A Study on Thinking Patterns of Large Reasoning Models in Code Generation

Kevin Halim
kevincon001@e.ntu.edu.sg
Nanyang Technological University
Singapore

Sin G. Teo
teosg@i2r.a-star.edu.sg
Institute for Infocomm Research (I2R), A*Star
Singapore

Ruitao Feng
ruitao.feng@scu.edu.au
Southern Cross University
Australia

Zhenpeng Chen
zhenpeng.chen@ntu.edu.sg
Nanyang Technological University
Singapore

Yang Gu
yang.gu24@u.nus.edu
National University of Singapore
Singapore

Chong Wang
chong.wang@ntu.edu.sg
Nanyang Technological University
Singapore

Yang Liu
yangliu@ntu.edu.sg
Nanyang Technological University
Singapore

arXiv:2509.13758v1 [cs.SE] 17 Sep 2025

Abstract

Currently, many large language models (LLMs) are utilized for software engineering tasks such as code generation. The emergence of more advanced models known as large reasoning models (LRMs), such as OpenAI's o3, DeepSeek R1, and Qwen3. They have demonstrated the capability of performing multi-step reasoning. Despite the advancement in LRMs, little attention has been paid to systematically analyzing the reasoning patterns these models exhibit and how such patterns influence the generated code. This paper presents a comprehensive study aimed at investigating and uncovering the reasoning behavior of LRMs during code generation. We prompted several state-of-the-art LRMs of varying sizes with code generation tasks and applied open coding to manually annotate the reasoning traces. From this analysis, we derive a taxonomy of LRM reasoning behaviors, encompassing 15 reasoning actions across four phases.

Our empirical study based on the taxonomy reveals a series of findings. First, we identify common reasoning patterns, showing that LRMs generally follow a human-like coding workflow, with more complex tasks eliciting additional actions such as scaffolding, flaw detection, and style checks. Second, we compare reasoning across models, finding that Qwen3 exhibits iterative reasoning while DeepSeek-R1-7B follows a more linear, waterfall-like approach. Third, we analyze the relationship between reasoning and code correctness, showing that actions such as unit test creation and scaffold generation strongly support functional outcomes, with LRMs adapting strategies based on task context. Finally, we evaluate lightweight prompting strategies informed by these findings, demonstrating the potential of context- and reasoning-oriented prompts to improve LRM-generated code. Our results offer insights and practical implications for advancing automatic code generation.

CCS Concepts

• **Software and its engineering**;

Keywords

Large Reasoning Models, Code Generation, Taxonomy

ACM Reference Format:
Kevin Halim, Sin G. Teo, Ruitao Feng, Zhenpeng Chen, Yang Gu, Chong Wang, and Yang Liu. 2025. A Study on Thinking Patterns of Large Reasoning Models in Code Generation. In . ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Large language models (LLMs) have become increasingly prominent in software engineering, particularly in the domain of automated code generation. Tools like GitHub Copilot [15] demonstrate the utility of LLMs by producing code that is often syntactically correct and contextually plausible [2]. These models assist with tasks such as code completion, code summarization, and bug fixing, offering substantial productivity gains. However, LLM-generated code often lacks deeper semantic understanding, leading to issues such as functional requirement violations [41], hallucinations [22], and outputs that do not align with user intent. Additionally, their "black-box" nature [42] and limited capacity for robust reasoning pose serious challenges to their reliability in practical software development.

The emergence of Large Reasoning Models (LRMs) represents a significant advancement in model reasoning capabilities. Notable examples include OpenAI's o3 [32], Anthropic Claude 4 [1], Qwen3 [38], and QwQ [35]. These models generate explicit intermediate reasoning steps, commonly called *reasoning traces* , which offer valuable insight into how outputs are produced [29]. In contrast to

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Conference'17, Washington, DC, USA
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

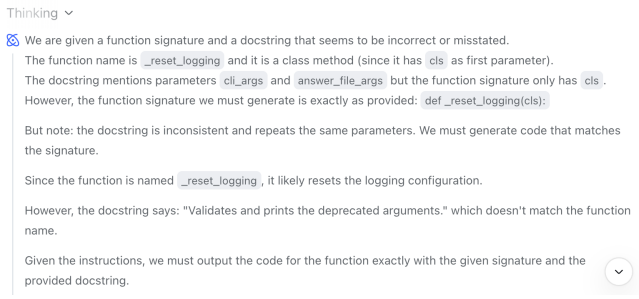

Figure 1: An motivating example where DeepSeek-R1 is thinking before generating code.

traditional LLMs that primarily rely on pattern matching, LRMs are designed for multi-step deliberation and better generalization to complex problems [5]. This enhanced reasoning improves effectiveness and also plays a crucial role in transparency and interpretability, fostering understanding, building user trust, and ensuring alignment in application tasks. For instance, as illustrated in Figure 1, DeepSeek-R1 identifies a given signature as a class method based on the `cls` parameter and detects ambiguities in the signature and docstring during its internal reasoning before generating code.

On the other hand, to better understand the reasoning capabilities of language models, recent research has proposed various taxonomies describing how models reason across different domains. Several studies [4, 29, 30, 33] focus on general problem-solving, critique, mathematical reasoning, and structured reasoning phases. While these taxonomies offer valuable insights, they are not tailored to code-related tasks, limiting their relevance to code generation. In contrast, other works [27, 37] examine reasoning in code generation, highlighting issues in logical consistency and control flow. However, these studies evaluate traditional LLMs and do not investigate the reasoning traces (as shown Figure 1) of large reasoning models (LRMs). As a result, existing literature either overlooks code generation or focuses on models not designed for explicit reasoning traces. This raises an important question: *How do LRMs perform reasoning in code generation, and are there identifiable patterns in their reasoning behaviors?*

Taxonomy. To bridge this gap, we analyze reasoning traces of LRMs in code generation, aiming to uncover and model common reasoning behaviors in how models interpret prompts and generate code, thereby offering insights to improve code generation techniques. We focus on five open-source Qwen-series LRMs, including DeepSeek-R1-7B, Qwen3-(1.7B, 8B, 14B), and QwQ-32B, evaluated on Python tasks from the CoderEval benchmark [39] using prompts designed to elicit explicit reasoning. To identify reasoning behaviors, we apply an open coding method to manually annotate 1,150 traces across different task complexities from the five models, noting recurring reasoning actions. We organize these behaviors into four phases, merging and refining similar actions to construct a taxonomy of 15 reasoning actions, spanning from requirements gathering to reflection.

Study. Building on this taxonomy, we conduct an empirical study to address a series of research questions. **RQ1:** We identify common reasoning patterns in LRM-based code generation by analyzing the sequence of reasoning actions within traces. The results show that LRMs generally follow a human-like coding workflow—analyzing requirements, clarifying ambiguities, comparing solutions, implementing code, and reviewing for defects—while simpler tasks involve lighter reasoning, and more complex tasks trigger additional actions such as scaffolding, flaw detection, or style checks. **RQ2:** We compare reasoning behaviors across different LRMs. The results indicate that Qwen3 models exhibit highly similar, iterative reasoning patterns, whereas DeepSeek-R1-7B follows a more linear, waterfall-like approach, likely reflecting differences in the reasoning traces used during training. **RQ3:** We analyze how reasoning behaviors impact functional correctness by correlating Pass@1 with reasoning actions and patterns. Unit test creation and complete code generation strongly support correctness, and LRMs adjust their reasoning strategies such as knowledge recall, alternative exploration, scaffold code generation, and edge case generation based on task dependency and context. **RQ4:** We evaluate the feasibility of two lightweight prompting strategies as potential improvements over the initial prompt, motivated by the key findings. The results highlight the potential of incorporating context or reasoning-oriented guidelines into prompts to enhance LRM-generated code. Beyond the empirical results, we also derive practical implications for researchers and developers to facilitate the advancement and application of LRM code generation.

The contributions of this paper are as follows:

- We develop a comprehensive taxonomy of 15 reasoning actions across 4 phases in LRM-based code generation.
- We conduct an empirical study that uncovers common reasoning patterns, highlights differences in reasoning behaviors across models, analyzes the impact of reasoning behaviors on functional correctness, and evaluates the feasibility of lightweight prompting improvements.
- We release a dataset of 1,150 annotated reasoning traces to support future research on the reasoning capabilities of LRMs.

2 Study Setup

Figure 2 presents an overview of the methodology employed in this study. We begin by prompting five selected large reasoning models (LRMs) to generate reasoning traces on 230 code generation tasks. Two human annotators then apply open coding methodology on the collected $230 \times 5 = 1,150$ reasoning traces to manually construct a taxonomy of reasoning behaviors exhibited by LRMs, resulting in 15 distinct reasoning actions organized across 4 phases.

Building on this taxonomy, we conduct an empirical study to address a series of research questions. First, we examine the common patterns of action combinations and analyze the rationales underlying these reasoning patterns (**RQ1**). Next, we compare different LRMs in terms of the reasoning actions and patterns they employ, highlighting similarities and discrepancies (**RQ2**). We then investigate how reasoning behaviors influence the functional correctness of generated code and distill key insights (**RQ3**). Finally, we explore the feasibility of lightweight prompting-based strategies for improving LRMs and share the lessons learned (**RQ4**). The research questions are listed as follows:

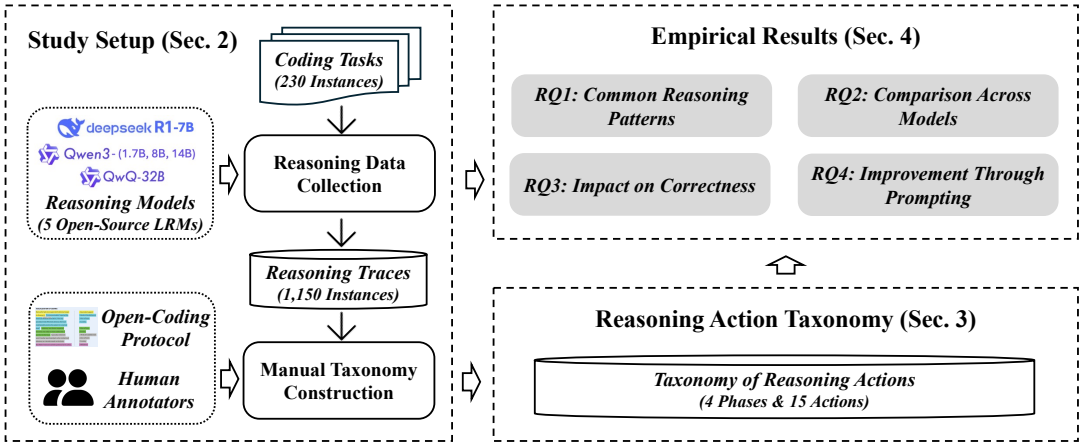

Figure 2: Methodology overview of our study.

- **RQ1 (Common Reasoning Patterns)**: How do LRMs combine and perform individual reasoning actions during code generation?
- **RQ2 (Comparison Across Models)**: To what extent do different LRMs exhibit similar or divergent reasoning behaviors?
- **RQ3 (Impact on Correctness)**: How do the identified reasoning behaviors affect the functional correctness of the generated code, and what key insights can be drawn from the analysis?
- **RQ4 (Improvement Through Prompting)**: How can prompting strategies informed by these behaviors enhance the effectiveness of LRMs?

2.1 LRM Selection

We select target reasoning models according to the following criteria: First, their internal reasoning traces must be accessible. This excludes certain commercial models, such as OpenAI's o-series, which do not provide public API access to such traces. Second, the selected models should span a range of different sizes. Based on these criteria, we choose the following open-source models.

- **DeepSeek-R1-7B** [11]: A large reasoning model developed by DeepSeek AI based on the Qwen2.5-Math-7B model. Despite its relatively small size, it demonstrates competitive performance comparable to that of larger models.
- **Qwen3-(1.7B, 8B, 14B)** [38]: A family of reasoning-focused models developed by Alibaba's Qwen team, offering significant improvements over prior versions such as Qwen2.5 and QwQ. This study employs the 1.7B, 8B, and 14B variants to represent lightweight, small, and medium-large sizes, respectively.
- **QwQ-32B** [35]: A 32-billion-parameter reasoning model developed by Alibaba's Qwen team, trained using supervised fine-tuning followed by reinforcement learning. QwQ demonstrates strong reasoning capabilities, achieving performance on par with state-of-the-art models.

To better reflect real-world usage and explore the model's natural reasoning patterns, we adopt the default configuration (temperature is 0.6) as recommended by model vendors [12] and typically used by end users.

2.2 Reasoning Data Collection

To collect reasoning data, we need to determine the code generation dataset on which to run the LRMs. We apply the following criteria for dataset selection: First, it should be constructed from realistic code repositories and widely adopted in prior studies; Second, it should cover code generation tasks that involve both standalone and non-standalone settings; Third, it should categorize tasks into different levels of difficulty. Based on these criteria, we selected CoderEval [39] from a pool of widely used benchmarks such as HumanEval [7], MBPP [2], ClassEval [13], and DevEval [25].

CoderEval is specifically designed to emulate real-world code generation challenges and is a comprehensive benchmark that simultaneously incorporates dependency-aware tasks and multi-level difficulty settings. It comprises 230 tasks, each for Python and Java, ranging from *self-contained* snippets to *project-runnable* tasks requiring cross-file dependencies:

- ***self-contained***: Fully isolated tasks that can be executed without any dependencies.
- ***slib-runnable***: Tasks requiring only standard libraries, with no need for additional packages.
- ***plib-runnable***: Tasks that depend on third-party libraries that must be installed.
- ***class-runnable***: Tasks with dependencies outside the function but within the same class.
- ***file-runnable***: Tasks with dependencies outside the class but within the same source file.
- ***project-runnable***: Tasks relying on code from other files within the same project.

Although CoderEval provides both Python and Java subsets, the Java subset suffers from an imbalanced distribution of dependency levels—for example, the *class-runnable* and *file-runnable* categories contain only 1 and 5 tasks, respectively. Such sparsity limits the reliability of reasoning behavior comparisons across categories. Moreover, a small-scale pilot study revealed no substantial differences in reasoning behaviors between Java and Python. Given these

factors, and our primary interest in analyzing reasoning capabilities, we focus on the Python portion of the dataset.

Each task is presented to selected large reasoning models (LRMs) using the following prompt template that include both docstrings and function signatures. Across 230 tasks processed with the five LRMs, we obtain a total of 1,150 reasoning traces for analysis.

```
You are a Python software engineer.
Generate Python code based on the following function signature and docstring.
Output ONLY the code generated, in python markdown format.
[Function Signature]
[Docstring]
```

2.3 Manual Taxonomy Construction

To analyze the reasoning processes and patterns exhibited by LRMs, we adopt an open coding protocol [23] on the reasoning traces to manually construct a taxonomy of reasoning behaviors. The construction process consists of two phases:

Open-Coding Annotation. We begin by randomly sampling tasks from each dependency level, ensuring statistical validity with a 95% confidence level and a 5% margin of error. This sampling yields 145 tasks and their corresponding 725 reasoning traces from five LRMs. For each reasoning trace, two annotators with more than five years programming experience independently examine the content to identify the underlying behaviors. They assign short, descriptive phrases, termed ***reasoning actions***, to sentences, paragraphs, or sections that represent fine-grained reasoning behaviors. During this process, the annotators use the original prompt (docstring and function signature) as context to accurately interpret the trace. Next, the annotators iteratively group similar codes into reasoning actions, refining the taxonomy through repeated review of both the codes and traces. A reasoning trace may be mapped to multiple categories when multiple reasoning actions are present. Disagreements between annotators are resolved through discussion with an arbitrator until full consensus is reached.

Reliability Validation. To validate the reliability of the taxonomy, the two annotators independently annotate the remaining 425 reasoning traces (corresponding to 85 tasks) using the obtained coding schema. Each trace is reviewed to assign appropriate reasoning actions. If a trace cannot be classified, it is temporarily placed in a *Pending* category. Inter-rater agreement is then measured using Cohen's Kappa [9], yielding a value of 0.7054, which indicates substantial agreement [24]. This demonstrates the robustness of our coding schema and annotation process. Conflicts in annotation are resolved through discussion with the arbitrator. For reasoning traces classified as *Pending*, the arbitrator also assists in identifying the underlying reasoning actions and determining whether new actions should be introduced. Through this process, we added a new reasoning action, *Style Check*, and reassigned all *Pending* traces into the updated taxonomy. The final taxonomy and annotations were reviewed and approved by all participants.

3 Reasoning Behavior Taxonomy

Based on our manual annotations, we develop a taxonomy of 15 reasoning actions across 4 different phases, as illustrated in Figure 3.

Phase 1: Requirements Gathering: In this phase, the LRMs collect and analyze all information necessary to understand the task specified in the user prompt. This involves identifying the task itself, interpreting the task-specific context, and detecting any constraints or conditions that might influence the solution. By performing these steps, the LRMs try to establish a clear understanding of both the functional intents and any additional requirements that will guide subsequent planning and code generation.

- **Task Identification (TSK)**: At the start of the reasoning, the LRMs typically identify or reiterate the task description provided in the prompt.

  E.g., *"... I need to generate Python code based on the given function signature and docstring. ..."*
  — by DeepSeek-R1-7B on Task 62b45df15108cfac7f2109dc

- **Context Understanding (CTX)**: The LRMs examine the task-specific context conveyed through the function signature and docstring in the user prompt, as these elements generally capture the functional intents. In our generation setting, this context primarily includes code construct elements such as variables, parameters, return values, and the basic functional logic described in the signature and docstring. For example, the case below shows DeepSeek-R1-7B recognizing code elements like `self` and `self.messages`, and analyzing the intended functionality of the generation. Moreover, we expect that supplying richer prompt context would lead to a broader set of analyzed contextual elements, such as retrieved functions in retrieval-augmented generation (RAG) settings.

  E.g., *"... The function is called* `status_str` *and it's an instance method because it has* `self` *as the first parameter. The docstring says it should return a string by visiting the sorted* `self.messages` *list. For each element, it adds the prefix and the element. ..."*
  — by DeepSeek-R1-7B on Task 62b45df15108cfac7f2109dc

- **Constraint Identification (CST)**: The LRMs identify limiting factors or constraints applied to inputs, outputs, or processing logic based on the given prompt. These constraints often represent *additional* conditions that define either functional or non-functional requirements. For example, in the case below, the generated function is designed to be called recursively, which raises potential performance concerns that need to be addressed. This action is frequent because many docstrings and function signatures describe inputs or outputs in terms of specific value ranges or types.

  E.g., *"... The docstring says that this function is called recursively to update a partial* `last_applied_manifest` *from a Kubernetes API response. ..."*
  — by DeepSeek-R1-7B on Task 62b869ebb4d922cb0e688cc6

Phase 2: Solution Planning: In this phase, the LRMs rationalize about how to solve the task and begin structuring a solution before generating complete code. This involves recalling relevant programming knowledge (e.g., language features, libraries, or common idioms), constructing the control flow and data flow of the program, and, when necessary, comparing alternative approaches to determine the most suitable strategy. In addition, LRMs may recognize ambiguities or missing details in the prompt and prepare assumptions to resolve them in subsequent steps.

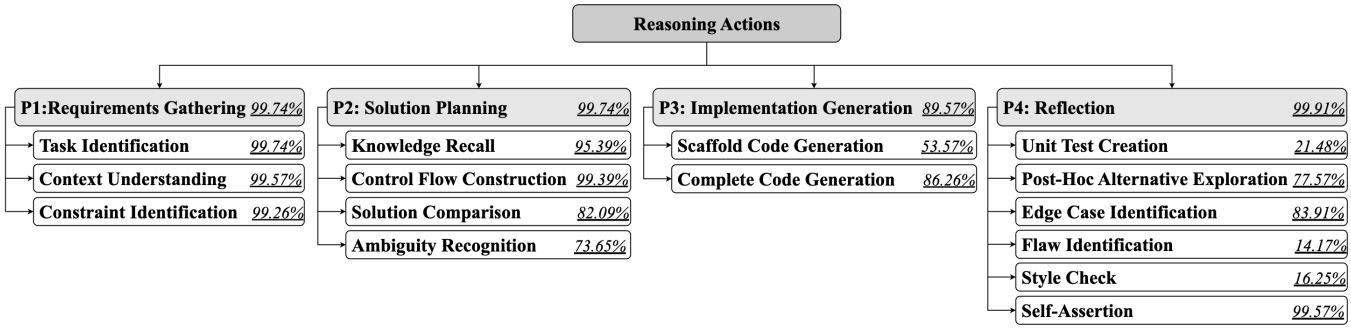

Figure 3: Reasoning action taxonomy in LRM-based code generation.

- **Knowledge Recall (KRL)**: The LRMs recall or leverage prior knowledge about programming languages, libraries, data structures, or common coding patterns to guide subsequent solution steps. For example, in the case below, DeepSeek-R1-7B uses its knowledge of Python's `defaultdict` and its methods to determine the correct import and inform the next steps in the solution planning.

> ***E.g.,*** *"... Hmm, in Python, the* `collections` *module has a* `defaultdict` *that has a* `popitem` *method. That method removes and returns the least frequently used item. So maybe the code should import* `defaultdict` *from* `collections`*. ..."*
>
> *— by DeepSeek-R1-7B on Task 62b8d23748ba5a41d1c3f497*

- **Control Flow Construction (CFL)**: The LRMs design the basic logical structure of the program, primarily including control flow and data flow, based on the task requirements and the context identified in the previous phase. This involves determining the sequence of operations, branching logic, and data handling steps. In the example below, DeepSeek-R1-7B constructs step-by-step logic, including module imports, method definition, and conditional checks, illustrating how it organizes control and data flow to solve the task.

> ***E.g.,*** *"... Then, define the* `validate` *method. Inside the method, I'll use* `os.path.exists` *to check if the path is valid. If it is, return* `True`*; else, return* `False`*. ..."*
>
> *— by DeepSeek-R1-7B on Task 62b45df05108cfac7f2109ce*

- **Solution Comparison (CMP)**: After constructing a preliminary solution structure, the LRMs may compare alternative strategies. They evaluate different approaches in light of the current context and select the one most suitable for the user's request. In the example below, Qwen3-14B compares two similar functions, reasoning about input types and processing logic to determine an appropriate strategy. Note that, in our current setting without a full project context, the final choice among alternatives often relies on assumptions or random selection.

> ***E.g.,*** *"... Another approach: look for similar functions. The other function is* `update_last_applied_manifest_dict_from_resp`*, which probably takes a dict and updates it with the response. So this function, for a list, would iterate over each item in the list and apply the dict function to each item. But the parameters here are lists, so maybe the* `observer_schema` *and response are lists of dicts, and the function processes each element in the list. ..."*
>
> *— by Qwen3-14B on Task 62b869ebb4d922cb0e688cc6*

- **Ambiguity Recognition (AMB)**: The LRMs sometimes detect potential ambiguities or missing information in the prompt or current context. They flag these issues and prepare to resolve them by making "reasonable" assumptions in the subsequent phase. In the example below, Qwen3-14B identifies unclear instructions in the prompt, highlighting the need for assumptions or clarification. Such ambiguities often stem from vague docstrings or lack of awareness of the broader project context, including existing logic elsewhere in the codebase.

> ***E.g.,*** *"... The function signature has 'commands' as a list, and 'args' as another parameter. That's a bit unclear. Maybe the 'commands' is the main command, and 'args' is the list of arguments. Or perhaps 'commands' is the list of commands to execute in sequence. But the docstring says "Run the given command(s) with the given arguments using a subprocess." So maybe it's supposed to run multiple commands, each with its own arguments? ..."*
>
> *— by Qwen3-8B on Task 62ece4992e6aefcf4aabbd83*

Phase 3: Implementation Generation: In this phase, the LRMs translate the solution plan designed in Phase 2 into concrete code implementation. The process typically occurs in stages, from scaffold code to a fully formed candidate function.

- **Scaffold Code Generation (SCG)**: The LRMs sometimes generate partial or pseudo code that outlines the structure and main operations of the solution, without producing the final candidate code. This allows the models to roughly verify logic, identify missing pieces, or iteratively refine the implementation. In the example below, DeepSeek-R1-7B outputs a single expression for joining sorted messages with a prefix, illustrating the structure of the intended function without completing the full definition.

E.g., *"... Like this:*
`return ", ".join(f"{prefix}{msg}" for msg in sorted(self.messages))` *..."*
— by DeepSeek-R1-7B on Task 62b45df15108cfac7f2109dc

- **Complete Code Generation (CCG)**: After scaffolding, the LRMs may produce the complete candidate code that implements the solution. This output includes all necessary components, such as function definitions, control flow, and expressions, ready for review or execution. In the example below, DeepSeek-R1-7B combines the previously scaffolded expression into a full method definition, demonstrating how scaffolded ideas are integrated into a function.

E.g., *"... Putting it all together, the function becomes:*
```
def status_str(self, prefix=""):
    return ", ".join(f"{prefix}{msg}" for msg in sorted(self.messages))
```
..."
— by DeepSeek-R1-7B on Task 62b45df15108cfac7f2109dc

Phase 4: Reflection: In this phase, the LRMs critically evaluate the generated code to ensure it satisfies the task requirements, adheres to coding best practices, and handles possible edge cases. Reflection is both a verification and refinement phase, where the model reviews its own outputs, anticipates potential failures, and improves robustness and clarity. The main goals are correctness, reliability, maintainability, and alignment with the user's implicit or explicit expectations.

- **Unit Test Creation (UTC):** The LRMs generate test cases derived from the prompt, docstrings, or inferred requirements to explicitly verify code correctness. This ensures the generated code produces the expected outputs for typical inputs and helps uncover obvious bugs early. However, since the LRMs themselves perform the test execution and validation, the results may not fully guarantee reliability. In the example below, DeepSeek-R1-7B designs a test for a valid IP address to confirm that the function correctly identifies it as valid, illustrating how test cases are tied directly to the prompt requirements.

E.g., *"... Let me think about some test cases. For example:* `"192.168.0.1"` *should return* `True`. *..."*
— by DeepSeek-R1-7B on Task 62ece4992e6aefcf4aabbd84

- **Post-Hoc Alternative Exploration (ALT):** The LRMs evaluate alternative strategies that differ from the initially generated code. This helps ensure that no potentially better approach is overlooked, and allows comparison against different libraries, algorithms, or coding patterns. In the example below, QWQ-32B considers multiple libraries for XML processing before settling on the correct `xmlsec` implementation, illustrating how the model iteratively explores alternatives for robustness. The difference from Solution Comparison (CMP) is that ALT reflects whether alternative implementations are considered after generating code, whereas CMP focuses on comparing potential solutions before committing to a concrete implementation.

E.g., *"... In reality, XML signatures can be more complex because they might sign specific parts of the document, use references, etc. Alternatively, maybe the function is using the* `xmlsec` *library, which is a Python binding for XML Security. Oh right,* `xmlsig` *module might be part of that. ..."*
— by QWQ-32B on Task 630629d052e177c0ba46a0a1

- **Edge Case Identification (EGC):** The LRMs proactively reason about uncommon or extreme input scenarios that could trigger failures. This aims to strengthen code robustness by ensuring proper handling of unexpected or incomplete data. In the example below, Qwen3-1.7B identifies missing dictionary keys and empty lists, demonstrating attention to possible input anomalies and safe handling of these cases.

E.g., *"... Now, considering possible edge cases: what if the fixity is empty? Or if the* `'files'` *key is missing? ..."*
— by Qwen3-1.7B on Task 62b45e175108cfac7f210a19

- **Flaw Identification (FLW):** The LRMs examine the code for logical or semantic errors and revise it to correct mistakes. This step ensures that the solution behaves as intended across all valid inputs. In the example below, QWQ-32B identifies that using `None` would incorrectly filter triples and revises the logic to account for all object nodes, illustrating careful reasoning about functional correctness.

E.g., *"... But wait, the triple is (node, prop, parent). So, the object of the triple is the parent. So, if there's any such triple, then the node has a parent. In the code above, the triple is (node, prop, None), which would return all triples where the subject is node, the predicate is prop, and the object is None. But that's not correct. Because the object can be any node. ..."*
— by QWQ-32B on Task 630629d052e177c0ba46a0a1

- **Style Check (STY):** The LRMs evaluate whether the code adheres to coding conventions, formatting standards, and documentation best practices. Proper style ensures readability, maintainability, and consistency with user expectations. In the example below, DeepSeek-R1-7B confirms that the code is properly indented, syntactically correct, and well-documented, highlighting the model's attention to presentation and clarity.

E.g., *"... I'll make sure the code is properly indented and follows Python syntax. ..."*
— by DeepSeek-R1-7B Task ID: 62b8d22f48ba5a41d1c3f488

- **Self-Assertion (SFA):** The LRMs provide final closure by asserting that the generated code fulfills the task requirements. This step signals confidence in the correctness and completeness of the solution. In the example below, Qwen3-8B reaffirms that the code correctly initializes the `_Converter` instance and satisfies the intended function, demonstrating the model's self-assertion process.

E.g., *"... I think that's a reasonable approach. So the final code would be as above. But since the user hasn't provided specifics, this is an assumption. However, given the problem statement, this seems like a logical implementation. ..."*
— by Qwen3-8B on Task 62b45df05108cfac7f2109ce

Finding 1: We identified 15 reasoning actions across four phases: Requirements Gathering, Solution Planning, Implementation Generation, and Reflection. Nearly all reasoning traces include all four phases, with only a small portion (10%) omitting Implementation Generation. Among the 15 actions, most appear frequently in the reasoning process, while Scaffold Code Generation (SCG), Unit Test Creation (UTC), Flaw Identification (FLW), and Style Check (STY) occur relatively less often (< 50%).

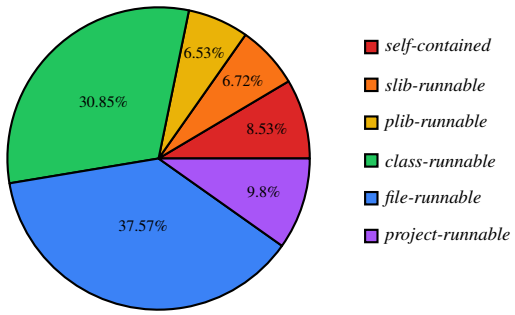

Figure 4: Distribution of the top 5 common patterns across different dependency levels.

4 Empirical Results

Based on the taxonomy, we conduct a series of experiments and present empirical results to address the research questions.

4.1 RQ1: Common Reasoning Patterns

We count the reasoning actions in the traces and analyze the common patterns exhibited by the LRMs. A reasoning pattern is defined as the complete combination of individual reasoning actions that appear within a single trace.

Overall Results. Table 1 presents the top five most common reasoning patterns observed in reasoning traces. The most frequent pattern, FP1, accounts for approximately 17% of all traces. This aligns with the action frequencies, as all actions within FP1 appear in more than half (> 50%) of the collected traces. This frequent pattern generally reflects a human-like coding process, encompassing requirement analysis from multiple perspectives, clarification of ambiguities, comparison of alternative solutions, code implementation, and subsequent review to identify potential defects. Frequent patterns FP3, FP4, and FP5 are extensions of FP1, incorporating UTC, FLW, or STY, respectively. These variations typically depend on the specific challenges faced by the LRMs: for example, when validation or verification is required, the model performs Unit Test Creation (UTC) to clarify logic; when inconsistencies are noticed, it performs Flaw Identification (FLW); and when readability or conventions are at stake, it enforces code Style Checks (STY). While FP2 is a simplified version of FP1, where the LRM perceived that it is not necessary to perform the intermediary step of Scaffold Code Generation (SCG) and directly work on the final code.

The notable outlier among the top five is FP5. Unlike the others, FP5 omits Implementation Generation and Reflection. In this pattern, after performing Requirements Gathering (P1) and Solution Planning (P2), the LRMs directly output generated code as the final answer without additional reasoning or review on implementation. This behavior tends to occur in straightforward tasks where the solution is clear and no further analysis is deemed necessary.

Breakdown Results. Figure 4 shows how the top five patterns are distributed across different dependency levels. These patterns appear most often in complex levels such as *class-runnable*, *file-runnable*, and *project-runnable*, where solving the tasks require a broader context. This suggests that LRMs tend to fall back on recurring reasoning routines when dealing with complex or underspecified problems. While providing stability, this also introduces extra reasoning steps and overhead for simpler tasks at the *self-contained*, *slib-runnable*, and *plib-runnable* levels. For those more straightforward cases, LRMs usually apply simpler patterns without Ambiguity Recognition (AMB), since the prompts often already define problems clearly without relying on in-project context.

> **Finding 2:** LRMs most often follow a human-like coding workflow of analyzing requirements, clarifying ambiguities, comparing solutions, implementing code, and reviewing for defects, with variations such as scaffolding, flaw detection, or style checks depending on task difficulty. These patterns appear most frequently in complex dependency levels, while simpler tasks are handled with lighter reasoning that avoids unnecessary steps like ambiguity recognition.

4.2 RQ2: Comparison Across Models

Table 2 presents the top 5 most common reasoning patterns for each LRM. The results indicate that the Qwen3 models exhibit nearly identical dominant patterns, suggesting that models within the same series display highly consistent reasoning behaviors regardless of parameter size. Notably, the highlighted pattern corresponds to FP1 introduced in Section 4.1, which explains its strong prevalence across reasoning traces. Although QWQ-32B places slightly greater emphasis on Style Check (STY), its overall reasoning patterns remain broadly consistent with the Qwen3 models, reflecting its role as their predecessor developed by the same group. By contrast, DeepSeek-R1-7B often deviates from this trend, frequently omitting the Implementation Generation phase in its reasoning trace. Moreover, in our experiments, DeepSeek-R1-7B follows a more *linear, waterfall-like* reasoning style, whereas other models adopt a more *cyclical, iterative* approach. This difference is evident in its lower frequency of Solution Comparison (CMP), Ambiguity Recognition (AMB), Edge Case Identification (EGC), and Alternative Comparison (ALT), which occur in only 15–46% of traces compared to over 80% in the Qwen3 and QWQ-32B models' traces. Such divergence underscores the strong influence of training data, as Qwen-family models and DeepSeek models were trained on different reasoning traces, which in turn shaped their distinct reasoning styles.

> **Finding 3:** LRMs in the Qwen3 series exhibit highly similar reasoning behaviors across parameter sizes, generally following an *iterative, cyclical* style. In contrast, DeepSeek-R1-7B adopts a more *linear, waterfall-like* reasoning pattern, reflecting a distinct approach that may stem from differences in the reasoning traces used during training.

4.3 RQ3: Impact on Correctness

To assess the impact of reasoning behaviors on functional correctness, we analyze the statistical correlation between the Pass@1 metric and the observed reasoning actions and patterns. Table 3 reports Pass@1 results for each LRM across different dependency levels. The results for Qwen3 models with and without reasoning indicate that enabling reasoning has minimal impact on correctness. Consequently, our analysis focuses on the characteristics of reasoning behaviors rather than the binary setting of reasoning enabled

Table 1: Top-5 common reasoning action patterns across all reasoning traces. The most frequent pattern, along with its occurrences in other patterns, is highlighted.

Action Pattern	Frequency # (%)
FP1: TSK, CTX, CST, KRL, CFL, CMP, AMB, SCG, CCG, ALT, EGC, SFA	207 (17.48%)
FP2: TSK, CTX, CST, KRL, CFL, CMP, AMB, CCG, ALT, EGC, SFA	188 (16.35%)
FP3: TSK, CTX, CST, KRL, CFL, CMP, AMB, SCG, CCG, UTC, ALT, EGC, SFA	57 (4.96%)
FP4: TSK, CTX, CST, KRL, CFL, CMP, AMB, SCG, CCG, ALT, EGC, FLW, SFA	53 (5.61%)
FP5: TSK, CTX, CST, KRL, CFL, CMP, AMB, SCG, CCG, ALT, EGC, STY, SFA	52 (4.52%)
Others	599 (52.09%)

Table 2: Top 5 common reasoning patterns across LRMs. The most frequent pattern (FP1 in RQ1) shared among LRMs is highlighted.

DeepSeek-R1-7B	Qwen3-1.7B	Qwen3-8B	Qwen3-14B	QWQ-32B
TSK, CTX, CST, KRL, CFL, SFA	TSK, CTX, CST, KRL, CFL, CMP, AMB, CCG, ALT, EGC, SFA	TSK, CTX, CST, KRL, CFL, CMP, AMB, SCG, CCG, ALT, EGC, SFA	TSK, CTX, CST, KRL, CFL, CMP, AMB, SCG, CCG, ALT, EGC, SFA	TSK, CTX, CST, KRL, CFL, CMP, AMB, SCG, CCG, ALT, EGC, SFA
TSK, CTX, CST, KRL, CFL, STY, SFA	TSK, CTX, CST, KRL, CFL, CMP, AMB, SCG, CCG, ALT, EGC, SFA	TSK, CTX, CST, KRL, CFL, CMP, AMB, CCG, ALT, EGC, SFA	TSK, CTX, CST, KRL, CFL, CMP, AMB, CCG, ALT, EGC, SFA	TSK, CTX, CST, KRL, CFL, CMP, AMB, SCG, CCG, ALT, EGC, STY, SFA
TSK, CTX, CST, KRL, CFL, CMP, AMB, SCG, CCG, ALT, EGC, SFA	TSK, CTX, CST, KRL, CFL, CMP, AMB, SCG, CCG, ALT, EGC, FLW, SFA	TSK, CTX, CST, KRL, CFL, CMP, AMB, SCG, CCG, ALT, EGC, FLW, SFA	TSK, CTX, CST, KRL, CFL, CMP, AMB, SCG, CCG, ALT, EGC, FLW, SFA	TSK, CTX, CST, KRL, CFL, CMP, AMB, SCG, CCG, UTC, ALT, EGC, SFA
TSK, CTX, CST, KRL, CFL, CMP, SFA	TSK, CTX, CST, KRL, CFL, CMP, AMB, CCG, ALT, EGC, STY, SFA	TSK, CTX, CST, KRL, CFL, CMP, AMB, SCG, CCG, ALT, EGC, STY, SFA	TSK, CTX, CST, KRL, CFL, CMP, AMB, SCG, CCG, UTC, ALT, EGC, SFA	TSK, CTX, CST, KRL, CFL, CMP, AMB, CCG, ALT, EGC, SFA
TSK, CTX, CST, KRL, CFL, SCG, SFA	TSK, CTX, CST, KRL, CFL, CMP, AMB, SCG, CCG, ALT, EGC, STY, SFA	TSK, CTX, CST, KRL, CFL, CMP, AMB, CCG, ALT, EGC, FLW, SFA	TSK, CTX, CST, KRL, CFL, CMP, AMB, CCG, EGC, SFA	TSK, CTX, CST, KRL, CFL, CMP, AMB, SCG, CCG, UTC, ALT, EGC, STY, SFA

versus disabled. For DeepSeek-R1-7B and QWQ-32B, which do not support disabling reasoning, only results with reasoning (*w/ R*) are reported.

4.3.1 Individual Reasoning Actions. We employ the phi-coefficient (φ) [10] to quantify the correlation between each reasoning action and the correctness of the generated code. A positive φ indicates that the presence of an action correlates with higher correctness, a negative φ suggests the action is associated with lower correctness, and values near zero imply little to no correlation.

Figure 5 presents the correlations between each reasoning action and the correctness of the generated code. Among all actions, Unit Test Creation (UTC) shows the strongest influence, with a weak positive correlation to correctness. This indicates that generating unit tests helps LRMs validate their own logic, promoting more thorough reasoning and increasing the chance of producing correct outputs. By contrast, Constraint Identification (CST), Ambiguity Recognition (AMB), and Solution Comparison (CMP) exhibit weak negative correlations. These actions often appear when prompts and contexts are unclear or under-specified, leading LRMs to make additional assumptions. Such assumptions may be incorrect, while overly rigid constraints can prematurely narrow the solution space, and repeated ambiguity recognition or comparison may lead to reasoning loops and inefficiency. As a result, these behaviors increase cognitive load without reliably improving—and sometimes even reducing—the correctness of the final solution.

4.3.2 Combined Reasoning Actions. We further examine whether specific combinations of reasoning actions influence the correctness of the generated code. To this end, we apply the Apriori algorithm [19], a classic association rule mining method commonly used to discover frequent itemsets and their correlations within large datasets. By treating reasoning actions as items and reasoning traces as transactions, Apriori allows us to identify combinations of actions that are strongly associated with successful code generation outcomes. It is worth noting that the mined combinations may be subsequences of the complete reasoning patterns identified in RQ1.

Overall Results. Using this approach, we mine several action combinations that are linked to generated code successfully passing all test cases for a given task. Table 4 reports the top 10 such combinations, where the ratio indicates the proportion of conversions resulting in correct solutions. The presence and co-occurrence of Unit Test Creation (UTC), Self-Assertion (SFA), and Task Identification (TSK) are positively correlated with higher success rates. Consistent with the findings in the previous section, UTC emerges as the strongest individual predictor of correctness, while its combination with TSK and SFA achieves the highest conversion ratio from generation attempts to correct outputs. This suggests that by having a deeper understanding of the problem, LRMs can construct more comprehensive unit tests, which improves their ability to self-assert solutions and increases the likelihood of passing test cases.

Breakdown Results. Breaking down the results by task dependency level, Table 5 presents the top-10 combinations mined by the Apriori algorithm that lead to passing all test cases. Due to the sparsity of pass results on higher complexity level tasks, no common combinations emerge even with a minimal confidence threshold (0.02) at the *project-runnable* level tasks, alongside a significant drop in confidence level for *class-runnable* and *file-runnable* tasks of around $0.2 - 0.3$ compared to the lower complexity tasks with a

Table 3: Pass@1 scores on code generation tasks (%). *w/ R* and *w/o R* denote results with and without reasoning enabled, respectively. Since DeepSeek-R1-7B and QWQ-32B do not support disabling reasoning, only *w/ R* results are reported for these models.

Category	DeepSeek-R1-7B	Qwen3-1.7B		Qwen3-8B		Qwen3-14B		QWQ-32B
	w/ R	*w/ R*	*w/o R*	*w/ R*	*w/o R*	*w/ R*	*w/o R*	*w/ R*
self-contained	28.57	40.00	42.86	51.43	48.57	51.43	48.57	54.29
slib-runnable	25.00	39.29	35.71	50.00	50.00	50.00	53.57	46.43
plib-runnable	19.05	19.05	23.81	28.57	28.57	28.57	28.57	38.10
class-runnable	10.91	12.73	14.55	23.64	14.55	23.64	21.82	18.52
file-runnable	10.29	17.65	23.53	25.00	23.53	25.00	26.47	19.12
project-runnable	4.35	4.35	13.04	4.35	8.70	13.04	4.35	4.35
Overall	15.22	21.30	24.78	29.57	27.39	30.87	30.00	27.83

Figure 5: Correlation between reasoning actions and code correctness.

Table 4: Top 10 mined common action combinations leading to passing tests.

No	Action Combinations
1	TSK, UTC, SFA
2	TSK, UTC
3	TSK, CTX, UTC, SFA
4	TSK, CFL, UTC, SFA
5	TSK, CTX, UTC
6	TSK, CFL, UTC
7	TSK, CTX, CFL, UTC, SFA
8	TSK, CTX, CFL, UTC
9	TSK, UTC, EGC, SFA
10	TSK, KRL, UTC, SFA

confidence value of > 0.5. Consistent with the overall results, the overlaps across the five levels highlight Unit Test Creation (UTC) as the most frequent action associated with correct outcomes.

Beyond these shared actions, each dependency level exhibits distinct patterns. There are noticeably fewer diverse actions for *self-contained* tasks. This reflects that such tasks with no dependencies and reduced complexity allow LRMs to focus on identifying tasks, and perform verification through Unit Test Creation (UTC). *Slib-runnable* tasks, which rely on standard libraries, are typically functionally well-defined and thus elicit relatively straightforward reasoning behaviours. With the reliance in standard libraries, LRM would need to perform Context Identification (CTX) to distinguish the need for third-party or standard libraries and perform the least Self-Assertion (SFA), which could indicate that LRM is overly confident in the generated code. In contrast, *plib-runnable* tasks frequently involve Knowledge Recall (KRL) and Alternative Exploration (ALT) with both implementation actions of Scaffold Code Generation (SCG) and Complete Code Generation (CCG), indicating that LRMs tend to explore key components, such as usage patterns of public library APIs and verify the results, compare with solutions using other libraries, before producing a final, correct implementation. *Class-runnable* tasks, which rely on code defined within the same class, have a noticeable lack of Edge Case Generation (EGC), Unit Test Creation (UTC), and Complete Code Generation in (CCG), suggesting that LRMs prioritize compatibility with defined code over robustness and exploratory code generation. Finally, *file-runnable* tasks with dependencies outside of the class require a lot of context not provided within the docstrings, and the LRM also performed fewer Unit Test Creation (UTC) relative to tasks that require lower-level dependencies. However, unlike *class-runnable* tasks, *file-runnable* tasks LRMs would perform Complete Code Generation (CCG) and Edge Case Generation (EGC), suggesting that at this dependency level, LRMs may be less able to produce final code directly and instead perform additional reasoning processes to ensure correctness of the output. The differences across dependency levels demonstrate that LRMs can, to some extent, adapt their reasoning behaviours to produce accurate solutions based on the provided task descriptions (docstrings) and code contexts (function signatures).

Finding 4: LRMs adapt their reasoning strategies based on task dependency levels: simpler, self-contained or standard-library tasks trigger minimal, focused reasoning with unit test creation, while tasks relying on public libraries or external files induce more exploratory and verification-focused behaviors. Tasks with intra-class dependencies prioritize compatibility to existing code context over robustness, whereas tasks with broader file-level dependencies elicit additional reasoning steps, such as edge case handling and complete code generation, to ensure correctness. Overall, LRM reasoning is context-sensitive, adjusting the depth and type of actions according to task complexity and dependency structure.

4.3.3 Key Observations and Insights. We further analyze the reasoning behaviors of LRMs and highlight both the positive and negative aspects of specific patterns.

Preference in library selection. Within the KRL (Knowledge Recall) action, LRMs may propose solutions using libraries that are more efficient than naive approaches. For example:

Table 5: Top 10 mined combinations across dependency types leading to test pass. Pass results are sparse at the project-runnable levels, preventing the emergence of common combinations even under a minimal confidence threshold (0.02). Highlighted are the frequent intersections shared across all combinations.

No	*self-contained*	*slib-runnable*	*plib-runnable*	*class-runnable*	*file-runnable*
1	TSK, UTC, SFA	TSK, CTX, CFL, UTC, SFA	TSK, CTX, CST, KRL, CFL, UTC, SFA	TSK, CTX, CST, KRL, CFL, SFA	TSK, CTX, CST, KRL, CFL, SFA
2	TSK, CTX, UTC, SFA	TSK, CTX, CFL, UTC	TSK, CTX, CST, KRL, CFL, UTC, EGC, SFA	TSK, CTX, CST, KRL, SFA	TSK, CTX, CST, KRL, SFA
3	TSK, CFL, UTC, SFA	TSK, CTX, KRL, CFL, UTC, SFA	TSK, CTX, CST, KRL, CFL, CCG, UTC, SFA	TSK, CTX, KRL, CFL, SFA	TSK, CTX, CST, CFL, SFA
4	TSK, CTX, CFL, UTC, SFA	TSK, CTX, KRL, CFL, UTC	TSK, CTX, CST, KRL, CFL, CCG, UTC, EGC, SFA	TSK, CTX, KRL, SFA	TSK, CTX, CST, SFA
5	TSK, UTC, EGC, SFA	TSK, CTX, CFL, CCG, UTC, SFA	TSK, CTX, CST, KRL, CFL, UTC, ALT, EGC, SFA	TSK, CTX, CST, CFL, SFA	TSK, CTX, CST, KRL, CFL, EGC, SFA
6	TSK, CTX, UTC, EGC, SFA	TSK, CTX, CFL, CCG, UTC	TSK, CTX, CST, KRL, CFL, UTC, ALT, SFA	TSK, CTX, CST, CFL	TSK, CTX, CST, KRL, CFL, CCG, SFA
7	TSK, CFL, UTC, EGC, SFA	TSK, CTX, CFL, UTC, EGC, SFA	TSK, CTX, CST, KRL, CFL, CCG, UTC, ALT, EGC, SFA	TSK, CTX, CST, SFA	TSK, CTX, CST, CFL, EGC, SFA
8	TSK, CTX, CFL, UTC, EGC, SFA	TSK, CTX, CFL, UTC, EGC	TSK, CTX, CST, KRL, CFL, CCG, UTC, ALT, SFA	TSK, CTX, CST	TSK, CTX, CST, CFL, CCG, SFA
9	TSK, CTX, KRL, UTC, SFA	TSK, CTX, KRL, CFL, UTC, EGC	TSK, CTX, CST, KRL, CFL, SCG, CCG, UTC, ALT, EGC, SFA	TSK, CTX, CFL, SFA	TSK, CTX, CST, KRL, CFL, CCG, EGC, SFA
10	TSK, CTX, CCG, UTC, SFA	TSK, CTX, KRL, CFL, UTC, EGC	TSK, CTX, CST, KRL, CFL, SCG, UTC, SFA	TSK, CTX, CFL	TSK, KRL, CFL, CCG, EGC, SFA

E.g., *"But the regex approach might be more efficient, especially if there are many keys. Because the loop approach would scan the entire string for each key, whereas the regex can find all placeholders in one pass. ..."*

— by QWQ-32B on Task 62ece4982e6aefcf4aabbd62

However, this behavior has limitations. LRMs may misidentify the required library due to ambiguities in the docstring or confusion between Python libraries/functions with overlapping names or functionality. Additionally, LRMs may lack up-to-date knowledge of certain libraries, or in some cases may avoid using libraries altogether when none are explicitly specified in the prompts.

Ambiguity recognition and reliability of assumptions. LRMs can detect missing or ambiguous information in prompts, as shown by their high frequency of Ambiguity Recognition (AMB) on *class-runnable* (82.18%) and *file-runnable* (83.24%) tasks. These tasks often lack necessary contextual details in the docstring. However, ambiguity can also harm reasoning quality. As shown in Figure 6, unclear object types in input and return statements caused the LRM to fall into repetitive reasoning cycles without making progress.

Consistency in the implementation phase. The Implementation phase, which includes both scaffold and complete code generation, is the only phase not consistently present in reasoning traces, appearing in only 30% to 55% of cases across all dependency levels.. In some straightforward tasks, the LRMs skips scaffold or concrete code generation and move directly to cthe onclusion or review. Another observation is that the Complete Code Generation (CCG) action does not always align with the final output. Reasoning traces may contain documentation-like explanations that are omitted in the final output, partly because the study's prompt explicitly required outputting only code. This highlights the need for caution when analyzing reasoning traces, since they may not fully correspond to the generated output.

Test case creation and reliability issues. LRMs display software-engineering-like behavior by generating unit tests for their own code without explicit prompting. However, this behavior is inconsistent. LRMs occasionally generate test cases for lower-dependency tasks, such as *self-contained* (40%) and *slib-runnable* (32%) levels, but they rarely do so for higher-dependency tasks, where ambiguity and missing context make test construction more difficult (< 11%). More severely, when test cases are created, their correctness is unreliable. In Figure 7, the LRM generates test cases but fails to compute the correct expected output for test case #3. While it constructs the correct reasoning steps, the final computation is flawed. This aligns with prior work identifying LRMs' struggles with mathematical reasoning under direct prompting [8, 16].

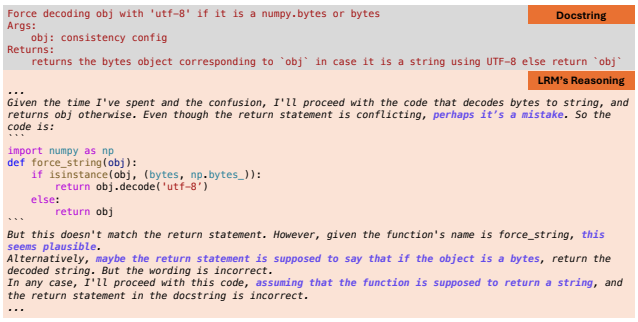

Figure 6: An example of harmful ambiguity recognition: Qwen3-1.7B becomes stuck in a loop of clarifying ambiguities and making assumptions (see highlighted sentences).

Self-assertion and model confidence. LRMs consistently exhibit self-assertion (SFA), affirming that their generated code satisfies the task requirements. However, this does not always reflect genuine confidence. As illustrated in Figure 8, LRMs sometimes recognize flaws during reasoning but, due to constraints such as limited context length or reasoning complexity, still output the flawed code. This mirrors human developers who, under time pressure, may knowingly submit imperfect solutions [3].

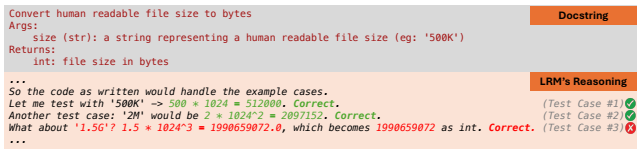

Figure 7: An example of unreliable unit test creation: Qwen3-14B generates three test cases and deems them all passed by its generated code, but test case #3 contains an incorrect expected output.

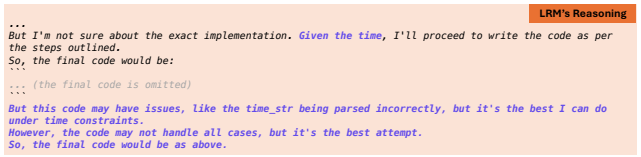

Figure 8: An example of invalid self-assertion: Qwen3-1.7B acknowledges potential issues but proceeds with the output nonetheless (see highlighted sentences).

Finding 5: LRMs demonstrate a range of reasoning behaviors that mirror aspects of human problem solving, including efficient library use, ambiguity recognition, test case creation, and self-assertion. However, these behaviors are inconsistent: library choices may be outdated or incorrect, ambiguity often leads to flawed assumptions, implementation phases can be skipped or misaligned with outputs, and generated test cases are frequently unreliable. Self-assertion is consistently present but does not guarantee genuine confidence, as LRMs may still output flawed solutions. Together, these findings highlight both the adaptability and the current limitations of LRMs in performing reliable reasoning and producing correct code.

4.4 RQ4: Prompting-based Improvements

We further evaluate the feasibility of two lightweight prompting-based strategies as potential improvements over our initial prompt, motivated by the key findings from the previous section:

- *GUIDE*: Our previous analysis identifies Unit Test Creation (UTC) as the most positively correlated action with test passing. Based on this, we design a modified prompt that explicitly guides LRMs to emulate unit test cases, as shown in Figure 9b.
- *CONT*: Since LRMs often struggle with ambiguity due to insufficient context, we test whether providing additional contextual information can improve performance. The modified prompt is illustrated in Figure 9a.

Table 6 compares the Pass@1 scores of the modified prompts against the original prompt. Overall, both modified prompts yield slight improvements across most LRMs, with the exception of the Qwen3-14B variant on the *CONT* prompting. For more complex tasks with higher dependencies, such as at the *project-runnable* level, incorporating guidelines helps LRMs generate better code, resulting in improved performance. The consistent gains across different LRM series and parameter sizes indicate that integrating reasoning patterns into prompts holds promise as an effective strategy in prompt engineering and merits further investigation. However, due to the intrinsic randomness of LRMs, we cannot conclusively claim that these prompting strategies will always improve performance.

```
You are a Python software engineer.
Generate Python code based on the following function signature and docstring.
Do NOT include any explanation, reasoning, or markdown formatting.
Output ONLY the code generated, in python markdown format.

## Tips
You should follow a test-driven development approach, first generating comprehensive unit tests before writing the actual code.

[Function Signature]
[Human Docstring]
```

(a) *GUIDE* prompt template.

```
You are a Python software engineer.
Generate Python code based on the following function signature and docstring.
Do NOT include any explanation, reasoning, or markdown formatting.
Output ONLY the code generated, in python markdown format.

## Context
Imported Packages: [Package dependencies]
Within file: [File dependencies]
Within class: [Class dependencies]

[Function Signature]
[Human Docstring]
```

(b) *CONT* prompt template.

Figure 9: Prompt templates for the two lightweight prompting methods, with newly added lines highlighted and lines removed are bolded.

Finding 6: Our results highlight the potential of incorporating context or reasoning guidelines into prompts to enhance LRM-generated code. The *GUIDE* prompt variant, which integrates a UTC-enhanced design, shows slight improvements across different LRMs.

5 Discussion

5.1 Implications for Researchers

Reasoning Behavior Visualization. Our study introduces a taxonomy of fine-grained reasoning actions, showing that LRMs exhibit distinct reasoning patterns across different tasks. To deepen understanding of these reasoning traces, future research can develop visualization techniques grounded in our taxonomy and annotated data, thereby helping to open the black box of model reasoning in code generation. Such visualizations can enhance the interpretability of LRMs and foster greater developer trust in their outputs.

Reasoning Capability Enhancement. Our study investigates the feasibility of prompting-based improvement strategies, showing that while these methods yield modest gains, there remains considerable room for advancement. Future research can explore more sophisticated approaches to enhancing reasoning capabilities. For instance, we observe that DeepSeek-R1 tends to adopt a linear, waterfall-style reasoning process, whereas Qwen3 models favor an iterative reasoning process, achieving higher functional correctness. This suggests the potential of generating higher-quality reasoning traces that emulate realistic iterative programming practices for model fine-tuning. Likewise, the strong positive correlation between UTC and test pass rates highlights the promise of incorporating the test-driven development (TDD) paradigm into the LRMs for code generation.

Table 6: Pass@1 score for code generation task (%) with and without prompt modifications.

Category	DeepSeek-R1-7B			Qwen3-1.7B			Qwen3-8B			Qwen3-14B			QWQ-32B 32B		
	Original	CONT	*GUIDE*	Original	CONT	*GUIDE*	Original	CONT	*GUIDE*	Original	CONT	*GUIDE*	Original	CONT	*GUIDE*
self-contained	28.57	37.14	28.57	40.00	40.00	34.29	51.43	42.86	54.29	51.43	48.57	57.14	54.29	51.43	54.29
slib-runnable	25.00	25.00	21.43	39.29	28.57	32.14	50.00	50.00	46.43	50.00	39.29	50.00	46.43	53.57	50.00
plib-runnable	19.05	19.05	19.05	19.05	23.81	19.05	28.57	23.81	33.33	28.57	38.10	33.33	38.1	33.33	38.10
class-runnable	10.91	12.73	14.55	12.73	23.64	20.00	23.64	23.64	21.82	23.64	23.64	25.45	18.52	20.00	16.36
file-runnable	10.29	10.29	10.29	17.65	16.18	19.12	25.00	30.88	22.06	25.00	26.47	25.00	19.12	23.53	22.06
project-runnable	4.35	0.00	4.35	4.35	8.70	8.70	4.35	8.70	13.04	13.04	8.70	13.04	4.35	13.04	17.39
overall	15.22	16.52	15.65	21.30	23.04	22.17	29.57	30.43	30.00	30.87	30.00	32.61	27.83	30.43	30.00

5.2 Implication for Software Developers

Prompt/Context Engineering for LRMs. Since LRMs inherently exhibit CoT-like reasoning, some argue that carefully crafted prompts are unnecessary when using them. However, our study suggests that prompt engineering remains important for effective code generation. When designing prompts, practitioners should provide concise instructions while ensuring that the problem statement and contextual information, such as well-written docstrings, relevant libraries, classes, or attributes, are explicit and complete. Our findings show that incomplete, unclear, or ambiguous prompts can hinder LRMs' reasoning and negatively impact performance. This insight also highlights the emerging role of context engineering, which focuses on supplying precise and comprehensive contextual information tailored to the task requirements.

Reasoning Process Inspection. Our study provides insights into the limitations of LRMs' reasoning in code generation. For instance, LRMs still struggle to reliably perform tasks such as generating tests or maintaining sound assumptions. This indicates that developers should not only carefully review and validate the code produced by LRMs for potential flaws but also examine and verify the models' reasoning traces.

6 Threats to Validity

External Validity. Threats to external validity concern the extent to which our findings generalize across different languages. Due to the imbalanced distribution of dependency scopes in Java, we focused our analysis on reasoning traces in Python. Exploring reasoning traces for other programming languages remains an important direction for future work. Another potential threat is the quality of code generation tasks considered. We mitigated this by using CoderEval, a widely adopted dataset of real-world programming tasks that spans multiple dependency levels.

Internal Validity. Threats to internal validity mainly relate to manual annotation and taxonomy construction. To address this, we applied standard conflict-resolution strategies during taxonomy development to ensure reliability. Additionally, since the model temperature is non-zero, as recommended by the LRM publishers [12], outputs are inherently nondeterministic. In our experiments, setting the temperature to zero led to inconsistent behavior, including timeouts and incoherent repetitive reasoning. To mitigate potential reproducibility concerns, we documented all model responses and outputs in our replication package.

7 Related Work

LLMs in Code Generation. Large language models (LLMs) have been increasingly applied to software engineering tasks, particularly code generation [14, 17, 26, 36, 40, 43]. Existing literature [6, 20, 28, 31] has primarily focused on the GPT series, most notably GPT-4 as the primary benchmark for performance evaluation [18]. Alongside general-purpose models, several LLMs have been explicitly trained for code-related tasks, such as Code Llama [34] and Qwen2.5-Coder [21]. To assess these models, various datasets have been created, such as HumanEval [7], MBPP [2], ClassEval [13], DevEval [25], and CoderEval [39].

LLM Reasoning. With the emergence of large reasoning models (LRMs) and greater transparency enabled by their explicit intermediate reasoning steps, there has been a growing interest in analysing the models' reasoning traces to understand their internal decision-making processes better. Recent studies have sought to categorise the reasoning behaviour of large language models across various domains. Marjanovic et al. [29], inspired by common human reasoning, derive a general taxonomy of DeepSeek-R1 reasoning patterns on a diverse set of problems. In their study, Ming et al. [30] explore and derive a taxonomy on LLM reasoning behavior as a critique and when faced with critique. Plaat et al. [33] conduct a survey and propose a 3-stage taxonomy on LLM reasoning for math problems. Bandyopadhyay et al. [4] derives common phases on how LLM would reason as a pipeline/structure. Several studies have directly addressed reasoning within the context of code generation. Wei et al.[37] investigate the reasoning model's capability in performing code generation on the competitive programming domain and highlight some flaws in the code generated by LLM. Liu et al. [27] in their proposed framework evaluation CodeMind, highlight reasoning model capability and limitation on control flow, and show that code generation ability, despite being correlated, does not imply code reasoning. Prompt variation has also been explored as a tool for influencing reasoning behavior. In their study, Chatziveroglou et al. [6] explored the effect on LLM reasoning through prompt variation on math problems. Mu et al. [31] in ClarifyGPT propose a framework to enhance LLM on code generation through ambiguity detection and clarification.

8 Conclusion

In this paper, we conduct an empirical study of reasoning behaviors in large reasoning models (LRMs) for code generation. We develop a taxonomy of 15 reasoning actions across four phases, reveal common and model-specific reasoning patterns, and show how

these behaviors affect functional correctness. Finally, we demonstrate that lightweight, reasoning-oriented prompting strategies can further improve code generation. These findings advance our understanding of LRM reasoning and provide practical guidance for automated software development.

9 Data Availability

All of the artefacts and LLM output are made publicly available and can be accessed here:
https://github.com/ReasoningPattern/ReasoningPattern

References

[1] Anthropic. 2025. Introducing Claude 4. https://www.anthropic.com/news/claude-4 Accessed July 2025.

[2] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021).

[3] Robert D Austin. 2001. The effects of time pressure on quality in software development: An agency model. *Information systems research* 12, 2 (2001), 195–207.

[4] Dibyanayan Bandyopadhyay, Soham Bhattacharjee, and Asif Ekbal. 2025. Thinking machines: A survey of llm based reasoning strategies. *arXiv preprint arXiv:2503.10814* (2025).

[5] Maciej Besta, Julia Barth, Eric Schreiber, Ales Kubicek, Afonso Catarino, Robert Gerstenberger, Piotr Nyczyk, Patrick Iff, Yueling Li, Sam Houliston, Tomasz Sternal, Marcin Copik, Grzegorz Kwaśniewski, Jürgen Müller, Łukasz Flis, Hannes Eberhard, Zixuan Chen, Hubert Niewiadomski, and Torsten Hoefler. 2025. Reasoning Language Models: A Blueprint. doi:10.48550/arXiv.2501.11223 arXiv:2501.11223 [cs].

[6] Giannis Chatziveroglou, Richard Yun, and Maura Kelleher. 2025. Exploring llm reasoning through controlled prompt variations. *arXiv preprint arXiv:2504.02111* (2025).

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

[8] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168* (2021).

[9] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and psychological measurement* 20, 1 (1960), 37–46.

[10] Harald Cramér. 1999. *Mathematical methods of statistics*. Vol. 9. Princeton university press.

[11] DeepSeek-AI. 2025. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. arXiv:2501.12948 [cs.CL] https://arxiv.org/abs/2501.12948

[12] DeepSeek-AI. 2025. DeepSeek-R1 Model on Hugging Face. https://huggingface.co/deepseek-ai/DeepSeek-R1 Accessed July 2025.

[13] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2023. Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation. *arXiv preprint arXiv:2308.01861* (2023).

[14] Sarah Fakhoury, Aaditya Naik, Georgios Sakkas, Saikat Chakraborty, and Shuvendu K Lahiri. 2024. Llm-based test-driven interactive code generation: User study and empirical evaluation. *IEEE Transactions on Software Engineering* (2024).

[15] Nat Friedman. 2022. Introducing github copilot: Your AI pair programmer. https://github.blog/news-insights/product-news/introducing-github-copilot-ai-pair-programmer/

[16] Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. Pal: Program-aided language models. In *International Conference on Machine Learning*. PMLR, 10764–10799.

[17] Qiuhan Gu. 2023. Llm-based code generation method for golang compiler testing. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2201–2203.

[18] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large Language Models for Software Engineering: A Systematic Literature Review. arXiv:2308.10620 [cs.SE] https://arxiv.org/abs/2308.10620

[19] Liusheng Huang, Huaping Chen, Xun Wang, and Guoliang Chen. 2000. A fast algorithm for mining association rules. *Journal of Computer Science and Technology* 15, 6 (2000), 619–624.

[20] Tao Huang, Zhihong Sun, Zhi Jin, Ge Li, and Chen Lyu. 2024. Knowledge-aware code generation with large language models. In *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*. 52–63.

[21] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024. Qwen2.5-Coder Technical Report. arXiv:2409.12186 [cs.CL] https://arxiv.org/abs/2409.12186

[22] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. 2023. Survey of hallucination in natural language generation. *ACM computing surveys* 55, 12 (2023), 1–38.

[23] Shahedul Huq Khandkar. 2009. Open coding. *University of Calgary* 23, 2009 (2009), 2009.

[24] J Richard Landis and Gary G Koch. 1977. The measurement of observer agreement for categorical data. *biometrics* (1977), 159–174.

[25] Jia Li, Ge Li, Yunfei Zhao, Yongmin Li, Huanyu Liu, Hao Zhu, Lecheng Wang, Kaibo Liu, Zheng Fang, Lanshen Wang, et al. 2024. Deveval: A manually-annotated code generation benchmark aligned with real-world code repositories. *arXiv preprint arXiv:2405.19856* (2024).

[26] Jia Li, Yunfei Zhao, Yongmin Li, Ge Li, and Zhi Jin. 2023. Acecoder: Utilizing existing code to enhance code generation. *arXiv preprint arXiv:2303.17780* (2023).

[27] Changshu Liu, Yang Chen, and Reyhaneh Jabbarvand. 2024. CodeMind: Evaluating Large Language Models for Code Reasoning. *arXiv preprint arXiv:2402.09664* (2024).

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

[29] Sara Vera Marjanović, Arkil Patel, Vaibhav Adlakha, Milad Aghajohari, Parishad BehnamGhader, Mehar Bhatia, Aditi Khandelwal, Austin Kraft, Benno Krojer, Xing Han Lù, et al. 2025. DeepSeek-R1 Thoughtology: Let's think about LLM Reasoning. *arXiv preprint arXiv:2504.07128* (2025).

[30] Yifei Ming, Zixuan Ke, Xuan-Phi Nguyen, Jiayu Wang, and Shafiq Joty. 2025. Helpful Agent Meets Deceptive Judge: Understanding Vulnerabilities in Agentic Workflows. *arXiv preprint arXiv:2506.03332* (2025).

[31] Fangwen Mu, Lin Shi, Song Wang, Zhuohao Yu, Binquan Zhang, ChenXue Wang, Shichao Liu, and Qing Wang. 2024. Clarifygpt: A framework for enhancing llm-based code generation via requirements clarification. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 2332–2354.

[32] OpenAI. 2025. Introducing OpenAI o3 and o4-mini. https://openai.com/index/introducing-o3-and-o4-mini/ Accessed July 2025.

[33] 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* (2024).

[34] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).

[35] Qwen Team. 2025. QwQ-32B: Embracing the Power of Reinforcement Learning. https://qwenlm.github.io/blog/qwq-32b/

[36] Weixi Tong and Tianyi Zhang. 2024. CodeJudge: Evaluating Code Generation with Large Language Models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (Eds.). Association for Computational Linguistics, Miami, Florida, USA, 20032–20051. doi:10.18653/v1/2024.emnlp-main.1118

[37] Minnan Wei, Ziming Li, Xiang Chen, Menglin Zheng, Ziyan Qu, Cheng Yu, Siyu Chen, and Xiaolin Ju. 2025. Evaluating and Improving Large Language Models for Competitive Program Generation. *arXiv preprint arXiv:2506.22954* (2025).

[38] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. 2025. Qwen3 Technical Report. *arXiv preprint arXiv:2505.09388* (2025).

[39] Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. 2024. Codereval: A benchmark of pragmatic code generation with generative pre-trained models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–12.

[40] Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. 2024. CodeAgent: Enhancing Code Generation with Tool-Integrated Agent Systems for Real-World Repo-level Coding Challenges. arXiv:2401.07339 [cs.SE] https://arxiv.org/abs/2401.07339

[41] Ziyao Zhang, Chong Wang, Yanlin Wang, Ensheng Shi, Yuchi Ma, Wanjun Zhong, Jiachi Chen, Mingzhi Mao, and Zibin Zheng. 2025. Llm hallucinations in practical code generation: Phenomena, mechanism, and mitigation. *Proceedings of the ACM on Software Engineering* 2, ISSTA (2025), 481–503.

[42] Haiyan Zhao, Hanjie Chen, Fan Yang, Ninghao Liu, Huiqi Deng, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, and Mengnan Du. 2024. Explainability for large language models: A survey. *ACM Transactions on Intelligent Systems and Technology* 15, 2 (2024), 1–38.

[43] Tianyu Zheng, Ge Zhang, Tianhao Shen, Xueling Liu, Bill Yuchen Lin, Jie Fu, Wenhu Chen, and Xiang Yue. 2024. Opencodeinterpreter: Integrating code generation with execution and refinement. *arXiv preprint arXiv:2402.14658* (2024).

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009