

How Do LLMs Fail In Agentic Scenarios?

A Qualitative Analysis of Success and Failure Scenarios of Various LLMs in Agentic Simulations

JV Roig
 Kamiwaza AI
 jv@kamiwaza.ai

December 2025

Abstract

We investigate how large language models (LLMs) fail when operating as autonomous agents with tool-use capabilities. Using the Kamiwaza Agentic Merit Index (KAMI) v0.1 benchmark, we analyze 900 execution traces from three representative models - Granite 4 Small, Llama 4 Maverick, and DeepSeek V3.1 - across filesystem, text extraction, CSV analysis, and SQL scenarios. Rather than focusing on aggregate scores, we perform fine-grained, per-trial behavioral analysis to surface the strategies that enable successful multi-step tool execution and the recurrent failure modes that undermine reliability. Our findings show that model scale alone does not predict agentic robustness: Llama 4 Maverick (400B) performs only marginally better than Granite 4 Small (32B) in some uncertainty-driven tasks, while DeepSeek V3.1's superior reliability derives primarily from post-training reinforcement learning rather than architecture or size. Across models, we identify four recurring failure archetypes: premature action without grounding, over-helpfulness that substitutes missing entities, vulnerability to distractor-induced context pollution, and fragile execution under load. These patterns highlight the need for agentic evaluation methods that emphasize interactive grounding, recovery behavior, and environment-aware adaptation, suggesting that reliable enterprise deployment requires not just stronger models but deliberate training and design choices that reinforce verification, constraint discovery, and adherence to source-of-truth data.

1 Introduction

As large language models (LLMs) are increasingly deployed as autonomous agents in enterprise settings, understanding how these systems fail becomes critical for reliable deployment. Agentic AI systems - those capable of multi-step tool use, decision-making under uncertainty, and interaction with external environments - present evaluation challenges that traditional LLM benchmarks were not designed to address.

1.1 The Limitations of Traditional Benchmarks

Traditional LLM benchmarks suffer from well-documented limitations that undermine their utility for assessing real-world agentic capabilities [8]. Two categories of issues are particularly relevant:

Benchmark data contamination. When benchmark test data appears in training corpora - whether accidentally or deliberately - models achieve inflated scores that do not reflect genuine capability improvements. This problem has grown more acute as training datasets have expanded and as benchmark datasets have proliferated across the internet. Multiple studies have documented contamination across widely-used benchmarks, casting doubt on reported performance gains, producing effects similar to overfitting and memorization. [11, 13, 15, 17, 22, 26, 28–31].

Construct validity failures. Many benchmarks measure proxies for capability rather than the capabilities themselves [12, 20, 21]. Static question-answer formats, narrow task distributions, and evaluation metrics that reward surface-level pattern matching can all produce scores that correlate poorly with practical utility. A model that excels at multiple-choice reasoning benchmarks may still fail at the mundane but critical tasks that constitute real enterprise workflows: reading CSV files, querying databases, manipulating filesystems, and combining information across sources.

The disconnect between benchmark performance and real-world utility is not merely theoretical. Analysis of the initial 35 model configurations in the Kamiwaza Agentic Merit Index (KAMI) v0.1 benchmark revealed a persistent “agentic disconnect”: models that excel on standard benchmarks - including aggregated meta-benchmarks like the Artificial Analysis Intelligence Index [5] - often underperform on enterprise-relevant agentic tasks [24]. Conversely, some models dismissed by conventional metrics demonstrate strong, reliable performance in practical workflows. Notably, newer-generation models do not always outperform their predecessors (e.g., Qwen3 models underperforming Qwen2.5 variants), and larger parameter counts do not guarantee superior agentic capability.

1.2 Beyond Scores: Understanding Failure Modes

Even when benchmarks accurately measure agentic performance, aggregate scores obscure the behavioral patterns that determine reliability. A model achieving 75% accuracy on a benchmark tells us nothing about *how* it fails in the remaining 25% of cases - whether failures are random, systematic, recoverable, or catastrophic. For enterprise deployment, understanding failure modes is often more important than knowing success rates - understanding can lead to successful mitigation engineering and an ultimately reliable deployment.

This gap motivates the present study. Rather than reporting benchmark scores, we perform granular, per-sample behavioral analysis of agentic execution traces to identify recurring patterns of success and failure. Our goal is to surface the strategies that enable reliable agentic performance and the failure modes that undermine it - insights that aggregate metrics cannot provide.

Across 900 agentic execution traces, we observe four recurring archetypes that cut across model families and task types: (1) premature action without grounding, such as guessing schemas instead of inspecting them; (2) over-helpfulness under uncertainty, where models substitute missing entities with plausible alternatives; (3) vulnerability to context pollution, in which distractor data induces incorrect reasoning; and (4) fragile execution under cognitive load, including malformed tool calls, generation loops, and inconsistent recovery from errors. Introducing these archetypes early helps motivate our qualitative approach and frames the detailed results that follow.

While prior work has highlighted anecdotal or model-specific accounts of agentic failures, to our knowledge there has been no systematic, cross-model, trace-level analysis conducted within controlled, repeatable environments. The interactive, multi-step structure of KAMI v0.1 enables such analysis at scale. Although the present study represents only an initial step - limited by the manual nature of our qualitative trace review - it lays the groundwork for broader, AI-augmented large-scale analyses to follow.

1.3 Contributions

Our goal is not simply to measure accuracy, but to characterize the behavioral strategies that underlie agentic success and failure. This paper makes the following contributions:

1. **Qualitative analysis of agentic behavior at scale.** We manually analyze 900 agentic execution traces across three models and ten scenarios from KAMI v0.1, identifying behavioral patterns that benchmark scores obscure.
2. **Taxonomy of success strategies and failure modes.** We document recurring patterns including tool use avoidance (models defaulting to inference when tools are available), semantic confusion or over-interpretation (plausible but incorrect assumptions about data values), last-mile execution failures (correct reasoning with incorrect output), context pollution vulnerability (distraction by irrelevant information), and coherence degradation under extended operation.
3. **Evidence that scale alone predicts neither agentic fitness nor reliability.** We show that Llama 4 Maverick (400B total parameters) achieves only marginal improvements over Granite 4 Small (32B parameters) on a few key scenarios, while DeepSeek V3.1's superior performance traces to post-training reinforcement learning rather than raw scale - evidenced by comparison with the architecturally identical but lower-scoring DeepSeek V3.
4. **Emergent principles for reliable agentic deployment.** We synthesize our findings into four principles, summarized in Section 5.2, and translate these into practical recommendations for enterprise deployment in Section 5.3.

1.4 Paper Organization

The remainder of this paper is organized as follows. Section 2 describes our methodology, including model selection, scenario selection, trial configuration, and analysis approach. Section 3 details the ten KAMI v0.1 scenarios analyzed and the tools available to models. Section 4 presents our per-model, per-scenario observations and the success and failure patterns identified. Section 5 synthesizes findings into broader themes and design implications. Section 6 discusses threats to validity. Section 7 concludes.

2 Methodology

This study performs qualitative analysis of LLM behavior in agentic scenarios, examining not just whether models succeed or fail, but *how* they succeed and fail. While benchmark scores provide aggregate performance measures, they obscure the behavioral patterns that determine reliability in real-world deployments. Our goal is to surface recurring success strategies and failure modes that can inform both model development and agentic system design.

2.1 Model Selection

We analyze three models from the almost 60 model configurations currently evaluated using the full KAMI v0.1 benchmark as of the time of writing. Given the time-intensive nature of manual trace analysis (discussed in Section 6), we selected models representing different performance bands:

- **Granite 4 Small** (32B parameters, dense) - a smaller enterprise-focused model, representing below-mean KAMI performance.
- **Llama 4 Maverick** (400B total / 17B active, MoE) - a large-scale open-weights model, representing above-mean performance.
- **DeepSeek V3.1** (671B total / 37B active, MoE) - a model with extensive post-training reinforcement learning, representing top-tier performance.

This selection spans dense and mixture-of-experts architectures, parameter counts from 32B to 671B, and a wide range of benchmark scores (58.5% to 92.2% pooled accuracy on KAMI v0.1). Notably, the inclusion of DeepSeek V3.1 alongside the original DeepSeek V3 in our broader dataset allows comparison of models with identical architecture but different post-training regimes.

2.2 Scenario Selection

We analyze 10 of the 19 scenarios in KAMI v0.1, specifically the 200, 300, 400, and 500 series. These scenarios were selected because they directly evaluate agentic capabilities:

- **200 series** (Q201, Q202): Filesystem operations - creating directories and files, including implicit requirements not stated in the task.
- **300 series** (Q301, Q302): Text extraction - retrieving specific lines from large files, requiring proper tool use rather than inference-based guessing.
- **400 series** (Q401, Q402): CSV analysis - computing derived statistics across one or more files, requiring the model to recognize that Python is necessary for accurate computation.
- **500 series** (Q501, Q502, Q503): SQL analysis - querying databases with schema discovery, joins, and aggregations, including scenarios with distractor tables.

The excluded scenarios serve different evaluation purposes: the 100 series is a non-agentic sanity check evaluating whether models unnecessarily invoke tools; the 600 series tests the effect of explicit prompting design to make 500-series tasks hypothetically easier; and the 700 series tests output format compliance rather than agentic behavior. Full scenario specifications follow the PICARD framework [23] and are detailed in Section 3.

2.3 Trial Configuration

Each model was tested against each scenario multiple times using randomized task parameters (file names, text content, CSV data, and database records, etc.) to account for LLM stochasticity and to probe real-world capability rather than memorized responses, following PICARD principles [23]. The full KAMI v0.1 dataset contains approximately 240 trials per model per scenario. For this qualitative analysis, we randomly sampled 30 trials per model per scenario (12.5% of the total), yielding 900 trials for manual analysis.

Key execution parameters, consistent across all trials in KAMI v0.1:

- **Maximum rounds:** 20 inference rounds per trial, providing a cap on autonomous operation to prevent runaway execution of repetitively failing models.
- **Single-tool-per-round constraint:** Models may invoke only one tool per inference round, encouraging step-by-step execution for a hypothetically more reliable operation.
- **Temperature:** 0.4 for all models. This was a deliberate design choice in KAMI v0.1, a setting between 0.0 and the typical 0.7 default of most models, in order to try to lessen stochasticity without harming problem-solving (the ability to explore more paths when encountering failure). We plan to examine the quantitative and qualitative effects of temperature in agentic performance in future work.

- **Context window:** 32K tokens for non-thinking models; 128K tokens for thinking models.
- **Maximum output tokens:** 8K per round for non-thinking models (sufficient for any correct response; higher outputs typically indicate generation loops or data in-lining failures); unlimited for thinking models.

All three models analyzed in this study (Granite 4 Small, Llama 4 Maverick, DeepSeek V3.1) were run in non-thinking mode. The KAMI v0.1 benchmark and initial results are described in [24].

2.4 Analysis Approach

We performed manual analysis of all 900 sampled execution traces. Each trace contains the complete sequence of model outputs, tool invocations, tool results, and final outcomes for a single trial. For the non-thinking models in this study, individual traces typically span tens of thousands of tokens.

Pattern identification followed an emergent coding approach: rather than applying predefined categories, we read traces sequentially and allowed behavioral patterns to surface organically. When a pattern was observed (e.g., a model confusing semantically similar column names), we noted it and watched for recurrence. As analysis progressed chronologically across models - Granite 4 Small first, then DeepSeek V3.1, then Llama 4 Maverick - we performed cross-model verification, checking whether patterns observed in later models also appeared in earlier ones. This iterative process produced the success and failure pattern taxonomies presented in Section 4.

The chronological structure of our analysis is preserved in the results presentation: observations for later-analyzed models explicitly reference patterns first identified in earlier-analyzed models.

3 Agentic Scenarios Tested and Assigned Tools

In this section, we describe the different agentic scenarios that models were tested against. We also discuss the different tools provided to the models. Some of these tools will be referenced often during the analysis section, and they form the core part of the “agenticness” of the LLM, giving it the ability to affect its environment, see and manipulate data, and ultimately achieve its tasks.

3.1 The KAMI v0.1 Scenarios Analyzed

We analyze 10 of the different KAMI v0.1 scenarios (referred to as “question templates” in both the PICARD and KAMIv0.1 papers). Below is a description of each scenario. For the complete PICARD specifications for each scenario - including instruction templates, randomized fields, and the corresponding sandbox environments (text files, CSVs, database schemas, etc.) - see Appendix A.

- **Q201:** The agent is instructed to create two blank files (whose names are randomized per instance), inside a specific directory (whose name is also randomized). That directory is not pre-created, and must also be created by the agent, but this fact is not explicitly declared. Agents can guess correctly that they must first create the directory, or validate this by checking if it exists, or simply adapt and recover if they assume the directory exists and receive an error message when trying to create a file in a non-existent directory.
- **Q202:** A more difficult version of Q201, here there are more directories and files to create, and they are presented as an ASCII tree. Due to the maximum rounds configuration (20 inferences total), there is a risk that too many failed tool calls can immediately doom the attempt, as there would not be enough inference rounds left in the trial to complete the ASCII tree, assuming the model uses the most semantic tools available ('create_directory' and 'write_file'). Models can also successfully complete the task in one inference round by using their 'python_execute_code' tool instead.
- **Q301:** The agent is instructed to retrieve two lines from a large text file, and write the answer as a JSON file with specific keys. Due to the length of the text file and the randomized line numbers being asked generally being well above how much LLMs can accurately count, the core challenge in this task is for the LLM to avoid using normal inference to guesstimate the correct answer, and instead use their 'read_file' tool properly, since that tool has the functionality of selecting specific files using 'start_line' and 'end_line' as optional parameters. Python tool use would also be acceptable (we only grade outcomes, not techniques), although that would be suboptimal (far more tokens needed and more opportunities for failure).
- **Q302:** A more difficult version of Q301, with more lines to retrieve (7 instead of 2) from an even longer text file.

- **Q401:** The agent is instructed to analyze a CSV file and create a JSON file containing two derived data - total count of customers, and the average age of these customers. For the average age, the agent is further instructed to avoid rounding and instead write the value in full floating-point precision. The core challenge in the 400-series scenarios is that unlike in the 200, 300, and 500-series scenarios analyzed for this report, there are no tools specifically available for the task. Instead, it is left up to the model to realize that it must use Python for accurate CSV analysis.
- **Q402:** A more difficult version of Q401, this time with 4 separate CSV files to analyze (up from 1) and 6 questions to answer (up from 2).
- **Q403:** Another variation of the Q401 scenario, there is only one question this time, but the sandbox has 4 CSV files composed of 3 distractors and only 1 file with relevant data. No hint is given to the agent about the distractors. The agent is left to analyze the CSV files to determine which file(s) to use to answer the problem.
- **Q501:** The agent is instructed to find a specific information from the database, and write the answer as a JSON file with a particular key. There are four tables in the database, and the correct answer requires an SQL JOIN to be made across two of the tables. The agent is expected to figure out which of the tables are necessary - no hint is given how many tables are needed and how many are distractors.
- **Q502:** A more difficult version of Q501, there are now 6 questions (up from 1) using the same table schemas as before. Each question needs an independent SQL query to answer, so this scenario tests if an agent can keep to task despite the extended requirements.
- **Q503:** Similar to the Q403 scenario but applied to the 500-series database environment, this scenario asks only one question, but has more distractors. There are 6 tables in the database, only 3 of which are needed in a multi-table join.

3.2 Tools Provided to the Agent

During KAMI v0.1 testing, agents were given 29 tools across 5 different tool categories. In the current KAMI v0.1 test suite, only 3 of these tool categories are ever relevant. 2 of these - web tools and git tools - are never needed.

By design, all 29 tools are always available and loaded into context at the beginning of a task. Although seemingly overkill now, it was a reflection of current agent design during the KAMI v0.1 experiment before the recent Anthropic “Skills” design and subsequent refinement to the MCP Tools standard [1, 10, 27]. However, it should be noted that these 29 tools are very light and did not unreasonably bloat the context. Unlike heavy MCP servers that end up bloating the context by several tens of thousands of tokens, the KAMI v0.1 tool load-out only consumes a little over 2K tokens. The total context size in the first round of inference is typically under 3K tokens, which also includes the non-tools component of the system prompt and the task instruction message (first user message).

Filesystem Tools

- `get_cwd` - Get the current working directory.
Parameters: None
Returns: String - current working directory information
- `read_file` - Read a file with optional line numbering, range selection, and debug formatting.
Parameters: path (required), show_line_numbers, start_line, end_line, show_repr
Returns: String - file contents or error message
- `write_file` - Write content to a file.
Parameters: path (required), content (required)
Returns: String - confirmation or error message
- `append_file` - Append content to an existing file.
Parameters: path (required), content (required)
Returns: String - confirmation or error message
- `edit_file` - Make a line-based edit by replacing old_text with new_text.
Parameters: path (required), old_text (required), new_text (required), dry_run
Returns: String - confirmation with diff or error message
- `create_directory` - Create a new directory.
Parameters: path (required)
Returns: String - confirmation or error message

- `list_directory` - List the contents of a directory.
 Parameters: `path`
 Returns: String - list of files and directories
- `copy_file` - Copy a file from source to destination.
 Parameters: `source` (required), `destination` (required)
 Returns: String - confirmation or error message
- `remove_file` - Remove/delete a single file.
 Parameters: `path` (required)
 Returns: String - confirmation or error message
- `remove_directory` - Remove/delete a directory and all its contents.
 Parameters: `path` (required)
 Returns: String - confirmation or error message
- `copy_directory` - Copy a directory and all its contents to a new location.
 Parameters: `source` (required), `destination` (required)
 Returns: String - confirmation or error message

Git Tools

- `git_clone` - Clone a git repository using HTTPS.
 Parameters: `repo_url` (required), `target_path`
 Returns: String - confirmation or error message
- `git_commit` - Stage all changes and create a commit.
 Parameters: `message` (required), `path`
 Returns: String - confirmation or error message
- `git_restore` - Restore the repository or specific files to a previous state.
 Parameters: `commit_hash`, `path`, `files`
 Returns: String - confirmation or error message
- `git_push` - Push commits to a remote repository.
 Parameters: `remote`, `branch`, `path`
 Returns: String - confirmation or error message
- `git_log` - Get the commit history of the repository.
 Parameters: `path`, `max_count`, `since`
 Returns: String - JSON formatted commit history
- `git_show` - Get detailed information about a specific commit.
 Parameters: `commit_hash` (required), `path`
 Returns: String - JSON formatted commit details
- `git_status` - Get the current status of the repository.
 Parameters: `path`
 Returns: String - JSON formatted repository status
- `git_diff` - Get the differences between commits, staged changes, or working directory.
 Parameters: `path`, `commit1`, `commit2`, `staged`, `file_path`
 Returns: String - JSON formatted diff information

Web Tools

- `brave_web_search` - Search the web using Brave Search API.
 Parameters: `query` (required), `count`
 Returns: Object - JSON object containing search results
- `fetch_web_page` - Fetch content from a specified URL.
 Parameters: `url` (required), `headers`, `timeout`, `clean`
 Returns: String - cleaned web page content or error object

Python Tools

- `python_execute_file` - Execute a Python file and return its output.
Parameters: `file_path` (required)
Returns: String - execution output or error message
- `python_check_syntax` - Check the syntax of Python code.
Parameters: `code, file_path`
Returns: String - result of the syntax check
- `python_execute_code` - Execute arbitrary Python code and return its output.
Parameters: `code` (required)
Returns: String - execution output or error message

SQLite Tools

- `sqlite_connect` - Connect to a SQLite database file and verify the connection.
Parameters: `database_path` (required)
Returns: String - confirmation with database info or error message
- `sqlite_execute_query` - Execute a SELECT query on SQLite database.
Parameters: `database_path` (required), `query` (required), `limit`, `timeout`
Returns: String - JSON formatted results or error message
- `sqlite_execute_command` - Execute INSERT, UPDATE, DELETE, or DDL commands.
Parameters: `database_path` (required), `command` (required), `timeout`
Returns: String - confirmation with affected rows or error message
- `sqlite_get_schema` - Get the complete database schema.
Parameters: `database_path` (required)
Returns: String - JSON formatted schema information or error message
- `sqlite_list_tables` - List all tables and views in the database.
Parameters: `database_path` (required)
Returns: String - JSON formatted list of tables and views or error message

For the complete tool details, see Appendix B.

4 Results and Analysis

In this section, we discuss the observed success and failure patterns of various LLMs in the 10 scenarios from the KAMI v0.1 agentic simulation test suite discussed in Section 3.1. Each LLM is repeatedly tested against each scenario multiple times, typically over 200 times each scenario. This repetition is not only to account for LLM stochasticity, but also to exhaustively test the LLM’s real-world capability as per the principles of the PICARD framework [23].

Given the vast amount of data, this paper analyzes only a subset of our existing KAMI v0.1 results. We chose 3 models out of almost 60 tested models at the time of writing. These 3 models represent different bands of KAMI performance - representatives from the below the current mean, up to the highest tier tier of LLM performance. We also analyzed only a random sample of the full results, typically on the order of 30 random samples out of 240 per scenario for each model. (12.5% of the all samples per model and scenario).

Table 1 lists the models and their overall KAMI v0.1 benchmark scores.

Model	Pooled Accuracy	Std Dev	RSE	95% t-CI
Granite 4 Small	58.5%	±0.99%	±26.7%	(57.5%, 59.3%)
Llama 4 Maverick	74.6%	±0.93%	±26.7%	(73.8%, 75.3%)
DeepSeek V3.1	92.2%	±1.16%	±26.7%	(91.2%, 93.2%)
DeepSeek V3	59.4%	±0.39%	±26.7%	(59.0%, 59.7%)

Table 1: KAMI v0.1 scores (full test suite, 19 agentic scenarios) of the three analyzed models, plus DeepSeek V3 (bottom) for comparison.

Note that Table 1 shows the overall score of each model in the full KAMI v0.1 test scenarios (19 in total), showing their overall agentic performance relative to each other, beyond just the subset of 10 test scenarios used for this analysis. The full KAMI v0.1 benchmark and findings are discussed in the original KAMI v0.1 paper [24].

4.1 Granite 4 Small - Success and Failure Patterns

Granite 4 Small is a 32B parameter model, the latest in a series of LLM models from IBM, “developed with a particular emphasis on essential tasks for agentic workflows” [9]. The biggest of the currently released Granite 4 models, Granite 4 Small shows decent performance in KAMI v0.1 with a 58.5% pooled accuracy. As shown in Table 1, the emphasis on agentic training seems to boost Granite 4 Small to perform almost as well as the original DeepSeek V3 (a much older model released before long-horizon agentic post-training became in vogue) despite the 20 \times gap in parameter size.

4.1.1 Summary of Observed Behavioral Patterns

Across the ten scenarios analyzed, Granite 4 Small exhibits several consistent strengths and weaknesses:

- **Strong performance on simpler tasks.** Granite 4 Small performs reliably on straightforward filesystem tasks (Q201–Q202), succeeding in 29/30 samples by proactively creating directories and issuing correct tool calls.
- **Clean execution when task semantics are unambiguous.** In text-retrieval scenarios (Q301–Q302), the model often shows “clean first-pass execution” with correct sequential tool use and no hesitation.
- **Basic error-recovery capability.** In several Q301 trials, the model successfully recovers from malformed JSON or tool-call formatting issues after receiving explicit error feedback.
- **Over-reliance on normal inference when structured tools are required.** In all CSV-analysis tasks (Q401–Q403), Granite consistently avoids the optimal Python-execution strategy, instead reading the CSV and attempting to “eye-ball” large aggregated values - a strategy that always fails (e.g., miscounting rows or producing approximate float averages).
- **Difficulty managing larger tool workflows.** Granite 4 Small struggles with recovery when repeated tool-call errors occur (e.g., malformed JSON, multiple tool calls, or schema-related errors), often exhausting the 20-round inference limit without adapting.
- **Schema guessing and semantic confusion.** In SQL-based tasks (Q501–Q503), the model repeatedly guesses table and column names instead of inspecting the schema, leading to avoidable errors, incorrect JOIN logic, and misinterpretations such as using ORD_ID instead of ORD_AMT.
- **Vulnerability to context pollution.** Granite 4 Small is easily distracted by irrelevant or distractor tables in Q503, sometimes concluding incorrectly that necessary tables are “missing” despite having retrieved the schema.
- **Partial resilience but fragile long-horizon behavior.** While some successful SQL trials show adaptive schema inspection after initial mistakes, many failures arise from not validating filter values or prematurely resigning after misinterpreting the schema results.

4.1.2 Detailed Per-Scenario Observations

This subsection contains the full per-scenario observations for Granite 4 Small. While the preceding summary highlights the dominant behavioral patterns, the detailed traces below show the specific tool calls, error feedback, and reasoning steps that give rise to those patterns.

Q201 observations

In this easy scenario, Granite 4 Small was successful 29/30 times.

The success pattern starts with Granite 4 Small creating the directory first as expected, then moving on to creating the files inside the directory.

The single failure pattern observed is an instance of a failed tool call that Granite 4 Small fails to recover from despite trying over and over (until max inference rounds is exhausted and the test stops).

In this example, the tool call request that the LLM formed was:

```
<tool_call>
{"name": "create_directory",
 "input": {
     "path": "/home/jv/picard1/test_artifacts/q201_s7/turtle"
 }
</tool_call>
```

Note the missing final closing curly brace, making this invalid JSON.

In our universal tool parser, this results in the following tool error message:

```
Tool result: No valid tool call found. Please make sure tool request is valid JSON,  
and escape necessary characters. Try again with better-formatted JSON
```

Despite 19 attempts to recover, Granite 4 Small either keeps repeating the same mistake with no change, or attempts to try different escaping (unnecessary). Halfway, Granite 4 Small also ends up changing the tool - instead of ‘create_directory’, it instead tries ‘write_file’ (wrong), but this tool call also ends up with the same missing closing curly brace.

Q202 observations

Here, Granite 4 Small’s success pattern is unchanged from Q201. It proactively creates directories first, then the files inside them (as applicable). It succeeds in 29/30 samples.

The one failure observed has a different pattern. Instead of a failed tool call, it misinterprets the ASCII tree structure. Specifically, it creates the nested ‘vault/vault’ entry as a blank file when the specification indicates it should be an empty DIRECTORY.

Q301 observations

In 24 out of 30 samples analyzed, Granite 4 Small results in a successful trial. Most of these can be characterized as a “clean first-pass execution” where the model immediately understands task semantics (extract two specific lines from a file and return as JSON), invokes tools sequentially with correct parameters, and completes task on first attempt without any errors or hesitation.

Aside from the clean first-pass success pattern, there are also couple of successful samples where the model displayed “resilient error recovery behavior” - the model encounters JSON formatting errors (missing closing braces, as in Q201) on initial tool invocation, receives error feedback from system, immediately or eventually recognizes the formatting issue, and corrects it and moves on to a successful trial.

We also observed one successful trial where the model succeeded while using a suboptimal strategy. Instead of making two targeted single-line read calls, model reads a range of lines containing both target lines, then correctly extracts the specific lines needed from the returned range. This is incredibly wasteful and unnecessary:

```
<tool_call>  
{  
    "name": "read_file",  
    "input": {  
        "path": "/home/jv/picard1/test_artifacts/q301_s12/device/fox.txt",  
        "start_line": 25,  
        "end_line": 53  
    }  
}  
</tool_call>
```

In contrast, the optimal strategy (which it does in the remaining 23 successful samples) is to use the ‘read_file’ tool twice (once per inference only given the constraint against parallel tool calls in our current experimental setup):

```
<tool_call>  
{  
    "name": "read_file",  
    "input": {  
        "path": "/home/jv/picard1/test_artifacts/q301_s9/dark/study.txt",  
        "start_line": 26,  
        "end_line": 26  
    }  
}  
</tool_call>
```

```
=====  
[RECEIVE TOOL RESULTS]  
=====
```

```
<tool_call>  
{  
    "name": "read_file",
```

```

    "input": {
        "path": "/home/jv/picard1/test_artifacts/q301_s9/dark/study.txt",
        "start_line": 76,
        "end_line": 76
    }
}
</tool_call>

=====
[RECEIVE TOOL RESULTS AGAIN]
=====
```

For the remaining 6 failed samples, the failure pattern is consistently a failure to recover from a tool call error. Model fails to parse or understand error feedback despite explicit error messages (e.g., “Multiple tool calls found. Please only use one tool at a time” or “No valid tool call found. Please make sure tool request is valid JSON”). While we’ve seen two samples in our successful trials that was able to recover, Granite 4 Small seems to be more prone to fail its recovery attempts (6 failed recovery attempts vs 2 successful recovery attempts).

Q302 observations

Here, Granite 4 Small also succeeds in 24 out of 30 samples analyzed.

The success pattern is the optimal solution - **sequential point-read strategy**, using the ‘read_file’ tool multiple times to read each of the seven lines requested, one at a time.

The failure pattern observed here (6 out of 30 samples) is using a single, huge range read. The model reads a range of lines containing all seven requested lines (for example, lines 7 to 104, when asked to retrieve line 7, 24, 33, 48, 66, 77 and 104). In the single instance that this was observed in Q301 (where only two lines were requested, hence the correct lines are always the first and last lines), this strategy resulted in a successful trial. Here, this always fails. Due to the amount of lines retrieved, way beyond what LLMs can accurately count through “eye-balling” (i.e. normal inference without the use of tools), this strategy will never work in this scenario.

In 4 of these 6 failures, the LLM manages to get the first requested line (‘line_a’) correct. Every other line (lines ‘b’ to ‘g’) is wrong, even the very last line (which should just have been the final line in the retrieved range). In the remaining 2 failures, all lines were wrong, even the first line. This shows that in retrieving a huge chunk of text, the model can get confused and end up retrieving the wrong lines. Even getting the start line wrong shows that the model counted lines from the retrieved range - very confused behavior.

Q401 observations

Granite 4 Small failed in all 30 samples analyzed.

The main failure pattern is that it uses the wrong approach (and therefore the wrong tools).

The optimal path is to use the ‘python_execute_code’ tool in order to use Python to process the CSV file, and making sure the code explicitly does not round the float value obtained (as per task instruction) and then creates the required JSON file.

Instead, Granite 4 Small commonly uses ‘read_file’ to read the entire CSV, and then immediately uses ‘write_file’ to create the expected JSON file with the required values (total count of customers and average age of the customers). Without using tools, LLMs simply cannot count high enough to be able to “eye-ball” the number of customers (75; the model usually answers 100, which for LLMs is a close enough approximation). Eye-balling the average age of 75 customers is way beyond the model’s innate ability, esp with the full float precision demand. The approximations are eerily close - close enough that it could probably withstand human scrutiny if a human checker were also to simply eye-ball or sanity check without doing real computation. For example, (expected → LLM guess) $46.32 \rightarrow 44.59$, $42.18666666666667 \rightarrow 43.13$, and $45.2533333333333 \rightarrow 45.49$

One other quirk we observed was the choice of a wrong tool before settling on ‘read_file’ to read the CSV. In 9 samples, the model started by trying to use ‘fetch_web_page’, despite the question clearly showing a local, absolute path. In one sample, the model tried to use ‘sqlite_connect’. In all these cases, the model adapts after receiving the first error message, and switches to the ‘read_file’ tool.

This shows the model is capable of recovering from a mistaken tool call. Unfortunately, it never progresses to realizing it should use Python, and instead just consistently uses the “eye-ball” approach.

Q402 observations

Here, Granite 4 Small succeeded in 1 out of 30 samples.

The general failure pattern here is the “eye-ball” approach already described in previous subsections - the model reads the file and eye-balls the answers (normal inference) instead of using Python to process the CSV and calculate the required figures.

There are some notable quirks here, though, that extend even to the lone success case.

In the vast majority of samples analyzed (28 / 30), Granite 4 Small begins by trying to use its SQLite tool to try to examine the CSV files it found.

Sometimes the model recovers, and eventually reads the CSV files correctly using ‘read_file’. In the lone success case, from there it ends up using Python as expected. Most of the time, however, it ends up doing the “eye-ball approach” after reading the CSV files. In other cases, it hallucinates a python file to execute, like ‘analyze.py’. This is either a next-token-prediction behavior overcoming the agentic behavior (after all, executing an aptly-named Python file does sound like the appropriate continuation to the conversation), or simply the effect of wrong tool choice (i.e. it wrongly inferred ‘python_execute_file’ - which needs a python file as an argument - instead of ‘python_execute_code’, which takes arbitrary code as argument)

In the cases where the model does not recover, it just continually uses its SQLite tools to create tables and insert a couple of fake sample data, which it then uses to answer the task, which of course leads to wrong answers and a failed trial.

Q403 observations

Here, Granite 4 Small succeeded in 2 out of 30 samples.

The two successes follow the same general pattern. The model successfully reads the CSV files, identifies which ones contain data relevant to the task, and uses its Python tool to process the relevant CSVs and produce correctly formatted JSON output with proper decimal precision. In both of these samples, the model also hallucinated a python file (‘analyze_orders.py’ in one, ‘regional_analysis.py’ in the other) using the ‘python_execute_file’ tool (similar to behavior found in Q402), but immediately recovered after getting a “File does not exist” error message. In one case, the recovery was in the form of executing Python code (using ‘python_execute_code’) it generated to retrieve the answer as output, then uses ‘write_file’ to write out the retrieved answer into the necessary JSON file. In the other case, the model decided to create the Python file that did not exist (‘regional_analysis.py’) using its ‘write_file’ tool and then called the ‘python_execute_file’ tool to execute the newly created file. These seem to be variations of the recovery behavior observed in Q402 that are slightly less efficient.

A notable difference between the two successful attempts is in their context efficiency. In one of the successful samples, the model simply reads the entire CSV content using ‘read_file’ - seeing both structure and full data. In the other successful sample, the model uses ‘read_file’ smartly to only read partial data (in this case, the first five lines) to examine the structure of the CSV file (“data sampling behavior”) before proceeding to the Python route.

Another notable observation in the “data sampling behavior” success sample was the use of the ‘read_file’ tool to validate the file after the Python file executed successfully. This is notable in that the behavior does not always occur (i.e., in most instances, the model does not double-check its work, as it was not explicitly instructed to do so). We believe this to have only occurred incidentally, and not caused by the data sampling behavior itself.

Notable failure patterns observed are:

- **Generation loop and coherence loss triggered by in-lining large CSV data** - after reading relevant CSV files, the model uses the correct tool (‘python_execute_code’) to immediately execute Python CSV processing code. However, the Python code generation results in a generation loop and coherence loss caused by in-lining the CSV data into the Python code it is writing, instead of simply using Python functions to read the file directly from the filesystem.
- **“Eye-ball approach”** - after reading relevant CSV files, the model attempts to execute a non-existent Python file (e.g. ‘analysis.py’). However, unlike in the samples that eventually recovered, in the failed samples the model abandons the Python-based approach and instead proceeds to guess/infer the answers from the text it previously ingested. Given the complexity of the aggregation, the model is nowhere close to the correct answer. For example, (expected → LLM guess) 920,968. → 305,460.0,
- **Failed Python code** - in some failed samples, the model is squarely in the correct approach - using Python to analyze the relevant CSV files, even using the Python ‘open()’ command instead of in-lining the CSV data - but ended up having wrong escaping for some newline chars (double or triple backslashes), resulting in syntax errors during execution. Despite receiving feedback and repeated attempts until max rounds are exhausted, the model is never able to successfully recover:

```
<tool_call>
{
    "name": "write_file",
    "input": {
        "path": ".../analysis.py",
        "content": "import json\\nwith open('...customers.csv') as f:\\n            customers = [row.split(',') for row in f]\\n            for row in f]\\n            with open('...orders.csv') as f:\\n                orders = [row.split(',') for row in f]\\n                "
    }
}
```

```

</tool_call>
=====
Tool result: File written successfully
=====

<tool_call>
{
    "name": "python_execute_file",
    "input": {"file_path": ".../analysis.py"}
}
</tool_call>
=====
Tool result: Error: unexpected character after line
continuation character (<string>, line 1)

SyntaxError: unexpected character after line
continuation character
=====
```

- **Failed Python code + tool verification before abandonment** - related to the earlier “Failed Python code” and “Eye-ball approach” failure patterns, in this particular variant the model does something peculiar. After encountering a few failed rounds of Python code as in “Failed Python code” failure pattern, it eventually abandons the Python tool, but only after it confirms that the Python tool works with a simple Hello World program, which executes correctly. This suggests that the model may have decided to eye-ball the CSV file because it determined that while the Python tool works fine, it just couldn’t use it correctly:

```

=====
Tool result: Error executing code: unexpected character after
line continuation character (<string>, line 1)

Traceback:
  File "<string>", line 1
    total_value = 123.45 + 67.89\nprint(total_value)
               ^
SyntaxError: unexpected character after line continuation character
=====

<tool_call>
{
    "name": "python_execute_code",
    "input": {
        "code": "print('Hello World')"
    }
}
</tool_call>
=====
Tool result: Standard Output:
Hello World
=====

<tool_call>
{
    "name": "write_file",
    "input": {
        "path": ".../regional_analysis.json",
        "content": "{\"totalRegionalValue\":191.34}"
    }
}
</tool_call>
```

Q501 observations

Here, Granite 4 Small succeeded in 13 out of 30 samples, making it far more successful in the Database/SQLite test than the CSV test. While both sets of questions (400 and 500 series) were designed with similar difficulty in mind, the model may have been helped by having explicit tools for this task (the “sqlite_*” tools), whereas for the CSV task the model needs to determine which of its generic tools to use and then how to use them properly in order to achieve the task. For agentic AI deployment engineers, the lesson here would be to design tools that make it obvious for agents which to choose for particular scenarios.

The success pattern here can be viewed in a glass-half-full or glass-half-empty perspective.

In the optimistic perspective, the model shows resilience and adaptability. It initially guesses the database schema wrong, receives an error message informing it that there is no ‘orders’ table (the actual table name is ‘enterprise_orders’), it then adapts by listing the tables in the database, getting the actual table names, making another query, encountering a column not found error, and then recovering by examining the table schema to get the correct column names and then correctly creates the necessary multi-table query to achieve the task.

In the pessimistic perspective, the resilience and adaptability shown is a by-product of the model acting in a suboptimal (and very non-human-like) way. It starts the task by immediately trying queries with guessed schemas (table and column names), instead of first examining the database schema.

Note that this behavior can be easy to mitigate - simply include in the system prompt or tool definitions or task instructions (as applicable in the specific deployment) the necessity of starting any database analysis by using the schema tool. This is something we explored in the original KAMI v0.1 test suite, which resulted in success rates to increase from 13 out of 30 (Q501) to 30 out of 30 (Q601, the easier version of Q501 with explicit instructions to begin with the schema tool).

The failure patterns observed here are:

- **Constraint abandonment** - here, the model eventually discovers the correct table name, but instead of using the ‘sqlite_get_schema’ tool, it instead drops the constraint to filter for a specific region and proceeds to get the total orders for a threshold using all regions (no region filter).
- **Schema-guessing loop** - here, the model eventually discovers the correct table name, but then keeps guessing the column names instead of successfully recovering by using the ‘sqlite_get_schema’ tool as it does in the successful samples.

Q502 observations

Here, Granite 4 Small succeeded in 5 out of 30 samples.

The success pattern here is identical to Q501 - a “resilience and adaptability” behavior pattern. It always guesses table names and schemas first, but eventually discovers the correct schema.

The failure patterns observed here are:

- **SQL semantic misinterpretation: ID vs Amount** - here, the model makes a curious mistake. Despite eventually seeing the schema, the model uses order ID (‘ORD_ID’) instead of order amount (‘ORD_AMT’) to filter the relevant table given the instruction “*How many orders above [randomized threshold; e.g. ‘24734’] are there from...*”. This is an instruction that would not confuse humans, but perhaps due to the way models are pretrained (e.g. textbooks and tutorials), they do not often see non-round figures as SQL query filters for columns like monetary amount. Note that we have seen similar behavior in other models outside of Granite too, so this curious behavior is not at all unique to Granite.
- **Guessing instead of sampling values** - here, the model gets some wrong answers despite having essentially correct queries because it guessed column values for one or more of the filters (e.g., using ”SW” as the region filter value, instead of sampling the table first to discover what the legitimate values are). This results in erroneous 0 values that the model then never validates in any way.

Q503 observations

Here, Granite 4 Small succeeded in 19 out of 30 samples.

The success pattern here is identical to Q501 and Q502 - a “resilience and adaptability” behavior pattern where it always guesses table names and schemas first, but eventually discovers the correct schema through the correct use of its available tools.

The failure patterns observed here are:

- **Resignation after schema retrieval** - after guessing the schema, the model eventually recovers and uses the schema tool. However, after seeing the schema (including that of the relevant table), the model decides that a necessary table is missing, and thus the task cannot be accomplished, for example:

```

<tool_call>
{
    "name": "sqlite_get_schema",
    "input": {
        "database_path": "/path/to/database.db"
    }
}
</tool_call>

=====
Tool result: [Table schemas printed]
=====

<tool_call>
{
    "name": "write_file",
    "input": {
        "path": ".../categoryRegionalRevenue.json",
        "content": "{\"totalCategoryRegionalRevenue\":null}"
    }
}
</tool_call>

=====
Tool result: File written successfully
=====


```

The total revenue generated from Beauty category products sold to customers in the North region could not be calculated because the required 'order_items' table is missing from the database schema. The result has been saved as 'null' in the JSON file.

Note that this scenario was explicitly designed with distractor tables, and the main challenge as designed is for the models to understand the problem and be able to determine which tables are appropriate as data sources. This failure pattern shows that Granite 4 Small can fail the core challenge. This happened rarely, in 4 of the 11 failed samples.

- **Schema or semantic confusion** - here the model ends up being distracted by the distractor tables, causing it to fixate on querying using a “BASE_PRICE * ORDER_AMT” approach, when the “ORDER_AMT” column (in the “ORDERS” table) is already the revenue column needed, and “BASE_PRICE” (in one of the distractor tables, “PRODUCTS”) is unnecessary in this task.

4.2 DeepSeek V3.1 - Success and Failure Patterns

An improved version of the original DeepSeek V3, DeepSeek calls this their ”first step toward the agent era.” [6] [7] As seen in Table 1, V3.1 improves significantly from V3 - a clear sign that their agentic-gearied post-training improved agentic fitness without an architecture change.

4.2.1 Summary of Observed Behavioral Patterns

DeepSeek V3.1 exhibits the strongest and most consistently reliable agentic behavior among the three models analyzed, with several clear success patterns and a smaller set of recurring failure modes:

- **Highly reliable execution in structured tasks.** DeepSeek V3.1 succeeds in all trials of Q201–Q202, Q302, Q401, Q403 and Q501, consistently following the optimal multi-step strategy with correct tool sequencing and no hesitation.
- **Systematic verification and value checking.** A defining characteristic of DeepSeek V3.1 is its tendency to proactively verify assumptions before proceeding, such as validating region codes (Q501), investigating suspicious “0” or “null” results, and re-checking schema or intermediate outputs when results seem implausible.
- **Robust error diagnosis and recovery.** When Python code fails, when schemas do not match expectations, or when early assumptions are wrong, the model typically engages in iterative debugging and correction. This reliable recovery ability is the primary driver of its overall dominance on the benchmark.

- **Flexible but grounded tool use.** In text-retrieval (Q301–Q302) and CSV-analysis (Q401–Q402) scenarios, DeepSeek V3.1 correctly switches between point-read, range-read with line numbers, full CSV reads, and Python code depending on context. The model rarely commits to a suboptimal approach once corrective feedback appears.
- **Occasional over-inclusion of data and context.** Some success paths include unnecessarily in-lining entire CSV files into Python strings, increasing context load without affecting correctness. Although not a failure, it reveals inefficiency that can be improved.
- **Vulnerability to semantic confusion from distractors.** In Q503, all of the failures (10/30 trials) stem from fixating on irrelevant tables (e.g., using `BASE PRICE * ORDER AMT` logic) instead of the needed `ORDER AMT` aggregation. This mirrors the same failure pattern seen in Granite 4 Small.
- **Tool-use fragility under extreme load.** Rare failures are linked to long-horizon error-recovery loops, context pollution, or large inline data blocks in Python code. These breakdowns reflect brittle behavior under high cognitive load rather than misunderstanding of the task.
- **Recovery skill as the key differentiator.** DeepSeek V3.1’s superiority does not come from avoiding errors, but from consistently recognizing, explaining, and correcting them. Its iterative, feedback-driven approach contrasts sharply with Granite’s failure to recover.

4.2.2 Detailed Per-Scenario Observations

This subsection contains the full per-scenario observations for DeepSeek V3.1. While the preceding summary highlights the dominant behavioral patterns, the detailed traces below show the specific tool calls, verification steps, error-diagnosis behaviors, and recovery strategies that give rise to those patterns.

Q201 observations

DeepSeek V3.1 succeeded in this task in all samples analyzed (30/30, 100%).

There are two different success patterns observed:

- **Proactive directory validation** - the model explicitly checks if the specified directory exists using its ‘list_directory’ tool, receives “Directory not found” error message from verification attempt, creates the directory, and then the two files.
- **Assuming directory exists + recovery** - here, the model assumes the directory already exists, and proceeds by using the ‘write_file’ tool to create the first file. After it receives an error message indicating that the directory doesn’t exist, it immediately recovers and creates the directory first.

It is worth noting that no samples from DeepSeek V3.1 showed the Granite 4 Small success pattern of simply immediately creating the directory (assuming or understanding that it also doesn’t exist).

A notable quirk found in some of the samples is the use of the ‘get_cwd’ tool at the beginning of the task. Although ultimately unnecessary as the task gives absolute paths, this shows some sort of defensive, context-aware mindset that may be useful in various scenarios.

Q202 observations

DeepSeek V3.1 succeeded in this task in all samples analyzed (30/30, 100%).

Only one success pattern is observed:

- **Proactive directory creation** - unlike in Q201, here the model explicitly creates all of the directories involved, assuming they are all non-existent, without any initial validation step. This is either the effect of the ASCII tree representation, or simply having multiple directories involved.

The quirk observed in Q201 also applies here, and is in fact far more prevalent. 27 of the 30 samples analyzed began with ‘get_cwd’. The defensive “get your bearings” behavior may be triggered more often here due to the scope of the filesystem creation task here compared to Q201.

Q301 observations

DeepSeek V3.1 succeeded in 27/30 samples analyzed in this scenario.

Only one success pattern is observed:

- **Clean sequential execution** - the model executes the file extraction task flawlessly by understanding task semantics immediately, invoking tools sequentially with correct parameters (read first line, read second line, write JSON), and producing exact output without deviations or recovery attempts.

All 3 failed samples follow the same failure pattern:

- **Unnecessary action - punctuation stripping** - Almost exactly like the success pattern behavior, except that when writing the JSON file, the model decides to not include the punctuation (always a period) at the end of the lines:

```
<tool_call>
{
    "name": "read_file",
    "input": {
        "path": ".../meadow/starlight.txt",
        "start_line": 76,
        "end_line": 76
    }
}
</tool_call>

=====
Tool result: '''Neque ea labore reprehenderit cillum amet aute
irure veniam accusantium sit enim accusantium dolorem.'''
=====
```

Now I'll create the JSON file with the retrieved lines using the specified keys:

```
<tool_call>
{
    "name": "write_file",
    "input": {
        "path": ".../lines.json",
        "content": "{\n    \"line_a\": \"Deserunt quisquam minim totam\nofficia enim veritatis eiusmod duis\", \n    \"line_b\": \"Neque\nea labore reprehenderit cillum amet aute irure veniam\naccusantium sit enim accusantium dolorem\"\n}"
    }
}
</tool_call>
```

Note the missing periods at the end of the 'line_a' and 'line_b' values, despite these being included in the tool results.

Q302 observations

DeepSeek V3.1 succeeded in 30/30 samples analyzed in this scenario, making more successful here than in Q301 which was designed to be easier.

Two observed success patterns:

- **Sequential point-read strategy** - as discussed earlier in Granite 4 Small Q302 observations, this optimal solution uses the 'read_file' tool multiple times to read each of the seven lines requested, one at a time, using appropriate 'start_line' and 'end_line' parameters each time.
- **Range-read with manual line extraction** - this was a failure pattern in Granite 4 Small. DeepSeek V3.1 succeeds here not because it can manually count better than most LLMs, but because it smartly uses another feature of the 'read_line' tool - showing line numbers with the output using the 'show_line_numbers' optional parameter.

The range-read success pattern appeared in only 4 out of the 30 samples. Although this is more efficient in terms of number of tools used, it unnecessarily wastes too much input tokens. For example, instead of loading the equivalent of 7 lines using the sequential point-read strategy, the range-read with manual line extraction strategy can load over 90 lines - an increase of over 10× more tokens.

Q401 observations

DeepSeek V3.1 succeeded in 30/30 samples analyzed in this scenario.

There are two success patterns observed:

- **Full CSV read + Python with in-lined CSV data** - the model starts by reading the CSV file in full, then creates Python code that in-lines the full CSV content.

- **Full CSV read + Python file I/O** - as above, but this time more correctly uses Python by using ‘open()’ to load the CSV data, instead of in-lining the full CSV content as a string.

An additional quirk seen is that DeepSeek V3.1 sometimes starts by sampling the CSV file (previewing only the first few lines to understand the structure and content) instead of a full read. However, as interesting as this may be, DeepSeek V3.1 never follows up on this behavior successfully. All samples showing this immediately proceed to a full read, then Python code that either in-lines the full data or uses ‘open()’. An ideal success pattern would have started with the sampling of CSV data, which would then proceed with Python code that uses ‘open()’ to load the file. This behavior could result in a successful trial while minimizing context usage, especially for cases where the source file is hundreds of lines, not just 75.

Q402 observations

DeepSeek V3.1 succeeded in 20/30 samples analyzed in this scenario.

There are two success patterns observed:

- **Clean execution** - the model executes the multi-source CSV analysis flawlessly from start to finish without encountering any errors or requiring recovery attempts. Follows optimal procedural strategy: sequential file reading → consolidated Python execution → final verification.
- **Error recovery and verification** - similar to the above, here the model sometimes encounters errors with its Python code (e.g., numpy serialization issues) or recognizes that a 0.0 output in one of the questions asked might be implausible. In these cases, it adapts and validates its current approach, eventually arriving at the correct final answers.

Failure patterns observed:

- **Guessing instead of sampling values** - the model gets some wrong answers despite having essentially correct strategy because it guessed column values for one or more of the filters necessary. This behavior is similar to the Q502 failure pattern observed earlier in Granite 4 Small
- **Semantic misunderstanding of filter value** - related to the above in that this could also be mitigated by simply sampling values from the CSV, here the model misunderstands the instruction “*inactive status orders*” as *STATUS != ‘active’*, rather than filtering for a specific value like “*inactive*”.
- **Semantic vs literal matching** - the model applies reasonable semantic logic to filter categorical data (interpreting ”East” as Northeast/Southeast, ”South” as all South* variants, ”North” as North/Northeast/Northwest), failing to recognize that the task requires literal exact-match filtering, as these regions are mutually exclusive in the sales CSV data.

All of the failure patterns observed stem from not sampling enough of the CSV data, or simply not sampling the necessary filter values. This could be achieved with either Python code (getting all distinct values for a column, like `status` or `region`), or by simply reading the CSV file in full (very context-heavy approach).

In all of the failed samples, the model never reads any of the CSV files in full, and only previews a few lines from each before proceeding with Python. We noted this as a potential ideal behavior in the preceding Q401 observations section. Now we see the potential drawback of that approach.

In an agentic deployment where metadata can be given in advance, giving the agent the correct metadata (fixed values for `status`, `regions`, etc.) can help mitigate these failure patterns.

Q403 observations

DeepSeek V3.1 succeeded in 30/30 samples analyzed in this scenario.

There are two success patterns observed:

- **Clean execution** - the model executes the multi- CSV analysis flawlessly, starting with sampling each of the CSV files, identifying the relevant CSV file among the group, and then writing Python code for the analysis. Follows optimal procedural strategy: sequential file reading → consolidated Python execution → final verification.
- **Error recovery and verification** - similar to the above, here the model encounters errors with its Python code, but recovers through successful error diagnosis and debugging.

Q501 observations

DeepSeek V3.1 succeeded in 30/30 samples analyzed in this scenario.

There is only one success pattern observed:

- **Clean execution** - the model executes the textbook database query pattern: inspects schema comprehensively ('sqlite.list_tables', 'sqlite_get_schema'), almost always proactively verifies region codes exist in the database before filtering, executes a single correct JOIN-based query, and writes properly formatted JSON output.

In this agentic scenario, DeepSeek V3.1 shows behavior that starkly contrasts with the previous Q402 behavior. Whereas in Q402 a common pitfall was due to the model not validating values such as which status or region codes actually exist in the CSV, here the model is heavily biased towards verifying which codes actually exist in the database (using a 'SELECT DISTINCT LOC_CD FROM enterprise_customer' query) - occurring in 27 out of the 30 successful samples. In the remaining 3 samples, it simply assumed the region code was exactly as written in the task instructions (which was the case).

Q502 observations

DeepSeek V3.1 succeeded in 14/30 samples analyzed in this scenario, it's lowest score in a scenario yet.

There is only one success pattern observed:

- **Clean execution** - the model executes the textbook database query pattern: inspects schema comprehensively ('sqlite.list_tables', 'sqlite_get_schema'), proactively verifies what necessary values exist in the database before filtering, and writes properly formatted JSON output. When encountering queries about entities that don't exist (for example, figures for a company that doesn't have relevant order records), the model verifies the entity existence and returns 0 for related metrics as appropriate.

There are two failure patterns observed:

- **Selective/incomplete schema verification** - the model performs schema exploration and verification for some categorical columns but not others, leading to queries that fail due to missing verification on certain fields. For example, the model may successfully verify status codes (STAT_CD) via DISTINCT query after receiving 0 results, but fails to apply the same diagnostic approach to location codes (LOC_CD), even when the first query returns 0 for the same reason. This verification step is important because, by design, some PICARD-randomized samples can have 0 as a valid answer, and only explicit verification will catch this.
- **Wrong adaptation to missing values** - the model searches for a requested company, determines it does not exist (correctly identifies via LIKE query or company list), then autonomously decides to substitute a similar company name without explicit instruction to do so. Model proceeds with the substitute company's data instead of returning 0. The decision is often made transparently (model acknowledges the substitution in reasoning) but violates the requirement to query for the exact company specified. Alternatively, it interprets the missing entity as something other than "return 0" - for example, treating a missing company filter as "all companies" and omitting the company condition entirely.

The failure mode of wrong adaptation is interesting. Modern LLMs are specifically tuned to be helpful. In regular conversation, this can take the form of completions turning into hallucinations. In agentic scenarios, we see that this "helpfulness" can result in computed data to be incredibly misleading, which would be a disaster in enterprise scenarios where agents at scale operate like this autonomously.

This is something the researchers have explicitly tested in the original KAMI v0.1 paper. Within the KAMI v0.1 test suite, we made an easier version of Q502, called Q602, where we explicitly added this hint to the question template: "*Begin by examining the schema to find relevant columns, and then do your analysis. Note that if the requested company data is not present in the database, then assume the answer is 0 for the relevant question.*"

Scenario	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Avg Score	Avg %
Q502	14	16	20	15	18	12	19	13	15.875	52.92%
Q602	27	25	28	28	25	28	23	26	26.25	87.50%

Table 2: Comparison of DeepSeek V3.1 agentic performance in KAMI v0.1 Q502 and Q602 scenarios

As shown in Table 2, the effect of this intervention is significant. Confusion over missing 'company' data was almost completely removed, and schema guessing was completely removed. The remaining failures in 602 mostly stem from missing 'status' data, which we left out of the hint. Understanding LLM inclinations and mitigating them through specific instructions is an effective approach to increasing enterprise reliability.

Q503 observations

DeepSeek V3.1 succeeded in 20/30 samples analyzed in this scenario, it's lowest score in a scenario yet.

There is only one success pattern observed:

- **Clean execution** - the model executes the optimal strategy without deviation. Performs schema inspection, validates table structure, constructs correct aggregation query using `SUM(ORDER_AMT)`, and returns results cleanly in first attempt.
- **Suspicious result validation** - the model executes the optimal strategy without deviation as in the first pattern. Here, it encounters a ‘0’ or ‘null’ value in its aggregation query, and it proceeds to investigate whether this is legitimate or a mistake. Returns 0 when appropriate.

There is only one failure pattern observed:

- **Schema or semantic confusion** - similar to one of the failure patterns observed in Granite 4 Small. The model ends up being distracted by the distractor tables, causing it to fixate on querying using a “`BASE PRICE * ORDER AMT`” approach, when the “`ORDER AMT`” column is all that is needed for the task.

It is interesting that DeepSeek V3.1, a much larger and generally more capable model than Granite 4 Small as shown by the difference in their KAMI v0.1 scores, also falls to the same semantic confusion as Granite 4 Small in 10 out of 30 samples. This could be a behavior created by a “Chekhov’s gun” effect - as Anthropic researchers put it, this is where “the model may have been naturally inclined to make use of all the information that it was provided.” [14] This highlights the need for careful context engineering - not just simply the amount of tokens in context, but also the quality of the content and the avoidance of irrelevant and potentially confusing data. [25] [19]. This remains a core challenge in enterprise agentic deployments, since by definition these deployments have a great degree of autonomy due to being agentic. But as we can see from both DeepSeekV3.1 (a giant model) and Granite 4 Small (a small model), the effect of context pollution with distractors can be significant.

4.3 Llama 4 Maverick

Llama 4 Maverick is a mixture-of-experts model with 400B total parameters, 128 experts and 17B active parameters, released by Meta in April 2025 [16]. It is the largest of the released Llama 4 models, and shows good performance in KAMI v0.1 (74.6%), scoring between Granite 4 Small (58.5%) and DeepSeek V3.1 (92.2%).

4.3.1 Summary of Observed Behavioral Patterns.

Llama 4 Maverick demonstrates strong raw capability on many individual actions, but its agentic performance is highly inconsistent and significantly less reliable than either DeepSeek V3.1. Its behavior exhibits a mixture of high-ceiling successes and unusually fragile failure modes:

- **High capability but unstable execution.** Maverick often begins tasks with correct reasoning and initially-valid tool selections, but its performance frequently degrades mid-execution. Many failures are not due to misunderstanding the task, but to breakdowns in execution stability, including malformed tool calls, loss of structure in JSON output, or forgetting earlier decisions.
- **Strong first-pass reasoning in simple contexts.** In straightforward filesystem or text-retrieval tasks (Q201–Q302), Maverick sometimes produces high-quality initial plans and correct tool sequences. When success occurs, it often looks “clean” and confident.
- **Severe vulnerability to cascading errors.** Once an initial mistake occurs - such as a malformed tool call, a misread schema, or an unexpected error message - the model frequently enters error loops, produces contradictory explanations, or repeatedly retries invalid operations. Recovery behavior can be much weaker than both DeepSeek and Granite.
- **Hallucinated structure and schema misinterpretation.** In CSV and SQL tasks (Q401–Q503), Maverick repeatedly hallucinates column names, invents schema fields, or conflates unrelated tables. This leads to JOIN logic that has no grounding in the actual schema and SQL statements that reference nonexistent fields.
- **Loss of coherence during debugging.** When attempting to correct errors, Maverick frequently loses track of prior constraints, producing self-contradictory explanations or switching strategies mid-trial. This behavior results in sudden drops in quality even after initially promising starts.
- **Overconfidence and over-helpfulness.** Maverick regularly asserts incorrect assumptions with high confidence, especially in the presence of uncertainty or missing information. It often fills in missing entities implicitly rather than verifying them via schema inspection or tool usage.
- **Context fragility and distraction.** Maverick is highly susceptible to context pollution in tasks with distractor tables (Q503). In many cases it fixates on irrelevant tables, ignores the correct source-of-truth, or reverts to earlier incorrect reasoning despite seeing correct schema information.

- **Occasional flashes of strong performance.** In the rare cases where Maverick executes without encountering errors, its reasoning and SQL generation can be surprisingly strong. However, these successes rely on exceptionally “clean” paths with minimal feedback or ambiguity.

4.3.2 Detailed Per-Scenario Observations

This subsection contains the full per-scenario observations for Llama 4 Maverick. While the preceding summary highlights the dominant behavioral patterns, the detailed traces below show the specific tool calls, breakdown points, error cascades, and recovery failures that give rise to those patterns.

Q201 observations

Llama 4 Maverick succeeded in this task in all samples analyzed (30/30, 100%).

There are two different success patterns observed:

- **Proactive directory validation** - similar to the pattern observed earlier in DeepSeek V3.1 Q201. The model starts by checking if the directory exists, finds it doesn’t, creates it, then creates the two other files inside it. This only happened in 1 out of the 30 samples analyzed.
- **Multiple tools constraint just-in-time learning and adaptation** - Maverick essentially does the same steps as above, except this happens after it tries to use multiple tools in parallel (all at once, in one round of inference) which the agentic server does not allow. This results in an error message that says: *“Tool Call Error: Multiple tool calls found. Please only use one tool at a time.”*. Maverick sees this and either immediately or eventually (0-2 more rounds triggering the same error) adapts to the constraint. Once it has adapted, it proceeds to the ideal path as above.

Hitting the multiple tool calls constraint is worth a short discussion.

Both Granite 4 Small and DeepSeek V3.1 never attempted this, despite this constraint being unmentioned in the agentic server (i.e. missing from the system prompt). This omission was accidental - an earlier version of our agentic server intended for KAMI v0.1 testing included the multiple tool calls constraint in the tool instructions section of the system prompt. However, it was accidentally removed in a patch, and went unnoticed until after the first batch of KAMI v0.1 tests were complete.

Llama 4 Maverick seems to heavily prefer (almost defaults to) parallel tool calls. This does not impair the model’s performance (beyond costing a couple of rounds of inference on average) as it quickly learns. This adaptation behavior is a significant observation. It reveals an important characteristic that is necessary for models in enterprise agentic deployments: the ability to *adapt to its environment and new constraints* that may be contrary to its post-training using various reinforcement learning techniques.

Q202 observations

Llama 4 Maverick succeeded in 29/30 samples analyzed.

There is only one success pattern observed:

- **Multiple tools constraint just-in-time learning and adaptation** - as in Q201, Llama 4 Maverick can take 1 or more rounds to adapt to the tool constraint (immediate or gradual adaptation), and then proceeds to the optimal solution using sequential ‘create_directory’ and ‘write_file’ tool calls, including directory validation through the ‘list_directory’ tool.

There single failure pattern:

- **Wrong tool choice + environment confusion leading to exhausting max rounds** - the model uses a generic Python tool instead of the more task-specific tools available. This is not bad by itself, since a Python approach can effectively one-shot the problem, if done right. Here, however, the Python attempts lead to eventual confusion - having no debugging or even just print statements, the model was unable to debug why its Python attempt did not successfully create the directories and files as intended. It only knew that the base directory was not created despite the code executing successfully (i.e., no syntax errors). Its attempts at debugging eventually led it to the wrong directory due to its use of the ‘get_cwd’ tool (which is the agentic server’s directory). Seeing the output of ‘get_cwd’, the model then fixated on this directory as the place to create the directory hierarchy of the task, forgetting the absolute path from the initial instructions. Eventually, even before being able to complete the task within the wrong directory, the maximum rounds allowed (20) are exhausted, stopping the trial.

What is interesting here is the effect of context pollution. When the model saw the ‘get_cwd’ output, it immediately acted as if that was the correct path, despite the initial instructions very clearly specifying the absolute path for the trial.

Q301 observations

Llama 4 Maverick succeeded in 29/30 samples analyzed.

There are three success patterns observed:

- **Multiple tools constraint just-in-time learning and adaptation** - this is a consistent characteristic of Llama 4 Maverick. Since every trial is independent and the model starts “fresh”, this almost always happens in the vast majority of trials in scenarios. Llama 4 Maverick does this in this scenario before proceeding to one of the other success patterns below.
- **Sequential point-read strategy** - similar to DeepSeek V3.1 behavior observed in Q301, the model executes the optimal sequence: read first line, read second line, and write JSON.
- **Range-read with manual line extraction** - similar to DeepSeek V3.1 behavior observed in Q302, the model does a range-read with explicit ‘show_line_numbers’ using the ‘read_file’ tool. This strategy only occurred rarely, observed in only 3 of the 29 successful samples.

There is one failure pattern observed:

- **Range-read without explicit line numbers / off-by-one error** - this is a behavior pattern that was also observed with Granite 4 Small. It worked for Granite 4 Small in the single instance where it was observed. Here, it did not work for Llama 4 Maverick. After receiving the chunk of unnumbered text lines, Maverick had an off-by-one error when it manually extracted the first line. It picked the second line in the chunk, instead of the first, even though it correctly specified the range-read.

Q302 observations

Llama 4 Maverick succeeded in 24/30 samples analyzed.

There are three success patterns observed:

- **Multiple tools constraint just-in-time learning and adaptation** - as noted earlier, a consistent characteristic in Llama 4 Maverick behavior, before moving on to a more specific success pattern
- **Sequential point-read strategy** - similar to the Q301 pattern, just with more lines to retrieve and more tool calls.
- **Range-read with manual line extraction** - similar to the Q301 pattern. This seems to occur here more frequently, observed in 6 of the samples analyzed, perhaps due to the task’s larger number of lines required to extract.

There are four failure patterns observed:

- **“Eye-ball approach”** - similar to Granite 4 Small, here Maverick reads the entire file (without line numbers enabled) and then proceeds to manually extract the seven lines needed using normal inference instead of tools (i.e.,“eye-balling” the text). As noted in the Granite 4 Small section, the amount of lines retrieved is always guaranteed to be way beyond what LLMs can accurately count through “eye-balling”, so this strategy always fails.
- **Generation loop and coherence loss** - similar to an earlier observed failure pattern in Granite 4 Small Q403. There, it was caused by in-lining large CSV data in Python code. Here it is caused by in-lining large text data (100 lines of randomized lorem ipsum text) in Python code.
- **Hallucination / placeholder completion** - in one observed instance, the model explicitly answers the question with what is clearly placeholder text:

```
I'll first extract the required lines and create a JSON object.
Then, I'll use the 'write_file' tool to write this JSON object
to the file .../lines.json.
```

The required lines are:

- Line 5: "Line 5 content"
- Line 17: "Line 17 content"
- Line 41: "Line 41 content"
- Line 48: "Line 48 content"
- Line 68: "Line 68 content"
- Line 83: "Line 83 content"
- Line 98: "Line 98 content"

The JSON object will be:

```
{
    "line_a": "Line 5 content",
    "line_b": "Line 17 content",
```

```

    "line_c": "Line 41 content",
    "line_d": "Line 48 content",
    "line_e": "Line 68 content",
    "line_f": "Line 83 content",
    "line_g": "Line 98 content"
}

Now, I'll use the 'write_file' tool.

```

```

<tool_start>
{
    "name": "write_file",
    "input": {
        "path": ".../lines.json",
        "content": "{\"line_a\": \"Line 5 content\", \"line_b\": \"Line 17 content\", \"line_c\": \"Line 41 content\",
        \"line_d\": \"Line 48 content\", \"line_e\": \"Line 68 content\", \"line_f\": \"Line 83 content\", \"line_g\": \"Line 98 content\"}"
    }
}
</tool_end>

```

- **Code generation instead of tool use** - in one observed instance, the model generates Python code, likely intending to use its Python tool, but never actually calls for the tool:

```

Now that I have the file content, I'll extract the required
lines and write them to a JSON file.
```python
import json

File content
content = """Quasi mollit...
```

```

Note how the completion is just literal Python code, without the actual tool call tags and syntax.

Q401 observations

Llama 4 Maverick succeeded in 22/30 samples analyzed.

There are two success patterns observed:

- **Clean execution** - the model proceeds with the ideal strategy without encountering any errors (no multiple tool cools, no Python code errors, no schema guessing). This only happened rarely, observed in 1 of the 22 successful samples analyzed.
- **Error recovery and verification** - similar to the pattern observed in DeepSeek V3.1 for Q402. This accounts for 21 of the 22 successful samples analyzed. Model assumes wrong column name (e.g. 'age' instead of 'AGE_YRS'), uses defensive coding (column existence checks or try/except), which prevents exceptions but produces silent failures (0.0 averages or 0 customers). Model then proactively reads back its output, recognizes the incorrect result (average age cannot be 0.0), inspects CSV headers, and corrects the code.

There are five failure patterns observed:

- **Wrong adaptation to missing values** - similar to the pattern observed in DeepSeek V3.1 Q502, here Maverick assumes different values to substitute for the 'Region'
- **Failed Python code** - in 2 out of 8 failed samples, the model encounters errors in its Python code, but unlike the general success case, here it fails to correctly debug the Python code.
- **Instruction / semantic confusion** - model initially gets correct float result (43.36) but task says “do not round”. Model misinterprets this as needing to preserve more precision, uses `format ('.16f')` which converts float to string, creating type error.

- **Hallucination / placeholder completion** - similar to the pattern observed in Maverick Q302, in one instance the model explicitly answers the question with hypothetical values, after being frustrated by failing to debug Python code:

The JSON file is still not visible in the directory listing.
Let's try to list the directory contents again after refreshing the directory.

If that doesn't work, we can try to create the JSON file manually using the 'write_file' tool.

First, let's try to execute the Python code again and capture any potential output or error messages by checking the directory and file existence before and after execution.

Since the previous execution didn't provide any output, let's directly try to create the JSON file manually.

To do that, we need the total customers and average age. Let's assume we have that information.

If we had the CSV data, we could calculate it. Since we don't have it, let's proceed with a hypothetical example.

Let's say 'total_customers' is 100 and 'average_age' is 35.5.

```
'''python
import json
```

```
data = {
    'total_customers': 100,
    'average_age': 35.5
}

json_content = json.dumps(data, indent=4)
'''
```

Now, let's write this JSON content to the file.

```
<tool_start>
{
    "name": "write_file",
    "input": {
        "path": ".../cust.json",
        "content": "{\n            \"total_customers\": 100,\n            \"average_age\": 35.5\n        }"
    }
}</tool_end>
```

Above is just one round of inference, a single completion. It knows it doesn't have real data yet due to Python errors that it hasn't been able to debug yet, but decides to write the hypotheticals as its final answer.

- **Generation loop and coherence loss** - similar to Maverick Q302. In the course of trying to debug its Python code, and even confirming that the tool works and that the problem is not a permissions problem in the directory (which it verified by writing a test file), the model eventually devolved into a generation loop with full loss of coherence, starting with a hallucination of tool results after its tool call:

The Python script file was created successfully.

Now, let's execute this script using the 'python_execute_file' tool.

```
<tool_start>
```

```

{
    "name": "python_execute_file",
    "input": {
        "file_path": ".../create_cust_json.py"
    }
</tool_end>

Let's wait for the tool result: No valid tool call found. Try
again with better-formatted JSON object (dictionary). Try again
the following the request is valid JSON object (dictionary)
The tool has completed. Try again the request is valid tool call
found. Try again (dictionary)
The tool has completed. Try again the request is valid tool call
(dictionary)
The tool has completed. Try again (dictionary)
Returns: String - the tool has completed. Try again (dictionary)
    Returns: String - tool has completed. Try again (dictionary)
    Returns: String - tool has completed. Try again (dictionary)
    Returns: String - the tool has completed. Try again
(dictionary)
    Returns: String - the tool (dictionary)
    Returns: String - the tool (dictionary)
    Returns: String (dictionary)
    [repeated pattern continues...]

```

Note that in the above snippet, everything after “<tool_end> Let’s wait for the tool result” is the model hallucinating the tool result (continuing generation instead of stopping), which quickly devolves into loss of coherence and generation loop.

Q402 observations

Llama 4 Maverick succeeded in only 2/30 samples analyzed.

There is only one success pattern observed:

- **Schema guessing with error-recovery approach** - the model guesses schema (assuming some CSV column names), encounters errors (using the Python tool) due to non-existent columns, realizes the mistake, iteratively examines each CSV to get the actual schema and adapts code until eventual success. Also consistently reads the final output file to validate python execution success.

There are five failure patterns observed:

- **Generation loop and coherence loss** - in 14 of the 28 failed samples, the model devolved into a generation loop with loss of coherence as it attempted to debug and adjust its Python code. From looking at the logs, we see no clear reasonable trigger. Unlike in former cases of coherence loss triggered by in-lining huge blocks of repetitive text or CSV data, here the model does no such thing. It correctly uses relevant Python I/O functions in its code, avoiding data in-lining.
- **Semantic confusion: ORDER_ID vs ORDER_AMT** - very similar to a failure pattern we saw earlier in Granite 4 Small Q502 (the database version of this Q402 CSV scenario). The model, when asked to find the total order value for orders above a specific, randomized threshold, and then write out that value as ‘high_value_total’, sometimes ends up using ‘ORDER_ID’ for its filter in Python/pandas, instead of ‘ORDER_AMT’. As we earlier hypothesized when first observed in Granite 4 Small Q502, this could be an artifact of the training data not usually containing very random numbers as monetary amount filters. This happened in 12 of the 28 failed samples, with some of these overlapping with the generation loop and coherence loss pattern.
- **Max rounds exhaustion** - in some failed samples, the model runs out of max rounds (20 maximum inferences) before succeeding at the task, stopping the trial and resulting in failure. The cause for the exhaustion can vary - sometimes it is

due to iterative schema discovery that consumes too many rounds, sometimes it is debugging Python errors (often also related to schema discovery).

Q403 observations

Llama 4 Maverick succeeded in 21/30 samples analyzed.

There are three success patterns observed:

- **Clean execution** - the model uses systematic approach: explores directory, reads multiple CSV schemas to find the needed CSV file, identifies ‘CUSTOMER_REGION’ field in ‘orders.csv’, applies pandas filtering/aggregation, writes JSON result, and verifies that the file was written and contains the expected answer. Execution is direct and efficient with no errors.
- **Error recovery and verification**- similar to the pattern above, this time the model encounters and recovers from errors (e.g., type errors in its Python code execution, or encountering the multiple-tool-calls constraint), showing successful resilience and adaptability behavior, and then proceeding back to the optimal pattern above until the final verification step.
- **Manual step-by-step sum**- a rare success pattern observed in only 1 of the 21 successful samples analyzed. The model reads CSV files but performs manual arithmetic instead of using pandas/Python. Extracts region-filtered rows by inspection, then adds ‘ORDER_VALUE’ entries step-by-step in natural language (“72739 + 65105 = 137844...”). No Python code execution for aggregation, manual row filtering and sum computation:

```
I've obtained the contents of orders.csv. Now, I'll calculate  
the total value of orders for customers from the Central region  
by summing the ORDER_AMOUNT where CUSTOMER_REGION is 'Central'.
```

The relevant orders are:

```
[ORDER_ID, ORDER_AMOUNT]  
- 924,72739  
- 2914,65105  
- 4677,19600  
- 9952,15711  
- 6395,42348  
- 7966,44511  
- 1690,62840  
- 5004,60213  
- 6152,29836  
- 8636,30459  
- 5339,49865  
- 8750,74691
```

Let's sum these values:

```
72739 + 65105 = 137844  
137844 + 19600 = 157444  
157444 + 15711 = 173155  
173155 + 42348 = 215503  
215503 + 44511 = 260014  
260014 + 62840 = 322854  
322854 + 60213 = 383067  
383067 + 29836 = 412903  
412903 + 30459 = 443362  
443362 + 49865 = 493227  
493227 + 74691 = 567918
```

The total value is 567918.0.

```
Now, I'll save this result as a JSON file at .../  
regional_analysis.json with the key 'totalRegionalValue'.
```

```
<tool_start>  
{
```

```

        "name": "write_file",
        "input": {
            "path": ".../regional_analysis.json",
            "content": "{\"totalRegionalValue\": 567918.0}"
        }
    }
</tool_end>
=====

```

```

Tool result: File written successfully
=====
```

The task is complete. I have successfully calculated the total value of all orders placed by customers from the Central region and saved the result as a JSON file at the specified location.

The total value of orders for customers from the Central region is 567918.0, and it is saved in .../regional_analysis.json with the key 'totalRegionalValue'.

There are three failure patterns observed:

- **Generation loop and coherence loss** - attempts to in-line huge CSV file in Python code, instead of using Python I/O functions, eventually resulting in coherence loss and infinite generation.
- **Semantic confusion: ‘REGION’** - the model makes plausible but incorrect assumption about region naming conventions. Interprets “South” to include “Southeast” and “Southwest” (or similar regional groupings like “Central” → “Midwest”, “North” → “Northeast”). Filters data by expanded region set, producing wrong totals.
- **Instruction-following / formatting: decimal point** - the model calculates correct numerical value using pandas but loses decimal formatting when writing result. Encounters Python ‘with’ statement syntax errors in one-line context, then falls back to write_file with manual JSON string construction - omits “.0” decimal point in manual string. However, in its final message, the model actually remembers the instruction to include the decimal point:

```

The result has been saved successfully to .../
regional_analysis.json.
```

To confirm, let's check the contents of the JSON file.

```

<tool_start>
{
    "name": "read_file",
    "input": {
        "path": ".../regional_analysis.json"
    }
}
</tool_end>
=====
```

```

Tool result: {"totalRegionalValue": 306765}
=====
```

The JSON file has been correctly created with the total regional value.

Task completed: The total value of all orders placed by customers from the Southeast region is 306765.0, and this result has been saved as a JSON file at .../regional_analysis.json with the key 'totalRegionalValue'.

Note how it wrote the value without decimal point (disobeying explicit task instructions), but actually described the answer having that decimal point ('306765.0') in its final message to conclude the trial.

Q501 observations

Llama 4 Maverick succeeded in 28/30 samples analyzed.

There are two success patterns observed:

- **Clean execution** - the model explicitly queries available values in filtered columns (e.g., `SELECT DISTINCT LOC_CD`) before constructing the main query. This verification step confirms valid filter values exist and reduces risk of silent failures from data misunderstandings.
- **Error recovery and verification** - unlike in the optimal strategy, here the model makes assumptions about the schema (lowercase snake_case naming like 'orders', 'region', 'order_total'), encounters errors, then systematically uses schema inspection tools ('sqlite_list_tables', 'sqlite_get_schema') to understand actual structure. Once schema is understood, the model reconstructs queries with correct table names, column names, and JOIN logic on proper foreign keys ('CUST_REF' = 'CUST_ID')

There is only one failure pattern for both of the failed samples:

- **Semantic confusion: ORDER ID vs ORDER AMT** - observed earlier in Maverick Q402 and Granite 4 Small Q502, the model uses order ID ('ORD_ID') instead of order amount ('ORD_AMT') to filter the relevant table given the instruction "*How many orders above [randomized threshold; e.g. '24734']*". We hypothesize about this behavior - see our comments in Granite 5 Small Q502 and Maverick Q402.

Q502 observations

Llama 4 Maverick succeeded in 13/30 samples analyzed.

There is only one success pattern observed:

- **Error recovery and verification** - when encountering constraint violations (e.g., "multiple tool calls not allowed") or schema mismatches (due to schema guessing instead of starting the task by using the schema tools), the model systematically diagnoses the problem, acknowledges the constraint or error, and adapts its execution strategy, eventually proceeding towards the optimal strategy.

There are four failure patterns observed:

- **Generation loop and coherence loss** - another instance where the coherence loss and generation loop was not triggered by in-lining massive data. After completing several queries successfully (5/6 done), the model enters an uncontrolled text generation loop producing repetitive fragments ('-directory', '-string', 'Returns:' repeated 1000+ times) instead of completing the final query and file write. This occurred in only 1 of the 17 failed samples analyzed.
- **Schema or semantic confusion** - another instance of a failure pattern observed in Granite 4 Small Q503 and DeepSeek V3.1 Q503. Note that in those scenarios it seems the schema or semantic confusion is triggered by a distractor table, while here all tables are necessary due to the different statistical values that the agent must retrieve. The model interprets requirements differently than intended, producing logically correct queries that solve the wrong subproblem. For example, interpreting "average order amount" as "average of (quantity × base_price)" rather than direct aggregation of order amounts. This was observed in only 1 out of 17 failed samples analyzed.
- **Guessing instead of sampling values** - the model gets some wrong answers despite having essentially correct strategy because it guessed column values for one or more of the filters necessary. This behavior is similar to a Granite 4 Small Q502 failure pattern, and DeepSeek V3.1 Q402 failure pattern. This is the dominant failure pattern in the samples analyzed.
- **Wrong adaptation to missing values** - the model searches for a requested company, determines it does not exist (correctly identifies via LIKE query or company list), then autonomously decides to substitute a similar company name without explicit instruction to do so. This failure pattern was also observed in DeepSeek V3.1 Q502, and in that previous section we hypothesized the cause of this behavior.

Q503 observations

Llama 4 Maverick succeeded in 19/30 samples analyzed.

There are two success patterns observed:

- **Clean execution** - follows optimal strategy cleanly and encounters no errors as a result. This includes systematic schema discovery first, instead of starting the process by guessing or assuming schema.

- **Error recovery and verification** - the model makes assumptions about the schema causing errors, and sometimes also tries multiple tool calls, triggering the tool calling constraint error. In all cases, the model takes the feedback and adapts, eventually going down the same path as the optimal strategy.

There are two failure patterns observed:

- **Schema or semantic confusion** - another instance of a failure pattern observed in Granite 4 Small Q503, DeepSeek V3.1 Q503, and Maverick Q502. The model gets distracted by the distractor tables, causing it to fixate on querying using a “BASE PRICE * ORDER AMT” approach, when the “ORDER AMT” column is all that is needed for the task. This instance of failure caused by context pollution due to distractors appearing again here is interesting - it has now affected all models analyzed, from a small one (Granite 4 Small = 32B params, dense) to a big one (Maverick = 400B total / 17B active) to an even bigger one (DeepSeek V3.1 = 671B total / 37B active).
- **Null Aggregation Handling Failure** - related to a DeepSeek V3.1 Q503 *success* pattern, “Suspicious result validation”. There, DeepSeek V3.1 does extra validation to investigate whether a ‘0’ or ‘null’ value is legitimate, then returns 0 whenever appropriate. Here, Maverick does almost the same thing, except for properly returning ‘0’, instead writing ‘null’ in the JSON file. The model understands SQL semantics (NULL for empty aggregations) but fails to map that to application-level semantics (0 for “no revenue”).

That Maverick fails to automatically default to 0 despite talking about revenue can be treated as a *soft* failure - the task never explicitly prompted for the data type of the JSON values and took it for granted that most good models will automatically realize ‘0’ is the correct reporting format for no revenue. This is something that is probably more a limitation on the side of KAMI v0.1 itself, and should be improved in v0.2 onwards. Just a slight hint about the expected data types would be good mitigation and ensure a fairer comparison between model scores.

5 Synthesis: Cross-Model Patterns in Agentic Failure and Success

Our granular analysis across three models and ten diverse agentic scenarios reveals that failure modes are not uniformly distributed by model scale or task type. Instead, they cluster around four interrelated behavioral archetypes, each reflecting a gap between the agent’s internal reasoning and the grounded, iterative reality of tool-mediated task execution. These patterns hold across model families and capability tiers - though their frequency and recoverability vary significantly.

Premature Commitment Without Grounding: All models - though to very different degrees - occasionally skip essential grounding steps such as inspecting file schemas before parsing, validating database table or column names before querying, or sampling categorical values before filtering. These lapses result in schema guessing, which is especially disastrous in Q402 (multi-CSV) and Q502 (multi-query SQL), where assumptions about column names or domain values lead to silent failures. Granite 4 Small almost always fails in this way; Llama 4 Maverick sometimes recovers after errors; and DeepSeek V3.1 still commits this error, but far less frequently.

Over-Helpfulness Leading Autonomous Substitution / Wrong Adaptation: When faced with missing or ambiguous entities (e.g., a company name not present in the database), Llama 4 Maverick and DeepSeek V3.1 sometimes autonomously substitute a “similar” entity or relax constraints to produce plausible answers. While well-intentioned, this violates task fidelity - especially in enterprise contexts where “0” is the correct answer for missing data. This behavior appears to stem from alignment tuning that over-optimizes for helpfulness and completion fluency, not precision under uncertainty.

Sensitivity to Context Pollution: Across all models, the presence of distractor files or tables (e.g., Q403 and Q503) triggers semantic overreach: agents attempt to incorporate irrelevant data (e.g., multiplying BASE_PRICE by ORDER_AMT despite ORDER_AMT already representing total revenue). DeepSeek V3.1 is not immune - 10/30 trials failed this way. This “Chekhov’s gun” tendency suggests that even the biggest and more modern LLMs are not yet robust to information overload; they treat all provided context as signal, not noise.

Fragile Tool-Use and Coherence Under Stress: Tool call formatting errors, generation loops, and coherence collapse appear most frequently under two conditions: (a) when models inline large data blocks (CSV or text) directly into code, and (b) when they encounter repeated error-recovery cycles. Granite 4 Small often stalls on malformed JSON; Llama 4 Maverick frequently enters infinite generation loops during Python debugging. These failures are not due to ignorance of the task, but to brittle execution under resource or cognitive load.

Crucially, recovery capability - not initial correctness - best predicts overall success. DeepSeek V3.1’s dominance stems not from never failing, but from consistently recognizing errors (e.g., implausible averages like 0.0 for age), diagnosing root causes (e.g., wrong column names), and iteratively refining tool use.

Because KAMI v0.1 is explicitly designed as an interactive, multi-step agentic benchmark that mirror real-world enterprise automation and conditions - rather than a static Q&A or code-completion task - it exposes failure modes invisible to traditional evaluations. These findings suggest that both future benchmarks and agentic training protocols must prioritize interactive

resilience in realistic enterprise scenarios - the ability to validate assumptions, interpret feedback, and adapt mid-execution - over one-shot accuracy or raw reasoning performance.

5.1 Error Recovery: Interactive Learning and Constraint Adaptation

Agentic reliability is determined not by error absence, but by how effectively errors are converted into corrective action. Table 3 summarizes the models:

| Capability | Granite | Maverick | DeepSeek |
|--|---------|--------------|-----------|
| Detect malformed tool calls | Medium | High | Very High |
| Successful use of error feedback to self-correct | Low | High | Very High |
| Escape perseverative loops | Poor | Medium | High |
| Self-debug Python/SQL | Poor | Inconsistent | Frequent |

Table 3: Comparative error-recovery traits across models.

Notable observations:

- Maverick is the only model to identify an unstated constraint (single tool per round) and learn it mid-trial.
- DeepSeek’s superior recovery indicates strong training on execution traces.
- Granite shows error recovery capabilities, but among the three has the lowest successful recovery rate.

5.2 Emergent Principles

Based on the qualitative analysis, we summarize four emergent principles that we believe are critical for reliable enterprise agentic AI deployments.

1. **Neither size nor general capability equal agentic reliability.** Maverick outperforms Granite overall, yet still performs badly in Q402. DeepSeek V3 - the same base model as V3.1 - barely scores better than Granite despite being 20× larger than IBM’s 32B model. Agentic reliability must be explicitly modeled and simulated, even as early as use case exploration or the beginning of proof-of-concept activities, using techniques like the PICARD framework.
2. **Error feedback is the new frontier for autonomy.** Post-training models must internalize tool semantics *and* system constraints. Tool messages, especially error feedback, must be treated as a first-class design challenge with the explicit goal of increasing the LLM’s odds of a successful recovery. Our analysis shows that recovery capability, not initial correctness, is the dominant predictor of overall success.
3. **Context quality matters more than context quantity.** Distractors cause systemic failures even in the largest model tested. This switches the challenge from just “how can we retrieve available related data?” to “how do we only retrieve the highest quality and most relevant data for the task?” As Q403 and Q503 demonstrate, more context frequently reduces reliability.
4. **Literal, source-of-truth alignment.** Enterprise agents must prioritize actual data over their priors. Behaviors like guessing schemas or hypothesizing values must be inhibited - either through RL finetuning, or as a function of context curation through system prompts, instruction enrichment, and tool message design. Grounded verification is essential for correctness under uncertainty.

These principles align with the design philosophy of both the KAMI v0.1 benchmark and its underlying PICARD framework. Rather than optimizing for clean, one-shot evaluation, KAMI v0.1 deliberately incorporates distractors, environmental constraints, and opportunities for multi-step recovery - precisely to surface the kinds of failure modes that undermine reliability in real-world enterprise deployments. The goal is not theoretical completeness, but actionable insight into what makes agentic systems robust under uncertainty.

5.3 Practical Recommendations for Enterprise Deployment

The emergent principles articulated above suggest several concrete practices for organizations deploying agentic AI systems in production environments.

- 1. Tool and Prompt Design.** The observation that models frequently bypass grounding steps - guessing schemas rather than inspecting them - indicates that tool descriptions and system prompts should explicitly mandate verification before action, assuming use cases where not all schemas can be known in advance and loaded into the system prompt. Similarly, error messages returned by tools should be designed not merely to indicate failure, but to suggest corrective paths, given that recovery capability is the dominant predictor of overall success.
- 2. Context Engineering.** The “Chekhov’s gun” effect observed across all models suggests that context should be curated aggressively. Providing agents with access to all available data sources may paradoxically reduce reliability by introducing distractor information that triggers semantic overreach. Where categorical filtering is required (e.g., region codes, status values), providing explicit enumerations of valid values in context can prevent the “guessing instead of sampling” failure mode observed repeatedly in our analysis. If categorical values cannot be known in advance and preloaded into context, the Tool and Prompt Design practice above applies - give the LLM a tool that allows for data inspection and instruct the LLM to use this tool instead of guessing filter values.
- 3. Architectural Safeguards.** For high-stakes operations, agentic systems should implement verification checkpoints requiring explicit confirmation of assumptions before proceeding with dependent actions. Additionally, runtime monitoring that detects generation anomalies - repetitive token patterns, unusual output lengths, or coherence degradation - can allow early termination of failing executions before they produce harmful outputs.
- 4. Model Selection and Evaluation.** Organizations selecting models for agentic deployment should evaluate not only first-pass accuracy but also behavior under error conditions. A model that fails gracefully and recovers reliably may be preferable to one with higher initial accuracy but brittle error handling. Evaluation protocols should include scenarios with realistic distractors to assess context pollution vulnerability - a failure mode largely invisible in conventional benchmark conditions.

6 Threats to Validity

We note several validity limitations:

- **Model sample scope.** Only three models were analyzed for the study; broader model diversity may yield different behaviors.
- **Scenario coverage.** KAMI v0.1 tasks emphasize tool-grounded data correctness and may not generalize to long-horizon planning.
- **Training secrecy.** Proprietary post-training does not allow attribution of observed behaviors to specific methods.
- **Execution environment effects.** The single-tool-per-round constraint influences strategy (especially Maverick).
- **Model Temperature.** Our choice of 0.4 was based on the desire to reduce stochasticity while preserving agentic flexibility, but the effects of alternative temperature settings remain unexplored.

Future work will explore the effects of model temperature on agentic performance more explicitly.

Mitigations planned for KAMI v0.2 include richer scenario coverage and explicit data-type expectations where appropriate.

In parallel, to mitigate the model sample scope limitation, we are working on a reliable AI-augmented approach to agentic execution trace analysis at scale. In this study, we perform the analysis manually. Manual analysis was not our first choice; earlier attempts at AI-augmented analysis - whether from Claude Code with various latest Claude models like Sonnet 4.5 and Haiku 4.5 [2–4], or from our self-hosted open LLMs like Qwen3-Next 80B-A3B Instruct or GLM 4.6 using different orchestration and prompting techniques [18, 32] - proved to be too unreliable. They produce very plausible sounding success and failure analysis patterns, even citing specific samples as instructed, but the findings do not always stand up when carefully scrutinized by the researchers. The challenge here is that the agentic execution traces produced by KAMI v0.1 runs are not only voluminous in terms of total number of files (over 200,000 LLM traces as of the time of writing, across over 40 different models), but also how each individual file can be very large: tens of thousands of overall tokens each for non-reasoning models, and hundreds of thousands of tokens each for reasoning models. A naive approach could never work (there is no context size big enough to hold that many traces at once - over 4GB of raw text), while strictly partitioning traces into a 1:1 ratio with inference (1 trace to 1 inference) means the LLM will only see a very isolated snapshot of performance. We experimented on many different approaches, including designing very specific tools, but have not yet reached the point where overall output is acceptable. We plan to address this topic in future work.

7 Conclusion

Our qualitative analysis of 900 agentic trials across three representative language models reveals that reliability in tool-mediated tasks is not determined solely by scale, but by the presence of specific behavioral strategies - many of which are shaped by post-training design rather than model size alone. Granite 4 Small (32B dense), Llama 4 Maverick (400B MoE), and DeepSeek V3.1 (671B MoE) exhibit markedly different failure profiles that do not follow a simple scaling law. For instance, Llama 4 Maverick achieves only 2/30 success in the multi-source CSV analysis task (Q402) - marginally above Granite 4 Small's 1/30 - despite its significantly larger size. In contrast, DeepSeek V3.1's robustness stems not from its massive MoE architecture, but from reinforcement learning that cultivates systematic habits of grounding, verification, and recovery.

Across the ten KAMI v0.1 scenarios analyzed, we observe four recurring failure archetypes:

- **Premature action without grounding:** Models frequently guess table or column names instead of inspecting schemas, leading to silent failures - especially in multi-file or multi-table settings (e.g., Q402, Q502).
- **Over-helpfulness under uncertainty:** When faced with missing entities (e.g., a company not present in the database), models sometimes substitute plausible alternatives or relax constraints, violating task fidelity, likely due to alignment tuning that over-prioritizes helpfulness.
- **Sensitivity to context pollution:** All models occasionally incorporate distractor data (e.g., multiplying BASE_PRICE by ORDER_AMT in Q503), reflecting a “Chekhov’s gun” bias: the assumption that all provided context is relevant.
- **Fragile execution under cognitive load:** Coherence collapse, generation loops, and tool-call formatting errors arise most often when models inline large data blocks into code or face repeated debugging cycles without structured feedback.

Critically, *recovery behavior - not initial correctness - best predicts overall success*. DeepSeek V3.1’s high performance derives from its consistent ability to interpret error messages, diagnose root causes (e.g., incorrect column names), and iteratively refine its approach. In contrast, Llama 4 Maverick often exhibits strong reasoning but struggles with sustained debugging, while Granite 4 Small lacks the self-monitoring needed to recover from early missteps.

These insights are only visible because the KAMI v0.1 benchmark evaluates agents in *interactive, multi-step, tool-mediated environments* - a design that mirrors real-world enterprise automation far more closely than static, one-shot evaluations. As such, our findings argue for a paradigm shift in both agentic evaluation and training:

- Future benchmarks should measure *agentic resilience* - the ability to validate assumptions, interpret feedback, and adapt mid-execution - rather than passive accuracy or reasoning scores.
- Training protocols should explicitly reward verification behaviors (e.g., schema inspection, output validation) over mere output fluency or pass@ k performance.

The path toward robust, enterprise-ready agentic AI lies not in scaling alone, but in cultivating disciplined habits of grounding, cautious inference, and structured error recovery - capabilities that, as our analysis demonstrates, can be learned, measured, and systematically improved. Future enterprise deployment must treat interactive error handling, constraint discovery, and context curation as first-class design challenges.

Data Availability

The raw experiment data, including all 900 execution traces analyzed in this study, will be made available at <https://docs.kamiwaza.ai/research/datasets>.

AI Usage Disclosure

The researchers used the following generative AI services to assist with the manuscript:

- Qwen Chat: Qwen3 Max
- Claude Desktop: Claude Opus 4.5
- ChatGPT: GPT 5 Instant

About 70% of the content was written directly by human researchers. As each section was written, we sometimes asked generative AI for review and suggestions. We routinely accept only about 50% of the suggestions. The suggestions we don't accept are either very low impact or stylistic changes that we do not prefer, or simply hallucinations or misunderstandings of the text.

About 30% of the content was generated directly through generative AI. The Abstract, Introduction, Methodology, and Conclusion were first generated from generative AI by feeding it the existing manuscript (what are now Sections 3 and 4) and our overall description of how we envision the paper to be, including certain reference material. In addition, all tables were created through generative AI directly using raw source data. For all of these sections and elements generated directly by generative AI, we use one of Claude Desktop, Qwen Chat, and ChatGPT to generate the first draft, and then have the other two do reviews. Then, human researchers do a final review, editing and integration.

In all cases, final editorial control, technical validation, and intellectual responsibility rest solely with the human authors. The authors take full responsibility for the accuracy and integrity of all content in this manuscript.

References

- [1] Anthropic. Introducing the model context protocol. <https://www.anthropic.com/news/model-context-protocol>, 2024. Anthropic News. Accessed: 2025-12-03.
- [2] Anthropic. Claude 3.7 sonnet and claude code. <https://www.anthropic.com/news/clause-3-7-sonnet>, February 2025. Accessed: December 2025.
- [3] Anthropic. Introducing claude haiku 4.5. <https://www.anthropic.com/news/clause-haiku-4-5>, October 2025. Accessed: December 2025.
- [4] Anthropic. Introducing claude sonnet 4.5. <https://www.anthropic.com/news/clause-sonnet-4-5>, September 2025. Accessed: December 2025.
- [5] Artificial Analysis. Artificial analysis: Independent analysis of ai models and api providers. <https://artificialanalysis.ai/>, 2025. Accessed: 2025-10-12.
- [6] DeepSeek. Deepseek-v3.1 release announcement. <https://api-docs.deepseek.com/news/news250821>, 2025. Accessed: 2025-11-28.
- [7] DeepSeek. X.com: Deepseek-v3.1 release announcement. https://x.com/deepseek_ai/status/1958417062008918312, 2025. Accessed: 2025-11-28.
- [8] Maria Eriksson, Erasmo Purificato, Arman Noroozian, Joao Vinagre, Guillaume Chaslot, Emilia Gomez, and David Fernandez-Llorca. Can we trust ai benchmarks? an interdisciplinary review of current issues in ai evaluation, 2025.
- [9] IBM. Ibm granite 4.0: Hyper-efficient, high performance hybrid models for enterprise. <https://www.ibm.com/new/announcements/ibm-granite-4-0-hyper-efficient-high-performance-hybrid-models>, October 2025. Accessed: 2025-11-28.
- [10] Adam Jones and Conor Kelly. Code execution with MCP: Building more efficient AI agents. <https://www.anthropic.com/engineering/code-execution-with-mcp>, 2025. Anthropic Engineering Blog. Accessed: 2025-12-03.
- [11] Shachar Kaufman, Saharon Rosset, Claudia Perlich, and Ori Stitelman. Leakage in data mining: Formulation, detection, and avoidance. *ACM Trans. Knowl. Discov. Data*, 6(4), 2012.
- [12] Jon Keegan. Everyone is judging ai by these tests. but experts say they're close to meaningless. The Markup, 2024.
- [13] Patrick Lewis, Pontus Stenetorp, and Sebastian Riedel. Question and answer test-train overlap in open-domain question answering datasets. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*, pages 1000–1008. Association for Computational Linguistics, 2021.
- [14] Aengus Lynch, Benjamin Wright, Caleb Larson, Kevin K. Troy, Stuart J. Ritchie, Sören Mindermann, Ethan Perez, and Evan Hubinger. Agentic misalignment: How llms could be an insider threat. *Anthropic Research*, 2025. <https://www.anthropic.com/research/agentic-misalignment>.

- [15] Inbal Magar and Roy Schwartz. Data contamination: From memorization to exploitation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 157–165. Association for Computational Linguistics, 2022.
- [16] Meta AI. The llama 4 herd: The beginning of a new era of natively multimodal ai innovation. <https://ai.meta.com/blog/llama-4-multimodal-intelligence/>, April 2025. Accessed: 2025-11-28.
- [17] Arvind Narayanan and Sayash Kapoor. Gpt-4 and professional benchmarks: the wrong answer to the wrong question. <https://www.normaltech.ai/p/gpt-4-and-professional-benchmarks>, 2023. Accessed: 2025-11-28.
- [18] Qwen Team. Qwen3-next: Towards ultimate training & inference efficiency. <https://qwen.ai/blog?id=4074cca80393150c248e508aa62983f9cb7d27cd>, September 2025. Accessed: December 2025.
- [19] Prithvi Rajasekaran, Ethan Dixon, Carly Ryan, and Jeremy Hadfield. Effective context engineering for ai agents. <https://www.anthropic.com/engineering/effective-context-engineering-for-ai-agents>, 2025. Accessed: 2025-11-28.
- [20] Inioluwa Deborah Raji, Emily Denton, Emily M Bender, Alex Hanna, and Amandalynne Paullada. Ai and the everything in the whole wide world benchmark. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*, 2021.
- [21] Richard Ren, Steven Basart, Adam Khoja, Alice Gatti, Long Phan, Xuwang Yin, Mantas Mazeika, Alexander Pan, Gabriel Mukobi, Ryan Hwang Kim, Stephen Fitz, and Dan Hendrycks. Safetywashing: Do ai safety benchmarks actually measure safety progress? In *The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2024.
- [22] Manley Roberts, Himanshu Thakur, Christine Herlihy, Colin White, and Samuel Dooley. Data contamination through the lens of time, 2023.
- [23] JV Roig. Testing what models can do, not what they've seen: Picard: Probing intelligent capabilities via artificial randomized data. Technical report, Independent Research, 2025.
- [24] JV Roig. Towards a standard, enterprise-relevant agentic ai benchmark: Lessons from 5.5 billion tokens' worth of agentic ai evaluations, 2025.
- [25] Philipp Schmid. The new skill in ai is not prompting, it's context engineering. <https://www.philschmid.de/context-engineering>, 2025. Accessed: 2025-11-28.
- [26] Kushal Tirumala, Aram Markosyan, Luke Zettlemoyer, and Armen Aghajanyan. Memorization without overfitting: Analyzing the training dynamics of large language models. In *Advances in Neural Information Processing Systems*, volume 35, pages 38274–38290, 2022.
- [27] Kenton Varda and Sunil Pai. Code mode: The better way to use MCP. <https://blog.cloudflare.com/code-mode/>, 2025. Cloudflare Blog. Accessed: 2025-12-03.
- [28] Cheng Xu, Shuhao Guan, Derek Greene, and M.-Tahar Kechadi. Benchmark data contamination of large language models: A survey, 2024.
- [29] Ruijie Xu, Zengzhi Wang, Run-Ze Fan, and Pengfei Liu. Benchmarking benchmark leakage in large language models, 2024.
- [30] Lifan Yuan, Yangyi Chen, Ganqu Cui, Hongcheng Gao, Fangyuan Zou, Xingyi Cheng, Heng Ji, Zhiyuan Liu, and Maosong Sun. Revisiting out-of-distribution robustness in nlp: Benchmark, analysis, and llms evaluations. *37th Conference on Neural Information Processing Systems (NeurIPS 2023) Track on Datasets and Benchmarks*, 2023.
- [31] Andy K. Zhang, Kevin Klyman, Yifan Mai, Yoav Levine, Yian Zhang, Rishi Bommasani, and Percy Liang. Language model developers should report train-test overlap, 2024.
- [32] Zhipu AI. Glm-4.6. <https://huggingface.co/THUDM/GLM-4.6>, September 2025. Hugging Face Model Card. Accessed: December 2025.

Appendix A Complete Test Definitions

Below are the complete PICARD test definitions for the ten scenarios analyzed. These definitions provide the full instruction templates and sandbox configurations used in KAMI v0.1, and allow readers to examine the exact inputs encountered by the models beyond the high-level descriptions in Section 3.1. Of particular relevance:

- **template** - the instruction text shown to the model, including randomized variables such as filenames, directory paths, and entity names.
- **sandbox_setup** - where specified, the environment artifacts created for the scenario, including CSV files (with filenames, columns, and data), text files (with content and length), and database files (tables, schemas, and data).

```
tests:  
  
#SECTION 200: File System  
  
- question_id: 201  
samples: 30  
template: "Create the following blank files: {{entity1}}.log and {{entity2}}.config in  
the {{artifacts}}/{{qs_id}}/{{entity3}} directory."  
scoring_type: "files_exist"  
files_to_check:  
- "{{artifacts}}/{{qs_id}}/{{entity3}}/{{entity1}}.log"  
- "{{artifacts}}/{{qs_id}}/{{entity3}}/{{entity2}}.config"  
  
- question_id: 202  
samples: 30  
template: |  
Create this directory structure, including all blank files specified, inside the  
folder '{{artifacts}}/{{qs_id}}':  
{{expected_structure}}  
scoring_type: "directory_structure"  
expected_structure:  
- "{{artifacts}}/{{qs_id}}/{{entity1}}/"  
- "{{artifacts}}/{{qs_id}}/{{entity1}}/{{entity2}}/"  
- "{{artifacts}}/{{qs_id}}/{{entity1}}/logs/"  
- "{{artifacts}}/{{qs_id}}/{{entity1}}/logs/{{entity3}}.log"  
- "{{artifacts}}/{{qs_id}}/{{entity4}}/"  
- "{{artifacts}}/{{qs_id}}/{{entity4}}/README.md"  
- "{{artifacts}}/{{qs_id}}/{{entity5}}/"  
- "{{artifacts}}/{{qs_id}}/{{entity5}}/{{entity6}}.{{entity7}}"  
- "{{artifacts}}/{{qs_id}}/{{entity5}}/{{entity6}}.{{entity8}}"  
- "{{artifacts}}/{{qs_id}}/{{entity5}}/{{entity6}}.{{entity9}}"  
  
#####  
#SECTION 300: Finding needles in files  
- question_id: 301  
samples: 30  
template: |  
Retrieve the following lines from this large text file  
{{artifacts}}/{{qs_id}}/{{entity1}}/{{entity2}}.txt:  
- "line_a": Line {{number1:1:40}}  
- "line_b": Line {{number2:51:90}}  
  
Write your response in a json file {{artifacts}}/{{qs_id}}/lines.json, using "line_a"  
and "line_b" as the keys.  
Each value should contain only the text content of that line, with no additional  
formatting.  
scoring_type: "readfile_jsonmatch"  
file_to_read: "{{artifacts}}/{{qs_id}}/lines.json"  
expected_content: |  
{
```

```

    "line_a": "{{file_line:{{number1:1:40}}:TARGET_FILE[notes_file]}}",
    "line_b": "{{file_line:{{number2:51:90}}:TARGET_FILE[notes_file]}}"
}
sandbox_setup:
components:
- type: "create_files"
  name: "notes_file"
  target_file: "{{{artifacts}}}/{{qs_id}}/{{entity1}}/{{entity2}}.txt"
  content:
    type: "lorem_lines"
    count: 100

- question_id: 302
  samples: 30
  template: |
    Retrieve the following lines from this large text file
    {{{artifacts}}}/{{qs_id}}/{{entity1}}/{{entity2}}.txt:
    - "line_a": Line {{number1:1:15}}
    - "line_b": Line {{number2:16:30}}
    - "line_c": Line {{number3:31:45}}
    - "line_d": Line {{number4:46:60}}
    - "line_e": Line {{number5:61:75}}
    - "line_f": Line {{number6:76:90}}
    - "line_g": Line {{number7:91:105}}

    Write your response in a json file {{{artifacts}}}/{{qs_id}}/lines.json, using "line_a",
    "line_b" ... until "line_g" as the keys.
    Each value should contain only the text content of that line, with no additional
    formatting.
  scoring_type: "readfile_jsonmatch"
  file_to_read: "{{{artifacts}}}/{{qs_id}}/lines.json"
  expected_content: |
{
  "line_a": "{{file_line:{{number1:1:15}}:TARGET_FILE[notes_file]}}",
  "line_b": "{{file_line:{{number2:16:30}}:TARGET_FILE[notes_file]}}",
  "line_c": "{{file_line:{{number3:31:45}}:TARGET_FILE[notes_file]}}",
  "line_d": "{{file_line:{{number4:46:60}}:TARGET_FILE[notes_file]}}",
  "line_e": "{{file_line:{{number5:61:75}}:TARGET_FILE[notes_file]}}",
  "line_f": "{{file_line:{{number6:76:90}}:TARGET_FILE[notes_file]}}",
  "line_g": "{{file_line:{{number7:91:105}}:TARGET_FILE[notes_file]}}"
}
sandbox_setup:
components:
- type: "create_files"
  name: "notes_file"
  target_file: "{{{artifacts}}}/{{qs_id}}/{{entity1}}/{{entity2}}.txt"
  content:
    type: "lorem_lines"
    count: 110

#####
#Section 400: CSV Processing
#####
- question_id: 401
  samples: 30
  template: "Create a JSON summary of {{{artifacts}}}/{{qs_id}}/{{entity1}}/crm_export.csv
  showing total customers (integer) and average age (full floating-point precision, do not
  round). Create {{{artifacts}}}/{{qs_id}}/cust.json with the JSON containing
  'total_customers' and 'average_age'."
  scoring_type: "readfile_jsonmatch"
  file_to_read: "{{{artifacts}}}/{{qs_id}}/cust.json"
  expected_content: '{"total_customers": {{csv_count:C_ID:TARGET_FILE[crm_data]}}},"

```

```

"average_age": {{csv_avg:AGE_YRS:TARGET_FILE[crm_data]}}'
sandbox_setup:
components:
- type: "create_csv"
  name: "crm_data"
  target_file: "{{artifacts}}/{{qs_id}}/{{entity1}}/crm_export.csv"
  content:
    headers: ["C_ID", "C_NAME", "AGE_YRS", "LOC_CD", "REG_DT"]
    header_types: ["id", "person_name", "age", "city", "date"]
    rows: 75

- question_id: 402
  samples: 30
  template: "Analyze the business data across multiple CSV files in {{artifacts}}/{{qs_id}}/ and create a comprehensive report at {{artifacts}}/{{qs_id}}/multi_source_analysis.json. Answer these questions: (1) How many contacts do we have in the {{semantic1:category}} industry in contacts.csv? (2) What is the average base price for {{semantic2:category}} category products in products.csv (full floating-point precision, do not round)? (3) How many customers do we have in the {{semantic3:region}} region? (4) What is the total order value for orders above {{number1:15000:35000:currency}} in orders.csv? (Include decimal point, e.g., 718313.0). (5) How many {{semantic4:status}} status orders are there in orders.csv? (6) What is the average quantity across all orders in orders.csv (full floating-point precision, do not round)? Create JSON with keys: 'industry_contact_count', 'category_avg_price', 'region_customer_count', 'high_value_total', 'status_order_count', 'avg_order_quantity'." 
  scoring_type: "readfile_jsonmatch"
  file_to_read: "{{artifacts}}/{{qs_id}}/multi_source_analysis.json"
  expected_content: "{\"industry_contact_count\": {{csv_count_where:COMP_ID:INDUSTRY::={{semantic1:category}}:TARGET_FILE[companies_csv]}}, \"category_avg_price\": {{csv_avg_where:BASE_PRICE:CATEGORY::={{semantic2:category}}:TARGET_FILE[products_csv]}}, \"region_customer_count\": {{csv_count_where:CUSTOMER_ID:REGION::={{semantic3:region}}:TARGET_FILE[customers_csv]}}, \"high_value_total\": {{csv_sum_where : ORDER_AMOUNT : ORDER_AMOUNT : > : {{number1:15000:35000:currency}} : TARGET_FILE[orders_csv]}}, \"status_order_count\": {{csv_count_where:ORDER_ID:STATUS::={{semantic4:status}}:TARGET_FILE[orders_csv]}}, \"avg_order_quantity\": {{csv_avg:QUANTITY:TARGET_FILE[orders_csv]}}}"
  sandbox_setup:
components:
- type: "create_csv"
  name: "companies_csv"
  target_file: "{{artifacts}}/{{qs_id}}/contacts.csv"
  content:
    headers: ["COMP_ID", "COMPANY", "INDUSTRY", "CONTACT_PERSON"]
    header_types: ["id", "company", "category", "person_name"]
    rows: "{{number2:40:50}}"

- type: "create_csv"
  name: "products_csv"
  target_file: "{{artifacts}}/{{qs_id}}/products.csv"
  content:
    headers: ["PROD_ID", "PROD_NAME", "CATEGORY", "BASE_PRICE", "SUPPLIER"]
    header_types: ["id", "product", "category", "price", "company"]
    rows: "{{number3:60:70}}"

- type: "create_csv"
  name: "customers_csv"
  target_file: "{{artifacts}}/{{qs_id}}/customers.csv"
  content:
    headers: ["CUSTOMER_ID", "CUSTOMER_NAME", "REGION", "DEPARTMENT", "SIGNUP_DATE"]
    header_types: ["id", "person_name", "region", "department", "date"]
    rows: "{{number4:40:50}}"

```

```

- type: "create_csv"
  name: "orders_csv"
  target_file: "{{artifacts}}/{{qs_id}}/orders.csv"
  content:
    headers: ["ORDER_ID", "ORDER_AMOUNT", "QUANTITY", "STATUS", "ORDER_DATE"]
    header_types: ["id", "currency", "score", "status", "date"]
    rows: "{{number5:40:50}}"

- question_id: 403
  samples: 30
  template: "Analyze the business data files in {{artifacts}}/{{qs_id}}/ and determine: What is the total value of all orders placed by customers from the {{semantic1:region}} region? (Include decimal point, e.g., 718313.0). Save your answer as a JSON file at {{artifacts}}/{{qs_id}}/regional_analysis.json with the key 'totalRegionalValue'.""
  scoring_type: "readfile_jsonmatch"
  file_to_read: "{{artifacts}}/{{qs_id}}/regional_analysis.json"
  expected_content: "{\"totalRegionalValue\": {{csv_sum_where : ORDER_AMOUNT : CUSTOMER_REGION : == : {{semantic1:region}} : TARGET_FILE[orders_with_regions_csv}}}}"
  sandbox_setup:
    components:
      - type: "create_csv"
        name: "companies_csv"
        target_file: "{{artifacts}}/{{qs_id}}/contacts.csv"
        content:
          headers: ["COMP_ID", "COMPANY", "INDUSTRY", "CONTACT_PERSON"]
          header_types: ["id", "company", "category", "person_name"]
          rows: "{{number2:40:50}}"

      - type: "create_csv"
        name: "products_csv"
        target_file: "{{artifacts}}/{{qs_id}}/products.csv"
        content:
          headers: ["PROD_ID", "PROD_NAME", "CATEGORY", "BASE_PRICE", "SUPPLIER"]
          header_types: ["id", "product", "category", "price", "company"]
          rows: "{{number3:40:50}}"

      - type: "create_csv"
        name: "customers_csv"
        target_file: "{{artifacts}}/{{qs_id}}/customers.csv"
        content:
          headers: ["CUSTOMER_ID", "CUSTOMER_NAME", "REGION", "DEPARTMENT", "SIGNUP_DATE"]
          header_types: ["id", "person_name", "region", "department", "date"]
          rows: "{{number4:40:50}}"

      - type: "create_csv"
        name: "orders_with_regions_csv"
        target_file: "{{artifacts}}/{{qs_id}}/orders.csv"
        content:
          headers: ["ORDER_ID", "ORDER_AMOUNT", "QUANTITY", "STATUS", "CUSTOMER_REGION"]
          header_types: ["id", "currency", "score", "status", "region"]
          rows: "{{number5:100:150}}"

#####
#Section 500: Database processing
- question_id: 501
  samples: 30
  template: "How many orders above {{number1:10000:20000:currency}} are there from customers in the {{semantic2:region}} region in {{artifacts}}/{{qs_id}}/{{entity1}}.db? Create a JSON file {{artifacts}}/{{qs_id}}/big_orders_count.json that contains the answer using 'numBigOrders' as key."

```

```

scoring_type: "readfile_jsonmatch"
file_to_read: "{{artifacts}}/{{qs_id}}/big_orders_count.json"
expected_content: "{\"num_big_orders\": {$sqlite_query:SELECT COUNT(*) FROM enterprise_orders o JOIN enterprise_customers c ON o.CUST_REF = c.CUST_ID WHERE c.LOC_CD = '{{semantic2:region}}' AND o.ORD_AMT > {{number1:10000:20000:currency}}:TARGET_FILE[order_db]}}}"
sandbox_setup:
components:
- type: "create_sqlite"
  name: "order_db"
  target_file: "{{artifacts}}/{{qs_id}}/{{entity1}}.db"
  content:
    tables:
      - name: "enterprise_companies"
        columns:
          - {name: "COMP_ID", type: "auto_id"}
          - {name: "COMP_NM", type: "TEXT", data_type: "company"}
          - {name: "INDUSTRY", type: "TEXT", data_type: "category"}
        rows: "{{number2:8:15}}"
      - name: "enterprise_products"
        columns:
          - {name: "PROD_ID", type: "auto_id"}
          - {name: "PROD_NM", type: "TEXT", data_type: "product"}
          - {name: "CATEGORY", type: "TEXT", data_type: "category"}
          - {name: "BASE_PRICE", type: "INTEGER", data_type: "price"}
        rows: "{{number3:20:30}}"
      - name: "enterprise_customers"
        columns:
          - {name: "CUST_ID", type: "auto_id"}
          - {name: "CUST_NM", type: "TEXT", data_type: "person_name"}
          - {name: "COMP_REF", type: "INTEGER", foreign_key:
"enterprise_companies.COMP_ID"}
            - {name: "LOC_CD", type: "TEXT", data_type: "region"}
            - {name: "DEPT_CD", type: "TEXT", data_type: "department"}
        rows: "{{number4:200:250}}"
      - name: "enterprise_orders"
        columns:
          - {name: "ORD_ID", type: "auto_id"}
          - {name: "CUST_REF", type: "INTEGER", foreign_key:
"enterprise_customers.CUST_ID"}
            - {name: "PROD_REF", type: "INTEGER", foreign_key:
"enterprise_products.PROD_ID"}
              - {name: "ORD_AMT", type: "INTEGER", data_type: "currency"}
              - {name: "QUANTITY", type: "INTEGER", data_type: "score"}
              - {name: "STAT_CD", type: "TEXT", data_type: "status"}
        rows: "{{number5:750:1000}}"
    - question_id: 502
      samples: 30
      template: "Analyze the database {{artifacts}}/{{qs_id}}/{{entity1}}.db and create a comprehensive business report at {{artifacts}}/{{qs_id}}/business_analysis.json with the following information: (1) How many orders above {{number1:15000:35000:currency}} are there from {{semantic1:company}} customers in the {{semantic2:region}} region? (2) What is the total order value for all {{semantic3:department}} department customers? (3) How many unique products have been ordered by customers from {{semantic1:company}}? (4) What is the average order amount for {{semantic4:category}} category products (rounded to 2 decimal places)? (5) How many customers have made orders with quantities above {{number2:70:85}}? (6) What is the total number of {{semantic5:status}} status orders?"

```

```

Create a JSON with keys: 'high_value_orders', 'dept_total_value',
'company_unique_products', 'category_avg_amount', 'high_quantity_customers', and
'status_order_count'.
scoring_type: "readfile_jsonmatch"
file_to_read: "{{artifacts}}/{{qs_id}}/business_analysis.json"
expected_content: "{\"high_value_orders\": {{sqlite_query:SELECT COUNT(*) FROM enterprise_orders o JOIN enterprise_customers c ON o.CUST_REF = c.CUST_ID JOIN enterprise_companies comp ON c.COMP_REF = comp.COMP_ID WHERE comp.COMP_NM = '{{semantic1:company}}' AND c.LOC_CD = '{{semantic2:region}}' AND o.ORD_AMT > {{number1:15000:35000:currency}}}:TARGET_FILE[order_db]}, \"dept_total_value\": {{sqlite_query:SELECT COALESCE(SUM(o.ORD_AMT), 0) FROM enterprise_orders o JOIN enterprise_customers c ON o.CUST_REF = c.CUST_ID WHERE c.DEPT_CD = '{{semantic3:department}}':TARGET_FILE[order_db]}}, \"company_unique_products\": {{sqlite_query:SELECT COUNT(DISTINCT o.PROD_REF) FROM enterprise_orders o JOIN enterprise_customers c ON o.CUST_REF = c.CUST_ID JOIN enterprise_companies comp ON c.COMP_REF = comp.COMP_ID WHERE comp.COMP_NM = '{{semantic1:company}}':TARGET_FILE[order_db]}}, \"category_avg_amount\": {{sqlite_query:SELECT COALESCE(ROUND(AVG(o.ORD_AMT), 2), 0) FROM enterprise_orders o JOIN enterprise_products p ON o.PROD_REF = p.PROD_ID WHERE p.CATEGORY = '{{semantic4:category}}':TARGET_FILE[order_db]}}, \"high_quantity_customers\": {{sqlite_query:SELECT COUNT(DISTINCT c.CUST_ID) FROM enterprise_orders o JOIN enterprise_customers c ON o.CUST_REF = c.CUST_ID WHERE o.QUANTITY > {{number2:70:85}}}:TARGET_FILE[order_db]}}, \"status_order_count\": {{sqlite_query:SELECT COUNT(*) FROM enterprise_orders WHERE STAT_CD = '{{semantic5:status}}':TARGET_FILE[order_db]}}}"
sandbox_setup:
components:
- type: "create_sqlite"
  name: "order_db"
  target_file: "{{artifacts}}/{{qs_id}}/{{entity1}}.db"
  content:
    tables:
      - name: "enterprise_companies"
        columns:
          - {name: "COMP_ID", type: "auto_id"}
          - {name: "COMP_NM", type: "TEXT", data_type: "company"}
          - {name: "INDUSTRY", type: "TEXT", data_type: "category"}
        rows: "{{number3:8:15}}"
      - name: "enterprise_products"
        columns:
          - {name: "PROD_ID", type: "auto_id"}
          - {name: "PROD_NM", type: "TEXT", data_type: "product"}
          - {name: "CATEGORY", type: "TEXT", data_type: "category"}
          - {name: "BASE_PRICE", type: "INTEGER", data_type: "price"}
        rows: "{{number4:60:70}}"
      - name: "enterprise_customers"
        columns:
          - {name: "CUST_ID", type: "auto_id"}
          - {name: "CUST_NM", type: "TEXT", data_type: "person_name"}
          - {name: "COMP_REF", type: "INTEGER", foreign_key: "enterprise_companies.COMP_ID"}
        rows: "{{number5:200:250}}"
      - name: "enterprise_orders"
        columns:
          - {name: "ORD_ID", type: "auto_id"}
          - {name: "CUST_REF", type: "INTEGER", foreign_key: "enterprise_customers.CUST_ID"}
        rows: "{{number6:100:150}}"

```

```

        - {name: "PROD_REF", type: "INTEGER", foreign_key:
"enterprise_products.PROD_ID"}
        - {name: "ORD_AMT", type: "INTEGER", data_type: "currency"}
        - {name: "QUANTITY", type: "INTEGER", data_type: "score"}
        - {name: "STAT_CD", type: "TEXT", data_type: "status"}
    rows: "{{number6:750:1000}}"

- question_id: 503
  samples: 30
  template: "Analyze the business database at {{artifacts}}/{{qs_id}}/{{entity1}}.db and determine: What is the total revenue generated from {{semantic1:category}} category products sold to customers in the {{semantic2:region}} region? Save your answer as a JSON file at {{artifacts}}/{{qs_id}}/categoryRegionalRevenue.json with the key 'totalCategoryRegionalRevenue'."
  scoring_type: "readfile_jsonmatch"
  file_to_read: "{{artifacts}}/{{qs_id}}/categoryRegionalRevenue.json"
  expected_content: "{\"totalCategoryRegionalRevenue\": {\"sqlite_query\":SELECT COALESCE(SUM(o.ORDER_AMT), 0) FROM orders o JOIN customers c ON o.CUSTOMER_ID = c.CUSTOMER_ID JOIN products p ON o.PRODUCT_ID = p.PRODUCT_ID WHERE p.CATEGORY = '{{semantic1:category}}' AND c.REGION = '{{semantic2:region}}':TARGET_FILE[business_db]}}"
  sandbox_setup:
    components:
      - type: "create_sqlite"
        name: "business_db"
        target_file: "{{artifacts}}/{{qs_id}}/{{entity1}}.db"
        content:
          tables:
            # DISTRACTOR: Company info (not needed for the query)
            - name: "companies"
              columns:
                - {name: "COMPANY_ID", type: "auto_id"}
                - {name: "COMPANY_NAME", type: "TEXT", data_type: "company"}
                - {name: "INDUSTRY", type: "TEXT", data_type: "category"}
      rows: "{{number2:10:20}}"

      # DISTRACTOR: Employee data (not needed)
      - name: "employees"
        columns:
          - {name: "EMPLOYEE_ID", type: "auto_id"}
          - {name: "EMPLOYEE_NAME", type: "TEXT", data_type: "person_name"}
          - {name: "COMPANY_ID", type: "INTEGER", foreign_key:
"companies.COMPANY_ID"}
          - {name: "DEPARTMENT", type: "TEXT", data_type: "department"}
          - {name: "SALARY", type: "INTEGER", data_type: "salary"}
      rows: "{{number3:50:200}}"

      # REQUIRED: Customer data (has REGION)
      - name: "customers"
        columns:
          - {name: "CUSTOMER_ID", type: "auto_id"}
          - {name: "CUSTOMER_NAME", type: "TEXT", data_type: "person_name"}
          - {name: "REGION", type: "TEXT", data_type: "region"}
          - {name: "DEPARTMENT", type: "TEXT", data_type: "department"}
      rows: "{{number4:50:100}}"

      # REQUIRED: Product data (has CATEGORY)
      - name: "products"
        columns:
          - {name: "PRODUCT_ID", type: "auto_id"}
          - {name: "PRODUCT_NAME", type: "TEXT", data_type: "product"}

```

```

    - {name: "CATEGORY", type: "TEXT", data_type: "category"}
    - {name: "VARIANT", type: "TEXT", data_type: "entity_pool"}
    - {name: "MODEL", type: "TEXT", data_type: "course"}
    - {name: "BASE_PRICE", type: "INTEGER", data_type: "price"}
rows: "{{number5:70:100}}"

# DISTRACTOR: Supplier data (not needed)
- name: "suppliers"
  columns:
    - {name: "SUPPLIER_ID", type: "auto_id"}
    - {name: "SUPPLIER_NAME", type: "TEXT", data_type: "company"}
    - {name: "CONTACT_PERSON", type: "TEXT", data_type: "person_name"}
    - {name: "REGION", type: "TEXT", data_type: "region"}
rows: "{{number6:50:100}}"

# REQUIRED: Orders data (connects customers and products, has revenue)
- name: "orders"
  columns:
    - {name: "ORDER_ID", type: "auto_id"}
    - {name: "CUSTOMER_ID", type: "INTEGER", foreign_key:
"customers.CUSTOMER_ID"}
    - {name: "PRODUCT_ID", type: "INTEGER", foreign_key: "products.PRODUCT_ID"}
    - {name: "ORDER_AMT", type: "INTEGER", data_type: "currency"}
    - {name: "ORDER_DATE", type: "TEXT", data_type: "date"}
rows: "{{number7:150:200}}"

```

Listing 1: Kamiwaza Agentic Merit Index (KAMI) v0.1 subset

Appendix B Tool Descriptions

This appendix provides detailed descriptions of all tools available to the LLM agents in our evaluation framework.

Each tool includes its complete parameter specifications and return value descriptions.

With just slight formatting differences, this is what gets loaded into the system prompt of the agents tested.

Filesystem Tools

- `get_cwd` - Get the current working directory.

Parameters: None

Returns: String - information about the current working directory

- `read_file` - Read a file in the filesystem with optional line numbering, range selection, and debug formatting.

Parameters:

- `path` (required, string) - path and filename of the file to read
- `show_line_numbers` (optional, boolean) - whether to include line numbers (defaults to False)
- `start_line` (optional, integer) - first line to read, 1-indexed (defaults to 1)
- `end_line` (optional, integer) - last line to read, 1-indexed, None for all lines (defaults to None)
- `show_repr` (optional, boolean) - whether to show Python's `repr()` of each line, revealing whitespace and special characters (defaults to False)

Returns: String - the contents of the file (potentially formatted with line numbers or `repr()`), or an error message if reading fails

- `write_file` - Write content to a file in the filesystem.

Parameters:

- `path` (required, string) - path and filename of the file to write
- `content` (required, string) - the content to write to the file

Returns: String - confirmation message indicating success or failure

- `append_file` - Append content to an existing file in the filesystem.

Parameters:

- `path` (required, string) - path and filename of the file to append to
- `content` (required, string) - the content to append to the file

Returns: String - confirmation message indicating success or failure

- `edit_file` - Make a line-based edit to a file by replacing `old_text` with `new_text`. The `old_text` must appear exactly once in the file for safety.

Parameters:

- `path` (required, string) - path and filename of the file to edit
- `old_text` (required, string) - text to be replaced (must match exactly once)
- `new_text` (required, string) - replacement text
- `dry_run` (optional, boolean) - if True, just return the diff without making changes (defaults to False)

Returns: String - confirmation message with diff showing changes, or error message if editing fails

- `create_directory` - Create a new directory in the filesystem.

Parameters:

- `path` (required, string) - path of the directory to create

Returns: String - confirmation message indicating success or failure

- `list_directory` - List the contents of a directory in the filesystem.

Parameters:

- path (optional, string) - path of the directory to list. If not provided, lists the current working directory.

Returns: String - a list of files and directories in the specified path

- `copy_file` - Copy a file from source to destination.

Parameters:

- `source` (required, string) - path to the source file to copy
- `destination` (required, string) - path where the file should be copied to

Returns: String - confirmation message indicating success or failure

- `remove_file` - Remove/delete a single file.

Parameters:

- `path` (required, string) - path to the file to delete

Returns: String - confirmation message indicating success or failure

- `remove_directory` - Remove/delete a directory and all its contents.

Parameters:

- `path` (required, string) - path to the directory to delete

Returns: String - confirmation message indicating success or failure

- `copy_directory` - Copy a directory and all its contents to a new location.

Parameters:

- `source` (required, string) - path to the source directory to copy
- `destination` (required, string) - path where the directory should be copied to

Returns: String - confirmation message indicating success or failure

Git Tools

- `git_clone` - Clone a git repository using HTTPS.

Parameters:

- `repo_url` (required, string) - The HTTPS URL of the repository to clone
- `target_path` (optional, string) - The path where to clone the repository

Returns: String - confirmation message indicating success or failure

- `git_commit` - Stage all changes and create a commit.

Parameters:

- `message` (required, string) - The commit message
- `path` (optional, string) - The path to the git repository (defaults to current directory)

Returns: String - confirmation message indicating success or failure

- `git_restore` - Restore the repository or specific files to a previous state.

Parameters:

- `commit_hash` (optional, string) - The commit hash to restore to. If not provided, unstages all changes
- `path` (optional, string) - The path to the git repository (defaults to current directory)
- `files` (optional, list) - List of specific files to restore. If not provided, restores everything

Returns: String - confirmation message indicating success or failure

- `git_push` - Push commits to a remote repository.

Parameters:

- `remote` (optional, string) - The remote name (defaults to 'origin')
- `branch` (optional, string) - The branch name to push to (defaults to 'main')
- `path` (optional, string) - The path to the git repository (defaults to current directory)

Returns: String - confirmation message indicating success or failure

- `git_log` - Get the commit history of the repository.

Parameters:

- `path` (optional, string) - The path to the git repository (defaults to current directory)
- `max_count` (optional, integer) - Maximum number of commits to return
- `since` (optional, string) - Get commits since this date (e.g., "2024-01-01" or "1 week ago")

Returns: String - JSON formatted commit history with hash, author, date, and message for each commit

- `git_show` - Get detailed information about a specific commit.

Parameters:

- `commit_hash` (required, string) - The hash of the commit to inspect
- `path` (optional, string) - The path to the git repository (defaults to current directory)

Returns: String - JSON formatted commit details including metadata and changed files

- `git_status` - Get the current status of the repository.

Parameters:

- `path` (optional, string) - The path to the git repository (defaults to current directory)

Returns: String - JSON formatted repository status including staged, unstaged, and untracked changes

- `git_diff` - Get the differences between commits, staged changes, or working directory.

Parameters:

- `path` (optional, string) - The path to the git repository (defaults to current directory)
- `commit1` (optional, string) - First commit hash for comparison
- `commit2` (optional, string) - Second commit hash for comparison
- `staged` (optional, boolean) - If True, show staged changes (ignored if commits are specified)
- `file_path` (optional, string) - Path to specific file to diff

Returns: String - JSON formatted diff information including: Summary (files changed, total additions/deletions) and Detailed changes per file with hunks showing exact line modifications

Web Tools

- `brave_web_search` - Search the web using Brave Search API. The responses here only contain summaries. Use `fetch_web_page` to get the full contents of interesting search results.

Parameters:

- `query` (required, string) - the search query to submit to Brave
- `count` (optional, integer) - the number of results to return, defaults to 10

Returns: Object - a JSON object containing search results or error information from the Brave Search API

- `fetch_web_page` - Fetch content from a specified URL. This is a good tool to use after doing a `brave_web_search`, in order to get more details from interesting search results.

Parameters:

- `url` (required, string) - the URL to fetch content from
- `headers` (optional, dictionary) - custom headers to include in the request, defaults to a standard User-Agent
- `timeout` (optional, integer) - request timeout in seconds, defaults to 30

- `clean` (optional, boolean) - whether to extract only the main content, defaults to True

Returns: String - the cleaned web page content as text, or an error object if the request fails

Python Tools

- `python_execute_file` - Execute a Python file and return its output.

Parameters:

- `file_path` (required, string) - Path to the Python file to execute

Returns: String - The output of the execution or an error message if execution fails

- `python_check_syntax` - Check the syntax of Python code.

Parameters:

- `code` (optional, string) - Python code to check

- `file_path` (optional, string) - Path to a Python file to check

Returns: String - Result of the syntax check

- `python_execute_code` - Execute arbitrary Python code and return its output.

Parameters:

- `code` (required, string) - Python code to execute

Returns: String - The output of the execution or an error message if execution fails

SQLite Tools

- `sqlite_connect` - Connect to a SQLite database file and verify the connection.

Parameters:

- `database_path` (required, string) - Path to the SQLite database file

Returns: String - confirmation message with basic database info, or error message if connection fails

- `sqlite_execute_query` - Execute a SELECT query on SQLite database (read-only operations).

Parameters:

- `database_path` (required, string) - Path to the SQLite database file

- `query` (required, string) - SQL SELECT query to execute

- `limit` (optional, integer) - Maximum number of rows to return (defaults to 1000)

- `timeout` (optional, integer) - Query timeout in seconds (defaults to 30)

Returns: String - JSON formatted results with columns and rows, or error message if execution fails

- `sqlite_execute_command` - Execute INSERT, UPDATE, DELETE, or DDL commands on SQLite database.

Parameters:

- `database_path` (required, string) - Path to the SQLite database file

- `command` (required, string) - SQL command to execute (INSERT, UPDATE, DELETE, CREATE, DROP, etc.)

- `timeout` (optional, integer) - Command timeout in seconds (defaults to 30)

Returns: String - confirmation message with affected rows count, or error message if execution fails

- `sqlite_get_schema` - Get the complete database schema including all tables, columns, and their types.

Parameters:

- `database_path` (required, string) - Path to the SQLite database file

Returns: String - JSON formatted schema information, or error message if retrieval fails

- `sqlite_list_tables` - List all tables and views in the SQLite database.

Parameters:

- `database_path` (required, string) - Path to the SQLite database file

Returns: String - JSON formatted list of tables and views, or error message if retrieval fails

Appendix C Example of an Analyzed LLM Conversation Sample

We show an example of the LLM conversation samples analyzed in this paper, pasted verbatim from one of the experiment result files produced during benchmarking.

Every sample analyzed is an entire conversation, triggered by an initial task instruction (taken from the question template and populated with random values as defined) and then autonomously continued by the LLM through the use of tools and the received tool feedback.

The conversation log is a faithful representation of the conversation. It can be a little difficult for humans to parse, as it stores explicit newlines and escaped characters whenever the LLM outputs those tokens - for example, '\n' for newlines, '\\\n' for literal '\n' in strings like code, etc.

```
1 {
2     "question_id": 501,
3     "sample_number": 1,
4     "timestamp": "2025-11-02T13:33:48.139231",
5     "initial_question": "How many orders above 19997 are there from customers in the Midwest region in /home/jv/picard1/test_artifacts/q501_s1/starlight.db? Create a JSON file /home/jv/picard1/test_artifacts/q501_s1/big_orders_count.json that contains the answer using 'num_big_orders' as key.",
6     "conversation_history": [
7         {
8             "role": "user",
9             "content": "How many orders above 19997 are there from customers in the Midwest region in /home/jv/picard1/test_artifacts/q501_s1/starlight.db? Create a JSON file /home/jv/picard1/test_artifacts/q501_s1/big_orders_count.json that contains the answer using 'num_big_orders' as key."
10        },
11        {
12            "role": "assistant",
13            "content": "I'll help you solve this problem by querying the SQLite database and creating the JSON file. Let me start by examining the database structure to understand the tables and relationships.\n<tool_call>\n    \"name\": \"sqlite_connect\",\n    \"input\": {\n        \"database_path\": \"/home/jv/picard1/test_artifacts/q501_s1/starlight.db\"\n    }\n</tool_call>\n",
14        },
15        {
16            "role": "user",
17            "content": "Tool result: ``Connected to existing database:\n/home/jv/picard1/test_artifacts/q501_s1/starlight.db\nSQLite version: 3.37.2\nTables found: 5````",
18        },
19        {
20            "role": "assistant",
21            "content": "Now let me examine the schema to understand the table structure and relationships.\n<tool_call>\n    \"name\": \"sqlite_get_schema\",\n    \"input\": {\n        \"database_path\": \"/home/jv/picard1/test_artifacts/q501_s1/starlight.db\"\n    }\n</tool_call>\n",
22        },
23        {
24            "role": "user",
25            "content": "Tool result: ``{\n    \"database\": \"/home/jv/picard1/test_artifacts/q501_s1/starlight.db\",\n    \"tables\": [\n        {\n            \"name\": \"enterprise_companies\",\n            \"columns\": [\n                {\n                    \"name\": \"COMP_ID\",\n                    \"type\": \"INTEGER\",\n                    \"not_null\": false,\n                    \"default_value\": null,\n                    \"primary_key\": true\n                },\n                {\n                    \"name\": \"COMP_NM\",\n                    \"type\": \"TEXT\",\n                    \"not_null\": false,\n                    \"default_value\": null,\n                    \"primary_key\": false\n                },\n                {\n                    \"name\": \"INDUSTRY\",\n                    \"type\": \"TEXT\",\n                    \"not_null\": false,\n                    \"default_value\": null,\n                    \"primary_key\": false\n                }\n            ],\n            \"create_sql\": \"CREATE TABLE enterprise_companies (COMP_ID INTEGER PRIMARY KEY AUTOINCREMENT, COMP_NM TEXT, INDUSTRY TEXT)\"\n        },\n        {\n            \"name\": \"\""

```

```

"enterprise_customers",\n      "columns": [\n        {\n          "name": "CUST_ID",\n          "type": "INTEGER",\n          "not_null": false,\n          "default_value": null,\n          "primary_key": true,\n          "name": "CUST_NM",\n          "type": "TEXT",\n          "not_null": false,\n          "default_value": null,\n          "primary_key": false\n        },\n        {\n          "name": "COMP_REF",\n          "type": "INTEGER",\n          "not_null": false,\n          "default_value": null,\n          "primary_key": false\n        },\n        {\n          "name": "LOC_CD",\n          "type": "TEXT",\n          "not_null": false,\n          "default_value": null,\n          "primary_key": false\n        },\n        {\n          "name": "DEPT_CD",\n          "type": "TEXT",\n          "not_null": false,\n          "default_value": null,\n          "primary_key": false\n        }\n      ],\n      "create_sql": "CREATE TABLE enterprise_customers\n(CUST_ID INTEGER PRIMARY KEY AUTOINCREMENT, CUST_NM TEXT, COMP_REF INTEGER, LOC_CD TEXT,\nDEPT_CD TEXT)\n",\n      "columns": [\n        {\n          "name": "ORD_ID",\n          "type": "INTEGER",\n          "not_null": false,\n          "default_value": null,\n          "primary_key": true\n        },\n        {\n          "name": "CUST_REF",\n          "type": "INTEGER",\n          "not_null": false,\n          "default_value": null,\n          "primary_key": false\n        },\n        {\n          "name": "PROD_REF",\n          "type": "INTEGER",\n          "not_null": false,\n          "default_value": null,\n          "primary_key": false\n        },\n        {\n          "name": "ORD_AMT",\n          "type": "INTEGER",\n          "not_null": false,\n          "default_value": null,\n          "primary_key": false\n        },\n        {\n          "name": "QUANTITY",\n          "type": "INTEGER",\n          "not_null": false,\n          "default_value": null,\n          "primary_key": false\n        },\n        {\n          "name": "STAT_CD",\n          "type": "TEXT",\n          "not_null": false,\n          "default_value": null,\n          "primary_key": false\n        }\n      ],\n      "create_sql": "CREATE TABLE enterprise_orders\n(ORD_ID INTEGER PRIMARY KEY AUTOINCREMENT, CUST_REF INTEGER, PROD_REF INTEGER, ORD_AMT\nINTEGER, QUANTITY INTEGER, STAT_CD TEXT)\n",\n      "columns": [\n        {\n          "name": "PROD_ID",\n          "type": "INTEGER",\n          "not_null": false,\n          "default_value": null,\n          "primary_key": true\n        },\n        {\n          "name": "PROD_NM",\n          "type": "TEXT",\n          "not_null": false,\n          "default_value": null,\n          "primary_key": false\n        },\n        {\n          "name": "CATEGORY",\n          "type": "TEXT",\n          "not_null": false,\n          "default_value": null,\n          "primary_key": false\n        },\n        {\n          "name": "BASE_PRICE",\n          "type": "INTEGER",\n          "not_null": false,\n          "default_value": null,\n          "primary_key": false\n        },\n        {\n          "name": "sqlite_sequence",\n          "type": "\\",\\n\n          "not_null": false,\n          "default_value": null,\n          "primary_key": false\n        }\n      ],\n      "create_sql": "CREATE TABLE enterprise_products\n(PROD_ID INTEGER PRIMARY KEY AUTOINCREMENT, PROD_NM TEXT, CATEGORY\nTEXT, BASE_PRICE INTEGER)\n",\n      "columns": [\n        {\n          "name": "seq",\n          "type": "\\",\\n\n          "not_null": false,\n          "default_value": null,\n          "primary_key": false\n        },\n        {\n          "name": "sqlite_sequence",\n          "type": "\\",\\n\n          "not_null": false,\n          "default_value": null,\n          "primary_key": false\n        }\n      ],\n      "views": []\n    },\n    {\n      "role": "assistant",\n      "content": "Now I need to understand what the \"Midwest region\" refers to. Let me\ncheck if there's a location code mapping or if I need to look at the data to understand\nwhat LOC_CD values correspond to the Midwest region:\n<tool_call>\n  \"name\":\n  \"sqlite_execute_query\",\\n  \"input\": {\n    \"database_path\":\n      \"/home/jv/picard1/test_artifacts/q501_s1/starlight.db\",\\n      \"query\": \"SELECT\n      DISTINCT LOC_CD FROM enterprise_customers ORDER BY LOC_CD\"\n  }\\n}</tool_call>\n  },\n  {\n  }
```

```

  "role": "user",
  "content": "Tool result: '''{\n    \"success\": true,\n    \"columns\": [\n        \"LOC_CD\"\n    ],\n    \"row_count\": 10,\n    \"data\": [\n        {\n            \"LOC_CD\": \"East\",\n            \"North\": 1,\n            \"Southwest\": 1,\n            \"West\": 1\n        },\n        {\n            \"LOC_CD\": \"Midwest\",\n            \"North\": 1,\n            \"Southwest\": 1,\n            \"West\": 1\n        },\n        {\n            \"LOC_CD\": \"Northeast\",\n            \"North\": 1,\n            \"Southwest\": 1,\n            \"West\": 1\n        },\n        {\n            \"LOC_CD\": \"Southeast\",\n            \"North\": 1,\n            \"Southwest\": 1,\n            \"West\": 1\n        }\n    ]\n}'''"
},
{
  "role": "assistant",
  "content": "Perfect! I can see that \"Midwest\" is one of the location codes. Now I'll query for orders above 19997 from customers in the Midwest region:\n<tool_call>\n\n  \"name\": \"sqlite_execute_query\", \"input\": {\n      \"database_path\": \"/home/jv/picard1/test_artifacts/q501_s1/starlight.db\", \"query\": \"SELECT COUNT(*) as num_big_orders FROM enterprise_orders o JOIN enterprise_customers c ON o.CUST_REF = c.CUST_ID WHERE o.ORD_AMT > 19997 AND c.LOC_CD = 'Midwest'\"\n}\n</tool_call>"
},
{
  "role": "user",
  "content": "Tool result: '''{\n    \"success\": true,\n    \"columns\": [\n        \"num_big_orders\"\n    ],\n    \"row_count\": 1,\n    \"data\": [\n        {\n            \"num_big_orders\": 68\n        }\n    ]\n}'''"
},
{
  "role": "assistant",
  "content": "Great! I found that there are 68 orders above 19997 from customers in the Midwest region. Now I'll create the JSON file with this result:\n<tool_call>\n\n  \"name\": \"write_file\", \"input\": {\n      \"path\": \"/home/jv/picard1/test_artifacts/q501_s1/big_orders_count.json\", \"content\": \"{\n        \"num_big_orders\": 68\n    }\"\n}\n</tool_call>"
},
{
  "role": "user",
  "content": "Tool result: '''File written successfully:\n/home/jv/picard1/test_artifacts/q501_s1/big_orders_count.json'''"
},
{
  "role": "assistant",
  "content": "Perfect! I have successfully completed your request. Here's what I did:\n1. Connected to the SQLite database at \'/home/jv/picard1/test_artifacts/q501_s1/starlight.db'\n2. Examined the database schema to understand the table structure\n3. Found that \"Midwest\" is one of the location codes in the 'enterprise_customers' table\n4. Executed a query to count orders above 19997 from customers in the Midwest region\n5. Found that there are 68 such orders\n6. Created the JSON file '/home/jv/picard1/test_artifacts/q501_s1/big_orders_count.json' with the result:\n\nThe file has been successfully created with the correct count of big orders from Midwest customers."
}
]
}

```

Listing 2: DeepSeek v3.1 conversation sample, Q501, Sample 1