clustered according to their programming behavior after being run against the generated tests. One solution is selected from each cluster to constitute the final pool of candidates.

REx. 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 their problem formulation, each "arm" is a program, and "pulling the arm" corresponds to refining a program. The heuristic reward is the fraction of specifications (test cases) satisfied by the program. Using this sampling technique, they are able to select the program according to its probability of giving the best reward. This is equivalent to solving the programming task using the fewest LLM calls possible. They show promising results on competitive programming problems.

S*. Li et al. (2025a) takes a hybrid approach to sampling, first generating N diverse programs in parallel then refining the programs using iterative debugging. Their iterative debugging is informed by execution results on public test cases. The revision process is complete once the program passes all public test cases or reaches the max number of attempts. They evaluate on code generation benchmarks, including LiveCodeBench and CodeContests.

B.3.2 Search

Tree of Thoughts and Guided Tree-of-Thought. In (Yao et al., 2023a)'s work, Tree-of-thoughts, or ToT, takes inspiration from the human nature of problem-solving, where people solve a problem by searching through a combinatorial problem search space. The ToT paradigm 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. The language model's reasoning is used as the heuristic, which contrasts with traditional approaches that use learned or programmed rules. To travers the tree, ToT uses classic search strategies: breadth-first search (BFS) or depth-first search (DFS).

Similarly, guided tree-of-thought (Long, 2023) also uses a tree-search algorithm, where the LLM is used as a heuristic for generating search steps. GToT uses prompting to reach an intermediate solution to a problem, then introduces a checker, which assesses the correctness or validity of the intermediate solution. A controller module oversees the entire tree search and can control backtracking if a partial solution is invalid or unpromising, allowing the system to explore long-range reasoning.

Ouédraogo et al. (2024) explore the effectiveness of various prompting techniques, including ToT and GToT, on the task of test generation. They show that GToT prompting is effective in generating syntactically-correct and compilable test suites, and can also lead to test suites with superior code coverage.

Outcome-Refining Process. (Yu et al., 2024) propose ORPS, Outcome-Refining Process Supervision for code generation. Their paradigm performs beam-search over a "reasoning tree." In this tree, each state captures the complex nature of code; a state contains information about the theoretical reasoning, code implementation, and execution outcome of a potential solution. The beam-search implementation works as follows: for each state, multiple reasoning chains are stored at once. For every chain, the algorithm (1) updates the state with LM-generated reasoning and code implementation (2) runs the code to obtain feedback (i.e. records its performance on tests, its memory usage, number of AST nodes, etc), then (3) uses the same LM as a critic to generate a critique and numerical "step" reward. Using these notions of self-refinement and self-critique, only the most promising solution paths are retained.

C Agents

CodeTree. CodeTree (Li et al., 2024) frames code generation as a tree-search problem using a combination of planning, execution-guided reasoning, and sampling. CodeTree employs heuristic strategies similar to other search-based approaches, using testing pass rate (as in REx, S*) combined with LM critique as a heuristic (as in ORPS, ToT/GToT) to guide the traversal of the tree. Unlike other approaches, it uses a collaborative, multi-agent framework; each sub-agent is specialized for a particular type of reasoning. The thinker agent generates a strategy plan and decides the number of samples to generate; the solver implements the plan and generates code solutions; the critic agent, which scores the solution based on test pass rate and robustness, then decides whether to accept, dismiss, or refine a solution. The debugger then outputs a refined program using the critic agent's feedback. Once the code has been generated and feedback obtained, the solution is added as a tree node along with the relevant attributes, like the strategy plan.

ToC. ToC (Ni et al., 2024) also presents the reasoning process as a tree. They represent nodes in a similar way to CodeTree, using the thought, generated code, and execution results as attributes of the node. Contrary to CodeTree, which uses a combination of test-pass rates and a soft score to judge

robustness of a solution, ToC uses a binary heuristic: execution pass or execution fail. If a node fails, the children will be explored until a viable solution is reached. ToC integrates execution-based reasoning with a search-based, multi-strategy sampling approach. Following code execution, they incorporate a post-execution reflective phase, which allows them to perform iterative improvements. Here, they leverage multiple models and varying temperature settings to expand the diversity of potential solutions.

Arora et al., 2024 take inspiration from modularization and develop MASAI, a modular SE agent with 5 sub-agents for different tasks: Test Template Generator, Issue Reproducer, Edit Localizer, Fixer, and Ranker. CodeR (Chen et al., 2024a) is a multi-agent framework with task graphs for resolving issues. Similar to role-based teams of humans that resolve issues, the framework also defines roles and actions like Manager, Reproducer, Fault Localizer, Editor and Verifier. PairCoder (Zhang et al., 2024a) is inspired by the software development practice of pair programming. It incorporates two collaborative agents: NAVIGATOR agent for high-level planning and DRIVER for specific implementation. HyperAgent (Phan et al., 2024) is a multi-lingual (Python/ Java), multi-agent system that emulates the workflow of human developers. It consists of four specialized agents called Planner, Navigator, Code Editor and Executor, which are capable of managing the full SE task life-cycle from planning to verification. AgileCoder (Nguyen et al., 2024) is a multi-agent system that uses sprints and agile roles (e.g., Product Manager, Developer, Scrum Master) to coordinate work based on user input.

D 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., 2024a).

MBPP (Austin et al., 2021a) (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., 2023a) 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., 2021a) 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., 2025a), 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.

E 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., 2025a) 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.

E.1 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 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 $n@k$.

F Results Tables

We manually inspected every work in our survey and collated self-reported and cross-reported entries on *common* benchmarks. We report on benchmarks that intersect across approaches and use intersecting models/benchmarks to make observations of their trends. In our surveyed works, they were the following: APPS (Hendrycks et al., 2021b), HumanEval (Chen et al., 2021b), HumanEval+ (Liu et al., 2023b), HumanEval-ET (Dong et al., 2025b), multi-language benchmarks HumanEval-X (Zheng et al., 2023a) and HumanEval-XL (Peng et al., 2024b). MBPP (Austin et al., 2021b), MBPP+, MBPP-sanitized (Austin et al., 2021c), MBPP-ET. See tables: 4, 5, 6, 8, 7.

G Observations Extended

G.1 Observation 1:

Chain of Grounded Objectives (CGO) outperforms Self-Planning and ClarifyGPT with gpt-3.5 on MBPP-S; it is also better than Self-Planning on MBPP+. This also holds true for Llama-3-8B-Instr, where CGO is better than Self-Planning. On MBPP and MBPP+ with gpt-4o-mini, ScoT is better than Self-Planning (Table 7).

G.2 Observation 2:

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 (a modular approach) also outperforms SCoT on APPs overall with gpt-3.5 (Table 4, 7, 8).

G.3 Observation 3:

MuFix and Self-Debugging surpass other CoT baselines (CGO, SCoT, Self-Plan, Clarify-GPT) on HumanEval (gpt-3.5). Revisiting Self-Debugging beats PlanSearch on HE+ (Claude-3.5-Sonnet). MuFix and Self-Debugging outperform ClarifyGPT on MBPP-ET (gpt-3.5), further reinforcing dominance of execution-based methods. On MBPP with gpt-3.5, Self-Debugging surpasses SCoT by a large margin. MuFix and Self-Debugging outperform UniCoder on HE. The findings hold true on the APPS benchmark, where MuFix outperforms CodeChain, SCoT, and Self-Planning with gpt-3.5. This is true for DeepSeek-Coder as well, where MuFix, Self-Debugging, and CYCLE models, which are smaller-sized parameter models but finetuned, outperform SCoT. (Tables 4, 7, 8)

G.4 Observation 4:

CodeTree outperforms Revisiting Self Debugging on MBPP+ with gpt-4o. ORPS outperforms MoT and other structure-based and plan-based approaches (like SCoT and Self-Planning) on MBPP with gpt-4o-mini. This is also true for MBPP with DeepSeekCoder, ORPS outperforms UniCoder by a large margin. REx with gpt-4 also claims to achieve the state-of-the art on APPS, with roughly 70%. S* also beats PlanSearch on LCB with o1-mini and 4o-min. (Tables 5, 7)

G.5 Observation 5:

PairCoder and AgileCoder significantly 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 MuFix, Self-Debugging, and UniCoder; also with DeepSeek-Coder, PairCoder outperforms UniCoder on MBPP. Also true for gpt-4 on MBPP-S, where PairCoder outperforms ClarifyGPT. (Tables 4, 7, 8)

G.6 Observation 6:

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^3ToolEval, ToC is better than CodeAct. Moreover, SWE-Search, which combines inference scaling in an agentic approach, dominates the leaderboard on SWE-Bench Lite. (Tables 5, 6, 7, 8).

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., 2023b)	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	–	–	–	–	$\sim$ 70

Table 4: Performance across the APPS benchmark (Hendrycks et al., 2021b), 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 $k = 1{,}000$.

Approach	Model	LCB	CodeContests	M^{3}ToolEval
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., 2023b)	ChainCoder-1B	-	$\sim$ 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 5: Performance across the **LiveCodeBench (LCB)**, **CodeContests (test set)**, and **M^{3}ToolEval**. 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., 2024c)	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 6: 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 and 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 7: Performance on the **MBPP+**, **MBPP**, **MBPP-ET**, and **MBPP-sanitized** benchmarks. All results are reported as $pass@1$.

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 and Zhang, 2025)	DeepSeek-R1	88.4	95.1	–	–	94.5
	gpt-4o-mini	83.5	92.1	–	–	91.5
MSCoT (Pan and 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., 2024b)	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	–	–	–
	SemCoder-6.7B	68.9	73.2	–	–	–

Table 8: Performance on the **HumanEval+**, **HumanEval**, **HumanEval-XL**, **HumanEval-X**, and **HumanEval-ET** benchmarks. All results are reported as $pass@1$.

Approach	CoT			Execution-based		Inference Scaling			Other		
	Plan	Struct	FT	Gen-AI Tests	Benchmark Tests	Sampling	LM Heuristic	Exec. Heuristic	MV	RR	RL
PlanSearch	✓				✓	✓					
Self-Planning, ClarifyGPT	✓										
SCoT,CGO,MoT,CodeChain		✓									
UniCoder,ChainCoder,MSCoT		✓	✓								
COTTON	✓		✓								
SemCoder		✓	✓	✓							
MSCoT		✓	✓								
Self-Debug					✓				✓		
CodeCOT,AlphaCodium	✓			✓	✓						
Revisit Self-Debug				✓	✓				✓		
μFix	✓			✓							
LEVER					✓	✓				✓	
CYCLE					✓	✓					
LEDEX	✓				✓	✓					✓
ORPS				✓	✓						
GToT							✓				
S*				✓	✓	✓				✓?	
REx											

Table 9: LLM Reasoning based approaches for code tasks and key components. CoT (Chain-of-Thought); Exe-based (Execution-based feedback); GenAI Tests (Generated Tests with LLMs); MV (Majority Vote); RR (Re-Ranking); RL (Reinforcement-Learning). Each approach has a dominant strategy by which we categorize our taxonomy: CoT and Planning , Execution-driven , and sampling or search . For agentic see Tab. 10.

Approach	Workflow	Reason. Model			Agent Optim.		Inf. Scaling
		SFT	RL	Verifier	Multi-Agent	Tools	
Agentless	✓						
AutoCodeRover	✓					✓	
SWE-Agent						✓	
CodeAct		✓				✓	
OpenHands, MASAI, CodeR, AgileCoder					✓	✓	
PairCoder	✓				✓		
HyperAgent					✓		
Lingma, SWE-Fixer	✓	✓					
SWE-Gym		✓		✓		✓	✓
SWE-RL			✓				
CodeTree					✓		✓
ToC							✓
SWE-Search					✓	✓	✓

Table 10: In our taxonomy Agents are classified as employing one of the following techniques 1. Workflow 2. Reasoning Model improvement 3. Agent optimization 4. Inference scaling. However many agents employ multiple techniques. For example, SWE-Gym is classified in Reasoning model improvement category, but they also train a verifier model for inference scaling. This table highlights such nuances.