SWE-InfraBench: Evaluating Language Models on Cloud Infrastructure Code

Natalia Tarasova **Enrique Balp-Straffon** **Aleksei Iancheruk** **Yevhenii Sielskyi**
Nikita Kozodoi **Liam H. Byrne** **Jack Butler** **Dayuan Jiang** **Marcin Czelej**
Andrew Ang **Yash Shah** **Roi Blanco** **Sergei Ivanov**

Amazon Web Services

Abstract

Building infrastructure-as-code (IaC) in cloud computing is a critical task, underpinning the reliability, scalability, and security of modern software systems. Despite the remarkable progress of large language models (LLMs) in software engineering – demonstrated across many dedicated benchmarks – their capabilities in developing IaC remain underexplored. Unlike existing IaC benchmarks that predominantly center on declarative paradigms such as Terraform and involve generating entire codebases from scratch, our benchmark reflects the incremental code edits common in enterprise development with imperative tools like the AWS CDK. We present SWE-InfraBench, a diverse evaluation dataset sourced from dozens of real-world IaC codebases that challenge LLMs to perform realistic code modifications in AWS CDK repositories. Each example requires models to implement changes to existing codebases based on natural language instructions, with success determined by passing provided test cases. These tasks demand sophisticated reasoning about cloud resource dependencies and implementation patterns beyond conventional code generation challenges. Our evaluation results reveal significant limitations in current LLMs showing that even state-of-the-art systems struggle with many tasks – the best model, Sonnet 3.7, succeeds in only 34% of cases, while specialized reasoning models like DeepSeek R1 achieve just 24% success.

1 Introduction

Modern cloud computing platforms enable unprecedented scalability and automation, but realizing its potential requires managing highly complex infrastructure configurations [8; 12]. Infrastructure as Code (IaC) has emerged as a DevOps practice to meet this need, treating the specification of cloud resources as software artifacts [32; 27; 25]. With IaC, cloud environments (servers, networks, etc.) are defined and provisioned using code, bringing software engineering rigor (version control, code review, continuous integration) to infrastructure management. In broad terms, IaC approaches fall into two paradigms: declarative and imperative. A declarative IaC tool describes the desired end state of the infrastructure, leaving it to the system to "figure out" the necessary actions. An imperative IaC tool, in contrast, specifies the exact sequence of commands to reach that end state [6; 23; 36].

Two dominant frameworks exemplify these paradigms in today's cloud IaC landscape: Terraform and AWS Cloud Development Kit (CDK). Terraform is a widely-adopted declarative IaC tool that uses a domain-specific language (HCL) to let operators specify the final state of infrastructure resources [7]. While this approach has strengths (platform-agnostic abstractions, a robust planning mechanism), it also has notable limitations for complex, evolving systems. Purely declarative configurations can become brittle – small changes may require non-intuitive refactoring or state file surgery – and

39th Conference on Neural Information Processing Systems (NeurIPS 2025). Workshop: Evaluating the Evolving LLM Lifecycle

Figure 1: Each SWE-InfraBench task includes an IaC repository, natural language modification instructions, and automated tests verifying the correctness and integration of the generated solution.

inflexible, as HCL lacks the expressive power of general-purpose languages for handling conditional logic, loops, or abstractions.

The AWS CDK directly addresses these pain points by taking an imperative developer-friendly approach [3; 1; 17]. CDK allows developers to define cloud infrastructure using familiar programming languages (TypeScript, Python, etc.), leveraging the full power of those languages to encode loops, reuse components, and incorporate conditionals. This means infrastructure definitions can be developed in the same codebase and environment as the application code, with complete IDE support and testing frameworks [29; 37; 29]. The two philosophies yield different developer experiences: Terraform favors stability but limits agility, while CDK supports incremental change and continuous delivery. Since industrial software development often demands iterative experimentation and close coupling with application logic, we adopt AWS CDK for this benchmark, which enables incremental and testable infrastructure.

Surprisingly despite widespread industrial adoption of IaC frameworks, most of the coding evaluation benchmarks for LLMs focused on programming tasks such as code completion and translation [20; 11; 35; 15; 28], debugging and unit test synthesis [30; 33; 19; 16; 34], but not on IaC generation. Evaluation of LLMs specifically for imperative infrastructure-as-code editing tasks remains a substantially underexplored research area due to unique challenges associated with interpreting and maintaining consistency across large stateful codebases, as well as the inherent difficulty of curating relevant examples that require specialized in-domain knowledge.

To address the gap in IaC evaluation, we introduce SWE-InfraBench, a benchmark dataset designed to assess language models' capabilities on IaC tasks within the AWS CDK framework. SWE-InfraBench comprises 100 diverse examples, each containing complete CDK repositories, modification instructions reflecting real-world development scenarios, and multiple unit tests verifying the correctness of LLM-generated implementations. To the best of our knowledge, SWE-InfraBench is the first specialized benchmark for evaluating language models on CDK editing tasks, distinguishing itself from existing IaC benchmarks [22; 38] in two critical dimensions. First, previous benchmarks primarily target Terraform and its declarative syntax, whereas our benchmark leverages the imperative programming languages used by CDK, a framework widely adopted in industry for infrastructure development. Second, SWE-InfraBench emphasizes incremental codebase modifications rather than complete application synthesis, reflecting the iterative development patterns predominant among cloud practitioners in enterprise environments.

Our work makes the following contributions:

1. We provide a new collection of a hundred instruction-based tasks for modification of IaC codebases that are challenging to modern LLMs, collected from 34 real-world assets. Each task includes a full AWS CDK project with a specific modification instruction and a suite of unit tests that must pass after the change.
2. We introduce a systematic LLM-driven pipeline for generating and validating new benchmark examples from arbitrary CDK repositories. This approach enables anyone to create realistic tasks from any IaC project and ensures each example is valid by verifying that the modified code passes all tests after synthesis.
3. We conduct a comprehensive experimental analysis across diverse LLM architectures, including both proprietary and open-source models with varying parameter scales and architectural designs, establishing baseline performance metrics for this domain. Going

beyond individual LLM models, we evaluate performance of multi-turned agents enhanced with error messages, test results, and RAG on IaC documentation.

2 Related Work

LLMs for Infrastructure-as-Code Generation

IaC has become crucial for modern DevOps and MLOps, enabling programmatic management of cloud resources. Research has explored using LLMs to automate IaC generation. Early work investigated generating Ansible YAML from natural language [21; 31]. However, recent evaluations reveal LLMs perform significantly worse on IaC tasks compared to general Python generation [22], highlighting limitations in domain-specific configuration automation. Benchmarks like CloudEval-YAML [38] and IaC-Eval [22] address this gap, with IaC-Eval showing GPT-4 achieving only 19.36% pass@1 accuracy on human-curated IaC scenarios, despite scoring 86.6% on EvalPlus. These benchmarks focus on declarative frameworks such as Terraform and ask to generate entire codebase from prompt instructions, which is different from our SWE-InfraBench that targets imperative infrastructure code modifications within AWS CDK repositories. Finally, the dataset LIG-MM [24] explores symbolic and neural methods to address the formal reasoning challenges in IaC tasks.

Code Generation and Editing Benchmarks Advancements in LLM code generation evaluation have created more comprehensive benchmarks spanning from single functions to multi-file problems. Meanwhile, research has expanded code editing benchmarks beyond traditional bug fixing to address additional tasks like code completion. HumanEval [10] remains a popular benchmark for evaluating code generation, with recent extensions targeting multilingual [5], editing [39], and reasoning variants [26]. However, these tasks often focus on short, self-contained functions. CrossCodeEval [13] addresses this limitation by introducing multi-file completion problems that require reasoning across files and navigating real codebases – bringing code benchmarks closer to industrial development settings. In order to better simulate real world engineering tasks on identifying location where the code editing is required, SWE-PolyBench, [35] was introduced as an incremental development to the SWE-bench [20] as the first dataset that mirrors real-world software engineering specifically covering over 2000 curated coding challenges across Java, JavaScript, TypeScript and Python along with evaluation metrics like live level localization and concrete Syntax Tree node level retrieval. Similarly, Mercury [14] evaluates models not only on correctness but also on runtime performance, highlighting inefficiencies in LLM-generated code limiting real-world utility. In code editing, the focus has largely been on bug fixing, with fewer benchmarks addressing tasks like code completion. Initiatives like CodeEditorBench [18]and CanItEdit [9] have aimed to fill this gap. CodeEditorBench tried to address this gap, by creating a dataset of tasks to help assess the performance of LLMs in tasks like code debugging, code translation, code requirement switching and even code polishing. But the use of competitive programing problems meant that the real-world replicability of the performance was not tenable. CanItEdit, introduces a dataset of instructional editing problems with a novel ExcessCode metric [18; 9].

3 SWE-InfraBench

SWE-InfraBench consists of a collection of IaC codebases implemented using AWS CDK. Each codebase is paired with a prompt that contains a natural language instruction to generate new code blocks extending the existing infrastructure. For each codebase, we also provide multiple unit tests that enable verification of potential solutions. The task for an LLM is to produce a code change that satisfies the requirements outlined in the prompt and passes the associated tests. On average, one codebase includes 10 files spanning 30 thousand characters that are provided to the LLM as context. Further statistics of the dataset tasks can be found in Appendix A and B.

CDK code is synthesized into AWS CloudFormation [2] templates – JSON or YAML files that describe the desired AWS resources and their relationships. This process enables automated testing of infrastructure definitions without actual deployment, which is particularly valuable since deploying cloud infrastructure can be time-consuming and resource-intensive. This makes CDK a good candidate for benchmarking LLMs on cloud infrastructure generation tasks.

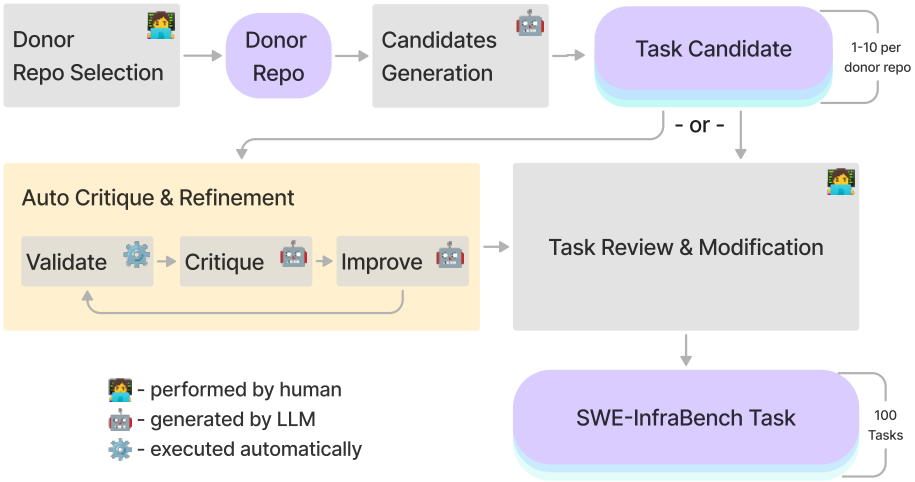

Figure 2: SWE-InfraBench task instances are created from open-source and custom developed IaC repositories in a semi-automated manner with a rigorous human engineers oversight.

3.1 Benchmark Construction

Creating a high-quality IaC dataset requires identifying precise, robust examples with clear objectives and verifiable outcomes. To achieve this at scale and with consistent quality, we developed a multi-stage pipeline combining human expertise with LLMs assistance, which is shown in Figure 2.

Stage I: Donor Repository Selection and Task Generation. We begin by identifying high-quality source repositories implementing IaC. Our benchmark draws from both open-source repositories and custom-developed sources, with licensing details provided in Appendix C. Using these repositories, an LLM (typically Claude Sonnet 3.5 or 3.7) analyzes the codebase and generates initial task suggestions. Each task includes a code block that should be masked (also referred to as the canonical solution), a prompt with functional requirements, and unit tests to verify correctness of generated solutions. Human domain experts then review these suggested tasks and select the most promising candidates.

Stage II: Critique and Refinement. Each task candidate undergoes either manual refinement or an automated improvement process. For automated refinement, we use a critic-based approach:

1. *Validate*: The system validates the tests by running them on the canonical solution (masked code) to ensure they are passed by a correct implementation of the required functionality.

2. *Critique*: Two different critic models (Sonnet 3.7, o3, or o4-mini), which are distinct from the generator, perform a comprehensive evaluation of the task suggestion across four key dimensions:

- *Prompt vs. Functional Requirements Alignment*: Evaluates whether the prompt accurately describes all necessary functional aspects of the masked code, is sufficiently detailed for correct implementation, avoids ambiguities, and balances completeness with avoiding implementation details.
- *Implementation Generality*: Assesses whether the tests allow for multiple valid implementation approaches that satisfy the functional requirements, rather than enforcing a specific implementation pattern.
- *Test Quality*: Analyzes each individual test to determine its purpose, how it relates to prompt requirements, and whether it is appropriately written to verify the requirement without being overly restrictive.
- *Test Completeness*: Verifies that the tests collectively cover all requirements mentioned in the prompt, including edge cases and error conditions.

The second critic model is only evaluated if the first one was satisfied. Critics provide detailed feedback with specific corrections and code improvements. Both critics must approve all four dimensions for the suggestion to pass. The full critic prompt template used for evaluation is provided in Appendix E.1.

3. *Improve*: Based on validation results and critics' feedback, the generator LLM refines the suggested task, addressing any identified issues in the prompt or tests. The prompt for the generator model is provided in Appendix E.2.

Throughout this process, human experts can guide the refinement by providing specific instructions to both the generator and critics, allowing for targeted improvements while maintaining the benefits of automated evaluation.

Stage III: Expert Review and Finalization. Human experts review and perform optional final modifications in each task to ensure:

- **Fairness**: The prompt and tests are properly aligned so any valid solution following the prompt will pass the tests;
- **Test coverage**: Tests adequately verify the prompt requirements;
- **Challenge level**: The task presents an appropriate difficulty for state-of-the-art models;
- **Automatic validation**: The canonical solution passes all tests.

This multi-faceted approach ensures that our benchmark contains systematically vetted and validated infrastructure-as-code tasks that provide a rigorous evaluation framework for LLMs.

The final benchmark tasks are stored as JSON files containing the context files (with masking tags), natural-language prompts, canonical solutions, tests, and CDK version information. See Appendix D.2 for a task format preview and D.1 for example prompts.

3.2 Dataset Collection Challenges

Building a high-quality IaC benchmark required substantial engineering expertise. Each task needed rigorous human oversight to ensure it was challenging, objective and fair, with aligned prompts, tests, and solutions. This process demanded experienced IaC developers with deep cloud knowledge. On average, an engineer could produce five high-quality dataset items per day, which required us to employ tens of cloud practitioners collaborating on building the dataset.

As outlined in Section 3.1, we developed a semi-automated pipeline using LLMs to jumpstart creation, provide refinement, and offer critique. While this helped to derive examples from existing repositories, human oversight remained essential for the final validation and quality control. A major obstacle was the lack of test coverage in open-source IaC repositories. Most open-source IaC projects lack comprehensive tests for CloudFormation templates, requiring us to develop tests for each task ourselves.

By releasing our task generation pipeline alongside the benchmark, we aim to establish a scalable process that directs the focus of the human expertise on the area where it matters most: refining and validating machine-generated tasks rather than creating new items from scratch.

3.3 Problem Definition

Model Input. Given a textual instruction describing a desired functionality within the IaC codebase, a language model is tasked with generating appropriate code modifications that fulfill the described objective. Specifically, the model receives a detailed prompt outlining the components, their interactions, and constraints. For example, a prompt might instruct the model to set up an event-driven pipeline involving components such as a message broker, data delivery stream, object storage bucket, access control policies, and event routing rules. Examples of the prompts can be found in Appendix D.1. The entire codebase is provided to the LLM as additional context. The resulting generated solution is represented as structured code edits that can be directly integrated into the existing codebase.

Evaluation Metrics. To objectively evaluate the correctness of the generated solutions, we integrate generated code modifications into the target codebase and subsequently execute predefined unit tests designed to validate the implemented functionality. These tests check for the presence and correctness

Table 1: Performance metrics for proprietary and open-source LLMs on IaC generation tasks. Models are grouped by source type and provider, and sorted from the most recent to the oldest variant within each group. † indicates LLMs executed with reasoning capabilities. ‡ Claude 3.7 was executed without extended reasoning. Top results are in **bold**.

Source	Company	Model	Correctness	Generation Success	Passed Tests Share
Proprietary	Anthropic	Claude 3.7 Sonnet ‡	**34%**	**100%**	**53.1%**
		Claude 3.5 Sonnet V2	32%	**100%**	47%
		Claude 3.5 Sonnet	29%	**100%**	47.9%
		Claude 3 Haiku	8%	88%	10.7%
	Google	Gemini 2.5 Pro (03-25 Preview) †	29%	97%	42.5%
		Gemini 2.5 Pro (05-06 I/O Edition Preview) †	29%	95%	41.1%
		Gemini 2.0 Flash Lite	5%	79%	11.5%
		Gemini 2.0 Flash	4%	43%	6.9%
	OpenAI	OpenAI o3 †	23%	**100%**	30.8%
		OpenAI o4-mini †	23%	99%	32.1%
		GPT-4.1	18%	**100%**	26.4%
		GPT-4o Mini	4%	84%	7%
Open-Source	DeepSeek	DeepSeek R1 †	24%	**100%**	34.9%
	Mistral	Mistral Large	14%	89%	20.3%
		Codestral	9%	96%	15%
	Meta	LLaMA 3.1 405B Instruct	9%	97%	13%
		LLaMA 3.1 70B Instruct	3%	97%	7.7%
		LLaMA 4 Maverick 17B Instruct	8%	94%	13%
		LLaMA 4 Scout 17B Instruct	2%	76%	2.4%
	Alibaba	Qwen2.5 72B Instruct Turbo	0%	94%	2.7%

of the infrastructure components described by the prompt, verifying criteria such as resource creation, correct permissions and configurations, and successful integration between components. A solution is considered correct if it integrates into the codebase without errors and passes all associated unit tests.

The key evaluation metrics for our benchmark vary according to the number of trials for each model on each task. The considered metrics are:

- **pass@k** as defined in [10]: Probability that LLM completes a task at least once in k trials.
- **Correctness**: Average number of tasks completed (equals to pass@1 for one-trial experiments). A task is completed if all of its tests are passed.
- **Generation Success**: Proportion of generated solutions that follow the required format and could be integrated within existing codebase.
- **Passed Tests Share**: Proportion of tests across all examples that are passed.

4 Experimental Results

We evaluate 20 state-of-the-art language models on 100 SWE-InfraBench tasks. Each model is given a single attempt to generate a solution for each task. The solutions are integrated into the task context repository, and tests are run on the resulting CloudFormation output to assess correctness.

SWE-InfraBench proves to be challenging, with even the top-performing models solving less than 35% of the tasks completely. This highlights the complexity of IaC generation, which requires both domain knowledge, precise syntax adherence as well as deep prompt and context understanding.

Table 1 shows the results for 20 models on the metrics defined in Section 3.3, averaged over all code generation tasks. The best performance is achieved by the Claude Sonnet family of models (3.7, 3.5 V2 and 3.5). While these models reportedly excel in code generation benchmarks [4], such as SWE-bench [20], it is also worth noting that the input prompt format used for all the Anthropic models incorporates additional XML tags, unlike the default prompt template for other LLMs (see Appendix E.3). Notably, reasoning models such as Gemini 2.5 Pro, DeepSeek R1, OpenAI o3, and o4-mini demonstrate stronger capabilities than non-reasoning models such as GPT-4.1, Claude 3 Haiku, and Gemini 2.0 Flash. This aligns with our expectations, as IaC generation requires substantial reasoning to understand the requirements, determine appropriate resources, and establish correct relationships between components. Nevertheless, as shown in Appendix F, reasoning models require an increased token budget.

Models generally achieve high generation success rates (over 94%), indicating they could follow the output format, but their ability to produce fully correct solutions varies significantly. This suggests the challenge is not understanding the requirements but implementing the correct functionality. The "Passed Tests Share" metric shows that even when models do not solve tasks completely, they often implement substantial portions of the code correctly.

Since a substantial share of the tasks are created from open-source data, we separately confirm that tasks originating from open-source repositories are not less challenging than custom-developed ones. The details of this analysis are provided in Appendix H.

4.1 Consistency Analysis

To evaluate models reliability, we conduct a separate multi-trial analysis on a selected subset of models. For each task, we execute five independent solution attempts (trials) per model, allowing us to measure both peak performance and consistency. This approach addresses two critical questions: (1) can a model solve a given task at least once in K attempts (pass@K), and (2) how consistently does the model produce the same outcome across attempts?

As shown in Table 2, higher pass@k values indicate that models can solve more tasks given multiple attempts, which is valuable for iterative development scenarios. At the same time, average correctness across five generations for the best-performing models falls below 30%, indicating that it is challenging to achieve consistently accurate generations. These results indicate that in practical applications, multiple generation attempts can increase success rates, particularly for complex IaC tasks where small syntax variations can determine the successful infrastructure creation. We provide further analysis of the relationship between the average correctness and success consistency in Appendix I.

Table 2: Model Performance with Multiple Independent Trials

Model	Average Correctness	pass@1	pass@2	pass@5
Claude 3.7 Sonnet	**28.8%**	**34%**	36%	41%
Gemini 2.5 Pro (03-25 Preview)	26.4%	28%	**38%**	**47%**
DeepSeek R1	24.6%	24%	33%	46%
OpenAI o3	22.8%	23%	30%	45%
LLaMA 3.1 405B Instruct	6.8%	8%	11%	13%

Average Correctness: Average success rate across all trials (all tests passed)
pass@1: Probability of solving the task in a single attempt
pass@2: Probability of solving the task in at least one of two attempts
pass@5: Probability of solving the task in at least one of five attempts

4.2 Error Type Distribution

Our analysis of error types across 20 LLMs reveals distinct patterns in their ability to generate IaC solutions, as shown in Figure 3. Incorrect property usage in CDK constructs and other syntax errors make a considerable part of all failures. Such issues can be caused both by models lacking information about particular components syntax as well as models having a knowledge cutoff that is earlier than more recent CDK versions.

These findings suggest that agent-like approaches that allow models to make an attempt to generate the code, analyze the errors, access recent documentation and refine solutions could substantially improve performance, particularly for syntax-related issues. This would better mirror the iterative development process that human developers use when working with IaC problems.

4.3 Multi-Turn Agent Performance

To investigate the potential benefits of iterative approaches, we implement a two-turn agent that provides models with feedback from their initial attempts. When a model fails to solve a task on the first try, error messages and test results are fed back to the model for a second attempt. The feedback mechanism is implemented with two distinct verbosity configurations: low-verbosity (V_L), which provides basic error messages and test pass/fail counts, and high-verbosity (V_H), which includes

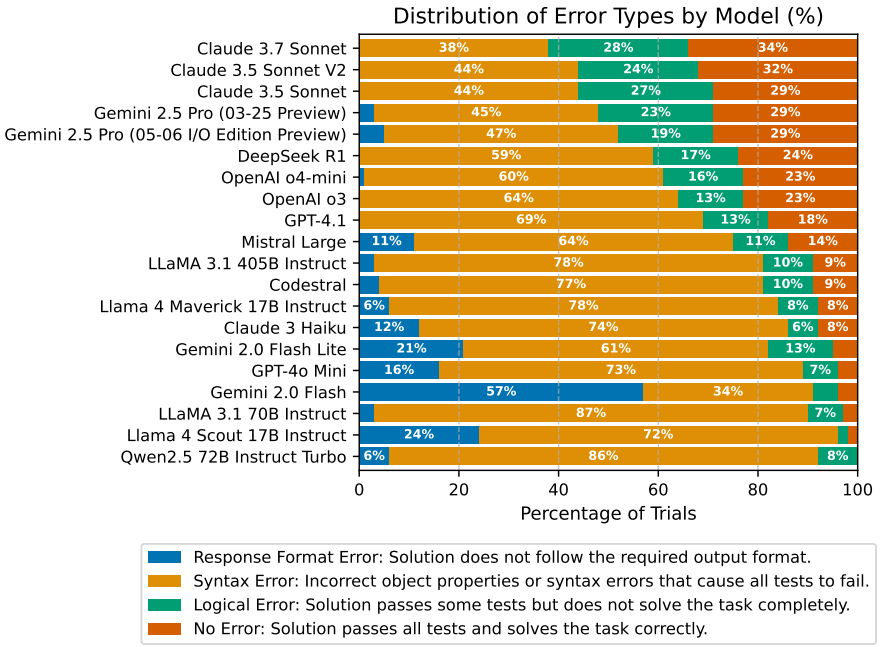

Figure 3: Distribution of error types across trials. *Syntax errors* are the dominant failure type (40–85% of errors), primarily stemming from incorrect property usage in CDK constructs. *Logical errors* (5–30%) occur when models produce syntactically valid code that fails to meet task requirements or capture context. *Response format errors* are relatively rare, with Claude and OpenAI models demonstrating particular strength in format adherence.

comprehensive output from tests with full tracebacks and detailed diagnostic information (see Table 6 for a complete specification).

Additionally, we enhance this approach with a Retrieval-Augmented Generation (RAG) system that operates in a two-phase process. In the first phase, along with generating the initial solution, models produce relevant keywords for documentation retrieval. These keywords are used to fetch the top-k most relevant documentation pages from AWS documentation. If the initial solution fails, the second phase incorporates both the error feedback and the retrieved documentation into the context for the model's second attempt.

Our results demonstrate that multi-turn interaction substantially improves performance across all models, as shown in Table 3. Claude 3.7 Sonnet achieves the highest correctness score of 64% with the high-verbosity configuration (among standard two-turn configurations), representing a significant absolute improvement of 30 percentage points over its one-turn baseline. Additional details on error distribution changes for multi-turn approach can be found in Appendix J.

The interaction between model scale and verbosity reveals other interesting patterns. While larger models demonstrate substantial improvements with increased verbosity, smaller models like Mistral Large show more modest gains (two percentage points gain in correctness score for standard configuration when going from V_L to V_H). This pattern indicates that the ability to effectively utilize detailed error information may be contingent on model capacity.

The verbosity effect proves particularly noteworthy, with V_H configurations consistently outperforming V_L across LLMs, suggesting that detailed diagnostic information enables more effective error correction. RAG configurations add further complexity to the performance landscape, with Claude 3.5 Sonnet V2 achieving the highest correctness score of 65% – a substantial 33 percentage points improvement over its one-turn performance. The effectiveness of RAG varies significantly across model architectures. Smaller models like LLaMA 4 Maverick show modest but consistent improvements with RAG, while some high-performing models like GPT-4.1 demonstrate reduced effectiveness (26%) compared to their standard two-turn performance (48%). This heterogeneity suggests that RAG implementation may require model-specific optimization strategies.

Table 3: Comparison of One-Turn vs. Two-Turn LLM Performance (with and without RAG).

Model	Correctness (↑)						Passed Tests Share (↑)					
	One-Turn	Two-Turn		Two-Turn + RAG		Best	One-Turn	Two-Turn		Two-Turn + RAG		Best
		V_L	V_H	V_L	V_H			V_L	V_H	V_L	V_H	
Claude 3.7 Sonnet	34.0%	44.0%	64.0%	51.0%	60.0%	**64.0%**	53.1%	49.5%	71.6%	60.6%	66.1%	**71.6%**
Claude 3.5 Sonnet V2	32.0%	52.0%	56.0%	52.0%	65.0%	**65.0%**	47.0%	60.9%	66.1%	65.5%	74.2%	**74.2%**
Gemini 2.5 Pro (03-25 Preview)	29.0%	41.0%	40.0%	41.0%	33.0%	**41.0%**	42.5%	41.5%	41.4%	41.3%	33.0%	**42.5%**
DeepSeek R1	24.0%	40.0%	43.0%	45.0%	39.0%	**45.0%**	34.9%	49.0%	51.7%	51.9%	49.5%	**51.9%**
GPT-4.1 (2025-04-14)	18.0%	40.0%	48.0%	46.0%	26.0%	**48.0%**	26.4%	48.0%	55.9%	56.7%	30.9%	**55.9%**
Mistral Large	14.0%	21.0%	23.0%	14.0%	17.0%	**23.0%**	20.3%	24.2%	28.6%	17.9%	23.7%	**28.6%**
LLaMA 4 Maverick 17B Instruct	8.0%	15.0%	18.0%	21.0%	21.0%	**21.0%**	13.0%	20.5%	23.7%	25.6%	27.3%	**27.3%**

Correctness: Percentage of completions where the solution passed all test cases.
Passed Tests Share: Average percentage of unit tests passed, including partial completions.
Two-Turn + RAG: Two-step prompting approach with retrieval-augmented generation support.
V_L/V_H: Low/High verbosity configurations.

These results warrant several important considerations. First, full test failure tracebacks may reveal details that might inadvertently help models correct their solutions without genuine understanding of the underlying problems. The error messages often contain specific values expected for CloudFormation properties, potentially allowing models to pattern-match rather than reason about the proper solution. This may make the observed performance gains overoptimistic with regards to real-world use cases, where the true CloudFormation templates are unknown. Second, we observe that DeepSeek R1 allocates a substantial number of tokens to reasoning during its second attempt, which requires the raise of maximum output token parameter to prevent it from exhausting the token limit before generating a complete solution. Finally, the multi-turn approach, while effective, comes with increased computational costs, both in terms of token consumption and inference time.

Our results suggest that enabling iterative refinement with detailed feedback is a promising direction for enhancing model capabilities. The consistent advantage of high-verbosity configurations highlights the importance of detailed diagnostic information in enabling successful iterative refinement, particularly for more sophisticated model architectures. However, the effectiveness of enhancement strategies like RAG is not uniform across model architectures.

5 Conclusion

This work introduces SWE-InfraBench, the first benchmark designed to evaluate language models in modifying imperative infrastructure-as-code frameworks, specifically AWS CDK, on realistic cloud development tasks. SWE-InfraBench presents challenging IaC instances, each requiring LLMs to understand modification instructions, reason about cloud resource dependencies, and generate precise code changes within existing CDK repositories, verified automatically via unit tests. Our analysis reveals that these tasks, mirroring iterative enterprise workflows, pose significant difficulties for current models; our extensive experiments show that the best performing LLM Claude 3.7 struggles, with the top single-attempt success rate reaching only 34%, although multi-turn agentic approaches show promise, boosting performance up to 65%. Other strong LLMs, including reasoning models such as DeepSeek r1, falls short with only 24% pass rate.

SWE-InfraBench currently centers on AWS CDK (Python), which is a popular way of building IaC applications in the cloud. We hope in the future to broaden the benchmark to encompass additional cloud services and IaC frameworks. Similarly, while not a focus of the present work, a natural extension is to enable agents with various tools such as search engine, bash script, python environment and others that could facilitate development of IaC code modifications. Our results with two-turn LLMs show that this is a promising direction. Finally, although we employ execution-based testing for model evaluation, this method alone cannot fully assess the quality of generated infrastructure code. Automatically generated solutions may be syntactically correct and pass unit tests, yet still suffer from inefficiencies, misconfigurations, or security risks when deployed. Future evaluation strategies could incorporate static analysis tools, cost simulation, or LLM-as-a-Judge review to address these gaps.

References

[1] Amazon Web Services: Increasing developer velocity using aws cloud development kit with godaddy (2025), https://aws.amazon.com/solutions/case-studies/godaddy-cdk-case-study/, accessed: 2025-05-05

[2] Amazon Web Services: What is aws cloudformation? - aws cloudformation (2025), https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/Welcome.html, accessed: 2025-05-05

[3] Amazon Web Services: What is the aws cdk? - aws cloud development kit (aws cdk) v2 (2025), https://docs.aws.amazon.com/cdk/v2/guide/home.html, accessed: 2025-05-05

[4] Anthropic: Claude 3.7 sonnet system card (February 2025), https://www.anthropic.com/claude-3-7-sonnet-system-card, technical report detailing Claude 3.7 Sonnet, a hybrid reasoning model

[5] Athiwaratkun, B., Gouda, S.K., Wang, Z., Li, X., Tian, Y., Tan, M., Ahmad, W.U., Wang, S., Sun, Q., Shang, M., et al.: Multi-lingual evaluation of code generation models. arXiv preprint arXiv:2210.14868 (2022)

[6] Ava: Terraform vs AWS CloudFormation vs AWS CDK. https://kudulab.io/posts/2023-2-terraform-vs-cdk-vs-cloudformation/ (2023), accessed: 2025-05-09

[7] Buchh, I.: Two billion downloads of the terraform aws provider shows value of iac for infrastructure management. https://aws.amazon.com/blogs/aws-insights/two-billion-downloads-of-the-terraform-aws-provider-shows-value-of-iac-for-infrastructure-management/ (2023), aWS Insights Blog, 11 Oct 2023

[8] Canalys: Worldwide cloud infrastructure services expenditure increased 20% year on year in q4 2024 to us$86 billion (2025), https://www.canalys.com/newsroom/worldwide-cloud-service-q4-2024, accessed: 2025-05-05

[9] Cassano, F., Li, L., Sethi, A., Shinn, N., Brennan-Jones, A., Ginesin, J., Berman, E., Chakhnashvili, G., Lozhkov, A., Anderson, C.J., et al.: Can it edit? evaluating the ability of large language models to follow code editing instructions. arXiv preprint arXiv:2312.12450 (2023)

[10] Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H.P., et. al, J.K.: Evaluating large language models trained on code (2021)

[11] Chowdhury, N., Aung, J., Shern, C.J., Jaffe, O., Sherburn, D., Starace, G., Mays, E., Dias, R., Aljubeh, M., Glaese, M., Jimenez, C.E., Yang, J., Ho, L., Patwardhan, T., Liu, K., Madry, A.: Introducing swe-bench verified (2024), https://openai.com/index/introducing-swe-bench-verified/, accessed: 2025-03-02

[12] Delta, E.: How many companies use cloud computing in 2024? [10 statistics] (2025), https://edgedelta.com/company/blog/how-many-companies-use-cloud-computing, accessed: 2025-05-05

[13] Ding, Y., Wang, Z., Ahmad, W., Ding, H., Tan, M., Jain, N., Ramanathan, M.K., Nallapati, R., Bhatia, P., Roth, D., et al.: Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. Advances in Neural Information Processing Systems **36**, 46701–46723 (2023)

[14] Du, M., Luu, A.T., Ji, B., Liu, Q., Ng, S.K.: Mercury: A code efficiency benchmark for code large language models. arXiv preprint arXiv:2402.07844 (2024)

[15] Du, X., Liu, Y., Wang, Y., Zhang, W., Li, X.: Classeval-t: A class-level code translation benchmark. arXiv preprint arXiv:2411.06145 (2024)

[16] Etsenake, D., Nagappan, M.: Understanding the human-llm dynamic: A literature survey of llm use in programming tasks. arXiv preprint arXiv:2410.01026 (2024)

[17] Frois, J., Padrão, L., Oliveira, J., Xavier, L., Tavares, C.: Terraform and aws cdk: A comparative analysis of infrastructure management tools. In: Anais do XXXVIII Simpósio Brasileiro de Engenharia de Software. pp. 623–629. SBC, Porto Alegre, RS, Brasil (2024). https://doi.org/10.5753/sbes.2024.3577, https://sol.sbc.org.br/index.php/sbes/article/view/30404

[18] Guo, J., Li, Z., Liu, X., Ma, K., Zheng, T., Yu, Z., Pan, D., Li, Y., Liu, R., Wang, Y., et al.: Codeeditorbench: Evaluating code editing capability of large language models. arXiv preprint arXiv:2404.03543 (2024)

[19] Jiang, J., Wang, F., Shen, J., Kim, S., Kim, S.: A survey on large language models for code generation. arXiv preprint arXiv:2406.00515 (2024)

[20] Jimenez, C.E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., Narasimhan, K.: Swe-bench: Can language models resolve real-world github issues? arXiv preprint arXiv:2310.06770 (2023)

[21] Kawaguchi, M., Mizutani, K., Iguchi, N.: An implementation of misconfiguration prevention system using language model for a network automation tool. IEICE Proceedings Series **72**(S5-8) (2022)

[22] Kon, P.T., Liu, J., Qiu, Y., Fan, W., He, T., Lin, L., Zhang, H., Park, O.M., Elengikal, G.S., Kang, Y., et al.: Iac-eval: A code generation benchmark for cloud infrastructure-as-code programs. Advances in Neural Information Processing Systems **37**, 134488–134506 (2024)

[23] Larssen, E.: It's time to retire terraform. https://blog.realkinetic.com/its-time-to-retire-terraform-30545fd5f186 (2024), real Kinetic Blog, Apr 23, 2024

[24] Liu, C., Wu, X., Feng, Y., Cao, Q., Yan, J.: Towards general loop invariant generation: A benchmark of programs with memory manipulation. Advances in Neural Information Processing Systems **37**, 129120–129145 (2024)

[25] Morris, K.: Infrastructure as Code: Managing Servers in the Cloud (2nd Edition). O'Reilly Media (2020)

[26] Muennighoff, N., Liu, Q., Zebaze, A., Zheng, Q., Hui, B., Zhuo, T.Y., Singh, S., Tang, X., von Werra, L., Longpre, S.: Octopack: Instruction tuning code large language models (2023)

[27] Pahl, C., Gunduz, N.G., Sezen, O.C., Ghamgosar, A., El Ioini, N.: Infrastructure as code: Technology review and research challenges. In: Proc. of the 15th Int. Conf. on Cloud Computing and Services Science (CLOSER) (2025)

[28] Pan, R., Ibrahimzada, A.R., Krishna, R., Sankar, D., Wassi, L.P., Merler, M., Sobolev, B., Pavuluri, R., Sinha, S., Jabbarvand, R.: Lost in translation: A study of bugs introduced by large language models while translating code. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering. pp. 1–13 (2024)

[29] Poccia, D.: AWS Cloud Development Kit (CDK) – TypeScript and Python are Now Generally Available. https://aws.amazon.com/blogs/aws/aws-cloud-development-kit-cdk-typescript-and-python-are-now-generally-available/ (2019), aWS News Blog, 11 Jul 2019

[30] Prasad, A., Stengel-Eskin, E., Chen, J.C.Y., Khan, Z., Bansal, M.: Learning to generate unit tests for automated debugging. arXiv preprint arXiv:2502.01619 (2025)

[31] Pujar, S., Buratti, L., Guo, X., Dupuis, N., Lewis, B., Suneja, S., Sood, A., Nalawade, G., Jones, M., Morari, A., et al.: Automated code generation for information technology tasks in yaml through large language models. In: 2023 60th ACM/IEEE Design Automation Conference (DAC). pp. 1–4. IEEE (2023)

[32] Quattrocchi, G., Tamburri, D.A.: Infrastructure as code. IEEE Software **40**(1), 37–40 (2023). https://doi.org/10.1109/MS.2022.3212034

[33] Rahman, S., Kuhar, S., Cirisci, B., Garg, P., Wang, S., Ma, X., Deoras, A., Ray, B.: Utfix: Change aware unit test repairing using llm. Proceedings of the ACM on Programming Languages **9**(OOPSLA1), 143–168 (2025)

[34] Raihan, N., Newman, C., Zampieri, M.: Code llms: A taxonomy-based survey. In: 2024 IEEE International Conference on Big Data (BigData). pp. 5402–5411. IEEE (2024)

[35] Rashid, M.S., Bock, C., Zhuang, Y., Buccholz, A., Esler, T., Valentin, S., Franceschi, L., Wistuba, M., Sivaprasad, P.T., Kim, W.J., et al.: Swe-polybench: A multi-language benchmark for repository level evaluation of coding agents. arXiv preprint arXiv:2504.08703 (2025)

[36] Wicher, F.: Why we chose cloud development kit for terraform over hashicorp configuration language for our infrastructure. https://andamp.io/blog/why-we-chose-cloud-development-kit-for-terraform-over-hashicorp-configuration-language-for-our-infrastructure (2025), andamp Engineering Blog, Mar 19, 2025

[37] Wiggers, S.J.: AWS Cloud Development Kit (CDK) Is Generally Available, Enhancing Coding Cloud Infrastructure. InfoQ News, July 2019 (2019), https://www.infoq.com/news/2019/07/amazon-aws-cdk-ga/

[38] Xu, Y., Chen, Y., Zhang, X., Lin, X., Hu, P., Ma, Y., Lu, S., Du, W., Mao, Z.M., Cai, D., et al.: Cloudeval-yaml: A practical benchmark for cloud configuration generation. Proceedings of Machine Learning and Systems **6**, 173–195 (2024)

[39] Yu, H., Shen, B., Ran, D., Zhang, J., Zhang, Q., Ma, Y., Liang, G., Li, Y., Wang, Q., Xie, T.: Codereval: A benchmark of pragmatic code generation with generative pre-trained models. In: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering. pp. 1–12 (2024)

Appendix

A SWE-InfraBench Characteristics

SWE-InfraBench tasks contain examples for multiple CDK versions ranging from 2.0.0 to more recent version like 2.189.1, with the majority of tasks suitable for 2.178.2 and newer (see Figure 5). The tasks also vary in number and length of context files, prompt length and canonical solution length. The key dataset statistics are illustarted in Figure 4.

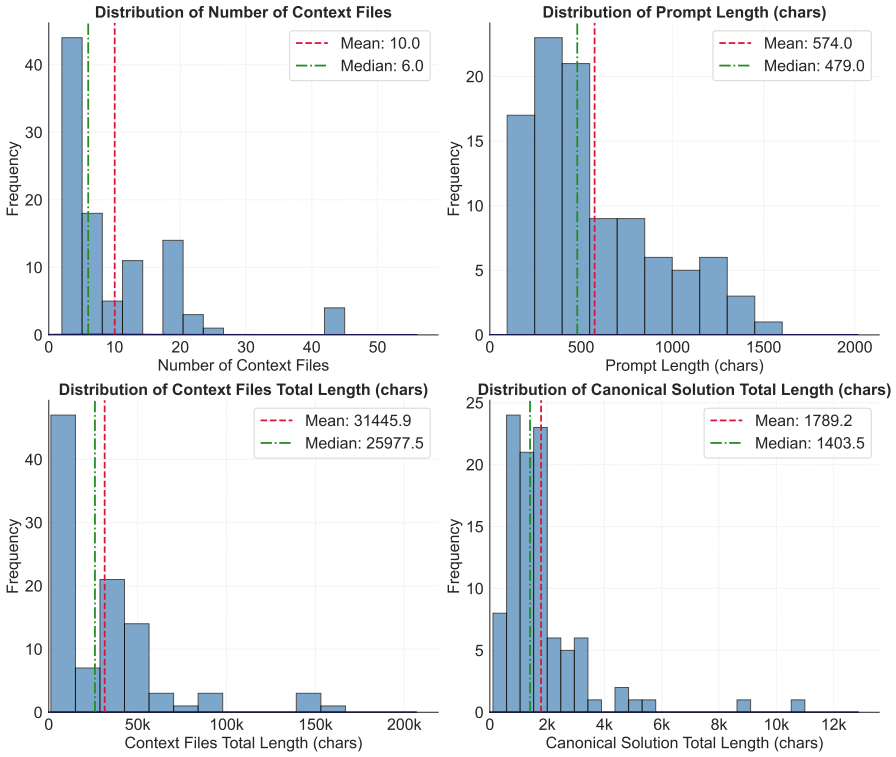

Figure 4: SWE-InfraBench Statistics on Context Size and Solution Length

B CDK Versions Distribution

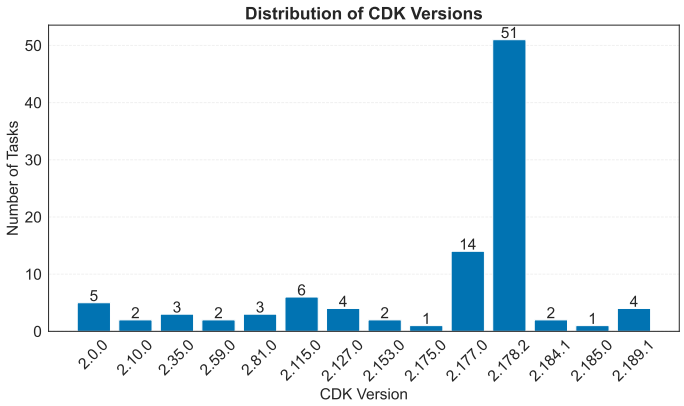

Figure 5: Distribution of AWS CDK Versions Across SWE-InfraBench Tasks

Figure 5 shows the distribution of AWS CDK versions across the SWE-InfraBench tasks. The benchmark predominantly utilizes version 2.178.2, highlighting a concentration on recent library versions, while also including several earlier versions to ensure diversity and represent real-world variability in CDK projects.

C Source Repositories Licences

SWE-InfraBench tasks are derived from open-source repositories and custom-developed sources. Note that original code snippets were substantially modified to create the tasks. See Table 4 for details on open-source repositories.

Table 4: Open-source repositories used as sources for SWE-InfraBench tasks

Repository Name	Source	License
aws-cdk-examples	aws-samples/aws-cdk-examples	Apache-2.0
generative-ai-cdk-constructs-samples	aws-samples/generative-ai-cdk-constructs-samples	Apache-2.0
generative-ai-ml-latam-samples	aws-samples/generative-ai-ml-latam-samples	MIT-0
aws-cdk-lambda-import-export-redshift-ddl	aws-samples/aws-cdk-lambda-import-export-redshift-ddl	MIT-0
amazon-elasticache-demo-using-aws-cdk	aws-samples/amazon-elasticache-demo-using-aws-cdk	MIT-0
deploy-langfuse-on-ecs-with-fargate	aws-samples/deploy-langfuse-on-ecs-with-fargate	MIT-0

All repository URLs are prefixed with `https://github.com/`

D Tasks Preview

D.1 Prompt Examples

SWE-InfraBench includes a diverse set of prompts that test various aspects of infrastructure-as-code generation. See examples of such prompts below:

API Gateway Integration

Generate code to create an API Gateway SampleAPI-EventBridge-Multi-Consumer that integrates with the event producer with proxy integration. Do not add a catch-all route in api routing (use proxy=False). Add a resource named 'items' to the root of the API and create a POST method for this resource.

Bedrock Agent Setup

Create a Bedrock agent using CDK with the following requirements in the CfnAgent construct.
Set the agent name, description, model and instruction from the class attributes.
Also, set the agent resource role ARN from the previously created role and add idle session time of 600 seconds.
We want the draft version to be always in sync via auto preparation and add test alias tags.
Finally include action groups and collaboration if provided and ensure the removal policy destroys the agent when the stack is deleted

WhatsApp IAM Policy

Create CDK code to add IAM policies to a Lambda function, granting permissions for social-messaging, transcribe, and bedrock services.
The policies should have separate policies for
1) allow sending whatsapp messages and getting media from them for all cell numbers in all regions and accounts
2) access to any transcribe action on all resources
3) and access to invoke* all agents, inference profiles and models in oregon

D.2 Task Example

Figure 6 illustrates the structure of a task from SWE-InfraBench. Each task is stored as a JSON file containing the prompt, context files, canonical solution, and tests.

Example Task: API Gateway Integration with EventBridge


```
{
  "task_id": "67ad6ef1-fb3e-45e6-b5e1-83ae385528b5",
  "entry_point": "api-eventbridge-lambda+api_gateway_integration",
  "prompt": "Generate code to create an API Gateway
  ↪  SampleAPI-EventBridge-Multi-Consumer
           that integrates with the event producer with proxy
           ↪  integration.
           Do not add a catch-all route in api routing (use
           ↪  proxy=False).
           Add a resource named 'items' to the root of the API and
           ↪  create a
           POST method for this resource.",
  "cdk_version": "2.178.2",
  "context": {
    "app.py": "#!/usr/bin/env python3\n\nfrom aws_cdk import
    ↪  App\n\nfrom api_eventbridge_lambda.api_eventbridge_lambda
    ↪  import ApiEventBridgeLambdaStack...",
    "api_eventbridge_lambda/api_eventbridge_lambda.py": "from
    ↪  constructs import Construct\nfrom aws_cdk import...",
    "lambda/event_consumer_lambda.py": "import json\nimport
    ↪  logging\n\nlogger = logging.getLogger()...",
    "lambda/event_producer_lambda.py": "import json\nimport
    ↪  boto3\nimport datetime..."
  },
  "canonical_solution": {
    "api_eventbridge_lambda/api_eventbridge_lambda.py": [
      "--- without_solution\n\n+++ with_solution\n\n@@ -102,0 +103,7
      ↪  @@\n\n+        # defines an API Gateway REST API resource
      ↪  backed by our \"atm_producer_lambda\" function.\n\n+
      ↪  api = api_gw.LambdaRestApi(self,
      ↪  'SampleAPI-EventBridge-Multi-Consumer',\n\n+
      ↪  handler=event_producer_lambda,\n\n+
      ↪  proxy=False\n\n+                                )\n\n+
      ↪  items = api.root.add_resource(\"items\")\n\n+
      ↪  items.add_method(\"POST\")  # POST /items\n"
    ]
  },
  "tests": {
    "test_api_gateway_integration.py": "import aws_cdk as cdk\nfrom
    ↪  aws_cdk.assertions import Template, Match..."
  }
}
```


Figure 6: Example task from InfraBench showing the JSON structure. Context and test files are truncated for brevity.

E Prompt Templates

E.1 Critic Prompt Template

The following is the template used for the critic in the InfraBench dataset construction process:

Critic Prompt Template

As an AI assistant specialized in Infrastructure as Code, your task is to critique a dataset item created for an LLM code generation challenge.
You will be given:
1. A repository description
2. The full repository code
3. A pre-suggestion with a specific code section to mask
4. The actual code that was masked
5. The generated human language prompt
6. The generated test file
Your job is to critically evaluate:
1. Whether the prompt accurately describes what needs to be implemented
2. Whether the tests effectively validate all requirements stated in the prompt
3. Whether the prompt and tests are general enough to allow various valid solutions that preserve the functionality of the masked code while remaining compatible with the overall repository structure
REPOSITORY DESCRIPTION

```
[Repository description is provided here]
```

REPOSITORY CONTENT

```
[Repository files are provided here]
```

PRE-SUGGESTION

```
Item Name: [Item name]
File Path: [File path]
Start Line: [Start line]
End Line: [End line]
Complexity: [Complexity]
Rationale: [Rationale]
```

ACTUAL CODE TO BE MASKED

```
[Masked code is provided here]
```

GENERATED PROMPT

```
[Generated prompt is provided here]
```

GENERATED TEST FILE
Path: [Test file path]

```
[Test file content is provided here]
```

GENERATOR NOTES

```
[Generator notes are provided here]
```

CRITIQUE GUIDELINES
Prompt vs Functional Requirements Evaluation
1. Does the prompt clearly describe ALL the necessary functional aspects needed for the masked code?
2. Is it sufficiently detailed for someone to implement the solution correctly without seeing the masked code?

3. Are there any ambiguities or missing requirements that would prevent a correct implementation?
4. Does it avoid revealing the actual implementation details while still being complete?
5. If something can be inferred unequivocally from the repository code, it does not need to be specified in the prompt.
Generality of Tests Evaluation
1. Tests should be general enough so that they pass if a developer or LLM follows the prompt accurately, regardless of the specific implementation details.
2. The prompt should give only the minimum necessary instructions needed to explain the functional requirements, while considering how the tests are built.
3. Tests should verify functionality rather than specific implementation approaches - they should allow for multiple valid solution patterns that fulfill the prompt.
4. Evaluate if tests are overly restrictive by enforcing a particular implementation approach when other valid approaches could fulfill the same requirements.
5. Tests won't be accepted if they don't pass with the original masked code (tested elsewhere) but consider as well if other reasonable implementations would pass.
Test Evaluation
For each test in the test file:
1. What is this test specifically checking for?
2. Is this test testing for something that's explicitly stated in the prompt?
3. A test is valid if it tests integration with functionality that exists elsewhere in the code base (not just the masked section).
4. Is the test appropriately written to verify the requirement?
5. If the test might fail with some valid implementations (including the original masked code), should the prompt be more explicit or should the test be less restrictive?
Test Completeness Evaluation
1. Do the tests collectively verify ALL requirements mentioned in the prompt?
2. Are there any requirements in the prompt that aren't tested?
3. Are there any tests for requirements not mentioned in the prompt?
4. Are all edge cases and error conditions properly tested?
Return the response in this JSON format:

```
{
    "prompt_vs_functional": {
        "explanation": "Detailed explanation of whether the prompt
    accurately describes all necessary functional aspects of the
    masked code",
        "corrections": "Specific corrections with concrete
    implementation suggestions - provide exact wording changes or
    additions to the prompt", // Only include if there are issues
        "example_improvements": "Suggested rewrites of problematic
    sections with specific language", // Only include if there are
    issues
        "accept": true|false  // Conclusion based on the
    explanation
    },
    "generality": {
        "explanation": "Analysis of whether the tests allow for
    multiple valid implementations that satisfy the requirements in
    the prompt",
        "issues": "Identification of any tests that could fail
    with valid implementations that follow the prompt", // Only
    include if there are issues
        "suggested_improvements": "Concrete suggestions for making
    tests more general while still validating functionality", //
    Only include if needed
        "accept": true|false  // Whether the tests are
    sufficiently general
    },
    "tests": {
```

```
        "test_name_1": {
            "purpose": "What this test is checking for",
            "coverage": "Explanation of how this relates to prompt
    requirements",
            "suggested_improvements": {
                "explanation": "Why the test needs improvement",
    // Only include if needed
                "code_snippet": "Complete improved version of the
    test with code fixes", // Provide actual code implementation
                "rationale": "Explanation of why this
    implementation is better"
            }, // Only include if needed
            "accept": true|false  // Conclusion based on the
    critic's analysis of the test quality
        },
        "test_name_2": {
            "purpose": "What this test is checking for",
            "coverage": "Explanation of how this relates to prompt
    requirements",
            "suggested_improvements": {
                "explanation": "Why the test needs improvement",
    // Only include if needed
                "code_snippet": "Complete improved version of the
    test with code fixes", // Provide actual code implementation
                "rationale": "Explanation of why this
    implementation is better"
            }, // Only include if needed
            "accept": true|false  // Conclusion based on the
    critic's analysis of the test quality
        }
        // Add entries for each test in the test file
    },
    "tests_completeness": {
        "explanation": "Analysis of whether the tests completely
    cover all prompt requirements",
        "missing_tests": ["list", "of", "requirements", "that",
    "should", "be", "tested", "but", "aren't"],
        "corrections": "Specific additional tests needed if the
    evaluation fails", // Only include if there are issues
        "accept": true|false  // Conclusion based on the analysis
    },
    "feedback": "Detailed feedback explaining all issues and
    providing clear guidance for improvements"
}
```

Be rigorous in your evaluation. The goal is to ensure high-quality dataset items that will effectively test LLM code generation capabilities. Only provide the JSON response, no additional explanation. The response will be parsed with json.loads(response) so be sure json format is correct. Instead of triple-quotes use characters.
IMPORTANT CUSTOM INSTRUCTIONS

```
[Custom instructions from human reviewers are provided here when
    available. These instructions provide special guidance in the
    evaluation process.]
```

E.2 Generator Prompt Template

The following is the template used for the generator in the InfraBench dataset construction process:

Generator Prompt Template

As an AI assistant specialized in Infrastructure as Code, your task is to create a high-quality dataset item for an LLM code generation challenge.
You will be given:
1. A repository description
2. A pre-suggestion with a specific code section to mask
3. The full repository content
Based on this information, you need to:
1. Create a clear, detailed human language prompt describing what code needs to be generated
2. Develop comprehensive pytest tests that validate the generated code meets all requirements
REPOSITORY DESCRIPTION

```
[Repository description is provided here]
```

PRE-SUGGESTION

```
Item Name: [Item name]
File Path: [File path]
Start Line: [Start line]
End Line: [End line]
Complexity: [Complexity]
Rationale: [Rationale]
```

PREVIOUS FEEDBACK
When available, this section may include:
PREVIOUS TASK SUGGESTION

```
{
  "item_name": "example_item",
  "file_path": "path/to/file",
  "insert_points": {
    "start_line": 10,
    "end_line": 20
  },
  "prompt": "previous prompt text",
  "test_file": {
    "path": "tests/test_example.py",
    "content": "previous test file content"
  },
  "generator_notes": "previous notes"
}
```

IF THERE ARE VALIDATION ERRORS
The tests failed when run against the original masked code. Here are the errors:

```
Error details from validation step
```

Make sure the original code will pass the tests, and write them correctly according to what you see in these error logs.
CRITIQUE FEEDBACK

```
[Detailed feedback from the critique step explaining issues with
    the previous suggestion.]
```

GUIDELINES FOR CREATING A QUALITY DATASET ITEM
For the human language prompt:
1. COMPLETENESS: Describe ALL functional aspects needed for the code.
2. GENERALITY: Allow for multiple possible valid solutions that fulfill the requirements.
3. CLARITY: Be specific about requirements but avoid dictating implementation specifics.
4. PRECISION: Include all requirements that would allow someone to implement the solution correctly.
5. CONTEXT: Describe the functionality, purpose, and integration with other components.

6. ESSENTIAL DETAILS ONLY: Specify necessary parameters and behaviors, but avoid over-constraining the solution.
7. NO SPOILERS: DO NOT include the actual implementation details or code snippets.
8. IMPLEMENTATION FREEDOM: Focus on "what" needs to be achieved, not "how" it must be done.
9. NO HINTS: If something can be inferred unequivocally from the repository code, it does not need to be specified in the prompt.
For the test file: 1. COMPREHENSIVE COVERAGE: Tests must verify ALL aspects mentioned in the prompt
2. EXPLICIT PURPOSE: Each test should clearly indicate what requirement it's checking
3. APPROPRIATE VERIFICATION: Tests must use assertions that correctly validate the implementation
4. PRACTICALITY: Tests MUST pass when run against the original code that will be masked
5. ROBUSTNESS: Tests should fail if important requirements are not met
6. COMPLETENESS: No requirement from the prompt should be left untested
7. STRUCTURE: Use appropriate fixtures and mocks, follow pytest best practices
REPOSITORY CONTENT File to be masked: [File path]

```
[File content is provided here]
```

```
[All other relevant context files are provided here]
```

Return the response in this JSON format:

```
{
    "item_name": "SAME_AS_PRE_SUGGESTION",
    "file_path": "path/to/file",
    "insert_points": {
        "start_line": number,
        "end_line": number
    },
    "prompt": "detailed description of what needs to be generated",
    "test_file": {
        "path": "tests/test_something.py",
        "content": "complete content of the test file including
    imports, fixtures, and test cases"
    },
    "generator_notes": "You don't need to accept all the
    suggestions from the feedback, but give an explanation of your
    approach, learnings regarding test syntax for the cannonical
    solution to pass, detailed design decisions, rationale for
    implementation choices, and how you've ensured generality in
    the prompt and tests. There will be a new critic, so explain
    your decisions without assuming the critic understands the
    current feedback. IMPORTANT: give an analysis of possible flaws
    in the generated prompt and tests focusing on its generality
    (if various valid solutions would be accepted), coverage and
    alignment between prompt and tests, etc."
}
```

Ensure that:
1. The insert_points are within the pre-suggestion range
2. The prompt is comprehensive and covers ALL requirements
3. The test_file content is complete and will validate ALL requirements
4. The tests MUST pass when run against the original masked code
Only provide the JSON response, no additional explanation.
IMPORTANT CUSTOM INSTRUCTIONS

```
[Custom instructions from human reviewers are provided here when
    available. These instructions provide special guidance in the
    generation process.]
```

E.3 Zero-shot Direct Solver

Anthropic template

You are tasked with implementing a solution based on the following prompt:

```
<example_prompt>
{example_prompt}
</example_prompt>
```

Ensure the solution is compatible with AWS Python CDK version aws-cdk-lib = `{cdk_version}`.
You have access to the following context files:

```
<context_files>
{context_files}
</context_files>
```

Your task is to provide git-style unified diffs that show the changes needed to implement the solution. For each file that needs changes, provide a unified diff. Only add content, do not remove any lines
Provide your response as a JSON object where: - Keys are the file paths - Values are arrays containing the unified diffs for each change section
The diff format should be:

```
<output_format>
--- without_solution
+++ with_solution
@@ -line,count +line,count @@
    context lines
+added lines
    context lines
</output_format>
```

Example response format:

```
<example_response>
{
    "path/to/file1.py": [
        "--- without_solution\\n+++ with_solution\\n
        @@ -10,3 +10,5
        @@\\n     existing_line\+
        new_line1\\n+    new_line2\\n
        existing_line"
    ]
}
</example_response>
```

Only provide the JSON response, no additional explanation. The response will be parsed with `json.loads(response_text)` so make sure the string is correct JSON. Do NOT include ```` ```json ````
Ensure the diffs include proper line numbers and context.

Default template

You are tasked with implementing a solution based on the following prompt:

```
{example_prompt}
```

Ensure the solution is compatible with AWS Python CDK version aws-cdk-lib = `{cdk_version}`.
You have access to the following context files:

```
{context_files}
```

Your task is to provide git-style unified diffs that show the changes needed to implement the solution. For each file that needs changes, provide a unified diff. Only add content, do not remove any lines
Provide your response as a JSON object where: - Keys are the file paths - Values are arrays containing the unified diffs for each change section
The diff format should be:

```
--- without_solution
+++ with_solution
@@ -line,count +line,count @@
    context lines
+added lines
    context lines
```

Example response format:

```
{
    "path/to/file1.py": [
        "--- without_solution\\n+++ with_solution\\n
        @@ -10,3 +10,5
        @@\\n     existing_line\+    new_line1
        \\n+    new_line2\\n
        existing_line"
    ]
}
```

Only provide the JSON response, no additional explanation. The response will be parsed with `json.loads(response_text)` so make sure the string is correct JSON. Do NOT include ```` ```json ````
Ensure the diffs include proper line numbers and context.

E.4 Zero-shot Two-Turn Solver

First Turn Prompt

```
# Task overview
You are tasked with implementing a solution
based on the following prompt:
{example_prompt}

# Steps
- Have a look at the version of aws-cdk-lib
    - Current version is {cdk_version}
- Read each all of the context files ("# Context files" section)
- Try to understand what should be done
- Read task overview
("# Task overview" section)
- If provided, read related AWS documentation
("# Related documentation" section)
- Provide solution with a specified format
("# Response format" section)

# Response format
## General guidelines
Provide your solution as a JSON object with:
1. File paths as keys
2. Arrays of unified diffs as values

## Diff format specification
```

````
Each diff must follow this structure:
```
--- without_solution
+++ with_solution
@@ -line,count +line,count @@
 context lines
+added lines
 context lines
```

## Important rules:
- Only ADD content, do not remove any lines
- Include proper line numbers and context
- Ensure diffs are properly formatted

## Example JSON response:
```json
{{
    "path/to/file1.py": [
        "--- without_solution\n+++ with_solution\n
        @@ -10,3 +10,5 @@\n
        existing_line\n+    new_line1\n+    new_line2\n
        existing_line"
    ]
}}
```

## Additional guidelines
- Provide ONLY the JSON response
- No additional explanation
- Response must be valid JSON (will be parsed with json.loads())
- If you want to think before returning response,
be short and concise

# Context files
Here are the context files to analyze:
{context_files}
````

Second Turn Prompt

```
# Task overview
You are tasked with fixing errors in an LLM-generated solution
based on the following initial prompt:
{example_prompt}

# Steps
- Have a look at the version of aws-cdk-lib
    - Current version is {cdk_version}
- Read each all of the context files ("# Context files" section)
- Try to understand what should be done
- Read task overview ("# Task overview" section)
    - This is an initial formulation of the task
    that LLM have used to generate its solution
- Read previous attempt solution code
("# Previous attempt" section)
    - This is the solution generated by LLM which fails checks
- Read error traceback ("# Error message" section)
```

```
    - Interpret error message
    - Identify specific error type (syntax, type, logic, etc.)
- If provided, read related AWS documentation
("# Related documentation" section)
- Provide error analysis
    - Put it in "error_analysis"
- Provide a working solution with a specified format
    - Put it in "regenerated_solution"

# Response format
## General guidelines
Provide your solution as a JSON object with:
1. "error_analysis": Your complete error analysis
2. "regenerated_solution": Object containing file paths and
their diffs

## Diff format specification
Each diff must follow this structure:
```
--- without_solution
+++ with_solution
@@ -line,count +line,count @@
 context lines
+added lines
 context lines
```

## Important rules:
- Only ADD content, do not remove any lines
- Include proper line numbers and context
- Ensure diffs are properly formatted

## Example JSON response:
```json
{{
    "error_analysis": "The error occurred because...",
    "regenerated_solution": {{
        "path/to/file1.py": [
            "--- without_solution\n+++ with_solution\n
            @@ -10,3 +10,5
            @@\n     existing_line\n+    new_line1\n
            +    new_line2\n
            existing_line"
        ]
    }}
}}
```

Additional guidelines
- Provide ONLY the JSON response
- No additional explanation
- Response must be valid JSON (will be parsed with json.loads())
- If you want to think before returning response,
be short and concise

Related documentation
Here are supporting documentation pieces:
```markdown
```
```

```
{documentation}
```

Context files
Here are the context files to analyze:
{context_files}

Previous attempt
Previous solution, which needs fixing:
{solution_code}

Error message
Error message of the previous solution:
{error_message}
```

### E.5 Zero-shot Two-Turn Solver (RAG)

For this configuration, same prompt template from previous configuration is used for second turn. First turn template is different from the default two-turn solver configuration and allows in the same time to generate solution and keywords to search for in the documentation:

**First Turn Prompt**

```
Task overview
You are tasked with implementing a solution
based on the following prompt:
{example_prompt}

Context files
Here are the context files to analyze:
{context_files}

Steps
- Have a look at the version of aws-cdk-lib
    - Current version is {cdk_version}
- Read task overview ("# Task overview" section)
- Read each all of the context files ("# Context files" section)
- Try to understand what should be done
- If provided, read related AWS documentation
("# Related documentation" section)
- Provide solution with a specified format
("# Response format" section)
- Provide documentation support keywords

Documentation Support
To confirm your implementation you should provide keywords in
"search"
in key:
- Include a "search" key in your JSON response
- Provide up to 5 specific keywords
related to the AWS services
and features you need help with
- Example: "search": "data access policies opensearch"
    - Avoid using underscores or other similar
    special characters, query should be human-readable
    - Keywords should reflect the resources used in the stack
```
```

````
    or errors seen during execution,
    i.e. 'opensearch' or 'appsync'
    - Avoid too generic keywords like 'aws' or 'cloudformation'
- Keywords you provide will be used in the next interaction
in case solution does not pass the tests

# Response format
## General guidelines
Provide your solution as a JSON object with:
1. File paths as keys
2. Arrays of unified diffs as values

## Diff format specification
Each diff must follow this structure:
```
--- without_solution
+++ with_solution
@@ -line,count +line,count @@
 context lines
+added lines
 context lines
```

## Important rules:
- Only ADD content, do not remove any lines
- Include proper line numbers and context
- Ensure diffs are properly formatted

## Example JSON response:
```json

{{
    "path/to/file1.py": [
        "--- without_solution\n+++ with_solution\n
        @@ -10,3 +10,5
        @@\n     existing_line\n+    new_line1\n
        +    new_line2\n
        existing_line"
    ],
    "search": "..."
}}
```


## Additional guidelines
- Provide ONLY the JSON response
- No additional explanation
- Response must be valid JSON (will be parsed with json.loads())

------

Your JSON response (start with ```json):
````

F Model Configurations

Table 5 overviews the models used in our experiments and provides their invocation parameters. Some reasoning models (DeepSeek R1, OpenAI o3, OpenAI o4-mini) have **4 times bigger token budget**, Gemini 2.5 models require **5 times more**, reaching more than 20K maximum tokens per task. This was done to allow for longer outputs for reasoning models to avoid result truncation.

Table 5: Models Invocation Parameters

Model	Model ID	Invocation Parameters
Claude 3 Haiku	anthropic.claude-3-haiku-20240307-v1:0	max_tokens=4096; temperature=0.25; top_p=0.9
Claude 3.5 Sonnet	anthropic.claude-3-5-sonnet-20240620-v1:0	max_tokens=4096; temperature=0.25; top_p=0.9
Claude 3.5 Sonnet V2	anthropic.claude-3-5-sonnet-20241022-v2:0	max_tokens=4096; temperature=0.25; top_p=0.9
Claude 3.7 Sonnet	claude-3-7-sonnet-latest	max_tokens=4096; temperature=0.25
Codestral	codestral-latest	max_tokens=4096; temperature=0.25
DeepSeek R1	deepseek-ai/DeepSeek-R1	max_tokens=16384; temperature=0.25
GPT-4.1	gpt-4.1	max_output_tokens=4096; temperature=0.25
GPT-4o Mini	gpt-4o-mini	max_output_tokens=4096; temperature=0.25
Gemini 2.0 Flash	gemini-2.0-flash	max_output_tokens=4096; temperature=0.25
Gemini 2.0 Flash Lite	gemini-2.0-flash-lite	max_output_tokens=4096; temperature=0.25
Gemini 2.5 Pro (03-25 Preview)	gemini-2.5-pro-preview-03-25	max_output_tokens=20480; temperature=0.25
Gemini 2.5 Pro (05-06 I/O Edition Preview)	gemini-2.5-pro-preview-05-06	max_output_tokens=20480; temperature=0.25
LLaMA 3.1 405B Instruct	meta.llama3-1-405b-instruct-v1:0	max_tokens=4096; temperature=0.25; top_p=0.9
LLaMA 3.1 70B Instruct	us.meta.llama3-3-70b-instruct-v1:0	max_tokens=4096; temperature=0.25; top_p=0.9
LLaMA 4 Maverick 17B Instruct	meta.llama4-maverick-17b-instruct-v1:0	max_tokens=4096; temperature=0.25
LLaMA 4 Scout 17B Instruct	meta.llama4-scout-17b-instruct-v1:0	max_tokens=4096; temperature=0.25
Mistral Large	mistral-large-latest	max_tokens=4096; temperature=0.25
OpenAI o3	o3	max_output_tokens=16384
OpenAI o4-mini	o4-mini	max_output_tokens=16384
Qwen2.5 72B Instruct Turbo	Qwen/Qwen2.5-72B-Instruct-Turbo	max_tokens=4096; temperature=0.25

G Verbosity Configurations for Two-Turn Solvers

Our experimental framework uses pytest as the testing library, with two distinct verbosity configurations designed to evaluate how different levels of feedback detail affect model performance. Here are the details on each of the configurations:

Table 6: Verbosity Parameters

	Low-verbosity (V_L)	**High-verbosity (V_H)**
Configuration		
Flags	`-q -tb=no -no-summary`	$\emptyset$
Output Features		
Pass/fail counter	✓	✓
Exception	✓	✓
Traceback	×	✓
Test files names	×	✓
Test functions names	×	✓

H Models Performance by Donor Repository Type

We analyzed the performance of models across different repository sources to investigate potential biases in our benchmark. The SWE-InfraBench dataset comprises 100 tasks derived from 34 distinct base repositories, with 66 examples originating from open-source repositories and 34 from custom-developed sources. While each task underwent substantial engineering modifications regardless of its origin, we sought to determine whether tasks derived from open-source repositories might be

less difficult compared to custom-developed ones due to models being potentially trained on the open-source data.

As shown in Table 7 for the majority of models the tasks derived from open-source repositories are equally or even more challenging. One important consideration is that randomness in the generation process and limited group sizes (34 tasks in the smallest group) suggest caution in the results interpretation.

Table 7: Correctness of proprietary and open-source LLMs on the whole benchmark and separately for tasks created from open-source and from custom created repositories. Models are grouped by source type and provider, and sorted from most recent to oldest variant within each group. † indicates LLMs executed with reasoning capabilities. ‡ Claude 3.7 was executed without extended reasoning. Top results are in **bold**.

Source	Company	Model	Correctness		
			All Tasks	Custom Derived Tasks	Open-Source Derived Tasks
Proprietary	Anthropic	Claude 3.7 Sonnet ‡	**34%**	35%	**33%**
		Claude 3.5 Sonnet V2	32%	32%	32%
		Claude 3.5 Sonnet	29%	32%	27%
		Claude 3 Haiku	8%	12%	6%
	Google	Gemini 2.5 Pro (03-25 Preview) †	29%	**44%**	21%
		Gemini 2.5 Pro (05-06 I/O Edition Preview) †	29%	38%	24%
		Gemini 2.0 Flash Lite	5%	6%	5%
		Gemini 2.0 Flash	4%	0%	6%
	OpenAI	OpenAI o3 †	23%	29%	20%
		OpenAI o4-mini †	23%	26%	21%
		GPT-4.1	18%	18%	18%
		GPT-4o Mini	4%	9%	2%
Open-Source	DeepSeek	DeepSeek R1 †	24%	26%	23%
	Mistral	Mistral Large	14%	12%	15%
		Codestral	9%	15%	6%
	Meta	LLaMA 3.1 405B Instruct	9%	12%	8%
		LLaMA 3.1 70B Instruct	3%	6%	2%
		LLaMA 4 Maverick 17B Instruct	8%	9%	8%
		LLaMA 4 Scout 17B Instruct	2%	3%	2%
	Alibaba	Qwen2.5 72B Instruct Turbo	0%	0%	0%

I Consistency Analysis

We quantified consistency using a "Success Consistency" metric. For each model and example, the "Task Success Consistency Gap" was calculated as the difference between the best and worst attempt correctness:

$$\text{Task Success Consistency Gap} = \max_{i \in \text{attempts}} (\text{correctness}_i) - \min_{i \in \text{attempts}} (\text{correctness}_i) \tag{1}$$

Success Consistency is then defined as:

$$\text{Success Consistency} = 1 - \text{Task Success Consistency Gap} \tag{2}$$

This value is averaged over all examples for each model. A value of 1 means perfectly consistent performance across attempts (either always failing or always succeeding). A value of 0 indicates maximum inconsistency (some attempts succeed while others fail).

As indicated in Section 4 all LLMs demonstrate substantial improvement when given multiple opportunities to solve a task. Gemini 2.5 Pro achieves the highest pass@5 rate, indicating that for 47% of tasks, at least one of five attempts produces a fully correct solution. This represents a considerable improvement over its single-attempt performance. DeepSeek R1 and OpenAI o3 show similar patterns, with pass@5 rates consistently higher than pass@1. Claude 3.7 Sonnet, however, shows higher average correctness of the results. It explains how Claude 3.7 Sonnet with its slightly lower pass@5 of 41% achieves the top position in one-trial benchmarking.

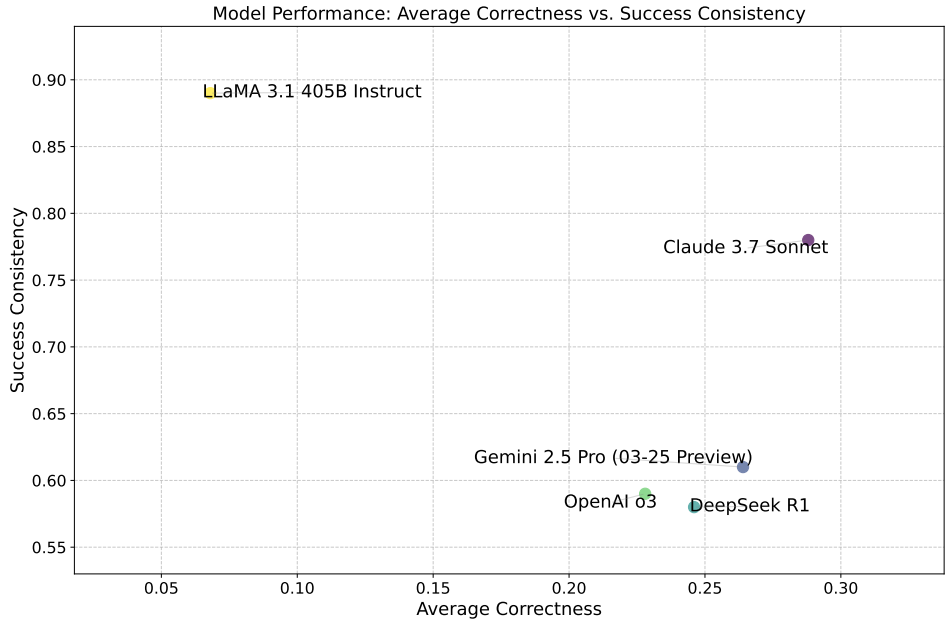

Figure 7: Average Correctness vs Consistency. Models positioned higher on average correctness, notably Claude 3.7 Sonnet, tend to demonstrate greater consistency, providing similar outcomes across repeated attempts. In contrast, models such as Gemini 2.5 Pro, DeepSeek R1, and OpenAI o3, despite achieving significant improvements when allowed multiple attempts, exhibit less consistency, indicating more variability in their success across trials.

Figure 7 illustrates the relationship between average correctness across trials and success consistency for tested models. We observed that higher-performing on average Claude 3.7 Sonnet demonstrated better consistency, while Gemini 2.5 Pro, DeepSeek R1 and OpenAI o3 are not as consistent in providing correct results on this dataset.

J Error Type Distribution

Figure 8 illustrates how error distributions change when models are given a second attempt with feedback (high verbosity configuration) from their first attempt. All models benefit from the second attempt, improving both on syntax and logical errors. However, a subset of LLMs without reasoning, namely GPT-4.1, Mistral Large and Llama 4 Maverick 17B Instruct, retain the same percentage of logical errors, mostly correcting the syntax ones. Such a behavior indicates that even high verbosity level is insufficient to address these more complex, implementation-specific failures. On the other hand, Claude Sonnet models, as well as Gemini 2.5 and DeepSeek R1, that further leverage their reasoning capabilities, showcase the considerable improvement on this matter as a prove of better analysis of underlying infrastructure dependencies and relationships.

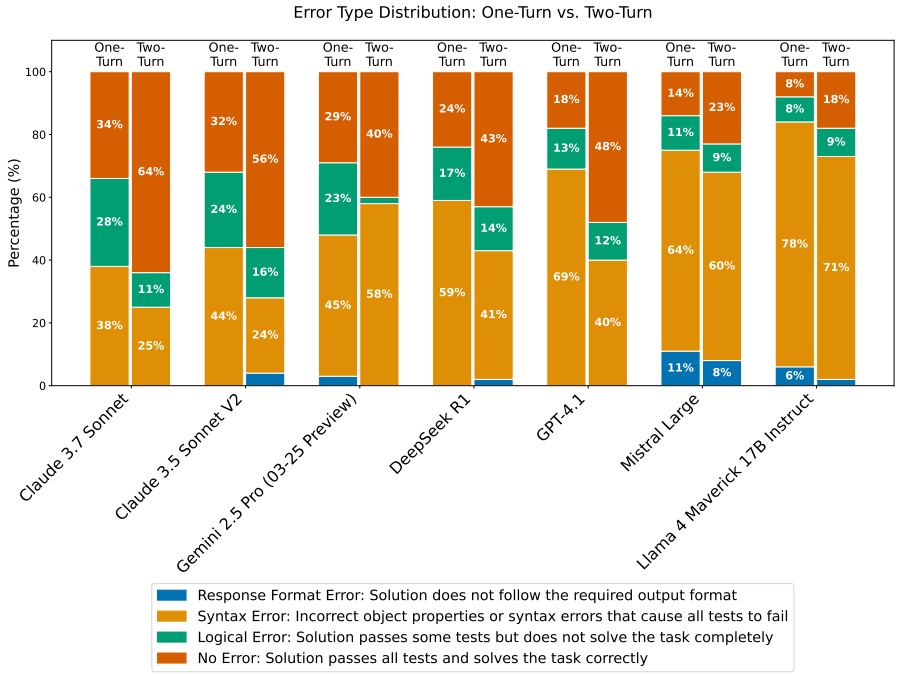

Figure 8: Error type distribution comparison between one-turn and two-turn (without RAG, high verbosity) approaches. All models show improved performance in the two-turn approach. The Claude family models demonstrate balanced improvement by reducing both syntax and logical errors. Gemini model primarily addresses logical errors in its second attempt, while GPT-4.1 and DeepSeek R1 shows substantial reduction in syntax errors. While most examples benefit from the two-turn approach, a small percentage show regression due to the inherent randomness in the generation process.