Accepted at ACL 2026

Large Language Models for IT Automation Tasks: Are We There Yet?

Md. Mahadi Hassan
University of Central Florida
mdmahadi.hassan@ucf.edu

John Salvador
University of Central Florida
johnsalvador@ucf.edu

Akond Rahman
Auburn University
azr0154@auburn.edu

Santu Karmaker
University of Central Florida
santu@ucf.edu

Abstract

LLMs show promise in code generation, yet their effectiveness for IT automation tasks, particularly for tools like Ansible, remains understudied. Existing benchmarks rely primarily on synthetic tasks that fail to capture the needs of practitioners who use IT automation tools. We present ExITBench (Execution-based IT Automation Benchmark), a benchmark of 126 diverse tasks (e.g., configuring servers and managing files) in which each task captures state reconciliation—a core property of IT automation tools. ExITBench evaluates LLMs' ability to generate functional Ansible automation scripts via dynamic execution in controlled environments. We evaluate 14 open-source and 3 proprietary LLMs and find that GPT-4.1-Mini achieves the best pass@10 rate of 23.9%, while Claude-3.5-Sonnet achieves the best pass@1 performance. To explain the low performance, we analyze 1,517 execution failures across the evaluated LLMs and identify two prevalent semantic error categories: failures in state-reconciliation reasoning (42.117% combined from variable (12.287%), host (10.363%), path (10.511%), and template (8.956%) issues) and deficiencies in module-specific execution knowledge (26.203% combined from attribute & parameter (17.617%) and module (8.586%) errors). Our findings reveal key limitations in LLMs' ability to address state reconciliation and apply specialized module knowledge, indicating that reliable IT automation with LLM-based agents need major advances in state reasoning and domain-specific execution.

1 Introduction

DevOps practitioners rely on configuration management tools like Ansible for IT automation tasks. In order to complete these tasks, practitioners use scripts, which are referred to as automation scripts (Parnin et al., 2017). While these scripts save time and manage thousands of servers (ansible, 2022), practitioners still struggle to develop them correctly for intended IT automation tasks (Begoug et al., 2023). These challenges stem from (i) domain-specific languages with distinct syntax and semantics (Rahman et al., 2020), (ii) diverse IT automation tasks across operating systems and cloud platforms (Begoug et al., 2023), and (iii) state reconciliation, which demands accurate infrastructure assessment and regulation (Hassan et al., 2024). Unsurprisingly, these challenges have sparked practitioner frustration and concern (Tanzil et al., 2023; NFsaavedra, 2024).

Given the success of large language models (LLMs) in code generation (Li et al., 2024a; Chen et al., 2021), we hypothesize that they are well-suited for automating IT tasks through automation scripts. For automation scripts, it is not enough to generate syntactically correct code; models must also accurately interpret task requirements and produce scripts that achieve the desired system state to ensure successful execution (Hassan et al., 2024). Even minor errors such as using an incorrect module or misplacing a variable, can lead to catastrophic security risks or system failures. While benchmarks (Chen et al., 2021; Iyer et al., 2018; Odena et al., 2021) have spurred progress in code generation, they emphasize static correctness and overlook a critical question: *Can LLM-generated code actually provide executable solutions for IT automation tasks*? Indeed, existing approaches often lack dynamic execution testing or operate in constrained settings—for example, Ansible Wisdom (Pujar et al., 2023) relies on BLEU scores, Ansible Lightspeed (Hat, 2023) focuses on isolated tasks, and IaC-Eval (Kon et al., 2024) uses artificially generated configurations curated by human annotators. As a result, LLMs' ability to generate robust, executable scripts for real-world IT automation tasks remains under-explored.

To address this gap, we present **ExITBench**, a benchmark for evaluating LLMs on their abil-

ity to generate *executable* automation scripts from real-world, user-authored natural language prompts. ExITBench includes **126** tasks, carefully selected across seven key IT automation tasks (Begoug et al., 2023): (1) *Server Configuration*, (2) *Networking*, (3) *Policy Configuration*, (4) *Templating*, (5) *Deployment Pipelines*, (6) *Variable Management*, and (7) *File Management* (details in Section 3.1). Each task must satisfy specific *operational constraints* that reflect the intended system state described by the user.

ExITBench is different from previous work in multiple ways. *First*, benchmarks like IaC-Eval (Kon et al., 2024) assess whether generated code aligns with infrastructure intent specifications, while ExITBench focuses on functional correctness via dynamic execution of automation scripts in realistic environments—marking a clear contrast with static-based approaches. *Second*, To ensure reliable execution, we enrich LLM prompts with key context—such as file paths, configurations, and initial states—bridging the gap between vague instructions and executable scripts. *Third*, ExITBench specifically evaluates how LLMs handle state reconciliation-a fundamental property of IT automation tools like Ansible (Hassan et al., 2024) where the orchestrator infers desired states from scripts, compares them with current states, and applies only necessary changes (Rahman and Parnin, 2023). ExITBench tests this capability by validating tasks in controlled environments with predefined initial states, verifying if the resulting system state matches user requirements.

Using our test suite, we evaluated 17 LLMs–14 open-source and three proprietary–by varying prompt specificity (TELeR levels (Santu and Feng, 2023)) and sampling temperature (Section 3.4). Overall, success rates in achieving the desired system state were strikingly low, particularly on the first attempt (`pass@1`), with *Templating* and *Variable Management* standing out as the most challenging IT automation categories.

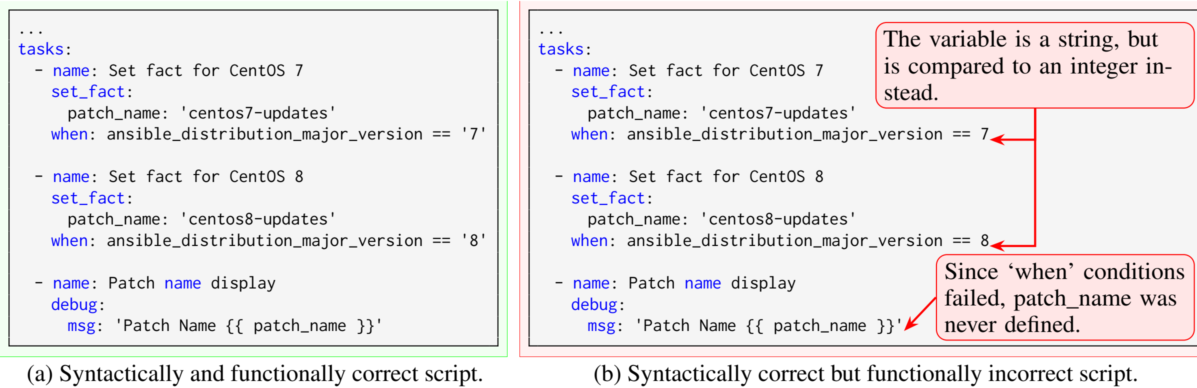
Analysis of 1,517 execution failures—cases where LLM-generated scripts failed to reach the intended system state—reveals two fundamental error categories: state reconciliation reasoning failures (42.117%), where models struggle to track and manage system state across tasks, and gaps in module-specific execution knowledge (26.203%). Additionally, we observe sampling trade-offs: higher temperatures improve diversity and `pass@10` specially for complex cases, while lower temperatures enhance reliability and `pass@1`. Our error taxonomy provides insights for improving LLMs in IT automation and advancing research on executable reasoning. Our contributions are:

1. **A Benchmark for IT Automation Tasks based on Execution:** ExITBench not only evaluates syntax, but also evaluates operational correctness, using real-world IT automation tasks with execution-based validation. Artifacts used to construct ExITBench are available online (pasergroup, 2023).
2. **Error Taxonomy:** We identify nine specific error categories in LLM-generated automation scripts—variable issues, host issues, path issues, attribute configuration, template errors, logic & compliance problems, module errors, output format errors, and syntax errors—establishing a standard taxonomy to reveal fundamental gaps in LLMs' ability to track state and follow instructions.

2 Related Works

Large Language Models (LLMs) are widely evaluated on benchmarks (Chen et al., 2021; Odena et al., 2021; Iyer et al., 2018), which focus on static code generation but overlook real-world executability. Enhanced benchmarks (Yu et al., 2024b; Zhuo et al., 2024a; Jimenez et al., 2024; Yang et al., 2025; Lai et al., 2023; Zheng et al., 2025; Li et al., 2024b; Xie et al., 2024; Zhu et al., 2025; Peng et al., 2025) introduce more realistic tasks, while multilingual evaluations (Peng et al., 2024; Awal et al., 2025; Luo et al., 2025) test across languages. Semantic parsing benchmarks (Long et al., 2016; Yu et al., 2018; Yin et al., 2018a; Li et al., 2025; Yu et al., 2024a) assess natural language to code translation but often overlook system-level correctness in IT automation contexts.

Within the IT automation domain specifically, benchmarks like IaC-Eval (Kon et al., 2024) (using human-curated synthetic scenarios), WISDOM-Ansible (Pujar et al., 2023) (using BLEU scores instead of execution), ITBench (Jha et al., 2025) (using hypothetical persona-based tasks) and others (Khan et al., 2025; Srivatsa et al., 2024; Scheuner et al., 2014; Ragothaman and Udayakumar, 2024; Hat, 2023) evaluate LLMs on infrastructure tasks but do not include tasks that corresponds to IT automation queries made by real-world practitioners. This gap is particularly problematic because IT automation requires grounding language in exe-



(a) Syntactically and functionally correct script. (b) Syntactically correct but functionally incorrect script.

Figure 1: Two syntactically correct code snippets used in Ansible-based IT automation where one is functionally correct. Evaluating syntactic correctness is not enough to determine an LLM's capability for solving IT automation tasks, which requires an execution-based benchmark.

cutable actions and system state transitions. *State reconciliation* is fundamental to tools like Ansible, where automation tools infer desired states, compare with current states, and apply necessary changes (Rahman and Parnin, 2023; Hassan et al., 2024). While existing studies (Wang et al., 2024a; Anandayuvaraj et al., 2024; Dou et al., 2024; Wang et al., 2024b; Chen et al., 2024) examine LLM failure patterns, they overlook challenges in reasoning about environment state and module-specific execution knowledge in IT automation.

ExITBench addresses these gaps with executable IT automation tasks from real-world issues, enabling dynamic LLM evaluation and introducing an error taxonomy that exposes key failures by LLMs in handling state reconciliation.

3 Methodology

To effectively evaluate LLM-generated code for real-world IT automation needs, ExITBench uses Ansible, an open-source tool where users write YAML playbooks to define the desired state of a system using task-specific modules. This state-based approach matters because syntactic correctness alone doesn't guarantee correct execution. Hence, ExITBench tasks require LLMs to generate playbooks that reliably achieve a specific target system state using state reconciliation.

3.1 Task Collection and Curation

To build an execution-focused benchmark that truly tests the ability to achieve specific system states, we started with a corpus of 52,727 Stack Overflow posts on IT Automation (Begoug et al., 2023), from which all personally identifiable information, such as user IDs and email addresses was excluded. Sourced for its real-world scenarios and inherent natural language ambiguities (Yin et al., 2018b), this corpus was refined into high-quality executable tasks through a rigorous, two-step Data Curation Phase. First, we performed an expert-driven, purposive **stratified sampling**. To ensure a diverse benchmark, we categorized the initial corpus into seven key IT automation domains based on prior analysis. Our team then manually reviewed posts within each domain to select a proportionate set of "Sampled Issues" that best aligned with observed distributions. Second, The "Sampled Issues" then underwent a final **manual curation and filtering** stage. During this phase, each potential task was assessed against several criteria: its relevance to core Ansible functionalities (e.g., involving common modules like ansible.builtin and community.general), the clarity of the user's intent, and the feasibility of implementation and validation within our defined environment.

This curation process resulted in 126 executable tasks that constitute the Execution-based IT Automation Task Benchmark (ExITBench). These tasks span seven core domains of IT automation, aligning with common Ansible use cases (Begoug et al., 2023). The distribution is as follows: Server Configuration (27), File Management (18), Variable Management (18), Policy Configuration (17), Networking (16), Deployment Pipelines (15), and Templating (15).

To mitigate the risk of data leakage, our benchmark design explicitly ensures that simple memorization is insufficient for success. Tasks are curated from problem descriptions, not from accepted code solutions. Crucially, functional correctness is evaluated via execution inside our controlled, benchmark-specific Docker environment, which uses custom hostnames (ubuntu1, alpine1, centos1, redhat1), pre-defined file paths, initial system states, and a synthetic multi-node subnet

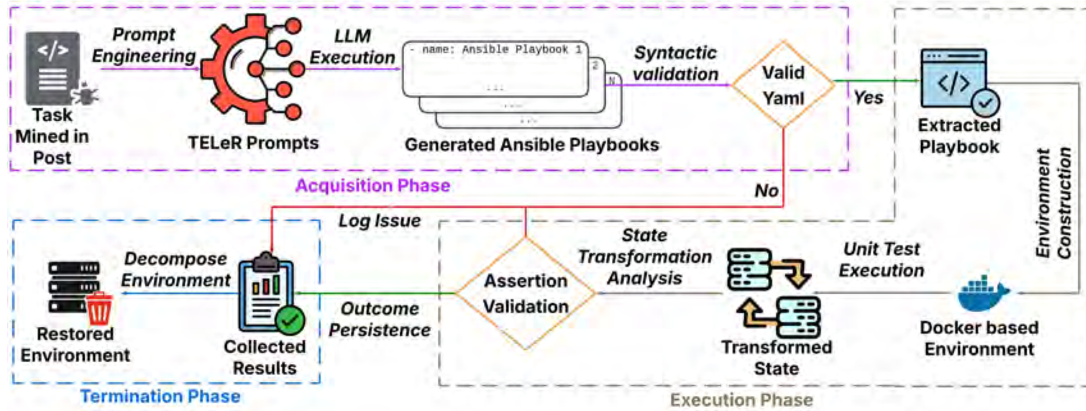


Figure 2: The ExITBench evaluation pipeline, showing different phases for testing LLM-generated Ansible code.

(10.1.1.0/24). A model that verbatim reproduces a memorized Stack Overflow answer would therefore very likely fail our state-based assertions, since those assertions verify the resulting system state rather than the surface-form similarity to a reference solution. We acknowledge that residual contamination risk cannot be fully eliminated, as models may generalize from seen patterns; this is discussed further in the Limitations (Section 9). A detailed breakdown of our data sourcing, filtering criteria, manual validation process, and data leakage mitigation is provided in Appendix A.

3.2 Execution-focused Test Case Development

We transform the curated tasks into executable test cases with validation of whether LLM-generated code achieves the intended system state. Here, the system state is defined by operational constraints on elements like file contents, service status, and configuration values. For each task, we implemented the following process:

1. **Context Analysis**: We identified the implicit system requirements, dependencies, and environmental constraints from the original question and answers. This involved careful examination of both the question text and accepted solutions to extract the underlying automation intent, often requiring domain expertise to interpret implied requirements not explicitly stated.
2. **State Definition**: For each task, we formalized both an initial (pre-automation) and target (desired post-execution outcome) system state. This required translating ambiguous user requirements into precise, verifiable configurations that could be automatically validated.
3. **Parameter Identification**: We extracted key variables that might affect execution outcomes, such as file paths, service names, configuration values, and host-specific settings. This step was crucial for ensuring that our test cases could properly evaluate how LLMs handle variable substitution, path resolution, and template rendering-core capabilities for effective IT automation.
4. **Determining Functional Correctness**: For each task, we defined assertions to verify state reconciliation by checking file contents, service status, and configuration values, ensuring the automation goal is met. Figure 1 illustrates the distinction between syntactic and functional correctness in Ansible automation. While both scripts pass syntax validation, the right implementation fails because it incorrectly compares a string variable with integers and subsequently references an undefined variable. The left implementation correctly handles state reconciliation by using proper string comparison and default values, demonstrating why execution-based validation is essential.

To ensure consistent testing, we containerized each scenario with precisely controlled initial states and task-specific verification scripts. This development—led by authors with Ansible and Python expertise—produced the “Test Case Collection” for **ExITBench**, comprising **733 test cases** across **126 tasks**. Our approach detects subtle state reconciliation failures missed by static analysis or basic logging. Using robust test cases, we built a testing framework and execution pipeline to systematically evaluate LLM-generated solutions.

3.3 Testing Framework & Execution Pipeline

To assess operational correctness, we use a **Dynamic Validation Process** shown in Figure 2 that executes each generated playbook and checks it against task-specific constraints defined in Section 3.2. The pipeline consists of three phases:

Acquisition Phase: This phase selects an automation issue from the test case collection and uses TELeR prompts (Section 3.4) to generate can-

didate playbooks via LLMs. Playbooks are then validated for YAML and Ansible syntax; Invalid scripts are logged and skipped, while valid ones proceed to execution.

Execution Phase: For each valid playbook, this phase starts by setting up a task-specific, isolated Docker environment. The playbook runs within this environment, triggering a system state transformation. Custom Python validation scripts (Section 3.2) then analyze the resulting state to verify whether the operational constraints were met, which determines the final Pass or Fail outcome.

Termination Phase: This phase finalizes the evaluation, recording the Pass/Fail status, logs, and errors. After each task, the Docker environment is reset to its default state.

3.4 Experimental Setup

Using the ExITBench benchmark and evaluation pipeline, we evaluated 17 LLMs—14 open-source (Table 7) and 3 proprietary (GPT-4.1-Mini, Claude-3.5-Sonnet, and Gemini-2.5-Flash-Lite)—on their ability to generate correct Ansible playbooks across 126 real-world tasks. The open-source models span recent state-of-the-art systems in the 3B–15B range, chosen for strong performance on established code benchmarks (Zhuo et al., 2024b). This size range offers a balance between competitiveness and computational accessibility, supporting reproducibility with available GPU resources (Hasan et al., 2025). We also explored Ansible Lightspeed (Sahoo et al., 2024), a specialized Ansible-focused model, but it lacks a suitable API for scientific use and requires a paid license and extra tooling, preventing its integration in this work.

Two decoding parameters were systematically varied: **TELeR prompt levels** and **sampling temperature**. TELeR Levels 1–3 (Santu and Feng, 2023) adjusted prompt specificity; higher levels encode more structured task descriptions. We focused on these levels due to the lack of few-shot examples and external documents for higher-level prompting. Sampling temperature (Ackley et al., 1985) was set at 0.2, 0.4, 0.6, and 0.8 to balance deterministic and exploratory behavior. For each (model, task, TELeR level, temperature) combination, we generated 15 scripts for robust performance estimation.

Generated playbooks were evaluated using the **pass@k** metric ($k \in 1, 3, 5, 10$)(Chen et al., 2021). A sample was considered successful only if it was syntactically valid, executed correctly in our test framework (Section3.3), and met the expected outcome. All experiments were run on uniform hardware (NVIDIA H100 GPUs).

4 Results and Analysis

4.1 Overall Performance Across Models

Our evaluation (Table 1) reveals a significant performance gap: the proprietary Claude-3.5-Sonnet achieves the highest Pass@1 score (18.8%) among all evaluated models, exceeding GPT-4.1-Mini's 14.7% on single-attempt success, suggesting greater reliability under greedy decoding. However, GPT-4.1-Mini surpasses Claude-3.5-Sonnet on Pass@10 (23.9% vs. 22.2%), indicating a higher diversity ceiling when given multiple attempts. This pattern suggests that while Claude-3.5-Sonnet may generate more consistently correct outputs on the first try, GPT-4.1-Mini benefits more from sampling exploration. GPT-4.1-Mini and Claude-3.5-Sonnet together establish a strong upper-bound reference well above the best open-source model, Qwen2.5-Coder-7B-it (12.0 Pass@10), confirming a significant capability gap.

Model	@1	@3	@5	@10
GPT-4.1-Mini	14.7	19.9	**21.8**	**23.9**
Claude-3.5-Sonnet	**18.8**	**20.7**	21.4	22.2
Gemini-2.5-Flash-Lite	6.4	10.3	12.1	14.7
Qwen2.5-Coder-7B-it	3.5	6.7	8.7	12.0
DeepSeek-Coder-V2-it	3.1	5.5	6.7	8.5
WizardCoder-15B	3.1	3.4	3.6	3.7
Llama-3.1-8B-it	2.2	3.8	4.8	6.6
Phi-3.5-mini-it	2.2	3.4	4.1	5.0
Qwen2.5-7B-it	2.0	3.3	4.0	5.0
Codegemma-7B-it	1.3	2.5	3.3	4.6
CodeLlama-13B-it	1.0	1.6	2.1	3.0
Llama-3.2-3B-it	1.1	1.6	2.0	2.7
Starcoder2-7B	0.8	1.8	2.6	3.9
CodeLlama-7b-it	0.9	1.3	1.6	2.2
Vicuna-7B-it	0.5	1.1	1.5	2.3
DeepSeek-Distill-L	0.7	0.9	1.0	1.2
DeepSeek-Distill-Q	0.2	0.6	0.7	0.8

Table 1: $pass@k$ for selected LLMs on **ExITBench** (avg. over tasks, prompts, temps). Bold highlights indicate the best-performing model for each pass@k column.

Among the open-source models, pretraining data composition affects performance, with top models like Qwen2.5-Coder-7B (70% code/20% text (Hui et al., 2024; Yang et al., 2024)) and DeepSeek-Coder-V2 (60% code/30% text (Zhu et al., 2024)) combines code and language input—key for understanding prompts and generating correct Ansible. Qwen2.5-Coder-7B's edge over its base model underscores how code specialization benefits from strong natural language grounding.

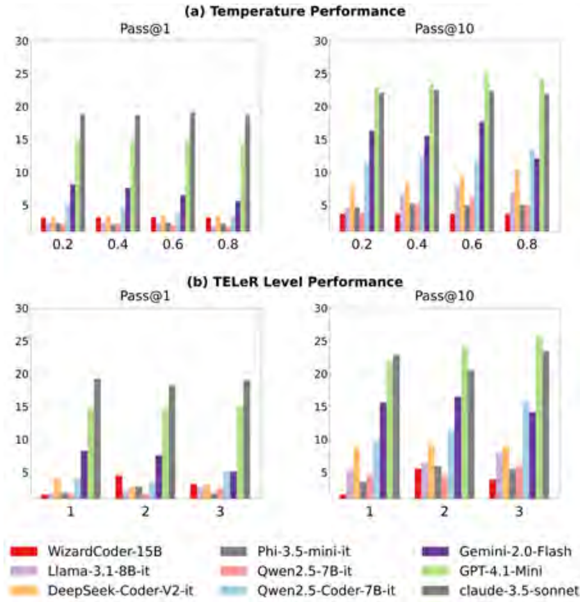


Figure 3: Impact of decoding parameters for the top 9 LLMs. (a) Pass@1 and Pass@10 across temperature settings. (b) Pass@1 and Pass@10 across TELeR levels.

The general-purpose Llama-3.1-8B (15T+ tokens (Dubey et al., 2024)) outperformed code-specialized CodeLlama models (e.g., 13B, 85% code/500B tokens (Rozière et al., 2023)), suggesting massive scale and language understanding can be more vital than high code ratios for instruction-grounded IT automation tasks. These trends hint that success in ExITBench might be influenced not only by code volume but also by the interplay of code exposure, general language understanding, and dataset scale.

4.2 Impact of Temperature and Prompting

Beyond overall success rates, this section analyzes how generation configurations—specifically sampling temperature and prompt detail—influence LLM performance on IT automation tasks.

Sampling Temperature: Sampling temperature impacted performance, revealing a trade-off. As shown in Figure 3 (a), lower temperatures (e.g., 0.2) maximized pass@1 scores by favoring reliable, deterministic outputs. Conversely, higher temperatures (0.6–0.8) boosted pass@10 by increasing output diversity, aiding discovery in complex tasks. This highlights a practical precision-versus-exploration dilemma for tuning generation.

Prompt Detail (TELeR Levels): While Figure 3 (b) shows higher TELeR levels often boost Pass@10 accuracy for capable models like GPT-4.1-Mini by leveraging richer input, increased prompt detail is not uniformly advantageous. It can also result in overly complex or 'over-engineered' solutions (Section 6), with the overall benefit varying by model capability.

Model	@1	@3	@5	@10
Qwen2.5-Coder-7B-it	**5.6**	**9.8**	**12.1**	**15.5**
Phi-3.5-mini-it	**4.2**	**6.3**	**7.3**	**8.3**
DeepSeek-Coder-V2-it	3.3	5.5	6.6	8.3
Llama-3.1-8B-it	2.8	4.8	5.9	7.5
Codegemma-7B-it	2.6	4.4	5.4	7.0
Starcoder2-7B	0.8	1.9	2.6	3.8
Vicuna-7B-it	0.5	1.1	1.4	2.0
WizardCoder-15B	2.7	2.9	3.2	3.4
CodeLlama-13B-it	2.7	4.9	6.1	8.1

Table 2: pass@k with error-avoidance prompts (avg. across configs).

Error-Aware Prompting: Given the performance gap between proprietary and open-source models, we investigated whether error-aware prompting could bridge this gap by specifically targeting the common failure patterns of the open-source models. We used prompts informed by our failure taxonomy (Section 5), including hints against common mistakes. However, as shown in Table 2, this intervention yielded only marginal pass@k improvements (1-4 percentage points) indicating that prompt modifications alone poorly mitigate models' core challenges in state reconciliation reasoning and module-specific knowledge.

4.3 Performance by IT Automation Domains

Figure 4 analyzes LLM performance across IT automation domains, with rows representing domains, columns as model families, and color intensity indicating pass@1 or pass@10 rates.

While models performed better on routine tasks like Deployment Pipeline and File Management, categories requiring precise state and variable handling—notably Templating and Variable Management—proved far more challenging, exhibiting the lowest pass@k scores. This suggests these state-sensitive domains demand reasoning beyond basic syntax or module use, aligning with our error analysis (Section 5) where difficulties in state reconciliation related issues are prevalent.

4.4 Throughput and Latency Analysis

To address practitioner concerns about accuracy costs, we conducted a throughput and latency analysis. Results show a clear accuracy-performance trade-off. For instance, GPT-4.1-Mini—the most accurate model—has a generation latency of 4.09s, over twice that of the faster Gemini-2.5-Flash-Lite at 1.8s, highlighting the practical cost of higher accuracy. Full performance metrics for all models are in Table 8 (Appendix F).

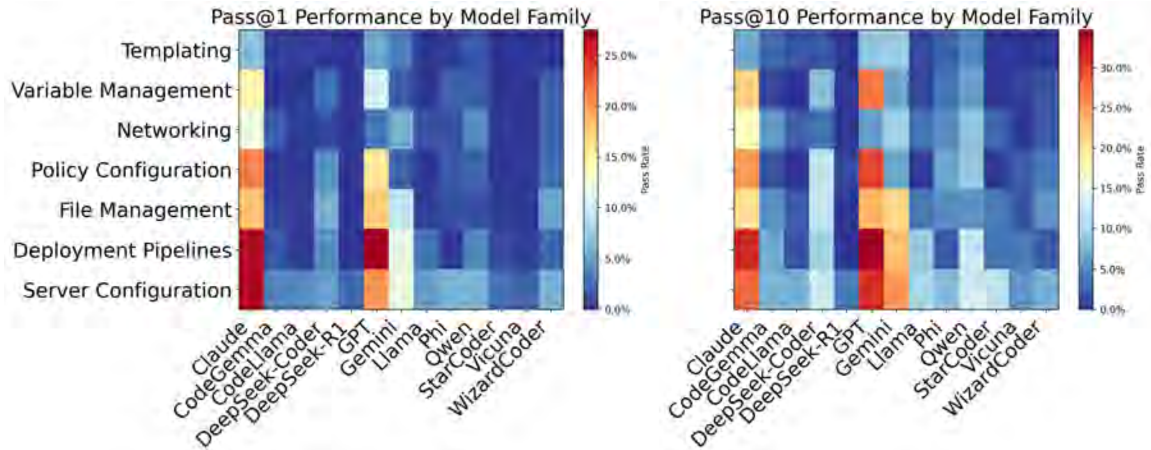


Figure 4: Pass@1 and Pass@10 Data for IT Automation Tasks Across Model Families

Model	Variable	Host	Path	Attr. & Param.	Template	Logic & Compliance	Module	Output	Syntax
GPT-4.1-Mini	8.51	1.06	15.96	**28.72**	12.77	1.06	9.57	2.13	20.21
Claude-3.5-Sonnet	12.32	7.25	0.72	**34.06**	0.00	19.57	0.00	23.91	2.17
Gemini-2.5-Flash-Lite	**23.66**	9.68	6.45	12.90	10.75	9.68	5.38	3.23	18.28
Codegemma-7B-it	**21.95**	3.66	24.39	18.29	10.98	4.88	4.88	2.44	8.54
CodeLlama-7B-it	19.44	8.33	15.28	6.94	**23.61**	5.56	8.33	4.17	8.33
CodeLlama-13B-it	8.65	9.62	10.58	10.58	4.81	3.85	5.77	0.96	**45.19**
DeepSeek-Coder-V2-it	1.69	**32.20**	20.34	10.17	10.17	3.39	13.56	0.00	8.47
Llama-3.1-8B-it	3.30	13.19	10.99	10.99	4.40	7.69	17.58	5.49	**26.37**
Llama-3.2-3B-it	13.04	16.30	8.70	14.13	3.26	3.26	15.22	0.00	**26.09**
Phi-3.5-mini-it	**26.09**	8.70	15.22	16.30	14.13	6.52	10.87	0.00	2.17
Qwen2.5-7B-it	7.22	8.25	10.31	**22.68**	5.15	7.22	19.59	2.06	17.53
Qwen2.5-Coder-7B-it	13.51	7.21	11.71	**19.82**	8.11	0.90	5.41	17.12	16.22
Starcoder2-7B	4.94	0.00	6.17	**25.93**	11.11	11.11	13.58	1.23	28.40
Vicuna-7B	10.42	8.33	0.00	10.42	10.42	14.58	4.17	0.00	**41.67**
WizardCoder-15B	7.37	28.42	6.32	7.37	14.74	1.05	0.00	4.21	**30.53**
DeepSeek-Distill-L	0.00	1.22	1.22	6.10	1.22	3.66	4.88	0.00	**81.71**
DeepSeek-Distill-Q	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	**100.00**
Aggregation	**12.287**	**10.363**	**10.511**	**17.617**	**8.956**	**6.810**	**8.586**	**5.551**	**19.319**

Table 3: Distribution of Error Types (%) Across Models. Bold values indicate the most frequent error category per model. To avoid skew, syntax-heavy error distributions from reasoning-distilled models are excluded from the aggregated percentages.

5 Error Taxonomy

To understand what lies behind the low pass@k values, we conducted a qualitative analysis of 1,517 execution failures across all models, developing a taxonomy of errors in LLM-generated automation scripts for real-world tasks.

Error analysis reveals that reasoning-distilled models often fail at basic syntax, suggesting that general reasoning doesn't easily transfer to structured domains like IT automation without explicit domain grounding. Other LLMs mostly fail beyond syntax, with errors clustering into two core semantic categories. First, **module-specific execution gaps** are common. Models often identify the right Ansible modules but misapply them, frequently producing Attribute or Parameter Errors. This suggests they know what action is needed but not how to configure it correctly. Second, widespread **state-reconciliation reasoning failures** are a defining challenge across nearly all evaluated models. These include variable and path handling errors, reflecting challenges in tracking state across playbooks, roles, and templates. Models also struggle with Jinja2 template logic, showing difficulty in reasoning about dynamic content generation even when the correct module is identified. The distribution of these state-related errors across variables, hosts, paths, and templates varies across model architectures, suggesting different blind spots in contextual reasoning.

This analysis reveals key gaps from basic syntax issues in some models to deeper failures in state reasoning and module-specific knowledge, explaining the low pass@k performance on executable IT

automation tasks and pointing to clear targets for future improvement.

Error Type	T=0.2	T=0.8
Attribute type mismatch	15	15
Invalid YAML	13	9
Wrong host name	3	7
Conflicting action	5	1
Variable issue	0	1
Wrong attribute	0	1
Template error	0	1
Missing hosts field	0	2

Table 4: Error diversity with low and high temperatures for the *targeted host execution* task.

6 Qualitative Examples

To complement the quantitative results, we qualitatively analyze Qwen2.5-Coder-7B-it—the top-performing open-source model in our evaluation—on representative tasks, revealing trade-offs induced by decoding parameter choices.

```
...
tasks: ...
- name: Send email if any host changes
  mail:
    host: localhost
    ...
  when: mail_result.rc == 0 and
      ansible_check_mode
```

Listing 1: Use of Check Mode to Control Email Sending

In the `targeted host execution` task (centos1), higher sampling temperatures increased both output diversity and error variety, with the number of distinct error categories rising from four at $T = 0.2$ to eight at $T = 0.8$ (Table 4). Temperature also affected correctness: in the `check_mode` task, only samples generated at $T = 0.8$ correctly applied `check_mode` (Listing 1), whereas lower temperatures did not yield any correct instances. In the *multi-server directory setup* task, TELeR Level 3 prompts produced functionally correct but overengineered solutions, including redundant file creation and validation steps that could reduce maintainability.

```
- name: Ensure base directory exists
  file:
    path: /tmp/servers
    state: directory

- name: Create server directories
  file:
    path:''/tmp/servers/server{{item}}''
    state: directory
  loop: ''{{ range(1, 4) | list }}''
```

Listing 2: Overengineered Directory Setup

In the *nested dictionary iteration* task, the model often reused variable names from the original Stack Overflow post instead of adhering to the benchmark constraints. At $T = 0.2$, 20/30 samples failed due to undefined loop variables, whereas $T = 0.8$ produced diverse set of incorrect variants (Listing 2).

These case studies show that even the strongest open-source models are highly sensitive to temperature, prompt design, and task complexity: higher temperatures increase exploration but also broaden the error surface, while more detailed prompts can improve correctness at the cost of complexity.

7 Discussion

Evaluation using **ExITBench** reveals significant challenges for current LLMs in generating operationally correct Ansible code, highlighting a critical gap between syntactic validity and reliable execution for complex, state-based IT automation. Low success rates (pass@10 max 23.9%) reveal LLMs struggle with functional correctness for real-world instructions, even with context. Our analysis of the open-source models indicates that models pre-trained on both extensive code and substantial natural language data outperform those trained on less balanced corpora. This suggests success in IT automation demands not just coding ability but also robust interpretation of complex user instructions.

Furthermore, our analysis of generation parameters showed that sampling temperature provides a crucial lever for balancing reliability (**pass@1**) and exploration (**pass@10**). Higher TELeR levels offered some multi-sample gains for certain models, but these gains were often offset by increased solution complexity. Similarly, incorporating error-aware guidance did not improve performance *significantly*. This suggests that even with stronger prompts, models struggle due to more fundamental limitations—such as reasoning about system state, interpreting variable scopes, and adhering to procedural constraints. Furthermore, as our performance analysis showed, achieving higher accuracy often comes at the cost of increased latency, adding another layer of complexity.

To see if models could fix their own mistakes, we tested them in an agentic self-correction study. The experiment focused on a representative task where models commonly failed: an OS-version detection playbook that resulted in a `fatal: ... 'install_patch_name' is undefined` error when the target host's OS did not match a spe-

cific version. For each iteration, the agent was provided with the previous failing code, the full Ansible execution log, and the structured output from our unit tests (e.g., `correct_version_printed = False`). The results of this case study are summarized in Table 5. As the table illustrates, a stark capability gap emerged between the proprietary and open-source models. The two proprietary models included in this case study, GPT-4.1-Mini and Gemini-2.5-Flash-Lite, were able to correctly diagnose the root cause from the feedback and produce a functionally correct script in a single correction attempt. In contrast, the top-performing open-source models struggled. Models like Qwen-2.5-Coder-7B-it and LLaMa-3.1-8B-it made superficial fixes that resolved the initial execution crash but failed to address the underlying logical error, thus still failing the functional test cases. The DeepSeek-Coder-V2-it model became stuck in a failure loop, unable to make meaningful progress. This case study provides further evidence that while simple agentic loops are a promising direction, they do not automatically resolve the fundamental state-reconciliation reasoning failures we identified, particularly for open-source models.

Model	Outcome	Notes
Proprietary Models		
GPT-4.1-Mini	Success	Solved root cause in 1 correction attempt.
Gemini-2.5-Flash-Lite	Success	Solved root cause in 1 correction attempt.
Open-Source Models		
Qwen-2.5-Coder-7B-it	Failure	Fixed initial crash but failed to solve the underlying logical error after 5+ iterations.
DeepSeek-Coder-V2-it	Failure	Got stuck in a failure loop and could not resolve the error after 5+ iterations.
LLaMa-3.1-8B-it	Failure	Fixed initial crash but failed to solve the underlying logical error after 5+ iterations.

Table 5: Agentic Self-Correction Performance.

These difficulties—especially in managing variable resolution and template logic—manifest clearly in task-specific performance and error patterns. Models consistently struggled with tasks that require precise state and variable handling, namely **Templating** (using Jinja2) and **Variable Management** (Section 4.3). Our error analysis confirms this pattern, revealing that conventional LLMs primarily fail due to state reconciliation reasoning issues (42.117%) and limited module-specific execution knowledge (26.203%), confirming that tracking state changes and applying domain-specific knowledge are primary bottlenecks.

Our findings imply that current open-source LLMs exhibit critical deficits in state reasoning and precise execution, hindering reliable IT automation. Overcoming key bottlenecks, such as variable and template management, demands more than prompt tuning—pointing to needs for architectural or domain-specific enhancements. Consequently, the observed high failure rates mandate dynamic execution validation, as ExITBench provides, since static checks alone cannot ensure operational safety and correctness.

More positively, **ExITBench** introduces a novel and challenging benchmark—difficult even for leading open-source models such as DeepSeek-Coder and Qwen and thereby lays the foundation for future research in AI-powered IT automation, including advances in model architectures, fine-tuning strategies, and domain-specific reasoning.

8 Conclusion

Ensuring the reliability of LLMs for complex IT automation tasks remains a significant challenge, largely due to the limitations of existing evaluation methods. We presented **ExITBench**, a benchmark focused on execution and real-world practitioner needs, to evaluate LLMs. Our comprehensive evaluation of 17 LLMs revealed low success rates in achieving desired system states, with predominant failures stemming not merely from syntax but from critical deficiencies in state-reconciliation reasoning and the application of module-specific execution knowledge. These fundamental bottlenecks, which prompt engineering alone proved insufficient to resolve, underscore that current LLMs are not yet consistently reliable for autonomous IT automation. Significant advances in models' stateful reasoning and domain adaptation capabilities are therefore necessary. We posit that challenging, execution-based benchmarks like ExITBench are essential for guiding and measuring future progress towards truly dependable LLM-based automation that correctly achieves the intended system state. We have publicly shared the artifacts used for our evaluation online (paser-group, 2023) to facilitate further research in this domain.

9 Limitations

While ExITBench provides valuable insights into LLMs' capabilities for IT automation tasks, several limitations should be acknowledged.

First, our evaluation covers 17 LLMs spanning 14 open-source models (3B–15B parameters) and 3 proprietary models. While this provides a wide baseline, extending the study to larger proprietary flagship models such as GPT-4o or Gemini-2.5-Pro would provide an even more complete picture of the upper performance ceiling.

Second, our benchmark's sample size (126 tasks) represents a limited subset of the vast IT automation landscape. While carefully curated to cover diverse categories, this sample may not capture all edge cases or specialized scenarios. With 126 tasks, pass@k point estimates are subject to sampling variance. Future versions of ExITBench will expand the task set, targeting 300+ tasks across the existing seven domains and at least two new domains (e.g., Cloud Provisioning, Container Orchestration), to reduce variance and improve the statistical reliability of model comparisons.

Third, although our environment-dependent execution design substantially reduces the risk that models succeed through memorization of Stack Overflow solutions (Section 3.1), we cannot fully rule out that models may have generalized useful abstractions — such as common Ansible module patterns — from pre-training data. Future work could include a held-out test set constructed from post-cutoff practitioner queries to provide a contamination-free upper bound.

Finally, while our new agentic case study explores a simple, multi-turn self-correction scenario, the primary benchmark still focuses on the generation of individual playbooks rather than the complex, multi-playbook orchestration common in large enterprise environments.

Acknowledgments

This research was partially funded by the U.S. National Science Foundation (NSF) Award # 2247141, Award # 2312321, and the U.S. National Security Agency (NSA) Award # H98230-21-1-0175. This work has benefitted from Dagstuhl Seminar 23082 "Resilient Software Configuration and Infrastructure Code Analysis."

References

Marah I Abdin, Sam Ade Jacobs, Ammar Ahmad Awan, Jyoti Aneja, Ahmed Awadallah, Hany Awadalla, Nguyen Bach, Amit Bahree, Arash Bakhtiari, Harkirat S. Behl, Alon Benhaim, Misha Bilenko, Johan Bjorck, Sébastien Bubeck, Martin Cai, Caio César Teodoro Mendes, Weizhu Chen, Vishrav Chaudhary, Parul Chopra, and 68 others. 2024. Phi-3 technical report: A highly capable language model locally on your phone. *CoRR*, abs/2404.14219.

D.H. Ackley, G.E. Hinton, and T.J. Sejnowski. 1985. A learning algorithm for boltzmann machines. *Cognitive Science*, 9:147–169.

Dharun Anandayuvaraj, Matthew Campbell, Arav Tewari, and James C Davis. 2024. Fail: Analyzing software failures from the news using llms. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, page 506–518, New York, NY, USA. Association for Computing Machinery.

ansible. 2022. Adb uses red hat ansible automation platform to boost infrastructure management. `https://www.redhat.com/en/resources/asian-development-bank-case-study`. [Online; accessed 25-Sep-2023].

Anthropic. 2024. Claude 3.5 Sonnet Model Card Addendum. Technical report, Anthropic.

Rabiul Awal, Mahsa Massoud, Zichao Li, Aarash Feizi, Suyuchen Wang, Christopher Pal, Aishwarya Agrawal, David Vazquez, Siva Reddy, Juan A. Rodriguez, Perouz Taslakian, Spandana Gella, and Sai Rajeswar. 2025. WebMMU: A benchmark for multimodal multilingual website understanding and code generation. In *ICLR 2025 Third Workshop on Deep Learning for Code*.

Mahi Begoug, Narjes Bessghaier, Ali Ouni, Eman Abdullah AlOmar, and Mohamed Wiem Mkaouer. 2023. What do infrastructure-as-code practitioners discuss: An empirical study on stack overflow. In *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2023, New Orleans, LA, USA, October 26-27, 2023*, pages 1–12. IEEE.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. Evaluating large language models trained on code. *CoRR*, abs/2107.03374.

Qihong Chen, Jiawei Li, Jiecheng Deng, Jiachen Yu, Justin Tian Jin Chen, and Iftekhar Ahmed. 2024. A deep dive into large language model code generation mistakes: What and why? *CoRR*, abs/2411.01414.

Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, Luke Marris, Sam Petulla, Colin Gaffney, Asaf Aharoni, Nathan Lintz, Tiago Cardal Pais, Henrik Jacobsson, Idan Szpektor, Nan-Jiang Jiang, and 3290 others. 2025. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *Preprint*, arXiv:2507.06261.

DeepSeek-AI. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *Preprint*, arXiv:2501.12948.

Shihan Dou, Haoxiang Jia, Shenxi Wu, Huiyuan Zheng, Weikang Zhou, Muling Wu, Mingxu Chai, Jessica Fan, Caishuang Huang, Yunbo Tao, Yan Liu, Enyu Zhou, Ming Zhang, Yuhao Zhou, Yueming Wu, Rui Zheng, Ming Wen, Rongxiang Weng, Jingang Wang, and 5 others. 2024. What's wrong with your code generated by large language models? an extensive study. *CoRR*, abs/2407.06153.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, and 82 others. 2024. The llama 3 herd of models. *CoRR*, abs/2407.21783.

Md Mahade Hasan, Muhammad Waseem, Kai-Kristian Kemell, Jussi Raskua, Juha Ala-Rantalaa, and Pekka Abrahamsson. 2025. Assessing small language models for code generation: An empirical study with benchmarks. *arXiv preprint arXiv:2507.03160*.

Md. Mahadi Hassan, John Salvador, Shubhra Kanti Karmaker Santu, and Akond Rahman. 2024. State reconciliation defects in infrastructure as code. *Proc. ACM Softw. Eng.*, 1(FSE):1865–1888.

Red Hat. 2023. Ansible lightspeed. Online; accessed 4-May-2025. `https://www.ansible.com/solutions/ansible-lightspeed`.

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

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping language to code in programmatic context. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018*, pages 1643–1652. Association for Computational Linguistics.

Saurabh Jha, Rohan R. Arora, Yuji Watanabe, Takumi Yanagawa, Yinfang Chen, Jackson Clark, Bhavya Bhavya, Mudit Verma, Harshit Kumar, Hirokuni Kitahara, Noah Zheutlin, Saki Takano, Divya Pathak, Felix George, Xinbo Wu, Bekir O Turkkan, Gerard Vanloo, Michael Nidd, Ting Dai, and 21 others. 2025. ITBench: Evaluating AI agents across diverse real-world IT automation tasks. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pages 27134–27197. PMLR.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. 2024. Swe-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Ahmad Faraz Khan, Azal Ahmad Khan, Anas Mohamed, Haider Ali, Suchithra Moolinti, Sabaat Haroon, Usman Tahir, Mattia Fazzini, Ali R. Butt, and Ali Anwar. 2025. Lads: Leveraging llms for ai-driven devops. *Preprint*, arXiv:2502.20825.

Patrick Tser Jern Kon, Jiachen Liu, Yiming Qiu, Weijun Fan, Ting He, Lei Lin, Haoran Zhang, Owen Park, George Elengikal, Yuxin Kang, Ang Chen, Mosharaf Chowdhury, Myungjin Lee, and Xinyu Wang. 2024. Iac-eval: A code generation benchmark for cloud infrastructure-as-code programs. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-Tau Yih, Daniel Fried, Sida I. Wang, and Tao Yu. 2023. DS-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 18319–18345. PMLR.

Jia Li, Ge Li, Yongmin Li, and Zhi Jin. 2025. Structured chain-of-thought prompting for code generation. *ACM Trans. Softw. Eng. Methodol.*, 34(2).

Jia Li, Ge Li, Xuanming Zhang, Yihong Dong, and Zhi Jin. 2024a. Evocodebench: An evolving code generation benchmark aligned with real-world code repositories. *CoRR*, abs/2404.00599.

Jia Li, Ge Li, Xuanming Zhang, Yunfei Zhao, Yihong Dong, Zhi Jin, Binhua Li, Fei Huang, and Yongbin Li. 2024b. Evocodebench: An evolving code generation benchmark with domain-specific evaluations. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*.

Chen Long, Panupong Pasupat, and Percy Liang. 2016. Simpler context-dependent logical forms via model projections. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 1456–1465.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, and 47 others. 2024. Starcoder 2 and the stack v2: The next generation. *Preprint*, arXiv:2402.19173.

Hengyu Luo, Zihao Li, Joseph Attieh, Sawal Devkota, Ona de Gibert, Shaoxiong Ji, Peiqin Lin, Bhavani Sai Praneeth Varma Mantina, Ananda Sreenidhi, Raúl Vázquez, Mengjie Wang, Samea Yusofi, and Jörg Tiedemann. 2025. Gloteval: A test suite for massively multilingual evaluation of large language models. *Preprint*, arXiv:2504.04155.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*.

NFsaavedra. 2024. What Infrastructure as Code issues drive you crazy? `https://www.reddit.com/r/sysadmin/comments/1apr87w/`. [Online; accessed 18-May-2025].

Augustus Odena, Charles Sutton, David Martin Dohan, Ellen Jiang, Henryk Michalewski, Jacob Austin, Maarten Paul Bosma, Maxwell Nye, Michael Terry, and Quoc V. Le. 2021. Program synthesis with large language models. In *n/a*, page n/a, n/a. N/a.

OpenAI. 2025. Openai api reference.

OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, and 262 others. 2024. Gpt-4 technical report. *Preprint*, arXiv:2303.08774.

C. Parnin, E. Helms, C. Atlee, H. Boughton, M. Ghattas, A. Glover, J. Holman, J. Micco, B. Murphy, T. Savor, M. Stumm, S. Whitaker, and L. Williams. 2017. The top 10 adages in continuous deployment. *IEEE Software*, 34(3):86–95.

paser-group. 2023. paser-group/llm-for-iac. `https://github.com/paser-group/llm-for-iac`. [Online; accessed 18-April-2026].

Jinjun Peng, Leyi Cui, Kele Huang, Junfeng Yang, and Baishakhi Ray. 2025. Cweval: Outcome-driven evaluation on functionality and security of llm code generation. *Preprint*, arXiv:2501.08200.

Qiwei Peng, Yekun Chai, and Xuhong Li. 2024. Humaneval-xl: A multilingual code generation benchmark for cross-lingual natural language generalization. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation, LREC/COLING 2024, 20-25 May, 2024, Torino, Italy*, pages 8383–8394. ELRA and ICCL.

Saurabh Pujar, Saurabh Sinha, Sandeep Kumar, Saurabh Tiwary, Saurabh Sinha, and Saurabh Tiwary. 2023. Automated code generation for information technology tasks in yaml through large language models. *arXiv preprint arXiv:2305.02783*.

Hariharan Ragothaman and Saai Krishnan Udayakumar. 2024. Optimizing service deployments with nlp based infrastructure code generation - an automation framework. In *2024 IEEE 2nd International Conference on Electrical Engineering, Computer and Information Technology (ICEECIT)*, pages 216–221.

Akond Rahman, Effat Farhana, and Laurie Williams. 2020. The 'as code' activities: development anti-patterns for infrastructure as code. *Empirical Software Engineering*, 25(5):3430–3467.

Akond Rahman and Chris Parnin. 2023. Detecting and characterizing propagation of security weaknesses in puppet-based infrastructure management. *IEEE Transactions on Software Engineering*, 49(6):3536–3553.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, and 6 others. 2023. Code llama: Open foundation models for code. *CoRR*, abs/2308.12950.

Priyam Sahoo, Saurabh Pujar, Ganesh Nalawade, Richard Genhardt, Louis Mandel, and Luca Buratti. 2024. Ansible lightspeed: A code generation service for it automation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, page 2148–2158, New York, NY, USA. Association for Computing Machinery.

Shubhra Kanti Karmaker Santu and Dongji Feng. 2023. Teler: A general taxonomy of LLM prompts for benchmarking complex tasks. In *Findings of the Association for Computational Linguistics: EMNLP 2023, Singapore, December 6-10, 2023*, pages 14197–14203. Association for Computational Linguistics.

Joel Scheuner, Philipp Leitner, Jürgen Cito, and Harald Gall. 2014. Cloud work bench – infrastructure-as-code based cloud benchmarking. In *2014 IEEE 6th International Conference on Cloud Computing Technology and Science*, pages 246–253.

Shivam Mehta. 2023. The need for sampling temperature and differences between probabilistic machine learning models.

Kalahasti Ganesh Srivatsa, Sabyasachi Mukhopadhyay, Ganesh Katrapati, and Manish Shrivastava. 2024. A survey of using large language models for generating infrastructure as code. *CoRR*, abs/2404.00227.

Minaoar Hossain Tanzil, Masud Sarker, Gias Uddin, and Anindya Iqbal. 2023. A mixed method study of devops challenges. *Information and Software Technology*, 161:107244.

Zhijie Wang, Zijie Zhou, Da Song, Yuheng Huang, Shengmai Chen, Lei Ma, and Tianyi Zhang. 2024a. Where do large language models fail when generating code? *CoRR*, abs/2406.08731.

Zhijie Wang, Zijie Zhou, Da Song, Yuheng Huang, Shengmai Chen, Lei Ma, and Tianyi Zhang. 2024b. Where do large language models fail when generating code? *CoRR*, abs/2406.08731.

Yiqing Xie, Alex Xie, Divyanshu Sheth, Pengfei Liu, Daniel Fried, and Carolyn Rose. 2024. Codebench-gen: Creating scalable execution-based code generation benchmarks. *Preprint*, arXiv:2404.00566.

An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, Guanting Dong, Hao-ran Wei, Huan Lin, Jialong Tang, Jialin Wang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Ma, and 40 others. 2024. Qwen2 technical report. *arXiv preprint arXiv:2407.10671*.

John Yang, Carlos E. Jimenez, Alex L. Zhang, Kilian Lieret, Joyce Yang, Xindi Wu, Ori Press, Niklas Muennighoff, Gabriel Synnaeve, Karthik R. Narasimhan, Diyi Yang, Sida Wang, and Ofir Press. 2025. Swe-bench multimodal: Do AI systems generalize to visual software domains? In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018a. Learning to mine aligned code and natural language pairs from stack overflow. In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, page 476–486, New York, NY, USA. Association for Computing Machinery.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018b. Learning to mine aligned code and natural language pairs from stack overflow. In *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*, pages 476–486. ACM.

Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. 2024a. Codereval: A benchmark of pragmatic code generation with generative pre-trained models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA. Association for Computing Machinery.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Zhang, and Dragomir Radev. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. *arXiv preprint arXiv:1809.08887*.

Zhaojian Yu, Yilun Zhao, Arman Cohan, and Xiao-Ping Zhang. 2024b. Humaneval pro and MBPP pro: Evaluating large language models on self-invoking code generation. *CoRR*, abs/2412.21199.

Heri Zhao, Jeffrey Hui, Joshua Howland, Nam Nguyen, Siqi Zuo, Andrea Hu, Christopher A. Choquette-Choo, Jingyue Shen, Joe Kelley, Kshitij Bansal, Luke Vilnis, Mateo Wirth, Paul Michel, Peter Choy, Pratik Joshi, Ravin Kumar, Sarmad Hashmi, Shubham Agrawal, Zhitao Gong, and 7 others. 2024. Codegemma: Open code models based on gemma. *CoRR*, abs/2406.11409.

Dewu Zheng, Yanlin Wang, Ensheng Shi, Ruikai Zhang, Yuchi Ma, Hongyu Zhang, and Zibin Zheng. 2025. Humanevo: An evolution-aware benchmark for more realistic evaluation of repository-level code generation. *Preprint*, arXiv:2406.06918.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric. P Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. Judging llm-as-a-judge with mt-bench and chatbot arena. *Preprint*, arXiv:2306.05685.

Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, and 1 others. 2024. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *arXiv preprint arXiv:2406.11931*.

Qiming Zhu, Jialun Cao, Yaojie Lu, Hongyu Lin, Xianpei Han, Le Sun, and Shing-Chi Cheung. 2025. Domaineval: An auto-constructed benchmark for multi-domain code generation. *Proceedings of the AAAI Conference on Artificial Intelligence*, 39(24):26148–26156.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen Gong, Thong Hoang, Armel Randy Zebaze, Xiaoheng Hong, Wen-Ding Li, Jean Kaddour, Ming Xu, Zhihan Zhang, and 14 others. 2024a. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *CoRR*, abs/2406.15877.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, and 1 others. 2024b. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*.

A Benchmark Curation and Validation Process

This appendix provides additional details on the curation and validation process for the ExITBench benchmark tasks, supporting the methodology described in Section 3.1.

Data Source and Timeframe The initial corpus for our benchmark consisted of 52,727 Stack Overflow posts related to IT Automation, as identified by (Begoug et al., 2023). This dataset encompasses posts spanning from 2012 to 2022, ensuring that our benchmark is derived from a wide and temporally diverse range of real-world practitioner challenges.

Manual Curation and Author Expertise The benchmark was developed by Authors 1 and 2, who together have over 4 years of hands-on experience with Ansible and more than 8 years with Python, spanning both industry and academic contexts. Their expertise in DevOps, systems administration, and scientific computing informed the rigorous manual validation of 200 sampled Stack Overflow posts, resulting in 126 finalized tasks and 733 test cases. Each candidate task was carefully reviewed to assess the clarity of user intent, feasibility of implementation, and the ability to define precise validation criteria within a controlled Docker environment. This expert-driven process ensured the technical accuracy, realism, and consistency of the final benchmark.

Conflict Resolution During the manual curation phase, any disagreements among the authors regarding a task's suitability, scope, or validation logic were resolved through a consensus-based approach. Each contested task was discussed collaboratively until a unanimous decision on its inclusion, exclusion, or modification was reached, ensuring a high-quality and consistent final benchmark.

Data Leakage Mitigation We acknowledge the risk of data leakage, as the LLMs evaluated may have been trained on portions of Stack Overflow. Our mitigation strategy focused on ensuring that success requires more than simple memorization. We curated tasks from problem descriptions, not just accepted code solutions. More importantly, our benchmark's core requirement is **functional correctness via execution** in our specific, controlled environment. This requires models to adapt solutions to our predefined hostnames, file paths, and initial system states, a task that verbatim code from Stack Overflow would almost always fail. This design ensures we test for true generation and state reconciliation capability, not just retrieval.

Initial Filtering Criteria The initial corpus for our benchmark consisted of 52,727 Stack Overflow posts related to IT Automation, as identified by Begoug et al. (Begoug et al., 2023). This dataset encompasses posts spanning from **2012 to 2022**, ensuring our benchmark is derived from a wide range of real-world practitioner challenges. These posts were automatically filtered based on the following criteria:

1. **Replicability:** The described problem needed to be potentially replicable within a containerized Linux environment, excluding issues specific to non-containerizable hardware, proprietary systems unavailable in Docker, or GUI interactions.
2. **Core Ansible Modules:** The problem or its likely solution needed to involve modules primarily from the `ansible.builtin`, `ansible.netcommon`, `ansible.utils`, or `community.general` collections.

Posts from the initial Stack Overflow corpus (Begoug et al., 2023) were automatically filtered based on:

1. **Replicability:** The described problem needed to be potentially replicable within a containerized Linux environment, excluding issues specific to non-containerizable hardware, proprietary systems unavailable in Docker, or GUI interactions.
2. **Core Ansible Modules:** The problem or its likely solution needed to involve modules primarily from the `ansible.builtin`, `ansible.netcommon`, `ansible.utils`, or `community.general` collections, focusing on common, well-supported functionalities. Posts relying heavily on obscure, deprecated, or highly specialized external collections were typically excluded.

Manual Curation Criteria The set of posts selected via stratified sampling (initially 200 posts) underwent manual review. Posts were excluded during this final curation if:

1. **Feasibility Issues:** Upon closer inspection, the task required external services, hardware,

credentials, or network configurations impractical or insecure to replicate reliably within the isolated Docker test environment.

2. **Ambiguity/Incompleteness:** The problem description was too vague, lacked crucial details, or the accepted SO solution was unclear, incomplete, or non-functional, preventing the formulation of a clear, testable task and validation criteria.
3. **Relevance Issues:** The post, despite keywords, did not ultimately represent a core Ansible automation task solvable via the targeted modules or involved primarily debugging Ansible itself rather than using it for automation.

Sampling Rationale and Representativeness While our benchmark construction relied on expert-driven stratified sampling and manual curation (rather than random selection for inferential statistics), we acknowledge the standard statistical framework for estimating sample representativeness. Assuming a simple random sample from 52,727 posts, a sample size of 200 (with a 95% confidence level and a maximum population proportion of 60%) yields a margin of error of approximately 6.8% for estimating proportions in the original corpus.[1] However, since our process involves expert curation and stratification by topic (rather than purely random sampling), this margin of error provides only a general reference point. As such, traditional confidence intervals do not strictly apply to our benchmark, but we include this calculation for transparency regarding sample size representativeness.

This rigorous process yielded the final 126 tasks for benchmark development. The distribution of curated tasks across seven IT automation domain is presented in Table 6.

Category	Curated Tasks
Deployment Pipelines	15
File Management	18
Networking	16
Policy Configuration	17
Server Configuration	27
Templating	15
Variable Management	18
Total Curated	**126**

Table 6: Distribution of the 126 IT automation tasks across the 7 IT automation domains.

[1] https://www.calculator.net/sample-size-calculator.html?type=2&cl2=95&ss2=200&pc2=60&ps2=52727&x=Calculate#findci

B IT Automaton Task Categories

Based on (Begoug et al., 2023), the seven categories of IT automation tasks we use are:

1. **Server Configuration:** Managing server environments, package installation/removal, service states (start, stop, restart), user/group management, basic system settings.
2. **Policy Configuration:** Defining security settings (firewalls, SELinux), compliance rules, access controls, system policy enforcement.
3. **Networking:** Configuring network interfaces, IP addressing, routing, DNS, network services (DHCP, NTP), basic network device interactions.
4. **Deployment Pipelines:** Automating steps in application deployment, including fetching artifacts, managing dependencies, deploying code, database migrations, basic CI/CD tasks.
5. **Variable Management:** Handling variable definition, scope (host, group, play), precedence, lookup plugins, complex data structures, and accessing facts.
6. **Templating:** Using dynamic templates (primarily Jinja2) to generate configuration files based on variables and logic.
7. **File Management:** Creating, modifying (lineinfile, blockinfile, replace), deleting files and directories, managing permissions, copying/fetching files.

C Docker-based Execution Environment

To ensure consistent, isolated, and reproducible execution for both reference solutions and LLM-generated playbooks, we designed a standardized Docker-based testing environment. This environment simulates a small, heterogeneous infrastructure network. We configured a dedicated Docker network with the subnet `10.1.1.0/24` and assigned the gateway address `10.1.1.254`. Within this network, we provisioned four distinct Docker containers acting as target nodes, each running a common Linux distribution frequently encountered in production systems:

- `ubuntu1`: Ubuntu Linux (IP: `10.1.1.1`)
- `alpine1`: Alpine Linux (IP: `10.1.1.2`)
- `centos1`: CentOS Linux (IP: `10.1.1.3`)
- `redhat1`: Red Hat Enterprise Linux (or a compatible distribution like Rocky/Alma Linux) (IP: `10.1.1.4`)

Each container was equipped with SSH access and

Python, prerequisites for Ansible control. The specific target node(s) for each benchmark task were determined by the task's requirements, often specified implicitly (e.g., tasks involving 'apt' target Ubuntu, 'yum'/'dnf' target CentOS/Red Hat) or explicitly in the adapted task description. This multi-node, multi-distribution setup allows for evaluating the correctness and portability of Ansible playbooks across diverse target systems, ensuring generated solutions are robust and not overly fitted to a single environment. Each test execution (detailed in Section 3.3) occurs within a fresh instance of this environment.

D Experiment Design

D.1 Evaluation Metrics

We evaluate the functional correctness of generated Ansible playbooks using the **pass@k** metric (Chen et al., 2021). This metric estimates the probability that at least one of the top k generated samples for a task passes all validation criteria. A sample is considered "correct" (c) only if it is syntactically valid, executes successfully without errors, achieves the desired functional outcome verified by our task-specific validation tests (Section 3.2), and is idempotent, all within the testing framework described in Section 3.3.

We use the unbiased estimator for pass@k, averaged across all benchmark tasks:

$$pass@k = \mathbb{E}_{\text{Tasks}} \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right] \quad (1)$$

where n is the total number of samples generated per task and c is the count of correct samples. In our experiments, we report results for $k \in \{1, 3, 5, 10\}$. We generated $n = \langle \text{Specify your value of } n \rangle$ samples per task for each LLM configuration (Section 3.4) to ensure robust estimation. While pass@k is the primary metric, analysis may also consider intermediate failure points like syntax errors.

D.2 LLM Selection and Configuration

The list of LLMs and their versions are given in Table 7. We evaluated a diverse set of fourteen large language models (LLMs), including both code-specialized models (Code LLMs) and general-purpose instruction-tuned models (General LLMs). Our selection spans multiple model families, parameter scales, and release dates to capture variations in architecture, training data, and objectives. Table 7 lists the specific models used in our study along with their key characteristics.

Each model was prompted to generate solutions for the 126 Ansible benchmark tasks (Section 3.2). For every task, we explored variations in the generation process using multiple prompting strategies and sampling temperatures, **as detailed in Section 3.4**. To compute the *pass@k* metrics (Section D.1), we generated $n = 15$ code samples for each unique combination of task, model, prompt style, and temperature setting.

E Experimental Execution

The inference process for generating Ansible code samples from all selected Large Language Models (LLMs) was conducted on a single NVIDIA H100 GPU to ensure hardware consistency. For each of the 14 open-source and three proprietary LLMs (Section 3.4), we generated responses for every one of the 126 benchmark tasks (Section 3.2).

As detailed in Section 3.4, we tested three distinct prompt structures (derived from TELeR Levels 1-3) and four different sampling temperature settings (0.2, 0.4, 0.6, 0.8). For each unique combination of model, task, prompt style, and temperature setting, we generated $n = 15$ independent samples, as required for the *pass@k* evaluation (Section D.1).

This resulted in a substantial number (22,680) of generated IT automation script per model. Where On average Each model needed 20 GPU hours to generate all those scripts. We user NVIDIA H100 GPU having 80GB memory to generate scripts from LLMs. So go generate the whole results we needed 280 GPU hours (Approximately).

Executing all experiments on uniform hardware ensures that observed performance differences can be attributed to the models and prompting strategies themselves, enabling a fair comparison across all configurations. The execution and evaluation of each generated sample followed the pipeline described in Section 3.3.

E.1 NLP packages

We use the following python packages for building our experiments: openai, tiktoken, accelerate, ansible, ansiblemetrics, colorama, datasets, ipython, numpy, pandas, tokenizers, torch, tqdm, transformers, argcomplete, simple_term_menu, paramiko, scp, pynvml, scipy, bitsandbytes, openpyxl, docker, bs4, protobuf, sentencepiece, pymysql, natsort,

LLM Name	Abbreviation	Family	Size	Type	Download Date
WizardLMTeam/WizardCoder-15B-V1.0 (Luo et al., 2023)	WizardCoder-15B	WizardCoder	15B	Code LLM	Jan 24
codellama/CodeLlama-7b-Instruct-hf (Rozière et al., 2023)	CodeLlama-7B-it	CodeLlama	7B	Code LLM	Apr 24
codellama/CodeLlama-13b-Instruct-hf (Rozière et al., 2023)	CodeLlama-13B-it	CodeLlama	13B	Code LLM	Apr 24
bigcode/starcoder2-7b (Lozhkov et al., 2024)	Starcoder2-7B	StarCoder	7B	Code LLM	Jul 24
google/codegemma-7b-it (Zhao et al., 2024)	Codegemma-7B-it	CodeGemma	7B	Code LLM	Aug 24
Qwen/Qwen2.5-Coder-7B-Instruct (Hui et al., 2024)	Qwen2.5-Coder-7B-it	Qwen2.5	7B	Code LLM	Sept 24
Qwen/Qwen2.5-7B-Instruct (Yang et al., 2024)	Qwen2.5-7B-it	Qwen2.5	7B	Code LLM	Sept 24
lmsys/vicuna-7b-v1.5 (Zheng et al., 2023)	Vicuna-7B-it	Vicuna	7B	Generic LLM	Mar 24
meta-llama/Llama-3.1-8B-Instruct (Dubey et al., 2024)	Llama-3.1-8B-it	Llama 3.1	8B	Generic LLM	Oct 24
meta-llama/Llama-3.2-3B-Instruct (Dubey et al., 2024)	Llama-3.2-3B-it	Llama 3.2	3B	Generic LLM	Oct 24
microsoft/Phi-3.5-mini-instruct (Abdin et al., 2024)	Phi-3.5-mini-it	Phi 3.5	3.8B	Generic LLM	Sept 24
deepseek-ai/DeepSeek-Coder-V2-Lite-Instruct (Zhu et al., 2024)	DeepSeek-Coder-V2-it	DeepSeek	7B	Code LLM	Mar 25
deepseek-ai/DeepSeek-R1-Distill-Llama-8B (DeepSeek-AI, 2025)	DeepSeek-Distil-L	DeepSeek	8B	Reasoning LLM	Mar 25
deepseek-ai/DeepSeek-R1-Distill-Qwen-7B (DeepSeek-AI, 2025)	DeepSeek-Distil-Q	DeepSeek	7B	Reasoning LLM	Mar 25
GPT-4.1-Mini (OpenAI et al., 2024)	GPT-4.1-Mini	GPT-4.1	N/A	Generic LLM	N/A
Claude-3.5-Sonnet (Anthropic, 2024)	Claude-3.5-Sonnet	Claude	N/A	Generic LLM	N/A
gemini-2.5-flash-lite-preview-06-17 (Comanici et al., 2025)	Gemini-2.5-Flash-lite	Gemini-2.5	N/A	Generic LLM	N/A

Table 7: Selected LLMs for Evaluation and Their Configuration Details. LLM Types indicate primary training focus (Code vs General). Sizes are approximate parameters.

LLM	Avg Latency (s)	Avg Throughput (tok/s)
Qwen2.5-Coder-7B-it	5.15	76.84
Qwen2.5-7B-it	2.49	136.47
WizardCoder-15B	1.38	225.42
CodeLlama-13B-it	5.71	78.10
CodeLlama-7B-it	7.18	40.02
starcoder2-7b	3.40	140.38
Llama-3.1-8B-it	2.12	175.43
Llama-3.2-3B-it	2.24	259.99
DeepSeek-Coder-V2-it	8.84	39.46
Vicuna-7B-it	3.85	51.24
Codegemma-7B-it	3.19	122.85
Phi-3.5-mini-it	5.55	101.25
DeepSeek-Distil-Q	2.56	108.00
DeepSeek-Distil-L	2.40	146.77
GPT-4.1-Mini	4.09	161.41
Gemini-2.5-Flash-lite	1.80	272.62
Claude-3.5-Sonnet	6.41	57.3

Table 8: LLM Latency and Throughput Comparison

umap-learn, sentence-transformers, hdbscan.

F Throughput and Latency Analysis

To address practitioner concerns about the cost of accuracy, we conducted a throughput and latency analysis. Our findings reveal a clear accuracy-performance trade-off. For example, GPT-4.1-Mini, the most accurate model, has a generation latency of 4.09s, more than double that of the faster Gemini-2.5-Flash-Lite at 1.8s. This highlights the practical costs associated with achieving higher accuracy. In Table 8 we present the latency and throughput comparison of different LLMs.

G Agentic Case Study

This appendix provides a detailed walkthrough of the agentic self-correction case study summarized in the Discussion (Section 7). This study was designed to investigate whether an interactive, multi-turn framework could allow LLMs to overcome the initial generation failures observed in our main benchmark by analyzing their own errors and refining their output. The experiment focused on a representative task where models commonly failed: an OS-version detection playbook that resulted in a `fatal: ... 'install_patch_name' is undefined` error when the target host's OS

did not match a specific version. For each iteration, the agent was provided with the previous failing code, the full Ansible execution log, and the structured output from our unit tests (e.g., `correct_version_printed = False`).

The results of this case study are summarized in Table 5. As the table illustrates, a stark capability gap emerged between the proprietary and open-source models. The proprietary models, GPT-4.1-Mini and Gemini-2.5-Flash-Lite, were able to correctly diagnose the root cause from the feedback and produced a functionally correct script in a single correction attempt. In contrast, the top-performing open-source models struggled. Models like Qwen-2.5-Coder-7B-it and LLaMa-3.1-8B-it made superficial fixes that resolved the initial execution crash but failed to address the underlying logical error, thus still failing the functional test cases. The DeepSeek-Coder-V2-it model became stuck in a failure loop, unable to make meaningful progress. This detailed analysis provides strong evidence for the conclusion made in the Discussion: while simple agentic loops are a promising direction, they do not automatically resolve the fundamental state reconciliation reasoning failures we identified, particularly for the open-source models.

H Decoding Strategies

We systematically varied two key aspects influencing LLM generation: the structure of the input prompt and the sampling temperature.

H.1 Prompt Design based on TELeR Level of Detail

To explore how the amount of information provided in the prompt affects Ansible code generation, we adapted the TELeR taxonomy (Santu and Feng, 2023), which categorizes prompts along dimensions including Turn, Expression, Level of Detail, and Role. Our study focused specifically on the **Level of Detail** dimension. This dimension describes the richness of the input prompt, ranging from minimal instruction (Level 0) to incorporating external documents (Level 5) or requesting explanations (Level 6).

Based on our objective of evaluating core Ansible generation capabilities and the nature of our benchmark tasks (derived from SO posts without readily available ideal solutions for few-shot examples or relevant external documents for RAG), we constrained our prompt variations to specific TELeR levels:

- **Excluded Levels:** We excluded Level 0 (minimal detail, likely insufficient guidance), Level 4 (requiring few-shot examples or evaluation guidelines not suitable for our setup), Level 5 (requiring external documents for RAG, which were not available per task), and Level 6 (focused on explainability, not core generation).
- **Utilized Levels:** Consequently, our experiments employed three prompt variations corresponding to **TELeR Levels** 1, 2, and 3. These levels represent increasing detail within the prompt itself, ranging from basic instruction (Level 1) to more elaborated descriptions potentially including constraints or hints derived from the stack overflow post analysis (Level 3), without relying on external examples or documents. Those prompts are presented in Table 9-11.
- **Error aware prompts:** We also experimented with how much error-aware prompts affect the LLM generation. We inject the errors to avoid that we got from Error Taxonomy (Section 5). We inject the error information in all three level of prompts presented in Table 12- 14.

This systematic variation allows us to assess the sensitivity of different LLMs to the level of detail provided directly within the prompt for Ansible task generation.

H.2 Temperature Variation

The temperature parameter significantly influences the randomness and creativity of LLM outputs during sampling (Ackley et al., 1985; OpenAI, 2025). Lower temperatures (closer to 0) make the model's output more deterministic, favoring the highest probability tokens at each step, leading to more focused but potentially less diverse results. Higher temperatures (e.g., approaching 1.0) increase randomness, allowing the model to sample lower probability tokens more often, potentially leading to more novel or diverse solutions but also increasing the risk of errors or nonsensical output (Shivam Mehta, 2023).

To understand the trade-off between correctness and diversity for Ansible code generation, we systematically generated samples using four distinct temperature settings: **0.2, 0.4, 0.6, and 0.8**. This range allows us to observe performance from near-deterministic generation (0.2) to significantly more stochastic generation (0.8). The impact of temperature is analyzed in Section 4.

Component	Content
system_role	As an expert Ansible developer and Linux systems administrator, your role is to analyze Stack Overflow posts and transform them into practical, self-contained, and well-documented solutions while ensuring cross-distribution compatibility, adhering to infrastructure-as-code best practices, considering distribution-specific differences, validating network requirements, ensuring proper YAML syntax, and verifying completeness without assuming external dependencies or pre-existing configurations.
prompt	Transform the given Stack Overflow post, including its title ({{title}}) and body ({{body}}), into a production-ready Ansible playbook solution that adheres to the specified target environment, which consists of a network with a subnet of 10.1.1.0/24 and a gateway at 10.1.1.254, along with four nodes—ubuntu1 (10.1.1.1) running Ubuntu Linux, alpine1 (10.1.1.2) running Alpine Linux, centos1 (10.1.1.3) running CentOS Linux, and redhat1 (10.1.1.4) running Red Hat Linux—while ensuring cross-distribution compatibility, complying with the implementation constraint ({{constraint}}), and delivering a self-contained, production-ready playbook that includes a brief analysis of the problem and solution approach, incorporates all necessary variables and files, follows YAML best practices, features comprehensive error handling, and is enclosed in triple backticks (“‘) for automated Python processing.

Table 9: Level 1 TELeR (Santu and Feng, 2023) Prompt Structure for IT Automation Tasks. Placeholder expressions enclosed in double curly braces (e.g., {{title}}, {{body}}, {{constraint}}) indicate variable components that will be dynamically replaced with task-specific information.

I Artifact License and AI Use

The source code and data artifact accompanying this paper is made publicly available under the permissive **MIT License**. This license permits any individual or entity to freely use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the software, with the sole condition that the original copyright notice and this permission notice be included in all copies or substantial portions of the software. We have chosen this license to maximize the accessibility and impact of our research, encouraging its adoption and extension by the community.

AI assistants like ChatGPT and Claude were used for minor assistance in editing, and improving clarity of the manuscript.

J Qwen 2.5 Coder 7B Instruct Model Case Study

Our detailed case study of Qwen2.5-Coder-7B, the top-performing model in our evaluation, revealed several nuanced behaviors and challenges. Initial observations indicated inconsistent performance trends related to sampling temperature, with some tasks showing improved success at higher temperatures (an "upward trend") while others performed better at lower temperatures (a "downward trend"). Similarly, the impact of TELeR prompt levels was not uniform across all tasks, prompting further investigation into when increased prompt detail was genuinely beneficial versus when it was not effectively utilized. The 'Templating' domain was identified as consistently underperforming for this model, suggesting that the task constraints or the model's approach to this category might require specific attention.

Further investigation into these patterns provided deeper insights. A key observation was that on higher TELeR levels of prompt detail, Qwen2.5-Coder-7B tended to generate overly complicated or 'over-engineered' playbooks. In some instances, these more complex Level 3 responses, while potentially containing elements of a correct solution, ultimately failed due to issues introduced by this over-complication. We also observed instances where the model appeared to treat code snippets or patterns directly from Stack Overflow posts as factual or directly applicable without sufficient adaptation to the specific task constraints presented in our benchmark. These findings suggest that even for high-performing models, achieving reliable and minimally complex solutions may require very clear, unambiguous instructions, potentially needing to 'spoon-feed' critical details to avoid

Component	Content
system_role	As an expert Ansible developer and Linux systems administrator, you possess deep knowledge of various Linux distributions, networking, and infrastructure automation. Your role involves analyzing Stack Overflow posts and transforming them into practical Ansible solutions while ensuring cross-distribution compatibility in playbooks and adhering to infrastructure-as-code best practices. When approaching problems, you should think step-by-step, carefully considering distribution-specific differences, validating network requirements, ensuring proper YAML syntax, and verifying the completeness of each solution. Assumptions about external dependencies or pre-existing configurations should be avoided, as all solutions must be self-contained and well-documented.
prompt	Transform the given Stack Overflow post, including its title (title) and body (body), into a production-ready Ansible playbook solution that adheres to the specified target environment and implementation constraint. The target environment consists of a network with a subnet of 10.1.1.0/24 and a gateway at 10.1.1.254, along with four nodes: ubuntu1 (10.1.1.1) running Ubuntu Linux, alpine1 (10.1.1.2) running Alpine Linux, centos1 (10.1.1.3) running CentOS Linux, and redhat1 (10.1.1.4) running Red Hat Linux. The implementation must comply with constraint: 'constraint' while ensuring cross-distribution compatibility. The deliverable should include a brief analysis of the problem and the solution approach, along with a fully self-contained, production-ready Ansible playbook that functions across all specified distributions. The playbook must incorporate all necessary variables and files, feature comprehensive error handling, and adhere to YAML best practices. Additionally, it must be enclosed in triple backticks (“‘) for automated Python processing. The primary focus is on creating a robust, directly implementable solution for the target environment.

Table 10: Level 2 TELeR (Santu and Feng, 2023) Prompt Structure for IT Automation Tasks. Placeholder expressions enclosed in double curly braces (e.g., {{title}}, {{body}}, {{constraint}}) indicate variable components that will be dynamically replaced with task-specific information.

Component	Content
system_role	You are an expert Ansible developer and Linux systems administrator with deep knowledge of different Linux distributions, networking, and infrastructure automation. Your role is to: 1. Analyze Stack Overflow posts and transform them into practical Ansible solutions 2. Ensure cross-distribution compatibility in playbooks 3. Follow infrastructure-as-code best practices You should: - Think step-by-step when analyzing problems - Consider distribution-specific differences - Validate network requirements - Ensure proper YAML syntax - Verify solution completeness Never assume external dependencies or pre-existing configurations. All solutions should be self-contained and well-documented.
prompt	Transform the following Stack Overflow post into an Ansible playbook solution. STACK OVERFLOW POST: Title: title Body: body TARGET ENVIRONMENT: Network: - Subnet: 10.1.1.0/24 - Gateway: 10.1.1.254 Nodes: 1. ubuntu1 (10.1.1.1) - Ubuntu Linux 2. alpine1 (10.1.1.2) - Alpine Linux 3. centos1 (10.1.1.3) - CentOS Linux 4. redhat1 (10.1.1.4) - Red Hat Linux IMPLEMENTATION CONSTRAINT: constraint DELIVERABLE REQUIREMENTS: 1. Brief analysis of the problem and your solution approach 2. Production-ready, self-contained Ansible playbook that: • Works across all specified distributions • Includes all necessary variables and files • Features comprehensive error handling • Follows YAML best practices 3. Must be enclosed in triple backticks (“‘) for automated Python processing Note: Focus on creating a robust, production-ready playbook that can be directly implemented in the target environment.

Table 11: Level 3 TELeR (Santu and Feng, 2023) Prompt Structure for IT Automation Tasks. Placeholder expressions enclosed in double curly braces (e.g., {{title}}, {{body}}, {{constraint}}) indicate variable components that will be dynamically replaced with task-specific information.

Component	Content
system_role	You are an expert Ansible developer and Linux administrator; always generate playbooks that comply with given constraint, avoid undefined variables and invalid paths, and carefully select modules appropriate for the specified constraint and runtime environment and use only its appropriate attributes, ensuring valid YAML and correct Ansible syntax, and follow the output pattern in constraint.
prompt	Transform the given Stack Overflow post, including its title (title) and body (body), into a production-ready Ansible playbook solution that adheres to the specified target environment, which consists of a network(subnet- 10.1.1.0/24) with four nodes—ubuntu1, alpine1, centos1, and redhat1, complying with the implementation constraint (constraint), and delivering a self-contained, production-ready playbook, incorporates all necessary variables and uses correct paths, follows YAML best practices, and is enclosed in triple backticks for automated Python processing.

Table 12: Level 1 TELeR (Santu and Feng, 2023) Prompt Structure with error awareness for IT Automation Tasks. Placeholder expressions enclosed in double curly braces (e.g., {{title}}, {{body}}, {{constraint}}) indicate variable components that will be dynamically replaced with task-specific information.

Component	Content
system_role	As a senior Ansible automation engineer with deep Linux expertise, you generate robust, production-ready playbooks for diverse distributions. Always follow best practices, strictly comply with constraint. Make sure to avoid using undefined variables and invalid file paths. Select modules that are suitable for the given constraints and runtime environment, and use only their correct attributes. Ensure that the generated YAML is valid and free from Ansible syntax or logic errors. The output follows the specified format outlined in the constraint.
prompt	Transform the given Stack Overflow post, including its title (title) and body (body), into a production-ready Ansible playbook solution that adheres to the specified target environment and implementation constraint. The target environment consists of a network with a subnet of 10.1.1.0/24 and a gateway at 10.1.1.254, along with four nodes: ubuntu1 (10.1.1.1), alpine1 (10.1.1.2), centos1 (10.1.1.3), and redhat1 (10.1.1.4). The implementation must comply with constraint: 'constraint'. The deliverable should include a brief analysis of the problem and the solution approach, along with a fully self-contained, production-ready Ansible playbook that functions across all specified distributions. Make sure to avoid using undefined variables and invalid file paths. Select modules that are suitable for the given constraints and runtime environment, and use only their correct attributes. Ensure that the generated YAML is valid and free from Ansible syntax or logic errors. The output follows the specified format outlined in the constraint, and adhere to YAML best practices. Additionally, it must be enclosed in triple backticks (“‘) for automated Python processing. The primary focus is on creating a robust, directly implementable solution for the target environment.

Table 13: Level 2 TELeR (Santu and Feng, 2023) Prompt Structure with error awareness for IT Automation Tasks. Placeholder expressions enclosed in double curly braces (e.g., {{title}}, {{body}}, {{constraint}}) indicate variable components that will be dynamically replaced with task-specific information.

Component	Content
system_role	You are an expert Ansible developer. When generating playbooks, always: 1. Strictly comply with given constraint (including output format, filenames, and requirements) 2. Define all variables before use and avoid undefined variable errors 3. Avoid using invalid file paths. 4. Carefully examine the constraint and runtime environment, and select modules appropriate for both. 5. Use only valid attributes for the selected modules. 6. Ensure valid YAML and correct Ansible syntax and structure. 7. Avoid logic errors. 8. Make sure the output follows the format specified in the constraint.
prompt	Transform the following Stack Overflow post into an Ansible playbook solution. STACK OVERFLOW POST: Title: title Body: body TARGET ENVIRONMENT: Network: - Subnet: 10.1.1.0/24 - Gateway: 10.1.1.254 Nodes: 1. ubuntu1 (10.1.1.1) - Ubuntu Linux 2. alpine1 (10.1.1.2) - Alpine Linux 3. centos1 (10.1.1.3) - CentOS Linux 4. redhat1 (10.1.1.4) - Red Hat Linux IMPLEMENTATION CONSTRAINT: constraint DELIVERABLE REQUIREMENTS: 1. Brief analysis of the problem and your solution approach 2. Production-ready, self-contained Ansible playbook that: • Works across all specified distributions • Includes all necessary variables and files • Avoid using invalid file paths. • Carefully examine the constraint and runtime environment, and select modules appropriate for both. • Use only valid attributes for the selected modules. • Make sure the output follows the format specified in the constraint. • Follows Ansible's best practices 3. Must be enclosed in triple backticks for automated Python processing Note: Focus on creating a robust, production-ready playbook that can be directly implemented in the target environment.

Table 14: Level 3 TELeR (Santu and Feng, 2023) Prompt Structure with error awareness for IT Automation Tasks. Placeholder expressions enclosed in double curly braces (e.g., {{title}}, {{body}}, {{constraint}}) indicate variable components that will be dynamically replaced with task-specific information.

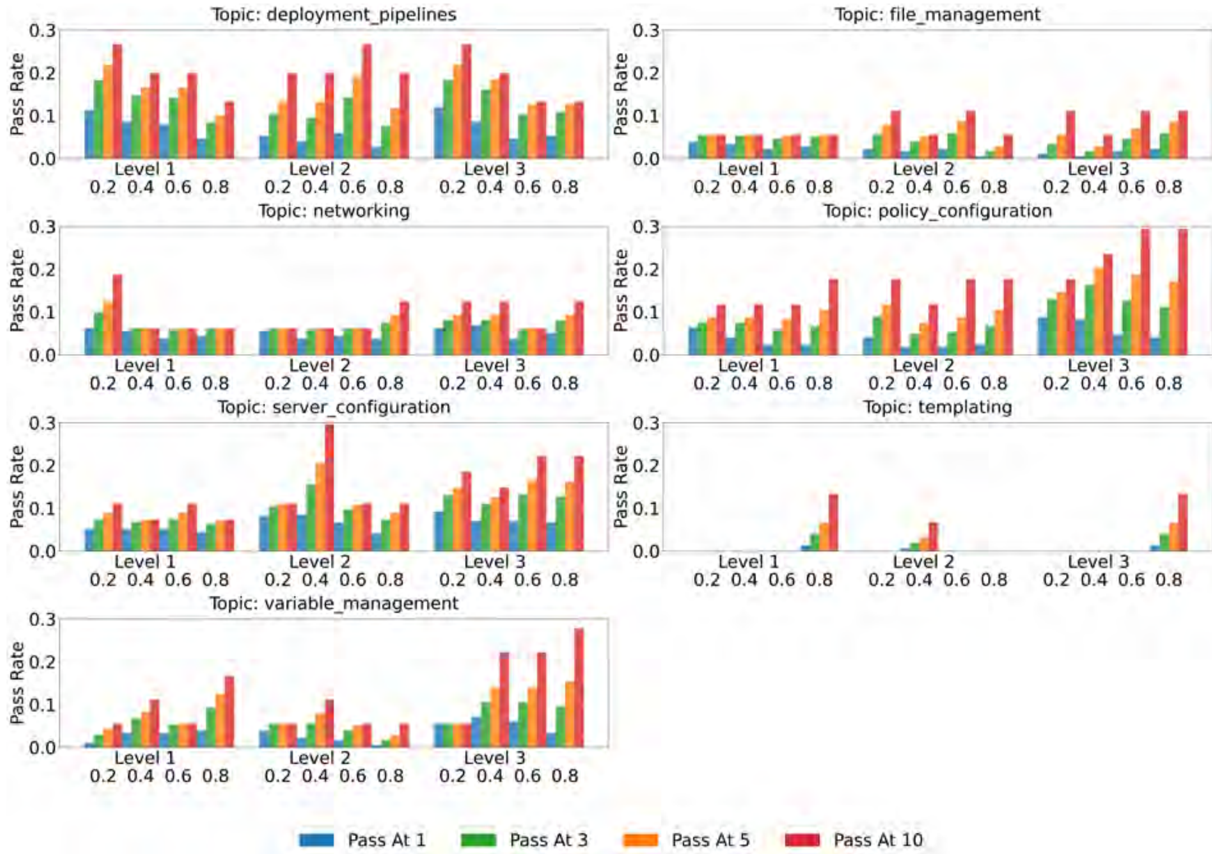


Figure 5: Performance of Qwen-2.5-Coder-7B model.

misinterpretation or over-complication.

To further understand these temperature-related performance trends, we conducted an error message analysis. This study revealed several key observations regarding how sampling temperature influences solution diversity and success patterns. We found that any upward trend in performance with increasing temperature predominantly occurred for the Pass@10 metric, while Pass@1 typically showed a downward trend. Specifically, when an upward trend was observed at higher temperatures, it often corresponded to a small number of successful generations (typically 1–3 out of 30 samples per issue) for highly customized tasks where memorized or common solutions likely failed. This suggests that higher temperatures, by generating a more diverse set of outputs and consequently a wider array of error messages, enabled the model to occasionally discover correct, albeit rare, solutions through increased exploration. Conversely, the downward trend in Pass@1 (i.e., better performance at lower temperatures) was more common for less customized problems, where the LLM might have already encountered similar code patterns, making deterministic generation more effective.

J.1 Upward Trend

Here, I have included the cases for which we see an upward trend and their error analysis.

J.1.1 Task ID: 68411894

For this task, the error trend can be found in Table 15.

Table 15: Error counts by temperature for Task ID 68411894.

Error Name	T=0.2	T=0.8
Undefined Loop Variable	20	10
No Tasks	2	0
Wrong File Lookup	1	0
Wrong Path	5	7
Define Variable Itself & Fails	1	0
Invalid Code	0	3
Undefined Variable	0	1
Wrong Variable Access	0	2
Wrong Syntax for Block Used	0	1
Not Loaded Data	0	1

J.1.2 Task ID: 44134617, Pass Count: 2

For Task ID 44134617, which involved defining server variables and then ensuring a specific directory structure and file creation (base directory /tmp/servers, individual server directories like /tmp/servers/server*, and a file within each), successful instances were observed with TELER Level 3 prompts (two passes). However, when examining failures at a sampling temperature of T=0.8, common issues included incorrect path usage (5 instances), such as attempting to use /tmp/server* directly where a loop or specific server name was required, and incorrect host targeting (1 instance).

J.2 Task ID: 44738112, Pass Count: 1

Error diversity can be found in Table 16.

J.3 Task ID: 51137660, Pass Count: 1

Table 17 summarizes the diversity of error types observed at different temperatures, highlighting how higher sampling temperature leads to a broader range of failure modes.

J.4 Task ID: 60687917, Pass Count: 1

Table 18 summarizes the diversity of error types observed at different temperatures, highlighting how higher sampling temperature leads to a broader range of failure modes.

J.5 Task ID: 28606876, Pass Count: 1

Table 19 summarizes the diversity of error types observed at different temperatures, highlighting how higher sampling temperature leads to a broader range of failure modes.

J.6 Task ID: 39922427, Pass Count: 2

Table 20 summarizes the diversity of error types observed at different temperatures, highlighting how higher sampling temperature leads to a broader range of failure modes.

J.7 Task ID: 46927378, Pass Count: 1

Table 21 summarizes the diversity of error types observed at different temperatures, highlighting how higher sampling temperature leads to a broader range of failure modes.

K Detailed Unit Test Cases

K.1 Server Configuration: User Check

```
---
- name: Check if user exists in /etc/
    passwd on all compute nodes
  hosts: all
  become: no
  vars:
    username: "{{ lookup('env', '
        username') }}"
  tasks:
    - name: Check if user exists in /etc
        /passwd
      command: grep "^{{ username }}:" /
          etc/passwd
      register: user_check
      ignore_errors: yes

    - name: Print result based on user
        existence
      debug:
        msg: "{{ username }} is present"
      when: user_check.rc == 0

    - name: Print result based on user
        absence
      debug:
        msg: "Not Found"
      when: user_check.rc != 0
```

Listing 3: User Check

Expected Outcome: The expected outcome involves the successful execution of the automation task, ensuring it completes without error for at least one designated user. Verification also includes confirming that the output accurately reports the number of specified items or users that were not found during the process. Furthermore, the system must correctly display or log the count of users who were successfully processed or identified in accordance with the task's specific objectives.

K.2 Policy Configuration: File existence check

```
--
- name: WAMUI Prerequisites Installer
  hosts: localhost
  gather_facts: no
  tasks:
    - name: Ensure required files exist
      copy:
        src: data/metadata/44738112/file
            .bin
        dest: /tmp/file.bin
        mode: '0755'

    - name: Run file.bin with the expect
         module
      expect:
        command: '/tmp/file.bin'
        responses:
          'PRESS <ENTER> TO CONTINUE:':
              '\n'
      register: expect_result

    - name: Display expect module output
      debug:
```

Table 16: Error diversity by temperature.

Issue Description	Count (T=0.2)	Count (T=0.8)
ERROR! conflicting action statements	2	-
assert and fail message together	1	-
invalid code (syntactic/parsing errors)	9	8
invalid path (file/directory not found)	17	12
wrong inventory (host not in inventory)	3	-
wrong host (targeting incorrect host)	-	6
undefined variable	-	1
command not found (in shell module, etc.)	-	1
does not execute file (e.g., script module issue)	-	2
invalid operand used (e.g., in conditional)	-	1

Table 17: Error diversity by temperature.

Error Type	T = 0.2	T = 0.8
Undefined variable	27	10
Wrong attribute accessed	6	present
Wrong function applied	1	–
Syntax issue	–	13
Invalid attribute used	–	present
Invalid YAML	–	3
Does not filter non-running services	–	2
Could not filter out services	–	1
Unsupported parameter used	–	1
Invalid operand used	–	1

```
        msg: "{{ expect_result.
          stdout_lines }}"
```

Listing 4: File existance check.

Expected Outcome: The expected outcome for this task is the successful execution of the playbook on the controller node. This entails utilizing the specified file, 'data/metadata/44738112/file.bin', located on the controller. Additionally, verification includes ensuring that all intended output is correctly printed to localhost and confirming that the playbook operated on the correct inventory, targeting the controller node as specified.

K.3 Networking: Configure rax

```
---
- name: Conditional Group Variable
    Example
  hosts: localhost
  connection: local
  gather_facts: no

  vars:
    rax_count: '2'
    rax_group: 'web_group'
```

```
  tasks:
    - name: Check if rax_count is
        greater than 1
      set_fact:
        rax_group: 'web_group'
      when: rax_count | int > 1

    - name: Debug rax_count
      debug:
        msg: "rax_count is {{ rax_count
          }}"

    - name: Debug rax_group
      debug:
        msg: "rax_group is {{ rax_group
          }}"
```

Listing 5: Ansible Playbook for rax Configuration

Expected Outcome: Check if rax group has correct count..

K.4 Deployment Pipelines: Server existence check and write on file

```
---
- name: Check service existence and
    write to file
```

Table 18: Error diversity for Task ID 60687917 by temperature.

Error Type	**T = 0.2**	**T = 0.8**
Type mismatch in comparison	15	15
Invalid YAML	13	9
Wrong host condition	3	3
Wrongly setting var to default	2	–
Conflicting action statement	5	1
Wrong host	1	4
Undefined variable	–	1
Wrong attribute for block	–	1
Template error	–	1
Hosts field not set	–	2

Table 19: Error diversity for Task ID 28606876 by temperature.

Error Type	**T = 0.2**	**T = 0.8**
Cannot resolve socket	3	3
Invalid YAML	14	9
`lsof` not found in shell	1	1
Undefined attribute used	5	–
Undefined variable used	2	–
Timeout	1	1
Wrong service checked	2	1
Malformed data passed	2	–
Invalid value in hosts	–	1
Unsupported parameter used	–	2
Wrong attribute used	–	4
Invalid variables specified	–	1
Loop inside block is invalid	–	1
Unbalanced Jinja block	–	1

```
hosts: all
become: yes
tasks:
  - name: Check if dummy_service.txt
      exists
    stat:
      path: /tmp/63688612/
          dummy_service.txt
    register: service_check

  - name: Write "The Service Exists"
      to test.txt if file exists
    copy:
      content: "The Service Exists"
      dest: /tmp/63688612/test.txt
    when: service_check.stat.exists

  - name: Write "The Service Does Not
      Exist" to test.txt if file does
      not exist
    copy:
      content: "The Service Does Not
          Exist"
      dest: /tmp/63688612/test.txt
    when: not service_check.stat.
        exists
```

Listing 6: Ansible Playbook for Server existence check and write on file

Expected Outcome: The task involves performing a file-based service existence check. Specifically, the automation script must determine if a file exists at the path /tmp/63688612/dummy_service.txt. Based on this check, it must write either 'The Service Exists' or 'The Service Does Not Exist' into the file /tmp/63688612/test.txt.

The verification process occurs in two stages. First, with the dummy_service.txt file absent, the playbook is expected to pass execution on all nodes. Success in this stage is further confirmed by verifying that the test.txt file is created on all nodes and that its content correctly states 'The Service Does

Table 20: Error diversity for Task ID 39922427 by temperature.

Error Type	T = 0.2	T = 0.8
Wrong inventory	8	4
Invalid variable	9	–
Invalid path	8	–
Undefined path	–	1
Undefined variable	–	2
Connection refused	–	1
Syntax issue	–	1
Check mode usage:		
Does not use check mode	present	–
Uses check mode	–	present

Table 21: Error diversity for Task ID 46927378 by temperature.

Error Type	T = 0.2	T = 0.8
Wrong URL	20	14
Invalid path	7	2
Invalid attribute for Ansible	6	–
Undefined variable	10	8
Format error	1	1
Wrong host	2	–
Timeout	1	–
Invalid YAML	1	4
No module/action in task	–	5
Unnecessary package install	–	2
Invalid role used	–	1
Wrong attribute for block	–	1
Wrong module used	–	3
Missing argument	–	1

Not Exist'. Subsequently, the dummy_service.txt file is introduced into the environment. The playbook is then executed again, and it is expected to pass on all nodes. Verification for this second stage involves checking that the test.txt file is present on all nodes and, crucially, that its contents now correctly reflect 'The Service Exists'.

K.5 Variable Management: Transform Directory Substitution

```
---
- name: Transform People Dictionary
  hosts: localhost
  gather_facts: false

  vars_files:
    - "data/metadata/35605603/result.yml
      "

  tasks:
  - name: Initialize the transformed
     dictionary
    set_fact:
      people_dict: {}

  - name: Populate the dictionary with
      names and genders
    set_fact:
      people_dict: "{{ people_dict |
          combine({ item.item.name:
          item.stdout }) }}"
    loop: "{{ people.results }}"

  - name: Print the transformed
     dictionary
    debug:
      var: people_dict
```

Listing 7: Ansible Playbook for Transforming directory

Expected Outcome: The task requires utilizing the people variable, which is to be sourced from the file 'data/metadata/35605603/result.yml'. The

playbook's primary action is to print this variable after it has been converted into a dictionary format. This operation is to be performed specifically on the controller node.

Verification involves several checks: first, the playbook must be executed. Its successful completion without errors is then assessed. Crucially, the output must be inspected to ensure that the correct dictionary values derived from the people variable are printed. Finally, it is verified that the playbook ran on the intended inventory, targeting only the controller node.

K.6 Templating: Check DNS reverse File

```
---
- name: Check for DNS reverse files on
    compute nodes
  hosts: all
  become: yes

  vars:
    dns_reverse_dir: "/tmp/test_dns"
    dns_reverse_pattern: "^named\\."

  tasks:
    - name: Find DNS reverse files
      find:
        paths: "{{ dns_reverse_dir }}"
        patterns: "{{
            dns_reverse_pattern }}"
        use_regex: yes
      register: find_results

    - name: Print found DNS reverse
        files
      debug:
        msg: "Node {{ inventory_hostname
             }}: Found file - {{ item.
            path }}"
      loop: "{{ find_results.files }}"
      when: find_results.files | length
          > 0

    - name: Handle case when no DNS
        reverse files are found
      debug:
        msg: "Node {{ inventory_hostname
             }}: No DNS reverse files
            found."
      when: find_results.files | length
          == 0
```

Listing 8: Ansible Playbook for Checking DNS reverse file

Expected Outcome: The task requires the automation script to iterate through each compute node, identify any DNS reverse files located within the /tmp/test_dns directory, and print their names. The output for each found file should adhere to the format: "Node <node_name>: Found file - <file_path>". A critical aspect of this task is the graceful handling of errors, particularly if the /tmp/test_dns directory or any expected files do not exist.

Verification of the generated solution is multifaceted. Initially, the correctness of debug arguments used in the playbook is checked. The playbook is then tested in a scenario where the /tmp/test_dns directory is absent, ensuring that this condition is handled gracefully on each node. Subsequently, the directory is created, and the playbook's behavior with an empty directory is verified on all nodes. The testing progresses by partially populating the directory with some DNS reverse files; the solution must correctly handle this partial existence on each node. Throughout these stages, a key validation point is that the script accurately prints the names of any existing files in the specified format.

K.7 File Management: Copy file to destination

```
---
- name: Copy files to destination
    directory
  hosts: all
  become: yes
  vars:
    file_path: "/tmp/destination_dir"

  tasks:
    - name: Create destination directory
         if it does not exist
      file:
        path: "{{ file_path }}"
        state: directory
        mode: '0755'

    - name: Copy files to destination
        directory
      copy:
        src: "data/metadata/{{ item }}"
        dest: "{{ file_path }}/{{ item
            }}"
        mode: '0755'
      loop:
        - "file1.txt"
        - "file2.txt"
```

Listing 9: Ansible Playbook for copy file to destination

Expected Outcome: Directory and file exist in all nodes.

L Results

L.1 Overall Performance Across Models

Overall Performance of all models is presented in Figure 5 - 20.

Table 22 summarizes the performance of open-source LLMs, showing the number of tasks within

Table 22: Summary of open-source LLMs performance across the seven IaC task categories. '# Tasks' indicates the number of benchmark tasks in each category. Models listed solved at least one task unless noted in the last column. Counts in parentheses indicate the number of tasks solved by that specific model within the category. Assumes 'solved' means pass@1 > 0 for at least one configuration.

Category	**# Tasks**	**Best Model(s) (Solved)**	**Other Successful Models (Solved)**	**Models with No Success (Solved=0)**
Variable Management	8	Qwen/Qwen2.5-Coder-7B-Instruct (8)	meta-llama/Llama-3.1-8B-Instruct (3), microsoft/Phi-3.5-mini-instruct (2), lmsys/vicuna-7b-v1.5 (2), WizardLMTeam/WizardCoder-15B-V1.0 (1), google/codegemma-7b-it (1)	codellama/CodeLlama-13b-Instruct-hf, codellama/CodeLlama-7B-Instruct-hf, Qwen/Qwen2.5-7B-Instruct, meta-llama/Llama-3.2-3B-Instruct, bigcode/starcoder2-7b, [DS-R1-Llama], [DS-R1-Qwen]
Networking	4	Qwen/Qwen2.5-Coder-7B-Instruct (4)	Qwen/Qwen2.5-7B-Instruct (3), bigcode/starcoder2-7b (3), codellama/CodeLlama-7B-Instruct-hf (2), meta-llama/Llama-3.1-8B-Instruct (2), google/codegemma-7b-it (2), microsoft/Phi-3.5-mini-instruct (1), lmsys/vicuna-7b-v1.5 (1), codellama/CodeLlama-13b-Instruct-hf (1)	[DS-R1-Qwen]
Templating	4	Qwen/Qwen2.5-Coder-7B-Instruct (3)	google/codegemma-7b-it (2), microsoft/Phi-3.5-mini-instruct (2), codellama/CodeLlama-7B-Instruct-hf (1), codellama/CodeLlama-13b-Instruct-hf (1), Qwen/Qwen2.5-7B-Instruct (1), meta-llama/Llama-3.1-8B-Instruct (1)	WizardLMTeam/WizardCoder-15B-V1.0, bigcode/starcoder2-7b, lmsys/vicuna-7b-v1.5, meta-llama/Llama-3.2-3B-Instruct, [DS-R1-Llama], [DS-R1-Qwen]
Deployment Pipelines	4	Qwen/Qwen2.5-Coder-7B-Instruct (4), meta-llama/Llama-3.1-8B-Instruct (4)	codellama/CodeLlama-13b-Instruct-hf (3), google/codegemma-7b-it (2), microsoft/Phi-3.5-mini-instruct (2), lmsys/vicuna-7b-v1.5 (2), bigcode/starcoder2-7b (2), Qwen/Qwen2.5-7B-Instruct (1), WizardLMTeam/WizardCoder-15B-V1.0 (1)	[DS-R1-Llama], [DS-R1-Qwen]
Policy Configuration	7	Qwen/Qwen2.5-Coder-7B-Instruct (7)	lmsys/vicuna-7b-v1.5 (4), microsoft/Phi-3.5-mini-instruct (2), google/codegemma-7b-it (2), WizardLMTeam/WizardCoder-15B-V1.0 (1), bigcode/starcoder2-7b (1), meta-llama/Llama-3.1-8B-Instruct (1), [DS-R1-Llama] (2), meta-llama/Llama-3.2-3B-Instruct (1)	codellama/CodeLlama-13b-Instruct-hf, codellama/CodeLlama-7B-Instruct-hf, Qwen/Qwen2.5-7B-Instruct
Server Configuration	11	Qwen/Qwen2.5-Coder-7B-Instruct (11)	meta-llama/Llama-3.1-8B-Instruct (10), bigcode/starcoder2-7b (9), google/codegemma-7b-it (8), codellama/CodeLlama-13b-Instruct-hf (8), Qwen/Qwen2.5-7B-Instruct (8), codellama/CodeLlama-7B-Instruct-hf (7), lmsys/vicuna-7b-v1.5 (5), meta-llama/Llama-3.2-3B-Instruct (5), microsoft/Phi-3.5-mini-instruct (4), WizardLMTeam/WizardCoder-15B-V1.0 (3)	—
File Management	5	google/codegemma-7b-it (5), meta-llama/Llama-3.1-8B-Instruct (5)	Qwen/Qwen2.5-Coder-7B-Instruct (4), bigcode/starcoder2-7b (3), codellama/CodeLlama-13b-Instruct-hf (3), WizardLMTeam/WizardCoder-15B-V1.0 (2), lmsys/vicuna-7b-v1.5 (2), Qwen/Qwen2.5-7B-Instruct (2), microsoft/Phi-3.5-mini-instruct (2), meta-llama/Llama-3.2-3B-Instruct (2), codellama/CodeLlama-7B-Instruct-hf (1)	[DS-R1-Llama], [DS-R1-Qwen]

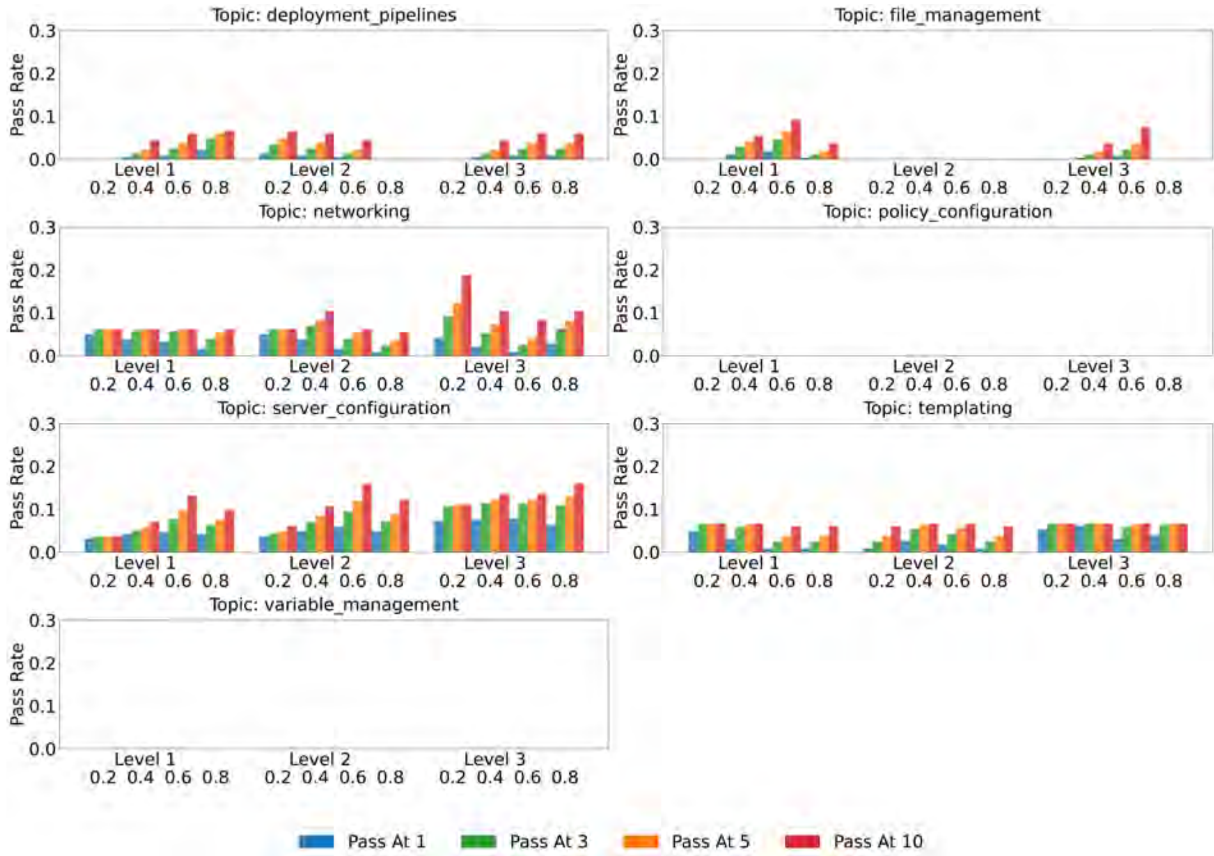


Figure 6: Performance of Qwen-2.5-7B-instruct-it model.

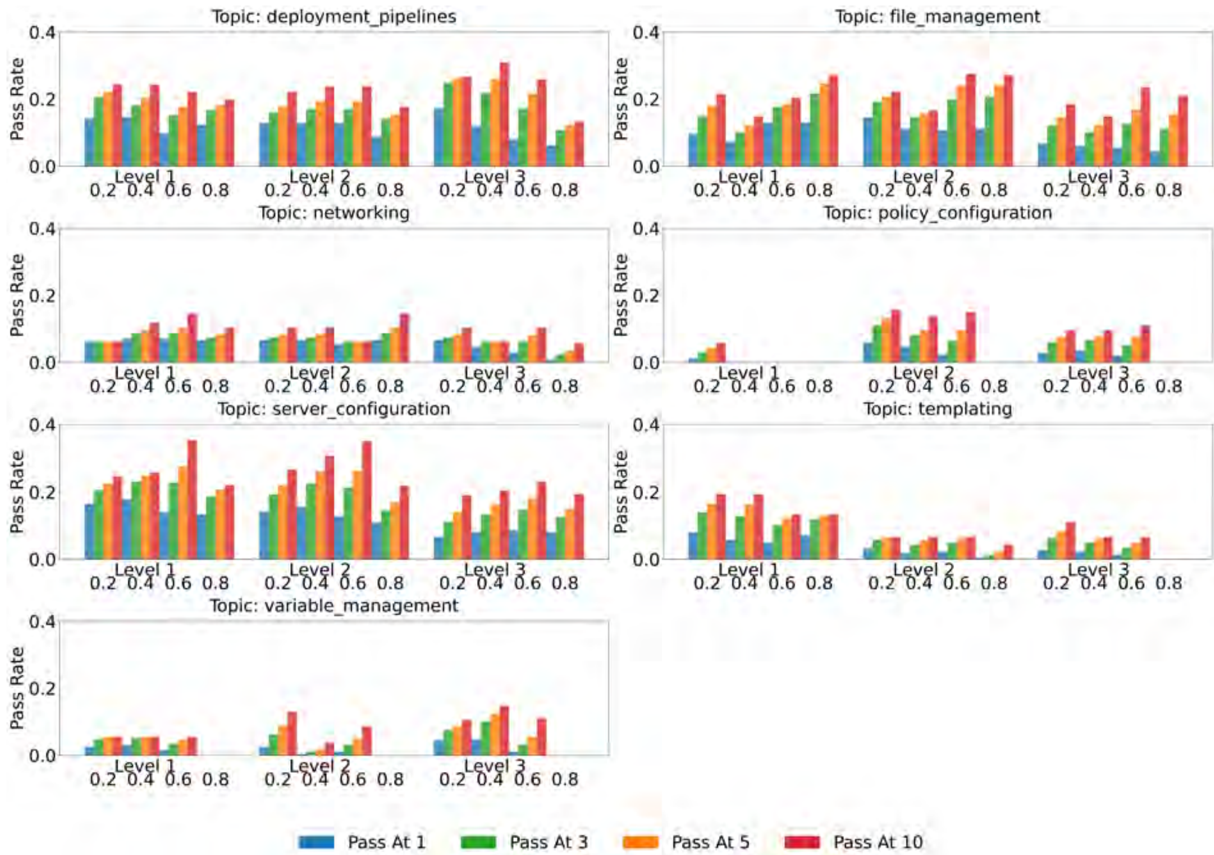


Figure 7: Performance of Gemini-2.5-Flash model.

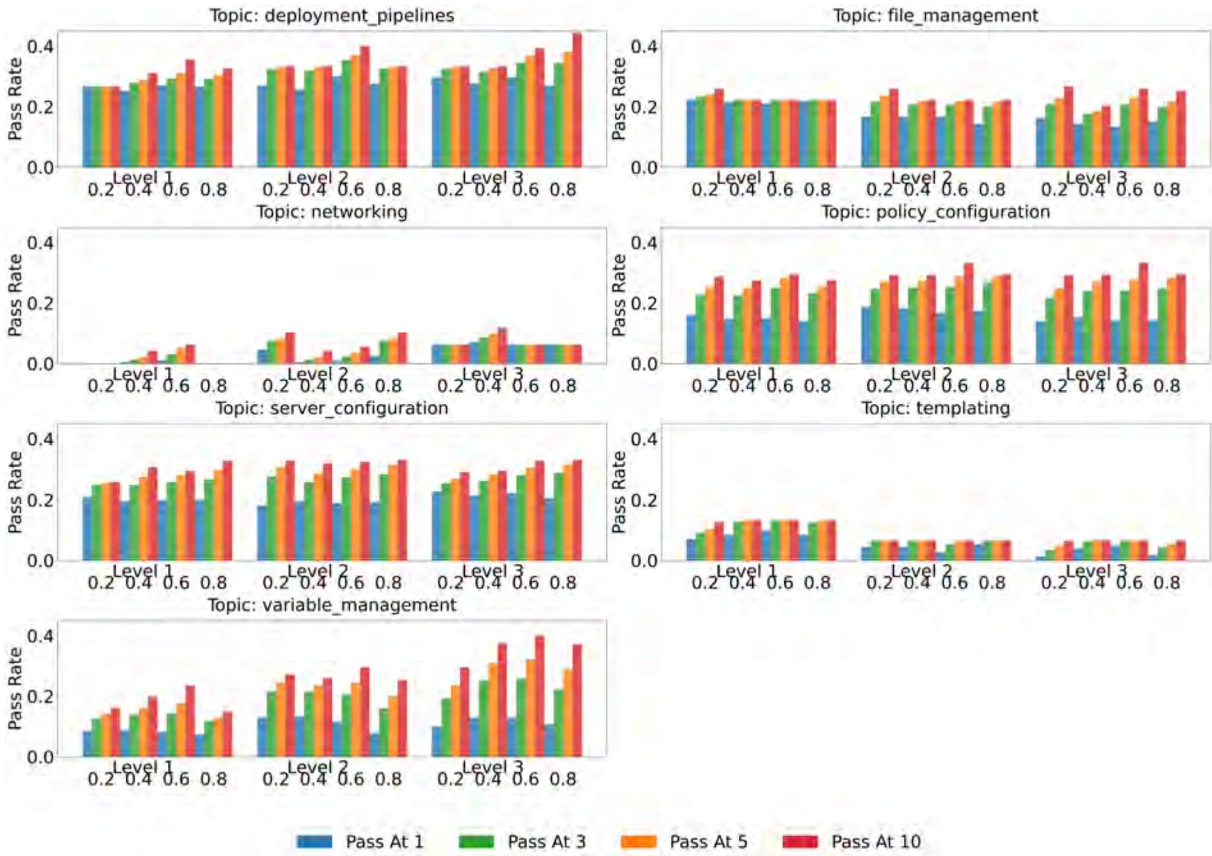


Figure 8: Performance of GPT-4.1-Mini model.

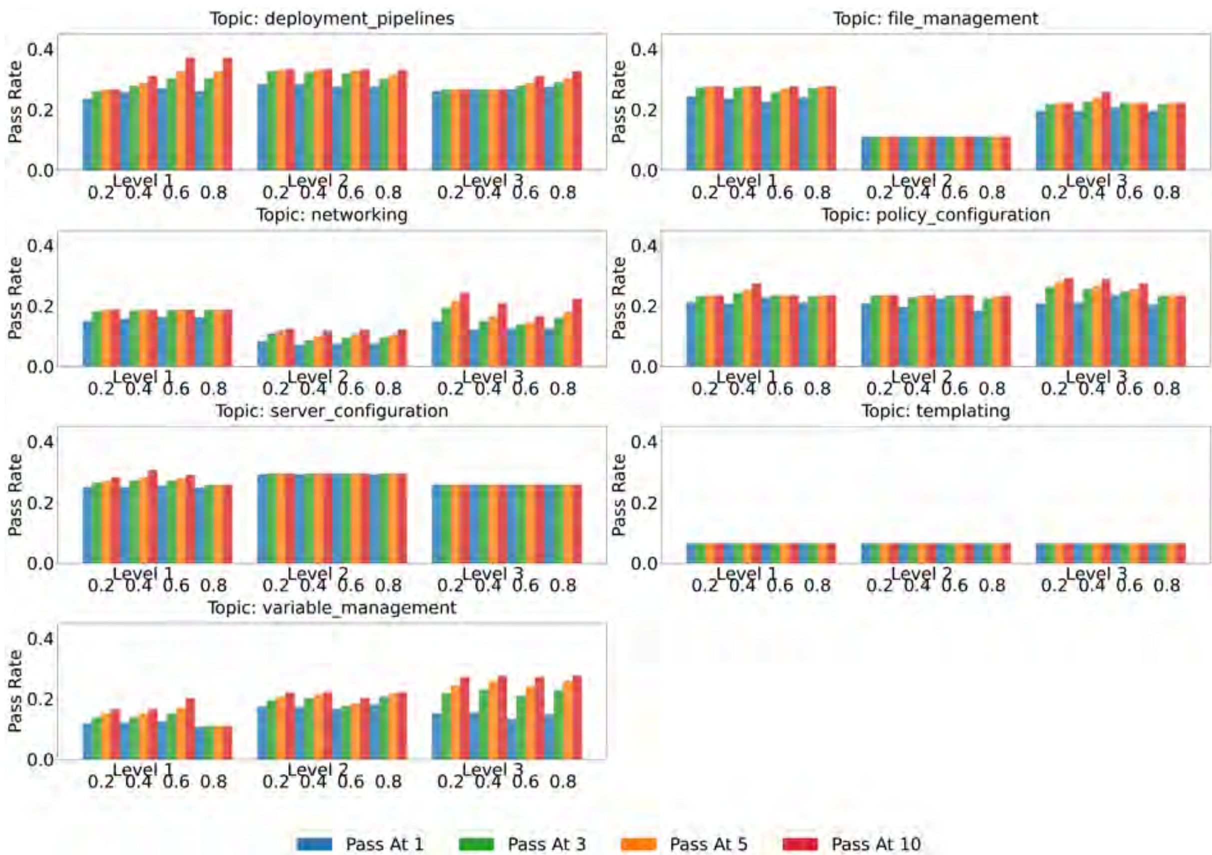


Figure 9: Performance of Claude-3.5-Sonnet model.

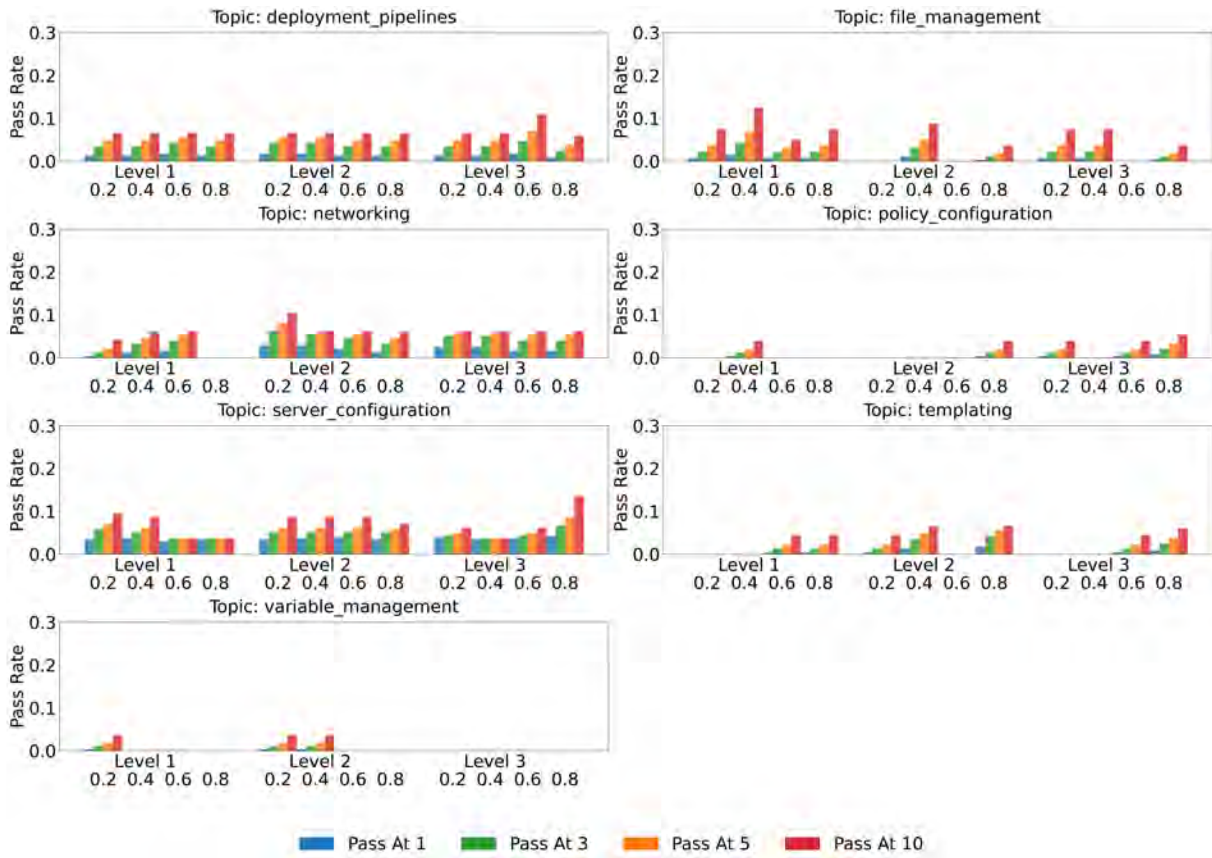


Figure 10: Performance of CodeGemma-7B-it model.

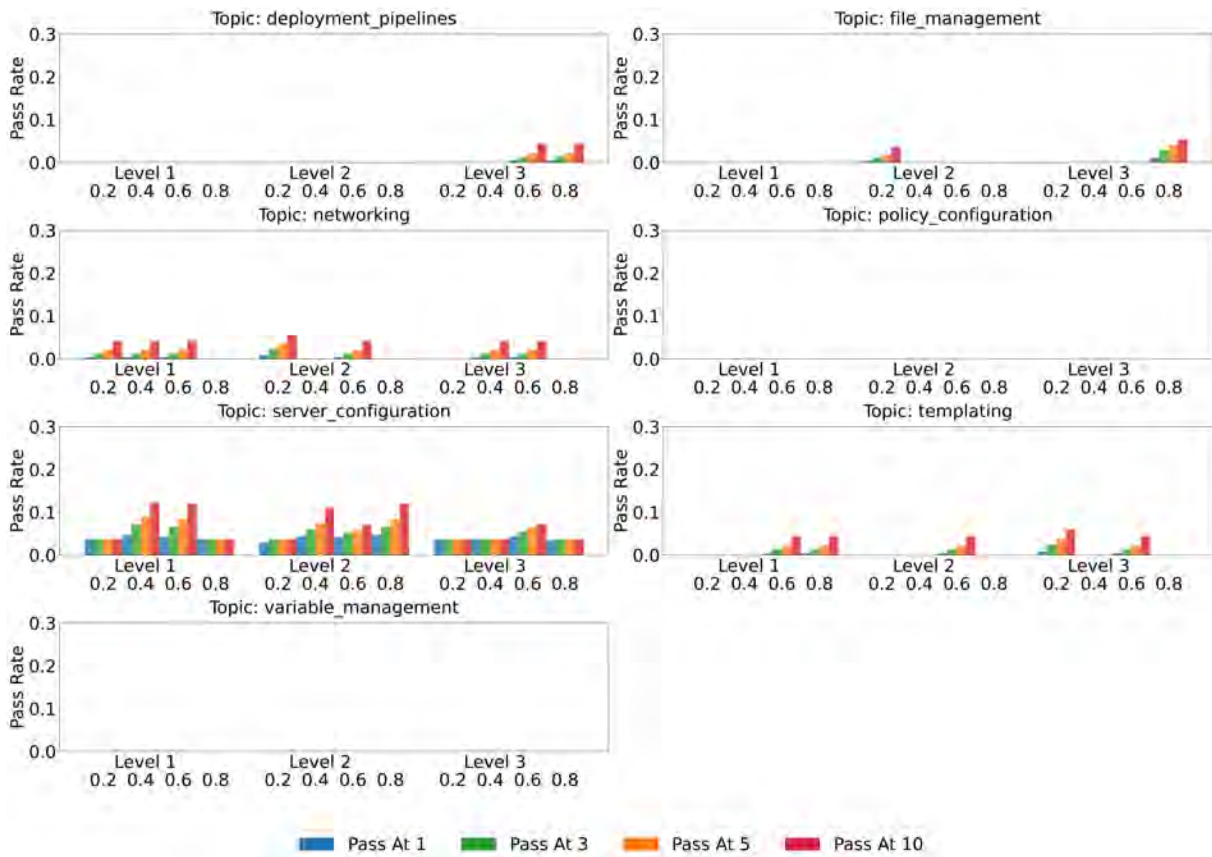


Figure 11: Performance of CodeLLaMa-7B-it model.

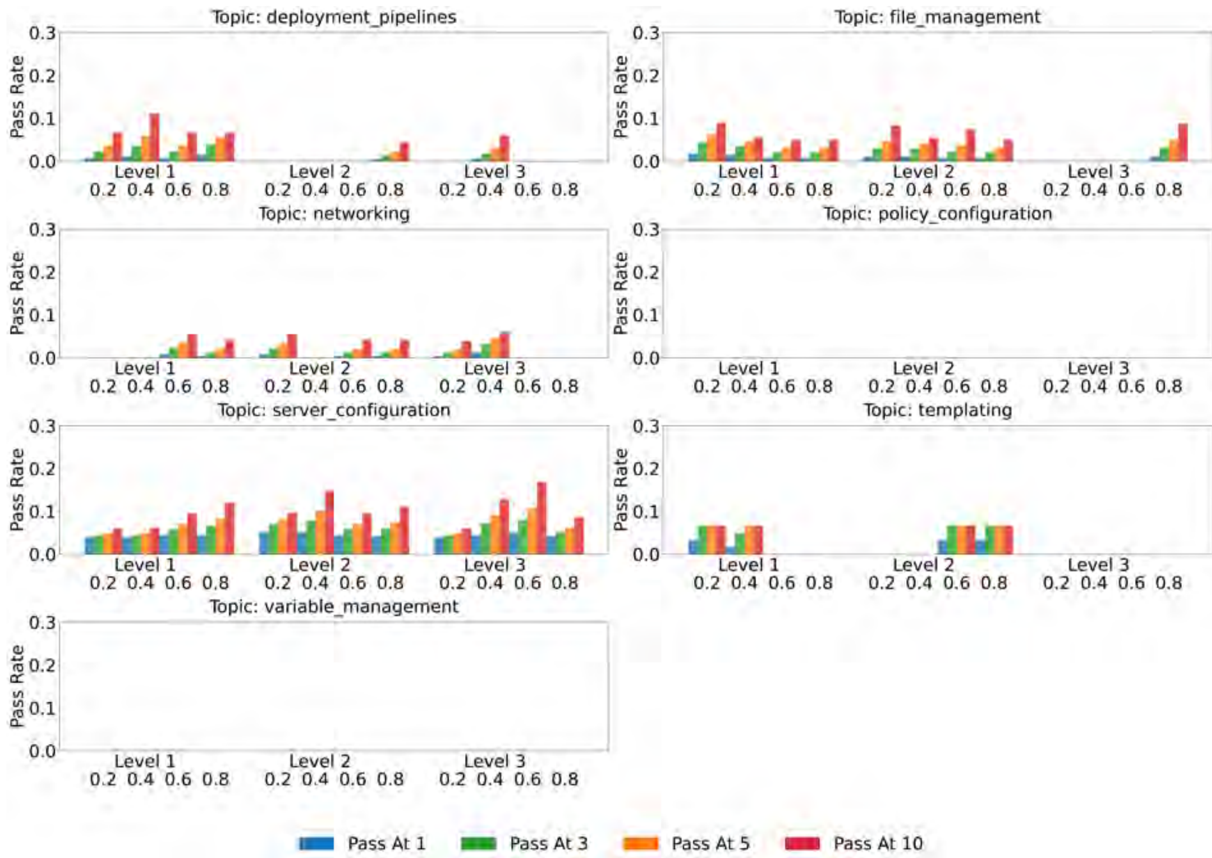


Figure 12: Performance of CodeLLaMa-13B-it model.

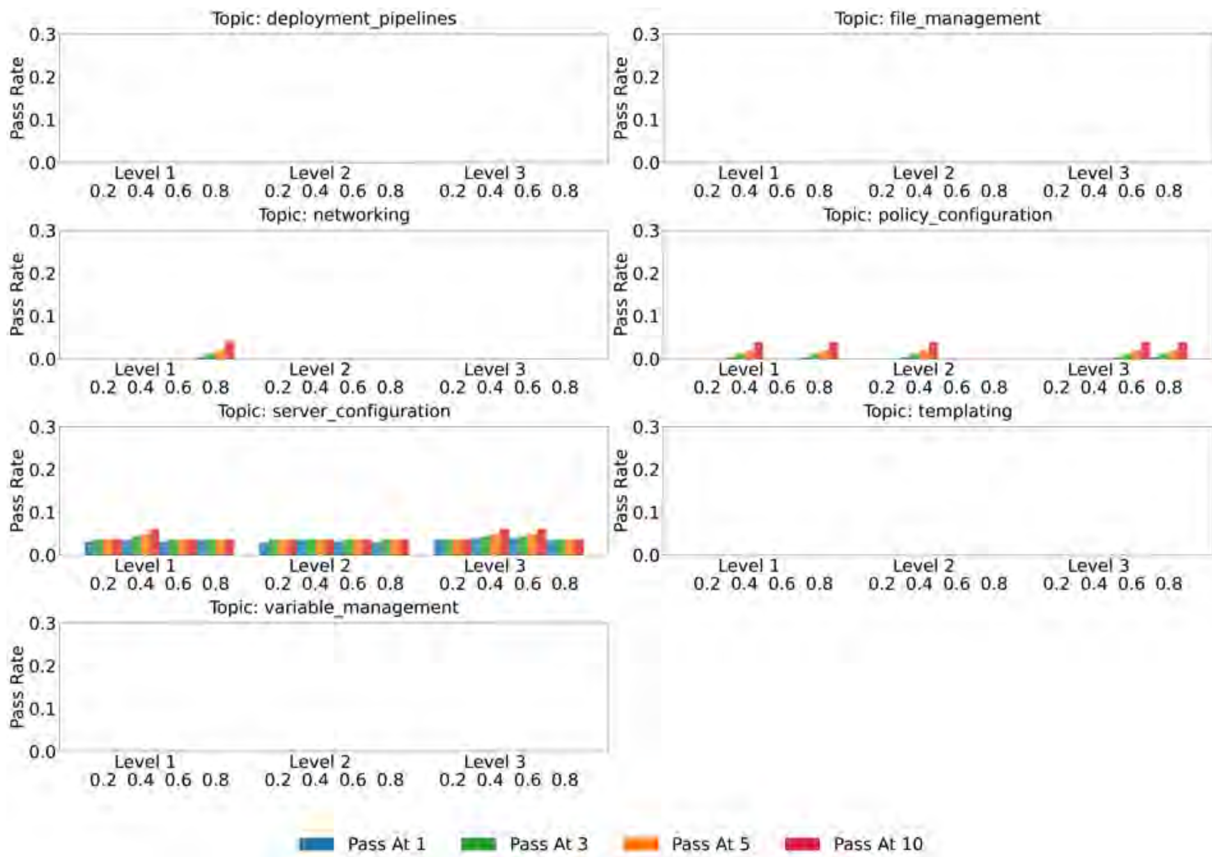


Figure 13: Performance of DeepSeek-R1-Distil-LLaMa-8B model.

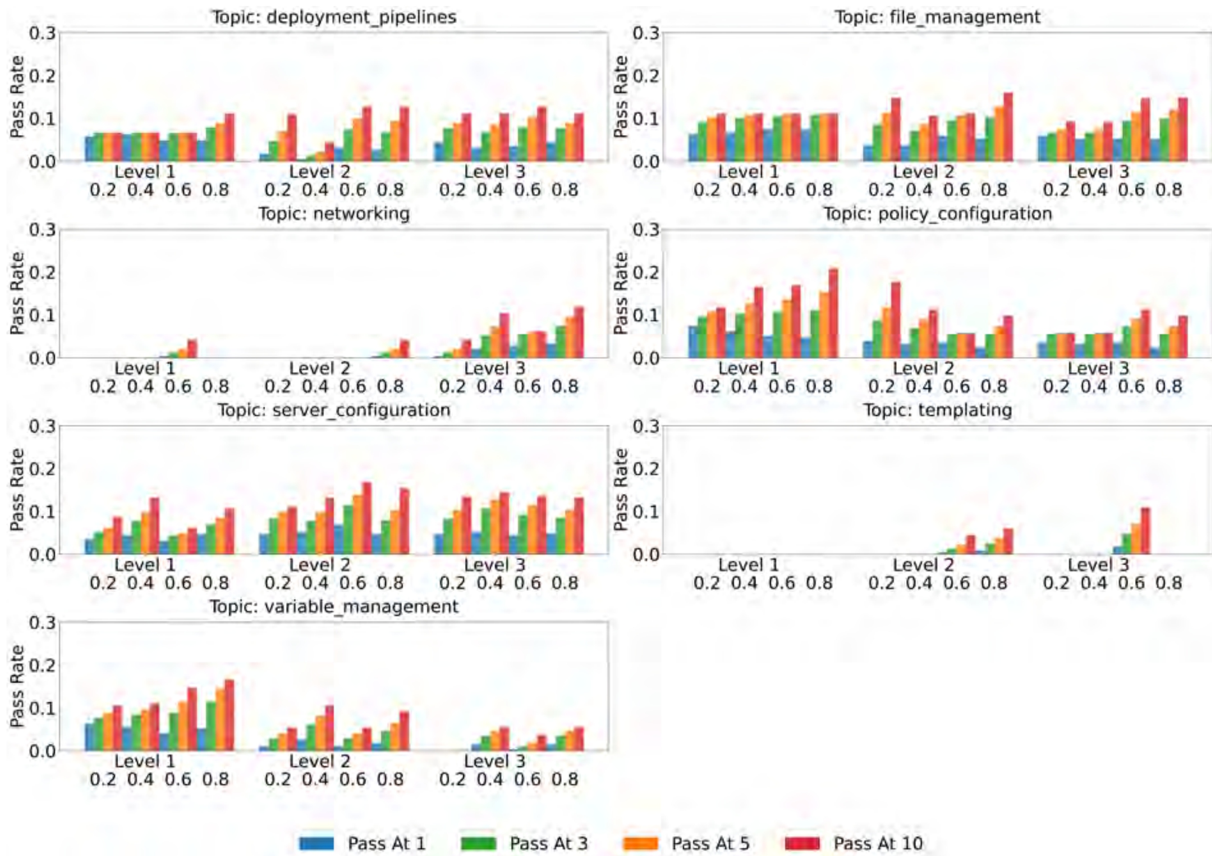


Figure 14: Performance of DeepSeek-Coder-V2-it model.

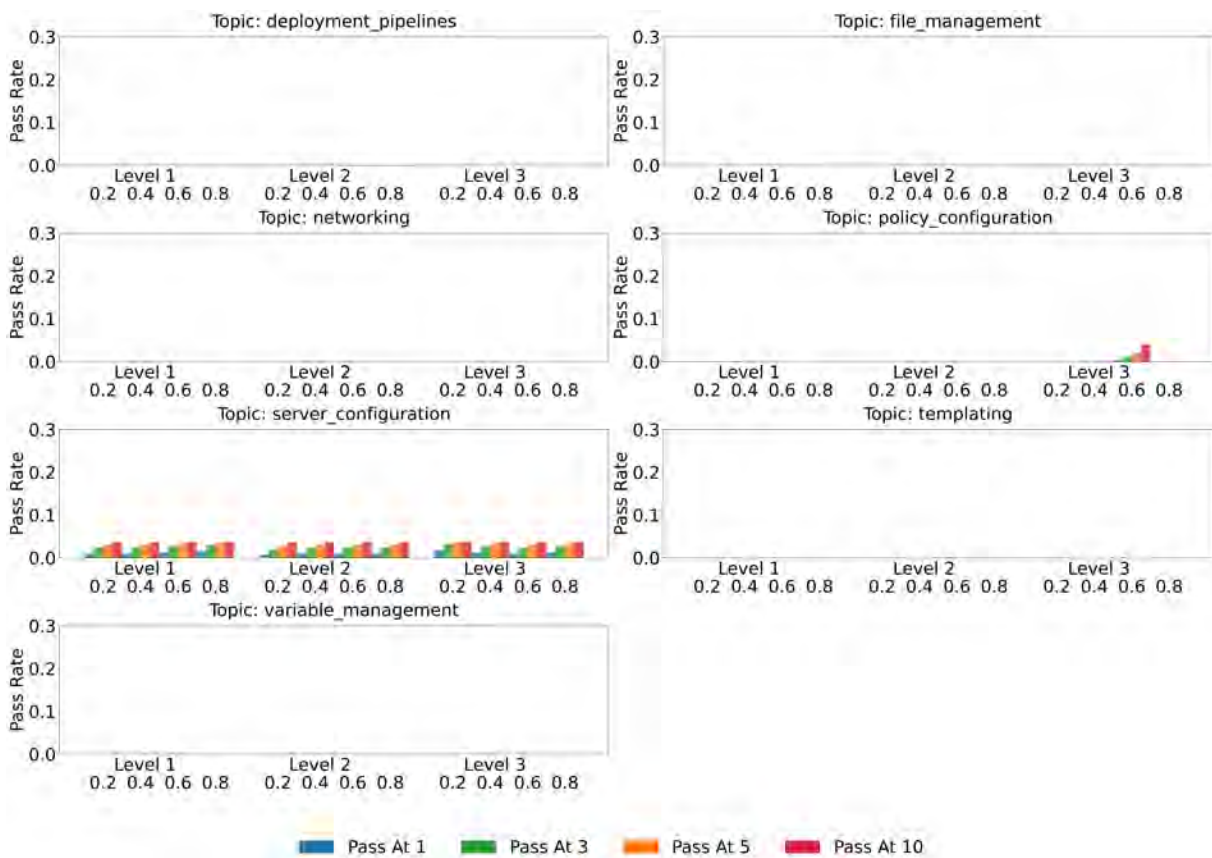


Figure 15: Performance of DeepSeek-R1-Distil-Qwen-7B model.

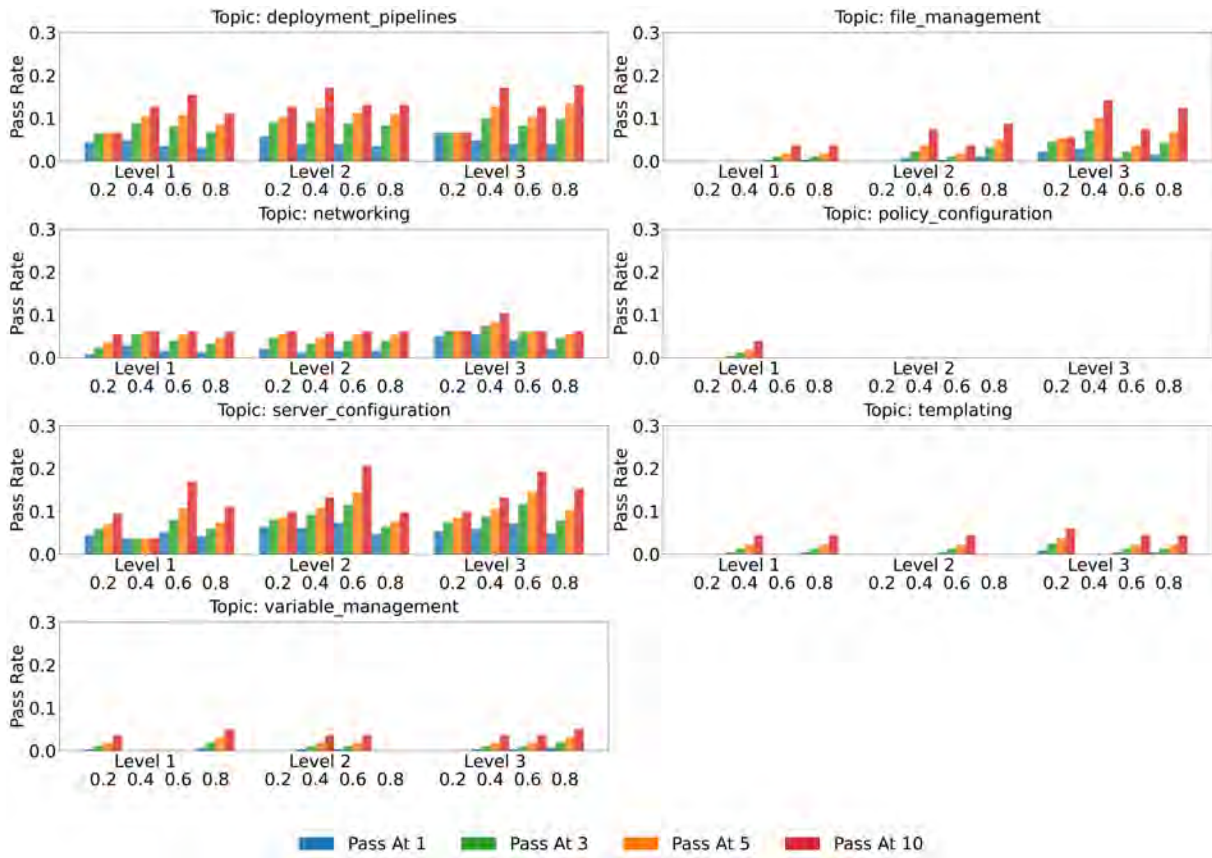


Figure 16: Performance of LLaMa-3.1-8B-it model.

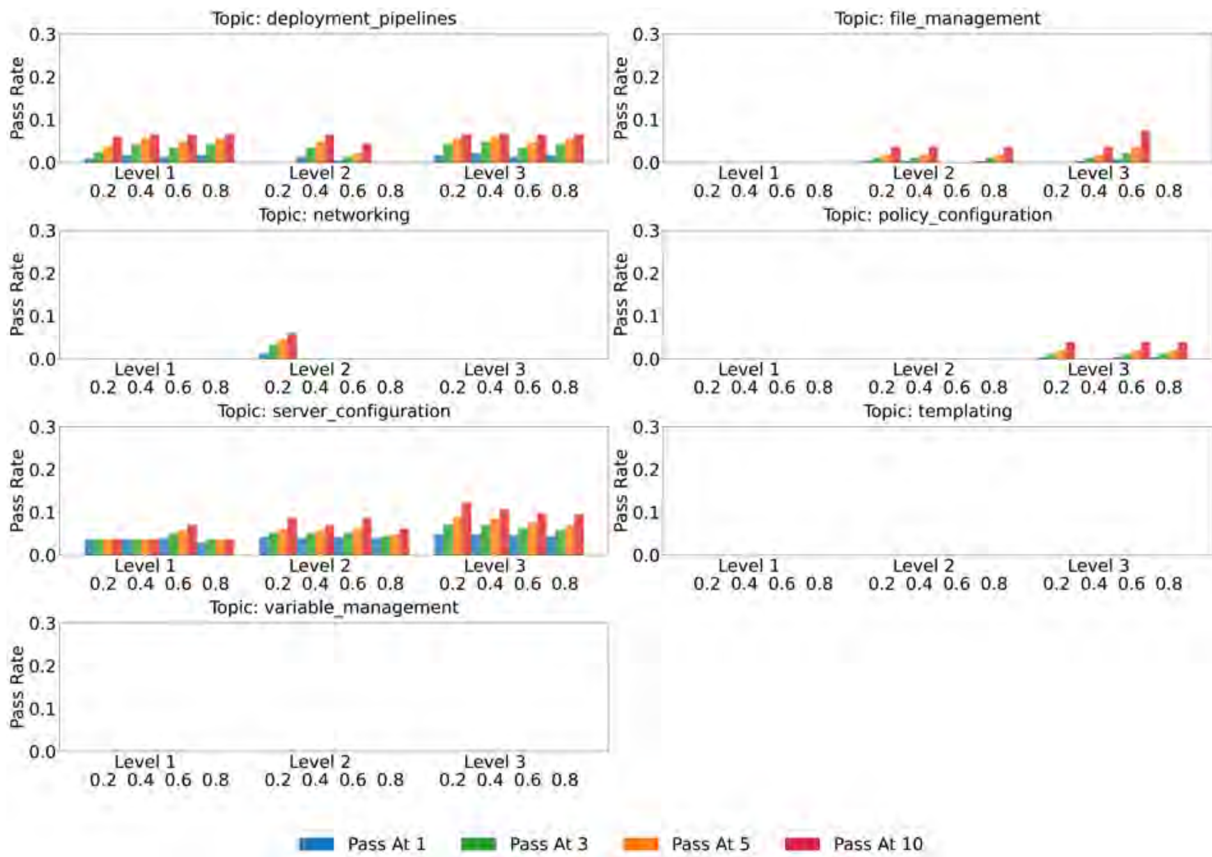


Figure 17: Performance of LLaMa-3.2-3B-it model.

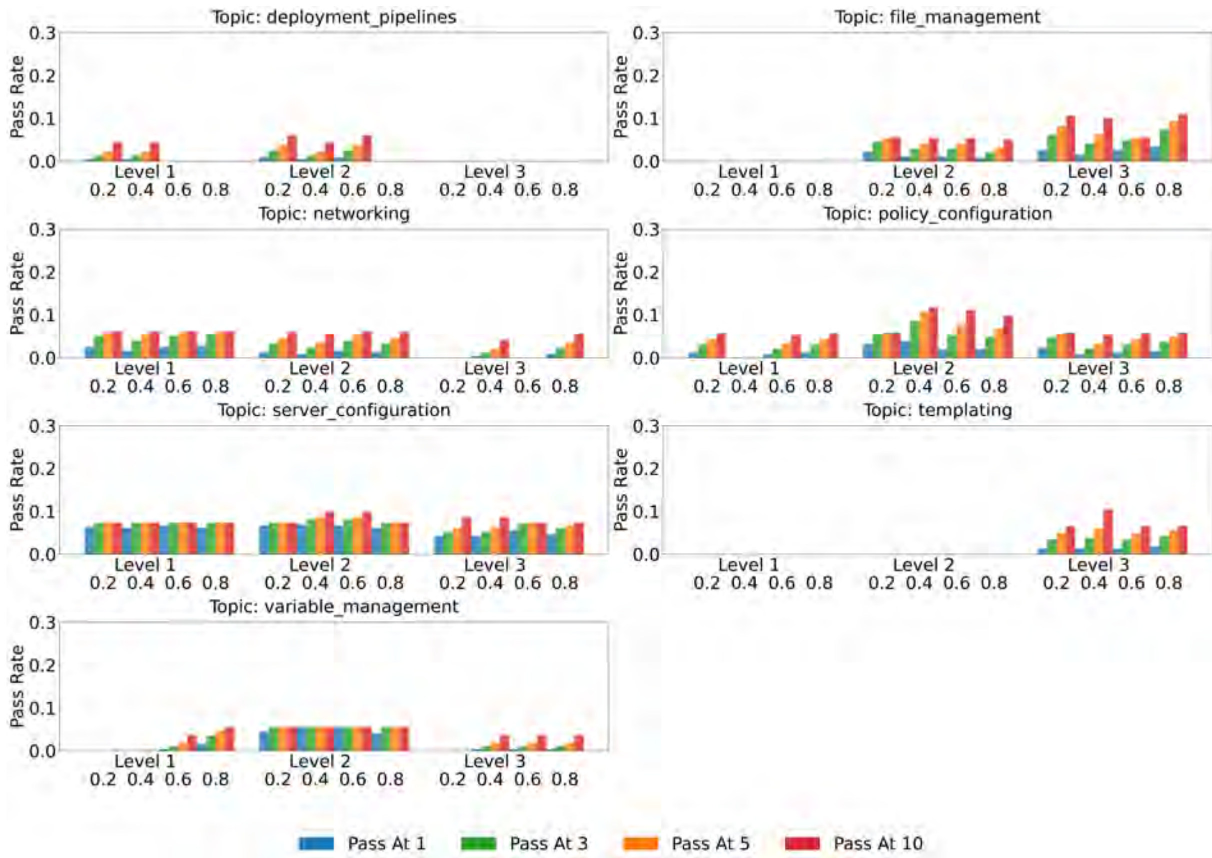


Figure 18: Performance of Microsoft-phi-3.5-Mini-it model.

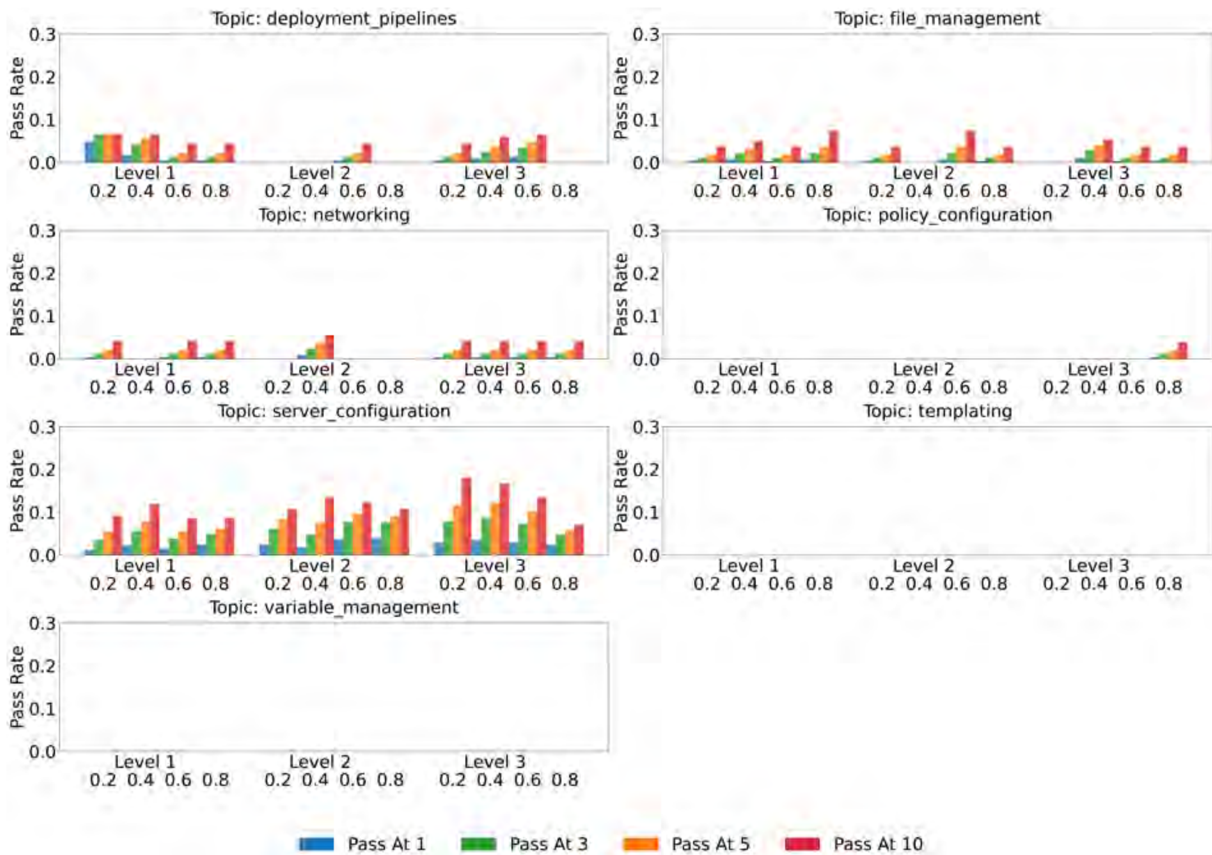


Figure 19: Performance of StarCoder2-7B model.

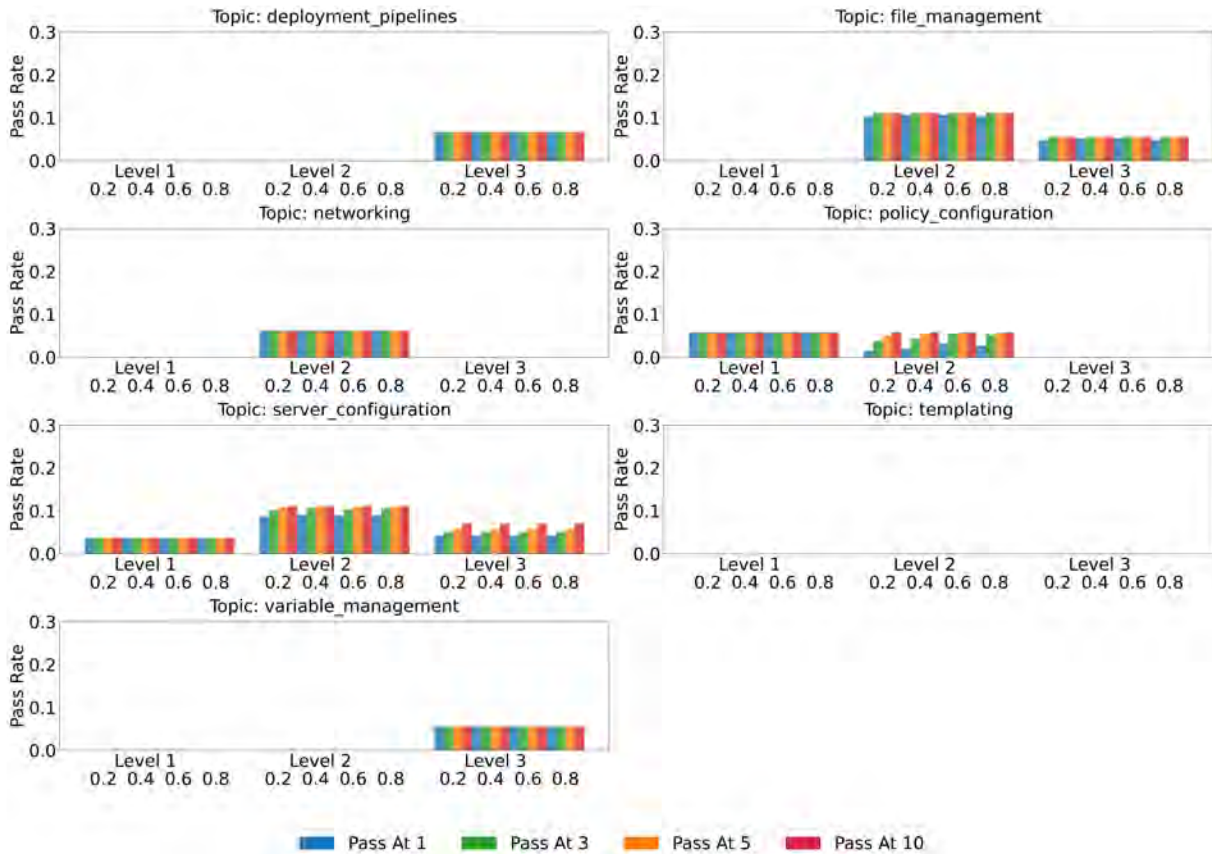


Figure 20: Performance of WizardCoder-15B model.

each category and identifying the best-performing models, other models that successfully solved at least one task, and models that failed to solve any tasks in that category. A task was considered "solved" by a model if it produced at least one functionally correct and idempotent solution (pass@10 > 0, see Section D.1) across all tested prompt and temperature configurations.

M Error Taxonomy

M.1 Overall Trend

As we saw the low pass@k values, we wanted to investigate what really lies behind the numbers. So we did an extensive qualitative study where we studied 1411 failed cases spanning across task categories and models. In the process, we have created a taxonomy of errors in open-source models for Ansible code when solving user-generated real-world issues.

Our error taxonomy spans nine categories that capture the full spectrum of failures in Ansible playbook generation. Most strikingly, reasoning-distilled models exhibit a fundamentally different failure profile than standard code generation models-despite their enhanced reasoning capabilities, they overwhelmingly fail at basic syntax, suggesting that general reasoning abilities do not transfer to domain-specific code generation without appropriate knowledge of the target language.

For standard code generation models, the error distribution reveals more nuanced challenges. Attribute & Parameter Errors emerge as the most prevalent semantic failure mode, where models correctly identify appropriate modules but fail to configure them properly. This suggests that models understand "what" to do but struggle with "how" to do it correctly-a crucial distinction from traditional code generation tasks that often focus on algorithmic logic rather than precise configuration.

Variable handling and path navigation errors collectively account for a substantial portion of failures, highlighting the challenge of state tracking in IaC. Unlike traditional programming, where variable scope is often contained within functions, Ansible's variable handling spans playbooks, roles, and templates, creating a more complex state management problem that current models struggle to navigate.

Template issues, particularly with Jinja2, reveal

a fundamental limitation in models' ability to reason about the interaction between static configuration and dynamic content generation-a core requirement for flexible infrastructure automation. This challenge is compounded by the dual-language nature of templating, where models must simultaneously reason about Ansible's declarative structure and Jinja2's programming constructs.

The distribution of errors across models also reveals architectural influences: some models struggle predominantly with variable issues, while others face challenges with host targeting or attribute configuration. These patterns suggest that different pre-training approaches and model architectures may create distinct blind spots in handling IaC's unique requirements.

To better understand the failure modes of LLM-generated Ansible playbooks, we performed a systematic error analysis for each IaC category. For each category, we selected a representative task and, for each model, sampled 15 failed playbooks stratified by temperature and TELeR prompt level. Each sampled playbook was manually reviewed and categorized by error type. The following subsections and tables summarize the most frequent error patterns observed for each model in each category.

M.1.1 Networking: Task ID 24269533

This networking task involves defining two variables and printing them, making it relatively straightforward. As expected, the pass rate across models was relatively high. However, common error types still emerged across models, often related to basic syntax, variable definition, and adherence to constraints.

Many models, such as CodeGemma and CodeLLaMa variants, failed by referencing undefined or out-of-scope variables. Several models (e.g., CodeLLaMa, LLaMA-3.2-3B, Phi-3.5, WizardCoder) included tasks that were explicitly prohibited in the prompt, such as provisioning a web server, indicating prompt misunderstanding or prompt misalignment. Invalid host configuration and YAML syntax issues also appeared frequently (notably in Qwen-Instruct and StarCoder2), which hindered execution even when the task logic was close to correct. A few models, such as DeepSeek-R1-Distil-Qwen, provided inconsistent or incomplete playbook structures without producing valid YAML. Overall, the results highlight that even in simple tasks, LLMs often struggle with instruction-following discipline and basic Ansible conventions.

M.1.2 Policy configuration: 43065965

We analyzed the policy configuration task (Task ID: 43065965) across nine models to assess how well they handle structured API requests, variable access, and output formatting. Despite being relatively straightforward, the task revealed consistent model deficiencies. Most models failed due to incorrect or undefined variables, improper templating logic, and omission of key steps like printing the response. Notably, even high-performing models such as LLaMA 3.1 8B and WizardCoder hallucinated logic or over-templatized their script.

M.1.3 Templating: Task ID 43628823

Templating tasks revealed consistent and widespread failure patterns across nearly all models. The core of this task revolves around checking for file existence and formatting output appropriately—yet many models exhibited significant challenges with YAML validity, variable access, and templating logic. Common problems included the use of undefined or out-of-scope variables, incorrect output formatting, unsupported parameters in file-finding modules, and incomplete or malformed playbooks.

Models such as WizardCoder and Vicuna consistently failed due to incorrect output formats and improper attribute access. Instruction-tuned models like CodeLLaMa and LLaMA-3.1/3.2 also struggled with hallucinating unsupported attributes or creating invalid YAML. While models like CodeGemma and Phi-3.5 demonstrated better intent alignment, they often failed due to weak error handling, incorrect assumptions about Ansible variable structures, or incomplete implementations. Notably, DeepSeek-R1-Distil-Qwen-7B failed to generate even syntactically valid playbooks, suggesting a lack of training exposure to structured IaC generation.

M.1.4 Deployment Pipeline: Task ID 63688612

This task evaluates whether models can configure a multi-step deployment process by writing outputs to files on Linux-based compute nodes. A striking trend across nearly all models was the repeated misuse of Windows-specific Ansible modules (e.g., `win_stat`, `win_copy`) in a clearly defined Linux environment. This mistake appeared even in top-tier models like LLaMA-3.1-8B and

CodeGemma-7B, indicating possible memorization from Windows-heavy training data.

Additionally, a number of models—such as Phi-3.5 and Qwen-Coder—failed due to logic errors where file write operations were skipped, or unsupported Ansible patterns like applying `line_in_file` on non-existent files. Some models hallucinated variables or services (Qwen-Coder), or misunderstood role delegation instructions. Others, such as DeepSeek-R1 variants and StarCoder2, produced incomplete or structurally incoherent playbooks, with no semantic alignment to the task description. WizardCoder notably failed to follow the code formatting template, blending text and code inappropriately. Overall, this task highlighted not just module-level errors but also broader issues with infrastructure assumptions and procedural execution logic.