GitChameleon 2.0: Evaluating AI Code Generation Against Python Library Version Incompatibilities

Diganta Misra[1,2]*, **Nizar Islah**[3,10]*, **Victor May**[4] ,
Brice Rauby[3,5] , Zihan Wang[6] , Justine Gehring[3,7,8] , Antonio Orvieto[1,2,9] ,
Muawiz Chaudhary[3] , Eilif B. Muller[3,10] , Irina Rish[3,10] , Samira Ebrahimi Kahou[3] , Massimo Caccia[11]
[1]ELLIS Institute Tübingen [2]MPI-IS Tübingen [3]Mila Quebec AI Institute
[4]Google [5]Polytechnique Montréal [6]McGill University, Montréal
[7]Moderne [8]Gologic [9]Tübingen AI Center
[10]Université de Montréal [11]ServiceNow Research

Correspondence:
diganta.misra@tue.ellis.eu, nizar.islah@mila.quebec

Abstract

The rapid evolution of software libraries poses a considerable hurdle for code generation, necessitating continuous adaptation to frequent version updates while preserving backward compatibility. While existing code evolution benchmarks provide valuable insights, they typically lack execution-based evaluation for generating code compliant with specific library versions. To address this, we introduce **GitChameleon 2.0**, a novel, meticulously curated dataset comprising 328 Python code completion problems, each conditioned on specific library versions and accompanied by executable unit tests. **GitChameleon 2.0** rigorously evaluates the capacity of contemporary large language models (LLMs), LLM-powered agents, code assistants, and RAG systems to perform version-conditioned code generation that demonstrates functional accuracy through execution. Our extensive evaluations indicate that state-of-the-art systems encounter significant challenges with this task; enterprise models achieving baseline success rates in the 48-51% range, underscoring the intricacy of the problem. By offering an execution-based benchmark emphasizing the dynamic nature of code libraries, **GitChameleon 2.0** enables a clearer understanding of this challenge and helps guide the development of more adaptable and dependable AI code generation methods.

1 Introduction

Large language models (LLMs) are increasingly integral to software development, being adopted for tasks like code generation and review [Council, 2024, Lambiase et al., 2025].

Despite LLM advancements like larger context windows [Su et al., 2023], faster inference [Dao et al., 2022], and high performance on general coding benchmarks [Hendrycks et al., 2021, Chen et al., 2021], a critical capability remains under-evaluated: generating code that is compliant with a specific library version. This task of version-switching, which is essential for robust development in environments with fixed or legacy dependencies, is not well-verified in contemporary LLMs.

Existing benchmarks, while valuable, often focus on migrating codebases to newer versions (i.e., code evolution) or use non-executable evaluation methods. They do not fully address the challenge of gen-

*Equal contribution.

39th Conference on Neural Information Processing Systems (NeurIPS 2025) Workshop: Deep Learning For Code.

Problem Statement

Instruction: Write a *custom_violinplot* function that visualizes x and y from a Pandas DataFrame; scales the bandwidth to 1.5. Use the library *Seaborn* version 0.13.0.

```
import seaborn as sns
from matplotlib.axes import axes

def custom_violinpolot(data: pd.DataFrame)
                -> Axes:
    return
```

Attempted Solution

Model: gpt-4o-mini

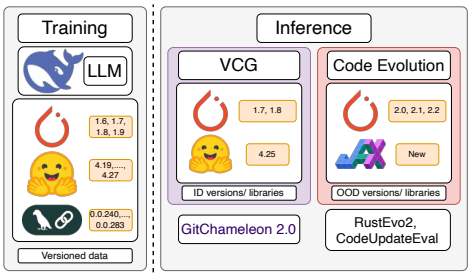
Solution:

```
sns.violinplot(x='x', y='y', data=data,
bw=1.5)
```

Validation Result:

```
AssertionError: bw parameter should not be
used. Use bw_method and bw_adjust instead.
```

(a) In this **GitChameleon 2.0** problem, the `gpt-4o-mini` model produced an incorrect solution due for `seaborn.violinplot` by using the deprecated `bw` parameter, instead of the appropriate `bw_method` and `bw_adjust` required by the specified library version.



(b) *An illustration of two evaluation paradigms for code generation models.* **Code Evolution** (right) assesses model capabilities on *out-of-distribution (OOD)* data, using library versions or new libraries not encountered during training. In contrast, **Version-Conditioned Generation (VCG)** (left) focuses on the practical ability to generate code for specific, *in-distribution (ID)* library versions that the model has seen before.

erating new, functionally correct code for a static version constraint. For instance, PyMigBench [Islam et al., 2023] provides comprehensive datasets of real-world, inter-library migrations, rather than focusing on executable, intra-library tasks conditioned on specific versions. CodeUpdateArena [Liu et al., 2025] valuably assesses LLM knowledge editing using synthetically generated API updates for functions in popular libraries, a different approach from using documented historical breaking changes. Other relevant studies, such as Wang et al. [2024b], investigate the propensity of LLMs to generate code with deprecated APIs, which does not entirely cover the broader capability of generating software that adheres to precise, user-specified library versions involving various types of API changes.

Code Evolution vs. Version Conditioned Generation (VCG). Existing code evaluation benchmarks often focus on assessing the code evolution or migration capabilities of LLMs, where changes occur only in the forward direction and typically involve unseen library versions or entirely new libraries. This framing inherently makes the task out-of-distribution (OOD), as illustrated in Figure 1b. In contrast, version-conditioned generation (VCG)—the ability of LLMs to produce code aligned with specific, previously seen library versions—is critical for practical deployment. It enables models to function reliably in real-world production environments or constrained settings where the libraries in use may not be the latest stable versions. To better evaluate this capability, a benchmark must pose problems that are strictly *in-distribution (ID)* with respect to the relevant library version(s) required to solve them.

To bridge this gap, our work introduces **GitChameleon 2.0**, an executable benchmark designed to assess the capability of LLMs and AI agents in generating version-aware Python code. **GitChameleon 2.0** features problems centered on documented breaking changes from popular libraries, requiring models to produce solutions for explicitly specified versions (an illustrative example is shown in Figure 1a). The development of such a benchmark faces challenges in meticulously curating version-specific breaking changes from library changelogs and crafting corresponding testable scenarios. Our comprehensive evaluation of diverse LLM-based tools on **GitChameleon 2.0** reveals critical limitations in existing systems' ability to handle library versioning.

In summary, our contributions are highlighted as follows:

- We introduce a novel code completion benchmark **GitChameleon 2.0** consisting of 328 Python-based version-conditioned problems, including visible tests for self-debugging and documentation references for Retrieval-Augmented Generation (RAG).

- We present a comprehensive empirical study on **GitChameleon 2.0**, evaluating the capabilities of a diverse range of contemporary AI code generation systems, including AI agents, IDE-integrated and CLI-based coding assistants, and RAG-based LLM pipelines.
- We reveal critical limitations in the ability of current AI systems to adhere to specific versioning constraints and highlight factors impacting their performance, thereby providing insights to steer the development of more adaptable and dependable AI code generation methods.

2 GitChameleon 2.0 Benchmark

We introduce **GitChameleon 2.0**, a manually authored benchmark that comprises 328 Python-based version-conditioned problems focused on popular code libraries. To evaluate performance on **GitChameleon 2.0**, each problem is accompanied by a suite of assertion-based unit tests, enabling a thorough execution-based assessment of potential solutions. The dataset was constructed through careful manual effort, with over 350 hours invested in identifying historical breaking changes, crafting problem statements, and validating unit tests. In the following sections, we detail the dataset structure, dataset statistics, evaluation metrics, and sample verification process.

2.1 Dataset Structure

Each dataset sample includes a problem related to a breaking change in a Python library.

To validate a candidate solution, we provide a suite of tests, consisting of a comprehensive suite of **Hidden Tests** to be used for model performance evaluation and ranking and a concise **Visible Test** to provide execution feedback for Self-Debugging [Chen et al., 2023] experiments.

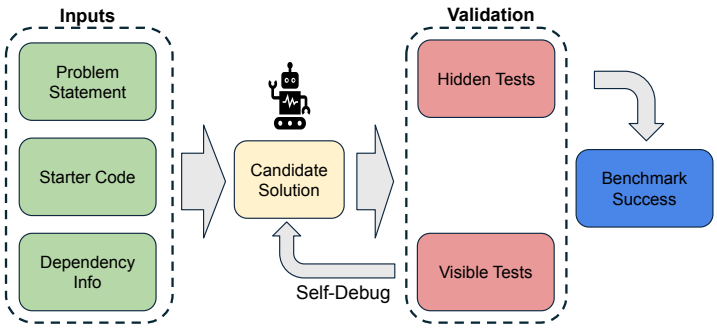
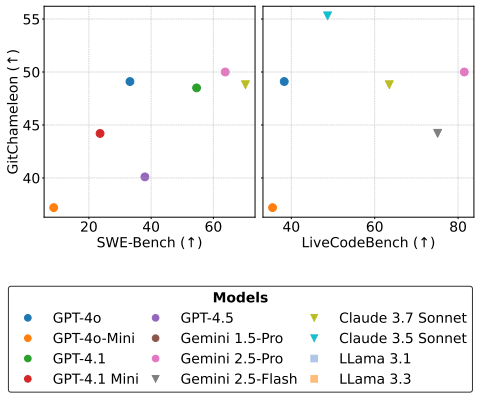


Figure 2: An illustration of the workflow for a single example within **GitChameleon 2.0**. The inputs, comprising the Problem Statement, Starter Code, and Dependency Info, are processed by an LLM or an AI agent to generate a Candidate Solution. This candidate solution then undergoes validation using the Hidden Tests to determine success on the benchmark. Results from the Visible Tests can be fed back into the solution method for self-debugging.

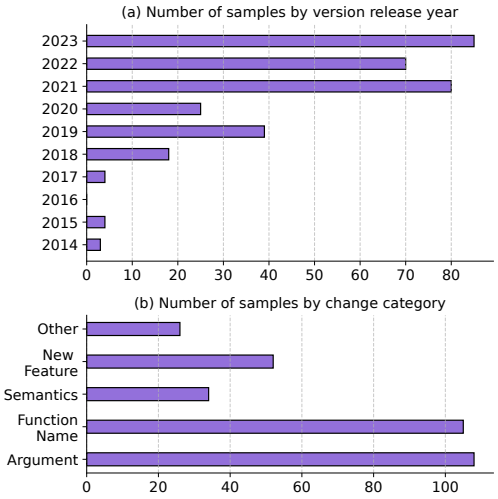
The detailed structure of dataset samples is presented in Table 4. For a schematic of the workflow for evaluating a method against a sample from **GitChameleon 2.0**, see Figure 2.

2.2 Evaluation Metrics

The benchmark metric is the success rate on hidden tests, which directly penalizes version mismatches that cause runtime errors during our execution-based validation. As a secondary metric, we use the API Hit Rate Wang et al. [2024a]: the percentage of generated solutions that correctly call all APIs specified in the ground-truth solution. Note that this hit rate can be lower than the success rate, as functionally correct alternative solutions may use different APIs.



(a) **Can you predict GitChameleon 2.0 performance from other code generation benchmarks?** Here we present the Spearman (ρ) and Pearson (r) correlations between **GitChameleon 2.0**, SWE-Bench [Jimenez et al., 2024], and LiveCodeBench [Jain et al., 2024]. GitChameleon exhibits a moderate correlation with SWE-Bench, with ρ of 0.550 and r of 0.675; and a weak correlation with LiveCodeBench, with ρ of 0.214 and r of 0.130.



(b) **Dataset Statistics.** (a) Most versions in **GitChameleon 2.0** were released between 2021–2023, with a few in earlier years. (b) The most common type of change between versions was an argument or attribute change, while semantic or functional changes were least common.

Figure 3: **Analysis of the GitChameleon 2.0 benchmark**. **Left**: Performance correlation against `SWE-Bench` and `LiveCodeBench`. **Right**: Dataset statistics showing (a) the distribution of samples by version release year and (b) the frequency of different API change categories.

2.3 Statistics

As demonstrated in Fig. 3b(a), most of the samples in **GitChameleon 2.0** are from versions of libraries released in the years 2021-2023. We intentionally use versions that fall within the training window of most evaluated models. The challenge is therefore not one of data contamination, but of **control and disambiguation**: when a model has been exposed to multiple library versions, can it correctly generate code for the specific version required by the prompt.

Further details about the benchmark and its construction process are presented in Appendix A.

3 Empirical Study

We evaluate **GitChameleon 2.0** in a comprehensive selection of settings, including Greedy Decoding, Chain-of-Thought [Wei et al., 2023], Self-Debugging [Chen et al., 2023], RAG [Lewis et al., 2020], Multi-Step Agents [Yao et al., 2023] and enterprise Coding Assistant software products, to assess their ability to generate version-specific executable code.

This section first presents the experimental setup, then reports the experiment results in each setting, and finally shows a breakdown of the observed results along a few key dimensions.

3.1 Experimental Setup

In this section, we present the experimental setup used for each of our settings. To ensure version compliance, we use a dual control mechanism: the target version is explicitly included in the model's prompt, and the validation environment is configured with that exact library version. All prompts are shown in Appendix I. For prompt optimization, we used the Anthropic Prompt Improver [2]. Further automated prompt optimization efforts did not make a significant change, as described in Table 10.

[2] https://docs.anthropic.com/en/docs/build-with-claude/prompt-engineering/prompt-improver

3.1.1

Greedy Decoding We configured the generation parameters with a sampling temperature of 0 and a `top_p` value of 0.95. We had specified a structured output schema that specifies the fields `Answer` and `Explanation`, where both are of type `string`.

3.1.2 Zero-Shot Chain-Of-Thought (CoT)

We had used the same generation parameters as for Greedy Decoding and an output schema that specifies the fields `Answer` and `Steps`, where the former is a of type `string` and the latter is a list of `string`.

3.1.3 Self-Debugging

On examples that failed with Greedy Decoding, we employed the method described in Chen et al. [2023] to feed the visible test error trace along with the model's explanation of its output back to the model.

3.1.4 Retrieval-Augmented Generation

We designed a RAG Lewis et al. [2020] pipeline where we first constructed a vectorized database (VectorDB) by embedding each sample's relevant API documentation with the OpenAI `text-embedding-3 large` model OpenAI [2024]. The corpus used for constructing the VectorDB included 536 documents, with 140 samples having 1 associated document, 168 having 2 associated documents and 20 having 3 documents.

Subsequently, we used DocPrompting Zhou et al. [2022] to query the VectorDB to generate solutions.

3.1.5 Multi-Step Agent

We conducted experiments with a tool-calling agent, as implemented by the smolagents Roucher et al. [2025] framework. This agent implementation mostly follows the ReAct Yao et al. [2023] method, but, it alternates between acting and planning Li [2024] steps.

Following the Agentic RAG approach [Singh et al., 2025], we had equipped the agent with a grounding tool in order to assess its capability to independently fetch relevant info for solving the benchmark problems. To this end, we had experimented with the following grounding tools: DuckDuckGo Search DuckDuckGo [2025], Perplexity Perplexity AI [2024], and Gemini with Grounding Google [2025].

Additionally, we examined agentic multi-step self-debugging Jin et al. [2024] by including or omitting a code execution sandbox tool Rabin et al. [2025], which provides the needed dependencies for each example. The sandbox takes a Python program as input and outputs the standard output from the program.

3.2 Experiment Results

This section presents the benchmark results in each setting, as described in the **Experimental Setup** section (3.1). Table 1 contains the results for Greedy Decoding, Self-Debug and Zero-Shot CoT.

3.2.1 Greedy Decoding

We observe that the largest Enterprise-grade models, including `Claude 3.7 Sonnet`, `Gemini 2.5 Pro`, `GPT-4.1`, `GPT-4o`, and `o1`, exhibit comparable hidden success rates, generally falling within the 48–51% range. Among these `o1` (51.2% hidden) achieves the highest hidden success rate.

The open-weight Llama models are notably behind, even the recently released `Llama 4 Maverick FP8` (40.8% hidden success rate).

Model size clearly impacts performance: for instance, `Gemini 2.5 Flash` trails its `Pro` counterpart by nearly 12% on hidden tests (38.1% vs. 50.0%). Similarly, the `mini` and `nano` series within the GPT family (e.g., `GPT-4.1-mini`, `GPT-4.1-nano`, `GPT-4o-mini`) consistently show lower performance

than their larger full-size siblings, with differences on hidden tests ranging from approximately 4 to 15 points.

3.2.2 Zero-Shot Chain-Of-Thought

This approach does not uniformly improve LLM performance across all models. While some models demonstrate significant gains in hidden success rates, a substantial number of enterprise-grade models and their smaller variants experience performance degradation.

For instance, notable improvements in hidden success rates are observed in models such as `Llama 3.1 Instruct Turbo` (from 30.2% to 36.6%, a +6.4 point increase) and `o3-mini` (from 45.1% to 50.9%, a +5.8 point increase).

Conversely, several models exhibit a decrease in performance with CoT. Prominent examples include `Gemini 2.0 Flash` (from 44.2% to 36.0%) and even the top-performing `o1` (from 51.2% to 41.2%).

Model	Greedy Decoding			Greedy with Self-Debug			Zero-shot CoT	
	Success Rate (%)		API Hit Rate (%)	Success Rate (%)		API Hit Rate (%)	Success Rate (%)	API Hit Rate (%)
	Hidden	Visible		Hidden	Visible		Hidden	
Open-Weights Models								
Llama 3.1 Instruct Turbo	30.2±2.5	38.1±2.7	39.7±2.7	52.1±2.8	69.2±2.5	41.5±2.7	36.6±2.7	35.3±2.6
Llama 3.3 Instruct Turbo 70B	36.3±2.7	43.3±2.7	36.4±2.7	53.0±2.8	70.1±2.5	37.4±2.7	37.5±2.7	37.2±2.7
Llama 4 Maverick 400B	40.8±2.7	46.6±2.8	**49.5**±2.8	58.5±2.7	72.3±2.5	**46.8**±2.8	**46.6**±2.8	41.3±2.7
Qwen 2.5-VL Instruct 72B	**48.2**±2.8	**55.5**±2.7	43.8±2.7	**64.6**±2.6	**77.4**±2.3	45.3±2.7	45.1±2.7	**43.0**±2.7
Enterprise Models								
Claude 3.7 Sonnet	48.8±2.8	55.8±2.7	46.0±2.8	65.9±2.6	75.9±2.4	47.6±2.8	45.1±2.7	43.4±2.7
Gemini 1.5 Pro	45.1±2.7	51.5±2.8	46.8±2.7	62.5±2.8	72.6±2.4	48.6±2.7	43.3±2.7	44.6±2.8
Gemini 2.0 Flash	44.2±2.7	50.6±2.8	43.8±2.7	**70.4**±2.7	79.0±2.4	49.4±2.7	36.0±2.6	41.8±2.7
Gemini 2.5 Pro	**50.0**±2.8	**61.0**±2.8	47.7±2.7	61.3±2.8	73.8±2.2	49.2±2.7	49.4±2.8	49.1±2.8
Gemini 2.5 Flash	38.1±2.6	41.8±2.7	45.4±2.7	65.9±2.8	73.2±2.4	45.8±2.7	30.8±2.5	**49.8**±2.8
GPT-4.1	48.5±2.8	49.1±2.8	46.8±2.7	63.4±2.8	76.8±2.1	48.3±2.7	47.9±2.8	44.5±2.7
GPT-4.1-mini	44.2±2.7	50.0±2.8	44.5±2.7	68.0±2.8	**79.3**±2.3	46.3±2.7	24.1±1.8	41.3±2.7
GPT-4.1-nano	33.8±2.6	35.1±2.6	43.1±2.7	67.7±2.7	74.4±2.6	45.8±2.7	11.9±1.8	32.1±2.5
GPT-4o	49.1±2.8	54.0±2.8	46.5±2.7	64.9±2.8	72.3±2.5	48.0±2.7	**50.3**±2.8	42.5±2.7
GPT-4o-mini	37.2±2.6	46.3±2.7	38.4±2.6	60.4±2.7	71.6±2.6	40.6±2.7	36.0±2.6	37.3±2.6
GPT-4.5	40.8±2.7	46.0±2.7	**52.8**±2.8	66.2±2.8	74.4±2.4	**54.4**±2.7	39.9±2.6	48.8±2.8
Grok 3	48.2±2.8	53.7±2.8	44.8±2.7	67.1±2.8	77.1±2.3	46.3±2.8	49.4±2.8	44.2±2.7
Mistral Medium 3	43.6±2.7	49.1±2.8	44.2±2.7	61.3±2.8	71.3±2.5	45.4±2.7	44.2±2.7	44.1±2.7

Table 1: Success rate on visible and hidden tests and API hit rate under the Greedy, Self-Debug, and Zero-shot CoT settings, grouped by OSS vs. Enterprise models. Model ranking on the benchmark is determined by **Hidden Success Rate**. Visible Success Rate figures are for context on Self-Debugging. The values after the ± symbol denote the standard error. The best result in each column is in bold. For full model details and citations, please refer to Appendix J.

3.2.3 LLM Self-Debugging

We evaluate the models' self-correction capabilities on problems that failed during greedy decoding. By providing the error trace from the visible test as feedback, we observe that this self-debugging process yields substantial performance gains, as detailed below.

Hidden Success Rate: Across models, Self-Debugging significantly improves the hidden success rates. Observed gains range from approximately 10% to 20%. For instance, `Llama 3.1`'s hidden success rate increases from 30% to 52.1%, and `GPT-4.1-mini` shows an improvement from 44% to 68%. This demonstrates the strong capability of modern LLMs to diagnose failures and generate corrected code.

Visible Success Rate: As expected, the improvement is even more pronounced on visible tests, ranging from 13 to 37 points. For instance, `GPT-4.1`'s success rate improves from 49% to 69%, `Claude 3.7 Sonnet`'s success rate improves from 56% to 83% and `Gemini 2.0 Flash` improves from 50% to 75%.

Visible-Hidden Gap Analysis: We analyze the effect of self-debugging on the **"Visible-Hidden Gap"**, which we define as the difference between the success rate on visible and hidden tests:

$$\text{Visible-Hidden Gap} = (\text{Success Rate}_{\text{visible}}) - (\text{Success Rate}_{\text{hidden}})$$

Figure 4 plots this gap for each model.

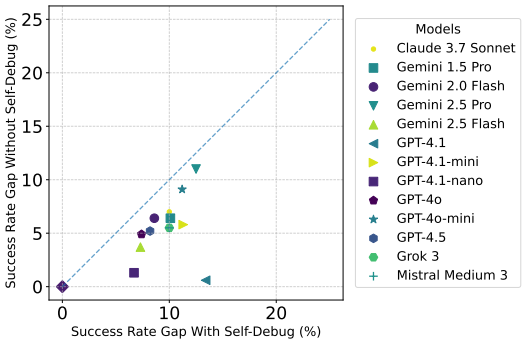


Figure 4: **Analysis of the Visible-Hidden Gap Before and After Self-Debugging**. We analyze how self-debugging affects the gap between the success rate on visible and hidden tests. We can see that for all models, the gap increases after self-debugging. This shows that self-debugging on visible tests has a limited ability to improve on the hidden tests.

3.2.4 Multi-Step Agent

We report the performance of Multi-Step Agents on **GitChameleon 2.0** in Table 2.

Model	Grounding Method	Success Rate (%)		API Hit Rate (%)	
		No Sandbox	Sandbox	No Sandbox	Sandbox
Claude Sonnet 3.5	DuckDuckGo	41.7±2.7	**55.3**±2.7	42.2±2.7	48.9±2.8
	Perplexity	44.1±2.7	51.4±2.8	41.8±2.7	46.0±2.8
	Grounded Gemini	40.0±2.7	53.7±2.8	41.0±2.7	45.2±2.7
Gemini 1.5 Pro	DuckDuckGo	46.0±2.8	49.8±2.8	47.4±2.8	50.3±2.8
	Perplexity	**46.5**±2.8	44.4±2.7	47.2±2.8	46.6±2.8
	Grounded Gemini	44.1±2.7	49.2±2.8	**49.7**±2.8	**51.2**±2.8
GPT-4o	DuckDuckGo	23.9±2.4	33.2±2.6	44.2±2.7	48.1±2.8
	Perplexity	33.5±2.6	41.5±2.7	43.2±2.7	44.7±2.7
	Grounded Gemini	25.4±2.4	50.0±2.8	46.5±2.8	44.2±2.7

Table 2: Multi-Step Agent performance with different models, grounding methods, and sandbox states. The values after the ± symbol denote the standard error. The best result in each column is in bold.

A clear and significant trend is the substantial increase in success rates for all models and grounding methods when giving the agent a sandbox tool. Overall, `Claude Sonnet 3.5` demonstrated the highest success rates with a sandbox, across all grounding methods, while `Gemini 1.5 Pro` demonstrated the best results without a sandbox.

3.2.5 Retrieval-Augmented Generation

Table 3 presents the performance of various models with RAG. Many models exhibit a significant (up to 10%) boost in success rate with RAG compared to greedy decoding alone. Notably, `GPT-4.1`,

[3]This version of the model is not FP8-quantized, unlike the one presented in Table 1

Model	Success Rate (%)	API Hit Rate (%)	Precision (%)	Recall (%)	MRR
Open-Weights Models					
Deepseek V3	**48.9**$\pm$2.8	48.5$\pm$2.8	41.6$\pm$2.2	50.4$\pm$2.8	0.62$\pm$0.03
Llama 4 Maverick[3]	45.1$\pm$2.7	**50.5**$\pm$2.8	41.2$\pm$2.2	49.8$\pm$2.8	0.61$\pm$0.03
Qwen3	41.8$\pm$2.7	39.6$\pm$2.7	36.3$\pm$2.0	46.9$\pm$2.8	0.56$\pm$0.03
Jamba 1.6 Large	41.8$\pm$2.7	47.1$\pm$2.8	**41.9**$\pm$2.2	**50.7**$\pm$2.8	**0.62**$\pm$0.03
Enterprise Models					
Claude 3.7 Sonnet	56.1$\pm$2.7	53.0$\pm$2.8	41.9$\pm$2.2	50.7$\pm$2.8	0.62$\pm$0.03
Claude 4 Sonnet	**59.4**$\pm$2.8	**55.8**$\pm$2.8	**41.9**$\pm$2.2	**50.7**$\pm$2.8	**0.62**$\pm$0.03
Gemini 2.5 Pro	56.7$\pm$2.7	51.1$\pm$2.8	41.9$\pm$2.2	50.7$\pm$2.8	0.62$\pm$0.03
GPT-4.1	58.5$\pm$2.7	51.8$\pm$2.8	41.2$\pm$2.2	50.1$\pm$2.8	0.61$\pm$0.03
Grok3	54.3$\pm$2.7	55.2$\pm$2.8	41.6$\pm$2.2	50.4$\pm$2.8	0.62$\pm$0.03
Mistral Medium 3	52.4$\pm$2.7	51.2$\pm$2.8	41.6$\pm$2.2	50.4$\pm$2.8	0.62$\pm$0.03
Devstral Small	43.3$\pm$2.7	45.1$\pm$2.8	41.6$\pm$2.2	50.4$\pm$2.8	0.62$\pm$0.03
Nova Pro	44.2$\pm$2.7	42.4$\pm$2.7	40.7$\pm$2.2	49.6$\pm$2.8	0.60$\pm$0.03

Table 3: RAG performance for a subset of models when retrieving $k = 3$ most relevant documents. The best success rate and API hit rate results for each model group are in bold. The values after the ± symbol denote the standard error. An extended version of the RAG experiment results is presented in Appendix C.

the best performing model achieves a success rate of 58.5%, up from 48.5% with greedy decoding. These results demonstrate that the benchmark is still challenging even with access to the library documentation, with over 40% of the problems remaining unsolved in the best case.

3.3 In-Depth Analysis of Findings

This section provides a detailed analysis of the experimental results, focusing on model performance across several key dimensions. These dimensions include the impact of different API change types, a comparison between success rate and API hit rate, and the effectiveness of self-debugging across various error types.

Comparison of Success Rate and API Hit Rate API hit rate shows a moderate positive Pearson correlation with hidden-test success under Greedy Decoding with the Pearson correlation coefficient ($r = 0.392$, $p = 0.097$, $N = 19$), indicating that models which invoke the ground truth APIs more often tend to perform better on hidden tests in the Greedy setting, but falls just short of statistical significance at 5% level. Under Zero-Shot CoT, the correlation remains similar in magnitude ($r = 0.483$) and is statistically significant ($p = 0.036$, $N = 19$). In the Self-Debug regime, however, the association becomes both stronger and highly significant ($r = 0.615$, $p = 0.011$, $N = 16$), demonstrating that when models can iteratively refine their outputs, invoking ground truth APIs becomes an especially reliable predictor of hidden-test performance.

Analysis of Performance by Type of API Change Figure 5 illustrates the performance of models across various API change types within the **GitChameleon 2.0** benchmark, revealing notable variations in success rates. Semantic changes were the most tractable, with success rates ranging from 60–80% with Self-Debug and 55–65% without. New-feature additions proved to be the most challenging, with success rates between 25–50% for Greedy Decoding and 50–65% for Self-Debug. Notably, the Code Assistant `Goose` exhibited a substantial discrepancy in its performance on semantic and function-name changes compared to argument changes and new features. This suggests a heightened sensitivity to change category for `Goose`, a characteristic not observed in the enterprise models or the `Claude`-powered tool-calling agent.

Self-Debug Error Categorization Figure 6 shows that self-debugging consistently lowers the rate of every class of traceback error, both in absolute numbers and relative terms:

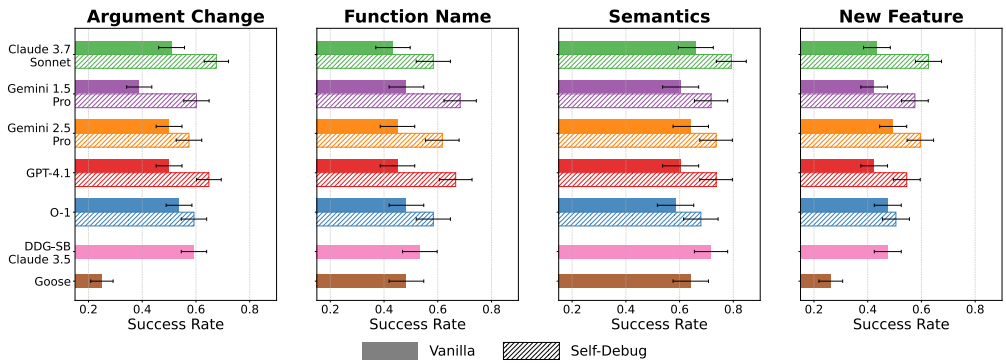


Figure 5: **Success Rate Breakdown by Type of Change**: We analyze success rates with and without self-debugging, grouped by the type of change. Light shaded bars represent values obtained from self-debugging. Standard error is drawn as a black line. We include `DDG-SB`, a Multi-Step Agent variant where DuckDuckGo is used for grounding and access to a sandbox is enabled. and the Coding Assistant `Goose`. Self-Debug results for these are omitted.

(a) **Raw Counts:** We observe that for all error categories—from the most common (`AssertionError` and `TypeError`) down to the rarest (`RuntimeError`)—applying Self-Debugging significantly lowers the total number of failures.

(b) **Percentage Reduction:** When normalized by the `Greedy Decoding` baseline, reductions span roughly 50% up to about 90%. The biggest relative improvements appear in the infrequent categories—such as `RuntimeError` and `SyntaxError`—while the common `AssertionError` and `TypeError` still see decrease in the range of 60-70%.

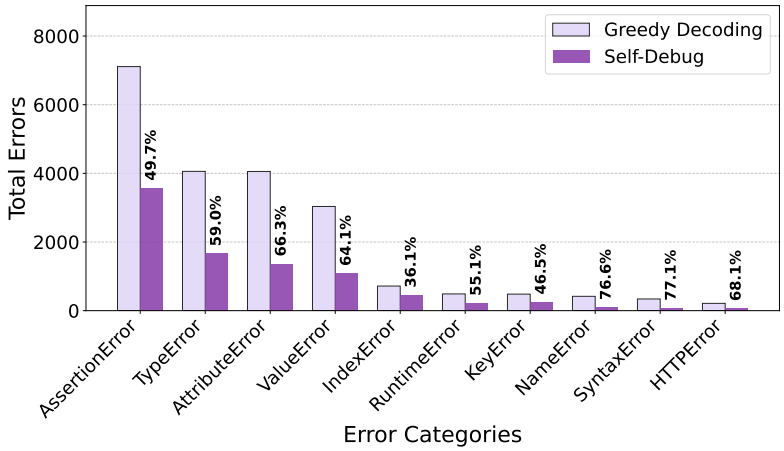


Figure 6: **Total error count for each category under Greedy decoding versus Self-Debug**. Self-Debug yields substantial decreases all types of errors.

4 Related Work

The continuous evolution of software libraries presents significant challenges for AI-driven code generation. This section reviews existing benchmarks designed to evaluate model performance in this context. Specialized frameworks developed to address the challenge are presented in appendix D.2

The challenge of evaluating large language models (LLMs) in the context of evolving software libraries and their versions has been approached by several benchmarks. These benchmarks, while valuable, often differ in scope, methodology, or evaluation techniques compared to **GitChameleon 2.0**.

PyMigBench Focusing on Python library migration, this benchmark uses 321 real-world instances, evaluating both individual code transformations and the functional correctness of entire migrated

segments via unit tests Islam et al. [2023]. PyMigBench revealed that LLMs often handle individual changes well but struggle with achieving full functional correctness, especially for complex argument transformations.

VersiCode Wu et al. [2024] and the dataset by Wang et al. Wang et al. [2024b] address library evolution but primarily depend on string matching for evaluation.

CodeUpdateArena Liu et al. [2025] investigates model adaptation to synthetically generated API updates for functions in popular libraries.

GitChameleon 2.0 distinguishes itself by focusing on the real-world scenario where developers are often constrained to specific library versions due to technical debt. Unlike CodeUpdateArena's synthetic changes, **GitChameleon 2.0** evaluates LLMs on their ability to generate code for actual, documented historical breaking changes within library versions they were likely exposed to during training. Furthermore, diverging from the string-matching evaluations of VersiCode and Wang et al. Wang et al. [2024b], **GitChameleon 2.0** is based on executable tests. This provides a more practical and rigorous assessment of functional accuracy in version-specific code generation. For an extended discussion of how **GitChameleon 2.0** is differentiated from existing work, please see Appendix D.2.

5 Conclusion

The rapid evolution of software libraries presents a critical challenge for LLM-powered AI systems in generating functionally correct, version-conditioned code. To address this, we introduce **GitChameleon 2.0**, a novel Python-based benchmark meticulously curated with version-conditioned problems and executable tests. Our extensive evaluation reveals that state-of-the-art LLMs, agents and code assistants currently struggle significantly with this task, achieving modest success rates.

By shedding light on current limitations and facilitating execution-based evaluation, **GitChameleon 2.0** aims to foster the development of more robust and adaptable code generation models for evolving software environments.

Limitations

While we aim to provide a comprehensive and holistic evaluation of LLMs on the task of version-conditioned generation, our benchmark is currently limited to Python and a small set of libraries. Moreover, we focus solely on code generation from natural language instructions, and do not evaluate version-to-version translation—i.e., converting code from one library version to another—even when both versions are in-distribution relative to the model's training. For instance, if a model has been trained on PyTorch versions 1.7, 1.8, and 1.9, it would be valuable to assess whether it performs better when given a solution in 1.8 and asked to upgrade to 1.9 or downgrade to 1.7. Finally, we do not include human evaluations, which could provide a baseline for estimating average human performance on this task.

References

Abubakar Abid, Ali Abdalla, Ali Abid, Dawood Khan, Abdulrahman Alfozan, and James Zou. Gradio: Hassle-free sharing and testing of ML models in the wild, June 2019.

Meta AI. Everything we announced at our first-ever LlamaCon. https://ai.meta.com/blog/llamacon-llama-news/, 2025. Discusses Llama 3.3 Instruct Turbo and Llama 4 Maverick.

Mohannad Alhanahnah, Yazan Boshmaf, and Benoit Baudry. DepsRAG: Towards managing software dependencies using large language models. *arXiv preprint arXiv:2405.20455v2*, 2024. URL https://arxiv.org/html/2405.20455v2.

Anthropic. Claude 3.7 Sonnet and Claude Code. https://www.anthropic.com/news/claude-3-7-sonnet, February 2025.

Arcee. Model Selection | Arcee AI Documentation — docs.arcee.ai. https://docs.arcee.ai/arcee-conductor/arcee-small-language-models/model-selection#caller-large-tool-use-and-function-call. [Accessed 15-07-2025].

Farnaz Behrang, Zhizhou Zhang, Georgian-Vlad Saioc, Peng Liu, and Milind Chabbi. Dr.fix: Automatically fixing data races at industry scale, 2025. URL https://arxiv.org/abs/2504.15637.

Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Andreas Joly, Bertrand Druillette, Gael Varoquaux, and Marion Gramfort. API design for machine learning software: experiences from the scikit-learn project. *arXiv preprint arXiv:1309.0238*, 2013.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *ArXiv*, July 2021.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug, 2023. URL https://arxiv.org/abs/2304.05128.

Keyuan Cheng, Xudong Shen, Yihao Yang, Tengyue Wang, Yang Cao, Muhammad Asif Ali, Hanbin Wang, Lijie Hu, and Di Wang. Codemenv: Benchmarking large language models on code migration, 2025. URL https://arxiv.org/abs/2506.00894.

Matteo Ciniselli, Alberto Martin-Lopez, and Gabriele Bavota. On the generalizability of deep learning-based code completion across programming language versions, 2024. URL https://arxiv.org/abs/2403.15149.

Google Cloud. Gemini 2.5 on Vertex AI: Pro, Flash & Model Optimizer Live. https://cloud.google.com/blog/products/ai-machine-learning/gemini-2-5-pro-flash-on-vertex-ai, April 2025. Discusses Gemini 2.5 Pro and Gemini 2.5 Flash.

Team Cohere, :, Aakanksha, Arash Ahmadian, Marwan Ahmed, Jay Alammar, Milad Alizadeh, Yazeed Alnumay, Sophia Althammer, Arkady Arkhangorodsky, Viraat Aryabumi, Dennis Aumiller, Raphaël Avalos, Zahara Aviv, Sammie Bae, Saurabh Baji, Alexandre Barbet, Max Bartolo, Björn Bebensee, Neeral Beladia, Walter Beller-Morales, Alexandre Bérard, Andrew Berneshawi, Anna Bialas, Phil Blunsom, Matt Bobkin, Adi Bongale, Sam Braun, Maxime Brunet, Samuel Cahyawijaya, David Cairuz, Jon Ander Campos, Cassie Cao, Kris Cao, Roman Castagné, Julián Cendrero, Leila Chan Currie, Yash Chandak, Diane Chang, Giannis Chatziveroglou, Hongyu Chen,

Claire Cheng, Alexis Chevalier, Justin T. Chiu, Eugene Cho, Eugene Choi, Eujeong Choi, Tim Chung, Volkan Cirik, Ana Cismaru, Pierre Clavier, Henry Conklin, Lucas Crawhall-Stein, Devon Crouse, Andres Felipe Cruz-Salinas, Ben Cyrus, Daniel D'souza, Hugo Dalla-Torre, John Dang, William Darling, Omar Darwiche Domingues, Saurabh Dash, Antoine Debugne, Théo Dehaze, Shaan Desai, Joan Devassy, Rishit Dholakia, Kyle Duffy, Ali Edalati, Ace Eldeib, Abdullah Elkady, Sarah Elsharkawy, Irem Ergün, Beyza Ermis, Marzieh Fadaee, Boyu Fan, Lucas Fayoux, Yannis Flet-Berliac, Nick Frosst, Matthias Gallé, Wojciech Galuba, Utsav Garg, Matthieu Geist, Mohammad Gheshlaghi Azar, Ellen Gilsenan-McMahon, Seraphina Goldfarb-Tarrant, Tomas Goldsack, Aidan Gomez, Victor Machado Gonzaga, Nithya Govindarajan, Manoj Govindassamy, Nathan Grinsztajn, Nikolas Gritsch, Patrick Gu, Shangmin Guo, Kilian Haefeli, Rod Hajjar, Tim Hawes, Jingyi He, Sebastian Hofstätter, Sungjin Hong, Sara Hooker, Tom Hosking, Stephanie Howe, Eric Hu, Renjie Huang, Hemant Jain, Ritika Jain, Nick Jakobi, Madeline Jenkins, JJ Jordan, Dhruti Joshi, Jason Jung, Trushant Kalyanpur, Siddhartha Rao Kamalakara, Julia Kedrzycki, Gokce Keskin, Edward Kim, Joon Kim, Wei-Yin Ko, Tom Kocmi, Michael Kozakov, Wojciech Kryściński, Arnav Kumar Jain, Komal Kumar Teru, Sander Land, Michael Lasby, Olivia Lasche, Justin Lee, Patrick Lewis, Jeffrey Li, Jonathan Li, Hangyu Lin, Acyr Locatelli, Kevin Luong, Raymond Ma, Lukáš Mach, Marina Machado, Joanne Magbitang, Brenda Malacara Lopez, Aryan Mann, Kelly Marchisio, Olivia Markham, Alexandre Matton, Alex McKinney, Dominic McLoughlin, Jozef Mokry, Adrien Morisot, Autumn Moulder, Harry Moynehan, Maximilian Mozes, Vivek Muppalla, Lidiya Murakhovska, Hemangani Nagarajan, Alekhya Nandula, Hisham Nasir, Shauna Nehra, Josh Netto-Rosen, Daniel Ohashi, James Owers-Bardsley, Jason Ozuzu, Dennis Padilla, Gloria Park, Sam Passaglia, Jeremy Pekmez, Laura Penstone, Aleksandra Piktus, Case Ploeg, Andrew Poulton, Youran Qi, Shubha Raghvendra, Miguel Ramos, Ekagra Ranjan, Pierre Richemond, Cécile Robert-Michon, Aurélien Rodriguez, Sudip Roy, Sebastian Ruder, Laura Ruis, Louise Rust, Anubhav Sachan, Alejandro Salamanca, Kailash Karthik Saravanakumar, Isha Satyakam, Alice Schoenauer Sebag, Priyanka Sen, Sholeh Sepehri, Preethi Seshadri, Ye Shen, Tom Sherborne, Sylvie Shang Shi, Sanal Shivaprasad, Vladyslav Shmyhlo, Anirudh Shrinivason, Inna Shteinbuk, Amir Shukayev, Mathieu Simard, Ella Snyder, Ava Spataru, Victoria Spooner, Trisha Starostina, Florian Strub, Yixuan Su, Jimin Sun, Dwarak Talupuru, Eugene Tarassov, Elena Tommasone, Jennifer Tracey, Billy Trend, Evren Tumer, Ahmet Üstün, Bharat Venkitesh, David Venuto, Pat Verga, Maxime Voisin, Alex Wang, Donglu Wang, Shijian Wang, Edmond Wen, Naomi White, Jesse Willman, Marysia Winkels, Chen Xia, Jessica Xie, Minjie Xu, Bowen Yang, Tan Yi-Chern, Ivan Zhang, Zhenyu Zhao, and Zhoujie Zhao. Command a: An enterprise-ready large language model, 2025. URL https://arxiv.org/abs/2504.00698.

Forbes Technology Council. Revolutionizing software development with large language models. https://www.forbes.com/councils/forbestechcouncil/2024/03/20/revolutionizing-software-development-with-large-language-models/, March 2024.

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

DeepSeek-AI. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025. URL https://arxiv.org/abs/2501.12948.

DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jiawei Wang, Jin Chen, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Litong Wang, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runxin Xu, Ruoyu Zhang, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Shuting Pan, T. Wang, Tao Yun, Tian Pei, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wanjia Zhao, Wei An, Wen Liu,

Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaokang Zhang, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingkai Yu, Xinnan Song, Xinxia Shan, Xinyi Zhou, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, Y. K. Li, Y. Q. Wang, Y. X. Wei, Y. X. Zhu, Yang Zhang, Yanhong Xu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Yu, Yi Zheng, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Ying Tang, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yu Wu, Yuan Ou, Yuchen Zhu, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yukun Zha, Yunfan Xiong, Yunxian Ma, Yuting Yan, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Z. F. Wu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhipeng Xu, Zhiyu Wu, Zhongyu Zhang, Zhuoshu Li, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Ziyi Gao, and Zizheng Pan. Deepseek-v3 technical report, 2025. URL https://arxiv.org/abs/2412.19437.

DuckDuckGo. DuckDuckGo: Privacy, simplified. https://duckduckgo.com/, 2025.

Lishui Fan, Mouxiang Chen, and Zhongxin Liu. Self-explained keywords empower large language models for code generation, 2024. URL https://arxiv.org/abs/2410.15966.

Google. Grounding with Google Search | Gemini API. https://ai.google.dev/gemini-api/docs/grounding, 2025.

Aric A Hagberg, Daniel A Schult, and Pieter J Swart. Exploring network structure, dynamics, and function using NetworkX. In *Proceedings of the 7th Python in Science Conference*, pages 11–15, 2008.

Charles R Harris, K Jarrod Millman, Stéfan J van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, Robert Kern, Matti Picus, Changqing Hoyer, Marten H van Kerkwijk, Alex Brett, Andrew Wen, Pete Zhang, Joe Igoe, Keith Featherstone, and Travis E Oliphant. Array programming with NumPy. *Nature*, 585(7825): 357–362, 2020.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with apps. *NeurIPS*, 2021.

J. D. Hunter. Matplotlib: A 2D graphics environment. *Computing in Science & Engineering*, 9(3): 90–95, 2007.

Amazon Artificial General Intelligence. The amazon nova family of models: Technical report and model card. *Amazon Technical Reports*, 2024. URL https://www.amazon.science/publications/the-amazon-nova-family-of-models-technical-report-and-model-card.

Mohayeminul Islam, Ajay Kumar Jha, Sarah Nadi, and Ildar Akhmetov. Pymigbench: A benchmark for python library migration. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pages 511–515, 2023. doi: 10.1109/MSR59073.2023.00075.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Haolin Jin, Zechao Sun, and Huaming Chen. Rgd: Multi-llm based agent debugger via refinement and generation guidance, 2024. URL https://arxiv.org/abs/2410.01242.

Kelsey Jordahl, Joris Van den Bossche, Martin Fleischmann, Jacob Wasserman, James McBride, Jeffrey Gerard, Jeff Tratner, Matthew Perry, Adrian Garcia Badaracco, Carson Farmer, Geir Arne

Hjelle, Alan D. Snow, Micah Cochran, Sean Gillies, Lucas Culbertson, Matt Bartos, Nick Eubank, maxalbert, Aleksey Bilogur, Sergio Rey, Christopher Ren, Dani Arribas-Bel, Leah Wasser, Levi John Wolf, Martin Journois, Joshua Wilson, Adam Greenhall, Chris Holdgraf, Filipe, and François. geopandas/geopandas: v0.8.1, July 2020. URL https://doi.org/10.5281/zenodo.3946761.

Kat Kampf. Create and edit images with Gemini 2.0 in preview. https://developers.googleblog.com/en/generate-images-gemini-2-0-flash-preview/, May 2025. Discusses Gemini 2.0 Flash.

Paul Kassianik, Baturay Saglam, Alexander Chen, Blaine Nelson, Anu Vellore, Massimo Aufiero, Fraser Burch, Dhruv Kedia, Avi Zohary, Sajana Weerawardhena, Aman Priyanshu, Adam Swanda, Amy Chang, Hyrum Anderson, Kojin Oshiba, Omar Santos, Yaron Singer, and Amin Karbasi. Llama-3.1-FoundationAI-SecurityLLM-Base-8B Technical Report. *arXiv preprint arXiv:2504.21039*, 2025. URL https://arxiv.org/abs/2504.21039. Cited for Llama 3.1 Instruct Turbo.

Sachit Kuhar, Wasi Uddin Ahmad, Zijian Wang, Nihal Jain, Haifeng Qian, Baishakhi Ray, Murali Krishna Ramanathan, Xiaofei Ma, and Anoop Deoras. Libevolutioneval: A benchmark and study for version-specific code generation, 2024. URL https://arxiv.org/abs/2412.04478.

Stefano Lambiase, Gemma Catolino, Fabio Palomba, Filomena Ferrucci, and Daniel Russo. Exploring individual factors in the adoption of llms for specific software engineering tasks, 2025. URL https://arxiv.org/abs/2504.02553.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented Generation for Knowledge-Intensive NLP Tasks. In *Advances in Neural Information Processing Systems*, volume 33, 2020.

James Li. ReAct vs Plan-and-Execute: A Practical Comparison of LLM Agent Patterns. https://dev.to/jamesli, November 2024.

Linxi Liang, Jing Gong, Mingwei Liu, Chong Wang, Guangsheng Ou, Yanlin Wang, Xin Peng, and Zibin Zheng. Rustevo: An evolving benchmark for api evolution in llm-based rust code generation, 2025. URL https://arxiv.org/abs/2503.16922.

Opher Lieber, Barak Lenz, Hofit Bata, Gal Cohen, Jhonathan Osin, Itay Dalmedigos, Erez Safahi, Shaked Meirom, Yonatan Belinkov, Shai Shalev-Shwartz, Omri Abend, Raz Alon, Tomer Asida, Amir Bergman, Roman Glozman, Michael Gokhman, Avashalom Manevich, Nir Ratner, Noam Rozen, Erez Shwartz, Mor Zusman, and Yoav Shoham. Jamba: A hybrid transformer-mamba language model, 2024. URL https://arxiv.org/abs/2403.19887.

Zeyu Leo Liu, Shrey Pandit, Xi Ye, Eunsol Choi, and Greg Durrett. Codeupdatearena: Benchmarking knowledge editing on API updates, 2025. URL https://openreview.net/forum?id=ecRyUAPshY.

Edward Loper and Steven Bird. NLTK: The natural language toolkit. In *Proceedings of the ACL-02 Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics*, pages 63–70, Philadelphia, Pennsylvania, USA, July 2002. Association for Computational Linguistics. URL https://aclanthology.org/W02-0109.

Stephan Lukasczyk and Gordon Fraser. Pynguin: Automated unit test generation for python. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pages 168–172, 2022.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. WizardCoder: Empowering code large language models with Evol-Instruct. June 2023.

Wes McKinney. Data Structures for Statistical Computing in Python. In *Proceedings of the 9th Python in Science Conference*, pages 51–56, 2010.

Mistral AI. Medium is the new large: Introducing mistral medium 3. https://mistral.ai/news/mistral-medium-3, May 2025. Accessed: 2025-05-17.

OpenAI. GPT-4o System Card. *arXiv preprint arXiv:2410.21276*, 2024. URL https://arxiv.org/abs/2410.21276. Cited for GPT-4o.

OpenAI. New embedding models and api updates. https://openai.com/index/new-embedding-models-and-api-updates/, Jan 2024. Accessed: 2025-07-28.

OpenAI. OpenAI o1 System Card. https://openai.com/index/openai-o1-system-card/, 2024. Discusses the o1 model series, including o1 and mentioning o3-mini.

OpenAI. Introducing GPT-4.1 in the API. https://openai.com/index/gpt-4-1/, April 2025a. Discusses GPT-4.1, GPT-4.1 mini, and GPT-4.1 nano.

OpenAI. Introducing GPT-4.5. https://openai.com/index/introducing-gpt-4-5/, February 2025b.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library, December 2019.

Perplexity AI. Getting started with Perplexity. https://www.perplexity.ai/hub/blog/getting-started-with-perplexity, 2024.

Qwen, :, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report, 2025. URL https://arxiv.org/abs/2412.15115.

Rafiqul Rabin, Jesse Hostetler, Sean McGregor, Brett Weir, and Nick Judd. Sandboxeval: Towards securing test environment for untrusted code, 2025. URL https://arxiv.org/abs/2504.00018.

Reka. RekaAI/reka-flash-3 · Hugging Face — huggingface.co. https://huggingface.co/RekaAI/reka-flash-3. [Accessed 15-07-2025].

Aymeric Roucher, Albert Villanova del Moral, Thomas Wolf, Leandro von Werra, and Erik Kaunismäki. 'smolagents': a smol library to build great agentic systems. https://github.com/huggingface/smolagents, 2025.

Aditi Singh, Abul Ehtesham, Saket Kumar, and Tala Talaei Khoei. Agentic retrieval-augmented generation: A survey on agentic rag, 2025. URL https://arxiv.org/abs/2501.09136.

Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding, 2023. URL https://arxiv.org/abs/2104.09864.

Gemini Team, Petko Georgiev, Ving Ian Lei, Ryan Burnell, Libin Bai, Anmol Gulati, Garrett Tanzer, Damien Vincent, Zhufeng Pan, Shibo Wang, Soroosh Mariooryad, Yifan Ding, Xinyang Geng, Fred Alcober, Roy Frostig, Mark Omernick, Lexi Walker, Cosmin Paduraru, Christina Sorokin, Andrea Tacchetti, Colin Gaffney, Samira Daruki, Olcan Sercinoglu, Zach Gleicher, Juliette Love, Paul Voigtlaender, Rohan Jain, Gabriela Surita, Kareem Mohamed, Rory Blevins, Junwhan Ahn, Tao Zhu, Kornraphop Kawintiranon, Orhan Firat, Yiming Gu, Yujing Zhang, Matthew Rahtz, Manaal Faruqui, Natalie Clay, Justin Gilmer, JD Co-Reyes, Ivo Penchev, Rui Zhu, Nobuyuki Morioka, Kevin Hui, Krishna Haridasan, Victor Campos, Mahdis Mahdieh, Mandy Guo, Samer Hassan, Kevin Kilgour, Arpi Vezer, Heng-Tze Cheng, Raoul de Liedekerke, Siddharth Goyal, Paul Barham, DJ Strouse, Seb Noury, Jonas Adler, Mukund Sundararajan, Sharad Vikram, Dmitry Lepikhin, Michela Paganini, Xavier Garcia, Fan Yang, Dasha Valter, Maja Trebacz, Kiran Vodrahalli, Chulayuth Asawaroengchai, Roman Ring, Norbert Kalb, Livio Baldini Soares, Siddhartha Brahma,

David Steiner, Tianhe Yu, Fabian Mentzer, Antoine He, Lucas Gonzalez, Bibo Xu, Raphael Lopez Kaufman, Laurent El Shafey, Junhyuk Oh, Tom Hennigan, George van den Driessche, Seth Odoom, Mario Lucic, Becca Roelofs, Sid Lall, Amit Marathe, Betty Chan, Santiago Ontanon, Luheng He, Denis Teplyashin, Jonathan Lai, Phil Crone, Bogdan Damoc, Lewis Ho, Sebastian Riedel, Karel Lenc, Chih-Kuan Yeh, Aakanksha Chowdhery, Yang Xu, Mehran Kazemi, Ehsan Amid, Anastasia Petrushkina, Kevin Swersky, Ali Khodaei, Gowoon Chen, Chris Larkin, Mario Pinto, Geng Yan, Adria Puigdomenech Badia, Piyush Patil, Steven Hansen, Dave Orr, Sebastien M. R. Arnold, Jordan Grimstad, Andrew Dai, Sholto Douglas, Rishika Sinha, Vikas Yadav, Xi Chen, Elena Gribovskaya, Jacob Austin, Jeffrey Zhao, Kaushal Patel, Paul Komarek, Sophia Austin, Sebastian Borgeaud, Linda Friso, Abhimanyu Goyal, Ben Caine, Kris Cao, Da-Woon Chung, Matthew Lamm, Gabe Barth-Maron, Thais Kagohara, Kate Olszewska, Mia Chen, Kaushik Shivakumar, Rishabh Agarwal, Harshal Godhia, Ravi Rajwar, Javier Snaider, Xerxes Dotiwalla, Yuan Liu, Aditya Barua, Victor Ungureanu, Yuan Zhang, Bat-Orgil Batsaikhan, Mateo Wirth, James Qin, Ivo Danihelka, Tulsee Doshi, Martin Chadwick, Jilin Chen, Sanil Jain, Quoc Le, Arjun Kar, Madhu Gurumurthy, Cheng Li, Ruoxin Sang, Fangyu Liu, Lampros Lamprou, Rich Munoz, Nathan Lintz, Harsh Mehta, Heidi Howard, Malcolm Reynolds, Lora Aroyo, Quan Wang, Lorenzo Blanco, Albin Cassirer, Jordan Griffith, Dipanjan Das, Stephan Lee, Jakub Sygnowski, Zach Fisher, James Besley, Richard Powell, Zafarali Ahmed, Dominik Paulus, David Reitter, Zalan Borsos, Rishabh Joshi, Aedan Pope, Steven Hand, Vittorio Selo, Vihan Jain, Nikhil Sethi, Megha Goel, Takaki Makino, Rhys May, Zhen Yang, Johan Schalkwyk, Christina Butterfield, Anja Hauth, Alex Goldin, Will Hawkins, Evan Senter, Sergey Brin, Oliver Woodman, Marvin Ritter, Eric Noland, Minh Giang, Vijay Bolina, Lisa Lee, Tim Blyth, Ian Mackinnon, Machel Reid, Obaid Sarvana, David Silver, Alexander Chen, Lily Wang, Loren Maggiore, Oscar Chang, Nithya Attaluri, Gregory Thornton, Chung-Cheng Chiu, Oskar Bunyan, Nir Levine, Timothy Chung, Evgenii Eltyshev, Xiance Si, Timothy Lillicrap, Demetra Brady, Vaibhav Aggarwal, Boxi Wu, Yuanzhong Xu, Ross McIlroy, Kartikeya Badola, Paramjit Sandhu, Erica Moreira, Wojciech Stokowiec, Ross Hemsley, Dong Li, Alex Tudor, Pranav Shyam, Elahe Rahimtoroghi, Salem Haykal, Pablo Sprechmann, Xiang Zhou, Diana Mincu, Yujia Li, Ravi Addanki, Kalpesh Krishna, Xiao Wu, Alexandre Frechette, Matan Eyal, Allan Dafoe, Dave Lacey, Jay Whang, Thi Avrahami, Ye Zhang, Emanuel Taropa, Hanzhao Lin, Daniel Toyama, Eliza Rutherford, Motoki Sano, HyunJeong Choe, Alex Tomala, Chalence Safranek-Shrader, Nora Kassner, Mantas Pajarskas, Matt Harvey, Sean Sechrist, Meire Fortunato, Christina Lyu, Gamaleldin Elsayed, Chenkai Kuang, James Lottes, Eric Chu, Chao Jia, Chih-Wei Chen, Peter Humphreys, Kate Baumli, Connie Tao, Rajkumar Samuel, Cicero Nogueira dos Santos, Anders Andreassen, Nemanja Rakićević, Dominik Grewe, Aviral Kumar, Stephanie Winkler, Jonathan Caton, Andrew Brock, Sid Dalmia, Hannah Sheahan, Iain Barr, Yingjie Miao, Paul Natsev, Jacob Devlin, Feryal Behbahani, Flavien Prost, Yanhua Sun, Artiom Myaskovsky, Thanumalayan Sankaranarayana Pillai, Dan Hurt, Angeliki Lazaridou, Xi Xiong, Ce Zheng, Fabio Pardo, Xiaowei Li, Dan Horgan, Joe Stanton, Moran Ambar, Fei Xia, Alejandro Lince, Mingqiu Wang, Basil Mustafa, Albert Webson, Hyo Lee, Rohan Anil, Martin Wicke, Timothy Dozat, Abhishek Sinha, Enrique Piqueras, Elahe Dabir, Shyam Upadhyay, Anudhyan Boral, Lisa Anne Hendricks, Corey Fry, Josip Djolonga, Yi Su, Jake Walker, Jane Labanowski, Ronny Huang, Vedant Misra, Jeremy Chen, RJ Skerry-Ryan, Avi Singh, Shruti Rijhwani, Dian Yu, Alex Castro-Ros, Beer Changpinyo, Romina Datta, Sumit Bagri, Arnar Mar Hrafnkelsson, Marcello Maggioni, Daniel Zheng, Yury Sulsky, Shaobo Hou, Tom Le Paine, Antoine Yang, Jason Riesa, Dominika Rogozinska, Dror Marcus, Dalia El Badawy, Qiao Zhang, Luyu Wang, Helen Miller, Jeremy Greer, Lars Lowe Sjos, Azade Nova, Heiga Zen, Rahma Chaabouni, Mihaela Rosca, Jiepu Jiang, Charlie Chen, Ruibo Liu, Tara Sainath, Maxim Krikun, Alex Polozov, Jean-Baptiste Lespiau, Josh Newlan, Zeyncep Cankara, Soo Kwak, Yunhan Xu, Phil Chen, Andy Coenen, Clemens Meyer, Katerina Tsihlas, Ada Ma, Juraj Gottweis, Jinwei Xing, Chenjie Gu, Jin Miao, Christian Frank, Zeynep Cankara, Sanjay Ganapathy, Ishita Dasgupta, Steph Hughes-Fitt, Heng Chen, David Reid, Keran Rong, Hongmin Fan, Joost van Amersfoort, Vincent Zhuang, Aaron Cohen, Shixiang Shane Gu, Anhad Mohananey, Anastasija Ilic, Taylor Tobin, John Wieting, Anna Bortsova, Phoebe Thacker, Emma Wang, Emily Caveness, Justin Chiu, Eren Sezener, Alex Kaskasoli, Steven Baker, Katie Millican, Mohamed Elhawaty, Kostas Aisopos, Carl Lebsack, Nathan Byrd, Hanjun Dai, Wenhao Jia, Matthew Wiethoff, Elnaz Davoodi, Albert Weston, Lakshman Yagati, Arun Ahuja, Isabel Gao, Golan Pundak, Susan Zhang, Michael Azzam, Khe Chai Sim, Sergi Caelles, James Keeling, Abhanshu Sharma, Andy Swing, YaGuang Li, Chenxi Liu, Carrie Grimes Bostock, Yamini Bansal, Zachary Nado, Ankesh Anand, Josh Lipschultz, Abhijit Karmarkar, Lev Proleev, Abe Ittycheriah, Soheil Hassas Yeganeh, George Polovets, Aleksandra Faust, Jiao

Sun, Alban Rrustemi, Pen Li, Rakesh Shivanna, Jeremiah Liu, Chris Welty, Federico Lebron, Anirudh Baddepudi, Sebastian Krause, Emilio Parisotto, Radu Soricut, Zheng Xu, Dawn Bloxwich, Melvin Johnson, Behnam Neyshabur, Justin Mao-Jones, Renshen Wang, Vinay Ramasesh, Zaheer Abbas, Arthur Guez, Constant Segal, Duc Dung Nguyen, James Svensson, Le Hou, Sarah York, Kieran Milan, Sophie Bridgers, Wiktor Gworek, Marco Tagliasacchi, James Lee-Thorp, Michael Chang, Alexey Guseynov, Ale Jakse Hartman, Michael Kwong, Ruizhe Zhao, Sheleem Kashem, Elizabeth Cole, Antoine Miech, Richard Tanburn, Mary Phuong, Filip Pavetic, Sebastien Cevey, Ramona Comanescu, Richard Ives, Sherry Yang, Cosmo Du, Bo Li, Zizhao Zhang, Mariko Iinuma, Clara Huiyi Hu, Aurko Roy, Shaan Bijwadia, Zhenkai Zhu, Danilo Martins, Rachel Saputro, Anita Gergely, Steven Zheng, Dawei Jia, Ioannis Antonoglou, Adam Sadovsky, Shane Gu, Yingying Bi, Alek Andreev, Sina Samangooei, Mina Khan, Tomas Kocisky, Angelos Filos, Chintu Kumar, Colton Bishop, Adams Yu, Sarah Hodkinson, Sid Mittal, Premal Shah, Alexandre Moufarek, Yong Cheng, Adam Bloniarz, Jaehoon Lee, Pedram Pejman, Paul Michel, Stephen Spencer, Vladimir Feinberg, Xuehan Xiong, Nikolay Savinov, Charlotte Smith, Siamak Shakeri, Dustin Tran, Mary Chesus, Bernd Bohnet, George Tucker, Tamara von Glehn, Carrie Muir, Yiran Mao, Hideto Kazawa, Ambrose Slone, Kedar Soparkar, Disha Shrivastava, James Cobon-Kerr, Michael Sharman, Jay Pavagadhi, Carlos Araya, Karolis Misiunas, Nimesh Ghelani, Michael Laskin, David Barker, Qiujia Li, Anton Briukhov, Neil Houlsby, Mia Glaese, Balaji Lakshminarayanan, Nathan Schucher, Yunhao Tang, Eli Collins, Hyeontaek Lim, Fangxiaoyu Feng, Adria Recasens, Guangda Lai, Alberto Magni, Nicola De Cao, Aditya Siddhant, Zoe Ashwood, Jordi Orbay, Mostafa Dehghani, Jenny Brennan, Yifan He, Kelvin Xu, Yang Gao, Carl Saroufim, James Molloy, Xinyi Wu, Seb Arnold, Solomon Chang, Julian Schrittwieser, Elena Buchatskaya, Soroush Radpour, Martin Polacek, Skye Giordano, Ankur Bapna, Simon Tokumine, Vincent Hellendoorn, Thibault Sottiaux, Sarah Cogan, Aliaksei Severyn, Mohammad Saleh, Shantanu Thakoor, Laurent Shefey, Siyuan Qiao, Meenu Gaba, Shuo yiin Chang, Craig Swanson, Biao Zhang, Benjamin Lee, Paul Kishan Rubenstein, Gan Song, Tom Kwiatkowski, Anna Koop, Ajay Kannan, David Kao, Parker Schuh, Axel Stjerngren, Golnaz Ghiasi, Gena Gibson, Luke Vilnis, Ye Yuan, Felipe Tiengo Ferreira, Aishwarya Kamath, Ted Klimenko, Ken Franko, Kefan Xiao, Indro Bhattacharya, Miteyan Patel, Rui Wang, Alex Morris, Robin Strudel, Vivek Sharma, Peter Choy, Sayed Hadi Hashemi, Jessica Landon, Mara Finkelstein, Priya Jhakra, Justin Frye, Megan Barnes, Matthew Mauger, Dennis Daun, Khuslen Baatarsukh, Matthew Tung, Wael Farhan, Henryk Michalewski, Fabio Viola, Felix de Chaumont Quitry, Charline Le Lan, Tom Hudson, Qingze Wang, Felix Fischer, Ivy Zheng, Elspeth White, Anca Dragan, Jean baptiste Alayrac, Eric Ni, Alexander Pritzel, Adam Iwanicki, Michael Isard, Anna Bulanova, Lukas Zilka, Ethan Dyer, Devendra Sachan, Srivatsan Srinivasan, Hannah Muckenhirn, Honglong Cai, Amol Mandhane, Mukarram Tariq, Jack W. Rae, Gary Wang, Kareem Ayoub, Nicholas FitzGerald, Yao Zhao, Woohyun Han, Chris Alberti, Dan Garrette, Kashyap Krishnakumar, Mai Gimenez, Anselm Levskaya, Daniel Sohn, Josip Matak, Inaki Iturrate, Michael B. Chang, Jackie Xiang, Yuan Cao, Nishant Ranka, Geoff Brown, Adrian Hutter, Vahab Mirrokni, Nanxin Chen, Kaisheng Yao, Zoltan Egyed, Francois Galilee, Tyler Liechty, Praveen Kallakuri, Evan Palmer, Sanjay Ghemawat, Jasmine Liu, David Tao, Chloe Thornton, Tim Green, Mimi Jasarevic, Sharon Lin, Victor Cotruta, Yi-Xuan Tan, Noah Fiedel, Hongkun Yu, Ed Chi, Alexander Neitz, Jens Heitkaemper, Anu Sinha, Denny Zhou, Yi Sun, Charbel Kaed, Brice Hulse, Swaroop Mishra, Maria Georgaki, Sneha Kudugunta, Clement Farabet, Izhak Shafran, Daniel Vlasic, Anton Tsitsulin, Rajagopal Ananthanarayanan, Alen Carin, Guolong Su, Pei Sun, Shashank V, Gabriel Carvajal, Josef Broder, Iulia Comsa, Alena Repina, William Wong, Warren Weilun Chen, Peter Hawkins, Egor Filonov, Lucia Loher, Christoph Hirnschall, Weiyi Wang, Jingchen Ye, Andrea Burns, Hardie Cate, Diana Gage Wright, Federico Piccinini, Lei Zhang, Chu-Cheng Lin, Ionel Gog, Yana Kulizhskaya, Ashwin Sreevatsa, Shuang Song, Luis C. Cobo, Anand Iyer, Chetan Tekur, Guillermo Garrido, Zhuyun Xiao, Rupert Kemp, Huaixiu Steven Zheng, Hui Li, Ananth Agarwal, Christel Ngani, Kati Goshvadi, Rebeca Santamaria-Fernandez, Wojciech Fica, Xinyun Chen, Chris Gorgolewski, Sean Sun, Roopal Garg, Xinyu Ye, S. M. Ali Eslami, Nan Hua, Jon Simon, Pratik Joshi, Yelin Kim, Ian Tenney, Sahitya Potluri, Lam Nguyen Thiet, Quan Yuan, Florian Luisier, Alexandra Chronopoulou, Salvatore Scellato, Praveen Srinivasan, Minmin Chen, Vinod Koverkathu, Valentin Dalibard, Yaming Xu, Brennan Saeta, Keith Anderson, Thibault Sellam, Nick Fernando, Fantine Huot, Junehyuk Jung, Mani Varadarajan, Michael Quinn, Amit Raul, Maigo Le, Ruslan Habalov, Jon Clark, Komal Jalan, Kalesha Bullard, Achintya Singhal, Thang Luong, Boyu Wang, Sujeevan Rajayogam, Julian Eisenschlos, Johnson Jia, Daniel Finchelstein, Alex Yakubovich, Daniel Balle, Michael Fink, Sameer Agarwal, Jing Li, Dj Dvijotham, Shalini Pal, Kai Kang, Jaclyn Konzelmann, Jennifer Beattie, Olivier Dousse, Diane Wu, Remi Crocker,

Chen Elkind, Siddhartha Reddy Jonnalagadda, Jong Lee, Dan Holtmann-Rice, Krystal Kallarackal, Rosanne Liu, Denis Vnukov, Neera Vats, Luca Invernizzi, Mohsen Jafari, Huanjie Zhou, Lilly Taylor, Jennifer Prendki, Marcus Wu, Tom Eccles, Tianqi Liu, Kavya Kopparapu, Francoise Beaufays, Christof Angermueller, Andreea Marzoca, Shourya Sarcar, Hilal Dib, Jeff Stanway, Frank Perbet, Nejc Trdin, Rachel Sterneck, Andrey Khorlin, Dinghua Li, Xihui Wu, Sonam Goenka, David Madras, Sasha Goldshtein, Willi Gierke, Tong Zhou, Yaxin Liu, Yannie Liang, Anais White, Yunjie Li, Shreya Singh, Sanaz Bahargam, Mark Epstein, Sujoy Basu, Li Lao, Adnan Ozturel, Carl Crous, Alex Zhai, Han Lu, Zora Tung, Neeraj Gaur, Alanna Walton, Lucas Dixon, Ming Zhang, Amir Globerson, Grant Uy, Andrew Bolt, Olivia Wiles, Milad Nasr, Ilia Shumailov, Marco Selvi, Francesco Piccinno, Ricardo Aguilar, Sara McCarthy, Misha Khalman, Mrinal Shukla, Vlado Galic, John Carpenter, Kevin Villela, Haibin Zhang, Harry Richardson, James Martens, Matko Bosnjak, Shreyas Rammohan Belle, Jeff Seibert, Mahmoud Alnahlawi, Brian McWilliams, Sankalp Singh, Annie Louis, Wen Ding, Dan Popovici, Lenin Simicich, Laura Knight, Pulkit Mehta, Nishesh Gupta, Chongyang Shi, Saaber Fatehi, Jovana Mitrovic, Alex Grills, Joseph Pagadora, Tsendsuren Munkhdalai, Dessie Petrova, Danielle Eisenbud, Zhishuai Zhang, Damion Yates, Bhavishya Mittal, Nilesh Tripuraneni, Yannis Assael, Thomas Brovelli, Prateek Jain, Mihajlo Velimirovic, Canfer Akbulut, Jiaqi Mu, Wolfgang Macherey, Ravin Kumar, Jun Xu, Haroon Qureshi, Gheorghe Comanici, Jeremy Wiesner, Zhitao Gong, Anton Ruddock, Matthias Bauer, Nick Felt, Anirudh GP, Anurag Arnab, Dustin Zelle, Jonas Rothfuss, Bill Rosgen, Ashish Shenoy, Bryan Seybold, Xinjian Li, Jayaram Mudigonda, Goker Erdogan, Jiawei Xia, Jiri Simsa, Andrea Michi, Yi Yao, Christopher Yew, Steven Kan, Isaac Caswell, Carey Radebaugh, Andre Elisseeff, Pedro Valenzuela, Kay McKinney, Kim Paterson, Albert Cui, Eri Latorre-Chimoto, Solomon Kim, William Zeng, Ken Durden, Priya Ponnapalli, Tiberiu Sosea, Christopher A. Choquette-Choo, James Manyika, Brona Robenek, Harsha Vashisht, Sebastien Pereira, Hoi Lam, Marko Velic, Denese Owusu-Afriyie, Katherine Lee, Tolga Bolukbasi, Alicia Parrish, Shawn Lu, Jane Park, Balaji Venkatraman, Alice Talbert, Lambert Rosique, Yuchung Cheng, Andrei Sozanschi, Adam Paszke, Praveen Kumar, Jessica Austin, Lu Li, Khalid Salama, Bartek Perz, Wooyeol Kim, Nandita Dukkipati, Anthony Baryshnikov, Christos Kaplanis, XiangHai Sheng, Yuri Chervonyi, Caglar Unlu, Diego de Las Casas, Harry Askham, Kathryn Tunyasuvunakool, Felix Gimeno, Siim Poder, Chester Kwak, Matt Miecnikowski, Vahab Mirrokni, Alek Dimitriev, Aaron Parisi, Dangyi Liu, Tomy Tsai, Toby Shevlane, Christina Kouridi, Drew Garmon, Adrian Goedeckemeyer, Adam R. Brown, Anitha Vijayakumar, Ali Elqursh, Sadegh Jazayeri, Jin Huang, Sara Mc Carthy, Jay Hoover, Lucy Kim, Sandeep Kumar, Wei Chen, Courtney Biles, Garrett Bingham, Evan Rosen, Lisa Wang, Qijun Tan, David Engel, Francesco Pongetti, Dario de Cesare, Dongseong Hwang, Lily Yu, Jennifer Pullman, Srini Narayanan, Kyle Levin, Siddharth Gopal, Megan Li, Asaf Aharoni, Trieu Trinh, Jessica Lo, Norman Casagrande, Roopali Vij, Loic Matthey, Bramandia Ramadhana, Austin Matthews, CJ Carey, Matthew Johnson, Kremena Goranova, Rohin Shah, Shereen Ashraf, Kingshuk Dasgupta, Rasmus Larsen, Yicheng Wang, Manish Reddy Vuyyuru, Chong Jiang, Joana Ijazi, Kazuki Osawa, Celine Smith, Ramya Sree Boppana, Taylan Bilal, Yuma Koizumi, Ying Xu, Yasemin Altun, Nir Shabat, Ben Bariach, Alex Korchemniy, Kiam Choo, Olaf Ronneberger, Chimezie Iwuanyanwu, Shubin Zhao, David Soergel, Cho-Jui Hsieh, Irene Cai, Shariq Iqbal, Martin Sundermeyer, Zhe Chen, Elie Bursztein, Chaitanya Malaviya, Fadi Biadsy, Prakash Shroff, Inderjit Dhillon, Tejasi Latkar, Chris Dyer, Hannah Forbes, Massimo Nicosia, Vitaly Nikolaev, Somer Greene, Marin Georgiev, Pidong Wang, Nina Martin, Hanie Sedghi, John Zhang, Praseem Banzal, Doug Fritz, Vikram Rao, Xuezhi Wang, Jiageng Zhang, Viorica Patraucean, Dayou Du, Igor Mordatch, Ivan Jurin, Lewis Liu, Ayush Dubey, Abhi Mohan, Janek Nowakowski, Vlad-Doru Ion, Nan Wei, Reiko Tojo, Maria Abi Raad, Drew A. Hudson, Vaishakh Keshava, Shubham Agrawal, Kevin Ramirez, Zhichun Wu, Hoang Nguyen, Ji Liu, Madhavi Sewak, Bryce Petrini, DongHyun Choi, Ivan Philips, Ziyue Wang, Ioana Bica, Ankush Garg, Jarek Wilkiewicz, Priyanka Agrawal, Xiaowei Li, Danhao Guo, Emily Xue, Naseer Shaik, Andrew Leach, Sadh MNM Khan, Julia Wiesinger, Sammy Jerome, Abhishek Chakladar, Alek Wenjiao Wang, Tina Ornduff, Folake Abu, Alireza Ghaffarkhah, Marcus Wainwright, Mario Cortes, Frederick Liu, Joshua Maynez, Andreas Terzis, Pouya Samangouei, Riham Mansour, Tomasz Kępa, François-Xavier Aubet, Anton Algymr, Dan Banica, Agoston Weisz, Andras Orban, Alexandre Senges, Ewa Andrejczuk, Mark Geller, Niccolo Dal Santo, Valentin Anklin, Majd Al Merey, Martin Baeuml, Trevor Strohman, Junwen Bai, Slav Petrov, Yonghui Wu, Demis Hassabis, Koray Kavukcuoglu, Jeff Dean, and Oriol Vinyals. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context, 2024. URL https://arxiv.org/abs/2403.05530.

The pandas development team. pandas-dev/pandas: Pandas, February 2020. URL https://doi.org/10.5281/zenodo.3509134.

Pauli Virtanen, Ralf Gommers, Travis E Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, et al. Scipy 1.0: fundamental algorithms for scientific computing in python. *Nature methods*, 17(3):261–272, 2020.

Chaozheng Wang, Shuzheng Gao, Cuiyun Gao, Wenxuan Wang, Chun Yong Chong, Shan Gao, and Michael R. Lyu. A systematic evaluation of large code models in api suggestion: When, which, and how, 2024a. URL https://arxiv.org/abs/2409.13178.

Chong Wang, Kaifeng Huang, Jian Zhang, Yebo Feng, Lyuye Zhang, Yang Liu, and Xin Peng. How and Why LLMs Use Deprecated APIs in Code Completion? an Empirical Study. *arXiv preprint arXiv:2312.14617*, 2024b.

Chong Wang, Kaifeng Huang, Jian Zhang, Yebo Feng, Lyuye Zhang, Yang Liu, and Xin Peng. LLMs Meet Library Evolution: Evaluating Deprecated API Usage in LLM-based Code Completion . In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 781–781, Los Alamitos, CA, USA, May 2025a. IEEE Computer Society. doi: 10.1109/ICSE55347.2025.00245. URL https://doi.ieeecomputersociety.org/10.1109/ICSE55347.2025.00245.

Chong Wang, Kaifeng Huang, Jian Zhang, Yebo Feng, Lyuye Zhang, Yang Liu, and Xin Peng. Llms meet library evolution: Evaluating deprecated api usage in llm-based code completion, 2025b. URL https://arxiv.org/abs/2406.09834.

Xingyao Wang. Introducing openhands lm 32b – a strong, open coding agent model. *All Hands AI Blog*, March 2025. URL https://www.all-hands.dev/blog/introducing-openhands-lm-32b----a-strong-open-coding-agent-model.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023. URL https://arxiv.org/abs/2201.11903.

Tongtong Wu, Weigang Wu, Xingyu Wang, Kang Xu, Suyu Ma, Bo Jiang, Ping Yang, Zhenchang Xing, Yuan-Fang Li, and Gholamreza Haffari. VersiCode: Towards version-controllable code generation. June 2024.

xAI. Grok-3. Official xAI announcement, 2025. URL https://x.ai/news/grok-3. Accessed May 17, 2025.

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

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

Sixiang Ye, Zeyu Sun, Guoqing Wang, Liwei Guo, Qingyuan Liang, Zheng Li, and Yong Liu. Prompt alchemy: Automatic prompt refinement for enhancing code generation, 2025. URL https://arxiv.org/abs/2503.11085.

Shuyan Zhou, Uri Alon, Frank F Xu, Zhiruo Wang, Zhengbao Jiang, and Graham Neubig. DocPrompting: Generating code by retrieving the docs. July 2022.

Library	The software library under test.
Library Version	The exact version of that library.
Task Description	A problem centered on a particular library change.
Initial Code	The Python snippet provided as a starting point.
Extra Dependencies	Any additional packages required to solve the task.
Hidden Tests	Comprehensive unit tests designed to maximize coverage. The success rate on these is the benchmark metric.
Visible Test	A concise test that validates the specific target behavior, intended to be used for Self-Debugging experiments.
Reference Solution	A correct, ground-truth implementation.
Reference Documents	A set of version-specific reference documents, to be used for RAG experiments.

Table 4: Problem column definitions for the **GitChameleon 2.0** dataset.

A Benchmark Details

This appendix provides additional details on the **GitChameleon 2.0** benchmark. We provide details on the dataset construction process, the structure of the dataset samples, on the processes for validating the examples and constructing the hidden tests, and finally present additional statistics regarding the dataset.

A.1 Dataset Construction Process

The examples were created by the authors, which took roughly 350 human hours. To construct that dataset, we compiled a list of popular Python libraries, focusing on those that had more than 1000 stars on Github as well as detailed documentation of changes between versions. For each library, we reviewed the change logs to identify breaking changes: deprecated functions, argument changes, alterations in behavior, and newly introduced functions.

For each identified change, we wrote a concise problem statement, starter code, expected solution and a suite of tests, consisting of a comprehensive suite of hidden tests to be used for model performance evaluation and ranking and a manually written concise visible test to be used for self-debugging experiments. We also added a ground-truth set of relevant documents for RAG experiments.

NOTE: Low-level changes—such as backend optimizations that do not alter the surface-level API—are not considered valid changes for our benchmark. For example, if between `Torch 1.7` and `Torch 1.8` the `torch.nn.Softmax()` function received a CUDA-based numerical stability improvement, this does not modify the API usage of `Softmax()` and is therefore not labeled as a change in our benchmark. Since most changes in mature libraries primarily impact backend functionality, collecting 328 valid samples required significant effort.

A.2 Structure of Dataset Samples

The main fields of each sample are given in Table 4. Additionally, each problem in **GitChameleon 2.0** is associated with metadata to assist in the analysis of the results, as described in Table 5. Each problem is classified with a type of API evolution change among the categories defined in Table 6.

A.3 Dataset Validation

To ensure the validity of the dataset examples, we followed the following process: First, we created a clean Docker container for each problem and installed the required dependencies into it. Then, we executed the visible and hidden validation tests to ensure that all are successful.

Change Category	The type of library-evolution changes, as defined in table 6.
Target Entity	The specific function or class under test.
Solution Style	"Functional" if only a function body is expected, or "Full" for a general code completion.
Web Framework Task	"Yes" if the problem exercises a web-development framework, otherwise "No."

Table 5: Metadata column definitions.

Change Category	Description
Argument or Attribute change	The API call to a function, method, or class has a change in arguments (e.g. name, order, new, deprecated argument) between versions.
Function Name change	The name of the API call has changed between versions (e.g. `pandas.append` to `pandas.concat`).
Semantics or Function Behavior change	The semantic / runtime behavior of the API call changed between versions (e.g. returning a different type).
New feature or additional dependency-based change	A feature was introduced in a specific version; therefore, to execute the same functionality, a model using an older version should make use of an additional dependency (e.g. `torch.special` was introduced in TORCH 1.10, previously one could use NUMPY for the same).

Table 6: Categories of API Evolution Changes

A.4 Hidden Test Construction

This section presents how we generated the hidden tests for each dataset example. These tests were generated by instructing the `Zencoder AI Coding Agent` [4] to create test files for each example, incorporating the appropriate dependency versions. The Zencoder agent, built on the GPT-4.1 base model, operated with internet search enabled and was granted execution access, allowing it to self-correct outputs that initially failed during runtime. Further errors encountered during verification were resolved by supplying error traces back to Zencoder or through an isolated instance of GPT-4o, supplemented with manual inspection and intervention where necessary. This process enabled us to construct a robust and comprehensive test suite, achieving a coverage of **96.5%**. The decision to use ZENCODER was motivated by limitations observed in alternative unit test generation approaches. Rule-based generators such as Pynguin Lukasczyk and Fraser [2022] fail to account for version differences among samples that share the same or similar problem statements. Meanwhile, AI-based unit test generators like Claude Code and EarlyAI[5] were not suitable: the former typically generated test classes where each sub-function was populated only with `pass()` statements, while the latter was restricted to functional-style problems and could not handle the more complex, class-based structures prevalent in **GitChameleon 2.0**.

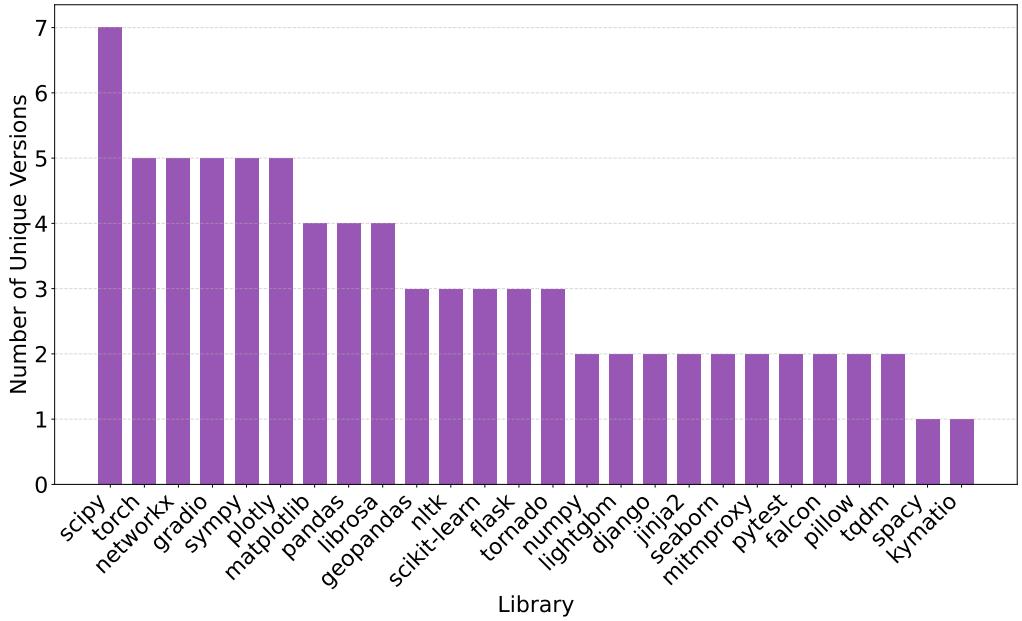
A.5 Additional Dataset Statistics

Figure 7 presents the number of unique versions per library and the number of samples per library.

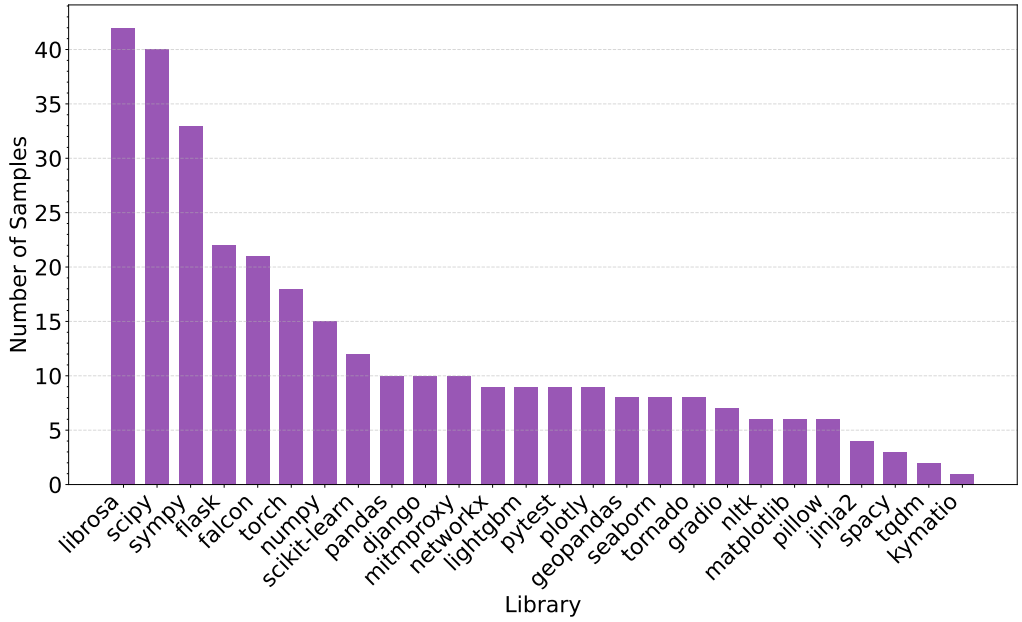
B Extra Methodologies: Reasoning, Sampling and Prompting

This section presents results from additional experimental methodologies:

[4] https://zencoder.ai
[5] https://www.startearly.ai/



(a) Number of unique versions per library.



(b) Number of samples per library.

Figure 7: Dataset library statistics. (a) The count of distinct versions identified for each library, presented in decreasing order of uniqueness. (b) The total frequency of samples containing each library, ordered by their occurrence count.

- **Temperature Sampling:** Results are shown in Table 8. We evaluate sampling at temperature $T = 0.8$ across 10 seeds using both the OpenAI and Gemini model suites. The performance difference compared to greedy decoding is minimal.

- **Reasoning Models:** Performance results for the OpenAI o-series reasoning models are provided in Table 7.
- **Self-Explained Keywords (SEK) Prompting:** We evaluate the SEK prompting method proposed by Fan et al. [2024], applied to both OpenAI and Gemini models. SEK involves a two-stage process: (1) *Keyword Extraction*, where the model generates relevant keywords for the coding task, and (2) *Keyword Categorization*, where keywords are ranked and classified into (a) Function, (b) General, and (c) Abstract categories. TF-IDF ranking is performed using a 50,000-document subset of the EVOL-CODEALPACA-V1 corpus Luo et al. [2023]. As shown in our empirical analysis, SEK does not yield significant improvements over greedy sampling, and in several cases underperforms relative to it. NOTE: Temperature $T = 0$ is used in both stages of SEK prompting.

Model	Vanilla Decoding			Vanilla with Self-Debug			Zero-shot CoT	
	Success Rate (%)		API Hit Rate (%)	Success Rate (%)		API Hit Rate (%)	Success Rate (%)	API Hit Rate (%)
	Hidden	Visible		Hidden	Visible		Hidden	
o1	**51.2**$_{\pm 2.8}$	**60.1**$_{\pm 2.7}$	42.1$_{\pm 2.7}$	57.6$_{\pm 2.7}$	68.6$_{\pm 2.6}$	**49.2**$_{\pm 2.8}$	41.2$_{\pm 2.7}$	**41.3**$_{\pm 2.7}$
o3-mini	44.5$_{\pm 2.7}$	52.7$_{\pm 2.8}$	40.6$_{\pm 2.7}$	**66.8**$_{\pm 2.6}$	**76.5**$_{\pm 2.3}$	45.7$_{\pm 2.8}$	**50.9**$_{\pm 2.8}$	40.7$_{\pm 2.7}$
o4-mini	48.2$_{\pm 2.8}$	57.0$_{\pm 2.7}$	**48.3**$_{\pm 2.8}$	63.1$_{\pm 2.7}$	75.0$_{\pm 2.4}$	45.4$_{\pm 2.7}$	–	–
codex-mini	48.5$_{\pm 2.8}$	58.2$_{\pm 2.7}$	47.5$_{\pm 2.8}$	–	–	–	32.0$_{\pm 2.6}$	37.9$_{\pm 2.7}$

Table 7: Success rate on visible and hidden tests and API hit rate under the Vanilla, Self-Debug, and Zero-shot CoT settings, for the OpenAI o-series models. Model ranking on the benchmark is determined by **Hidden Success Rate**. Visible Success Rate figures are for context on Self-Debugging. The best result in each column is in bold. For full model details and citations, please refer to Appendix J.

Model	Hidden Success Rate (%)	API Hit Rate (%)
o1	**50.5**$_{\pm 0.8}$	44.0$_{\pm 0.8}$
o3-mini	46.4$_{\pm 1.6}$	42.5$_{\pm 0.6}$
GPT-4.1	48.9$_{\pm 1.4}$	48.1$_{\pm 1.0}$
GPT-4.1-mini	45.9$_{\pm 1.3}$	46.9$_{\pm 0.6}$
GPT-4.1-nano	33.8$_{\pm 1.1}$	43.8$_{\pm 0.8}$
GPT-4o	47.2$_{\pm 1.2}$	45.1$_{\pm 0.9}$
GPT-4o-mini	40.2$_{\pm 1.2}$	41.0$_{\pm 1.1}$
Gemini 1.5 Pro	45.4$_{\pm 1.2}$	45.5$_{\pm 0.7}$
Gemini 2.5 Pro	41.0$_{\pm 3.4}$	**48.3**$_{\pm 1.7}$
Gemini 2.0 Flash	43.4$_{\pm 3.1}$	42.5$_{\pm 0.9}$
Gemini 2.5 Flash	46.4$_{\pm 0.8}$	46.8$_{\pm 1.2}$

Table 8: **Hidden Success Rate using temperature sampling ($T = 0.8$), averaged over 10 seeds.** A comparison to the greedy decoding baseline in Table 1 reveals that the changes in performance between greedy decoding and temperature sampling are mixed. For most models, the differences are small, but for a few specific models, the changes are big and noteworthy. For the majority of models evaluated (8 out of 11), the performance change is minor, typically within +/- 2 percentage points. For example, `Gemini-2.5-pro`, shows a notable decrease in success rate (-9.0 points).

C Extended Experiment Results and Analysis

This section contains the following additional experimental results:

Model	Hidden Success Rate (%)	API Hit Rate (%)
GPT-4o	$29.6_{\pm 2.5}$	$43.6_{\pm 2.7}$
GPT-4o-mini	$27.7_{\pm 2.5}$	$40.3_{\pm 2.7}$
GPT-4.1	$43.6_{\pm 2.7}$	$49.4_{\pm 2.8}$
GPT-4.1-mini	$41.2_{\pm 2.7}$	$44.0_{\pm 2.7}$
GPT-4.1-nano	$32.9_{\pm 2.6}$	$43.8_{\pm 2.7}$
GPT-4.5	$33.8_{\pm 2.6}$	$\mathbf{58.0}_{\pm 2.7}$
Gemini 1.5 Pro	$44.5_{\pm 2.7}$	$45.7_{\pm 2.8}$
Gemini 2.0 Flash	$41.2_{\pm 2.7}$	$43.4_{\pm 2.7}$
Gemini 2.5 Pro	$47.3_{\pm 2.8}$	$50.0_{\pm 2.8}$
Gemini 2.5 Flash	$\mathbf{48.2}_{\pm 2.8}$	$43.4_{\pm 2.7}$

Table 9: **Success and API hit rates under the SEK setting.** While SEK, being a two-round prompting scheme, is expected to outperform greedy decoding, we observe that it does not yield significant improvements. For example, with GPT-4.1, the success rate actually drops by 4.9% when using SEK compared to greedy decoding.

- An experiment on Automatic Prompt Optimization of the system prompt for Greedy Decoding is described in Table 10.
- An experiment on static analysis based generated solutions fixing to ensure model failures are not attributed to confounding factors like indentation problems and unused imports or variable declarations. Refer to Table 13 for further details.
- Table 11 contains an extended set of RAG results, including both additional models and the setting where only a single document is retrieved.
- Table 12 contains a set of results with IDE and CLI coding assistants, such as Claude Code, Goose, and Cline.

We also present the following additional analyses:

- A comparison of success rates between Self-Debug and Greedy Decoding, when broken down by version release year (Figure 8) and by library (Figure 9).
- A comparison of success rates between RAG and Greedy Decoding by library is shown in Figure 10.
- Figure 11 analyzes the intra-model sample agreement rates in the Greedy Decoding, Zero-Shot CoT and RAG settings.

Model	Best Round	Success Rate (%)	Δ (%)
GPT-4.1-mini	1	$42.1_{\pm 2.7}$	–2.1
GPT-4.1-nano	3	$37.5_{\pm 2.7}$	**+3.7**
GPT-4.1	1	$\mathbf{50.0}_{\pm 2.8}$	+1.5
GPT-4o	0	$49.1_{\pm 2.8}$	0.0

Table 10: **Automatic System Prompt Optimization results**. The prompt was optimized for at most 5 rounds using the method described in Ye et al. [2025], with early stopping if the improvement over previous round is less than 1.5%. We used GPT-4.1 as the mutation model and a random fixed 20% subset of the dataset for the optimization process. For the initial prompt, we use the same system prompt that we had used for our Greedy Decoding experiments, as given in Figure 15. We report the delta of the hidden test success rate, in comparison to the Greedy Decoding baseline. The results demonstrate the limited utility of further optimizing the prompts we had used in our experiments.

In addition to evaluating a generic agentic framework endowed with basic tools, we also analyze the performance of specialized AI coding assistant software.

Model	$k=1$		$k=3$				
	Success Rate (%)	API Hit Rate (%)	Success Rate (%)	API Hit Rate (%)	Precision (%)	Recall (%)	MRR
Open-Weights Models							
CommandA	43.6±2.7	43.9±2.7	**48.2**±2.8	45.4±2.7	**41.9**±2.7	**50.7**±2.8	**0.63**±0.03
CommandR 7B	23.2±2.3	36.3±2.7	23.2±2.3	35.6±2.6	41.6±2.7	50.4±2.8	0.62±0.03
Deepseek R1	**50.9**±2.8	44.8±2.7	51.2±2.8	**47.9**±2.8	41.5±2.7	50.1±2.8	0.62±0.03
Reka Flash-3	8.5±1.5	34.5±2.6	11.6±1.8	31.9±2.6	29.9±2.5	39.6±2.8	0.47±0.03
Jamba 1.6 Mini	18.0±2.1	35.4±2.6	29.3±2.5	40.4±2.7	41.6±2.7	50.1±2.8	0.62±0.03
OpenHands LM 32B v0.1	34.8±2.6	41.0±2.7	28.9±2.5	36.5±2.7	25.9±2.4	33.7±2.7	0.42±0.03
Llama 4 Scout	38.7±2.7	**45.1**±2.7	39.3±2.7	43.6±2.7	41.3±2.7	50.4±2.8	0.62±0.03
Enterprise Models							
Arcee CoderL	46.3±2.8	47.3±2.8	36.6±2.7	40.4±2.7	31.1±2.6	41.0±2.8	0.49±0.03
Claude 3.5 Haiku	43.6±2.7	47.9±2.8	43.0±2.7	47.5±2.8	41.9±2.7	50.7±2.8	0.62±0.03
Claude 3.5 Sonnet	8.5±1.5	18.6±2.1	49.4±2.8	51.5±2.8	41.9±2.7	50.7±2.8	0.62±0.03
Codestral	44.2±2.7	47.3±2.8	46.0±2.8	48.5±2.8	41.9±2.7	50.7±2.8	0.62±0.03
CommandR+	32.0±2.6	43.0±2.7	36.6±2.7	41.9±2.7	41.6±2.7	50.4±2.8	0.62±0.03
Gemini 2.5 Flash	**54.3**±2.8	**50.5**±2.8	**55.2**±2.8	**51.2**±2.8	**41.9**±2.7	**50.7**±2.8	**0.62**±0.03
GPT-4.1-mini	46.9±2.8	50.0±2.8	48.8±2.8	50.0±2.8	41.3±2.7	50.4±2.8	0.62±0.03
GPT-4.1-nano	38.1±2.7	45.1±2.7	37.8±2.7	45.0±2.7	41.3±2.7	50.4±2.8	0.62±0.03
GPT-4o-mini	41.5±2.8	45.4±2.7	43.3±2.8	46.8±2.8	41.0±2.7	50.1±2.8	0.62±0.03
GPT-4o	48.2±2.8	47.0±2.7	52.1±2.8	49.4±2.8	40.6±2.7	49.5±2.8	0.61±0.03
Inflection 3 Productivity	24.7±2.8	42.0±2.6	21.9±2.7	44.2±2.7	41.9±2.7	50.7±2.8	0.62±0.03
LFM 40B MoE	30.8±2.7	38.3±2.7	20.7±2.7	34.0±2.7	33.8±2.7	44.8±2.8	0.53±0.03

Table 11: RAG performance of additional models when retrieving $k = 1$ and $k = 3$ most relevant documents. Precision is shown only for $k = 3$ as it is equivalent to Recall in the $k = 1$ case. This table shows that retrieving three documents is better in almost all cases than retrieving a single document, despite the incurred false positives that arise due to most of the examples having less than three relevant documents.

For this setting, we examine both Command-Line Interface (CLI), such as `Claude Code`[6] coding assistants and Integrated Development Environment (IDE) coding assistants, such as `Cline`[7].

The input to the assistants is given as a Python file which consists of the required library, version and extra dependencies as in-line comments and subsequently the starter code. NOTE: All assistants had internet and terminal commands execution access.

We had furthermore ablated this setting versus giving the full problem statement as input. For instruct models, we run the model's parsed output as standalone code, and for base models, the concatenation of the starting code and model's parsed output (completion).

Table 12 presents the success rates of various CLI and IDE assistants on the visible and hidden tests in **GitChameleon 2.0**.

[6] https://docs.anthropic.com/en/docs/claude-code/overview
[7] https://cline.bot/

Name	Model	Success Rate (%)		API Hit Rate (%)	
		No-prob	Prob	No-prob	Prob
CLI Assistants					
Claude Code	Claude 3.7 Sonnet	32.0±2.6	48.8±2.8	**44.2**±2.7	45.5±2.7
Goose	GPT-4o	**36.3**±2.7	36.9±2.7	43.9±2.7	**54.5**±2.7
	GPT-4.1	19.2±2.2	**55.5**±2.7	41.7±2.7	53.0±2.8
IDE Assistants					
Cline	Claude 3.7 Sonnet	32.9±2.6	44.8±2.7	40.5±2.7	50.2±2.8
	GPT-4.1	38.4±2.7	**54.6**±2.7	42.4±2.7	48.8±2.8
	GPT-4.1-mini	27.1±2.5	42.1±2.7	32.9±2.6	**52.4**±2.8
	GPT-4.1-nano	38.1±2.7	54.6±2.7	42.4±2.7	48.8±2.8
	GPT-4o	**41.5**±2.7	–	42.7±2.7	–
Kilocode	Claude 3.7 Sonnet	30.2±2.5	–	**43.3**±2.7	–
Roocode	Claude 3.5 Sonnet	12.5±1.8	–	41.2±2.7	–

Table 12: Success and API-hit rates for CLI and IDE coding assistants, under the setting where the problem statement is given (**Prob**) and where it is not (**No-prob**), in which case we evaluate a scenario akin to tab code-completion. The results show that including the problem statement improves success rate by double-digit margins for 4 out of 5 cases evaluated.

When the problem statement is not given, `Cline` with GPT-4.1 achieves the best result, with a success rate of 38.4%. All assistants besides for `Goose` on GPT-4o demonstrate significant gains, ranging from 12 to 35 points, from including the problem statement.

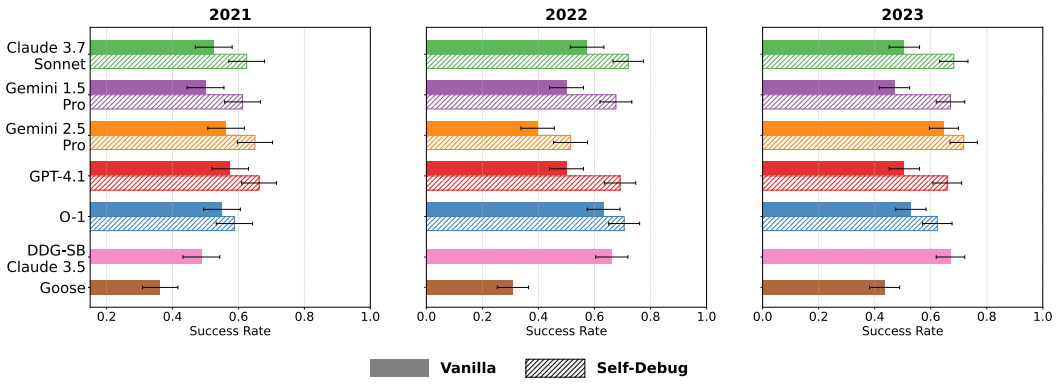


Figure 8: **Success Rate Breakdown by Version Release Year**. Lighter and darker shaded bars represent values obtained with and without Self-Debugging, respectively. Standard error is drawn as a black line. This plot shows that the release year does not significantly impact the results for most evaluated settings.

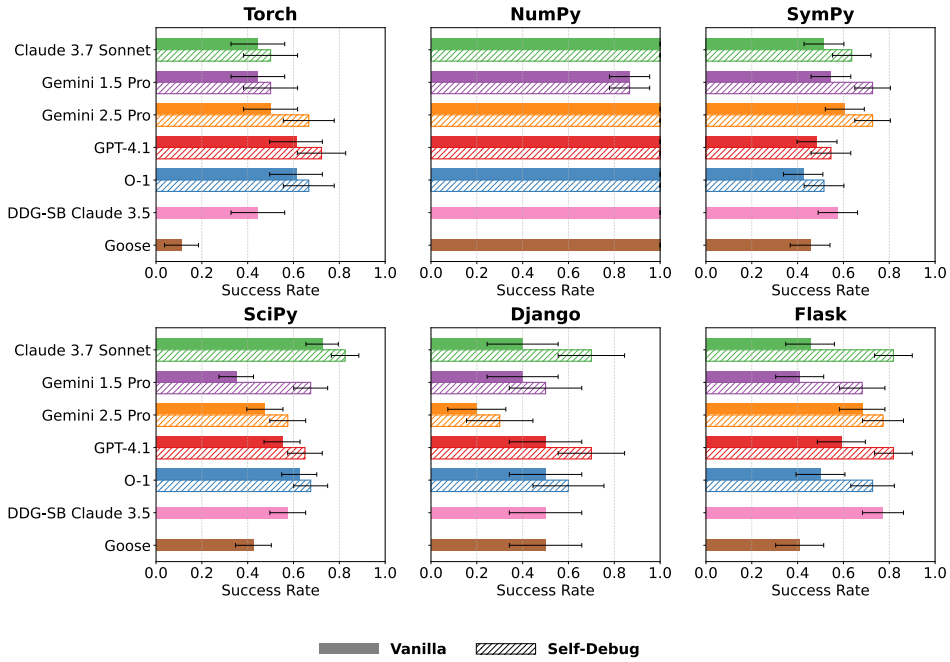
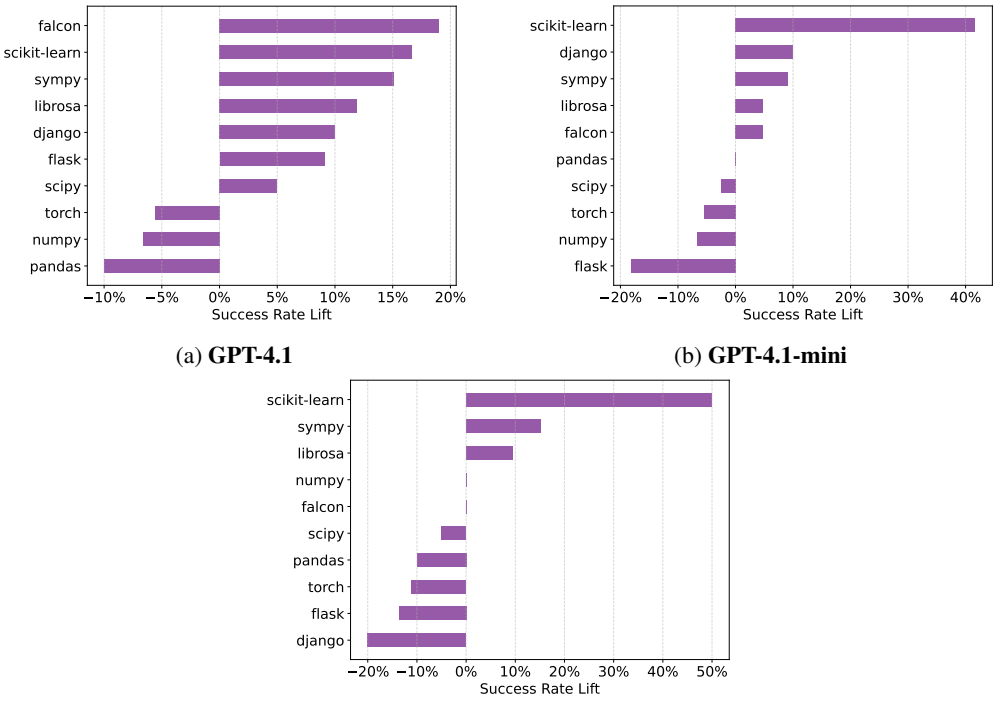


Figure 9: **Success Rate Breakdown by Library**. This figure shows the differences in success rate between the libraries included in **GitChameleon 2.0**. All evaluated settings do very well on NumPy, which is to be expected given the popularity of the library and the subsequent abundance of code that uses it. The success rates on the web development frameworks are notably lower than on the scientific computing libraries, perhaps due to having more complex abstractions.

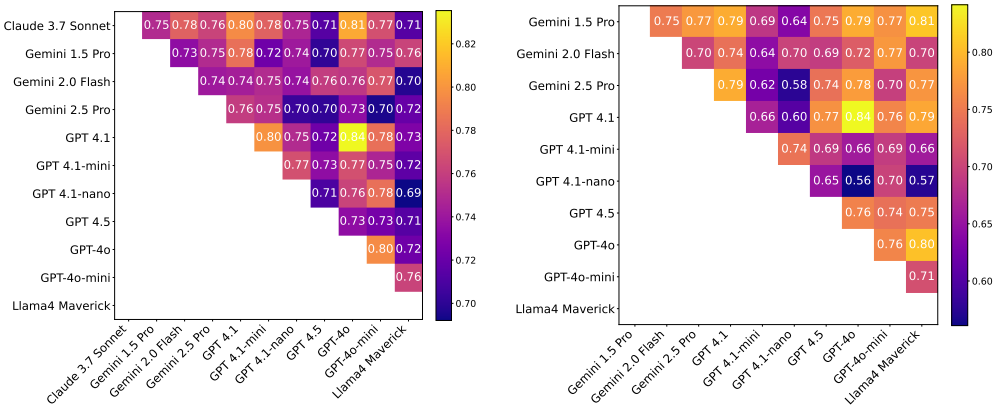


(a) **GPT-4.1**

(b) **GPT-4.1-mini**

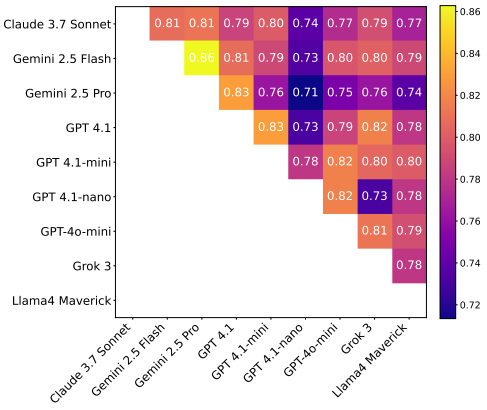
(c) **GPT-4.1-nano**

Figure 10: Δ **Success Rate of RAG over Greedy Decoding, per library**. The 10 most frequent libraries in **GitChameleon 2.0** are shown here. The plots demonstrate a trend where smaller models are less effective at using RAG, with the full-size `GPT-4.1` improving on 7 libraries, the `mini` version improving on 5 and the `nano` version improving only on 3.



(a) **Greedy Decoding**

(b) **Zero-Shot Chain-Of-Thought**



(c) **RAG (k=3)**

Figure 11: **Intra model sample agreement rates**. These plots show the rate of samples that have the same pass/fail result among all pairs of models, under the Greedy Decoding, Zero-Shot CoT and RAG settings. Each cell in these plots represents the agreement rate of a pair of models, with the rate also being color-coded. The high agreement rates in all three subfigures show that ensembling different models would have a limited effect on the success rates.

Assistant	Model	Linter	Pylint Score ↑	Success Rate (%)
Cline (IDE)	GPT-4.1	N/A	1.06	54.6$_{\pm 2.8}$
		Black + Isort	1.69	54.6$_{\pm 2.8}$
		Ruff	**2.64**	54.6$_{\pm 2.8}$
Goose (CLI)	GPT-4o	N/A	0.53	36.3$_{\pm 2.7}$
		Black + Isort	1.82	36.3$_{\pm 2.7}$
		Ruff	**2.92**	36.3$_{\pm 2.7}$
Claude Code (CLI)	Claude 3.7 Sonnet	N/A	0.00	48.8$_{\pm 2.8}$
		Black + Isort	1.92	48.8$_{\pm 2.8}$
		Ruff	**2.60**	48.8$_{\pm 2.8}$

Table 13: **Static Analysis and Auto-linting/Formatting.** Pylint[8] scores are averaged across code samples and are scored out of 10. The success rate numbers presented are the same as in Table 12 wherein Goose has no access to problem statement while Cline and Claude are provided with the same. We observe that the original generated solutions via coding assistants do not meet minimum quality standard requirements, however when improved via auto-linters like Black[9], ISort[10] and Ruff[11], their code quality improves but with no impact to the success rate. This demonstrates that there are no confounding errors like indentation issues, unused imports and other formatting issues influencing our evaluation results observed. NOTE: For Ruff formatting, we used the already formatted/ linted solutions via Black and ISort.

Benchmark	Language	Evaluation Method	Core Task	Source of Changes	Key Difference from GC 2.0
GitChameleon 2.0	Python	Execution-Based	**Generation for a static version:** Writes new code for a specific, often older, library version.	Real, documented historical breaking changes.	*(Baseline for comparison)*
CodeUpdateEval	Python	Execution-Based	**Code Updating**: Modifies existing code to work with a newer library version.	Real-world software update commits.	Focuses on migrating code **forward** to a newer version, not generating for a static one.
JavaVersionGenBench	Java	Execution-Based	**Code Updating**: Modifies existing Java code to handle version updates.	Real-world Java projects.	Focuses on the **Java** ecosystem and its specific language/tooling challenges.
LLM-Deprecated-API	Python	Non-Executable	**Deprecation Fixing**: Identifies and replaces specific deprecated API calls.	A curated list of deprecated APIs.	Uses a **non-executable** evaluation method and has a narrow scope focused only on API deprecation.
LibEvolutionEval	Python	Non-Executable	**Code Completion**: Fills in a missing part of a code snippet based on context.	API documentation and release notes.	Is a **completion-based** task that does not test functional correctness through execution.
RustEvo2	Rust	Execution-Based	**Code Repair**: Fixes existing code that fails to compile after a dependency update.	Real breaking changes from Rust libraries ("crates").	Focuses on the **Rust** ecosystem and a reactive, compiler-error-driven repair task.
CODEMENV	Python	Execution-Based	**Environment Compatibility:** Generates code that is compatible with a complex environment specification.	A broad set of environment configurations.	Has a broader focus on overall environment compatibility, not specifically on historical breaking changes.

Table 14: Detailed comparison of **GitChameleon 2.0** with related benchmarks across several key dimensions, highlighting differences in evaluation methodology, core task, and primary programming language.

D Related Work

This section discusses additional lines of related work and provides a detailed comparison of the differentation of **GitChameleon 2.0** versus previous efforts.

[8] https://pylint.pycqa.org/en/latest/index.html
[9] https://black.readthedocs.io/en/stable/
[10] https://pycqa.github.io/isort/
[11] https://docs.astral.sh/ruff/

D.1 Code Evolution Datasets

While the main text provides a high-level overview of the most similar benchmarks, this section offers a more detailed differentiation between **GitChameleon 2.0** and other relevant works. We categorize these benchmarks based on several key dimensions, including their evaluation method (execution-based vs. non-executable) and, most importantly, their core **task format (instruction-based generation vs. completion- or repair-based tasks)**. This distinction is critical as it tests different capabilities of language models.

D.1.1 Task Format: Instruction-Based Generation

GitChameleon 2.0 is fundamentally an **instruction-based** benchmark. For each problem, the model is given a natural language "Problem Statement" and starter code. The core challenge is to comprehend the user's intent and generate a new, functionally correct solution that adheres to specific version constraints. This tests a model's ability to translate human requirements into code.

D.1.2 Task Format: Code Update, Repair, and Completion

In contrast, many other benchmarks focus on tasks where the primary input is existing code, not a natural language instruction. The model's goal is to modify, repair, or complete a given code snippet.

Code Update and Repair Benchmarks A significant body of work evaluates a model's ability to modify or repair existing code.

- **CodeUpdateArena** [Liu et al., 2025] and **JavaVersionGenBench** [Ciniselli et al., 2024] are code modification benchmarks for Python and Java, respectively. They provide a model with a working piece of code and require it to be updated to a newer library version.
- **RustEvo2** [Liang et al., 2025] is a code repair benchmark for Rust. It provides a model with code that is broken due to a dependency update and asks it to generate a fix based on compiler errors.

These tasks are distinct from **GitChameleon 2.0**'s, as they test a reactive, corrective capability rather than the proactive generation of new code from a specification.

Completion-Based and Non-Executable Benchmarks Another category of benchmarks uses non-executable metrics or focuses on code completion.

- **LibEvolutionEval** [Kuhar et al., 2024] is a non-executable benchmark structured as a "fill-in-the-middle" **completion-based task**. Its evaluation is based on textual similarity metrics (e.g., F1 score), not the functional correctness of the code.
- **LLM-Deprecated-API** [Wang et al., 2025b], which we note in our introduction, focuses on replacing deprecated APIs. This is a specific type of repair task that is evaluated using non-executable string matching.
- **CODEMENV** [Cheng et al., 2025] evaluates a model's ability to generate code compatible with a complex environment specification. While execution-based, its task is primarily driven by satisfying technical constraints rather than implementing a distinct, high-level natural language instruction.

For a detailed breakdown, Table 14 contrasts **GitChameleon 2.0** with these related benchmarks across several key methodological dimensions.

D.2 Specialized Frameworks and Repair Techniques

Recognizing the unique challenges of library evolution, researchers and practitioners are developing specialized frameworks and automated repair techniques that often combine LLMs with other methods.

D.2.1 DepsRAG

This framework utilizes a multi-agent system built around RAG and Knowledge Graphs specifically for reasoning about software dependencies Alhanahnah et al. [2024]. It employs distinct agents managed by an LLM: one to construct and query the dependency KG, another for web searches, and a critic agent to review and refine the generated responses, aiming for higher accuracy in complex dependency analysis tasks.

D.2.2 Dr.Fix

This tool represents a family of approaches using LLMs, often combined with program analysis and RAG, for automated program repair. It focuses on fixing API misuse in LLM-generated code based on the taxonomy of misuse types. It employs a detect-reason-fix pipeline and demonstrates substantial improvements in repair accuracy metrics such as BLEU and Exact Match Behrang et al. [2025].

D.2.3 ReplaceAPI / InsertPrompt

These are lightweight, targeted techniques designed specifically to mitigate the use of deprecated APIs in LLM-based code completion. ReplaceAPI performs a direct, post-generation substitution of known deprecated API calls with their replacements, achieving high fix rates in evaluations Wang et al. [2025b]. InsertPrompt modifies the input prompt to discourage the generation of deprecated APIs in the first place. They serve as valuable baseline approaches for this specific problem Wang et al. [2025a].

D.2.4 Conclusion

These works indicate a trend towards hybrid and agentic systems, moving beyond single LLM calls to more sophisticated architectures that integrate LLMs with other methods for handling library evolution. **GitChameleon 2.0** serves as an essential resource for evaluating such systems.

In the subsequent sections we present qualitative sample analyses and model generation differences.

E Case Study: Code Assistant Failure With Search

Here we present an example where a coding assistant (Zencoder) fails at solving a problem in our benchmark even when it can search the internet for the correct API call. The task required using the `torch.special` module or equivalent `scipy.special` to calculate the the modified Bessel function of the first kind of order 0. This can be computed using `torch.special.i0` or `scipy.special.i0` if the version pre-dates the introduction of `torch.special`.

E.1 Inputs

Problem Statement: Calculate the modified Bessel function of the first kind, order 0 using pytorch's special functions if available in this version, otherwise you may use another library.

Environment

- Python Version: 3.7
- Pytorch Version: 1.10

Starting Code

```
import torch
def bessel_i0(input_tensor: torch.Tensor) -> torch.Tensor:
```

Listing 1: Initial Python function signature provided.

E.2 Model Attempt and Failure

The model's attempt involved manually implementing the modified Bessel function, handling small and large inputs as separate cases.

Assistant Trace

```
[truncated]
**Web search**
Done
Let me check if the torch.special module with Bessel functions is
    specifically available in PyTorch 1.10.0.


**Web search**
Done
The search results dont specifically confirm if torch.special.i0 is
    available in PyTorch 1.10.0. Let me search more specifically.


**Web search**
Done
Since Im not getting clear information about the availability of torch.
    special.i0 in PyTorch 1.10.0, Ill implement the function manually
    using the mathematical definition of the modified Bessel function of
    the first kind of order 0.
Now, lets implement the solution:
```

Listing 2: The assistant's trace while designing the solution.

Proposed Solution

```
def bessel_i0(input_tensor: torch.Tensor) -> torch.Tensor:
    result = torch.zeros_like(input_tensor)
    abs_x = torch.abs(input_tensor)

    # For small values (|x| < 15), use the series expansion
    small_mask = abs_x < 15.0
    small_x = abs_x[small_mask]

    if small_x.numel() > 0:
        small_result = torch.ones_like(small_x)
        term = torch.ones_like(small_x)
        x_squared_over_4 = (small_x / 2.0) ** 2

        # Sum the series until convergence or max iterations
        for k in range(1, 30):  # 30 terms should be enough for good
            precision
            term = term * x_squared_over_4 / (k ** 2)
            small_result = small_result + term

            # Check for convergence
            if torch.all(term < 1e-10 * small_result):
                break

        result[small_mask] = small_result

    # For large values (|x| $\geq$ 15), use the asymptotic approximation
    large_mask = ~small_mask
    large_x = abs_x[large_mask]

    if large_x.numel() > 0:
        large_result = torch.exp(large_x) / torch.sqrt(2 * math.pi *
            large_x)
        result[large_mask] = large_result
```

```
    return result}
```

Listing 3: The solution proposed by the assistant.

Error Traceback:

```
TestBesselI0.test_large_values ____________________________
def test_large_values(self):
    """Test bessel_i0 with large positive and negative values."""
    input_tensor = torch.tensor([10.0, -10.0, 20.0, -20.0], dtype=torch.
        float32)
    result = bessel_i0(input_tensor)
    expected = torch.from_numpy(scipy_i0(input_tensor.numpy()))

>       torch.testing.assert_close(result, expected)
E       AssertionError: Tensor-likes are not close!
```

Listing 4: Error trace for the implementation of `bessel_i0`.

Explanation: The failure in the manual implementation arises because the asymptotic approximation

$$I_0(x) \approx \frac{e^x}{\sqrt{2\pi x}}$$

has only $O(1/x)$ accuracy and omits higher-order correction terms, so for $x \gtrsim 20$ its relative error exceeds the test tolerance of 10^{-5}. To fix this, either use the built-in `torch.special.i0`(x) or include at least the first correction term in the expansion:

$$I_0(x) \approx \frac{e^x}{\sqrt{2\pi x}}\left(1 - \tfrac{1}{8x}\right),$$

which reduces the large-x error below the required threshold.

F Case Study: Self-Debugging in Batched Matrix Exponential Computation

This appendix details an instance where self-debugging helped rectify an initially incorrect solution from GPT-4.5 for computing the matrix exponential of batched matrices. The task required leveraging the `scipy.linalg.expm` function while ensuring compatibility with batched inputs.

F.1 Inputs

Problem Statement: Complete the function that computes the matrix exponential of batched matrices, non specified parameters should use the default value

Environment:

- Python Version: 3.10
- SciPy Version: 1.8.1

Starting Code

```
from scipy import linalg
import numpy as np

def compute_matrix_exponential(A: np.ndarray) -> np.ndarray:
    # Model needs to complete this function
    pass
```

Listing 5: Initial Python function signature provided.

F.2 First Model Attempt and Failure

The model's first attempt involved reshaping the input array and applying `scipy.linalg.expm` directly.

Initial Model Output

```
from scipy import llinalg
import numpy as np

def compute_matrix_exponential(A: np.ndarray) -> np.ndarray:
    return linalg.expm(A)
```

Listing 6: First attempt by the model to define `compute_matrix_exponential`.

Model's Explanation (Initial): *The function directly returns linalg.expm(A) because SciPy's expm computes the matrix exponential for a given matrix. Given that A is a batch of matrices, SciPy should naturally handle this batch input and apply the exponential operation to each matrix independently, producing the correct batched result.*

Result: `Failure`

Error Traceback Failure: Calling `linalg.expm(A)` on a 3D array $A \in \mathbb{R}^{b\times n\times n}$ makes SciPy treat it as one "matrix," and raises

`ValueError: expected a square matrix`

because it only supports 2D inputs.

```
Traceback (most recent call last):
  File "/app/repo/eval_venvs/gcham_venv_111/lib/python3.10/site-packages/scipy/sparse/linalg/_
      matfuncs.py", line 601, in _expm
    raise ValueError('expected a square matrix')
ValueError: expected a square matrix
```

Listing 7: Error traceback from the initial model output.

F.3 Self-Debugging Process and Correction

The self-debugging process recognized that simply calling linalg.expm was insufficient. Instead, `linalg.expm` must be applied individually to each 2D matrix within the batch. Additionally, a crucial pre-condition is that each matrix within the batch must be square.

Second Model Attempt (Post-Debugging) The model regenerated the solution, incorporating input validation and iterating over the individual matrices in the batch.

Regenerated Model Output

```
from scipy import linalg
import numpy as np

def compute_matrix_exponential(A: np.ndarray) -> np.ndarray:
    return np.stack([linalg.expm(A[i]) for i in range(A.shape[0]) if A[i
        ].shape[0] == A[i].shape[1]], axis=0)
```

Listing 8: Corrected `compute_matrix_exponential` after self-debugging.

Model's Explanation (Regenerated): *The corrected solution applies linalg.expm separately to each 2D square matrix in the batch using a list comprehension, then re-stacks the results into a 3D array. This ensures each call to expm receives a valid input, resolving the error.*

Result: `Success`

F.4 Analysis of the Correction

The regenerated output effectively addressed the shortcomings of the initial attempt:

- **Fix:** Apply `expm` to each 2D slice $A_i \in \mathbb{R}^{n \times n}$ and re-stack, ensuring every call gets a proper square matrix.

This revised approach correctly computes the matrix exponential for each matrix in the batch and handles the data dimensions appropriately, thus resolving the `ValueError` and fulfilling the problem requirements.

G Qualitative Analysis

This section presents additional qualitative studies of attempted solutions in the Greedy Decoding and the Chain-of-Thought settings.

G.1 Greedy Decoding

G.1.1 Example 1: (PyTorch)

Problem Statement: Calculate the logarithm of the cumulative distribution function of the standard normal distribution using available functions. If not available in PyTorch, use another library.

Environment:

- Python Version: 3.7
- Torch Version: 1.9.0
- Extra Dependencies: scipy==1.7.3, numpy==1.21.6

Starting Code

```
import torch
def log_ndtr(input_tensor: torch.Tensor) -> torch.Tensor:
    pass
```

Listing 9: Greedy Example 1 Starting Code

Solution

```
import numpy as np
    from scipy.stats import norm
    output = torch.from_numpy(norm.logcdf(input_tensor.numpy()))
    return output
```

Listing 10: Greedy Example 1 Solution

Claude 3.7 Failure Output

```
import torch
import math
from torch.distributions import Normal

def log_ndtr(input_tensor: torch.Tensor) -> torch.Tensor:
    normal = Normal(0, 1)
    return normal.cdf(input_tensor).log()
```

Listing 11: Claude 3.7 Output

Gemini 2.5 Pro Failure Output

```
import torch
from torch.distributions import Normal

def log_ndtr(input_tensor: torch.Tensor) -> torch.Tensor:
  loc = torch.tensor(0.0, device=input_tensor.device, dtype=input_tensor.
      dtype)
```

```
    scale = torch.tensor(1.0, device=input_tensor.device, dtype=input_
        tensor.dtype)

    standard_normal = Normal(loc=loc, scale=scale)

    log_cdf_values = standard_normal.log_cdf(input_tensor)

    return log_cdf_values
```

Listing 12: Gemini 2.5 Pro Output

Analysis of Model Outputs

Both Claude 3.7 and Gemini 2.5 Pro incorrectly attempt to compute the log-CDF of the standard normal distribution using PyTorch's `Normal` distribution. Claude 3.7 computes `normal.cdf(input_tensor).log()`, which is numerically unstable for large negative inputs since the CDF approaches zero and its logarithm can underflow to $-\infty$, leading to invalid results. Gemini 2.5 Pro mistakenly calls a non-existent `log_cdf` method on the PyTorch `Normal` object, causing an immediate `AttributeError` in PyTorch 1.9. This reflects a confusion between PyTorch and other libraries (such as SciPy or TensorFlow Probability) that do provide this method. In summary, neither model provides a robust or correct implementation; the only reliable approach in this environment is to use `scipy.stats.norm.logcdf` as shown in the reference solution, or to implement a numerically stable approximation in PyTorch when SciPy is unavailable.

G.1.2 Greedy Example 2 (SciPy)

Problem Statement: complete the following function that check if all the batch of matrices are invertible, using numpy 1.25.1.

Environment:

- Python Version: 3.10
- Scipy Version: 1.11.1

Starting Code

```
import warnings
from scipy.linalg import det
import numpy as np
warnings.filterwarnings('error')

def check_invertibility(matrices: np.ndarray) -> np.bool_:
    pass
```

Listing 13: Greedy Example 3 Starting Code

Solution

```
    return np.all(det(matrices))
```

Listing 14: Greedy Example 3 Solution

Gemini 2.5 Correct Output

```
import warnings
from scipy.linalg import det
import numpy as np
warnings.filterwarnings('error')

def check_invertibility(matrices: np.ndarray) -> np.bool_:
    try:
        # Check if matrices is a batch of matrices
        if matrices.ndim < 3:
            # If it's a single matrix, add a batch dimension
```

```
            matrices = np.expand_dims(matrices, axis=0)

        # Check if all matrices are square
        batch_size, rows, cols = matrices.shape
        if rows != cols:
            return np.bool_(False)

        # Calculate determinants for all matrices in the batch
        determinants = np.array([det(matrix) for matrix in matrices])

        # A matrix is invertible if its determinant is not zero
        # Check if all determinants are non-zero
        return np.bool_(np.all(determinants != 0))

    except Exception:
        # If any error occurs during computation, return False
        return np.bool_(False)
```

Listing 15: Gemini 2.5 Pro Output

GPT 4.1 Failure Output

```
import warnings
from scipy.linalg import det
import numpy as np
warnings.filterwarnings('error')

def check_invertibility(matrices: np.ndarray) -> np.bool_:
    for matrix in matrices:
        try:
            d = det(matrix)
            if np.isclose(d, 0):
                return np.bool_(False)
        except Warning:
            return np.bool_(False)
    return np.bool_(True)
```

Listing 16: GPT 4.1 Output

Analysis of Model Outputs

Gemini 2.5 Pro produces a robust solution that handles both single matrices and batches, verifies that matrices are square, computes the determinant for each matrix, and correctly checks if all determinants are non-zero to determine invertibility, returning `False` on any exception to ensure safe failure. In contrast, GPT 4.1's output contains a subtle but critical error in exception handling: the code attempts to catch `Warning` as an exception, but Python treats warnings and exceptions differently, so this approach is ineffective—especially since `warnings.filterwarnings('error')` escalates warnings to exceptions that should be caught as `Exception`. As a result, GPT 4.1's function may fail or produce unpredictable results in the presence of numerical issues or shape mismatches, despite using the correct logic for checking determinant values.

G.1.3 Greedy Example 3 (SymPy)

Problem Statement: Write a custom jacobi symbols function that compute the Jacobi symbol.

Environment:

- Python Version: 3.9
- Sympy Version: 1.13

Starting Code

```
import sympy
```

```python
def custom_jacobi_symbols(a: int, n: int) -> int:
    pass
```

Listing 17: Greedy Example 4 Starting Code

Solution

```python
    return sympy.jacobi_symbol(a, n)
```

Listing 18: Greedy Example 4 Solution

GPT 4.1 Correct Output

```python
import sympy

def custom_jacobi_symbols(a: int, n: int) -> int:
    return sympy.ntheory.jacobi_symbol(a, n)
```

Listing 19: GPT 4.1 Output

O1 Failure Output

```python
import sympy

def custom_jacobi_symbols(a: int, n: int) -> int:
    return sympy.ntheory.residues.jacobi_symbol(a, n)
```

Listing 20: O1 Output

Analysis of Model Outputs

GPT-4.1 provides the correct solution by calling `sympy.ntheory.jacobi_symbol`, which is the proper and documented API for computing the Jacobi symbol in SymPy 1.13. In contrast, O1 incorrectly uses `sympy.ntheory.residues.jacobi_symbol`, assuming that the function is nested in the `residues` submodule; however, in this version of SymPy, `jacobi_symbol` is not found there, resulting in an `AttributeError` on all tests. This error highlights a lack of up-to-date API knowledge or reliance on outdated documentation.

G.1.4 Greedy Example 4 (Flask)

Problem Statement: Complete the app set-up so that, when given a batch of matrix, the json encoding compute the determinants of each matrix, before flattening and converting the result to a list, we are using scipy 1.11.1. Do not run the app in your code.

Environment:

- Python Version: 3.10
- Flask Version: 3.0.0
- Extra Dependencoes: scipy==1.11.1

Starting Code

```python
import flask
import numpy as np
from scipy import linalg

app = flask.Flask('test1')
@app.route('/data')
def data(num_list):
    return flask.jsonify({'numbers': num_list})
def eval_app(app, data_fn, num_arr):
    with app.test_request_context():
        response = data_fn(num_arr)
```

```
        return response.get_data(as_text=True)

class MyCustomJSONHandler(flask.json.provider.DefaultJSONProvider):
    def default(self, obj):
        if isinstance(obj, np.ndarray) and len(obj.shape)==3 and obj.
            shape[-1]==obj.shape[-2] :
            pass
```

Listing 21: Greedy Example 5 Starting Code

Solution

```
    res = linalg.det(obj)
                return res.tolist()
            return super().default(obj)

    app.json_provider_class = MyCustomJSONHandler
    app.json = app.json_provider_class(app)
```

Listing 22: Greedy Example 5 Solution

Gemini 2.5 Pro Correct Output

```
import flask
import numpy as np
from scipy import linalg

app = flask.Flask('test1')
@app.route('/data')
def data(num_list):
    return flask.jsonify({'numbers': num_list})
def eval_app(app, data_fn, num_arr):
    with app.test_request_context():
        response = data_fn(num_arr)
        return response.get_data(as_text=True)

class MyCustomJSONHandler(flask.json.provider.DefaultJSONProvider):
    def default(self, obj):
        if isinstance(obj, np.ndarray) and len(obj.shape)==3 and obj.
            shape[-1]==obj.shape[-2] :
            # Calculate determinants for each matrix in the batch
            determinants = linalg.det(obj)
            # Convert the resulting 1D array of determinants to a list
            return determinants.tolist()
        # Fallback to the default serialization for other types
        return super().default(obj)

# Apply the custom JSON provider to the app
app.json = MyCustomJSONHandler(app)
```

Listing 23: Gemini 2.5 Pro Output

Claude 3.7 Failure Output

```
import flask
import json

app = flask.Flask('test')
@app.route('/data')
def data(num_set):
    return flask.jsonify({'numbers': num_set})

def eval(app, data_fn, num_set):
    with app.test_request_context():
        response = data_fn(num_set)
```

```
        return response.get_data(as_text=False)

def app_set_up(app: flask.Flask) -> None:
    def set_encoder(obj):
        if isinstance(obj, set):
            return sorted(list(obj))
        raise TypeError(f"Object of type {type(obj)} is not JSON
            serializable")

    app.json.encoder = lambda: json.JSONEncoder(default=set_encoder)
```

Listing 24: Claude 3.7 Output

Analysis of Model Outputs

Gemini 2.5 Pro provides a solution that correctly subclasses `flask.json.provider.DefaultJSONProvider` and overrides the `default` method to check for batches of square matrices, compute their determinants using `scipy.linalg.det`, and serialize the result as a list, fully aligning with both the requirements and Flask 3.0.0's JSON provider API. In contrast, Claude 3.7's output incorrectly tries to customize JSON serialization by modifying `app.json.encoder`—a property that is not available in Flask 3.0.0—and focuses on encoding sets rather than NumPy arrays, neglecting the determinant computation altogether. Consequently, Claude's approach fails with an `AttributeError` and does not satisfy the task requirements even if the encoder logic were correct, while Gemini's implementation is robust and functional for the specified environment.

G.2 Zero-Shot Chain-Of-Thought

G.2.1 CoT Example 1 (Torch)

Problem Statement: Calculate the modified Bessel function of the first kind, order 1 using PyTorch's special functions if available in this version, otherwise you may use another library.

Environment:

- Python Version: 3.7
- Torch Version: 1.9.0
- Extra Dependencies: scipy==1.7.3, numpy==1.21.6

Starting Code

```
import torch
def bessel_i1(input_tensor: torch.Tensor) -> torch.Tensor:
    pass
```

Listing 25: CoT Example 1 Starting Code

Solution

```
    import numpy as np
    from scipy.special import i1 as scipy_i1
    output = torch.from_numpy(scipy_i1(input_tensor.numpy()))
    return output
```

Listing 26: CoT Example 1 Solution

Claude 3.7 Correct Output

```
import torch
from scipy import special

def bessel_i1(input_tensor: torch.Tensor) -> torch.Tensor:
    # Convert PyTorch tensor to NumPy array
```

```
    input_numpy = input_tensor.cpu().numpy()

    # Apply the modified Bessel function of the first kind, order 1
    result_numpy = special.i1(input_numpy)

    # Convert back to PyTorch tensor
    result_tensor = torch.from_numpy(result_numpy).to(input_tensor.device
        )

    # Ensure the output has the same dtype as the input
    result_tensor = result_tensor.to(dtype=input_tensor.dtype)

    return result_tensor
```

Listing 27: Claude 3.7 Output

Gemini 2.5 Pro Failure Output

```
import torch
def bessel_i1(input_tensor: torch.Tensor) -> torch.Tensor:
    """Calculates the modified Bessel function of the first kind, order
        1.

    Args:
        input_tensor: The input tensor.

    Returns:
        A tensor containing the element-wise modified Bessel function of
            the first kind, order 1.
    """
    # Check if torch.special.i1 is available (it is in torch 1.9.0)
    if hasattr(torch.special, 'i1'):
        return torch.special.i1(input_tensor)
    else:
        # Fallback or error handling if needed for other versions
        # For torch 1.9.0, this else block should not be reached.
        raise NotImplementedError("torch.special.i1 is not available in
            this PyTorch version.")
```

Listing 28: Gemini 2.5 Pro Output

Analysis of Model Outputs

Claude 3.7 provides a robust solution by correctly identifying that `torch.special.i1` is not available in PyTorch 1.9.0 and instead leveraging `scipy.special.i1` to compute the modified Bessel function, converting the result back to a PyTorch tensor and ensuring proper device and dtype handling; this approach is fully compatible with the given environment and passes all tests. In contrast, Gemini 2.5 Pro mistakenly assumes that `torch.special.i1` exists in PyTorch 1.9.0, checks for its presence using `hasattr`, and raises a `NotImplementedError` when unavailable; however, since this function is absent in the specified version, the code fails every test due to the raised exception.

G.2.2 CoT Example 2 (Scikit-learn)

Problem Statement: Adapt the use of ‘manhattan distances‘ to obtain a pairwise distance matrix.

Environment:

- Python Version: 3.10
- Scikit-learn Version: 1.1
- Extra Dependencies: numpy==1.23.5

Starting Code

```
from sklearn.metrics.pairwise import manhattan_distances
import numpy as np
def get_pairwise_dist(X: np.ndarray,Y: np.ndarray) -> np.ndarray:
    distances = manhattan_distances(X, Y, sum_over_features=False)
    pass
```

Listing 29: CoT Example 2 Starting Code

Solution

```
    return np.sum(distances, axis=1)
```

Listing 30: CoT Example 2 Solution

GPT 4.1 Correct Output

```
from sklearn.metrics.pairwise import manhattan_distances
import numpy as np
def get_pairwise_dist(X: np.ndarray, Y: np.ndarray) -> np.ndarray:
    distances = manhattan_distances(X, Y)
    return distances
```

Listing 31: GPT 4,1 Output

Claude 3.7 Failure Output

```
from sklearn.metrics.pairwise import manhattan_distances
import numpy as np
def get_pairwise_dist(X: np.ndarray,Y: np.ndarray) -> np.ndarray:
    distances = manhattan_distances(X, Y, sum_over_features=False)
    return np.sum(distances, axis=2)
```

Listing 32: Claude 3.7 Pro Output

Analysis of Model Outputs

In this example, GPT 4.1 provides a correct implementation by calling `manhattan_distances(X, Y)` without the `sum_over_features` argument, which by default returns the desired pairwise distance matrix of shape $(n_{\text{samples}_X}, n_{\text{samples}_Y})$. In contrast, Claude 3.7 incorrectly sets `sum_over_features=False`, causing the output to be a 3D array of shape, and then applies `np.sum(distances, axis=2)`, which does not match the problem's intent and produces incorrect results if further summed incorrectly. The misuse of the `sum_over_features` parameter and the erroneous axis for reduction demonstrates a misunderstanding of the scikit-learn API and leads to a failure, while GPT 4.1's output is concise and correct for all valid input shapes.

G.2.3 CoT Example 3 (Falcon)

Problem Statement: Create a class named CustomRouter to manage your application's routes. The class should maintain an internal dictionary named routes for storing the mapping between URI templates and their associated resources. Implement an add_route method that accepts three arguments: a URI template, a resource, and additional keyword arguments. This method should use Falcon's routing utility—specifically, the map_http_methods function from the falcon.routing module—to generate a mapping of HTTP methods to resource handlers. If a 'fallback' parameter is provided in the keyword arguments, it should be passed to map_http_methods. The method should then store a tuple consisting of the resource and the generated method mapping in the routes dictionary, using the URI template as the key, and finally return the generated method mapping. You must not modify the provided starting code, which includes the empty CustomRouter class. Instead, complete the solution by dynamically adding the required method in the solution() function.

Environment:

- Python Version: 3.10

- Falcon Version: 2.0

Starting Code

```
class CustomRouter:
    def __init__(self):
        self.routes = {}


def solution() -> None:

    def add_route(
```

Listing 33: CoT Example 3 Starting Code

Solution

```
self, uri_template, resource, **kwargs):
        from falcon.routing import map_http_methods
        method_map = map_http_methods(resource, kwargs.get('fallback',
            None))
        self.routes[uri_template] = (resource, method_map)
        return method_map

    CustomRouter.add_route = add_route
```

Listing 34: CoT Example 3 Solution

O1 Correct Output

```
import falcon
from falcon.routing import map_http_methods

class CustomRouter:
    def __init__(self):
        self.routes = {}

def solution() -> None:
    def add_route(self, uri_template, resource, **kwargs):
        fallback = kwargs.pop('fallback', None)
        if fallback is not None:
            method_map = map_http_methods(resource, fallback=fallback)
        else:
            method_map = map_http_methods(resource)
        self.routes[uri_template] = (resource, method_map)
        return method_map
    CustomRouter.add_route = add_route
```

Listing 35: O1 Output

GPT 4.1 Failure Output

```
from falcon.routing import map_http_methods

def solution() -> None:
    def add_route(self, uri_template, resource, **kwargs):
        fallback = kwargs.pop('fallback', None)
        if fallback is not None:
            method_map = map_http_methods(resource, fallback=fallback)
        else:
            method_map = map_http_methods(resource)
        self.routes[uri_template] = (resource, method_map)
        return method_map

    CustomRouter.add_route = add_route
```

Listing 36: GPT 4.1 Output

Analysis of Model Outputs

In this case, the O1 output provides a fully correct solution by dynamically attaching an `add_route` method to the `CustomRouter` class, properly handling the optional `fallback` argument, and updating the `routes` dictionary according to the problem statement. The method is inserted via the `solution()` function without modifying the provided class definition, and it uses Falcon's `map_http_methods` utility to construct the method mapping as required. In contrast, the GPT 4.1 output omits the explicit definition of the `CustomRouter` class in its solution, violating the requirement to use the existing starting code. Although the logic within the `solution()` function is correct, the absence of a `CustomRouter` definition in the completed module would lead to a `NameError` or otherwise prevent the expected dynamic method attachment. The critical distinction is that O1 respects all constraints including not modifying the class definition directly, while GPT 4.1 provides an incomplete module, failing to meet the initialization requirements set by the problem.

H Logic vs. Knowledge Retention

The goal of our proposed benchmark, **GitChameleon**, is to evaluate a model's ability to retain version-specific knowledge—specifically, whether it can recall the functionalities associated with particular library versions it has been trained on. Notably, this capability is distinct from the ability to generate logically correct code. While we do not explicitly disentangle whether model failures on our evaluation suite stem from incorrect logic generation or incorrect API version usage, our benchmark is intentionally designed so that most problems primarily test knowledge retention rather than complex logic reasoning. For each problem in our dataset, we compute the number of logic-related nodes in the Abstract Syntax Tree (AST) of the ground-truth solution and present their distribution in Figure 12. As shown, most ground-truth solutions contain fewer than **five** logic-related AST nodes. This supports our claim that the benchmark is primarily designed to assess version-specific knowledge retention rather than complex logic-based code generation.

Table 15: Criteria for classifying AST nodes as logic-related.

Condition	Classification
Calling a user-defined function	✓
Calling built-in Python operators (e.g., +)	✓
Calling a math or utility function with non-obvious purpose	✓
Calling a library method (e.g., `torch.from_numpy`)	✗
Composing multiple calls together	✓

The criteria for classifying AST nodes as logic-related are provided in Table 15, and we include visualizations of the ASTs for two example ground-truth solutions for further illustration in Figures 13 and 14 respectively.

1. Sample ID: 0, Logic Nodes: 3

```
import torch
def log_ndtr(input_tensor: torch.Tensor) -> torch.Tensor:
    import numpy as np
    from scipy.stats import norm
    output = torch.from_numpy(norm.logcdf(input_tensor.numpy()))
    return output
```

Listing 37: Sample 0 Ground Truth Solution

2. Sample ID: 329, Logic Nodes: 0

```
import matplotlib.pyplot as plt
def use_seaborn() -> None:
    plt.style.use("seaborn")
```

Listing 38: Sample 329 Ground Truth Solution

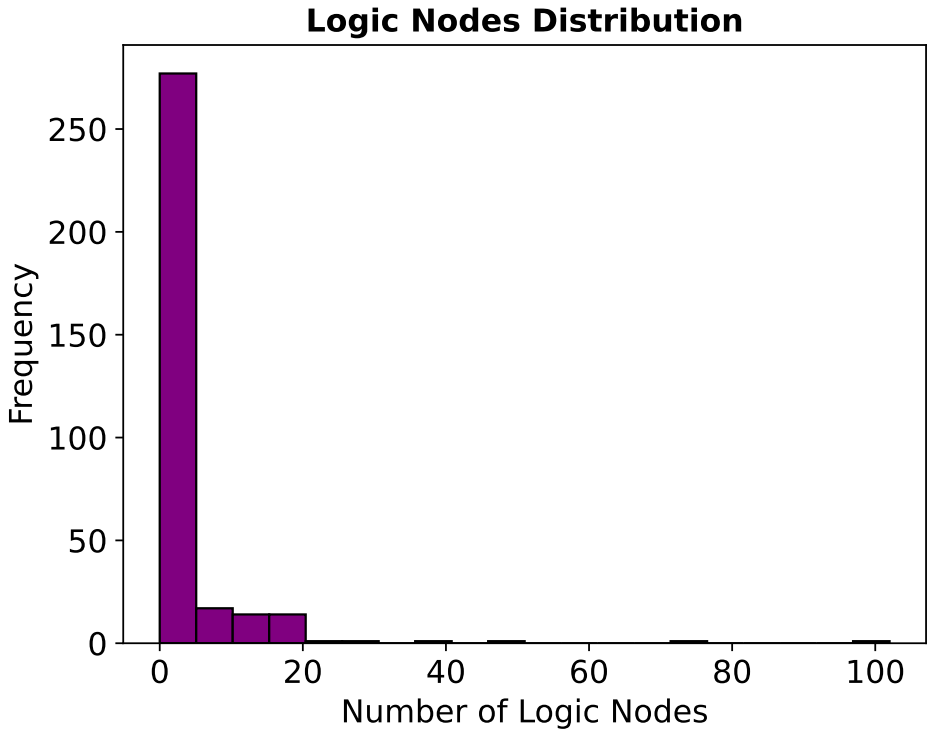


Figure 12: **Logic Nodes Distribution over samples' ground truth solutions' ASTs.** Most ground truth solutions have less than **five** logic nodes.

I Prompt Templates

This appendix contains all the prompts we had used for our experiments:

- The prompts for greedy sampling are given in Figure 15.
- The prompts for self-debugging are given in Figure 16.
- The prompt for the multi-step agent is given in Figure 17.
- The prompt for RAG is given in Figure 18.
- The prompt and file format for Coding Assistants are given in Figure 19.
- The prompt for SEK is given in Figure 20 (for keywords generation) and Figure 21 (for code generation).

J Artifacts and Model Details

This appendix provides citations for various artifacts and models mentioned in the paper.

J.1 Libraries

This is the full list of libraries included in **GitChameleon 2.0**.

- `PyTorch` [Paszke et al., 2019]

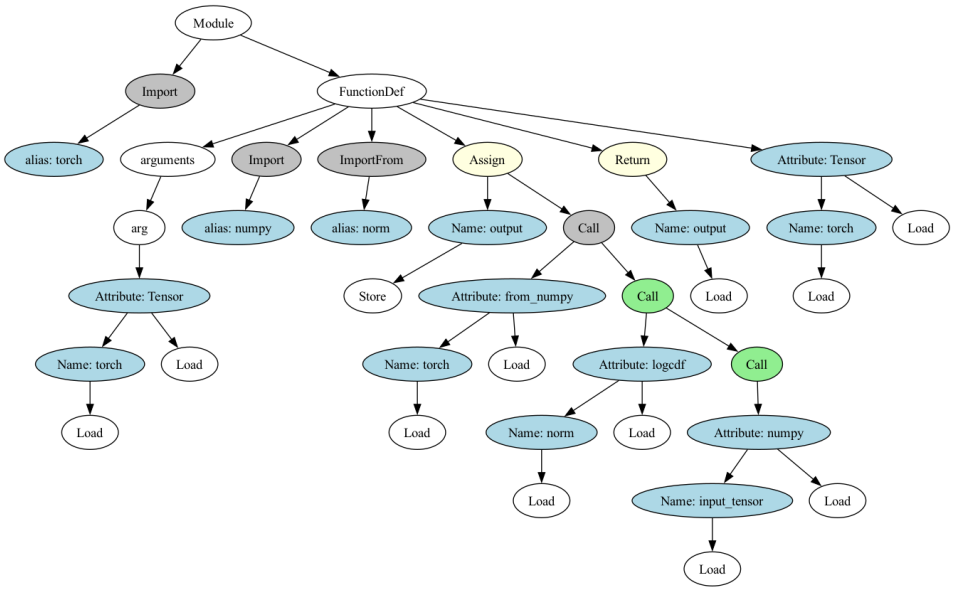


Figure 13: AST visualization for the ground-truth solution of Sample ID 0. The three color-coded `call` nodes (in grey and green) represent the logic-related components, classified under the "composing multiple calls together" category. The corresponding ground-truth code is shown in Code block 37 for reference.

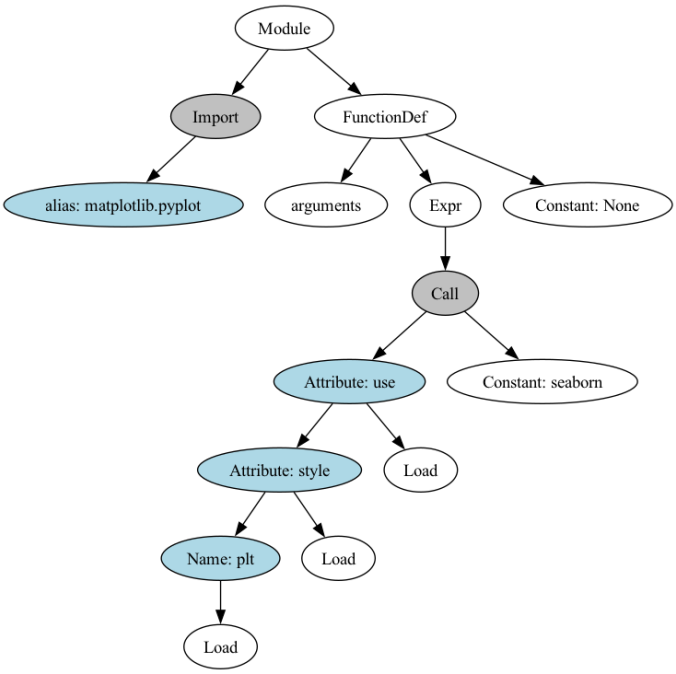


Figure 14: AST visualization for the ground-truth solution of Sample ID 329. No logic nodes are present, as the only `call` node corresponds to the "calling a library method" category. The ground-truth solution is provided for reference in Code block 38.

- `Geopandas` [Jordahl et al., 2020]

Figure 15: Prompts for Greedy Sampling

(a) System Prompt for Zero-Shot Prompting

````
You are a skilled Python
    programmer tasked with
    solving a coding problem.
    Your goal is to provide a
    clear, efficient, and correct
     solution that meets all the
    specified requirements.

Please provide your solution
    following these guidelines:

1. Use the required library in
    your solution.
2. Incorporate the provided
    starter code correctly.
3. Write your solution in Python.
4. Format your solution within a
    markdown code block.
5. Ensure your code is clean,
    efficient, and well-commented
    .
6. Output only the code block and
     nothing else.

Example output format:

```python
# [Your code here, incorporating
    the starter code]

# [Additional code and comments
    as needed]
```

After writing your solution,
    please review it to ensure
    all requirements are met and
    the code is correct and
    efficient.

Here are the key elements for
    this task:
````

(b) System Prompt for Chain-Of-Thought Prompting

````
You are a skilled Python
    programmer tasked with
    solving a coding problem.
    Your goal is to provide a
    clear, efficient, and correct
     solution that meets all the
    specified requirements.

First, let's think step-by-step.
    Then, please provide your
    solution following these
    guidelines:

1. Use the required library in
    your solution.
2. Incorporate the provided
    starter code correctly.
3. Write your solution in Python.
4. Format your solution within a
    markdown code block.
5. Ensure your code is clean,
    efficient, and well-commented
    .
6. Output nothing else after the
    code block.


Example output format:

[Step-by-step thinking]
```python
# [Your code here, incorporating
    the starter code]

# [Additional code and comments
    as needed]
```

After writing your solution,
    please review it to ensure
    all requirements are met and
    the code is correct and
    efficient.

Here are the key elements for
    this task:
````

(c) User Prompt

```
1. Required Library:
<library>
{{library}}
</library>

2. Python version:
<python>
{{python_version}}
</python>

2. Coding Problem:
<coding_problem>
{{coding_problem}}
</coding_problem>

3. Starter Code:
<starter_code>
```

- NLTK [Loper and Bird, 2002]
- NetworkX [Hagberg et al., 2008]
- GeoPy[12]
- Gradio [Abid et al., 2019]
- Scikit-Learn [Buitinck et al., 2013]
- Matplotlib [Hunter, 2007]
- PyCaret[13]
- Pandas [The pandas development team, 2020, McKinney, 2010]
- NumPy [Harris et al., 2020]
- LightGBM[14]
- spaCy [15]
- Django[16]
- SciPy [Virtanen et al., 2020]
- Flask[17]
- Jinja2[18]
- SymPy[19]
- Seaborn[20]
- mitmproxy[21] [22]
- pytest [23]
- Falcon web framework[24]
- Tornado web server[25]
- Plotly[26]
- Librosa[27]
- Pillow [28]
- tqdm [29]
- Kymatio[30]

[12] https://pypi.org/project/geopy/
[13] https://pycaret.org/
[14] https://lightgbm.readthedocs.io/
[15] https://spacy.io/
[16] https://www.djangoproject.com/
[17] https://flask.palletsprojects.com/
[18] https://jinja.palletsprojects.com/
[19] https://www.sympy.org/en/index.html
[20] https://seaborn.pydata.org/
[21] https://mitmproxy.org/
[22] https://mitmproxy.org/
[23] https://pytest.org/
[24] https://falconframework.org/
[25] https://www.tornadoweb.org/
[26] https://plotly.com/python/
[27] https://librosa.org/doc/latest/index.html
[28] https://python-pillow.org/
[29] https://github.com/tqdm/tqdm
[30] https://librosa.org/doc/latest/index.html

J.2 Models

Open-Weights Models

The following open-weights models were evaluated:

- `Llama 3.1 Instruct Turbo`: Kassianik et al. [2025]
- `Llama 3.3 Instruct Turbo 70B`: AI [2025]
- `Llama 4 Maverick 400B`: AI [2025]
- `Qwen 2.5-VL Instruct 72B`: Qwen et al. [2025]
- `Qwen 3 235B`:Yang et al. [2025]
- `Command A 111B`: Cohere et al. [2025]
- `DeepSeek R1 685B`: DeepSeek-AI [2025]
- `DeepSeek v3`: DeepSeek-AI et al. [2025]
- `Openhands LM 32B v0.1`: Wang [2025]
- `Reka Flash-3`: Reka
- `Jamba 1.6 Mini, Large`: Lieber et al. [2024]

Enterprise Models

The following enterprise models were evaluated:

- `Arcee CoderL`: Arcee
- `Claude 3.5 Haiku`[31]
- `Claude 3.5 Sonnet`[32]
- `Claude 3.7 Sonnet`: Anthropic [2025]
- `Claude 4 Sonnet`[33]
- `CommandR+`[34]
- `Gemini 1.5 Pro`: Team et al. [2024]
- `Gemini 2.0 Flash`: Kampf [2025]
- `Gemini 2.5 Pro`: Cloud [2025]
- `Gemini 2.5 Flash`: Cloud [2025]
- `GPT-4.1`: [OpenAI, 2025a]
- `GPT-4.1-mini`: [OpenAI, 2025a]
- `GPT-4.1-nano`: [OpenAI, 2025a]
- `GPT-4o`: OpenAI [2024]
- `GPT-4o-mini`: OpenAI [2024]
- `GPT-4.5`: OpenAI [2025b]
- `o1`: [OpenAI, 2024]
- `o3-mini`: OpenAI [2024]
- `codex-mini`[35]
- `Grok 3`: xAI [2025]
- `Mistral Medium 3`: Mistral AI [2025]

[31] https://www.anthropic.com/claude/haiku
[32] https://www.anthropic.com/news/claude-3-5-sonnet
[33] https://www.anthropic.com/claude/sonnet
[34] https://cohere.com/blog/command-r-plus-microsoft-azure
[35] https://platform.openai.com/docs/models/codex-mini-latest

- `Devstral Small`[36]
- `Inflection 3 Productivity`[37]
- `Liquid LFM 40B MoE`[38]
- `Nova Pro`:Intelligence [2024]

J.3 Coding Assistants (CLI/IDE)

The following coding assistants were studied as part of the experimentation pipeline:

- `Claude Code`[39] (CLI)
- `Goose`[40] (CLI)
- `Cline`[41] (IDE-VSCode)
- `RooCode`[42] (IDE-VSCode)
- `KiloCode`[43] (IDE-VSCode)

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and introduction accurately describe the paper's contribution as the introduction of GitChameleon 2.0, a new executable benchmark for version-conditioned code generation. The subsequent sections provide a comprehensive empirical study evaluating various models on this benchmark, fulfilling the claims made.

   Guidelines:

   - The answer NA means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A No or NA answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: The paper does discuss its limitations in section 5 Limitations, and for example in Appendix H, where it clarifies that the benchmark is primarily designed to test version-specific knowledge retention rather than complex logical reasoning.

   Guidelines:

[36] https://mistral.ai/news/devstral

[37] https://openrouter.ai/inflection/inflection-3-productivity

[38] https://www.liquid.ai/blog/liquid-foundation-models-our-first-series-of-generative-ai-models

[39] https://docs.anthropic.com/en/docs/claude-code/overview

[40] https://block.github.io/goose/

[41] https://cline.bot/

[42] https://roocode.com/

[43] https://kilocode.ai/

- The answer NA means that the paper has no limitation while the answer No means that the paper has limitations, but those are not discussed in the paper.
- The authors are encouraged to create a separate "Limitations" section in their paper.
- The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
- The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
- The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
- The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
- If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
- While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [NA]

   Justification: This paper introduces an empirical benchmark and presents an evaluation of existing models; it does not propose new theoretical results, theorems, or proofs.

   Guidelines:

   - The answer NA means that the paper does not include theoretical results.
   - All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
   - All assumptions should be clearly stated or referenced in the statement of any theorems.
   - The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
   - Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
   - Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: The paper provides detailed descriptions of the dataset construction (Appendix C), the execution environment, and the exact models evaluated (Appendix J), which are sufficient to reproduce the main experimental results.

   Guidelines:

   - The answer NA means that the paper does not include experiments.

- If the paper includes experiments, a No answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: The paper releases the GitChameleon 2.0 benchmark, including all data and evaluation scripts, as an open-source asset with instructions for use.

   Guidelines:

   - The answer NA means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so "No" is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).

- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

   Answer: [Yes]

   Justification: The experimental settings, including the models tested, specific library versions, and evaluation metrics are described in Section 3 and Appendix J.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: The paper reports concrete evaluation metrics (e.g. execution pass rates) across hundreds of problems (Table 1), with error bars where applicable, providing a clear and statistically robust comparison of model capabilities.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The authors should answer "Yes" if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
   - The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
   - The assumptions made should be given (e.g., Normally distributed errors).
   - It should be clear whether the error bar is the standard deviation or the standard error of the mean.
   - It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
   - For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g. negative error rates).
   - If error bars are reported in tables or plots, The authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [No]

   Justification: As the paper contains a large array of experiments run on our benchmark, with many failed runs due to various reasons, it was difficult to provide an accurate estimate of the total compute and per experiment compute to reproduce all results. However, if requested for a specific experiment, we could provide a ballpark estimate.

Guidelines:

- The answer NA means that the paper does not include experiments.
- The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
- The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
- The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics https://neurips.cc/public/EthicsGuidelines?

   Answer: [Yes]

   Justification: The research conforms to the code of ethics as it involves the creation of a benchmark dataset and the evaluation of existing models. This work does not involve human subjects, personal data collection, or other areas that would typically raise ethical concerns under the policy.

   Guidelines:

   - The answer NA means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer No, they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [NA]

    Justification: Enabling better evaluation of version-conditioned code generation can have downstream impacts in terms of LLMs with better backward compatible code generation. From an efficiency standpoint, this can save a significant amount of time for projects using older versions. Beyond this, we did not feel it was necessary to point out specific positive or negative societal impacts of version-specific code generation.

    Guidelines:

    - The answer NA means that there is no societal impact of the work performed.
    - If the authors answer NA or No, they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
    - The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
    - The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.

- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

    Answer: [NA]

    Justification: The paper introduces GitChameleon 2.0, which is an evaluation benchmark dataset, not a generative model or a large-scale scraped dataset that carries a high risk for misuse. Therefore, specific safeguards for the responsible release of high-risk models or data are not applicable to this work.

    Guidelines:

    - The answer NA means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: The paper properly credits and cites the original sources for existing assets used, such as other benchmarks (e.g., SWE-Bench) and the models evaluated (with full details in Appendix J).

    Guidelines:

    - The answer NA means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
    - If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

Justification: The primary new asset, the GitChameleon 2.0 benchmark dataset, is thoroughly documented in the paper, particularly its construction methodology (Appendix A) and usage (Section 3).

Guidelines:

- The answer NA means that the paper does not release new assets.
- Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
- The paper should discuss whether and how consent was obtained from people whose asset is used.
- At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

Answer: [NA]

Justification: The research methodology did not involve crowdsourcing or human subjects.

Guidelines:

- The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
- Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

Answer: [NA]

Justification: This research does not involve human subjects, so Institutional Review Board (IRB) approval was not applicable or required.

Guidelines:

- The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does not impact the core methodology, scientific rigorousness, or originality of the research, declaration is not required.

Answer: [Yes]

Justification: The paper's core methodology is the evaluation of LLMs, and it also explicitly describes using Zencoder/GPT-4 as a component in the benchmark's hidden test set construction process (Appendix A.4).

Guidelines:

- The answer NA means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy (https://neurips.cc/Conferences/2025/LLM) for what should or should not be described.

Figure 16: Prompts for Self-Debugging

(a) System Prompt

```
You are an expert programming
    assistant. Your task is to fix
    issues in a generated Python
    solution for a given
    programming problem. You are
    provided with:

- A problem statement
- Starter code
- A previously generated incorrect
    solution
- A top-level execution trace or
    error message
- Dependencies information (
    versions, libraries).

Please generate a corrected Python
    solution by following these
    strict guidelines:

1. Use the required libraries
    explicitly in your code.
2. Correctly incorporate the
    provided starter code - do not
    remove or alter its structure.
3. Write in standard Python syntax.
4. Wrap your entire solution within
     a single Markdown code block.
5. Do not include any text outside
    the code block - no
    explanations, comments,
    docstrings, or usage examples.
6. Ensure the code is clean,
    efficient, and syntactically
    valid.
7. Avoid interactive, stateful, or
    environment-dependent
    constructs (e.g., Django
    projects, web servers).
8. Your output must be executable
    in a non-interactive
    environment (e.g., a test
    harness or script runner).


Example output format:

```python
# [Your corrected code here]
```

Before submitting, carefully review
     your code for correctness,
    completeness, and adherence to
    all constraints.
```

(b) User Prompt

```
<Problem>
{problem}
</Problem>

<Python Version>
{python_version}
</Python Version>

<Library>
{library}
</Library>

<Version>
{version}
</Version>

<Extra Dependencies>
{additional_dependencies}
</Extra Dependencies>

<Starting Code>
{starting_code}
</Starting Code>

<Generated Solution>
{solution}
</Generated Solution>

<Trace>
{top_level_trace}
</Trace>
```

Figure 17: Tool-Calling Agent Prompt

````
You are to solve a coding problem in Python.

# Instructions:

* The coding problem requires using the library {library}=={version}. Try
     using the problem with only this library and the standard Python
    libraries.

* Do a thorough research on the web about how to solve the coding problem
     for the given library version. Repeat multiple times if needed.

* BEFORE FINISHING YOUR WORK, YOU MUST check your solution to the coding
    problem by running the `docker_problem_sandbox` tool.

* Use the `final_answer` tool to return a self-contained Python script
    that solves the problem. DO NOT INCLUDE ANY TEXT BESIDES FOR THE CODE
     IN THE FINAL ANSWER.

* The solution needs to be in a markdown code block.

* The solution needs to start with the starter code provided below.


# Coding Problem:

{problem}


# Starter Code:

```python
{starting_code}
```
````

Figure 18: RAG Prompt

```
You are an AI assistant specialized in solving Python programming
    problems using information derived from documentation.

Each query may specify particular libraries and version constraints. Your
     task is to generate a correct, efficient, and minimal Python
    solution that adheres strictly to these requirements.

Please follow these rules when crafting your response:

1. Use only the specified libraries and respect the given version
    constraints.
2. Incorporate any provided starter code as required.
3. Write only Python code- no in- line comments or usage examples. Do not
     provide anything in the response but the code.
4. Ensure the code is clean, minimal, and adheres to best practices.
5. The code must be executable in a non-interactive environment (e.g.,
    avoid frameworks like Django or code requiring a web server).Context:
{context}

Based on the above, respond to the user query below.

Query: {query}
```

Here, `{context}` refers to the context of the top-k retrieved documents from the vectorized database for that query and `{query}` is the same as the User Prompt given in Figure 15(c).

Figure 19: Prompt and File Format for Coding Assistants

(a) Prompt

```
 Solve each sample_{i}.py in this
    folder then subsequently save
    your solutions as py files
    with the same name in a
    separate subfolder called "{
    assistant name}" that just
    completes the starting code
    provided in the sample and
    uses the instructions written
    in the comments at the start
    of each file.
```

(b) Input File Format

```
# Complete using the following
    libraries and/or extra
    dependencies and their versions
    :
# problem statement: {problem}
# library: {library}
# version: {version}
# extra_dependencies: {
    extra_dependencies}
{starting_code}
```

(a) presents the prompt template we had used for our Coding Assistant experiments. (b) shows the format of the example files referenced in the prompt.

Figure 20: Prompts for SEK (Keyword Generation Stage)

(a) System Prompt

```
You are a seasoned Python developer
     at a Fortune 500 company who
    excels at analyzing complex
    code. Analyze the given code
    problem from the problem
    statement and starter code
    provided. Try to extract the
    keywords from the code problem.
     For each identified keyword:
1. Provide the keyword.
2. Give a formalized explanation of
     the keyword using technical
    languages.

Provided Format:
Keywords:[Keywords]
Explainations:[Formalized
    explanations]

Guidelines:
- Prioritize keywords that are
    crucial to understanding the
    input parameters, return
    content or supplementary
    information.
- Use precise languages in
    explanations and provide
    formalized definitions where
    appropriate.
- Ensure explanations are
    consistent with the behaviors
    expected based on the problem
    description.
- Limit to the top 1-3 important
    keywords to focus on core
    concepts.
- You are supposed to output a
    structured JSON output
    containing the extracted
    keywords and their
    corresponding formalized
    explanations in individual
    lists of strings. The keys for
    this JSON must be Keywords and
    Explainations.
- Strictly adhere to the provided
    format, do not output anything
    else.
```

(b) User Prompt

```
<Problem Statement>
{problem}
</Problem Statement>

<Starting Code>
{starting_code}
</Starting Code>
```

Figure 21: Prompts for SEK (Code Generation Stage)

(a) System Prompt

````
You are a skilled Python programmer tasked with solving a coding problem. Your goal is to provide a clear, efficient, and correct solution that meets all the specified requirements.

Please provide your solution following these guidelines:

1. Use the required library in your solution.
2. Incorporate the provided starter code correctly.
3. Write your solution in Python.
4. Format your solution within a markdown code block.
5. Ensure your code is clean and efficient.
6. Output only the code block and nothing else. Do not add any in-line comments, documentations, references or usage examples.
7. Make sure your code is executable in a non-interactive environment. For example, do not write code which requires building a Django project or deploying a web-app.

Example output format:

```python
# [Your code here, incorporating the starter code]
```

After writing your solution, please review it to ensure all requirements are met and the code is correct and efficient.

Here are the key elements for this task:
````

(b) User Prompt

```
<Python Version>
{python_version}
</Python Version>

<Library>
{library}
</Library>

<Version>
{version}
</Version>

<Extra Dependencies>
{extra_dependencies}
</Extra Dependencies>

<Problem Statement>
{problem}
</Problem Statement>

<Keywords>
Analyze the following key terms and their relationships within the problem context:
{General_Keywords}
{Abstract_Keywords}
</Keywords>

<Starting Code>
{starting_code}
</Starting Code>
```