# Agentic Repository Mining: A Multi-Task Evaluation[*]

Johannes Härtel
j.a.hartel@vu.nl
Vrije Universiteit Amsterdam
Netherlands

## Abstract

Mining software repositories often requires classifying artifacts like commits, reviews, code lines, or entire repositories into categories. Human labeling is expensive and error-prone; limited context frequently leads to misclassifications or uncertainty in labels. We investigate whether LLM agents that dynamically explore repositories through standard bash commands can match the classification quality of simple LLMs that receive pre-engineered context. Across four tasks, eight approach configurations, and 4943 classifications, agents achieve competitive accuracy despite retrieving their own context. The primary advantage is robustness: agents avoid context-window overflows and scale independently of artifact size. A manual diagnosis of 100 cases where approaches disagree with the ground truth reveals specification ambiguities and labels produced under limited context, suggesting that accuracy against such ground truth may underestimate approaches with broader context access.

## CCS Concepts

• **Software and its engineering → Software verification and validation**; Software development techniques.

## Keywords

mining software repositories, LLM agents, software artifact classification, dynamic context retrieval, empirical evaluation

## 1 Introduction

Mining software repositories often requires the classification and manual labeling of artifacts like commits, reviews, code lines, or entire repositories into categories. This is the empirical foundation for studies that examine such artifacts. Examples are studies on bug fixes [18], security defects [14], repositories [29], or maintenance effort [23].

*ML-based solutions.* Labeling by humans can be time-consuming, expensive, and error-prone. Directly using an ML classifier is a tempting solution but needs existing labeled data in the first place. Hybrid approaches exist, like active learning, that repeatedly train a classifier and then manually label promising candidates selected by the classifier [11, 14, 45]. Recently, zero-shot learning with LLMs started to offer a way to mitigate initial training data. Like a human annotator, the LLMs can be prompted with labeling guidelines and the artifacts to label. However, what contextual artifacts to include in the prompt is still a challenge to human engineering [4, 7, 30, 38].

*Agentic Repository Mining.* Agentic repository mining bypasses the need for a human to pre-engineer context. Given only a starting point like a commit SHA, the agent autonomously explores the repository through tool calls until it gathers sufficient evidence for classification [1, 12, 13]. This removes per-task context engineering but raises the question of whether self-retrieved context matches human-curated input. While recent work studies agents as subjects in repositories [9, 19, 27, 35], we use them to mine repositories.

*Research Questions.* We evaluate such an option by putting agents into a direct comparison to a simple LLM counterpart without tools but pre-engineered context. We target four classification tasks from related work with different repository artifacts (lines, comments, repositories, commits). All tasks have manually labeled ground truth; agents and LLMs get the same labeling guidelines as in the original studies. We ask three research questions:

- **RQ1:** How does agent and simple LLM accuracy compare to a human ground truth?
- **RQ2:** How do resource usage (tokens, time, costs) and failure modes compare between agents and simple LLMs?
- **RQ3:** Why do agents and simple LLMs disagree with the ground truth?

*Tasks. Herbold et al.* [18] label whether a changed line in a bug-fixing commit contributes to the fix or not; *Härtel* [14] determines whether a PR review comment discusses a potential security defect; *Munaiah et al.* [29] distinguish engineered software projects from homework, toy projects, or personal experiments; and *Levin et al.* [23] classify commits as corrective, adaptive, or perfective.

*Short Answers.* The major advantage of agents we find is their *robustness to context size.* Agents selectively read and thereby keep context, tokens, and cost largely independent of what artifacts are maximally available. Simple LLMs run into context overflow errors when artifacts get large. Agents are pricier by factors of 1.2 to 3.2, but pay off when artifacts exceed a certain size. Accuracy does not show a clear winner, but agents retrieve their own context rather than receiving it pre-engineered, making comparable accuracy a stronger result. We diagnose 100 cases where approaches disagree with the ground truth, noticing labeling guidelines need updates.

*Contributions.*

- We introduce a simple *agentic repository mining* framework to classify artifacts via bash commands in isolated containers with access to the repository.
- We compare accuracy, resource usage, and failure modes of agents with their simple counterparts across four tasks.
- We diagnose disagreements with the ground truth to understand where approaches fail and where labeling guidelines might need to be updated.

*Data Availability.* We provide code, prompts, and results under:

- **Figshare:** https://doi.org/10.6084/m9.figshare.31136470.v1

---

[*] Accepted at the 30th International Conference on Evaluation and Assessment in Software Engineering (EASE 2026).

- **GitHub:** https://github.com/johanneshaertel/EASE_2026_agentic_repository_mining

*Road-Map.* Sec. 2 introduces agents; Sec. 3 describes the compared approaches; Sec. 4 presents the tasks; Sec. 5 details our methodology; Sec. 6 reports results; Sec. 7 outlines threats and limitations; Sec. 8 discusses related work; and Sec. 9 concludes.

## 2 Background

Context matters for classification. In a recent postmortem, Anthropic reports expanding repository access for their code-review agents after finding that this context was necessary to detect a critical bug [5]. Choosing the right context for classification has been a long-standing problem in mining software repositories (MSR).

### 2.1 Human-Engineered Context

Human-engineered context in MSR even existed before the era of LLMs. Established studies examine what input features machine learning needs to make good classifications (preventing 'garbage in, garbage out'). Studies examine code attributes for defect classification [25, 46, 47], aggregation [44], or granularity [34].

With the rise of LLMs, context is now engineered as a prompt and passed to an LLM, but the same core challenge applies [7, 30, 38]. What context needs to be added to the prompt to get better classifications? A recent example by Antal et al. is adding metadata on security defects to the prompt to improve performance [4].

### 2.2 Human Context Retrieval

If humans are to classify an artifact, they may follow a special strategy, not looking at fixed numbers of lines or code attributes of the artifact, like classical ML approaches would do [36].

For example, if a classification depends on precise knowledge of a class at a certain revision, humans may clone the repository, check out the relevant revision, and search for the class in the working copy. This might reveal another class with more critical information, a class that might be difficult to anticipate beforehand. In a nutshell, such a sequence of bash commands, e.g., `git clone`, `git checkout`, and `rg ClassName`, has two important properties:

- First, it is **autoregressive**. Classification is done in multiple steps, and each step depends on previous results. There is no fixed vector of information but a growing sequence. A threshold might constrain exploration depth.
- Second, it uses **existing development tools**, in our example, bash access to explore the repository.

### 2.3 Agentic Context Retrieval

Agentic repository mining builds on the same strategy. We show an example from a run of our study in Fig. 1 where an agent extracts the necessary context by a cascade of bash commands to dynamically explore what is needed for the classification. The agent iteratively inspects files, searches code, and examines the history until it gathers sufficient evidence and concludes in the final step. Standard bash commands are the universal tool used; no specialized API or retrieval system is needed. Systematic evaluations of agents used for MSR are still in early stages [1, 12, 13].

**hartel_security: 9343ed067735**

Answer: no  Expected: **no**  Steps: **5**

**Q:** Does this PR review comment discuss a potential security defect?

**LLM RESPONSE** 2.6s

I'll analyze the PR review comment to determine if it discusses a potential security defect. First, let's look at the review comment content.

**TOOL CALL: EXECUTE_COMMAND**

```
cat /extra/review_body.txt
```

**CAT /EXTRA/REVIEW_BODY.TXT** EXIT: 0

```
Once this is merged, I'll open an issue to track the fact that we should change all the remainin
```

**LLM RESPONSE** 2.8s

Now let's look at the PR metadata to understand the context better:

**TOOL CALL: EXECUTE_COMMAND**

```
cat /extra/pull_data.json | grep -E "title|body" | head -10
```

**CAT /EXTRA/PULL_DATA.JSON | GREP -E "TITLE|BODY" | HEAD -10** EXIT: 0

```
  "title": "Set xpack.security.enabled to true for all licenses",
  "body": "This change sets the default value for xpack.security.enabled to true\r\nfor all lice
              "body": "We could probably skip mentioning the license here since this statement
              "body": "I think all of this can go.\r\nThis filter isn't installed if security
              "body": "Good point! I've changed the methods in this ActionFilter to not check
[truncated, 1963 chars total]
```

**LLM RESPONSE** 2.3s

Let me also check the full PR description and understand the changes being proposed:

**TOOL CALL: EXECUTE_COMMAND**

```
cat /extra/pull_data.json | grep -A20 "\"body\":" | head -20
```

**CAT /EXTRA/PULL_DATA.JSON | GREP -A20 "\"BODY\":" | HEAD -20** EXIT: 0

```
  "body": "This change sets the default value for xpack.security.enabled to true\r\nfor all lice
  "createdAt": "2021-04-27T09:50:48Z",
  "mergeCommit": {
    "oid": "fed790e4e4b71db7c02fe05016c236e9ea4cbbab"
  },
  "closedAt": "2021-08-09T06:36:02Z",
  "author": {
    "login": "jkakavas"
  },
  "reviewThreads": {
    "edges": [
      {
        "node": {
          "comments": {
            "nodes": [
              {
                "__typename": "PullRequestReviewComment",
                "path": "x-pack/plugin/security/src/main/java/org/elasticsearch/xpack/security/a
  ...
[truncated, 902 chars total]
```

**LLM RESPONSE** 2.3s

Let me look at the PR title again for clarity:

**TOOL CALL: EXECUTE_COMMAND**

```
cat /extra/pull_data.json | grep -E "\"title\":" | head -1
```

**CAT /EXTRA/PULL_DATA.JSON | GREP -E "\"TITLE\":" | HEAD -1** EXIT: 0

```
  "title": "Set xpack.security.enabled to true for all licenses",
```

**LLM RESPONSE** 5.7s

Based on this information, I understand that:

1. The PR is about changing the default value for `xpack.security.enabled` to true for all licenses
2. The review comment mentions the "Security On by Default" project
3. The comment states that packaging tests will need to be updated to work with security enabled
4. The reviewer is planning to open an issue to track this work after the current PR is merged

The review comment is discussing a change to the security configuration, but it's not discussing a vulnerability or defect. It's talking about future work that needs to be done to make tests compatible with the new default security settings. The context suggests this is part of a planned enhancement to make security enabled by default, rather than discussing a security defect.

ANSWER: no

**Figure 1: Agent trajectory: Green is LLM reasoning, blue is tool calls, and teal is tool output. The starting prompt is excluded to fit the page (see online material).**

Fig. 1 shows details like how selectively the agent explores the available content, only recovering the particular lines of the files needed to make the classification, not reading the entire files or repository. This is done by piping the results of `cat` into `grep` and `head`. The agent is aware of the schema of GitHub's PR JSON data, which is available as a file dump, as it knows there is just one title (see `head -1`). We provide such traces for all our 4943 experiments.

## 2.4 Agents in a Nutshell

Agentic approaches build upon LLMs using the same autoregressive text generation. The LLM predicts the next token given all previous tokens, iterating until a stop sequence is met. A simple invocation of an LLM typically generates text until it produces an answer.

Agents differ from standard LLM calls by interleaving generation with tool execution. An agent specializes the loop by generating until the generated text conforms to a valid tool invocation, like `... <bash> ls </bash>`. When the model outputs such a tool call, regular LLM-based generation pauses; in our example, by the stop sequence `</bash>`. The tool `ls` is executed, the tool output (listing of files and directories) is appended to the growing string, and the LLM resumes generation conditioned on the entire text so far, including the new tool output. This is done until the next tool call is generated or a final answer is produced. This interleaving of LLM generation and tool execution enables the dynamic context recovery shown in Fig. 1. We refer to the online appendix for our simplistic implementation of an agent that we use in our experiments.

The simplicity of Bash is not a limitation, being the universal tool call that orchestrates other tools. Our experiments focus on bash calls to `git` but extensions to what can be called are straightforward.

## 3 Approaches

We compare simple LLM baselines with fixed pre-engineered context against an agentic harness on top of the same LLMs that enables dynamic context exploration via tool execution logic. For simple LLM invocations, we evaluate chain-of-thought, no chain-of-thought, and a pure memorization baseline without any context. For agents, we evaluate two tool-calling variants.

## 3.1 Base Prompt

All compared approaches get labeling guidelines close to the human annotators in the original studies. This includes the classification question and the list of categories that are possible answers. We allow models to also answer with *unclear* as an additional category rather than guessing. What follows is specific to the approaches.

## 3.2 Simple LLMs

Simple LLM approaches are executed in a single turn until a final answer is produced. Therefore, additional context must be appended after the base prompt before passing it to the LLM. This needs to be engineered by us adding resources like commit diffs, file contents, or PR descriptions specific to each task. We keep this close to the original studies with partial deviations that we discuss in Sec. 4.

### 3.2.1 *Memorization.*

We evaluate classification without context, where we only append the minimal identification of central artifacts to see if the LLM can recall the correct label anyhow. Such an effect would be a strong indication that either the context or the final labels are memorized because they are in the training data. We avoid the category 'unclear' for this variant.

### 3.2.2 *Chain-of-Thought.*

We evaluate chain-of-thought prompting in two modes: with and without reasoning. The standard variant is instructed to produce a reasoning trace followed by the answer. The no-chain-of-thought variant (no-CoT) is explicitly instructed to produce only the final answer without any reasoning trace. We enforce this by a strong prompt that leads to a direct answer.

## 3.3 Agents

Our agentic approaches dynamically append context themselves in the agentic loop, having access to a shell executed in a sandboxed container where the repository is cloned. There is no need for task-specific engineering of the context. The sandboxed Docker environment ensures reproducibility and safety: each run starts from a clean repository state, and the agent cannot affect the host system. The container does not have internet access.

The container also includes additional information as files that are not directly available via `git` to align with what is given to the simple LLMs. This is data that would typically be retrieved over internet APIs that we did not allow the agents to have access to. Compared to the simple LLM approaches, such a setup keeps the starting prompt much smaller but does not limit the agents' information to a subset of what is available to the simple LLMs. We limit agents to 50 exploration steps to bound cost and runtime and explicitly record if the agent runs into the step limit.

### 3.3.1 *Tool Calling.*

We examine two variations in tool calling:

- **Native**: Modern LLM APIs provide native tool calling where the model can output structured tool invocations. Such APIs handle parsing specific to the tools and native to the model. It is a vendor-specific, standardized, but less transparent way of tool calling. We use the Converse API from Amazon Bedrock for this, limited to one native tool call per step (boto3, v1.40.15, https://pypi.org/project/boto3/). This call might contain multiple chained and piped bash commands.
- **Stop-Sequence**: We also implement the stop-sequence solution we have described in Sec. 2. The model writes tool calls as XML (`<bash> command </bash>`), the closing tag stops generation, we extract the command using basic string operations, and interpret the content as a bash call. This approach is API-agnostic, transparent, and simple, since the full interaction is visible in plain text.

### 3.3.2 *Prompt Caching.*

In contrast to simple LLM invocations, agentic interactions involve many turns, repeatedly resending the system prompt, and a growing conversation history. Transformer-based LLMs, as exposed through APIs such as Bedrock, are stateless across calls and predict the next token solely from the provided context. There is no persistent session state between invocations. As a consequence, token usage and associated costs grow with conversation length, which is a major factor when evaluating agents.

Prompt caching mechanisms can reduce these costs by marking prompt prefixes that persist across separate calls. When such prefixes are reused, they are billed at reduced rates, while model behavior remains unchanged. Among the selected models, prompt

**Table 1: Classification tasks: Samples N; context shows what each approach receives; simple LLMs only$^{S}$, agent only$^{A}$.**

| Task | Unit | N | Context |
|---|---|---|---|
| Munaiah | Repo | 172 | Dir. listing$^{S}$, README$^{S}$; full repo$^{A}$ |
| **Q:** Is this repository an engineered software project with general-purpose utility to users other than the developers themselves? — project, notproject, unclear | | | |
| Herbold | Line | 212 | Diff hunk w/ marked line; full repo$^{A}$ |
| **Q:** Does the marked line (»>) contribute to the bug fix, or is it something else? — bugfix, test, doc., refactoring, whitespace, unrelated, unclear | | | |
| Härtel | Review | 135 | Comment + PR metadata; full repo$^{A}$ |
| **Q:** Does this PR review comment discuss a potential security defect? — yes, no, unclear | | | |
| Levin | Commit | 129 | Commit message; full repo$^{A}$ |
| **Q:** What is the maintenance intent of this commit? — corrective, adaptive, perfective, unclear | | | |

caching is supported only by Claude 3.7 Sonnet via Bedrock. For the stop-sequence agent, we implement a manual caching scheme along a regular grid. For native tool calling, we rely on the Bedrock prompt caching over the Converse interface of boto3.

## 3.4 Models

We evaluate three models via Amazon Bedrock. All models are examined in simple LLM mode with chain-of-thought; the memorization and no-CoT variants are run with Claude 3.7 Sonnet only. Only Claude 3.7 Sonnet and Mistral Large 3 are used for agents. Mistral is limited to native tool calling.

- **Claude 3.7 Sonnet**: The vendor reports a context window of about 200K tokens. One million input tokens cost 3.0 USD, output tokens 15.0 USD, cached reads 0.3 USD, and cached writes 3.75 USD. In January 2026, only Sonnet 3.7 reliably used the stop-sequence tool calling in dry runs, so the other models have been excluded from this approach.
- **Mistral Large 3**: One million input tokens cost 0.5 USD, and output tokens cost 1.5 USD. The context size is 256K tokens. Prompt caching is not supported.
- **Llama 3.3 70B**: Llama did not show reliable tool-calling in our dry runs, so we excluded it from the agents. The context size is 128K tokens. One million input tokens cost 0.15 USD, output tokens 0.6 USD. Prompt caching is not supported.

Llama is selected as a popular open model, Mistral as a model that balances performance and costs, and Claude 3.7 Sonnet as the most expensive model with strong tool-calling capabilities.

## 4 Tasks

We evaluate the approaches on four MSR classification tasks from prior work, each with a manually labeled ground truth. The tasks vary in granularity, from entire repositories to individual lines. Table 1 summarizes the tasks and the context provided to each approach. We refer to the datasets by the first author of the papers.

## 4.1 Herbold et al.: Tangled Commits

Bug-fixing commits frequently contain unrelated changes, like refactoring, documentation updates, or whitespace cleanup. These *tangled commits* introduce noise in defect prediction studies that assume all lines in a bug-fix commit address the bug [18].

Herbold et al. created a dataset with line-level labels for changes in bug-fixing commits from 28 Java projects. Multiple annotators independently classified each changed line. We use 212 sampled lines from this dataset with 49 bugfix, 79 test, 32 documentation, 8 whitespace, 1 refactoring, and 42 unclear. We interpret lines with no consensus in the original study as unclear lines.

Simple LLMs and agents receive the diff hunk with the target line marked. The agent also has access to the full repository.

## 4.2 Härtel: Security Reviews

Code review is an opportunity to catch security defects before they reach production. Previous work of ours identifies which reviews discuss potential security defects to help prioritize fixes and understand how security issues surface during development [14].

Previous work used active learning with a fine-tuned language model to efficiently mine and classify four million GitHub PR reviews. For this work, we use a balanced sample of 135 reviews from the original data, with 49 discussing a potential security defect, 42 not discussing one, and 44 unclear cases.

The simple LLMs and agents receive the review comment text and PR metadata accessible via the GitHub API, including the full discussion thread. The agent also has access to the full repository.

Our previous study running on four million reviews did not pass the full metadata to the fine-tuned LLM for scalability reasons, which renders the available context different. Differences in context may affect classification, which we discuss in the results (Sec. 6).

## 4.3 Levin et al.: Maintenance Activities

Software maintenance activities can be classified into three categories [26]: corrective, adaptive, and perfective. Automatically classifying commits by such type supports project assessment.

Levin et al. manually labeled 1151 commits from 11 popular open-source projects [23] to eventually train classifiers. We use 129 of the provided manually labeled commits with 65 corrective, 22 adaptive, and 42 perfective.

The simple LLM receives only the commit message. The agent has access to the full repository.

## 4.4 Munaiah et al.: Repository Classification

GitHub hosts many repositories that are homework, assignments, backups, empty projects, or personal experiments. This is a problem for MSR studies that aim to generalize to *engineered software projects* rather than toy or personal projects [29].

Munaiah et al. trained classifiers on extractable repository features such as architecture, community activity, and continuous integration. For validation, two authors independently labeled 200 repositories; only consensus cases are included. We use the 172 repositories still accessible (91 project, 81 not-project).

The simple LLM receives the directory listing and README file appended to the prompt. The agent has access to the full repository.

# 5 Methodology

## 5.1 Sample Selection

For Munaiah, we use 172 accessible repositories of 200 originally labeled. For Herbold, we randomly sample 300 lines with at least three annotators; lines without consensus (≥75% agreement) are labeled unclear; 212 remain after exclusions. For Härtel, we draw a stratified sample of 50 per class; 135 remain. For Levin, we randomly sample 200 commits; 129 remain. We exclude 108 observations across 9 of 312 repositories due to excessive repository size (above 1000 MB). Herbold misses cases where we did not manage to verify the correct line and label association from the original dataset. Exclusions and filters apply to all approaches to ensure a fair comparison.

## 5.2 Prompts

Agentic and non-agentic approaches get closely matched prompts to ensure a fair comparison. We describe the details of prompt construction in Sec. 3 and Table 1. Full prompts can also be found in the replication package, where they are stored with the trajectories for each experiment. For validation, we sampled 20 experiments to confirm correct prompt construction.

## 5.3 Execution

We run experiments across four tasks and eight approaches on 648 samples per approach (after size exclusions, see Sec. 5.1). This yields 5184 attempts, of which 212 are systematic (memorization does not apply to Herbold lines) and 29 are genuine errors (mostly context overflow on simple LLMs; zero for agents), leaving 4943 valid classifications. Models are accessed via Amazon Bedrock to avoid infrastructure differences. We set the temperature to 1.0 and the maximum output tokens to 4096 for all approaches except the no-CoT variant (16 tokens). Top-p and other sampling parameters use Bedrock defaults. The experiments were executed in April 2026.

## 5.4 Measurements

We measure classification accuracy, resource usage (tokens, time, cost), exploration steps, and tool-use behavior to answer our research questions RQ1 and RQ2. Errors of the approaches are recorded and classified. Cost is estimated from the tokens using the prices of AWS US East (N. Virginia) as of January 2026. Such costs are a reasonable approximation of the resource usage behind the APIs.

## 5.5 Disagreement Analysis

To understand why approaches disagree with the ground truth and answer RQ3, we identify all cases where at least two non-experimental approaches disagree with the ground-truth, yielding 254 such cases across the tasks. We manually inspect a stratified sample of 100 cases and assign one of four diagnoses from Table 2.

**Table 2: Taxonomy for diagnosing disagreements**

| Diagnosis | Definition |
|---|---|
| Update label | Evidence supports a different label than the original ground truth; the ground truth should be revised. |
| Keep label | The original ground truth holds; the disagreeing approaches misclassify. |
| Specification | The task definition is ambiguous; both (multiple) labels are defensible interpretations. |
| Unresolvable | Insufficient information to determine the correct label. |

## 5.6 Hierarchical Accuracy Model

We use a hierarchical accuracy model to estimate approach accuracy with credible intervals. The model accounts for per-sample difficulty, yielding uncertainty that reflects the paired design where all approaches classify the same samples. For a single task, the model is specified as follows:

$$y_{s,a} \sim \text{Bernoulli}(\text{logit}^{-1}(\alpha_a + \theta_s)) \tag{1}$$

$$\theta_s \sim \text{Normal}(0, \sigma_\theta) \tag{2}$$

$$\alpha_a \sim \text{Normal}(0, 2) \tag{3}$$

$$\sigma_\theta \sim \text{HalfNormal}(2) \tag{4}$$

where $y_{s,a} \in \{0, 1\}$ indicates whether approach $a$ correctly classified sample $s$, $\alpha_a$ captures approach ability, and $\theta_s$ captures sample difficulty. Errors and timeouts can be excluded from the likelihood by setting those values to NaN. We fit one model per task using Stan [37]. Approach accuracy is derived as the posterior predictive mean $\text{acc}_a = S^{-1} \sum_s \text{logit}^{-1}(\alpha_a + \theta_s)$. Pairwise accuracy differences $\text{acc}_a - \text{acc}_b$ are computed directly from posterior samples, yielding credible intervals on differences. The online material shows validation of the model by fitting it to simulated data (see [17, 28]).

# 6 Results

## 6.1 Resource Usage

*Tokens.* Fig. 2 shows the token composition per experiment. Simple LLM approaches, except for memorization, use around 5–8K input tokens independent of the underlying model. Numbers are different because context overflow errors are excluded from the mean (and from cost aggregates), removing the highest token counts from the aggregation.

The occurrence of context overflow errors depends on the model's context limit and the specific prompt size. Llama has the smallest context size limit of 128K tokens, Sonnet lies around 200K tokens, and Mistral has the largest with 256K tokens. Simple Llama runs into 9 context-overflow errors, simple Sonnet into 7, simple Sonnet (no-CoT) into 6, and simple Mistral into 3. Simple Sonnet (memorization) runs into 0 errors because the approach does not get context that could exceed the limit.

Simple LLMs only output a reasoning trace and answer, no exploration steps or tool calls, so output tokens are low. Llama outputs on average 101 tokens, Mistral 217, simple Sonnet 189, simple Sonnet (memorization) 138, and simple Sonnet (no-CoT) 4 tokens.

Agentic approaches show an entirely different picture because tokens accumulate over multiple steps, and thereby numbers are much higher. Overall, Agent Mistral uses 8.5K fresh input tokens on average with 607 output tokens for reasoning traces and tool calls. Agent Sonnet (stopseq) gets 10.7K fresh input tokens on average plus 18.2K cache-read tokens, 4.9K cache-write tokens, with 720 output tokens for reasoning traces and tool calls. Agent Sonnet (native) uses the most tokens overall with 14.2K fresh input tokens

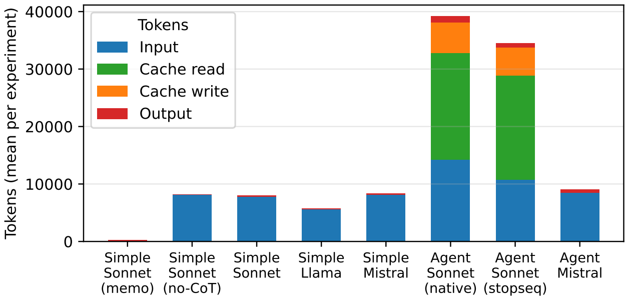

**Figure 2: Mean token usage by approach over all experiments.**

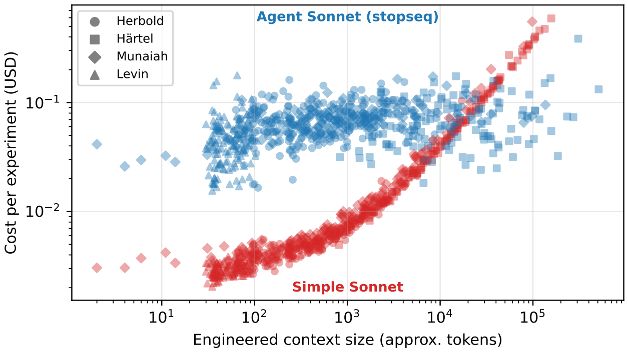

**Figure 3: Cost per experiment vs. engineered context size**

on average plus 18.5K cache-read tokens, 5.3K cache-write tokens, and 1.1K output tokens for reasoning traces and tool calls.

We see a significant amount of cached read tokens for both Sonnet approaches. Mistral does not support caching. Agents run into no context-window overflows, which is a major advantage in the robustness of agentic approaches. The simplistic stop-sequence approach appears to be more efficient in token use than native tool calling. This is likely because the stop-sequence approach cuts generation after the tool call is determined, which accelerates the feedback loop between tool output and next LLM generation.

*Cost.* We next analyze the cost per experiment because it correlates with the effective resource usage of each approach. Due to the different pricing schemes of the models, cost is not a one-to-one mapping from tokens. Provider costs serve as a practical proxy for resource use rather than a scientific end in themselves.

Fig. 3 reveals a fundamental difference in how the two paradigms scale. The engineered context size on the x-axis depicts the maximal input for an experiment, including all possible artifacts we feed into a simple approach. Thereby, the engineered context size almost perfectly correlates with the cost for a simple approach shown on the y-axis. Large READMEs, metadata dumps, or diffs directly cause the prompt to grow and increase the cost (red). Agent costs, by contrast, show near-zero correlation (blue).

Fig. 4 shows a corresponding cost distribution of all approaches. Simple approaches are dominated by outliers of exceptionally large inputs, but they are still the cheapest when averaged over all of our experiments. Simple Llama needs $0.0009/experiment, simple Mistral $0.0044/experiment, and simple Sonnet $0.026/experiment.

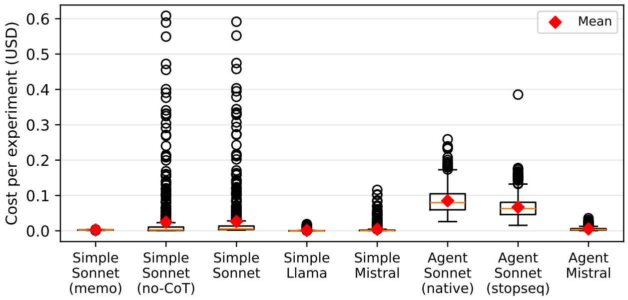

**Figure 4: Mean cost distribution per experiment by approach: Diamond markers (mean); boxes (median and quartiles).**

Agents incur higher costs on average, but by small factors when compared to the corresponding simple approaches. Agent Sonnet (native) needs an average of $0.085/experiment (factor 3.2), Agent Sonnet (stopseq) $0.067/experiment (factor 2.5), and Agent Mistral $0.0051/experiment (factor 1.2). Especially the low increase of costs for Mistral is surprising. Moderate cost increases can be explained by the selective exploration of agents that keeps the overall context size manageable. For experiments where the engineered context exceeds approximately 30K tokens, Agent Sonnet (stopseq) becomes more cost-effective than Simple Sonnet, as shown in Fig. 3. Hence, with a different distribution of input sizes, agents might even be cheaper on average than simple approaches.

Caching helps reduce input costs; cached reads are billed at 10% of the regular input price. For Sonnet native, we see 35.1% cost savings through caching, and for Sonnet stopseq, 40.4% savings. Our caching implementation has not been optimized, and we expect bigger savings with more advanced caching strategies.

*Time.* Simple LLM approaches complete fast: simple Llama runs in 1.9s, simple Mistral in 6.0s, simple Sonnet in 5.1s, simple Sonnet (no-CoT) in 1.5s, and simple Sonnet (memorization) in 3.7s per experiment. Agents naturally take longer due to sequential tool execution: Agent Sonnet (native) takes on average 29.3s, Agent Sonnet (stopseq) 25.4s, and Agent Mistral 21.3s per experiment. Overall, we consider time as a secondary argument for our use case, since such classification tasks can be run in parallel and offline.

## 6.2 Tool Use

Agents explore repositories through bash commands. Overall, Agent Sonnet (native) shows an average of 14.1 commands per experiment, with an exploration depth of 10.4 steps; Agent Sonnet (stopseq) needs 13.9 commands on 10.5 steps; Agent Mistral needs 6.1 commands on 5.7 steps. Commands higher than the step count indicate composition by piping or chaining in the single bash invocation the agents have per step.

Fig. 5 shows a breakdown of tool usage for agents. The most frequent commands are file inspection and ways to limit reads (`cat`, `head`, `grep`), navigation (`ls`, `find`), and Git operations (`git show`, `git checkout`). Agents compose commands differently. Sonnet pipes commands (`|`) preferentially, while Mistral also uses chaining

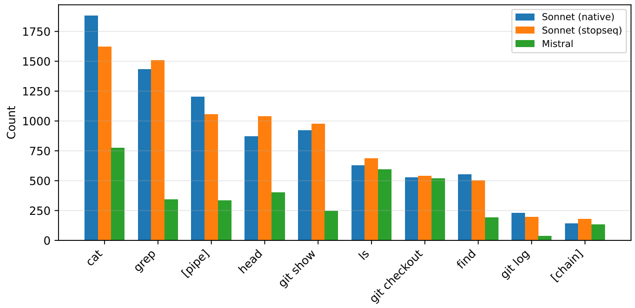

**Figure 5: Tool usage by agent approach (top 10 commands).**

**Table 3: Error overview by approach**

| | Non-fatal | Fatal | |
|---|---|---|---|
| | Tool Errors | Invalid Category | Context Overflow |
| Simple Sonnet (memo) | – | 3 (0.5%) | – |
| Simple Sonnet (no-CoT) | – | – | 6 (0.9%) |
| Simple Sonnet | – | – | 7 (1%) |
| Simple Llama | – | 1 (0.2%) | 9 (1%) |
| Simple Mistral | – | – | 3 (0.5%) |
| Agent Sonnet (native) | 1.7% | – | – |
| Agent Sonnet (stopseq) | 2.2% | – | – |
| Agent Mistral | 1.9% | – | – |

a lot (`&&`, `;`). We consider pipes as a more advanced way of composing commands, like using `grep` to filter `ls` output directly. This suggests the exploration of Sonnet is more sophisticated.

Table 3 summarizes errors across all approaches. Tool errors are classified by exit codes and command outputs; fatal errors with context overflow and invalid category invalidate the experiment and may be counted as wrong classifications depending on the analysis. Tool-calling errors on the bash level are non-fatal, as the agents can recover from them in the next step. All agents show comparable tool call error rates (1.7%, 2.2%, 1.9% for Sonnet native, stopseq, and Mistral). Invalid category outputs are rare overall, occurring only for simple Llama and simple Sonnet (memorization) with a total of 4 cases across all experiments.

## 6.3 Exploration Depth

Fig. 6 shows the distribution of steps that the agents need for each task. We see there is variation over tasks, but overall, Sonnet approaches consistently use more steps than Mistral, with 10.4, 10.5, and 5.7 steps on average for Sonnet native, stopseq, and Mistral.

Munaiah et al. (repository classification) and Härtel (security defects in reviews) require more exploration steps while Levin (commit classification) and Herbold (tangled commits) require fewer steps. We assume that commits are more self-contained, while repository and review classification require more context exploration. No agent hits the 50-step limit.

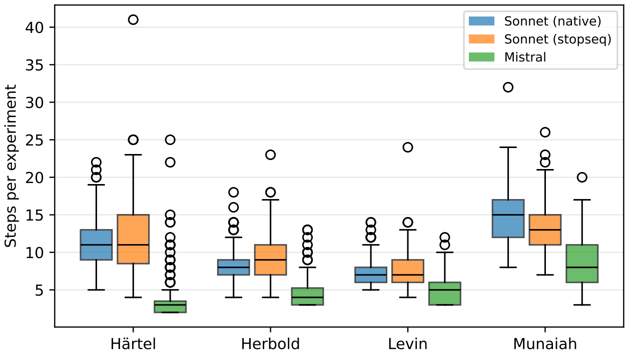

**Figure 6: Boxplot of steps per experiment by task and agent**

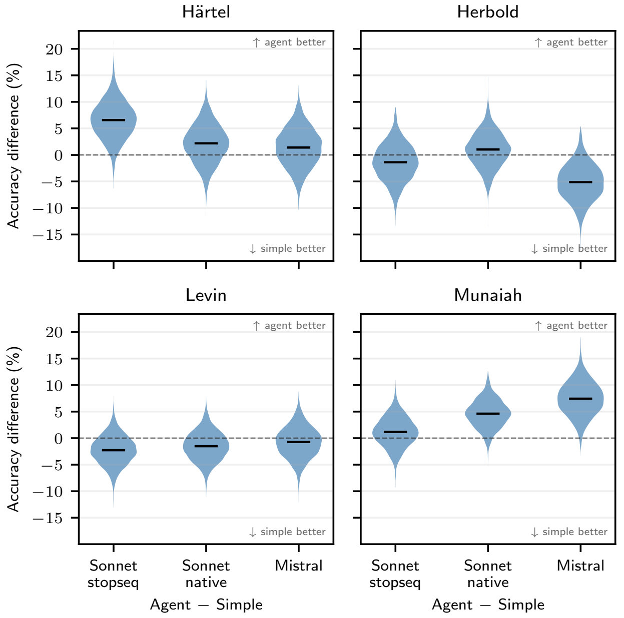

**Figure 7: Posterior of the accuracy difference as violin with fatal errors counted as incorrect classification.**

## 6.4 Classification Accuracy

We report on absolute accuracy specific to tasks in Table 4, 5, 6, and 7. Statistical analysis of the accuracy difference between agents and corresponding simple approaches can be found in Fig. 7. For Agent Sonnet native and stopseq, we use Simple Sonnet as the reference. For Agent Mistral, we use Simple Mistral as the reference.

*6.4.1 Herbold.* Table 4 shows that tangled commit line classification appears to be the hardest task for all approaches, with accuracy values between 46.2% and 56.1%. No approach runs into fatal errors, so there is no separate accuracy that needs to be reported. Testing no context with Sonnet (memorization) is excluded because we did not see a reasonable way to identify a specific line without repeating the actual diff in the context.

All approaches are better than random prediction with 14.3% accuracy or majority prediction with 37.3% accuracy. Overall, Mistral

**Table 4: Herbold (tangled commits): 95% credible intervals; *Acc.* excludes or includes errors; invalid categories (Inv. Cat.) and context overflow (Ctx. Ovfl)**

| **Approach** | **Acc.** | **Acc.** err=fail | Inv. Cat. | Ctx. Ovfl |
|---|---|---|---|---|
| Simple Sonnet (no-CoT) | 57.5% [53,62] | = | – | – |
| Simple Sonnet | 55.2% [51,60] | = | – | – |
| Simple Llama | 46.2% [42,51] | = | – | – |
| Simple Mistral | 54.7% [50,59] | = | – | – |
| Agent Sonnet (native) | 56.1% [51,61] | = | – | – |
| Agent Sonnet (stopseq) | 53.8% [49,58] | = | – | – |
| Agent Mistral | 49.5% [45,54] | = | – | – |
| Random prediction† | 14.3% | 14.3% | – | – |
| Majority prediction‡ | 37.3% | 37.3% | – | – |

† Uniform random ($K = 7$). ‡ Always predicts the most frequent label.

**Table 5: Härtel (security defects, columns as in Table 4).**

| **Approach** | **Acc.** | **Acc.** err=fail | Inv. Cat. | Ctx. Ovfl |
|---|---|---|---|---|
| Simple Sonnet (memo) | 31.9% [27,38] | = | – | – |
| Simple Sonnet (no-CoT) | 59.2% [54,64] | 57.0% [52,62] | – | 5 (4%) |
| Simple Sonnet | 59.7% [55,65] | 57.0% [52,62] | – | 6 (4%) |
| Simple Llama | 65.4% [62,71] | 61.5% [56,66] | 1 (1%) | 7 (5%) |
| Simple Mistral | 58.3% [54,64] | 57.0% [52,62] | – | 3 (2%) |
| Agent Sonnet (native) | 59.3% [54,64] | = | – | – |
| Agent Sonnet (stopseq) | 63.7% [59,68] | = | – | – |
| Agent Mistral | 58.5% [54,63] | = | – | – |
| Random prediction† | 33.3% | 33.3% | – | – |
| Majority prediction‡ | 36.3% | 36.3% | – | – |

**Table 6: Levin (maintenance activity, columns as in Table 4)**

| **Approach** | **Acc.** | **Acc.** err=fail | Inv. Cat. | Ctx. Ovfl |
|---|---|---|---|---|
| Simple Sonnet (memo) | 45.2% [39,52] | 44.2% [38,51] | 3 (2%) | – |
| Simple Sonnet (no-CoT) | 87.6% [82,91] | = | – | – |
| Simple Sonnet | 91.5% [87,94] | = | – | – |
| Simple Llama | 93.8% [89,96] | = | – | – |
| Simple Mistral | 89.9% [85,93] | = | – | – |
| Agent Sonnet (native) | 89.9% [85,93] | = | – | – |
| Agent Sonnet (stopseq) | 89.1% [84,92] | = | – | – |
| Agent Mistral | 89.1% [84,92] | = | – | – |
| Random prediction† | 25.0% | 25.0% | – | – |
| Majority prediction‡ | 50.4% | 50.4% | – | – |

**Table 7: Munaiah (repository, columns as in Table 4)**

| **Approach** | **Acc.** | **Acc.** err=fail | Inv. Cat. | Ctx. Ovfl |
|---|---|---|---|---|
| Simple Sonnet (memo) | 82.0% [78,85] | = | – | – |
| Simple Sonnet (no-CoT) | 73.1% [68,77] | 72.7% [68,77] | – | 1 (1%) |
| Simple Sonnet | 83.6% [79,87] | 83.1% [79,86] | – | 1 (1%) |
| Simple Llama | 65.9% [60,70] | 65.1% [60,70] | – | 2 (1%) |
| Simple Mistral | 70.9% [66,75] | = | – | – |
| Agent Sonnet (native) | 87.8% [83,91] | = | – | – |
| Agent Sonnet (stopseq) | 84.3% [80,88] | = | – | – |
| Agent Mistral | 78.5% [74,82] | = | – | – |
| Random prediction† | 33.3% | 33.3% | – | – |
| Majority prediction‡ | 52.9% | 52.9% | – | – |

and Llama approaches are slightly worse than Sonnet approaches. We explore the corresponding pairs of agents and simple counterparts in Fig. 7. Using a region of practical equivalence [21, 22] of ±5 accuracy points in difference, both Sonnet agents are practically equivalent to Simple Sonnet (83% and 86% probability). Agent Mistral is the only case where the posterior leans toward a meaningful deficit (51% probability of being practically worse than Simple Mistral), though the evidence is weak.

Since no approach runs into errors on this task, agents cannot benefit from their robustness advantage here, which leaves a pure accuracy comparison. Being practically equal to simple approaches, despite having to find context themselves, is the result for this task.

#### 6.4.2 *Härtel.*

Table 5 shows that our previous work on security defects is another harder task for all approaches. We see accuracy in a range between 58% and 66% without counting errors as wrong classifications. If context overflow errors are counted as wrong classifications, the accuracy of all simple approaches drops below the agents. What is surprising is that Simple Llama comes out best when excluding errors, almost coincidentally: it is the only approach that predicts ‘unclear’ with a non-negligible rate, which happens to be a good strategy for this task because roughly a third of the samples are indeed labeled as unclear.

Our sanity check using Sonnet (memorization) shows an accuracy of 31.9%, which is below the accuracy of a random prediction (33.3%) or a majority prediction (36.3%). We conclude that it is unlikely that the correct labels are in the training data.

Low performance might be caused by the original labels in [14] being produced without the pull request metadata. The resulting difference in available context, where our experiments have more context, lets approaches avoid ‘unclear’ and pick a category. We examine this in more detail in the disagreement analysis in Sec. 6.6.

The comparison of agents against their simple counterparts in Fig. 7 shows that the probability that any agent performs meaningfully worse is low (0%, 2%, 4% for stopseq, native, and Mistral). Agent Sonnet (stopseq) even has a 67% probability of being meaningfully better than Simple Sonnet, which we attribute to context overflow errors pulling Simple Sonnet accuracy down.

#### 6.4.3 *Levin.*

Table 6 shows numbers for maintenance activity classification. The original authors report 76% accuracy for a model trained on commit messages. Our approaches are not trained but reach accuracies between 87.6% and 93.8%, likely because LLMs encode enough maintenance-activity knowledge to classify from the commit context directly. Our sanity check with Sonnet (memorization) shows 45.2% accuracy, below the majority prediction with 50.4% accuracy but above a random prediction with 25.0% accuracy. This means the class distribution may be somewhat reflected in the training data, but the particular commit misses.

Comparing agents against their simple counterparts in Fig. 7, all three pairs are practically equivalent with 84%, 90%, and 91% probability of being within ±5 accuracy points. As with Herbold, no approach runs into errors on this task, so there is no robustness advantage for agents to leverage.

#### 6.4.4 *Munaiah.*

Table 7 shows that repository classification for engineered software has the widest accuracy spread across approaches, ranging from 65.1% to 87.8%. Our sanity check with Sonnet (memorization) shows a striking 82.0% accuracy, far above the majority prediction with 52.9% accuracy and the random prediction

**Table 8: Confusion matrix for 'unclear' handling.**

| | Ground Truth | | | |
|---|---|---|---|---|
| | Clear | | Unclear | |
| **Approach** (concludes →) | Clear | Unclear | Clear | Unclear |
| Simple Sonnet (memo) | 389 | 0 | 44 | 0 |
| Simple Sonnet (no-CoT) | 549 | 9 | 83 | 1 |
| Simple Sonnet | 551 | 6 | 84 | 0 |
| Simple Llama | 512 | 42 | 67 | 17 |
| Simple Mistral | 550 | 10 | 83 | 2 |
| Agent Sonnet (native) | 562 | 0 | 86 | 0 |
| Agent Sonnet (stopseq) | 562 | 0 | 85 | 1 |
| Agent Mistral | 560 | 2 | 85 | 1 |

with 33.3% accuracy. These numbers suggest a partial training-data overlap for this task. We cannot rule out that the model did see the labels, but we assume it is more likely that repositories to be classified are present in the training data. The trained knowledge on the repository might be sufficient to derive the label just knowing the GitHub repository name, which is the only input to memorization. Most surprising is that memorization beats Simple Sonnet (no-CoT), Simple Llama, Simple Mistral, and Agent Mistral.

Comparing agents against their simple counterparts in Fig. 7, Agent Mistral shows the clearest lean toward being meaningfully better (78% probability), while Agent Sonnet (native) has a weaker 44% probability of being meaningfully better. Only the stopseq pair is conclusively within the region of practical equivalence (90% probability of being within ±5 accuracy points). This is the task with the strongest signal for an agent advantage, primarily driven by Agent Mistral.

## 6.5 Uncertainty Handling

We show a detailed breakdown of which model predicts unclear regarding whether the ground truth is clear or unclear in Tab. 8. The accuracy of correctly matching unclear labels of the baseline is vanishingly low. In total, models avoid classifying as unclear. Simple Llama uses 9% unclear labels, simple Mistral 2%, and simple Sonnet 1%. We see that agents avoid labeling as unclear. Our assumption is that broader access gives confidence to pick a category and under-express uncertainty. Overall, these numbers conform to the understanding that LLMs still struggle with uncertainty [20, 24].

## 6.6 Disagreement Analysis

We have manually diagnosed 100 of 254 disagreement cases where at least two approaches (excluding memorization and no-CoT) disagree with the ground truth. Table 9 summarizes by task and diagnosis. We discuss our findings in the following text and make the data available online to provide a starting point for refining the original labeling guidelines we borrowed from the studies.

### 6.6.1 *Herbold.*

Disagreements predominantly trace to overlapping categories in the task specification. A frequent ambiguity is between whitespace and refactoring, where formatting adjustments can plausibly fall into either category. Similarly, when a commit refactors documentation or test code, the correct label depends on whether the *type of change* (refactoring) or the *type of artifact* (documentation, test) takes precedence — the guidelines do not resolve this. A related pattern is test code that verifies the bugfix: is it part of the fix or a test? Some *unclear* ground-truth labels could be updated with additional context.

**Table 9: Manual diagnosis of disagreements between approach and the ground truth with number of labeled cases, disagreeing cases, and total cases per task.**

| | Diagnosis (labeled) | | | | Cases | | |
|---|---|---|---|---|---|---|---|
| **Task** | Update | Keep | Spec. | Unres. | Lab. | Disag. | Total |
| Herbold | 7 | 6 | 18 | 3 | 34 | 128 | 212 |
| Härtel | 20 | 1 | 6 | 1 | 28 | 59 | 135 |
| Munaiah | 4 | 9 | 9 | 1 | 23 | 52 | 172 |
| Levin | 6 | 1 | 7 | 1 | 15 | 15 | 129 |
| **Total** | 37 | 17 | 40 | 6 | 100 | 254 | 648 |

### 6.6.2 *Härtel.*

The majority of our analysis reflects that our previous labeling was done without pull-request metadata, such as the surrounding comments. Additional context allows updating several *unclear* labels. We further diagnosed specification issues with the phrase: 'discusses a potential security defect'. This is unambiguous only if comments explicitly flag a security defect or are completely unrelated. Comments that touch on security mechanisms without pointing to a defect split classifications. This is a specification issue, not an approach failure.

### 6.6.3 *Levin.*

Levin shows low disagreements because of high accuracy. Maintenance activity categories appear to be well-defined. If approaches disagree, three patterns emerge. First, agentic approaches can recover context that the commit message misses. In one of the cases, a commit message reads "remove spam … and fix a list.size == 0," yet the actual diff only replaces `list.size()==0` with `list.isEmpty()` and logging. This is a perfective cleanup and different from ground truth classifying as a corrective fix. Second, approaches fall back on a repository's own classification taxonomy, such as those present in the change logs. This overrides a potentially better classification based on the actual change by the commit. Third, test-only commits split approaches because the guidelines do not draw a clear line between testing existing and new features.

### 6.6.4 *Munaiah.*

Deciding between engineered projects and non-engineered projects leaves a wide gray zone. Missing thresholds on the amount of necessary documentation, single-event applications, deprecated projects, and projects that have an architecture on the surface cause disagreement. Additional ambiguities arise for non-English naming conventions.

## 7 Threats and Limitations

*Data Leakage.* Leaked labels to LLM training might threaten our measurements. We test for this by a memorization baseline, where three of four tasks show negligible effects. Moreover, our comparison between agentic and simple approaches is backed by the same LLMs, which factors out such effects.

*Ground Truth Quality.* Biases and errors in the original labels can threaten our accuracy measurements. We therefore manually diagnosed 100 disagreement cases, making quality issues explicit.

*Setup Bias.* Specific prompt choices can influence behavior and threaten generalizability. We standardize prompts across approaches and tasks, mirroring the original labeling guidelines.

*Limitations of Sampling and Filtering.* Our findings are limited by the filters we apply to repository accessibility and size limits. This skews the sample toward projects that are smaller, active, or still public and away from larger or archived repositories.

*Limited Generalizability to Newer Models.* We rely on models not publicly available, which implies that a part of our experimental pipeline is out of our control. Different and newer releases may provide different results in the future. Our findings are limited to the model versions we report on. To compensate for this, we make the full trajectories available for a partial reproduction.

*Limited Feature Comparisons.* We limit our comparison to a range of basic agentic features against a simple LLM baseline across four tasks. More advanced strategies, such as summarization strategies, agentic planning, truncation policies, higher step limits, and internet access for the agents, can clearly change our results.

*Limited Baseline Comparisons.* We limit approaches to not require task-specific engineering, except for the simple LLM baselines. This disqualifies RAG approaches that rely on manually engineered retrieval strategies or any learning-based approaches.

## 8 Related Work

### 8.1 Classification and Labeling in SE

Authors of [43] propose an active learning framework for vulnerability classification and labeling called HARMLESS. It is a good example of pre-LLM vulnerability classification using a support vector machine. Pre-LLM examples of explicit engineering include rule-based repository mining [16] and feature-based API clustering [15]. Authors of [3] focus on classifying reviews of npm packages. Agentic classification techniques could potentially be applied here too, for a more profound analysis of contextual factors within the packages. In [41], code reviews from the OpenStack and Qt communities are examined. One conclusion is the benefit of combining manual review with automated techniques. In our task on security reviews, we take a step in this direction by examining how the review process can be supported by LLM agents. In [33], misconfigurations are examined. Misconfigurations could be another interesting target for agentic classification. Smells in infrastructure-as-code are examined in [32]. Plate et al. [31] discuss that current decision-making based on natural-language vulnerability descriptions and expert knowledge is difficult, time-consuming, and error-prone.

The approach we examine through contextualization by an agentic framework is capable of working with arbitrary context, including natural-language descriptions, and all sorts of output that git and Bash commands can provide. The agentic trajectories can also support semi-automated decision-making.

### 8.2 LLMs and Agents in SE

Commit messages that lack important information, potentially resolved by LLMs, are examined in [10]. Assumptions on the context are related to ours in that knowing the repository history could significantly improve the quality of commit messages. REPOAUDIT [12] combines static analysis with an LLM to direct checks to suspicious code locations. The agents we examine have flexible access to the commit history, using `git log`, `git diff`, and other commands, allowing access beyond what is described before. Recent work on the usage of LLMs for code review generation is presented in [8]. Such work shows the current standard: code is passed as context to the LLM. Agentic techniques that allow the LLM to flexibly retrieve code on their own are not yet considered. In [40], an empirical study on retrieval-augmented generation (RAG) for code generation is presented. This study makes clear that RAG assumes 'similar code snippets collected by a retrieval process' are the context. We examine retrieval by agents that do not need a manually engineered *similarity function* but a catalog of standard Bash commands to access a repository. In [1, 2], the authors use RAG and knowledge graphs to answer repository-related questions by adding retrieved files to the prompt; they report that poor retrieval was the main cause of failures. We see that issues with the similarity definition are common, and we bypass them by using standard Bash commands to access the repository.

The first agentic framework for executing triage in software engineering is presented in [42]. Cloud root-cause analysis with tool-augmented LLMs is explored in [39] where agents access monitoring data and logs autonomously to identify the root cause of cloud incidents. Such approaches are close to what we examine but have a different focus. SHERPA is described in [6] building on the idea of abstract state machines that sit on top of the LLM invocations. Such ideas stem from seeing reasoning as a sequence of steps, a core for running an agent.

Moreover, recent MSR work studies agents themselves as subjects, analyzing traces they leave in repositories like `AGENTS.md` or `CLAUDE.md` [9, 27, 35], or differences in agent-generated code, such as in tests [19]. We take the complementary perspective: using agents to mine artifacts in software repositories.

## 9 Conclusion

We evaluated agentic repository mining, where LLMs classify repository artifacts by dynamically exploring repository context through standard bash commands. Across four MSR tasks and eight approach configurations, agents achieve competitive accuracy despite retrieving their own context, while simple LLMs receive pre-engineered input. The primary advantage is robustness: agents avoid context-window overflows and scale independently of artifact size. A manual diagnosis of 100 cases where approaches disagree with the ground truth reveals specification ambiguities and labels produced under limited context, suggesting that accuracy against such ground truth may underestimate approaches with broader context access.

## Acknowledgments

This work is funded by EU grant No. 101120393 (Sec4AI4Sec). Anthropic's Claude Code (Opus 4.6) assisted coding and writing.

## References

[1] Samuel Abedu, Ahmad Abdellatif, and Emad Shihab. 2024. LLM-Based Chatbots for Mining Software Repositories: Challenges and Opportunities. In *EASE*. ACM, 201–210.

[2] Samuel Abedu, Laurine Menneron, SayedHassan Khatoonabadi, and Emad Shihab. 2025. RepoChat: An LLM-Powered Chatbot for GitHub Repository Question-Answering. In *MSR*. IEEE, 255–259.

[3] Mahmoud Alfadel, Nicholas Alexandre Nagy, Diego Elias Costa, Rabe Abdalkareem, and Emad Shihab. 2023. Empirical analysis of security-related code reviews in npm packages. *J. Syst. Softw.* 203 (2023), 111752.

[4] Gábor Antal, Bence Bogenfürst, Rudolf Ferenc, and Péter Hegedüs. 2025. Identifying Helpful Context for LLM-based Vulnerability Repair: A Preliminary Study. In *EASE*. ACM, 696–700.

[5] Anthropic. 2026. An update on recent Claude Code quality reports. https://www.anthropic.com/engineering/april-23-postmortem Accessed: 2026-04-24.

[6] Boqi Chen, Kua Chen, José Antonio Hernández López, Gunter Mussbacher, Dániel Varró, and Amir Feizpour. 2025. SHERPA: A Model-Driven Framework for Large Language Model Execution. *CoRR* abs/2509.00272 (2025).

[7] Zhenpeng Chen, Chong Wang, Weisong Sun, Guang Yang, Xuanzhe Liu, Jie M. Zhang, and Yang Liu. 2025. Promptware Engineering: Software Engineering for LLM Prompt Development. *CoRR* abs/2503.02400 (2025).

[8] Umut Cihan, Arda İçöz, Vahid Haratian, and Eray Tüzün. 2025. Evaluating Large Language Models for Code Review. *CoRR* abs/2505.20206 (2025).

[9] Shamse Tasnim Cynthia, Joy Krishan Das, and Banani Roy. 2026. Are We All Using Agents the Same Way? An Empirical Study of Core and Peripheral Developers Use of Coding Agents. In *MSR*. IEEE.

[10] Aleksandra Eliseeva, Yaroslav Sokolov, Egor Bogomolov, Yaroslav Golubev, Danny Dig, and Timofey Bryksin. 2023. From Commit Message Generation to History-Aware Commit Message Completion. In *ASE*. IEEE, 723–735.

[11] Xiuting Ge, Chunrong Fang, Meiyuan Qian, Yu Ge, and Mingshuang Qing. 2022. Locality-based security bug report identification via active learning. *Inf. Softw. Technol.* 147 (2022), 106899.

[12] Jinyao Guo, Chengpeng Wang, Xiangzhe Xu, Zian Su, and Xiangyu Zhang. 2025. RepoAudit: An Autonomous LLM-Agent for Repository-Level Code Auditing. *CoRR* abs/2501.18160 (2025).

[13] Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V. Chawla, Olaf Wiest, and Xiangliang Zhang. 2024. Large Language Model Based Multi-agents: A Survey of Progress and Challenges. In *IJCAI*. ijcai.org, 8048–8057.

[14] Johannes Härtel. 2025. Improved Labeling of Security Defects in Code Review by Active Learning with LLMs. In *EASE*. ACM, 1014–1023.

[15] Johannes Härtel, Hakan Aksu, and Ralf Lämmel. 2018. Classification of APIs by hierarchical clustering. In *ICPC*. ACM, 233–243.

[16] Johannes Härtel, Marcel Heinz, and Ralf Lämmel. 2018. EMF Patterns of Usage on GitHub. In *ECMFA (Lecture Notes in Computer Science)*. Springer, 216–234.

[17] Johannes Härtel and Ralf Lämmel. 2023. Operationalizing validity of empirical software engineering studies. *Empir. Softw. Eng.* 28, 6 (2023), 153.

[18] Steffen Herbold, Alexander Trautsch, Benjamin Ledel, Alireza Aghamohammadi, Taher Ahmed Ghaleb, Kuljit Kaur Chahal, Tim Bossenmaier, Bhaveet Nagaria, Philip Makedonski, Matin Nili Ahmadabadi, Kristóf Szabados, Helge Spieker, Matej Madeja, Nathaniel Hoy, Valentina Lenarduzzi, Shangwen Wang, Gema Rodríguez-Pérez, Ricardo Colomo-Palacios, Roberto Verdecchia, Paramvir Singh, Yihao Qin, Debasish Chakroborti, Willard Davis, Vijay Walunj, Hongjun Wu, Diego Marcilio, Omar Alam, Abdullah Aldaeej, Idan Amit, Burak Turhan, Simon Eismann, Anna-Katharina Wickert, Ivano Malavolta, Matús Sulír, Fatemeh H. Fard, Austin Z. Henley, Stratos Kourtzanidis, Eray Tuzun, Christoph Treude, Simin Maleki Shamasbi, Ivan Pashchenko, Marvin Wyrich, James Davis, Alexander Serebrenik, Ella Albrecht, Ethem Utku Aktas, Daniel Strüber, and Johannes Erbel. 2022. A fine-grained data set and analysis of tangling in bug fixing commits. *Empir. Softw. Eng.* 27, 6 (2022), 125.

[19] André C. Hora and Romain Robbes. 2026. Are Coding Agents Generating Over-Mocked Tests? An Empirical Study. In *MSR*. IEEE.

[20] Adam Tauman Kalai, Ofir Nachum, Santosh S. Vempala, and Edwin Zhang. 2025. Why Language Models Hallucinate. *CoRR* abs/2509.04664 (2025).

[21] John Kruschke. 2014. Doing Bayesian data analysis: A tutorial with R, JAGS, and Stan. (2014).

[22] John K Kruschke. 2018. Rejecting or accepting parameter values in Bayesian estimation. *Advances in methods and practices in psychological science* 1, 2 (2018), 270–280.

[23] Stanislav Levin and Amiram Yehudai. 2017. Boosting Automatic Commit Classification Into Maintenance Activities By Utilizing Source Code Changes. In *PROMISE*. ACM, 97–106.

[24] Stephanie Lin, Jacob Hilton, and Owain Evans. 2022. Teaching Models to Express Their Uncertainty in Words. *Trans. Mach. Learn. Res.* 2022 (2022).

[25] Tim Menzies, Jeremy Greenwald, and Art Frank. 2007. Data Mining Static Code Attributes to Learn Defect Predictors. *IEEE Trans. Software Eng.* 33, 1 (2007), 2–13.

[26] Audris Mockus and Lawrence G. Votta. 2000. Identifying Reasons for Software Changes using Historic Databases. In *ICSM*. IEEE Computer Society, 120–130.

[27] Seyedmoein Mohsenimofidi, Matthias Galster, Christoph Treude, and Sebastian Baltes. 2026. Context Engineering for AI Agents in Open-Source Software. In *MSR*. IEEE.

[28] Tim P Morris, Ian R White, and Michael J Crowther. 2019. Using simulation studies to evaluate statistical methods. *Statistics in medicine* 38, 11 (2019), 2074–2102.

[29] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. 2017. Curating GitHub for engineered software projects. *Empir. Softw. Eng.* 22, 6 (2017), 3219–3253.

[30] Md. Nahidul Islam Opu, Shaowei Wang, and Shaiful Chowdhury. 2025. LLM-Based Detection of Tangled Code Changes for Higher-Quality Method-Level Bug Datasets. *CoRR* abs/2505.08263 (2025).

[31] Henrik Plate, Serena Elisa Ponta, and Antonino Sabetta. 2015. Impact assessment for vulnerabilities in open-source software libraries. In *ICSME*. IEEE Computer Society, 411–420.

[32] Akond Rahman, Chris Parnin, and Laurie A. Williams. 2019. The seven sins: security smells in infrastructure as code scripts. In *ICSE*. IEEE / ACM, 164–175.

[33] Akond Rahman, Shazibul Islam Shamim, Dibyendu Brinto Bose, and Rahul Pandita. 2023. Security Misconfigurations in Open Source Kubernetes Manifests: An Empirical Study. *ACM Trans. Softw. Eng. Methodol.* 32, 4 (2023), 99:1–99:36.

[34] Foyzur Rahman and Premkumar T. Devanbu. 2011. Ownership, experience and defects: a fine-grained study of authorship. In *ICSE*. ACM, 491–500.

[35] Romain Robbes, Théo Matricon, Thomas Degueule, André C. Hora, and Stefano Zacchiroli. 2026. Promises, Perils, and (Timely) Heuristics for Mining Coding Agent Activity. In *MSR*. IEEE.

[36] Diomidis Spinellis and Georgios Gousios. 2018. How to analyze git repositories with command line tools: we're not in kansas anymore. In *ICSE (Companion Volume)*. ACM, 540–541.

[37] Stan Development Team. 2025. *Stan Reference Manual, 2.35*. https://mc-stan.org

[38] Mahan Tafreshipour, Aaron Imani, Eric Huang, Eduardo Santana de Almeida, Thomas Zimmermann, and Iftekhar Ahmed. 2025. Prompting in the Wild: An Empirical Study of Prompt Evolution in Software Repositories. In *MSR*. IEEE, 686–698.

[39] Zefan Wang, Zichuan Liu, Yingying Zhang, Aoxiao Zhong, Jihong Wang, Fengbin Yin, Lunting Fan, Lingfei Wu, and Qingsong Wen. 2024. RCAgent: Cloud Root Cause Analysis by Autonomous Agents with Tool-Augmented Large Language Models. In *CIKM*. ACM, 4966–4974.

[40] Zezhou Yang, Sirong Chen, Cuiyun Gao, Zhenhao Li, Xing Hu, Kui Liu, and Xin Xia. 2025. An Empirical Study of Retrieval-Augmented Code Generation: Challenges and Opportunities. *ACM Trans. Softw. Eng. Methodol.* 34, 7 (2025), 188:1–188:28.

[41] Jiaxin Yu, Liming Fu, Peng Liang, Amjed Tahir, and Mojtaba Shahin. 2023. Security Defect Detection via Code Review: A Study of the OpenStack and Qt Communities. In *ESEM*. IEEE, 1–12.

[42] Zhaoyang Yu, Minghua Ma, Xiaoyu Feng, Ruomeng Ding, Chaoyun Zhang, Ze Li, Murali Chintalapati, Xuchao Zhang, Rujia Wang, Chetan Bansal, Saravan Rajmohan, Qingwei Lin, Shenglin Zhang, Changhua Pei, and Dan Pei. 2025. Triangle: Empowering Incident Triage with Multi-LLM-Agents. *FSE* (2025).

[43] Zhe Yu, Christopher Theisen, Laurie A. Williams, and Tim Menzies. 2021. Improving Vulnerability Inspection Efficiency Using Active Learning. *IEEE Trans. Software Eng.* 47, 11 (2021), 2401–2420.

[44] Feng Zhang, Ahmed E. Hassan, Shane McIntosh, and Ying Zou. 2017. The Use of Summation to Aggregate Software Metrics Hinders the Performance of Defect Prediction Models. *IEEE Trans. Software Eng.* 43, 5 (2017), 476–491.

[45] Jiale Zhang, Liangqiong Tu, Jie Cai, Xiaobing Sun, Bin Li, Weitong Chen, and Yu Wang. 2022. Vulnerability Detection for Smart Contract via Backward Bayesian Active Learning. In *ACNS Workshops (Lecture Notes in Computer Science, Vol. 13285)*. Springer, 66–83.

[46] Thomas Zimmermann and Nachiappan Nagappan. 2008. Predicting defects using network analysis on dependency graphs. In *ICSE*. ACM, 531–540.

[47] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. 2007. Predicting Defects for Eclipse. In *PROMISE@ICSE*. IEEE Computer Society, 9.