CodeMonkeys: Scaling Test-Time Compute for Software Engineering

Ryan Ehrlich*†, Bradley Brown*†‡, Jordan Juravsky*†, Ronald Clark‡, Christopher Ré†, and Azalia Mirhoseini†

†Department of Computer Science, Stanford University
‡University of Oxford
rehrlich@stanford.edu, bradley.brown@cs.ox.ac.uk, jbj@stanford.edu, ronald.clark@cs.ox.ac.uk, chrismre@stanford.edu, azalia@stanford.edu

January 24, 2025

Abstract

Scaling test-time compute is a promising axis for improving LLM capabilities. However, test-time compute can be scaled in a variety of ways, and effectively combining different approaches remains an active area of research. Here, we explore this problem in the context of solving real-world GitHub issues from the SWE-bench dataset. Our system, named CodeMonkeys, allows models to iteratively edit a codebase by jointly generating and running a testing script alongside their draft edit. We sample many of these multi-turn trajectories for every issue to generate a collection of candidate edits. This approach lets us scale "serial" test-time compute by increasing the number of iterations per trajectory and "parallel" test-time compute by increasing the number of trajectories per problem. With parallel scaling, we can amortize up-front costs across multiple downstream samples, allowing us to identify relevant codebase context using the simple method of letting an LLM read every file. In order to select between candidate edits, we combine voting using model-generated tests with a final multi-turn trajectory dedicated to selection. Overall, CodeMonkeys resolves 57.4% of issues from SWE-bench Verified using a budget of approximately 2300 USD. Our selection method can also be used to combine candidates from different sources. Selecting over an ensemble of edits from existing top SWE-bench Verified submissions obtains a score of 66.2% and outperforms the best member of the ensemble on its own. We fully release our code and data at https://scalingintelligence.stanford.edu/pubs/codemonkeys/.

1 Introduction

The abilities of large language models (LLMs) to solve increasingly complex coding tasks have rapidly improved [11, 27, 3]. Modern LLMs can outperform some human contestants in programming competitions [29, 49] and have become increasingly popular programming assistants [40, 5]. LLMs are also improving at software engineering tasks such as solving real-world GitHub issues, as measured by the SWE-bench dataset [25]. A major driver of progress has come from increasing the amount of compute and data used for model training, which has reliably led to improvements in model capabilities [20, 26, 21]. However, the costs of continuing to scale model training are becoming prohibitively expensive for most organizations [17].

An alternative avenue for further improving model capabilities, including for coding tasks such as SWE-bench, is to scale test-time compute [29, 8, 47, 59]. This type of scaling increases the amount of

* Equal Contribution. Work done by BB as a visiting researcher at Stanford.

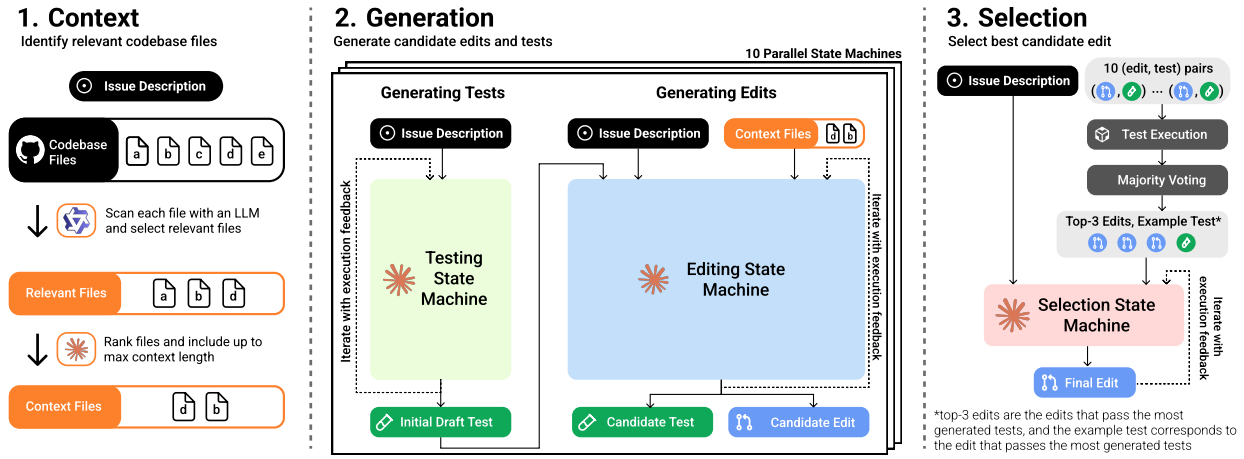

Figure 1: Overview of the CodeMonkeys system. **Left:** We retrieve codebase context by using models to first identify relevant files and then rank them relative to each other. **Middle:** We generate a codebase edit and testing script using a pair of multi-turn state machines that iterate based on execution feedback. We run these state machines multiple times in parallel to generate 10 edits and tests for every issue. **Right:** We select between candidate edits by identifying the candidates that pass the most generated tests and asking a model to decide between these top candidates. For details about our system's three state machines, see Figure 4.

computation expended during inference in order to produce higher-quality solutions. One approach to scaling test-time compute involves letting models deliberate for longer before outputting a final answer. This "serial" scaling can take the form of a chain-of-thought [36, 58] where models use many tokens of compute to reason through a problem, or through multi-turn interaction, where models iteratively respond to external feedback such as code execution results [62, 56, 60, 41]. Alternatively, test-time compute can be scaled in "parallel" by sampling multiple candidate solutions to a problem [57, 29, 30, 33]. In our previous work, Large Language Monkeys [8], we found that coverage – the fraction of problems in a dataset that are solved using any sample that was generated – often increases log-linearly with the number of samples, including when solving issues from SWE-bench.

While these coverage results provide encouraging evidence that scaling parallel test-time compute may be beneficial for SWE-bench, they do not directly provide an actionable recipe for using it to improve issue solve rates. In particular, generating multiple candidates introduces the problem of needing to select a final answer among them. Benefiting from high coverage requires a selection method that can distinguish between correct and incorrect answers. Additionally, in Large Language Monkeys, we used an off-the-shelf SWE-bench framework (Moatless Tools [64]) designed for generating a single solution. We generated multiple candidate edits by simply sampling from this framework repeatedly with a positive temperature. This raises the question: how would one design a system differently if benefiting from test-time compute scaling was a primary consideration?

The core contribution of this work is to explore this idea, presenting a system for solving SWE-bench instances designed specifically around scaling test-time compute (Figure 1). We segment the resolution of an issue into three major steps: 1) identifying relevant codebase context, 2) generating candidate codebase edits for resolving the issue, and 3) selecting among these candidate edits [60]. We scale serial compute when generating a codebase edit by enforcing that models also write a testing script alongside their edit, allowing them to iteratively revise their edits and tests in response to execution feedback. We scale parallel test-time compute by sampling many of these (edit, test) pairs for every SWE-bench issue. This combination of scaling achieves 69.8% coverage on SWE-bench

Stage	Claude Sonnet-3.5 API Costs				Local Costs	Total Cost
	Input	Output	Input Cache		Qwen-2.5	USD (%)
			Read	Write		
Relevance	0.00	0.00	0.00	0.00	334.02	334.02 (14.6%)
Ranking	0.00	11.92	1.10	6.90	0.00	19.92 (0.9%)
Gen. tests	10.60	295.15	21.60	112.64	0.00	439.99 (19.2%)
Gen. edits	14.67	353.95	636.82	360.58	0.00	1366.02 (59.6%)
Selection	0.52	51.12	15.17	65.14	0.00	131.95 (5.8%)
Total	25.79	712.14	674.69	545.26	334.02	2291.90 (100.0%)

Table 1: Breaking down the costs of running CodeMonkeys on all GitHub issues from SWE-bench Verified. All costs are in USD. Our system uses two LLMs: a primary model used for the ranking, generation, and selection stages (we use the Claude 3.5 Sonnet API [3]), and a cheaper model used for scanning codebases to identify relevant files (we run Qwen2.5-Coder-32B-Instruct [24] locally). For measuring costs with the Claude API, we use prices of $3/million input tokens, $0.3/million cache read tokens, $3.75/million cache write tokens, and $15/million output tokens. For details about estimating local inference costs, see Appendix C.

Verified. Interestingly, we find that different ways of allocating an inference budget between serial and parallel scaling often lead to similar coverage values. Additionally, our use of parallel scaling lets us amortize the cost of identifying relevant codebase context across multiple downstream samples. We adopt the simple method of letting an LLM scan every file, which contributes only 15% to total costs when run up-front once per issue.

To select between candidate codebase edits, we explore methods based on voting with model-generated tests [41, 60] and directly using models to do selection [41, 33]. We find that a combination of these two approaches works best, where test-based voting filters down the initial pool of candidates and a model selects among the remaining edits. Moreover, we find that model-based selection can be further improved with more serial compute by letting models write and run tests to distinguish between candidates. With this selection method, CodeMonkeys achieves an overall score of 57.4% on SWE-bench Verified (Table 2) while spending approximately 2300 USD on LLM inference (Table 1).

We also show that our approach to selection can be used to effectively combine generations from heterogeneous sources. We demonstrate this by assembling the "Barrel of Monkeys": an expanded pool of candidate edits that include the top-4 submissions on the SWE-bench Verified leaderboard. Selecting over this ensemble, which has a coverage of 80.8%, yields a score of 66.2% - higher than the top-performing ensemble submission of 62.8% and only 5.5% below the reported score of o3 (71.7%). We release our code along with all generated samples at `https://scalingintelligence.stanford.edu/pubs/codemonkeys/`.

2 Designing a SWE-bench Solver that Scales Test-Time Compute

Each SWE-bench instance consists of an issue description and a corresponding code repository. The objective is to edit one or more files in the codebase in order to resolve the issue. Edits can be automatically scored for correctness using the repository's testing suite. In our work, we focus on the Verified [12] split of SWE-bench, which contains instances that human annotators have classified

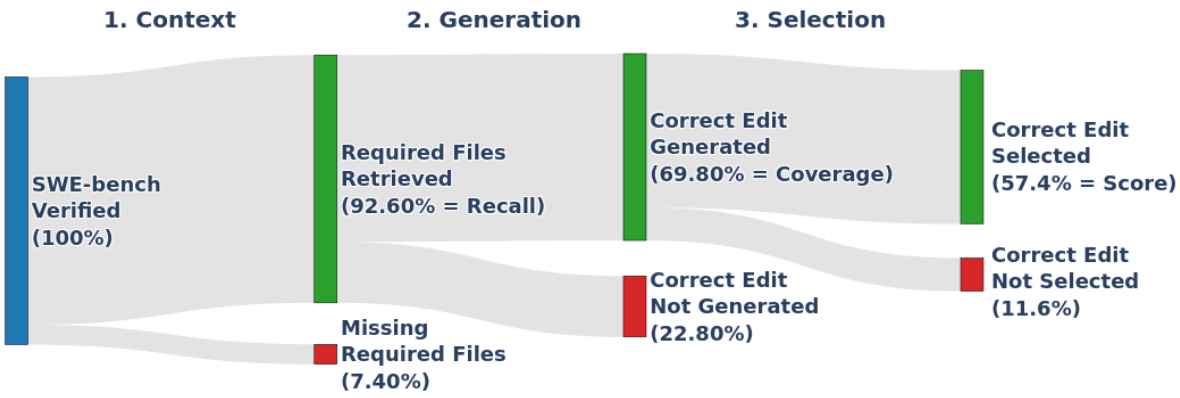

Figure 2: Measuring CodeMonkeys performance across the three subtasks we identify in Section 2 (context, generation, and selection). Note that modifying the approach to one subtask can influence the performance on other subtasks as well. For example, generating more candidate edits could increase coverage but make selection harder.

as "solvable" (i.e. by having unambiguous issue descriptions and test suites that do not filter out correct solutions). We decompose solving an instance from SWE-bench Verified into three sequential subtasks (Figure 1):

1. **Context:** Can we identify the codebase files that need to be edited[1] and put them in the context window? We can measure outcome of this subtask with **recall:** the fraction of problems where all needed files have been identified.

2. **Generation:** Can we produce a correct codebase edit among any of our sampled candidates? We can measure this outcome of this subtask with **coverage:** the fraction of problems where at least one generated edit is correct.

3. **Selection:** Can we choose a correct codebase edit from our collection of candidates? After completing this subtask, we can measure our final **score:** the fraction of problems in the dataset that are resolved by the edit our system submits.

We describe our approach to each subtask below, with per-subtask metrics reported in Figure 2 and a cost breakdown of our system in Table 1. Unless otherwise noted, all parts of our system use `claude-3-5-sonnet-20241022` [3]. We present additional results using DeepSeek-V3 [15] in Appendix A.

2.1 Identifying Relevant Codebase Context

One of the key challenges when solving SWE-bench instances is managing the large volume of input context. Most SWE-bench codebases contain millions of tokens worth of context. This exceeds the context lengths of most available models and would moreover be prohibitively expensive to process using frontier models. Existing approaches to managing SWE-bench context include using embedding models [64, 60], iterative expansion from a file tree [60], and giving models tools for browsing files [61, 45].

[1]When calculating recall, we determine if a file needs to be edited by checking if the file is edited in the official issue solution provided in the SWE-bench dataset. This calculation does not account for the existence of alternative solutions that resolve the issue by editing a different set of files.

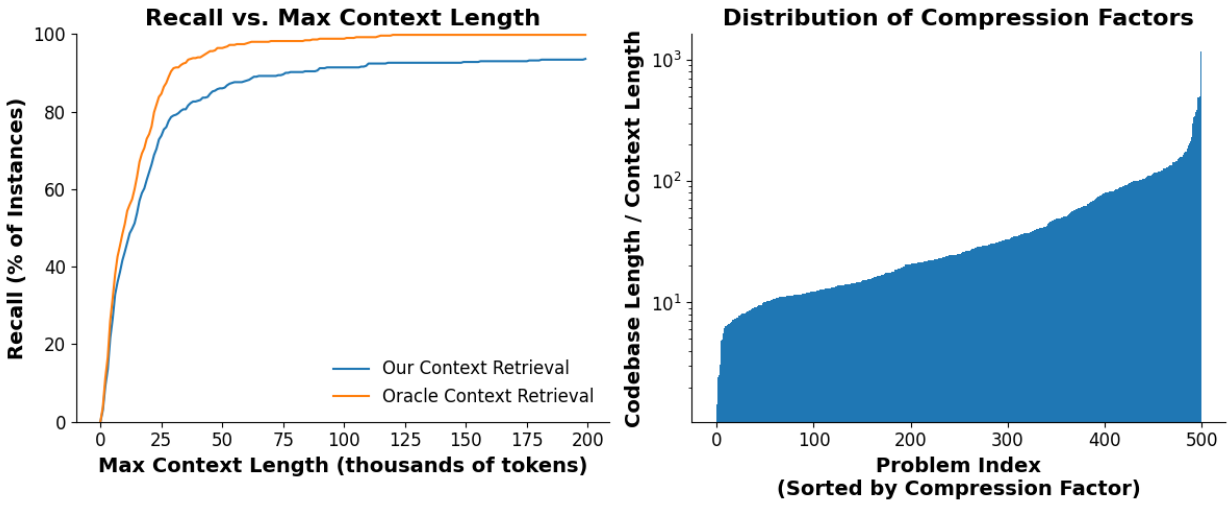

Figure 3: **Left**: Measuring recall (the fraction of SWE-bench problems with context windows that contain all needed files) as we increase the context window size limit. With the 128k token limit that we use for later experiments, 92.6% of instances have the correct files in context. **Right**: Visualizing the distribution of context compression factors across SWE-bench problems, i.e. the ratio between the cumulative token count of files scanned by the relevance model and the cumulative token count of files we include after relevance + ranking.

With our system, we know that we will generate a collection of candidate edits for every instance. Therefore, by choosing to share codebase context across all downstream edits, we can amortize the cost of context generation. This observation enables the simple approach of letting a model (we use Qwen2.5-Coder-32B-Instruct [24]) read every file in the codebase[2], decide whether each is relevant to the target issue, and only include relevant files in our context window [6]. Performing this codebase-wide scan once per instance contributes less than 15% to our total system costs (Table 1) and on average processes 2.94M tokens per problem. If we did not amortize this scan and instead reran it for each of the 10 edits we generate per problem, it would become the most expensive step. Sharing context across multiple downstream edits additionally saves costs by increasing hit rates when using prompt caching.

Even after the relevance filter, many problems still have contexts that are too long. To compress context further, we perform a model-based ranking procedure to order files by importance. First, as part of the initial codebase scan, we generate a concise summary for every file flagged as relevant that describes how the file relates to the target issue. We then construct a ranking prompt that includes each relevant file's name, summary, and token count. We ask the ranking model to include approximately 60,000 tokens of context in its ranking. Since the Claude API can be non-deterministic (even at temperature 0), we generate three completions from the ranking prompt and construct a final ranking by considering each file's average rank across the three repetitions. We construct a context window from this combined ranking by including the full contents all ranked files up to a limit of 128,000 tokens. On average, this leads to 74,570 tokens of codebase context, corresponding to an average 50.5x reduction in context size when compared with including every file that was assessed for relevance.

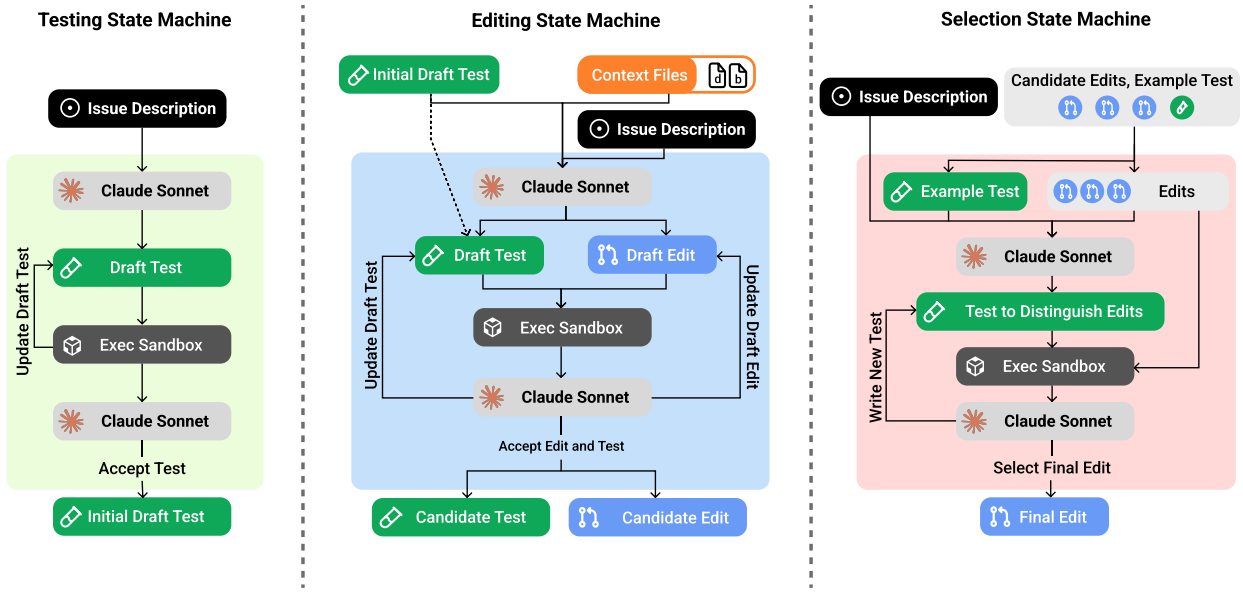

Figure 4: Details of the CodeMonkeys state machines. The **Testing State Machine** iteratively generates an initial draft of a testing script based on execution feedback from running the test on the codebase before any edits are applied. The **Editing State Machine** first generates an initial edit conditioned on the codebase context and the output test of the Testing State Machine. Then, it refines both the test and edit draft based on execution feedback from running the test before and after the edit is applied. The **Selection State Machine** first generates a test to distinguish between the top 3 candidate edits that pass the most testing scripts. Then, based on execution feedback of running this test with all of the candidate edits and on the codebase without edits, chooses to either create a new test script to further differentiate between the edits or selects a final edit.

2.2 Generating Candidate Codebase Edits with Corresponding Tests

With relevant parts of the codebase identified, we can begin generating candidate codebase edits for solving the target issue. We adopt a state machine abstraction [64] to model a multi-turn exchange where a model iteratively edits the target codebase in response to execution feedback. Additionally, like in Large Language Monkeys [8], we run multiple independent state machines for every SWE-bench instance, using a positive sampling temperature to introduce diversity across candidates. This approach provides us with two straightforward ways to scale test-time compute:

1. We can scale serial compute by increasing the maximum number of iterations performed per state machine[3].

2. We can scale parallel compute by increasing the number of independent state machines run per instance.

Importantly, we require that models generate and revise a test jointly with each edit. These tests are structured as standalone Python scripts that attempt to reproduce the GitHub issue and communicate their results using exit codes. Forcing models to write executable scripts alongside

[2]We only include Python files in our scan and exclude files inside of testing directories.

[3]Precisely, we limit the number of model completions per state machine. The model's initial generation and corrections to previous malformatted responses count against this limit.

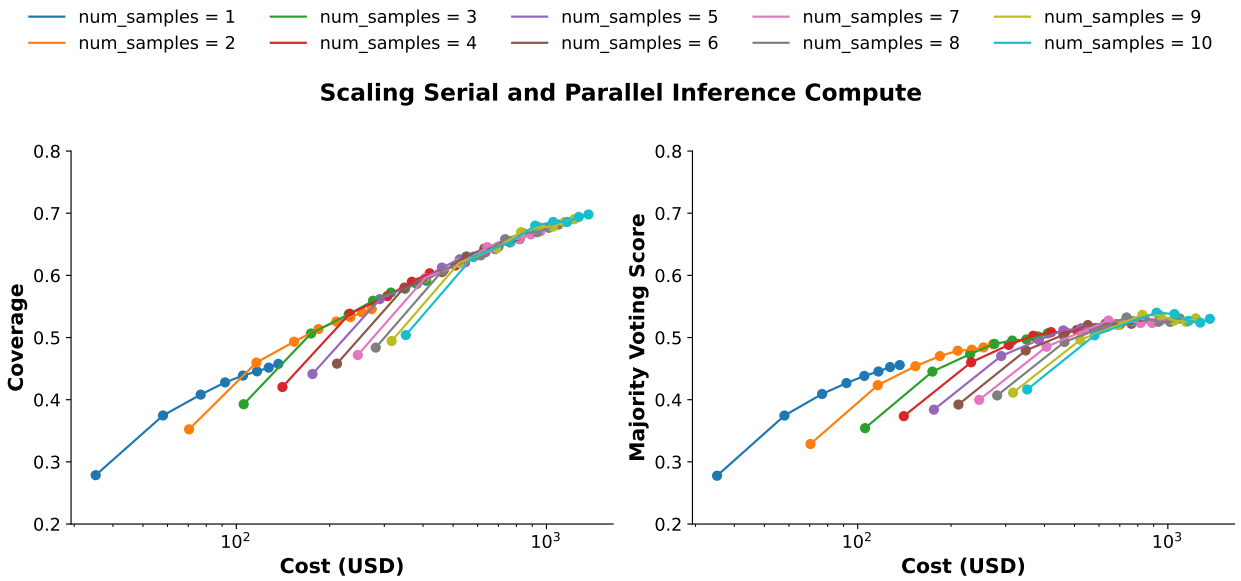

Figure 5: Measuring coverage (left) and score when using majority-voting selection (right) as we sweep over the number of serial iterations per editing state machine and the number of parallel state machines sampled per problem. Each colored curve corresponds to a different number of parallel state machines, and the dots along each curve correspond to increased numbers of sequential iterations per state machine. The first few serial iterations have a large impact on improving performance. However, past that point, different configurations with similar costs lead to similar performance, particularly for coverage.

edits provides models with richer feedback to guide their iteration and scale serial compute more effectively [23]. Additionally, tests serve as (imperfect) verifiers that can later assist with selecting between candidate edits (see Section 2.3).

Empirically, we find that models often require several iterations in order to write a functional test (e.g. because of configuration errors, easy-to-fix crashes, etc.). To allow models to focus on these steps first, we decompose this stage of our system into two back-to-back state machines:

1. An initial **testing state machine** which iterates on an initial draft of the test script.
2. A follow-up **editing state machine** which iterates on a codebase edit (in the form of an aider-style edit diff [16]). This state machine is seeded with the output of a testing state machine and can also revise the testing script as needed.[4]

The structure of these state machines is visualized in Figure 4 (left and middle panels), with additional details given in Appendix B.1. Our decomposition into separate testing and editing state machines also lowers system costs. Our cost table (Table 1) shows that prefix cache reads are the most expensive component of the editing state machine, in large part due to the codebase files that are included in the initial prompt. Since writing a testing script generally does not require codebase context, we can reduce prompt lengths (and therefore cache read costs) in our testing state machine

[4]Allowing models to continue revising tests during the editing state machine is important so that they can perform "two-sided debugging". In the testing state machine, models can keep iterating until their script correctly flags an issue when run on the unedited codebase. However, this can lead to tests that always report errors, even after a correct codebase edit has been applied. Allowing models to continue revising tests during the editing state machine lets them verify that their test fails pre-edit and passes post-edit.

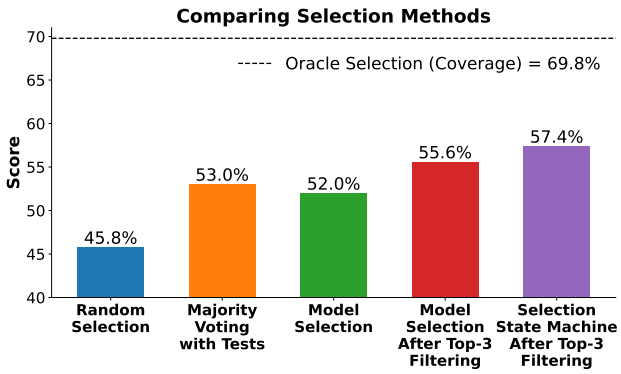

Figure 6: Comparing our selection methods when applied to the candidate edits generated by the CodeMonkeys editing state machines. Our best performing selection method – the selection state machine after top-3 filtering with generated tests – recovers approximately half of the difference between the random selection floor and the oracle selection ceiling (i.e. coverage).

Method	Score
Barrel of Monkeys (Oracle Selection)	80.8
o3	71.7
CodeMonkeys (Oracle Selection)	69.8
Barrel of Monkeys	66.2
Blackbox AI Agent	62.8
CodeStory	62.2
Learn-by-interact	60.2
devlo	58.2
CodeMonkeys	57.4
Emergent E1	57.2
Gru	57.0

Table 2: Comparing final SWE-bench Verified scores between the methods explored in this paper (bolded) and existing top approaches. Note that the Barrel of Monkeys results rely on the generations from existing submissions on the SWE-bench Verified leaderboard, and oracle selection methods are coverage numbers.

by omitting the files identified in Section 2.1. We additionally reduce prompt lengths by clearing the chat history between the testing and editing state machines.

We run 10 pairs of testing and editing state machines pairs for every instance in SWE-bench Verified and limit all state machines to eight iterations. On the left of Figure 5, we measure coverage as we sweep over both scaling parameters. Our best configuration uses all 10 state machines per instance and all eight iterations per state machine, achieving a coverage of 69.8%. Interestingly, after the first few iterations and state machines, a frontier emerges where configurations with similar total inference cost also have similar coverage, despite differences in how that cost is distributed across more state machines vs. more iterations. However, note that this does not mean that serial and parallel compute are fully interchangeable. In particular, two configurations with the same coverage will not necessarily obtain the same final score after selection. Scaling up parallel compute by generating many candidates can make selection more difficult, while generating a single, deep trajectory with a large number of iterations eliminates the selection problem entirely. Another distinction between the two types of scaling is that our setup only indirectly allows us to scale serial compute. We control the maximum number of iterations allowed within a single state machine. However, models can decide to approve of their test and/or edit and terminate a state machine before this iteration limit is reached. Increasing the iteration limit further does not help "stuck" state machines where models have already incorrectly approved of their work. In contrast, we can always scale parallel compute further by running an additional state machine, guaranteeing a "fresh start" where a model may be able to generate a correct codebase edit.

2.3 Selecting Between Candidate Edits

Under a best-case scenario with oracle selection (where our final score equals our coverage of 69.8%), CodeMonkeys would outperform all submissions on the SWE-bench Verified leaderboard and trail o3's reported score of 71.7% by only 1.9%. If, however, we instead selected edits at random, our expected score of only 45.8% would not even rank among the top 20 leaderboard submissions. This significant performance gap underscores the importance of accurate selection. We explore four

strategies for selecting among the 10 candidate edits that we generate per instance:

1. **Majority Voting with Tests:** We run each of the 10 model-generated tests on each of the 10 edits and select the edit that passes the most tests. If multiple edits tie with the most passes, we compute the expected score when picking randomly among them.

2. **Model Selection:** We use a model to select a candidate edit after prompting it with the issue description, codebase context, and candidate edits in git diff form.

3. **Model Selection After Top-3 Filtering:** We reuse the model selection prompt above, but select among only the three edits that pass the most generated tests. We break ties by favoring edits with shorter git diffs.

4. **Selection State Machine After Top-3 Filtering:** We upgrade the single-turn model-based selection approach to a full state machine that allows the model to write new testing scripts to differentiate between candidate edits (Figure 4 on the right and Appendix B.1). The initial prompt for this state machine includes the same information as for model-based selection, but also includes an example model-generated test from the earlier stages. At each iteration, the model can either select a final edit or generate a new test. Whenever a new test is written, we show the model the outputs when running the test on every candidate edit in addition to the unedited codebase. We reuse the top-3 filtering process described above to narrow down the initial pool of candidates.

We compare the performance of these selection methods in Figure 6. All four methods outperform random selection, with the selection state machine performing best. Note that the selection state machine is also the most expensive of the four methods we test; however, it contributes less than 10% to total system costs. We also see the benefit of the initial filter with majority voting: model selection without top-3 filtering underperforms pure majority voting, while model selection with filtering outperforms it. With our chosen approach of selection state machine + top-3 filtering, CodeMonkeys achieves a final score of 57.4% on SWE-bench Verified (Table 2), closing approximately half of the gap between random and oracle selection.

2.3.1 Barrel of Monkeys: Selecting over an Ensemble of Samples from Existing SWE-bench Submissions

In CodeMonkeys, we select among IID candidate edits generated by our method's state machines. Here, we demonstrate that our selection state machine is additionally helpful when combining candidate edits coming from heterogeneous sources. We create an ensemble of edits – the "Barrel of Monkeys" – by combining the final (already selected) edits from CodeMonkeys with the submissions from the top four entries on the SWE-bench Verified leaderboard[5]: Blackbox AI Agent [1], CodeStory Midwit Agent + swe-search [39], Learn-by-interact [48], and devlo [2]. The ensemble, with five samples per problem, has a combined coverage of 80.8%, which is notably higher than o3's reported score of 71.7%. While these two numbers are not directly comparable (o3's score corresponds to pass@1 while the Barrel of Monkeys' coverage corresponds to pass@5), we consider it important to highlight that (collectively) existing approaches can solve a significant fraction of instances from SWE-bench Verified. This further underscores the potential benefits of developing stronger methods for selection.

[5]As of January 15, 2025.

Since the Barrel of Monkeys begins with fewer initial candidates per problem than CodeMonkeys, we skip our initial test-based filtering and directly pass all edits to the selection state machine. The state machine's example test comes from the CodeMonkeys candidate edit that is part of the ensemble. Selecting over the ensemble achieves a score of 66.2%, outperforming the best-performing member of the ensemble in isolation (Blackbox AI Agent [1] with a score of 62.8%). However, since randomly selecting from this ensemble yields a score of 60.9%, our selection method recovers a smaller proportion of the gap between random selection and coverage here relative to selecting from the editing state machine.

3 Limitations and Future Work

In this work, we present the design of a system that successfully scales serial and parallel test-time compute in order to improve instance resolution rates on SWE-bench Verified. However, Figure 2 demonstrates that there is still room to improve each stage of our system:

- **Context:** Our file-level filter-and-rank procedure still omits relevant files for 7.4% of SWE-bench Verified instances. As models' usable context lengths grow and long-context processing becomes more efficient, we are hopeful that eventually this entire stage of our pipeline can be replaced with simply providing the entire codebase as context along with all relevant documentation [32]. Another factor to consider is that the base models already possess background knowledge on the repositories they are editing. This assumption allows us to avoid needing to provide the model with a more fundamental explanation about the purpose and usage of the package being edited (e.g. through documentation). This assumption will not hold when solving issues from new or private repositories that are not well-represented in the model training data, which will likely require modifications to our retrieval pipeline to attain high recalls.
- **Generating Edits and Tests:** We also see room for improving the CodeMonkeys state machines and further increasing coverage. In our state machines, we only provide models with execution feedback from their testing scripts, with each script attempting to reproduce the target issue. Additional execution feedback could come from model-written or existing regression tests that help prevent new bugs from being introduced. Moreover, like in Large Language Monkeys [8], CodeMonkeys incorporates diversity into generation exclusively through using a positive token sampling temperature. Distinct pairs of testing/editing state machines have no awareness of other state machines that are being made for the same problem, which can lead to redundant/non-diverse generations. Alternative approaches that add serial dependencies and inform models of previous attempts [53] could encourage greater diversity during generation.
- **Selection:** Our approach to selection recovers roughly half of the score gap between random and oracle selection when applied to CodeMonkeys, and an even smaller fraction of the gap when selecting over the Barrel of Monkeys. Improvements to selection methods, even without any changes to the candidate generation procedure, can therefore significantly raise overall SWE-bench scores. Similar to the potential improvements listed for generation, one source of selection signal that we do not exploit is the existing suite of tests inside of each repo. Agentless [60] and the Gemini team [33] both include these tests in their selection procedures.

Moreover, our work omits additional approaches to scaling test-time compute, such as ensembling different models together [43, 54] and leveraging dedicated reasoning models which are trained to

explore the space of candidate solutions in an extended chain of thought [37, 50]. Overall, we are excited about continued progress in methods for scaling test-time compute and systems that harness this scaling to solve real-world tasks.

4 Related Work

AI for Software Engineering: There has been considerable interest in applying LLMs to coding tasks [11, 27, 31]. Progress in this area can be measured with a diverse set of benchmarks that test models' abilities to complete functions from a prompt [11, 7], build entire libraries from a specification [63], and perform domain-specific programming tasks [38, 51]. In our work, we focus on SWE-bench [25], a dataset of real-world GitHub issues from popular Python repositories. The state-of-the-art on this benchmark has rapidly improved since its release: initial methods resolved less than 5% of instances from SWE-bench Verified [12], while current approaches can solve more than 60% [1, 39, 48]. This improvement can be attributed to stronger underlying models [3, 37, 33] in addition to better frameworks for equipping models with tools and guiding the issue resolution process.

These frameworks occupy a large design space. Some methods adopt a relatively hands-off approach that provide models with a suite of tools that they can use in an arbitrary order until they have resolved the issue [61, 45]. Other approaches (including CodeMonkeys) enforce a more strict structure on the issue resolution process, e.g. by introducing state machines [64] or a dedicated sequence of steps [60]. Similar to CodeMonkeys, Agentless [60] partitions issue resolution into steps for identifying relevant context, generating candidate edits, and selecting between these edits. However, our implementation of each of these steps (e.g. using a LLM-powered scan to identify context and using multi-turn feedback loops for generating edits and selecting between them) is distinct from Agentless.

Frameworks also differ in the types of tools that they provide to models. Some tools are general purpose, such as the ability to run shell commands or search the web [61, 45, 56]. Other tools are more narrow, such as editing files with the search-and-replace format introduced by Aider [16, 64, 60]. Some frameworks also provide models with tools for identifying relevant codebase context, such as by opening files or running a semantic search [64, 61, 45]. Existing frameworks for solving SWE-bench issues have also explored scaling test-time compute. Serial compute is commonly scaled by asking models to reflect/revise their work [64] or by providing them with execution or tool call feedback [56, 22, 41]. Notably, with Anthropic's framework for showcasing the updated Claude 3.5 Sonnet [45], Claude sometimes used hundreds of turns of feedback before submitting a correct solution. Agentless [60] and the Gemini team [33] scale parallel compute by generating multiple candidate edits and using repositories' existing unit tests to help select between them. Agentless additionally uses model-written reproduction tests during selection to check that candidate edits actually resolved the issue, while the Gemini team and SpecRover [41] incorporate model-based selection. Methods like SWE-search [4] and CodeStory [39] combine serial and parallel scaling by performing a tree search over intermediate states using model-based value estimation.

Scaling Test-Time Compute: Expending test-time compute via tree search has long been a successful strategy when designing AI systems to play games [10, 46, 9]. Recently, similar search techniques have also been combined with LLMs to successfully prove formal mathematical statements [52]. Across a wider variety of settings, allowing LLMs to use more tokens to think through a problem before outputting a final answer has led to large boosts in model reasoning capabilities [58, 36]. Explicitly optimizing models to perform this deliberation process (e.g. with reinforcement learning) has improved capabilities even further [37, 50, 14].

Existing work has also explored scaling parallel test-time compute by sampling multiple completions per problem [8, 59, 47]. Across math and coding tasks, repeatedly sampling with small models can obtain higher coverage than single samples from a larger model [19]. Repeated sampling can be particularly effective with coding tasks since the ability to run and test model outputs can assist with selection [29, 18, 28]. More general approaches to selection include using outcome reward models [13], process reward models [30, 55], or prompting-based verifier setups [47]. PlanSearch [53] shows that converting some parallel samples into sequential attempts can improve diversity, reducing the compute needed to achieve a given coverage. Archon [43], Mixture-of-Agents [54] and MALT [34] consider the setting where many different models can generate samples or act as sample verifiers and aggregators, exploring how performant combinations of these LLM components can be automatically discovered.

5 Acknowledgements

We are grateful to Benjamin Spector, Chris Fifty, Jerry Liu, Jon Saad-Falcon, Owen Dugan, Quinn McIntyre, Simon Guo, and Will Tennien for their helpful discussions and feedback throughout this project.

We gratefully acknowledge the support of NIH under No. U54EB020405 (Mobilize), NSF under Nos. CCF2247015 (Hardware-Aware), CCF1763315 (Beyond Sparsity), CCF1563078 (Volume to Velocity), and 1937301 (RTML); US DEVCOM ARL under Nos. W911NF-23-2-0184 (Long-context) and W911NF-21-2-0251 (Interactive Human-AI Teaming); ONR under Nos. N000142312633 (Deep Signal Processing); Stanford HAI under No. 247183; NXP, Xilinx, LETI-CEA, Intel, IBM, Microsoft, NEC, Toshiba, TSMC, ARM, Hitachi, BASF, Accenture, Ericsson, Qualcomm, Analog Devices, Google Cloud, Salesforce, Total, the HAI-GCP Cloud Credits for Research program, the Stanford Data Science Initiative (SDSI), and members of the Stanford DAWN project: Meta, Google, and VMWare. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views, policies, or endorsements, either expressed or implied, of NIH, ONR, or the U.S. Government.

This work was completed with the support of the Clarendon Fund Scholarships.

References

[1] Elevating swe-bench verified with blackbox agent. `https://blog.blackbox.ai/posts/swe-bench`, 2025.

[2] Your ai-developer teammate. `https://devlo.ai/`, 2025.

[3] Anthropic. Introducing computer use, a new claude 3.5 sonnet, and claude 3.5 haiku, 2024.

[4] Antonis Antoniades, Albert Örwall, Kexun Zhang, Yuxi Xie, Anirudh Goyal, and William Wang. Swe-search: Enhancing software agents with monte carlo tree search and iterative refinement, 2024.

[5] Anysphere Team. Series B and Automating Code, January 2025. Blog post announcing $105M Series B funding and company milestones.

[6] Simran Arora, Brandon Yang, Sabri Eyuboglu, Avanika Narayan, Andrew Hojel, Immanuel Trummer, and Christopher Ré. Language models enable simple systems for generating structured views of heterogeneous data lakes, 2023.

[7] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models, 2021.

[8] Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V. Le, Christopher Ré, and Azalia Mirhoseini. Large language monkeys: Scaling inference compute with repeated sampling, 2024.

[9] Noam Brown, Anton Bakhtin, Adam Lerer, and Qucheng Gong. Combining deep reinforcement learning and search for imperfect-information games. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, Red Hook, NY, USA, 2020. Curran Associates Inc.

[10] Murray Campbell, A. Joseph Hoane, and Feng-hsiung Hsu. Deep blue. *Artif. Intell.*, 134(1–2):57–83, jan 2002.

[11] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code, 2021.

[12] Neil Chowdhury, James Aung, Chan Jun Shern, Oliver Jaffe, Dane Sherburn, Giulio Starace, Evan Mays, Rachel Dias, Marwan Aljubeh, Mia Glaese, Carlos E. Jimenez, John Yang, Kevin Liu, and Aleksander Madry. Introducing SWE-bench Verified, August 2024. Blog post announcing a human-validated subset of SWE-bench for evaluating AI models' software engineering capabilities.

[13] Paul Christiano, Jan Leike, Tom B. Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences, 2017.

[14] DeepSeek-AI. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025.

[15] DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, et al. Deepseek-v3 technical report, 2024.

[16] Paul Gauthier. Aider is ai pair programming in your terminal. `https://aider.chat/`, 2024.

[17] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models, 2024.

[18] Ryan Greenblatt. Geting 50 `https://www.lesswrong.com/posts/Rdwui3wHxCeKb7feK/getting-50-sota-on-arc-agi-with-gpt-4o`, 2024.

[19] Michael Hassid, Tal Remez, Jonas Gehring, Roy Schwartz, and Yossi Adi. The larger the better? improved llm code-generation via budget reallocation, 2024.

[20] Joel Hestness, Sharan Narang, Newsha Ardalani, Gregory Diamos, Heewoo Jun, Hassan Kianinejad, Md. Mostofa Ali Patwary, Yang Yang, and Yanqi Zhou. Deep learning scaling is predictable, empirically, 2017.

[21] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre. Training compute-optimal large language models, 2022.

[22] Dong Huang, Jie M. Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation, 2024.

[23] Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet, 2024.

[24] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

[25] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues?, 2024.

[26] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models, 2020.

[27] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. Starcoder: may the source be with you!, 2023.

[28] Wen-Ding Li, Keya Hu, Carter Larsen, Yuqing Wu, Simon Alford, Caleb Woo, Spencer M. Dunn, Hao Tang, Michelangelo Naim, Dat Nguyen, Wei-Long Zheng, Zenna Tavares, Yewen Pu, and Kevin Ellis. Combining induction and transduction for abstract reasoning, 2024.

[29] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, December 2022.

[30] Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step, 2023.

[31] Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. Large language model-based agents for software engineering: A survey, 2024.

[32] Magic Team. 100M token context windows, aug 2024. Blog post.

[33] Shrestha Basu Mallick and Kathy Korevec. The next chapter of the gemini era for developers, 2024.

[34] Sumeet Ramesh Motwani, Chandler Smith, Rocktim Jyoti Das, Markian Rybchuk, Philip H. S. Torr, Ivan Laptev, Fabio Pizzati, Ronald Clark, and Christian Schroeder de Witt. Malt: Improving reasoning with multi-agent llm training, 2024.

[35] NVIDIA. Nvidia l40s data sheet. https://resources.nvidia.com/en-us-l40s/l40s-datasheet-28413, 2024. Accessed: 2024-01-14.

[36] Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, Charles Sutton, and Augustus Odena. Show your work: Scratchpads for intermediate computation with language models, 2021.

[37] OpenAI. Introducing openai o1, 2024.

[38] Anne Ouyang, Simon Guo, and Azalia Mirhoseini. Kernelbench: Can llms write gpu kernels?, 2024.

[39] Sandeep Kumar Pani. Sota on swebench-verified: (re)learning the bitter lesson, 2024.

[40] Tiernan Ray. Microsoft has over a million paying Github Copilot users: CEO Nadella. *ZDNET*, October 2023. Reports Microsoft's GitHub Copilot reaching 1 million paid users across 37,000 organizations.

[41] Haifeng Ruan, Yuntong Zhang, and Abhik Roychoudhury. Specrover: Code intent extraction via llms, 2024.

[42] RunPod. Runpod cloud gpu pricing. https://www.runpod.io/console/deploy, 2024. Accessed: 2024-01-14.

[43] Jon Saad-Falcon, Adrian Gamarra Lafuente, Shlok Natarajan, Nahum Maru, Hristo Todorov, Etash Guha, E. Kelly Buchanan, Mayee Chen, Neel Guha, Christopher Ré, and Azalia Mirhoseini. Archon: An architecture search framework for inference-time techniques, 2024.

[44] Nikhil Sardana, Jacob Portes, Sasha Doubov, and Jonathan Frankle. Beyond chinchilla-optimal: Accounting for inference in language model scaling laws, 2024.

[45] Erik Schluntz, Simon Biggs, Dawn Drain, Eric Christiansen, Shauna Kravec, Felipe Rosso, Nova DasSarma, and Ven Chandrasekaran. Raising the bar on SWE-bench Verified with Claude 3.5 Sonnet. Blog post, Anthropic, oct 2024.

[46] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm, 2017.

[47] Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling llm test-time compute optimally can be more effective than scaling model parameters, 2024.

[48] Hongjin Su, Ruoxi Sun, Jinsung Yoon, Pengcheng Yin, Tao Yu, and Sercan Ö. Arık. Learn-by-interact: A data-centric framework for self-adaptive agents in realistic environments, 2025.

[49] AlphaCode Team. Alphacode 2 technical report, 2024.

[50] Qwen Team. Qwq: Reflect deeply on the boundaries of the unknown, 2024.

[51] Minyang Tian, Luyu Gao, Shizhuo Dylan Zhang, Xinan Chen, Cunwei Fan, Xuefei Guo, Roland Haas, Pan Ji, Kittithat Krongchon, Yao Li, Shengyan Liu, Di Luo, Yutao Ma, Hao Tong, Kha Trinh, Chenyu Tian, Zihan Wang, Bohao Wu, Yanyu Xiong, Shengzhu Yin, Minhui Zhu, Kilian Lieret, Yanxin Lu, Genglin Liu, Yufeng Du, Tianhua Tao, Ofir Press, Jamie Callan, Eliu Huerta, and Hao Peng. Scicode: A research coding benchmark curated by scientists, 2024.

[52] Trieu H. Trinh, Yuhuai Wu, Quoc V. Le, He He, and Thang Luong. Solving olympiad geometry without human demonstrations. *Nature*, 625(7995):476–482, 2024.

[53] Evan Wang, Federico Cassano, Catherine Wu, Yunfeng Bai, Will Song, Vaskar Nath, Ziwen Han, Sean Hendryx, Summer Yue, and Hugh Zhang. Planning in natural language improves llm search for code generation, 2024.

[54] Junlin Wang, Jue Wang, Ben Athiwaratkun, Ce Zhang, and James Zou. Mixture-of-agents enhances large language model capabilities, 2024.

[55] Peiyi Wang, Lei Li, Zhihong Shao, R. X. Xu, Damai Dai, Yifei Li, Deli Chen, Y. Wu, and Zhifang Sui. Math-shepherd: Verify and reinforce llms step-by-step without human annotations, 2024.

[56] Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. Openhands: An open platform for ai software developers as generalist agents, 2024.

[57] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models, 2023.

[58] 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.

[59] Yangzhen Wu, Zhiqing Sun, Shanda Li, Sean Welleck, and Yiming Yang. Inference scaling laws: An empirical analysis of compute-optimal inference for problem-solving with language models, 2024.

[60] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying llm-based software engineering agents, 2024.

[61] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering, 2024.

[62] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models, 2023.

[63] Wenting Zhao, Nan Jiang, Celine Lee, Justin T Chiu, Claire Cardie, Matthias Gallé, and Alexander M Rush. Commit0: Library generation from scratch, 2024.

[64] Albert Örwall. Moatless tools. `https://github.com/aorwall/moatless-tools/tree/a1017b78e3e69e7d205b1a3faa83a7d19fce3fa6`, 2024.

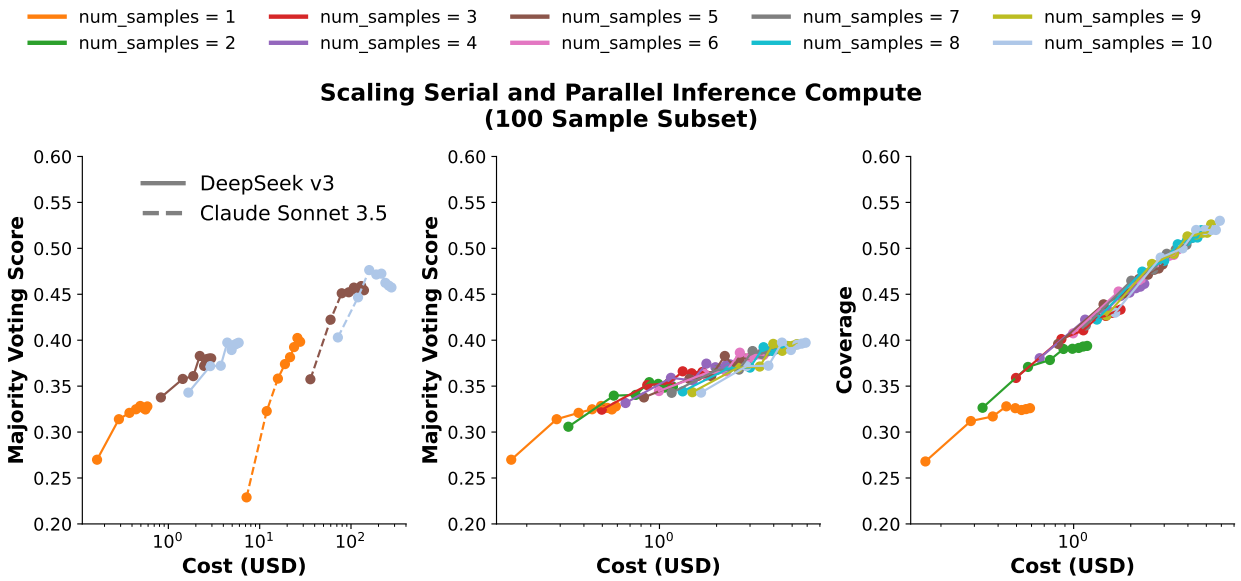

Figure 7: **Left:** Comparing the impact of scaling the number of parallel samples and sequential iterations on majority voting score between Claude and DeepSeek-V3. Each line corresponds to a fixed amount of samples, with each dot on the line being a different maximum number of sequential iterations. We see that although Claude can achieve a higher overall score, DeepSeek-V3 can achieve 86.8% percent of the score at a fraction of the cost. **Center:** A more granular view of the majority voting scaling for DeepSeek-V3. **Right:** Coverage for DeepSeek-V3 as a function of the number of parallel samples and sequential iterations. We highlight that coverage is continuing to scale with increased inference compute.

A DeepSeek-V3 Results

Here, we conduct an initial evaluation of DeepSeek-V3 [15] as a potential substitute for our system's use of Claude 3.5 Sonnet. We reuse the same codebase context files as our primary CodeMonkeys experiments and rerun our testing and editing state machines on a 100-instance random subset of SWE-bench Verified using DeepSeek-v3. In the Figure 7, we compare how coverage and majority voting score scale with the number of samples and iterations when compared to Claude Sonnet 3.5. We note that Claude Sonnet 3.5 is able to achieve a score of 45.74% on this subset of problems, which is 6.02% higher than the best score of DeepSeek-V3. However, the DeepSeek API is over an order of magnitude cheaper than that of Claude. These results highlight the potential benefits of generating many candidate solutions from a cheaper model like DeepSeek-V3 so long as a selection method can identify correct samples from large collections.

B Experimental Details

We release our code and trajectories at `https://scalingintelligence.stanford.edu/pubs/codemonkeys/`. This release includes:

- All the code needed to run CodeMonkeys on SWE-bench Verified.
- All the commands needed to generate the figures/tables in this paper.
- The complete trajectories taken by CodeMonkeys when solving each problem.

B.1 State Machine Details

All three CodeMonkeys state machines (the testing, editing, and selection state machines) follow the same structure. Each state machine begins by giving the model an initial prompt and an initial task. Once the model completes this initial task, a feedback loop is entered. At each iteration, the model is provided with some execution feedback from the previous iteration. The model is then allowed to either revise its work, triggering a new iteration of the loop, or approve its work, terminating the state machine. In Table 3, we provide the inputs, initial task, information given at each iteration, and the task at each iteration for all three state machines. For full prompts, please see the folder `codemonkeys/prompts` in our codebase.

	Testing State Machine	Editing State Machine	Selection State Machine
Information in Initial Prompt	GitHub issue description.	GitHub issue description, codebase context, final test script from a testing state machine, execution output when running the provided test on the unedited codebase.	GitHub issue description, candidate edits in git diff form, full contents of any codebase files that have been edited, test script from the edit that passed the most generated tests (breaking ties by using the shortest edit).
Initial Task	Write a test script that reproduces the issue. The script should exit with code 0 if the issue is fixed and exit with code 2 if the issue is not fixed.	Write a codebase edit that resolves the issue.	Write a test script for distinguishing between candidates and assessing their correctness.
Information in Iteration Prompt	Execution output when running the test on the codebase (which has not yet been edited).	Execution outputs when running the testing script on the unedited codebase and edited codebase.	Execution outputs when running the test script on a codebase after each edit has been applied, in addition to the unedited codebase.
Iteration Task	Rewrite the test script or approve of it (terminating the state machine).	Rewrite the edit, rewrite the test script, or approve of the (edit, test) pairs (terminating the state machine).	Write a new testing script, or make a selection among the candidate edits (terminating the state machine).

Table 3: Details of the Testing, Editing, and Selection State Machines.

B.2 Hyperparameters

We elaborate on other experimental details for each part in Table 4.

C Local Compute for Relevance

For the relevance stage, we run Qwen-2.5-Coder-32B-Instruct [24] locally in bf16 across 8xL40S GPUs. Each L40S has a bf16 compute throughput of 362.05 TFLOPS [35], giving our node a

Subtask	Stage	Parameter	Value
Context	Relevance	Model	Qwen-2.5-Coder-32B-Instruct
		Hardware	8xL40S
		Temperature	0.0
	Ranking	Model	Claude Sonnet 3.5
		Temperature	0.0
		Repetitions	3
Generation	Testing & Editing	Temperature (Sonnet 3.5)	0.5
		Temperature (DeepSeek-V3)	0.6
		Number of State Machines per Instance	10
		Maximum Iterations	8
		Generated Test Timeout (seconds)	100
Selection	—	Temperature	0.0
		Maximum Iterations	10
		Generated Test Timeout (seconds)	100

Table 4: CodeMonkeys hyperparameter summary.

theoretical throughput of:

$$8 \text{ devices} \cdot 362.05 \text{ TFLOPS/device} = 2{,}896.4 \text{ TFLOPS}$$

Since the model we run is a dense transformer with no parameter sharing, we can estimate its FLOPs per token as 2 * num_parameters [44]. Assuming a hardware utilization of 20% during inference, we obtain a per-node throughput of:

$$\text{Throughput} \approx \frac{0.2 \cdot 2{,}896.4 \cdot 10^{12} \text{ FLOPs/second}}{2 \cdot 32 \cdot 10^{9} \text{ FLOPs/token}} = 9{,}051 \text{ tokens/second}$$

Across the 500 instances in SWE-bench Verified, the relevance stage requires processing a total of $1.32084 \cdot 10^9$ tokens. To estimate the cost of compute, we use RunPod's pricing: an 8xL40S node costs $8.24 per hour [42], yielding a total cost of:

$$\begin{aligned}\text{Relevance cost} &\approx \frac{1.32083 \cdot 10^{9} \text{ tokens}}{9{,}051 \text{ tokens/second} \cdot 3{,}600 \text{ seconds/hour}} \cdot \$8.24/\text{hour} \\ &= 40.5 \text{ hours} \cdot \$8.24/\text{hours} \\ &= \$334.02\end{aligned}$$