Systems and Methods for Improving Large Language Models with Increased Inference Compute

Bradley Brown

St. Cross College

University of Oxford

A thesis presented for the degree of

Master's by Research

Trinity 2025

Abstract

Scaling the amount of compute used to train language models has led to massive increases in their capability. A relatively under-explored direction has been scaling the amount of compute used at inference time. This thesis explores the design space for test-time compute algorithms, focusing on the simple method of repeated sampling. Notably, this thesis demonstrates that coverage, the percent of problems solved by any sample, often scales log-linearly with the number of samples over four orders of magnitude. In domains where automatic verification tools are available, this allows Llama-8b to outperform the single-attempt performance of GPT4o across all tasks studied. These increases in coverage can often be modeled with an exponentiated power law, which enables forecasting coverage up to 10k samples (the largest sample size attempted) using an order of magnitude fewer samples with an average error of 3.84% across all models and datasets. In domains where tools for verification are available, these gains in coverage directly translate into increased capability. In other domains, methods for selecting a correct sample from a large sample collection are required. This thesis investigates this selection problem in the salient application of software engineering. Using a combination of unit-test based voting and model-based selection, half of the accuracy between the single-attempt lower bound and coverage upper bound can be recovered. Finally, this thesis explores the systems implications of test-time compute scaling and presents an exact implementation of attention that accelerates inference throughput in these settings by over 10x.

Table of Contents

1 | Introduction

The ability of large language models (LLMs) to solve coding, mathematics, and other reasoning tasks has improved dramatically over the past several years [90, 22, 1, 2]. Scaling the amount of training compute through bigger models, longer pre-training runs, and larger datasets has been a consistent driver of these gains [46, 63, 48].

In contrast, a comparatively limited investment has been made in scaling the amount of computation used during inference. Larger models do require more inference compute than smaller ones, and prompting techniques like chain-of-thought [121] can increase answer quality at the cost of longer (and therefore more computationally expensive) outputs. However, when interacting with LLMs, users and developers often restrict models to making only one attempt when solving a problem.

In this thesis, I explore how simple methods for increasing the amount of inference compute affect model capabilities, and develop systems to make these algorithms efficient. Notably, I focus on the method of repeated sampling (Figure 1.1) and scale the number of solutions that are sampled per problem. Existing work provides encouraging examples that repeated sampling can be beneficial in math, coding, and puzzle-solving settings [120, 92, 43, 69]. In Chapter 3, my goal is to systematically characterize these benefits across a range of tasks, models, and sample budgets.

The effectiveness of repeated sampling is determined by two key properties:

1. **Coverage:** As the number of samples increases, what fraction of problems are solved using any sample that was generated?
2. **Precision:** How often are correct samples identified from the collection of generations?

Exploring coverage first, I find that sampling up to 10,000 times per problem can significantly boost coverage on math and coding tasks. When solving CodeContests [69] programming problems using Gemma-2B [105], coverage increases by over 300x, from 0.02% with one sample to 7.1% with 10,000 samples. Interestingly, increases in coverage can often be modeled with an exponentiated power law. This inference scaling law forecasts coverage at 10k samples with a mean error of 2.53% using only 1k samples across all models and all tasks except for MiniF2F-MATH. These coverage increases are directly beneficial in settings with automatic

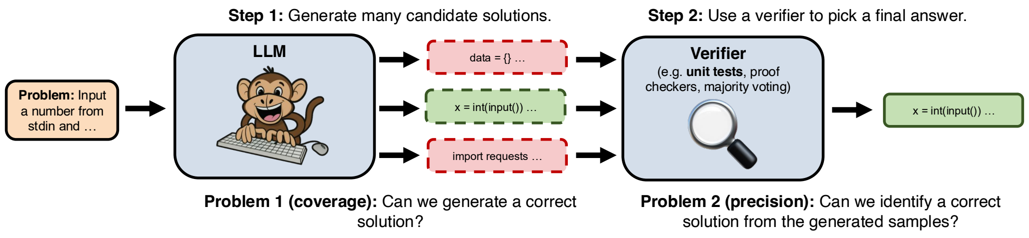

Step 1: Generate many candidate solutions.
Step 2: Use a verifier to pick a final answer.
Problem: Input a number from stdin and ...
LLM
data = {} ...
x = int(input()) ...
import requests ...
Verifier (e.g. unit tests, proof checkers, majority voting)
x = int(input()) ...
Problem 1 (coverage): Can we generate a correct solution?
Problem 2 (precision): Can we identify a correct solution from the generated samples?

Figure 1.1: An overview of repeated sampling. 1) Generate many independent candidate solutions for a given problem by sampling from an LLM with a positive temperature. 2) Use a domain-specific verifier (ex. unit tests for code) to select a final answer from the generated samples.

verification, such as GPU kernel generation [87], SWE tasks with comprehensive test suites, code transpilation, formal proof generation, and simulation (e.g. chip design).

Next, this thesis examines the precision problem in settings without automatic verifiers. As the number of samples increases, coverage improves because models generate correct solutions to problems they have not previously solved. Taking advantage of large-scale repeated sampling therefore requires the ability to find "needles in the haystack" and identify rare, correct generations. I show that rare correct generations are present across many problems (e.g. only 1 in 10k), explaining why self-consistency, which simply extracts the plurality answer, is ineffective at scale.

In Chapter 4, this thesis then examines how repeated sampling can be beneficial in the practically important domain of software engineering. Specifically, I investigate how to build a system to solve SWE-bench issues that is designed to leverage test-time compute. In this task, models are given access to a Python codebase and corresponding GitHub issue then have to make edits to the codebase in order to fix the issue, as measured by the repository's unit tests. Notably, the model does not have access to these unit tests when solving the GitHub issue. Therefore, to benefit from repeated sampling, a method is needed to select between many candidate solutions. In order to have a sufficiently high single-attempt performance, and keep the number of samples to select over relatively small, I first explore the benefit of scaling serial inference compute by having the model iteratively revise each candidate solution in response to execution feedback. Then, I explore various selection strategies over these candidate solutions that are based on voting with model-generated tests and directly using models to do selection. A combination of the two is the most successful, and recovers over half the performance between the single-attempt lower bound and the coverage upper bound when selecting between 10 candidate samples. Finally, I demonstrate how explicitly accounting for increased test-time compute enables certain design decisions that would be infeasible otherwise. Notably, because generating context can be amortized across all of the parallel samples, I am able to do a relatively expensive and high-quality context aggregation without it being the dominant cost in the system (Table 4.1).

As methods that leverage inference-time compute become increasingly used, addressing corresponding system challenges becomes critical. In the context of repeated sampling, we are commonly doing large-batch inference where sequences in the batch have overlapping prefixes. Previous attention implementations do not account for this, and do redundant loads of the shared prefix for each item in the batch. To solve this, in Chapter 5, this thesis introduces Hydragen: an exact implementation of attention designed for settings with large batches with shared prefixes. Hydragen decomposes attention into two parts: attention over the shared prefix which is shared over all sequences, and attention over the unique suffix which is independent per sequence. These separate computations are then combined using the softmax trick, introduced in Flash Attention [34]. In addition to lowering bandwidth by removing the redundant loads, this also enables the use of tensor cores for the shared prefix. Overall, this improves throughput by up to 10x over competitive baselines.

2 | Literature Review

This literature review provides an overview of work related to methods and systems for increasing the capability of LLMs through more test-time compute in LLMs. Additionally, I give background on the associated topics in AI for software engineering that are relevant for the methods that this thesis investigates. I will first give a brief overview of important components of the transformer architecture and sampling mechanisms that are necessary to understand the core investigation of repeated sampling (Section 2.1). Next, I will give an overview of methods that past work has used to increase test-time compute in neural networks to improve performance, with a focus on LLM-related applications (Section 2.2). After this general analysis of methods for scaling test-time compute, I will then elaborate on the most related methods that use multiple samples from an LLM to improve performance (Section 2.3). As this thesis investigates scaling laws for repeated sampling, I then elaborate on how past work investigates other neural network properties through a scaling law lens (Section 2.4). After discussing these general test-time compute methods, I focus on how people have applied additional test-time compute to software engineering, and also give a broader overview of the AI for software engineering domain (Section 2.5). The gains in coverage observed with repeated sampling are only useful if there is a verification mechanism that can select a correct sample from the large sample collections. In domains without external verification methods, this necessitates introducing a selection mechanism into the pipeline. In Section 2.6, I give an overview of relevant work for these selection and verification mechanisms. Finally, I conclude with a broad overview of systems research that has accelerated high-throughput workloads, with a focus on their application to test-time compute methods (Section 2.7).

2.1 Transformers and Language Models

The transformer architecture has enabled significant improvements in state-of-the-art language models [113]. One popular application of the transformer architecture is Large Language Models (LLMs), which work by modeling the distribution of large-scale text data. This is typically done using a causal decoder-based transformer which models the probability of each token, conditioned on all of the previous tokens in the sequence. Auto-regressive text generation works by iteratively sampling a next token from the model's output distribution. Temperature sampling gives a mechanism to control how much stochasticity is involved during generation by modulating the model's logits. Another common sampling technique to generate higher quality text is nucleus sampling [49], which restricts next token candidates to the highest tokens that occupy p percent of the probability mass, where p is a hyper-parameter. In my repeated sampling setup, I generate diverse candidates by running multiple generations using a positive temperature, and control quality for a subset of tasks using nucleus sampling.

A defining feature of transformer-based LLMs is that their performance consistently improves with scaling up data and model size [90, 21, 28, 47, 83]. These scaling laws have given researchers and developers confidence that increases in training scale will continue to lead to increased capabilities. LLM-powered assistants such as ChatGPT have been widely adopted and are currently used by over a hundred million users [77]. This underscores the significant impact of any capability or efficiency improvement for transformers.

2.2 Scaling Inference Compute

Methods that perform additional computation during inference have been successful across many areas of deep learning. Across a variety of game environments, state-of-the-art methods leverage inference-time search to examine many possible future game states before deciding on a move [23, 100, 20]. Similar tree-based methods can also be effective in combination with LLMs, allowing models to better plan and explore different approaches [127, 17, 107, 111]. Another axis for increasing LLM inference compute allows models to spend tokens deliberating on a problem before coming to a solution [125, 121, 129]. Additionally, multiple models can be ensembled together at inference time to combine their strengths [117, 25, 82, 115, 57]. Yet another approach involves using LLMs to critique and refine their own responses [75, 15]. Notably, Snell et al. [101] highlight that scaling inference compute for LLMs can be used to offset differences in model size. They study multiple inference scaling methods (searching against a verifier, iterative model revisions) on the MATH dataset with an inference budget up to 512 parallel samples focusing on final score. In contrast, this thesis focuses on the single method of repeated sampling and demonstrates the predictability of coverage as a function of parallel samples across four orders of magnitude on multiple tasks.

2.3 Repeated Sampling

Previous work has demonstrated that repeated sampling can improve LLM capabilities in multiple domains. One of the most effective use cases is coding [92, 26, 64], where performance continues to scale up to a million samples and verification tools (e.g. unit tests) are often available to automatically score every candidate solution. Recently, Greenblatt [43] shows that repeated sampling is effective when solving puzzles from the ARC challenge [27], observing log-linear scaling as the number of samples increases. In chat applications, repeated sampling combined with best-of-N ranking with a reward model can outperform greedily sampling a single response [56]. In domains without automatic verification tools, existing work shows that using majority voting [120], prompting an LLM [35], or training a model-based verifier [30, 71, 50, 118, 61], to decide on a final answer can improve performance on reasoning tasks relative to taking a single sample. Nguyen et al. [79] finds that performing majority voting over answers that exceed a threshold length can outperform voting across all answers. Concurrent with this thesis, Song et al. [102] finds that using the best available sample improves LLM performance on chat, math, and code tasks, sweeping up to a max of 128 samples. Additionally, Hassid et al. [44] find that when solving coding tasks, it can be more effective to draw more samples from a smaller model than draw fewer samples from a larger one.

2.4 Scaling Laws

Characterizing how scaling affects model performance can lead to more informed decisions on how to allocate resources. Scaling laws for LLM training find a power law relationship between loss and the amount of training compute and provide estimates for the optimal model and dataset size given a fixed compute budget [46, 62, 48]. Jones [59] finds scaling laws in the context of the board game Hex, observing that performance scales predictably with model size and the difficulty of the problem. Interestingly, they also show that performance scales with the amount of test-time compute spent while performing tree search. Recently, Shao et al. [97] observe scaling laws when augmenting LLMs with external retrieval datasets, finding that performance on retrieval tasks scales smoothly with the size of the retrieval corpus. Similar to this thesis, Wu et al. [122] also characterize the performance of models with more test-time compute. Differently than this thesis, they study multiple inference time strategies including a novel tree search method, but focus on only MATH and GSM8K with a sample budget corresponding to mid-hundreds of parallel samples.

2.5 AI for Software Engineering

There has been considerable interest in applying LLMs to coding tasks [26, 68, 73]. Progress in this area can be measured with a diverse set of benchmarks that test models' abilities to complete functions from a prompt [26, 14], build entire libraries from a specification [131], and perform domain-specific programming tasks [86, 106]. This thesis focuses on SWE-bench [58], 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 [29], while current approaches can solve more than 60% [4, 88, 104]. This improvement can be attributed to stronger underlying models [10, 84, 78] 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 [124, 96]. Other approaches enforce a more strict structure on the issue resolution process, e.g. by introducing state machines [136] or a dedicated sequence of steps [123]. Agentless [123] partitions issue resolution into steps for identifying relevant context, generating candidate edits, and selecting between these edits.

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 [124, 96, 119]. Other tools are more narrow, such as editing files with the search-and-replace format introduced by Aider [41, 136, 123]. Some frameworks also provide models with tools for identifying relevant codebase context, such as by opening files or running a semantic search [136, 124, 96]. 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 [136] or by providing them with execution or tool call feedback [119, 51, 93]. Notably, with Anthropic's framework for showcasing the updated Claude 3.5 Sonnet [96], Claude sometimes used hundreds of turns of feedback before submitting a correct solution. Agentless [123] and the Gemini team [78] 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 [93] incorporate model-based selection. Methods like SWE-search [11] and CodeStory [88] combine serial and parallel scaling by performing a tree search over intermediate states using model-based value estimation. This thesis further explores this design space of inference time compute algorithms for LLMs, with a focus on selection methods that leverage tools for verification such as execution feedback.

2.6 Model-based Verification

Benefiting from coverage increases requires a mechanism to select a correct sample from large sample collections. In domains without external tools for verification, this necessitates learning or introducing a verification mechanism that assesses the quality or correctness of samples. A common training paradigm for verification involves collecting a dataset of samples with associated correctness labels, then finetuning a lightweight head that predicts a scalar corresponding to the correctness label conditioned on the sample [31, 116]. Sample level correctness labels can come from human annotation, or model-estimated rewards [128]. Distinct from these Outcome-based Reward Models (ORMs) which evaluate the correctness of entire samples, Process Reward Models (PRMs) evaluate the correctness of every step in a generation and have shown very promising results [72, 112]. The overall correctness of the sample can then be summarized as the mean or max correctness over all steps. This step-level data can either be manually annotated [72], or estimated using Monte-Carlo samples [74, 55] that sample many completions starting from a given step, and estimate the value of the step as the fraction of those that are correct. Recent work has also shown with a specific parameterization of an outcome-based reward model (using a similar trick to DPO [91]) and modeling the reward as a ratio between new and reference logprobs), step-level rewards can be implicitly acquired [32]. Another emerging line of work is generative reward models [76, 130]. These models work by incorporating chain-of-thought mechanisms during verification in order to spend additional test-time compute deliberating on a final verification. LLM-as-a-judge methods [133] use pretrained LLMs to do this deliberation and final verification. In order to assess the quality of these reward models, RewardBench [66] tests models against different applications including reasoning benchmarks such as distinguishing between correct and incorrect math samples, as well as chat-based benchmarks where correct answers are decided based on human preferences. Recent work has also highlighted the importance of strong verifiers in repeated sampling setups [103]. They show that in some scenarios with high false positive rates and negative utility for passing incorrect samples, repeated sampling can be detrimental to overall performance.

2.7 LLM Systems

As methods that leverage increased test-time compute become increasingly used, corresponding systems changes that enable more efficient inference become increasingly important. Especially in repeated sampling workloads, managing large KV caches is a challenge when deploying LLMs. MQA [98] and GQA [8] modify the transformer architecture in order to reduce KV cache size, by decreasing the number of key-value attention heads and assigning multiple query heads to a single key-value head. Alternative approaches operate at a systems level, dynamically moving keys and values between GPU memory, CPU memory, and disk [99, 9, 53]. vLLM [65] introduces a virtual paging system that enables fine-grained KV cache management. This virtual paging can also avoid redundant storage of a prefix's keys and values. Concurrent with the work in this thesis, SGLang [134] also investigates and optimizes inference with sequences that have complicated prompt sharing patterns. Their RadixAttention algorithm dynamically scans incoming requests to find the largest subsequence that has already been processed, avoiding the recomputation of overlapping keys and values. Importantly, while both vLLM and RadixAttention avoid redundant storage of overlapping keys and values, unlike Hydragen, they do not optimize the attention computation itself.

Algorithms that leverage an understanding of the underlying hardware platform can significantly improve device utilization. Hardware-awareness has significantly improved the efficiency of the attention operation [89, 34, 33], reducing the memory requirements from N^2 to N while improving execution time by avoiding redundant memory transfers. In addition to improving input-output (IO) transfers, many GPU-aware algorithms (including Hydragen) focus on leveraging tensor cores [39], which can achieve over 10x more FLOPs per second than the rest of the GPU.

3 | Scaling Test-Time Compute with Repeated Sampling

In this chapter, I explore how the simple method of repeated sampling can be used to predictably improve model capability in certain domains. I focus on pass-fail tasks where a candidate solution can be scored as right or wrong. The primary metric of interest for these tasks is the *success rate:* the fraction of problems that are solved. With repeated sampling, I consider a setup where a model can generate many candidate solutions while attempting to solve a problem. The success rate is therefore influenced both by the ability to generate correct samples for many problems (i.e. coverage), as well as the ability to identify these correct samples (i.e. precision).

The difficulty of the precision problem depends on the availability of tools for sample verification. When proving formal statements in Lean, proof checkers can quickly identify whether a candidate solution is correct. In contrast, the tools available for verifying solutions to math word problems from GSM8K and MATH are limited, necessitating additional verification methods that select a single final answer among many (often conflicting) samples. Occupying a middle ground are tasks such as software engineering, where unit tests provide strong external verification, but may not be comprehensive. In Chapter 4, I investigate this setting in more detail.

I consider the following five tasks:

1. **GSM8K:** A dataset of grade-school level math word problems [30]. I evaluate on a random subset of 128 problems from the test set.

2. **MATH:** Another dataset of math word problems that are generally harder than those from GSM8K [24]. Similarly, I evaluate on 128 random problems from this dataset's test set.

3. **MiniF2F-MATH:** A dataset of mathematics problems that have been formalized into proof checking languages [132]. I use Lean4 as the language, and evaluate on the 130 test set problems that are formalized from the MATH dataset.

4. **CodeContests:** A dataset of competitive programming problems [69]. Each problem has a text description, along with a set of input-output test cases (hidden from the model) that can be used to verify the correctness of a candidate solution. I enforce that models write their solutions using Python3.

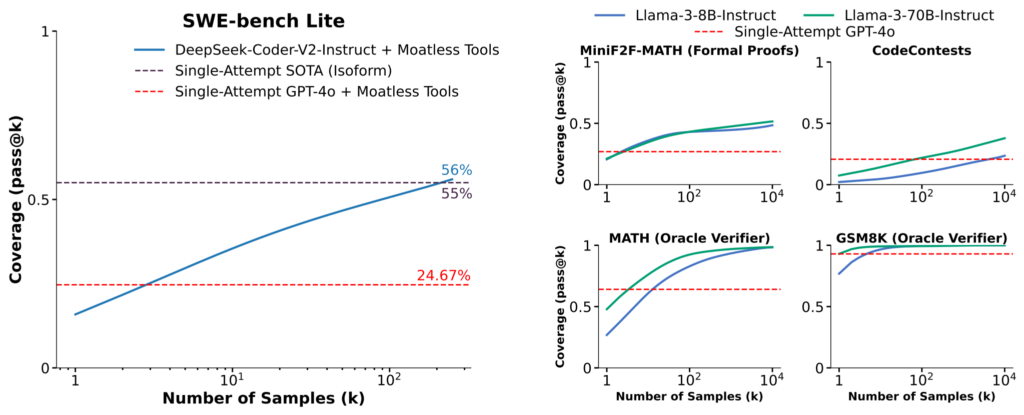


Figure 3.1: Across five tasks, coverage (the fraction of problems solved by at least one generated sample) increases with the number of samples. Notably, using repeated sampling, solve rates increase using an open-source method from 15.9% to 56% on SWE-bench Lite.

5. **SWE-bench Lite:** A dataset of real world GitHub issues, where each problem consists of a description and a snapshot of a code repository [58]. To solve a problem, models must edit files in the codebase (in the Lite subset of SWE-bench that I use, only a single file needs to be changed). Candidate solutions can be automatically checked using the repository's suite of unit tests.

Among these tasks, MiniF2F-MATH contains an oracle verifier in the form of the Lean4 proof checker, and CodeContests and SWE-bench Lite have tools for verification in the form of test cases and unit test suites, respectively. I begin by investigating how repeated sampling improves model coverage. Coverage improvements correspond directly with increased success rates for tasks with automatic verifiers and in the general case provide an upper bound on the success rate. In coding settings, the definition of coverage is equivalent to the commonly-used pass@k metric [26], where k denotes the number of samples drawn per problem. I use this metric directly when evaluating on CodeContests and SWE-bench Lite. For MiniF2F the metric is similar, with a “pass” defined according to the Lean4 proof checker. For GSM8K and MATH, coverage corresponds to using an oracle verifier that checks if any sample “passes” by outputting the correct final answer. To reduce the variance when calculating coverage, I adopt the unbiased estimation formula from Chen et al. [26]. In each experiment, I first generate N samples for each problem index i and calculate the number of correct samples C_i. I then calculate the pass@k scores at each $k \leq N$ of interest according to:

$$\text{pass@k} = \frac{1}{\text{\# of problems}} \sum_{i=1}^{\text{\# of problems}} \left(1 - \frac{\binom{N-C_i}{k}}{\binom{N}{k}}\right) \tag{3.1}$$

I use the numerically stable implementation of the above formula suggested in Chen et al. [26].

3.1 Repeated Sampling is Effective Across Tasks

In this Section, I establish that repeated sampling improves coverage across multiple tasks and a range of sample budgets. I evaluate Llama-3-8B-Instruct and Llama-3-70B-Instruct [42] on CodeContests, MiniF2F, GSM8K, and MATH, generating 10,000 independent samples per problem. For SWE-bench Lite, I use DeepSeek-Coder-V2-Instruct [37], as the required context length of this task exceeds the limits of the Llama-3 models. As is standard when solving SWE-bench issues, I equip the LLM with a software framework that provides the model with tools for navigating through and editing codebases. For these experiments, I use the open-source Moatless Tools library [136]. Note that solving a SWE-bench issue involves a back-and-forth exchange between the LLM and Moatless Tools. One sample for this benchmark refers to one entire multi-turn trajectory. To minimize costs, I restrict the number of samples per issue to 250, with all samples being drawn independent of one another.

I report the results in Figure 3.1. I also include the single-sample performance of GPT-4o on each task, as well as the (pass@1) state-of-the-art for SWE-bench Lite (Isoform [6]). Across all five tasks, I find that coverage smoothly improves as the sample budget increases. When drawing a single sample from all LLMs, GPT-4o outperforms the Llama and DeepSeek models at every task. However, as the number of samples increases, all three of the weaker models exceed GPT-4o's single-sample performance. In the case of SWE-bench Lite, 56% of problems are solved, exceeding the single-attempt SOTA of 55%.

3.2 Repeated Sampling is Effective Across Model Sizes and Families

The results from Section 3.1 indicate that repeated sampling improves coverage across multiple tasks. However, I only show this trend for three recent, instruction-tuned models with 8B or more parameters. I now show that these trends hold across other model sizes, families, and levels of post-training. I expand the evaluation to include a broader set of 13 models:

- **Llama 3:** Llama-3-8B, Llama-3-8B-Instruct, Llama-3-70B-Instruct.

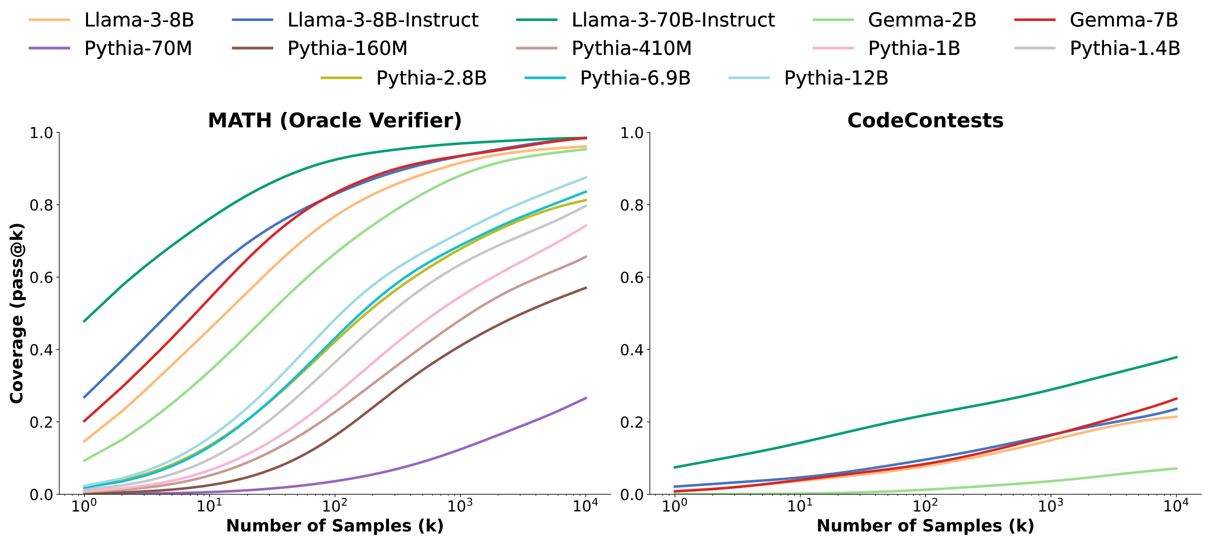


Figure 3.2: Scaling inference time compute via repeated sampling leads to consistent coverage gains across a variety of model sizes (70M-70B), families (Llama, Gemma and Pythia) and levels of post-training (Base and Instruct models).

- **Gemma:** Gemma-2B, Gemma-7B [105].
- **Pythia:** Pythia-70M through Pythia-12B (eight models in total) [18].

I restrict evaluation to the MATH and CodeContests datasets to minimize inference costs, reporting results in Figure 3.2. Coverage increases across almost every model I test, with smaller models showing some of the sharpest increases in coverage when repeated sampling is applied. On CodeContests, the coverage of Gemma-2B increases by over 300x, from a pass@1 of 0.02% to a pass@10k of 7.1%. Similarly, when solving MATH problems with Pythia-160M, coverage increases from a pass@1 of 0.27% to a pass@10k of 57%.

The exception to this pattern of increasing coverage across models is with the Pythia family evaluated on CodeContests. All Pythia models achieve zero coverage on this dataset, even with a budget of 10,000 samples. I speculate that this due to Pythia being trained on less coding-specific data than Llama and Gemma.

3.3 Repeated Sampling Can Help Balance Performance and Cost

One takeaway from the results in Sections 3.1 and 3.2 is that repeated sampling makes it possible to amplify a weaker model's capabilities and outperform single samples from stronger models. Here, I demonstrate that this amplification can be more cost-effective than using a stronger,

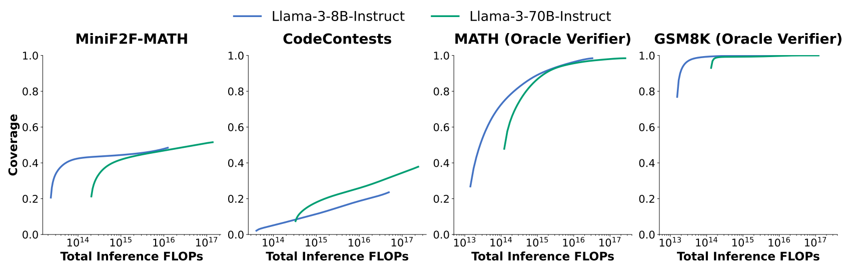


Figure 3.3: Comparing cost, measured in number of inference FLOPs, and coverage for Llama-3-8B-Instruct and Llama-3-70B-Instruct. The ideal model size depends on the task, compute budget, and coverage requirements. Note that Llama-3-70B-Instruct does not achieve 100% coverage on GSM8K due to an incorrectly labelled ground truth answer: see Appendix D.

more expensive model, providing practitioners with a new degree of freedom when trying to jointly optimize performance and costs.

I first consider FLOPs as a cost metric, examining the Llama-3 results from Section 3.1. I re-plot the results from Figure 3.1, now visualizing coverage as a function of total inference FLOPs instead of the sample budget. See Appendix F for details on FLOP calculations.

I present re-scaled results for MiniF2F, CodeContests, MATH, and GSM8K in Figure 3.3. Interestingly, the model that maximizes coverage for a given compute budget varies with the budget and task. On MiniF2F, GSM8K and MATH, Llama-3-8B-Instruct always obtains higher coverage than the larger (and more expensive) 70B model when the FLOP budget is fixed. However for CodeContests, the 70B model is almost always more cost effective. Note that examining FLOPs alone can be a crude cost metric that ignores other aspects of system efficiency [38]. In particular, repeated sampling can make use of high batch sizes and specialized optimizations that improve system throughput relative to single-attempt inference workloads [60, 13, 135].

I also examine the dollar costs of repeated sampling when solving SWE-bench Lite issues using API pricing[1]. Keeping the agent framework (Moatless Tools) constant, I consider making a single attempt per issue with Claude 3.5 Sonnet and GPT-4o, as well as repeated sampling using DeepSeek-Coder-V2-Instruct. We report the average cost per issue and issue resolution rate with each approach in Table 3.1. While the DeepSeek model is weaker than the GPT and Claude models, it is also over 10x cheaper. In this case, repeated sampling provides a cheaper alternative to paying a premium for access to strong models while achieving a superior issue solve rate.

[1]Accessed June 2024.

Model	Cost per attempt (USD)	Number of attempts	Issues solved (%)	Total cost (USD)	Relative total cost
DeepSeek-Coder-V2-Instruct	0.0072	5	29.62	10.8	1x
GPT-4o	0.13	1	24.00	39	3.6x
Claude 3.5 Sonnet	0.17	1	26.70	51	4.7x

Table 3.1: Comparing API cost (in US dollars) and performance for various models on the SWE-bench Lite dataset using the Moatless Tools agent framework. When sampled more, the open-source DeepSeek-Coder-V2-Instruct model can achieve the same issue solve-rate as closed-source frontier models for under a third of the price.

3.4 Scaling Laws for Repeated Sampling

Training scaling laws are able to accurately characterize the relationship between an LLM's loss and its training compute [46, 62, 48]. These laws have empirically held over many orders of magnitude and inspire confidence in model developers that continued investment in training will pay off. Crucially, scaling laws for training are predictive: they allow us to estimate the loss of a training run using the losses of much smaller runs. In this section, I attempt to derive analogous inference scaling laws for repeated sampling that can be similarly used for prediction. I will specifically test the predictive power of a candidate scaling law by extracting a subset of 1000 samples from the 10,000 generations and evaluating the scaling law's prediction of pass@10k.

3.4.1 Initial Candidate: Geometric CDFs

Given a fixed problem, model, and set of sampling hyperparameters (e.g. temperature), the model will generate a correct solution with some probability $p \in [0, 1]$. Therefore, the probability that a model will generate at least one correct answer when drawing k IID samples (i.e. the expected coverage c for a single problem) follows the CDF of a geometric distribution: $c = 1 - (1 - p)^k$

Across a dataset of n problems, expected coverage when drawing k samples is therefore given by:

$$c = \frac{1}{n} \sum_{i=1}^{n} [1 - (1 - p_i)^k]$$

where $p_i \in [0, 1]$ denotes the per-sample correctness probability of problem i.

While this formula is exact, it is hard to use it practically for prediction. This ground-truth model requires estimating p_i for every problem in the dataset, but the estimates may be inaccurate when based off of coverage values from small numbers of samples. In particular, the p_i estimates will be zero for hard problems that are not solved by any of the samples used for fitting, causing the scaling law to predict that these problems will never be solved, no matter how many samples are drawn. This highlights one of the implicit requirements for a good scaling law: it must be able to model the distribution of problem difficulties in the dataset, including problems that have not yet been solved.

Instead of estimating each problem's p_i, it is possible to try to fit a mixture of geometric distributions with a smaller number of components. When fitting these mixtures to the full coverage curves, they become increasingly accurate as I increase the number of components from one to five (Figure C.1). However, these mixtures perform poorly when fitted to only the pass@k values[2] for $1 \leq k \leq 100$, producing an average pass@10k prediction error of 22.61% across all tasks and models (Figure C.3 and Figure C.4).

3.4.2 Exponentiated Power Laws

In search of alternative candidates to capture the scaling properties of repeated sampling, I turn to the GPT-4 technical report [85]. There, the authors find that the relationship between a model's mean-log-pass-rate on coding problems and its training compute at a dataset level can be well-captured with a power law. I adopt the same function class but now using it for inference compute, modeling the log of coverage c (rather than the pass-rate) as a function of the number of samples k: $\log(c) \approx ak^b$, where $a, b \in \mathbb{R}$ are fitted model parameters. In order to directly predict coverage, I exponentiate both sides, ending up with the final model of $c \approx \exp(ak^b)$. I provide examples of fitted coverage curves in Figure 3.4, and additional curves in Appendix C.2 demonstrating that an exponentiated power law can approximate the coverage curves up to 1.05% mean average error across all models and datasets. In Figure 3.5 and Figure C.6 I show the results of predicting coverage up to 10k samples using the pass@k values for $k \leq 100$ using the same procedure as above. Unlike the mixture of geometrics, the coverage prediction generalizes to a greater number of samples, and on average, leads to a mean pass@10k prediction error of 2.53% across all settings excluding MiniF2F-MATH.

[2]Since the prediction setup uses a subset of 1000 samples, I only fit the scaling laws to values of $k \leq 100$ to ensure the estimates of pass@k are stable.

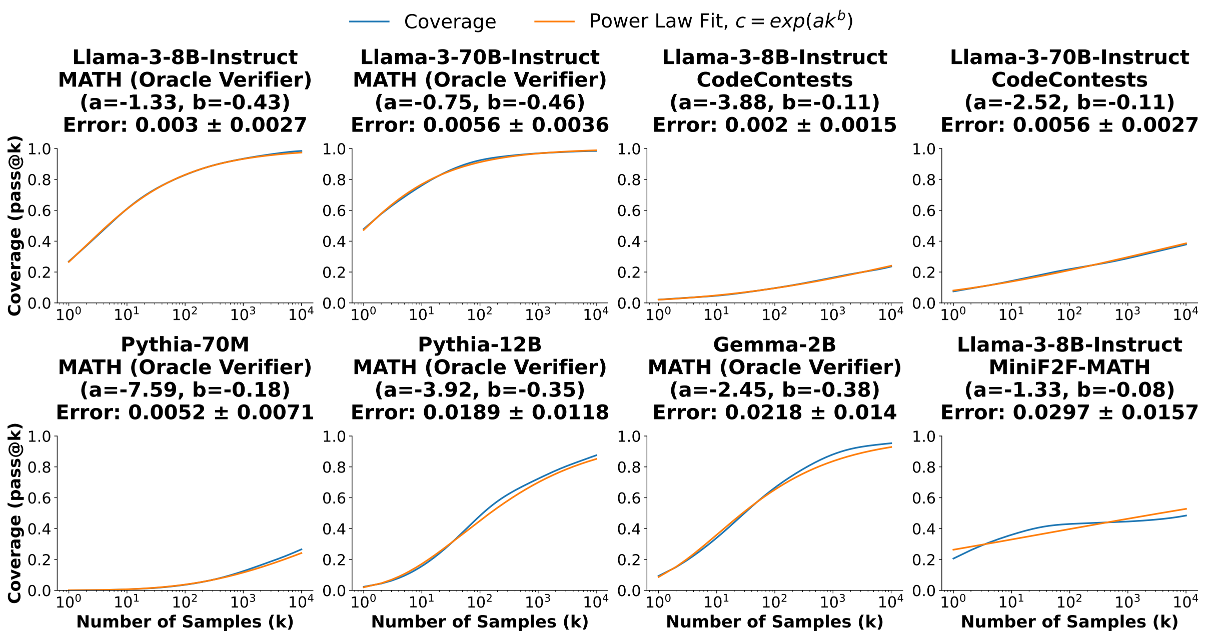


Figure 3.4: The relationship between coverage and the number of samples can be modelled with an exponentiated power law for most tasks and models. Note that some curves, such as Llama-3-8B-Instruct on MiniF2F-MATH, do not follow this trend closely. I show the mean and standard deviation of the error between the coverage curve and the power-law fit across 100 evenly sampled points on the log scale.

3.5 Characterizing the Precision Problem

So far, I have shown that repeated sampling leads to large and often predictable gains in coverage across all tasks studied. I now turn to the complementary problem of precision: given a collection of model samples, how often can a correct one be identified?

For some tasks, strong verification tools exist that can automatically assess all samples from a model and identify correct ones. For example, the correctness and performance of generated GPU kernels can be tested by comparing against a reference implementation [87]. Other tasks in this category include solving math proofs in a formal language such as Lean, hardware simulation, and transpiling code from one language to another for speed, security, or to replace legacy software.

Of the five tasks I evaluate in Section 3.1, only GSM8K and MATH lack tools for verification. In the rest of this section, I focus on characterizing the difficulty of the precision problem for these tasks. I show that the frequency of correct answers for many problems is very low (Section 3.5.1) and analyze the faithfulness of the chains of thought for these correct and rare samples (Section 3.5.2).

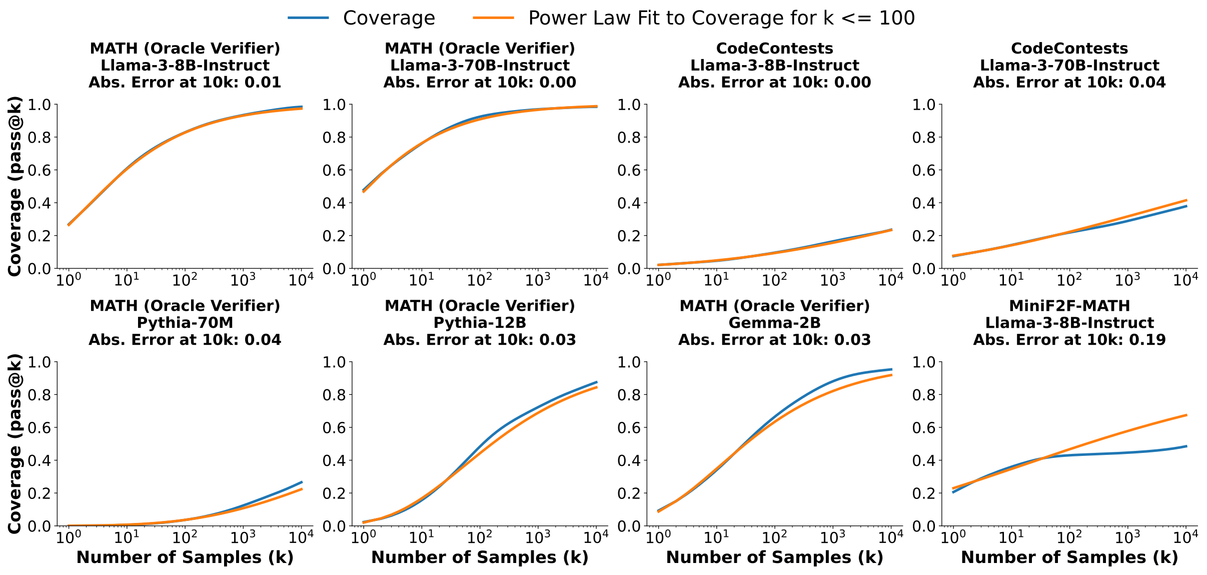


Figure 3.5: Predicting coverage values for high k by fitting a power law to the coverage curve for $k \leq 100$. Note that I recalculate the pass@k values used to do the power-law fitting using a random 1 k subset of the datapoints.

3.5.1 Many Problems Have a Long Tail of Correct Answers

In Section 3.1, coverage on GSM8K and (in particular) MATH continues to increase as the number of samples are scaled across several orders of magnitude. These increases mean that there are many problems in the datasets where correct solutions are sampled highly infrequently. As the number of samples increases and rare, correct solutions are generated for new problems, coverage improves. I visualize this in Figure 3.6, where I plot the distribution of sample correctness frequencies across GSM8K and MATH problems. I also indicate the problems where majority voting (also known as self-consistency [120]) is successfully able to select a correct answer. Since majority voting simply identifies the plurality answer among samples, it cannot harvest the tail of problems where correct answers are very rare. Rather, for a verifier to be truly scalable and convert the coverage improvements that are obtained with large-scale sampling into higher success rates, it must be able to find “needles in the haystack” and identify infrequent correct samples.

3.5.2 Manually Inspecting the Reasoning of Correct Samples

Given the importance of identifying these needle-in-a-haystack samples, it is reasonable to wonder how “hard” the task of verification is in these cases. With GSM8K and MATH, only a sample’s final answer is used for assessing correctness, with the intermediate chains of thought

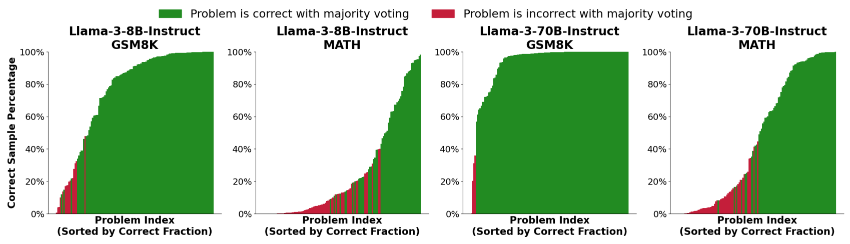


Figure 3.6: Bar charts showing the fraction of samples (out of 10,000) that are correct for each of the GSM8K and MATH problems that are evaluated. Each problem is represented with one bar, with the height of the bar corresponding to the fraction of samples with a correct final answer. Bars are green if self-consistency picked the correct answer and are red otherwise. Note that there are many problems with correct solutions, where the correct solutions are sampled infrequently.

being discarded. If models generated only nonsensical chains of thought before guessing a correct final answer, verification may not be any easier than solving the problem in the first place. I investigate this by manually inspecting the chains of thought for 105 GSM8K samples from Llama-3-8B-Instruct, 56 MATH samples from Llama-3-8B-Instruct, and 46 MATH samples from Llama-3-70B-Instruct. The results of this study are in Table 3.2. I find that the faithfulness of the chains of thought varies depending on both the task and problem difficulty. For the easier GSM8K dataset, 90% of the chains of thought are faithful, even for problems that the models only solve less than 10% of the time[3]. However, for the more difficult MATH dataset, the chain of thought faithfulness is largely correlated with the problem difficulty. Only one third of the inspected correct samples contain correct intermediate reasoning for problems the model gets correct less than 10% of the time. For these samples with incorrect reasoning, the model's chains of thought are coherent but contain incorrect assumptions or calculations. Overall, this indicates that for some tasks, fully recovering increases in coverage may be difficult. However, past work [71] shows that using a reward model to select the best answer can continue to improve success rate at least until 2048 samples. This provides promise that at least a portion of the gains observed in coverage can be translated to success rate for similar tasks that do not contain tools for verification.

[3]Interestingly, during this process I also identified one GSM8K problem that has an incorrect ground truth answer (see Appendix D). This incorrect GSM8K problem is also the only one that Llama-3-70B-Instruct did not generate a "correct" sample for across 10,000 attempts.

Dataset/Model	**Pass@1**	**# Problems**	**# CoTs Graded**	**# Correct**	**# Incorrect**	**% Correct**
GSM8K	0-10%	5	15	11	1*	73.3%
Llama-3-8B	10-25%	10	30	27	3	90.0%
	25-75%	29	30	28	2	93.3%
	75-100%	84	30	30	0	100%
MATH	0-0.1%	7	17	5	12	29.4%
Llama-3-8B	0.1-1%	17	10	1	9	10.0%
	1-10%	31	10	4	6	40.0%
	10-50%	42	10	6	4	60.0%
	50-100%	29	10	10	0	100%
MATH	0-0.1%	2	6	2	4	33.3%
Llama-3-70B	0.1-1%	6	10	4	6	40.0%
	1-10%	27	10	5	5	50.0%
	10-50%	29	10	6	4	60.0%
	50-100%	62	10	10	0	100%

*1 problem (3 CoTs) has an incorrect ground truth label, see Appendix D

Table 3.2: Manually evaluating the intermediate reasoning in model-generated chains-of-thought from the MATH and GSM8K samples. I run the evaluation by bucketing problems by difficulty (defined by pass@1), randomly sampling problems from each bucket, and randomly sampling several chains-of-thought per problem (three chains per problem for GSM8K, two per problem for MATH). I find that GSM8K chains-of-thought have correct intermediate reasoning most of the time for all difficulty buckets, while intermediate reasoning from MATH chains-of-thought are less likely to be correct for harder questions.

4 | Improving Software Engineering with Test-Time Compute

In the previous chapter, I showed that repeated sampling can significantly improve coverage for many tasks. One important application that I demonstrated this for was software engineering in the form of solving Github issues in SWE-bench Lite. However, I also showed that benefiting from these gains in coverage can be non-trivial in domains without verification tools to automatically select a correct sample. Additionally, the results in the previous chapters use an off-the-shelf SWE-bench framework (Moatless Tools [136]) designed for generating a single solution; multiple candidate edits were generated by simply sampling from this framework repeatedly with a positive temperature. This motivates the investigation: how can we design a system where benefiting from test-time compute was a primary consideration with final accuracy as the metric (i.e instead of coverage).

In this chapter, I investigate this idea, presenting a system for solving SWE-bench Verified[1] instances designed specifically around scaling test-time compute (Figure 1.1). I 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 [123]. I 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. I scale parallel test-time compute by sampling many of these (edit, test) pairs for every SWE-bench issue (i.e. repeated sampling). This combination of scaling achieves 69.8% coverage on SWE-bench Verified. Interestingly, I find that different ways of allocating an inference budget between serial and parallel scaling often lead to similar coverage values. Additionally, the use of parallel scaling enables amortizing the cost of identifying relevant codebase context across multiple downstream samples. I 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, I explore methods based on voting with model-generated tests [93, 123] and directly using models to do selection [93, 78]. I find that a

[1]A split of SWE-bench where all instances have been manually verified by humans. This split was not released when doing experiments in previous chapters, I avoid re-running those due to compute limitations.

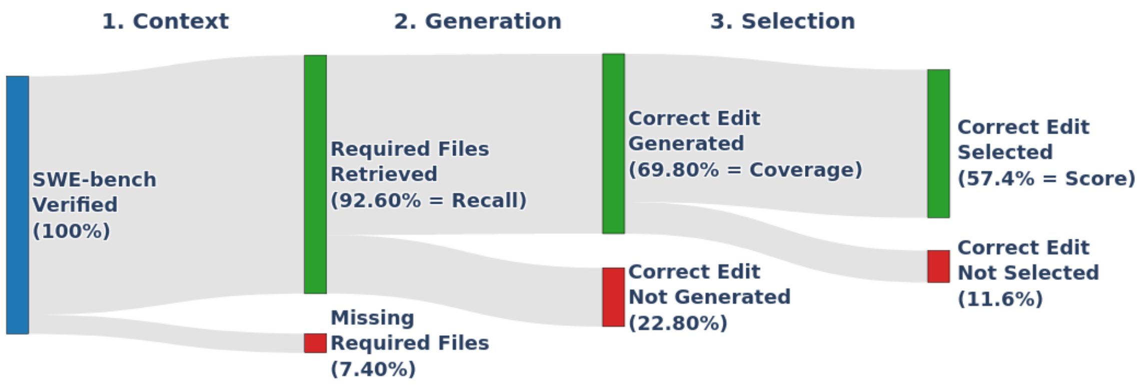


Figure 4.1: Measuring CodeMonkeys performance across the three subtasks identified in Section 4 (context, generation, and selection). Note that modifying the approach to one subtask can influence the performance on other subtasks as well. For example, generating more candidate edits could increase coverage but make selection harder.

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, I 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 4.2) while spending approximately 2300 USD on LLM inference (Table 4.1).

I also show that this approach to selection can be used to effectively combine generations from heterogeneous sources. I 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%).

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 this Chapter, I focus on the Verified [29] split of SWE-bench, which contains instances that human annotators have classified as “solvable” (i.e. by having unambiguous issue descriptions and test suites that do not filter out correct solutions). I decompose solving an instance from SWE-bench Verified into three sequential subtasks (Figure 1.1):

1. **Context:** Can the required codebase files be identified[2]? I can measure outcome of this

[2]When calculating recall, I determine if a file needs to be edited by checking if the file is edited in the official

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 4.1: Breaking down the costs of running CodeMonkeys on all GitHub issues from SWE-bench Verified. All costs are in USD. The system uses two LLMs: a primary model used for the ranking, generation, and selection stages (I use the Claude 3.5 Sonnet API [10]), and a cheaper model used for scanning codebases to identify relevant files (I run Qwen2.5-Coder-32B-Instruct [54] locally). For measuring costs with the Claude API, I 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 H.

subtask with **recall:** the fraction of problems where all needed files have been identified.

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

3. **Selection:** Can a correct codebase edit be selected from the collection of candidates? After completing this subtask, I can measure the final **score:** the fraction of problems in the dataset that are resolved by the submitted edit.

I describe the approach to each subtask below, with per-subtask metrics reported in Figure 4.1 and a cost breakdown of the system in Table 4.1. Unless otherwise noted, all parts of the system use `claude-3-5-sonnet-20241022` [10]. I present additional results using DeepSeek-V3 [36] in Appendix G.

4.1 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

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.

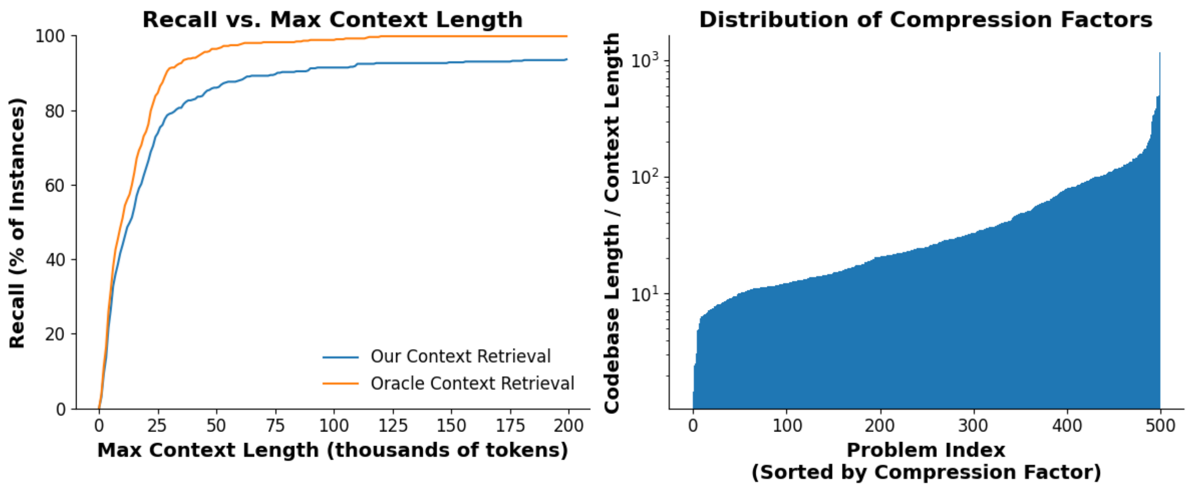


Figure 4.2: **Left**: Measuring recall (the fraction of SWE-bench problems with context windows that contain all needed files) as I increase the context window size limit. With the 128k token limit that I use for later experiments, 92.6% of instances have the correct files in context. **Right**: Visualizing 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 I include after relevance + ranking.

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 [136, 123], iterative expansion from a file tree [123], and giving models tools for browsing files [124, 96].

With CodeMonkeys, I know that I will generate a collection of candidate edits for every instance. Therefore, by choosing to share codebase context across all downstream edits, I can amortize the cost of context generation. This observation enables the simple approach of letting a model (I use Qwen2.5-Coder-32B-Instruct [54]) read every file in the codebase[3], decide whether each is relevant to the target issue, and only include relevant files in the context window [12]. Performing this codebase-wide scan once per instance contributes less than 15% to the total system costs (Table 4.1) and on average processes 2.94M tokens per problem. If I did not amortize this scan and instead reran it for each of the 10 edits I 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, I perform a model-based ranking procedure to order files by importance. First, as part of the initial codebase scan, I generate a concise summary for every file flagged as

[3]I only include Python files in the scan and exclude files inside of testing directories.

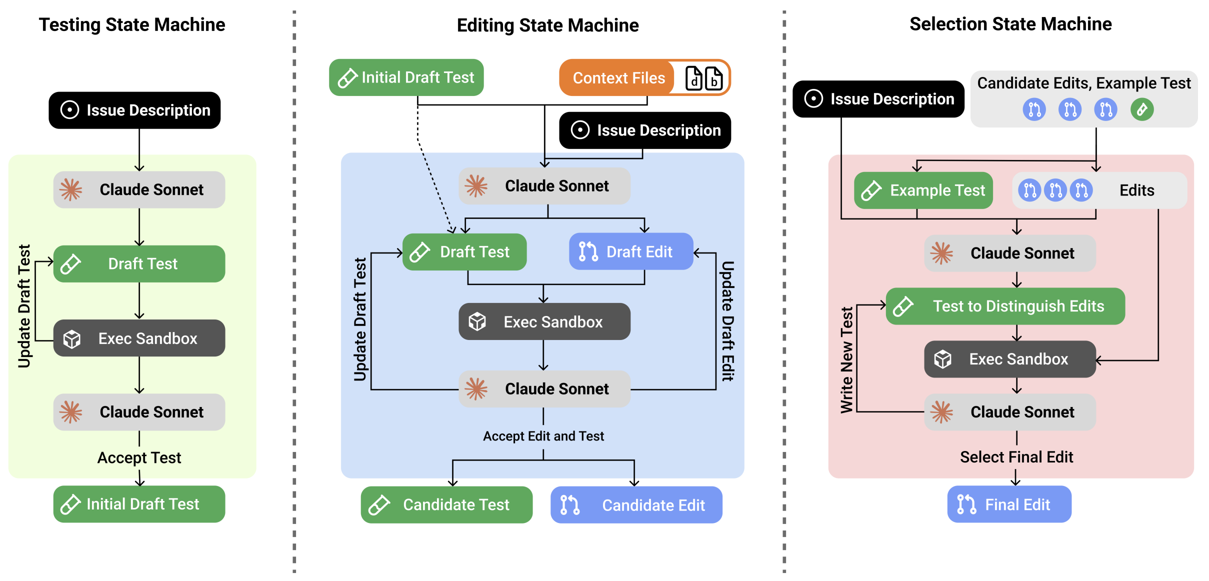


Figure 4.3: Details of the CodeMonkeys state machines. The **Testing State Machine** iteratively generates an initial draft of a testing script based on execution feedback from running the test on the codebase before any edits are applied. The **Editing State Machine** first generates an initial edit 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.

relevant that describes how the file relates to the target issue. I then construct a ranking prompt that includes each relevant file's name, summary, and token count. I 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), I generate three completions from the ranking prompt and construct a final ranking by considering each file's average rank across the three repetitions. I 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.

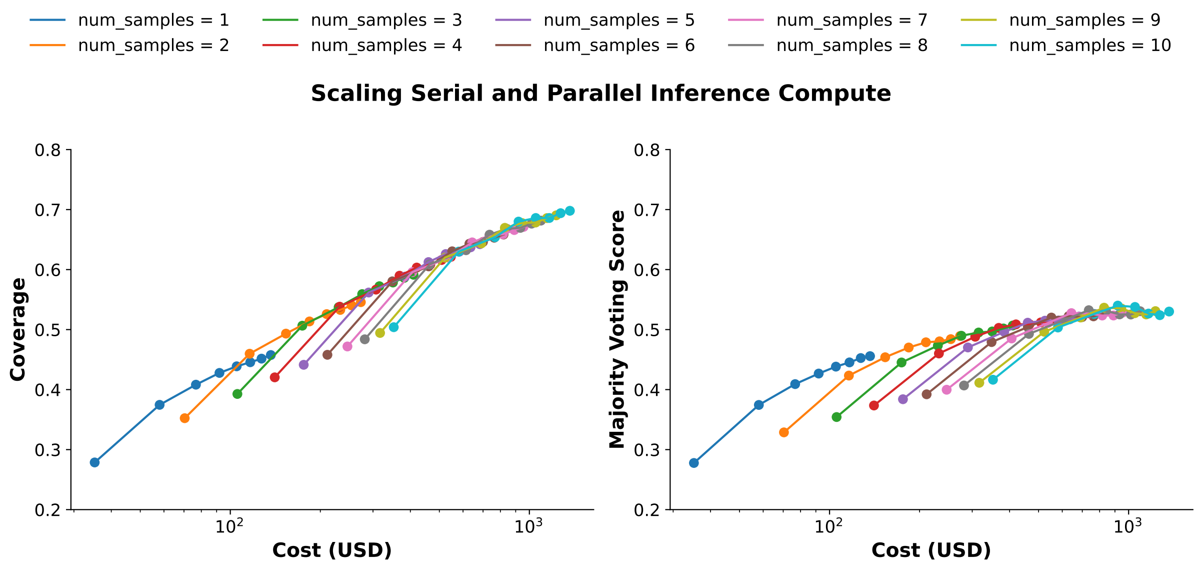


Figure 4.4: Measuring coverage (left) and score when using majority-voting selection (right) as I sweep over the number of serial iterations per editing state machine and the number of parallel state machines sampled per problem. Each colored curve corresponds to a different number of parallel state 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.

4.2 Generating Candidate Codebase Edits with Corresponding Tests

With relevant parts of the codebase identified, I can begin generating candidate codebase edits for solving the target issue. I adopt a state machine abstraction [136] 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 [19], I run multiple independent state machines for every SWE-bench instance, using a positive sampling temperature to introduce diversity across candidates. This approach provides two straightforward ways to scale test-time compute:

1. I can scale serial compute by increasing the maximum number of iterations performed per state machine[4].
2. I can scale parallel compute by increasing the number of independent state machines run per instance.

Importantly, I require that models generate and revise a test jointly with each edit. These tests

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

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 edits provides models with richer feedback to guide their iteration and scale serial compute more effectively [52]. Additionally, tests serve as (imperfect) verifiers that can later assist with selecting between candidate edits (see Section 4.3).

Empirically, I 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, I decompose this stage of the 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 [41]). This state machine is seeded with the output of a testing state machine and can also revise the testing script as needed.[5]

The structure of these state machines is visualized in Figure 4.3 (left and middle panels), with additional details given in Appendix G.1. The decomposition into separate testing and editing state machines also lowers system costs. The cost table (Table 4.1) 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, I can reduce prompt lengths (and therefore cache read costs) in the testing state machine by omitting the files identified in Section 4.1. I additionally reduce prompt lengths by clearing the chat history between the testing and editing state machines.

I 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 4.4, I measure coverage as I sweep over both scaling parameters. The 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

[5]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.

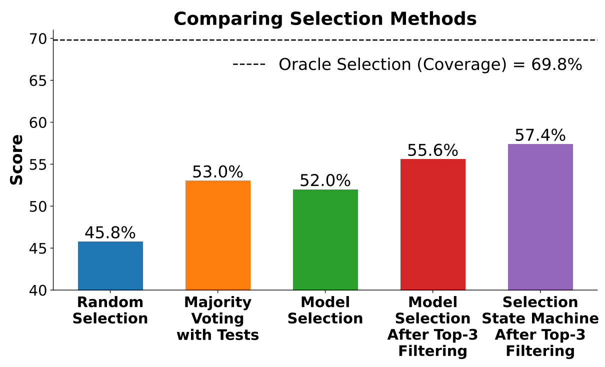


Figure 4.5: Comparing the selection methods when applied to the candidate edits generated by the CodeMonkeys editing state machines. The best performing selection method – the selection state machine after top-3 filtering with generated tests – recovers approximately half of the difference between the 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 4.2: Comparing final SWE-bench Verified scores between the methods explored in this Chapter (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.

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 my setup only indirectly allows scaling serial compute. I 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.

4.3 Selecting Between Candidate Edits

Under a best-case scenario with oracle selection (where final score equals the 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, I instead selected edits at random, the 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. I explore four strategies for selecting among the 10 candidate edits that are generated per instance:

1. **Majority Voting with Tests:** I 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, I compute the expected score when picking randomly among them.
2. **Model Selection:** I 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:** I reuse the model selection prompt above, but select among only the three edits that pass the most generated tests. I break ties by favoring edits with shorter git diffs.
4. **Selection State Machine After Top-3 Filtering:** WeIupgrade 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 4.3 on the right and Appendix G.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, I show the model the outputs when running the test on every candidate edit in addition to the unedited codebase. I reuse the top-3 filtering process described above to narrow down the initial pool of candidates.

I compare the performance of these selection methods in Figure 4.5. 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 I test; however, it contributes less than 10% to total system costs. I 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 the chosen approach of selection state machine + top-3 filtering, CodeMonkeys achieves a final score of 57.4% on SWE-bench Verified (Table 4.2), closing approximately half of the gap between random and oracle selection.

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

In CodeMonkeys, I select among IID candidate edits generated by my method's state machines. Here, I demonstrate that the selection state machine is additionally helpful when combining candidate edits coming from heterogeneous sources. I 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[6]: Blackbox AI Agent [4], CodeStory Midwit Agent + swe-search [88], Learn-by-interact [104], and devlo [5]. 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), I 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.

Since the Barrel of Monkeys begins with fewer initial candidates per problem than CodeMonkeys, I skip the 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 [4] with a score of 62.8%). However, since randomly selecting from this ensemble yields a score of 60.9%, the selection method recovers a smaller proportion of the gap between random selection and coverage here relative to selecting from the editing state machine.

[6]As of January 15, 2025.

5 | Accelerating Test-Time Compute Workloads with Hydragen

As methods for test-time compute become increasingly used, corresponding systems for running them efficiently become increasingly critical. In the previous chapter, I analyzed systems considerations for test-time compute oriented systems. In this chapter, I go one step down in the stack, and investigate the design of Hydragen: an exact attention implementation specifically designed for the special case of doing attention over a large batch of sequences that contain shared prefixes. This setting occurs when using repeated sampling where many samples are generated from the same starting prompt. Beyond repeated sampling, other common examples of this setting include a chatbot serving many users with shared system instructions (Figure 1.1 left), an assistant model using a few-shot prompt for solving domain-specific tasks [21], and competitive programming systems that sample many candidate solutions for a single problem [70]. As transformer-based LLMs [113] are deployed at increasingly large scales [77], improving their efficiency with shared prefixes can have a significant impact. In this work, I use a hardware-aware perspective to analyze and optimize this inference setting.

Shared prefixes create overlaps in the attention keys and values across sequences, presenting an opportunity for specialized optimization. Existing work [65] identifies that naive KV caching leads to redundant storage of the prefix's keys and values, and addresses this redundancy with a paged memory management strategy. While this optimization can significantly reduce GPU memory consumption, it does little to affect the speed of computing attention, which can often bottleneck end-to-end throughput with large batches. Since each sequence in the batch has a distinct (albeit overlapping) KV cache but only a single attention query when decoding, existing attention implementations like FlashAttention [34, 33] and PagedAttention [65] compute attention by performing many independent matrix-vector products. This approach is memory-bound when the KV cache is large, and moreover does not use hardware-friendly matrix multiplications. Both of these characteristics lead to poor performance on modern GPUs. Across successive hardware generations, GPU computational capability has improved at a significantly faster rate than memory bandwidth. Additionally, an increasingly large fraction of total GPU floating-point operations (FLOPs) are only available when using tensor cores, a specialized hardware feature that is dedicated to performing matrix-matrix products and not matrix-vector products (Figure 1.1 bottom right).

In this chapter, I demonstrate that shared prefixes enable more than just memory savings, and can additionally be used to improve decoding throughput. I identify that FlashAttention and PagedAttention redundantly read the prefix's keys and values from GPU memory when computing attention, regardless of whether the prefix is redundantly stored. In order to eliminate these redundant reads, I present Hydragen, an exact implementation of attention that is specialized for shared prefixes (Figure 1.1 middle). Hydragen decomposes full-sequence attention into separate attention computations over the prefix and suffixes. These sub-computations can be cheaply combined to recover the overall attention result (Section 5.2.1). With attention decomposition, Hydragen is able to efficiently compute attention over the prefix by batching together attention queries across sequences (Section 5.2.2). This inter-sequence batching replaces many matrix-vector products with fewer matrix-matrix products (Figure 1.1 top right), reducing redundant reads of the prefix and enabling the use of tensor cores.

Experimentally, I find that Hydragen can significantly improve LLM throughput in large-batch settings with shared prefixes. In end-to-end benchmarks, Hydragen increases the throughput of CodeLlama-13b [92] by up to 32x over vLLM [65], a high-performance inference framework that avoids redundant prefix storage but not redundant prefix reads. The attention operation in isolation can be accelerated by over 16x using Hydragen when compared to a state-of-the-art FlashAttention baseline, with benefits increasing as the batch size and shared prefix length grow. I also demonstrate that Hydragen's efficient processing of shared prefixes can influence algorithmic decisions on how to use LLMs most effectively. With a large batch size, Hydragen allows the shared prefix to grow from 1K tokens to 16K tokens with less than a 15% throughput penalty whereas vLLM throughput decreases by over 90%. On long document question answering tasks, I show that Hydragen can process 256 questions in less time than it takes a FlashAttention baseline to process 64 (Section 5.4.1). Finally, I demonstrate that Hydragen's attention decomposition and batching apply to more general patterns of prompt sharing than a single prefix-suffix split. When solving APPS competitive programming problems [45], where two levels of prompt sharing occur, I apply Hydragen hierarchically to maximize sharing and reduce evaluation time by an additional 55% over a single-level of prompt sharing (Section 5.4.2).

5.1 Background on Machine Learning Systems

5.1.1 Hardware Efficiency Considerations

GPU Performance Bottlenecks: GPUs possess a limited number of processors for performing computation and a limited amount of bandwidth for transferring data between processors and memory. When a program running on a GPU is bottlenecked waiting for compute units to finish processing, it can be classified as compute-bound. Alternatively, memory-bound programs are bottlenecked accessing GPU memory. To summarize a program's use of hardware resources, we can calculate its arithmetic intensity, defined as the total number of arithmetic operations performed divided by the total number of bytes transferred. Higher arithmetic intensities imply a greater use of computational resources relative to memory bandwidth.

Batching: Batching is a common optimization that can increase an operation's arithmetic intensity and reduce memory bottlenecks. Consider the example of computing matrix-vector products. To compute one product, each element of the input matrix is read from memory but is used in only a single multiply-accumulate. Therefore, the arithmetic intensity of the operation is low, and is memory-bound on GPUs. However, if many matrix-vector products need to be computed using the same matrix, we can batch the operations together into a single matrix-matrix product. In the batched operation, the cost of reading the input matrix is amortized over the batch of vectors. Each element of the input matrix is now used for many multiply-accumulates, increasing the arithmetic intensity of the overall operation and improving hardware utilization.

Tensor Cores: Modern GPUs (and other AI accelerators) are designed with specialized units for efficiently computing matrix multiplications. Effectively using these resources can be crucial for achieving good overall performance; on GPUs, tensor cores dedicated to matrix multiplications can compute over 10x more floating-point operations per second (FLOPS) than the rest of the GPU. This further motivates batching matrix-vector products into matrix-matrix products.

5.1.2 Attention and LLM Inference

The focus of this Chapter is optimizing attention in transformer-based LLMs. Scaled-dot-product (SDP) attention operates on a sequence of queries $Q \in^{N_q \times d}$, keys $K \in^{N_{kv} \times d}$, and values $V \in^{N_{kv} \times d}$, producing an output $O \in^{N_q \times d}$ defined as:

$$O = \text{SDP}(Q, K, V) = \frac{QK^T}{\sqrt{d}}V \tag{5.1}$$

I am particularly interested in the performance characteristics of attention during LLM text generation, as this is the computation that is exemplified with test-time compute algorithms. Generation begins with a prefill stage that processes the starting sequence of tokens that the LLM will complete. The prefill phase encodes the entire prompt in parallel using a single transformer forward pass. Therefore, when computing attention we have $N_q = N_{kv} \gg 1$ and as a result the multiplications in Equation 5.1 involving K^T and V are hardware-friendly matrix multiplications. After the prefill stage, completion tokens are iteratively decoded from the model, with one decoding step producing one new token and requiring one forward pass. Decoding is accelerated by the use of a KV cache, which stores the attention keys and values of all previous tokens in the sequence. The KV cache avoids the need for reprocessing the entire sequence during every decoding step, and instead only the most recent token is passed through the model. However, this leads to an attention computation where $N_q = 1$ while $N_{kv} \gg 1$, making the multiplications with K^T and V matrix-vector products. Attention during decoding is therefore memory-bound and does not use tensor cores.

5.1.3 Batched Inference

LLM inference throughput can be increased by generating text for a batch of sequences in parallel. With batched decoding, each forward pass of the model processes the most recent token from many sequences instead of only one. This batching increases the arithmetic intensity of transformer components such as the multilayer perceptron (MLP) blocks and allows these modules to use hardware-friendly matrix multiplications. However, batched text generation does not increase the intensity of attention, since every sequence has a distinct key and value matrix. Therefore, while other model components are able to use tensor cores during batched decoding, attention must be computed using many independent matrix-vector products. With large batch sizes or long sequence lengths, computing attention becomes increasingly expensive relative to rest of the transformer, decreasing throughput. Additionally, the storage footprint of the KV cache in GPU memory can exceed that of the model parameters when the batch size is large, imposing constraints on the maximum number of sequences that can be simultaneously processed.

5.1.4 Shared Prefixes

In this chapter, I investigate improving the throughput of batched text generation when the sequences in the batch share a common prefix. This scenario lends itself to specialized optimizations because shared prefixes lead to overlaps across sequences' key and value matrices. The causal attention mask in LLMs results in each token's activations being influenced only by previous tokens in the sequence. Therefore, if multiple sequences share a common prefix, the keys and values corresponding to the prefix tokens will be identical across sequences.

The key-value overlap introduced by shared prefixes presents two distinct directions for improving the inference process described in Section 5.1.3. Firstly, naive batched inference stores the KV cache separately for every sequence, leading to redundant storage of the prefix key and value vectors. Existing work has identified this redundancy and proposed an elegant virtual memory system to eliminate duplicate storage [65].

In this Chapter, I identify an additional opportunity to optimize the attention operation itself. When GPU kernels compute attention for each sequence in the batch using independent matrix-vector products, the prefix keys and values are repeatedly read from GPU memory, regardless of whether they are stored redundantly or not. I now propose an alternative approach to computing attention, which can simultaneously eliminate these redundant reads and enable the use of tensor cores.

5.2 Hydragen: Efficient Attention with Shared Prefixes

Hydragen is an exact implementation of attention that is optimized for shared prefixes, and is a combination of two techniques:

1. **Attention Decomposition:** Split full-sequence attention into separate attention computations over the shared prefix and unique suffixes that can be cheaply combined to recover the full attention result.
2. **Inter-Sequence Batching:** Efficiently compute attention over the prefix by batching together attention queries across sequences.

Attention decomposition allows isolating overlapping portions of the batch's key and value matrices, while inter-sequence batching exploits this overlap by replacing many matrix-vector products with a single matrix-matrix product. Pseudocode implementing Hydragen attention is provided in Appendix J.

5.2.1 Decomposing Attention Across Subsequences

As discussed in Section 5.1.4, sequences that share a common prefix have partially overlapping keys and values when computing attention. Hydragen seperates this computation with partial overlap into two separate operations: attention over the shared prefix, where there is total key-value overlap, and attention over unique suffixes, where there is no overlap.

Consider the general case where keys K and values V are partitioned across N_{kv} (the sequence/row dimension) into:

$$K = K_1 || K_2 \tag{5.2}$$

$$V = V_1 || V_2 \tag{5.3}$$

with $||$ denoting concatenation. We wish to avoid directly computing the desired quantity SDP $(Q, K, V))$, and instead calculate this value using the results of the sub-computations SDP $(Q, K_1, V_1))$ and SDP $(Q, K_2, V_2))$.

The challenge in partitioning attention is with the softmax operation, since the softmax denominator is calculated by summing over all exponentiated attention scores in the sequence. In order to combine the sub-computations, I use a denominator rescaling trick inspired by FlashAttention's blocked softmax computation [34]. When computing SDP $(Q, K_1, V_1))$ and SDP $(Q, K_2, V_2))$, I additionally compute and store the log-sum-exp (LSE$(Q, K) \in^{N_q}$) of the attention scores (equivalently, the log of the softmax denominator):

$$Q, K = \log\left(\text{sum}\left(\exp\left(\frac{QK^T}{\sqrt{d}}\right), \text{dim} = 1\right)\right) \tag{5.4}$$

Given the two partitioned attention outputs and their LSEs, we can calculate the final result SDP $(Q, K, V))$ by computing the full-sequence softmax denominator and rescaling the attention outputs accordingly:

$$\text{SDP}(Q, K, V) = \frac{\text{SDP}\,(Q, K_1, V_1))e^{Q,K_1} + \text{SDP}\,(Q, K_2, V_2))e^{Q,K_2}}{e^{Q,K_1} + e^{Q,K_2}} \tag{5.5}$$

I prove this formula in Appendix I.

5.2.2 Inter-Sequence Batched Prefix Attention

With attention decomposition, I am able to compute attention over the prefix as a standalone operation for every sequence. While this decomposition does not improve performance on its own (in fact, it introduces additional work in order to combine sub-computation outputs), it enables computing prefix attention much more efficiently over a batch of sequences.

Queries do not affect each other when computing attention, therefore if two sets of queries attend over identical keys and values, they can be merged into a single attention operation with a larger number of queries. With attention decomposition, this case now applies to each sequence's attention over the shared prefix. Since the prefix's keys and values across sequences are identical, Hydragen can batch each sequence's query vector together into one attention operation over a single sequence. Importantly, this batching significantly raises N_q and the arithmetic intensity of prefix attention, replacing many separate matrix-vector products with a single matrix-matrix product. By replacing multiple independent attention computations over the prefix with a single batched operation, Hydragen reduces the number of times that the prefix KV cache is read from GPU memory. Additionally, the tensor cores are now used during prefix attention which significantly improve hardware utilization.

Note that we are unable to apply inter-sequence batching when computing attention over suffixes, since the keys and values in each sequence's suffix are not identical. Suffix attention is therefore computed normally, with a single query per sequence.

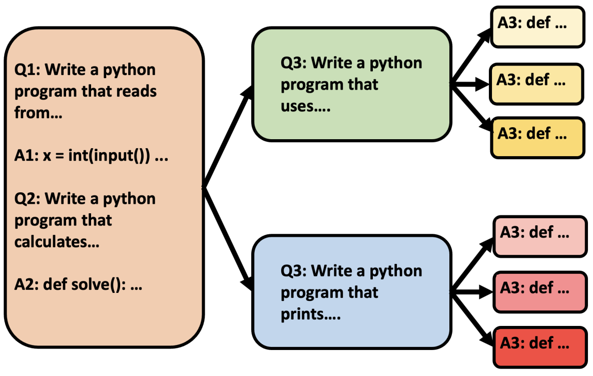

Q1: Write a python program that reads from...
A1: x = int(input()) ...
Q2: Write a python program that calculates...
A2: def solve(): ...
Q3: Write a python program that uses....
Q3: Write a python program that prints....
A3: def ...

Figure 5.1: An example of a batch of sequences with a hierarchical sharing pattern. This diagram depicts the setting of Section 5.4.2, which solves competitive programming problems using a few-shot prompt and by sampling many candidate solutions per problem. The few-shot prompt (orange) is globally shared across all sequences in the batch. However, the descriptions of each problem (green and blue) are only shared across the candidate solutions corresponding to that problem.

5.2.3 Hierarchical Sharing

So far, I have focused on the setting where all sequences in the batch share a common starting subsequence followed by suffixes that are distinct from one another. However, this excludes other forms of sharing that appear in important use cases. Sequences in the batch may not all start with a global prefix, and instead the batch may be divided into groups of overlapping sequences. Additionally, sharing may be more fine-grained than a simple prefix-suffix decomposition, with the overlap between sequences forming a tree structure where each node contains a token sequence that is shared by all descendants (see Figure 5.1 for an example). These forms of sharing are increasingly relevant as LLMs are applied in more complicated inference/search algorithms [126, 16, 80].

Hydragen naturally generalizes to these richer forms of sharing as well. To apply Hydragen to a tree of sequences, I replace attention decomposition over the prefix and suffix with attention decomposition at every vertex in the tree. I can then use inter-sequence batching across levels of the tree, so that the keys and values associated with one node in the tree are shared across the queries of all descendant nodes.

5.2.4 Estimating Throughput Improvements with Hydragen

Hydragen significantly improves the efficiency of attention with shared prefixes relative to approaches that compute attention independently for every sequence (Section 5.4). However, translating this targeted efficiency into end-to-end throughput improvements depends strongly on the details of the inference setting being considered. In order for Hydragen to meaningfully improve decoding speed in a particular setting, attention must be a major contributor to decoding time. For example, with small batch sizes or short sequence lengths, decoding speed is often bottlenecked not by attention, but by reading the parameters of the model from GPU memory. The benefits of Hydragen in this scenario will therefore be minimal. Similarly, given a fixed batch size and sequence length, we expect Hydragen to improve throughput more on a model that uses multi-headed attention than a similarly-sized model that uses multi-query attention [98] or grouped-query attention [8] in order to reduce the size of the KV cache. However, reducing the KV cache size allows for a larger batch size to fit within GPU memory constraints, which can further increase the speedup of using Hydragen.

As discussed in Section 5.1.3, the cost of attention becomes disproportionately high as the batch size grows, since the arithmetic intensity of most transformer operations improve while attention remains memory-bound. Hydragen greatly improves the hardware utilization of attention, making the comparison of attention FLOPs to other model FLOPs more useful when determining the maximum achievable speedup. In several experiments in Section 5.3, I include a "No Attention" baseline that only runs the non-attention components of the transformer in order to establish an upper bound for attainable throughput.

Another important consideration when predicting the benefits of Hydragen is the relative number of prefix (shared) tokens compared to suffix (unshared) tokens. Since Hydragen makes no optimizations to attention over suffixes, long suffixes can decrease generation throughput. I explore the impact of suffix length on attention speed in Section 5.4.

5.2.5 Implementation

I implement Hydragen for the Llama family of models [109, 110, 92]. The implementation is simple: I use no custom CUDA code and write Hydragen entirely in PyTorch[1] plus calls to a fast attention primitive. This contrasts with more sophisticated algorithms like PagedAttention, which require bespoke GPU kernels to read from and update the paged KV cache. I believe that Hydragen's simplicity will allow it to be easily ported to other hardware platforms such as TPUs, which also have hardware dedicated to fast matrix multiplications. In my implementation, I use version 2.3.6 of the `flash-attn` package when attending over the prefix, and a Triton kernel

[1]For non-hierarchical inputs, I have also written a Triton kernel for combining softmax denominators.

from xformers when attending over the suffix. The second kernel enables changing sequence lengths in the suffix KV cache across decoding steps while still adhering to the constraints required to use CUDA graphs.

5.3 Experiments

5.3.1 End-To-End Throughput

I benchmark end-to-end LLM throughput in the setting where many completions are sampled from a single prompt. This is a common technique for improving a model's ability at solving math and coding problems [92, 70]. The benchmarks evaluate Hydragen against four baselines:

1. **FlashAttention:** I perform inference without any shared prefix optimizations, as if all sequences in the batch were fully distinct. I compute full-sequence attention using the Triton kernel that Hydragen uses for suffix attention, and otherwise use the same codebase as Hydragen. This baseline redundantly stores the prefix's keys and values for every sequence in the batch, causing this method to run out of memory quickly.

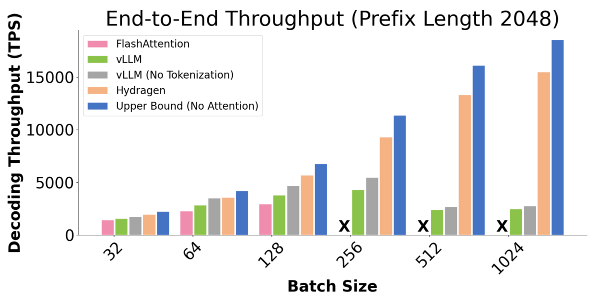
2. **vLLM:** I use version 0.2.7 of the `vllm` package, which uses the PagedAttention algorithm. vLLM avoids redundant storage of the prefix, allowing much larger batch sizes to be tested. Additionally, because of this non-redundant storage, PagedAttention can achieve a higher GPU cache hit rate when reading the prefix, reducing the cost of redundant reads.

3. **vLLM without Detokenization:** I disable incremental detokenization in vLLM (accomplished by commenting out one line in the vLLM codebase), which we observed to improve throughput.

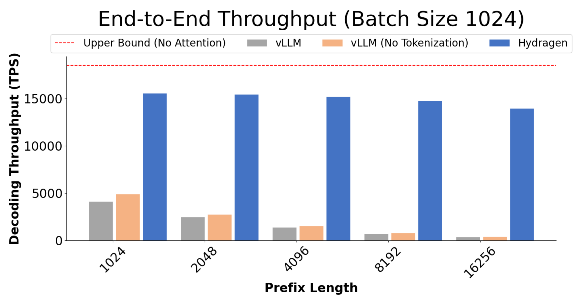
4. **No Attention:** I skip all self-attention computations in the transformer. This (functionally incorrect) baseline provides a throughput ceiling and helps to illustrate the cost of different attention implementations relative to the rest of the transformer. Note that the query, key, value, and output projections in the attention block are still performed.

I run the benchmarks on CodeLlama-13b [92] and distribute the model with tensor parallelism across eight A100-40GB GPUs in order to have enough GPU memory to store the KV cache with large batch sizes. In Figure 5.2(a), I fix the prefix length to 2048 and sweep over the batch size while generating 128 tokens per completion. When the batch size is small, non-attention operations contribute significantly to decoding time, with all methods reaching at least half of the throughput of no-attention upper bound. At these low batch sizes, Hydragen, the vLLM baselines, and the FlashAttention baselines have similar throughputs. However, as the batch size grows and attention over the prefix becomes increasingly expensive, Hydragen begins to significantly outperform the other baselines.

In Figure 5.2(b), I run a similar experiment, except now I hold the batch size constant at 1024 and sweep over the shared prefix length. The throughput of vLLM significantly decreases as the prefix grows, from just under 5k tokens/second with a prefix length of 1024 to less than 500 tokens/second with a prefix length of 16256. However, with Hydragen, throughput is much less unaffected despite the prefix growing by over 15k tokens. Moreover, across all sequence lengths tested, Hydragen throughput is always within 70% of the no-attention ceiling. I perform more in-depth sweeps over different models, prefix lengths, batch sizes, and numbers of generated tokens in Appendix K.1 - for smaller models and shorter completions lengths, Hydragen's speedup can exceed 50x. Additional evaluation setup details are in Appendix L.1.



(a)



(b)

Figure 5.2: Left: End-to-end decoding throughput in tokens per second (TPS) with CodeLlama-13b when generating multiple completions from a prompt containing 2048 tokens. An "x" indicates that FlashAttention does not have enough memory to run. As the batch size grows, Hydragen achieves a significantly higher throughput than vLLM baselines. Throughput with Hydragen always remains within 50% of the upper bound where attention is entirely removed from the model. Details are in Section 5.3.1. Right: Comparing decoding throughput of CodeLlama-13b between Hydragen, vLLM (with and without tokenization), and "No Attention", where the attention operation is removed from the model to demonstrate the throughput ceiling. In this scenario where the batch size is fixed, Hydragen improves throughput by up to 32x over the best baseline, with speedups increasing with prefix length.

5.4 Microbenchmarking Attention

I also perform more granular benchmarks comparing Hydragen attention against FlashAttention, in order to more precisely demonstrate the performance characteristics of the method. The microbenchmarks run on a single A100-40GB using eight query attention heads, one key and value head, and a head dimension of 128 (matching the setting of CodeLlama-34b when distributed across eight GPUs). I sweep over different batch sizes, prefix lengths, and suffix lengths, reporting results in Figure 5.3. The microbenchmarks corroborate the end-to-end measurements from Section 5.3.1 that the speedup with Hydragen increases as the batch size and prefix lengths grow. However, the microbenchmarks also highlight the significant impact of the suffix length on inference time. Hydragen computes attention over suffixes using memory-bound FlashAttention (without inter-sequence batching). As the suffix lengths grow, reading this portion of the KV cache becomes an increasingly significant contributor to total execution time. When generating text using Hydragen, this means that the first tokens decoded by the model are generated the fastest, with throughput decreasing over time as the lengths of completions (and therefore the lengths of suffixes) grow.

These microbenchmarks are also influenced by the hardware platform that they are run on. GPUs with a higher ratio of compute to memory bandwidth benefit more from Hydragen eliminating memory bottlenecks when attending over the prefix. I report results on other GPUs in Appendix K.2 and provide more evaluation details in Appendix L.2.

5.4.1 Long Document Question Answering

Additionally, I explore the performance of Hydragen on workloads involving very long documents. I construct a document by embedding synthetic facts into an excerpt of *War and Peace* [108]. The shared prefix, totalling 19947 tokens, contains both the document as well as five few-shot examples of question/answer pairs. The benchmark evaluates Yi-6B-200k [7] on its ability to answer questions based on the embedded facts. I run this benchmark across four A100-40GB GPUs using Hydragen in addition to the FlashAttention and no-attention baselines. Results are reported in Figure 5.4. I observe that processing time for the FlashAttention baseline rapidly grows far beyond the time of the no-attention baseline, highlighting how attention is the dominant operation for this configuration. Meanwhile, Hydragen's processing time remains

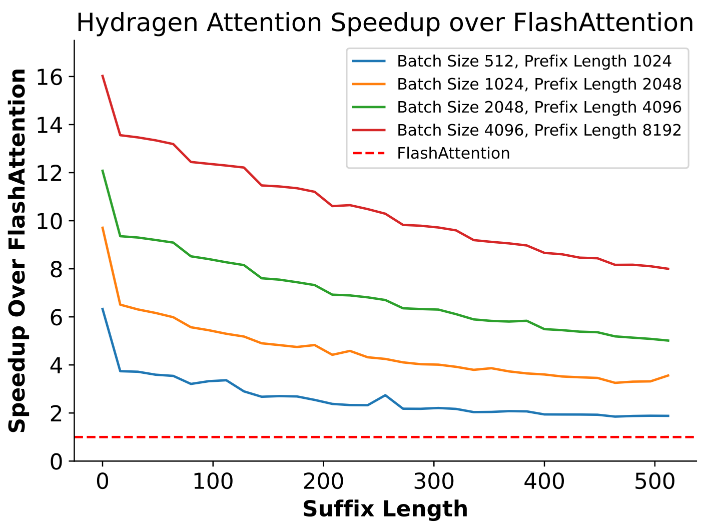


Figure 5.3: Measuring the speedup of Hydragen attention over FlashAttention across various batch sizes, shared prefix lengths and suffix lengths on a single A100-40GB GPU. I see that Hydragen results in faster inference in all cases, in particular when the ratio of shared length to unique length is high and the batch size is large. I observe even larger performance gains when running on an L40S (a GPU with a higher compute-to-bandwidth ratio than an A100), shown in in Figure K.1.

within 60% of the no-attention optimum. Notably, Hydragen can process 256 questions in less time than it takes the FlashAttention baseline to process 64 questions. I provide additional evaluation details in Appendix L.3.

5.4.2 Hierarchical Sharing in Competitive Programming

I lastly demonstrate the benefits of applying Hydragen to a setting with hierarchical sharing. In this experiment, I benchmark the total time required to evaluate CodeLlama-7b on 120 problems from the APPS dataset [45] using a two-shot prompt and 128 candidate programs per problem. When multiple problems are processed in a single batch, prompt overlap occurs across two levels: the few-shot prompt is shared across all sequences in the batch, while each problem's description is shared across all of the problem's candidate solutions (see Figure 5.5).

I run this benchmark using two methods:

1. **Single-Level Hydragen:** I use a single-level version of Hydragen to share the few-shot prompt across all sequences in the batch, and not share problem descriptions across candidate solutions. This leads to redundant storage of the problem description across all candidate solutions, reducing the maximum batch size that can be used.

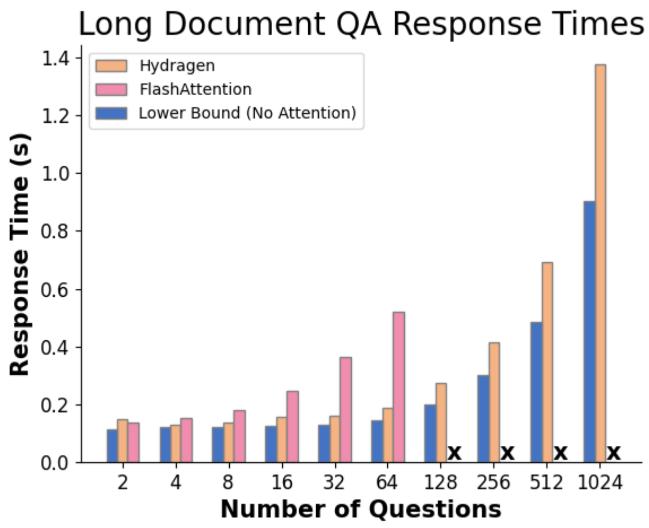


Figure 5.4: Time to answer questions about a 19947 token-long document when benchmarking Yi-6B-200k on four A100-40GB GPUs. An "x" indicates that FlashAttention does not have enough memory to run. Time to process prefix is excluded.

2. **Two-Level Hydragen:** I apply Hydragen across both levels of prompt overlap. This has the dual benefits of improving attention efficiency (by increasing the degree of sharing) as well as avoiding redundant storage, which enables increasing the batch size used for evaluation. I avoid conflating these benefits by evaluating two-level Hydragen twice: once with the same batch size used for single-level Hydragen, and once with an enlarged batch size.

I report results in Figure 5.5. I see that even when the batch size is held constant, adding a second level of sharing to Hydragen can improve attention efficiency and decrease dataset evaluation time by 18%. Furthermore, the memory saved due to not redundantly storing the problem description enables increasing the batch size, which in turn results in an additional 45% reduction in evaluation time. I provide additional evaluation details in Appendix L.4.

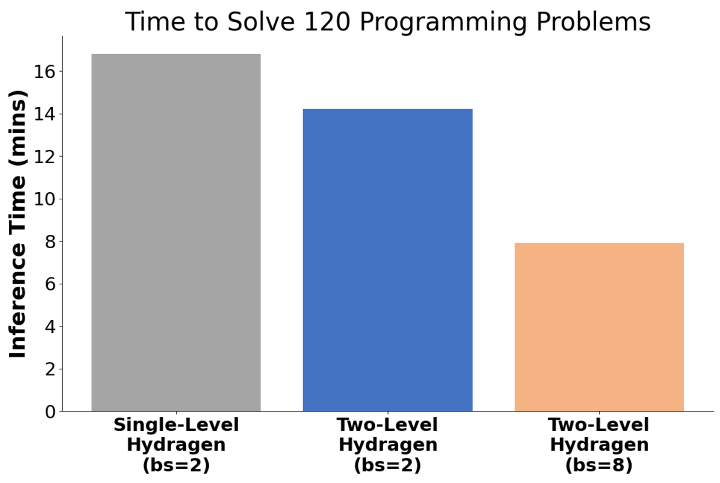


Figure 5.5: Time to run inference over a dataset of 120 APPS coding problems, sampling 128 solutions per problem with two few-shot examples. The batch size refers to the number of problems processed simultaneously. Across all Hydragen runs (both single and two-level), the few-shot prompt is shared across all sequences. By additionally sharing the problem description across generated candidate solutions, two-level Hydragen decreases overall inference time by an extra 55% over single-level Hydragen.

6 | Discussion

6.1 Limitations and Future Work

Improving Repeated Sampling: In my experiments, I explore only a simple version of repeated sampling where all attempts to a problem are generated independently of one another using the exact same prompt and hyperparameters. This setup can be refined to improve performance, particularly along the following directions:

1. **Solution Diversity:** A positive sampling temperature is the sole mechanism for creating diversity among samples. Combining this token-level sampling with other, higher-level approaches may be able to further increase diversity. For example, AlphaCode conditions different samples with different metadata tags.

2. **Multi-Turn Interactions**: In Chapter 4, I showed that performance increases when we allow models to iterate on their solutions with environmental feedback. Expanding this to other domains, such as formal theorem proving with MiniF2F, and rigorously quantifying the trade-offs between the amount of serial multi-turn interaction and parallel samples is a great direction for future study.

3. **Learning From Previous Attempts:** Currently, my experiments fully isolate attempts from each other until the final selection stage. Access to existing samples, particularly if verification tools can provide feedback on them, may be helpful when generating future attempts.

Verifiers: The results from Section 3.5 highlight the importance of improving sample verification methods when tools for automatically doing so are unavailable. Equipping models with the ability to assess their own outputs will allow repeated sampling to be scaled to far more tasks. In Section 4, I explore this selection problem in the domain of software engineering. Although initial results are promising, there is still a gap between the final performance after selection, and the upper bound if an oracle selection mechanism were available. Along this line, training domain-specific verifiers using the wide array of techniques summarized in Section 2.6 is a natural next step. I am also excited about methods that not only train models to be better verifiers, but also methods that train generators to produce solutions that are easier to verify. An alternative direction to developing model-based verifiers is to design converters that can make an unstructured task verifiable, for example by formalizing an informal math statement into a language like Lean so that proof checkers can be applied.

Integrating other test-time techniques: There has been increasing interest in methods that allow users to scale the amount of test-time compute per problem to get better capability. In addition to the repeated sampling approach that I explore, other works take inspiration from game playing agents [23, 100, 20], and explore how tree-based methods can also be used in combination with LLMs to better plan and explore different approaches [127, 17, 107, 111]. Another axis for increasing LLM inference compute allows models to spend tokens deliberating on a problem before coming to a solution [125, 121, 129]. Additionally, multiple models can be ensembled together at inference time to combine their strengths [117, 25, 82, 115, 57]. These methods aren't mutually exclusive, and examining the relationship between repeated sampling and these techniques would strengthen the results and be highly valuable to practitioners.

6.2 Conclusion

In this thesis, I explore repeated sampling as an axis for scaling compute at inference time in order to improve model performance. Across a range of models and tasks, repeated sampling can significantly improve the fraction of problems solved using any generated sample (i.e. coverage). When correct solutions can be identified (either with automatic verification tools or other verification algorithms), repeated sampling can amplify model capabilities during inference. This amplification can make the combination of a weaker model and many samples more performant and cost-effective than using fewer attempts from a stronger, more expensive model.

However, fully benefiting from repeated sampling requires a mechanism for selecting a correct answer from large sample collections. In domains that do not have tools for verification, this thesis showed that the standard approach of self-consistency is insufficient for doing this. As an investigation of how repeated sampling, and other test-time scaling techniques, can still be beneficial without automatic verification, this thesis then focused on how to scale software engineering performance with test-time computation. I explored how using a combination of model-generated unit tests and model selection can provide a significant lift in selection accuracy. This improved selection enabled an 11.6% accuracy increase in real-world software engineering tasks with repeated sampling. In addition to spending more parallel inference compute with a combination of repeated sampling and verification, I showed that spending more serial inference compute in the form of multi-turn tool interactions further improves software engineering performance.

In summary, this thesis demonstrates that test-time compute in the form of repeated sampling, leads to large gains in capability in domains that contain automatic verification and in a targeted study of a relevant domain without automatic verification. Repeated sampling, and other concurrent results showing how capability can be greatly improved with more test-time computation [84, 101]. As these methods become increasingly deployed, it is critical to develop efficient systems for running them. This thesis makes strides to address this challenge by presenting an algorithm for high-throughput attention in settings with large batches that contain overlapping prefixes, such as repeated sampling workloads.

Bibliography

Hello gpt-4o, 2024. URL https://openai.com/index/hello-gpt-4o/.

Claude 3.5 sonnet, 2024. URL https://www.anthropic.com/news/claude-3-5-sonnet.

Voyage ai, 2024. URL https://www.voyageai.com/.

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

Your ai-developer teammate. https://devlo.ai/, 2025.

Isoform, 2025. URL https://www.isoform.ai/.

01-ai. Yi, 2023. URL https://github.com/01-ai/Yi.git. Accessed: 2024-02-01.

Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico LebrÃ$^3n, andSumitSanghai. \quad Gqa \quad : \quad Traininggeneralizedmulti - querytransformermodelsfrommulti - headcheckpoints$, 2023.

Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, et al. Deepspeed-inference: Enabling efficient inference of transformer models at unprecedented scale. In *2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 646–660. IEEE Computer Society, 2022.

Anthropic. Introducing computer use, a new claude 3.5 sonnet, and claude 3.5 haiku, 2024. URL https://www.anthropic.com/news/3-5-models-and-computer-use.

Antonis Antoniades, Albert Ãrwall, Kexun Zhang, Yuxi Xie, Anirudh Goyal, and William Wang. Swe-search: Enhancing software agents with monte carlo tree search and iterative refinement, 2024. URL https://arxiv.org/abs/2410.20285.

Simran Arora, Brandon Yang, Sabri Eyuboglu, Avanika Narayan, Andrew Hojel, Immanuel Trummer, and Christopher RÃ©. Language models enable simple systems for generating structured views of heterogeneous data lakes, 2023. URL https://arxiv.org/abs/2304.09433.

Ben Athiwaratkun, Sujan Kumar Gonugondla, Sanjay Krishna Gouda, Haifeng Qian, Hantian Ding, Qing Sun, Jun Wang, Jiacheng Guo, Liangfu Chen, Parminder Bhatia, Ramesh Nallapati, Sudipta Sengupta, and Bing Xiang. Bifurcated attention: Accelerating massively parallel decoding with shared prefixes in llms, 2024. URL https://arxiv.org/abs/2403.08845.

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. URL https://arxiv.org/abs/2108.07732.

Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, Carol Chen, Catherine Olsson, Christopher Olah, Danny Hernandez, Dawn Drain, Deep Ganguli, Dustin Li, Eli Tran-Johnson, Ethan Perez, Jamie Kerr, Jared Mueller, Jeffrey Ladish, Joshua Landau, Kamal Ndousse, Kamile Lukosuite, Liane Lovitt, Michael Sellitto, Nelson Elhage, Nicholas Schiefer, Noemi Mercado, Nova DasSarma, Robert Lasenby, Robin Larson, Sam Ringer, Scott Johnston, Shauna Kravec, Sheer El Showk, Stanislav Fort, Tamera Lanham, Timothy Telleen-Lawton, Tom Conerly, Tom Henighan, Tristan Hume, Samuel R. Bowman, Zac Hatfield-Dodds, Ben Mann, Dario Amodei, Nicholas Joseph, Sam McCandlish, Tom Brown, and Jared Kaplan. Constitutional ai: Harmlessness from ai feedback, 2022.

Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Michal Podstawski, Hubert Niewiadomski, Piotr Nyczyk, and Torsten Hoefler. Graph of thoughts: Solving elaborate problems with large language models, 2023.

Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, and Torsten Hoefler. Graph of thoughts: Solving elaborate problems with large language models. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(16):17682â17690, March 2024. ISSN 2159-5399. 10.1609/aaai.v38i16.29720. URL http://dx.doi.org/10.1609/aaai.v38i16.29720.

Stella Biderman, Hailey Schoelkopf, Quentin Anthony, Herbie Bradley, Kyle O'Brien, Eric Hallahan, Mohammad Aflah Khan, Shivanshu Purohit, USVSN Sai Prashanth, Edward Raff, Aviya Skowron, Lintang Sutawika, and Oskar van der Wal. Pythia: A suite for analyzing large language models across training and scaling, 2023. URL https://arxiv.org/abs/2304.01373.

Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V. Le, Christopher RÃ©, and Azalia Mirhoseini. Large language monkeys: Scaling inference compute with repeated sampling, 2024. URL https://arxiv.org/abs/2407.21787.

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. ISBN 9781713829546.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020. URL https://arxiv.org/abs/2005.14165.

Murray Campbell, A. Joseph Hoane, and Feng-hsiung Hsu. Deep blue. *Artif. Intell.*, 134 (1â2):57â83, jan 2002. ISSN 0004-3702. 10.1016/S0004-3702(01)00129-1. URL https://doi.org/10.1016/S0004-3702(01)00129-1.

Guoxin Chen, Minpeng Liao, Chengxi Li, and Kai Fan. Alphamath almost zero: process supervision without process, 2024.

Lingjiao Chen, Jared Quincy Davis, Boris Hanin, Peter Bailis, Ion Stoica, Matei Zaharia, and James Zou. Are more llm calls all you need? towards scaling laws of compound inference systems, 2024. URL https://arxiv.org/abs/2403.02419.

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, 2021. URL https://arxiv.org/abs/2107.03374.

FranÃ§ois Chollet. On the measure of intelligence, 2019. URL https://arxiv.org/abs/1911.01547.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. Palm: Scaling language modeling with pathways, 2022.

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. URL https://openai.com/index/introducing-swe-bench-verified/. Blog post announcing a human-validated subset of SWE-bench for evaluating AI models' software engineering capabilities.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems, 2021.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems, 2021. URL https://arxiv.org/abs/2110.14168.

Ganqu Cui, Lifan Yuan, Zefan Wang, Hanbin Wang, Wendi Li, Bingxiang He, Yuchen Fan, Tianyu Yu, Qixin Xu, Weize Chen, Jiarui Yuan, Huayu Chen, Kaiyan Zhang, Xingtai Lv, Shuo Wang, Yuan Yao, Hao Peng, Yu Cheng, Zhiyuan Liu, Maosong Sun, Bowen Zhou, and Ning Ding. Process reinforcement through implicit rewards. https://curvy-check-498.notion.site/Process-Reinforcement-through-Implicit-Rewards-15f4fcb9c42180f1b498cc9b2eaf896f, 2025. Notion Blog.

Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning, 2023.

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher RÃ©. Flashattention: Fast and memory-efficient exact attention with io-awareness, 2022.

Jared Quincy Davis, Boris Hanin, Lingjiao Chen, Peter Bailis, Ion Stoica, and Matei Zaharia. Networks of networks: Complexity class principles applied to compound ai systems design, 2024. URL https://arxiv.org/abs/2407.16831.

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. URL https://arxiv.org/abs/2412.19437.

DeepSeek-AI et al. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model, 2024. URL https://arxiv.org/abs/2405.04434.

Mostafa Dehghani, Anurag Arnab, Lucas Beyer, Ashish Vaswani, and Yi Tay. The efficiency misnomer, 2022. URL https://arxiv.org/abs/2110.12894.

Daniel Y. Fu, Hermann Kumbong, Eric Nguyen, and Christopher RÃ©. Flashfftconv: Efficient convolutions for long sequences with tensor cores, 2023.

Leo Gao, Jonathan Tow, Baber Abbasi, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Alain Le Noac'h, Haonan Li, Kyle McDonell, Niklas Muennighoff, Chris Ociepa, Jason Phang, Laria Reynolds, Hailey Schoelkopf, Aviya Skowron, Lintang Sutawika, Eric Tang, Anish Thite, Ben Wang, Kevin Wang, and Andy Zou. A framework for few-shot language model evaluation, 12 2023. URL https://zenodo.org/records/10256836.

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

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. URL https://arxiv.org/abs/2407.21783.

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

Michael Hassid, Tal Remez, Jonas Gehring, Roy Schwartz, and Yossi Adi. The larger the better? improved llm code-generation via budget reallocation, 2024. URL https://arxiv.org/abs/2404.00725.

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.

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. URL https://arxiv.org/abs/1712.00409.

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.

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. URL https://arxiv.org/abs/2203.15556.

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration, 2020.

Arian Hosseini, Xingdi Yuan, Nikolay Malkin, Aaron Courville, Alessandro Sordoni, and Rishabh Agarwal. V-star: Training verifiers for self-taught reasoners, 2024.

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. URL https://arxiv.org/abs/2312.13010.

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. URL https://arxiv.org/abs/2310.01798.

HuggingFace. Hugging face accelerate. https://huggingface.co/docs/accelerate/index, 2022.

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.

Hyeonbin Hwang, Doyoung Kim, Seungone Kim, Seonghyeon Ye, and Minjoon Seo. Self-explore: Enhancing mathematical reasoning in language models with fine-grained rewards, 2024. URL https://arxiv.org/abs/2404.10346.

Robert Irvine, Douglas Boubert, Vyas Raina, Adian Liusie, Ziyi Zhu, Vineet Mudupalli, Aliaksei Korshuk, Zongyi Liu, Fritz Cremer, Valentin Assassi, Christie-Carol Beauchamp, Xiaoding Lu, Thomas Rialan, and William Beauchamp. Rewarding chatbots for real-world engagement with millions of users, 2023. URL https://arxiv.org/abs/2303.06135.

Dongfu Jiang, Xiang Ren, and Bill Yuchen Lin. Llm-blender: Ensembling large language models with pairwise ranking and generative fusion, 2023. URL https://arxiv.org/abs/2306.02561.

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. URL https://arxiv.org/abs/2310.06770.

Andy L. Jones. Scaling scaling laws with board games, 2021. URL https://arxiv.org/abs/2104.03113.

Jordan Juravsky, Bradley Brown, Ryan Ehrlich, Daniel Y Fu, Christopher Ré, and Azalia Mirhoseini. Hydragen: High-throughput llm inference with shared prefixes. *arXiv preprint arXiv:2402.05099*, 2024.

Jikun Kang, Xin Zhe Li, Xi Chen, Amirreza Kazemi, Qianyi Sun, Boxing Chen, Dong Li, Xu He, Quan He, Feng Wen, Jianye Hao, and Jun Yao. Mindstar: Enhancing math reasoning in pre-trained llms at inference time, 2024. URL https://arxiv.org/abs/2405.16265.

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.

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. URL https://arxiv.org/abs/2001.08361.

Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy Liang. Spoc: Search-based pseudocode to code, 2019. URL https://arxiv.org/abs/1906.04908.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 611–626, 2023.

Nathan Lambert, Valentina Pyatkin, Jacob Morrison, LJ Miranda, Bill Yuchen Lin, Khyathi Chandu, Nouha Dziri, Sachin Kumar, Tom Zick, Yejin Choi, Noah A. Smith, and Hannaneh Hajishirzi. Rewardbench: Evaluating reward models for language modeling, 2024. URL https://arxiv.org/abs/2403.13787.

Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, Yuhuai Wu, Behnam Neyshabur, Guy Gur-Ari, and Vedant Misra. Solving quantitative reasoning problems with language models, 2022. URL https://arxiv.org/abs/2206.14858.

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. URL https://arxiv.org/abs/2305.06161.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, RÃ©mi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson dâAutume, 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):1092â1097, December 2022. ISSN 1095-9203. 10.1126/science.abq1158. URL http://dx.doi.org/10.1126/science.abq1158.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, RÃ©mi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson dâAutume, 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):1092â1097, December 2022. ISSN 1095-9203. 10.1126/science.abq1158. URL http://dx.doi.org/10.1126/science.abq1158.

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.

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. URL https://arxiv.org/abs/2305.20050.

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. URL https://arxiv.org/abs/2409.02977.

Liangchen Luo, Yinxiao Liu, Rosanne Liu, Samrat Phatale, Meiqi Guo, Harsh Lara, Yunxuan Li, Lei Shu, Yun Zhu, Lei Meng, Jiao Sun, and Abhinav Rastogi. Improve mathematical reasoning in language models by automated process supervision, 2024. URL https://arxiv.org/abs/2406.06592.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback, 2023. URL https://arxiv.org/abs/2303.17651.

Dakota Mahan, Duy Van Phung, Rafael Rafailov, Chase Blagden, Nathan Lile, Louis Castricato, Jan-Philipp FrÃ¤nken, Chelsea Finn, and Alon Albalak. Generative reward models, 2024. URL https://arxiv.org/abs/2410.12832.

Aisha Malik. Openai's chatgpt now has 100 million weekly active users. https://techcrunch.com/2023/11/06/openais-chatgpt-now-has-100-million-weekly-active-users/, 2023. Accessed: 2023-11-06.

Shrestha Basu Mallick and Kathy Korevec. The next chapter of the gemini era for developers, 2024. URL https://developers.googleblog.com/en/the-next-chapter-of-the-gemini-era-for-developers/.

Alex Nguyen, Dheeraj Mekala, Chengyu Dong, and Jingbo Shang. When is the consistent prediction likely to be a correct prediction?, 2024. URL https://arxiv.org/abs/2407.05778.

Xuefei Ning, Zinan Lin, Zixuan Zhou, Zifu Wang, Huazhong Yang, and Yu Wang. Skeleton-of-thought: Large language models can do parallel decoding, 2023.

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

Isaac Ong, Amjad Almahairi, Vincent Wu, Wei-Lin Chiang, Tianhao Wu, Joseph E. Gonzalez, M Waleed Kadous, and Ion Stoica. Routellm: Learning to route llms with preference data, 2024. URL https://arxiv.org/abs/2406.18665.

OpenAI. Gpt-4 technical report, 2023.

OpenAI. Introducing openai o1, 2024. URL https://openai.com/o1/.

OpenAI et al. Gpt-4 technical report, 2024. URL https://arxiv.org/abs/2303.08774.

Anne Ouyang, Simon Guo, and Azalia Mirhoseini. Kernelbench: Can llms write gpu kernels?, 2024. URL https://scalingintelligence.stanford.edu/blogs/kernelbench/.

Anne Ouyang, Simon Guo, Simran Arora, Alex L Zhang, William Hu, Christopher Re, and Azalia Mirhoseini. Kernelbench: Can llms write efficient gpu kernels?, 2025.

Sandeep Kumar Pani. Sota on swebench-verified: (re)learning the bitter lesson, 2024. URL https://aide.dev/blog/sota-bitter-lesson.

Markus N. Rabe and Charles Staats. Self-attention does not need $o(n^2)$ memory, 2022.

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D. Manning, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model, 2024. URL https://arxiv.org/abs/2305.18290.

Baptiste RoziÃ¨re, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, JÃ©rÃ©my Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre DÃ©fossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas

Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code, 2023. URL https://arxiv.org/abs/2308.12950.

Haifeng Ruan, Yuntong Zhang, and Abhik Roychoudhury. Specrover: Code intent extraction via llms, 2024. URL https://arxiv.org/abs/2408.02232.

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

Nikhil Sardana, Jacob Portes, Sasha Doubov, and Jonathan Frankle. Beyond chinchilla-optimal: Accounting for inference in language model scaling laws, 2024. URL https://arxiv.org/abs/2401.00448.

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. URL https://www.anthropic.com/research/swe-bench-sonnet.

Rulin Shao, Jacqueline He, Akari Asai, Weijia Shi, Tim Dettmers, Sewon Min, Luke Zettlemoyer, and Pang Wei Koh. Scaling retrieval-based language models with a trillion-token datastore, 2024. URL https://arxiv.org/abs/2407.12854.

Noam Shazeer. Fast transformer decoding: One write-head is all you need, 2019.

Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Daniel Y. Fu, Zhiqiang Xie, Beidi Chen, Clark Barrett, Joseph E. Gonzalez, Percy Liang, Christopher RÃ©, Ion Stoica, and Ce Zhang. Flexgen: High-throughput generative inference of large language models with a single gpu, 2023.

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.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling llm test-time compute optimally can be more effective than scaling model parameters, 2024. URL https://arxiv.org/abs/2408.03314.

Yifan Song, Guoyin Wang, Sujian Li, and Bill Yuchen Lin. The good, the bad, and the greedy:

Evaluation of llms should not ignore non-determinism, 2024. URL https://arxiv.org/abs/2407.10457.

Benedikt Stroebl, Sayash Kapoor, and Arvind Narayanan. Inference scaling flaws: The limits of llm resampling with imperfect verifiers, 2024. URL https://arxiv.org/abs/2411.17501.

Hongjin Su, Ruoxi Sun, Jinsung Yoon, Pengcheng Yin, Tao Yu, and Sercan Ã. ArÄ±$k.Learn-by-interact : Adata-centricframeworkforself-adaptiveagentsinrealisticenvironments$, 2025.$URL$

Gemma Team et al. Gemma: Open models based on gemini research and technology, 2024. URL https://arxiv.org/abs/2403.08295.

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. URL https://arxiv.org/abs/2407.13168.

Ye Tian, Baolin Peng, Linfeng Song, Lifeng Jin, Dian Yu, Haitao Mi, and Dong Yu. Toward self-improvement of llms via imagination, searching, and criticizing, 2024. URL https://arxiv.org/abs/2404.12253.

Leo Tolstoy. *War and Peace*. 1869.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, TimothÃ©e Lacroix, Baptiste RoziÃ¨re, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models, 2023.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya

Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models, 2023.

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. ISSN 1476-4687. 10.1038/s41586-023-06747-5. URL https://doi.org/10.1038/s41586-023-06747-5.

Jonathan Uesato, Nate Kushman, Ramana Kumar, Francis Song, Noah Siegel, Lisa Wang, Antonia Creswell, Geoffrey Irving, and Irina Higgins. Solving math word problems with process- and outcome-based feedback, 2022. URL https://arxiv.org/abs/2211.14275.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023.

Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020. 10.1038/s41592-019-0686-2.

Fanqi Wan, Xinting Huang, Deng Cai, Xiaojun Quan, Wei Bi, and Shuming Shi. Knowledge fusion of large language models, 2024. URL https://arxiv.org/abs/2401.10491.

Haoxiang Wang, Wei Xiong, Tengyang Xie, Han Zhao, and Tong Zhang. Interpretable preferences via multi-objective reward modeling and mixture-of-experts, 2024. URL https://arxiv.org/abs/2406.12845.

Junlin Wang, Jue Wang, Ben Athiwaratkun, Ce Zhang, and James Zou. Mixture-of-agents enhances large language model capabilities, 2024. URL https://arxiv.org/abs/

2406.04692.

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. URL https://arxiv.org/abs/2312.08935.

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. URL https://arxiv.org/abs/2407.16741.

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.

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.

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. URL https://arxiv.org/abs/2408.00724.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying llm-based software engineering agents, 2024. URL https://arxiv.org/abs/2407.01489.

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. URL https://arxiv.org/abs/2405.15793.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models, 2022. URL https://arxiv.org/abs/2210.03629.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models, 2023.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models, 2023. URL https://arxiv.org/abs/2305.10601.

Weizhe Yuan, Richard Yuanzhe Pang, Kyunghyun Cho, Xian Li, Sainbayar Sukhbaatar, Jing Xu, and Jason Weston. Self-rewarding language models, 2024. URL https://arxiv.org/abs/2401.10020.

Eric Zelikman, Georges Harik, Yijia Shao, Varuna Jayasiri, Nick Haber, and Noah D. Goodman. Quiet-star: Language models can teach themselves to think before speaking, 2024. URL https://arxiv.org/abs/2403.09629.

Lunjun Zhang, Arian Hosseini, Hritik Bansal, Mehran Kazemi, Aviral Kumar, and Rishabh Agarwal. Generative verifiers: Reward modeling as next-token prediction, 2024. URL https://arxiv.org/abs/2408.15240.

Wenting Zhao, Nan Jiang, Celine Lee, Justin T Chiu, Claire Cardie, Matthias GallÃ©, and Alexander M Rush. Commit0: Library generation from scratch, 2024. URL https://arxiv.org/abs/2412.01769.

Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. Minif2f: a cross-system benchmark for formal olympiad-level mathematics. *arXiv preprint arXiv:2109.00110*, 2021.

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

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Jeff Huang, Chuyue Sun, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. Efficiently programming large language models using sglang, 2023.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. Sglang: Efficient execution of structured language model programs, 2024. URL https://arxiv.org/abs/2312.07104.

Albert Ãrwall. Moatless tools. https://github.com/aorwall/moatless-tools/

tree/a1017b78e3e69e7d205b1a3faa83a7d19fce3fa6, 2024.

A | Repeated Sampling Experimental Setup

A.1 Lean Formal Proofs

I report results on the 130 questions in the test set of the lean4 MiniF2F dataset that correspond to formalized MATH problems. This dataset is derived from the fixed version of the original MiniF2F dataset created by Zheng et al. [132]. I sample with a temperature of 0.5 and do not use nucleus sampling. I generate $10,000$ samples per problem. I use proofs of the following 5 theorems from the validation set as few-shot examples:

- `mathd_algebra_116`
- `amc12_2000_p5`
- `mathd_algebra_132`
- `mathd_algebra_11`
- `mathd_numbertheory_84`

The prompt consists of:

1. Few shot examples.
2. Header imports present in each problem in the HuggingFace dataset `cat-searcher/minif2f-lean` dataset, an upload of the lean4 MiniF2F dataset.
3. The theorem definition. In order to avoid leaking information about how to solve the theorem from its name, I replace the name of the theorem with `theorem_i`. $i \in \{1, 2, 3, 4, 5\}$ for the few-shot examples and $i = 6$ for the current problem.

I set 200 as the max token length for the generated solution. To grade solutions, I use the `lean-dojo 1.1.2` library with lean version `4.3.0-rc2`. I set a timeout of 10 seconds for every tactic step.

Few-Shot Example

Write a lean4 proof to the provided formal statement. You have access to the standard mathlib4 library.

~~~
```import Mathlib.Algebra.BigOperators.Basic
import Mathlib.Data.Real.Basic
import Mathlib.Data.Complex.Basic
import Mathlib.Data.Nat.Log
import Mathlib.Data.Complex.Exponential
import Mathlib.NumberTheory.Divisors
import Mathlib.Data.ZMod.Defs
import Mathlib.Data.ZMod.Basic
import Mathlib.Topology.Basic
import Mathlib.Data.Nat.Digits


open BigOperators
open Real
open Nat
open Topology
theorem theorem1
Int.floor ((9:ℝ) / 160 * 100) = 5 :=
by (
rw [Int.floor_eq_iff]
constructor
all_goals norm_num
)```
~~~

Example Prompt

Write a lean4 proof to the provided formal statement. You have access to the standard mathlib4 library.

```import Mathlib.Algebra.BigOperators.Basic
import Mathlib.Data.Real.Basic
import Mathlib.Data.Complex.Basic
import Mathlib.Data.Nat.Log
import Mathlib.Data.Complex.Exponential
import Mathlib.NumberTheory.Divisors
import Mathlib.Data.ZMod.Defs
import Mathlib.Data.ZMod.Basic
import Mathlib.Topology.Basic
import Mathlib.Data.Nat.Digits

open BigOperators
open Real
open Nat
open Topology
theorem theorem6
(b h v : )
($h_0$ : 0 < b ∧ 0 < h ∧ 0 < v)
($h_1$ : v = 1 / 3 * (b * h))
($h_2$ : b = 30)
($h_3$ : h = 13 / 2) :
v = 65 :=
by (

## A.2 CodeContests

I report results on the $140$ test set questions that do not include image tags in the problem description. I sample with a temperature of $0.6$ and a top-p value of $0.95$ following the experiments in
```

CodeLlama [92]. I generate 10,000 samples per problem. I use two few-shot examples from the training set that are randomly sampled per-problem. I set 1024 as the max token length for the generated solution. I use the same answer comparison function as [69] and use the concatenation of public, private, and generated tests to validate correctness of solutions.

Few-Shot Example

Q: Write python code to solve the following coding problem that obeys the constraints and passes the example test cases. The output code needs to read from and write to standard IO. Please wrap your code answer using ```:

Petya has equal...

...

(Rest of question and input/output examples)

...

A: ```

```
n = int(input())
a = int(input())
b = int(input())
ans = 6
cnt = 0
cur = 2
cnt += 2 * ((n - b) // a)
while cnt < 4:
    cur += 1
    cnt += (n // a)
ans = min(ans, cur)
if b * 2 <= n:
    cur, cnt = 0, 0
    cur = 1
    cnt += ((n - 2 * b) // a)
    while cnt < 4:
        cur += 1
        cnt += (n // a)
    ans = min(ans, cur)
print(ans)
```

```
```

Example Prompt

Q: Write python code to solve the following coding problem that obeys the constraints and passes the example test cases. The output code needs to read from and write to standard IO. Please wrap your code answer using ```:

This is the...

...

(Rest of question and input/output examples)

...

A:

A.3 MATH

I report results on 128 randomly selected test-set problems. I sample with a temperature of 0.6 and do not use nucleus sampling. I use the fixed 5 few-shot example from [67] for each problem. I generate $10,000$ samples per problem. I set 512 as the max token length for the generated solution. To grade solutions, I use the `minerva_math` functions from LMEval [40] to extract the model's final answer. I then check correctness if the extracted answer is an exact string match to the ground truth, or if the `is_equiv` function from `minerva_math` in LMEval evaluates to true.

Few-Shot Example

Problem:

If $\det \mathbf{A} = 2$ and $\det \mathbf{B} = 12$, then find $\det(\mathbf{A}\mathbf{B})$.

Solution:

We have that $\det(\mathbf{A}\mathbf{B}) = (\det \mathbf{A})(\det \mathbf{B}) = (2)(12) = \boxed{24}$. Final Answer: The final answer is 24. I hope it is correct.

Example Prompt

Problem:

What is the domain of the function

$$f(x)=\frac{(2x-3)(2x+5)}{(3x-9)(3x+6)}?$$

Express your answer as an interval or as a union of intervals.

Solution:

A.4 GSM8K

I report results on 128 randomly sampled test-set problems. I sample with a temperature of 0.6 and do not use nucleus sampling. I use 5 few-shot examples from the training set that are randomly sampled per-problem. I generate $10,000$ samples per problem. I set 512 as the max token length for the generated solution. To grade solutions, I follow LMEval [40] and extract answers using a regular expression that extracts the string after the quadruple hashes. Similar to MATH, I then assess correctness by checking if the extracted answer is an exact string match to the ground truth or if `is_equiv` evaluates to true.

Few-Shot Example

Question: James decides to replace his car. He sold his \$20,000 car for 80% of its value and then was able to haggle to buy a \$30,000 sticker price car for 90% of its value. How much was he out of pocket?

Answer: He sold his car for 20000*.8=\$<<20000*.8=16000>>16,000 He bought the new car for 30,000*.9=\$<<30000*.9=27000>>27,000 That means he was out of pocket 27,000-16,000=\$<<27000-16000=11000>>11,000

11000

Example Prompt

Question: Mary has 6 jars of sprinkles in her pantry. Each jar of sprinkles can decorate 8 cupcakes. Mary wants to bake enough cupcakes to use up all of her sprinkles. If each pan holds 12 cupcakes, how many pans worth of cupcakes should she bake?

Answer:

B | SWE-bench Lite

B.1 Experimental Setup

For the SWE-bench Lite experiments in Chapter 3, I use DeepSeek-Coder-V2-Instruct with the Moatless Tools agent framework (at commit *a1017b78e3e69e7d205b1a3faa83a7d19fce3fa6*). We use Voyage AI [3] embeddings for retrieval, the default used by Moatless Tools. We make no modifications to the model or framework, using them entirely as off-the-shelf components.

With this setup, I sample 250 independent completions for each problem using standard temperature-based sampling. To determine the optimal sampling temperature, I conducted a sweep on a random subset of 50 problems from the test set, testing temperatures of 1.0, 1.4, 1.6, and 1.8. Based on these results, I selected a temperature of 1.6 for the main experiments.

B.2 Test Suite Flakiness

During the analysis, I identified 34 problems in SWE-bench Lite whose test suites had flaky tests. Using the SWE-bench testing harness provided by the authors of SWE-bench, I tested each solution repeatedly: for some solutions, sometimes the solution was marked as correct, and other times it was marked as incorrect. In 30 of these 34 cases, I observed flakiness even on the correct solutions provided by the dataset authors. Table B.1 lists the problem IDs of the 34 instances with flaky tests.

Table B.1: Instance IDs of problems from SWE-bench Lite that have flaky tests.

Repository	**Instance IDs**
`django`	`django__django-13315, django__django-13447, django__django-13590, django__django-13710, django__django-13757, django__django-13933, django__django-13964, django__django-14017, django__django-14238, django__django-14382, django__django-14608, django__django-14672, django__django-14752, django__django-14915, django__django-14997, django__django-14999, django__django-15320, django__django-15738, django__django-15790, django__django-15814, django__django-15819, django__django-16229, django__django-16379, django__django-16400, django__django-17051`
`sympy`	`sympy__sympy-13146, sympy__sympy-13177, sympy__sympy-16988`
`requests`	`psf__requests-863, psf__requests-2317, psf__requests-2674, psf__requests-3362`
`scikit-learn`	`scikit-learn__scikit-learn-13241`
`matplotlib`	`matplotlib__matplotlib-23987`

An additional instance, `astropy__astropy-6938`, was flaky on some machines and not

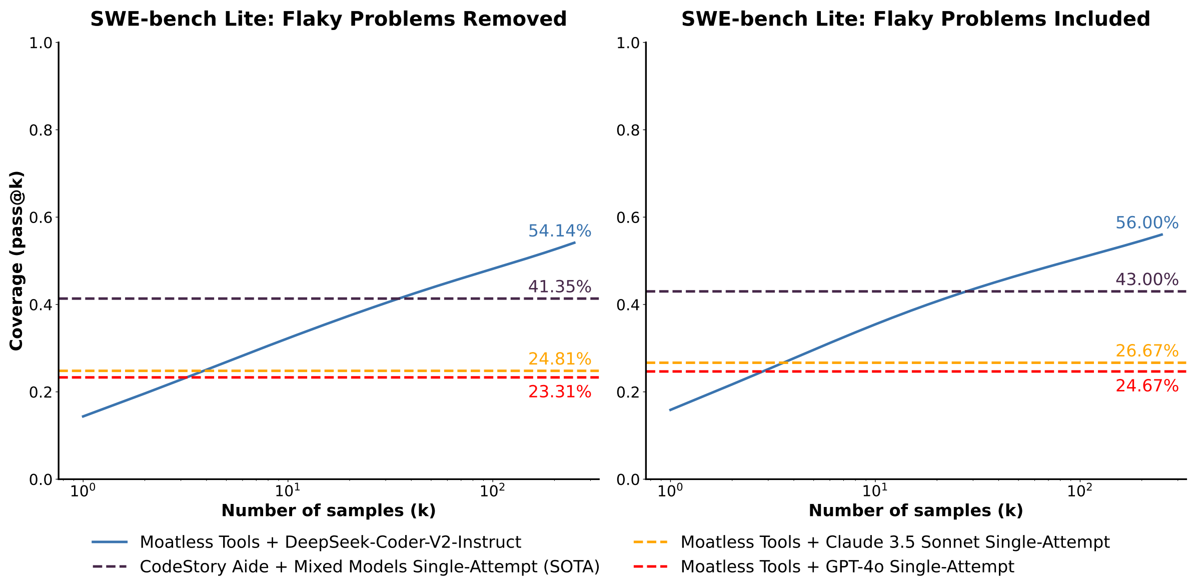


Figure B.1: SWE-bench Lite results, without and with problems that have flaky tests. For the graph on the left, all problems in Table B.1 are excluded. For the graph on the right, all problems are included. Note that the trend is the same with or without the flaky tests.

others. The authors of SWE-bench were able to reproduce the flakiness; however, I was unable to. My preliminary investigation indicates this specific issue is due to unpinned versions of dependencies in the docker environments that run the unit tests.

Here, I include results on a subset with the problems in Table B.1 removed (266 problems). For the full dataset evaluation, on any problem that has flaky tests, I run the test suite 11 times and use majority voting to determine whether a solution passed or failed. For the evaluation on the subset without flaky tests, all baselines I compare against release which problems they correctly solve, so I simply removed the problems with flaky tests and recomputed their scores.

C | Scaling Law Details

C.1 Experimental Details

To fit exponentiated power laws and mixture of geometrics to coverage curves, I first sample 40 points spaced evenly along a log scale from 1 to 10k (or 1 to 250 for SWE-bench Lite) and remove duplicates. I then use SciPy's [114] `curve_fit` function to find the parameters that best fit these points. I use the `lm` method for exponentiated power laws. I attempted to use the same method for the mixture of geometrics but found that some of the fit p estimates would be negative. To correct this, I use the `trt` method and ensure that the bounds of the fit parameters are between 0 and 1.

C.2 Additional Scaling Law Results

In Figure C.1, I show the results of fitting mixture of geometric curves to the empirical coverage curves for a different number of components. Note that as I increase the number of components, the mixture of geometrics becomes an increasingly accurate estimate of the coverage curve. I show results for a mixture of geometrics with 5 components for an additional set of models and tasks in Figure C.2.

I show the same results in Figure C.3 and Figure C.4, but only fit the first 100 pass@k values calculated on a $1k$ sized sample subset. I note that the mixture of geometric fit to only the first 100 samples does not accurately predict the coverage gains for a larger number of samples.

In Figure C.5, I show additional results fitting exponentiated power laws to coverage curves for an expanded set of datasets and models. I expand on the results in Figure 3.5 and show extra prediction results when fitting exponentiated power laws to the coverage curves for more models and datasets in Figure C.6.

C.3 Similarities in Coverage Curves Across Models

Interestingly, when comparing the coverage curves (with a log-scaled x-axis) of different models from the same family on the same task (see Figure 3.2), it appears that the traced S-curves have the same slope, but unique horizontal offsets. To investigate this further, I overlay the coverage

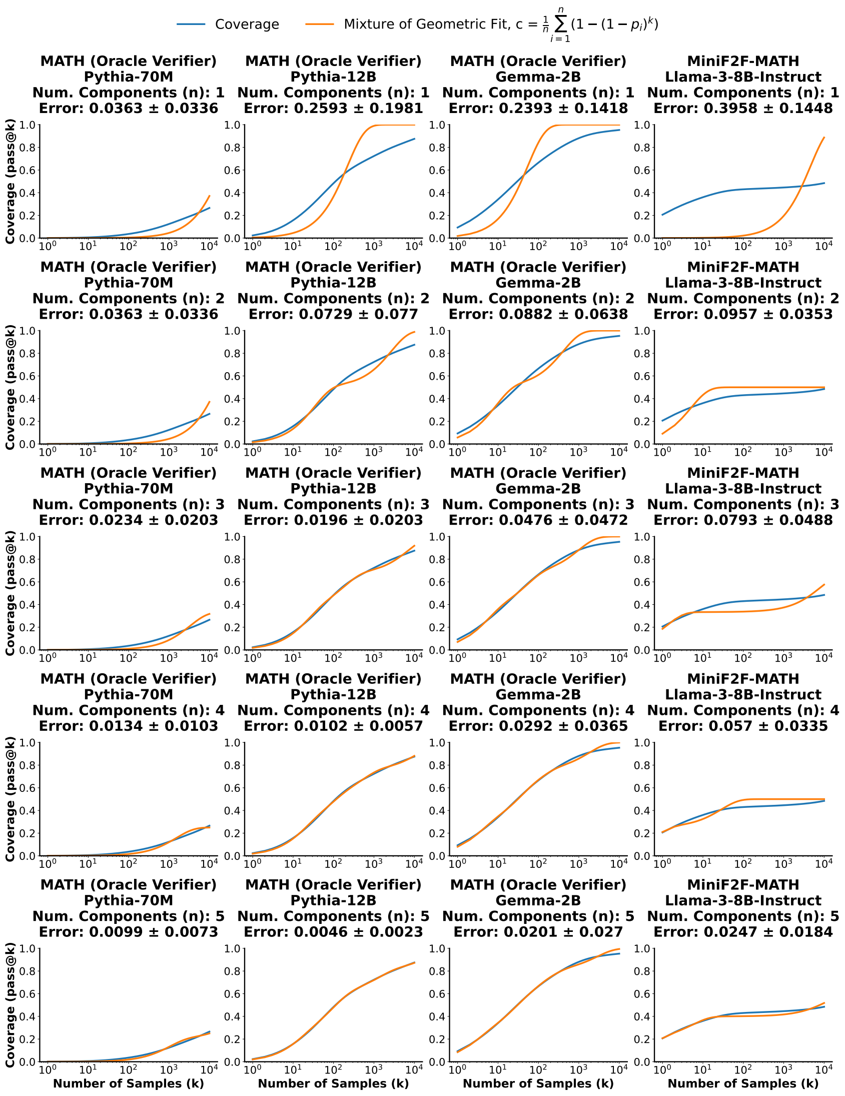


Figure C.1: Fitting mixture of geometrics to coverage curves for various numbers of components. I show the mean and standard deviation of the error between the coverage curve and the mixture of geometric fit across 100 evenly sampled points on the log scale.

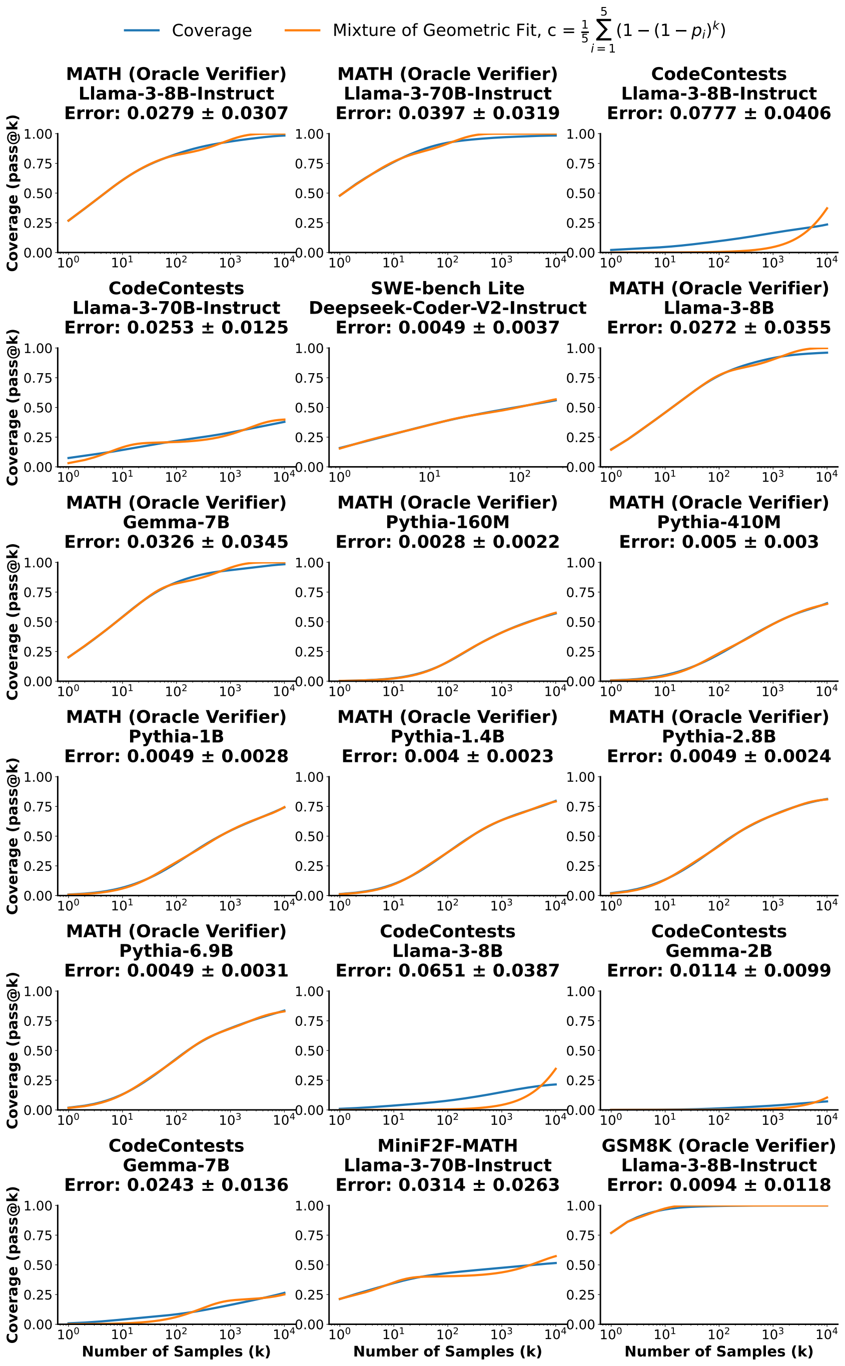


Figure C.2: Fitting mixture of geometrics to coverage curves for an expanded set of tasks and models. I show the mean and standard deviation of the error between the coverage curve and the mixture of geometric fit across 100 evenly sampled points on the log scale.

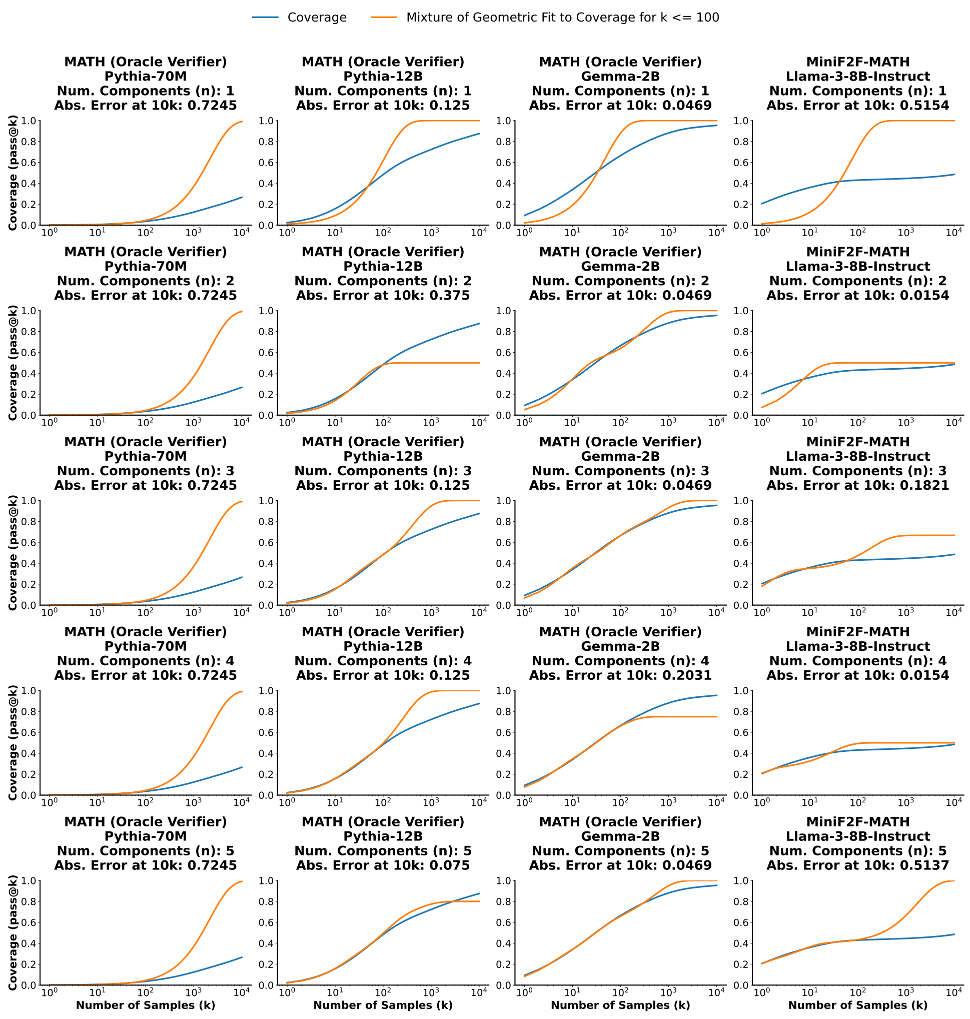


Figure C.3: Predicting coverage values for high k values by fitting a mixture of geometrics to the coverage curve for $k \leq 100$. Note that I recalculate the pass@k values used to do the power law fitting using a random $1k$ subset of the datapoints. I report the absolute prediction error at $10k$ samples.

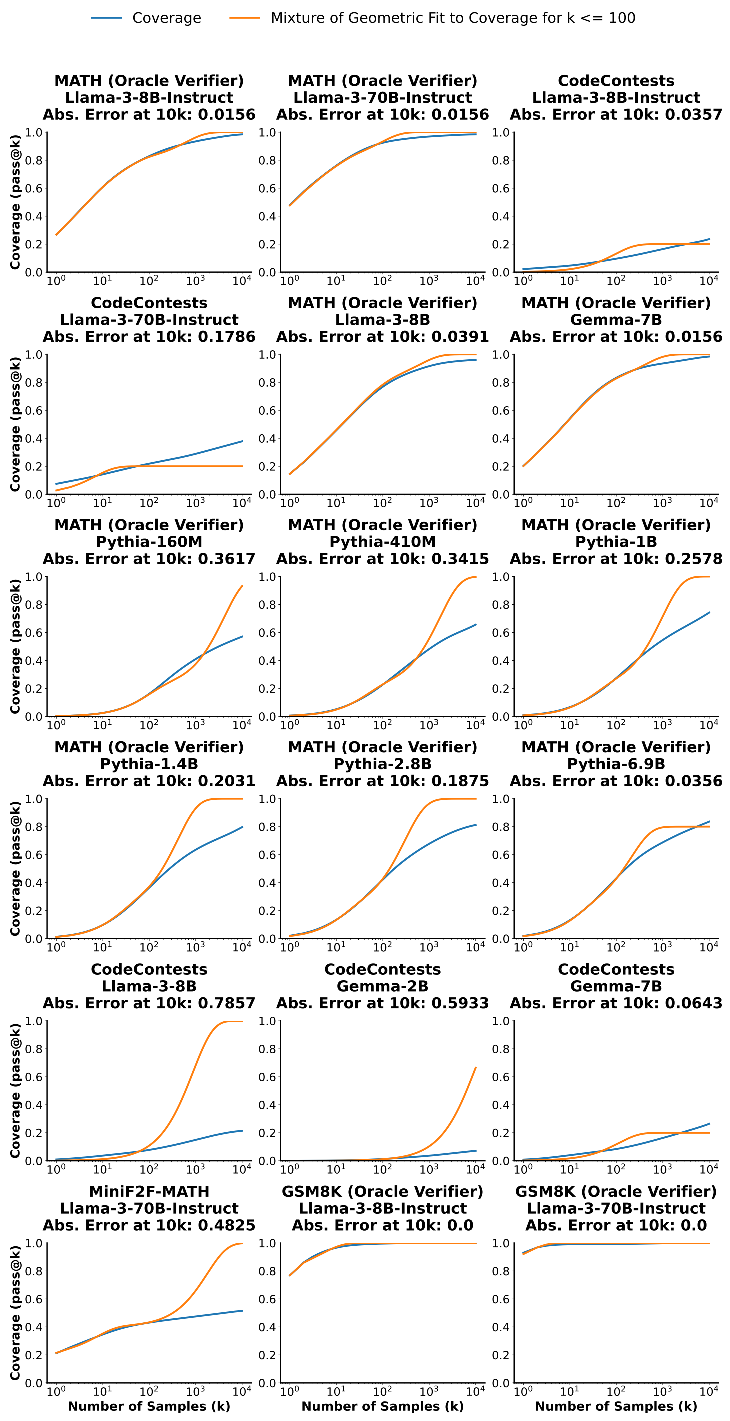


Figure C.4: Predicting coverage values for high k values by fitting a mixture of geometrics to the coverage curve for $k \leq 100$. Note that I recalculate the pass@k values used to do the power law fitting using a random $1k$ subset of the datapoints. I report the absolute prediction error at $10k$ samples.

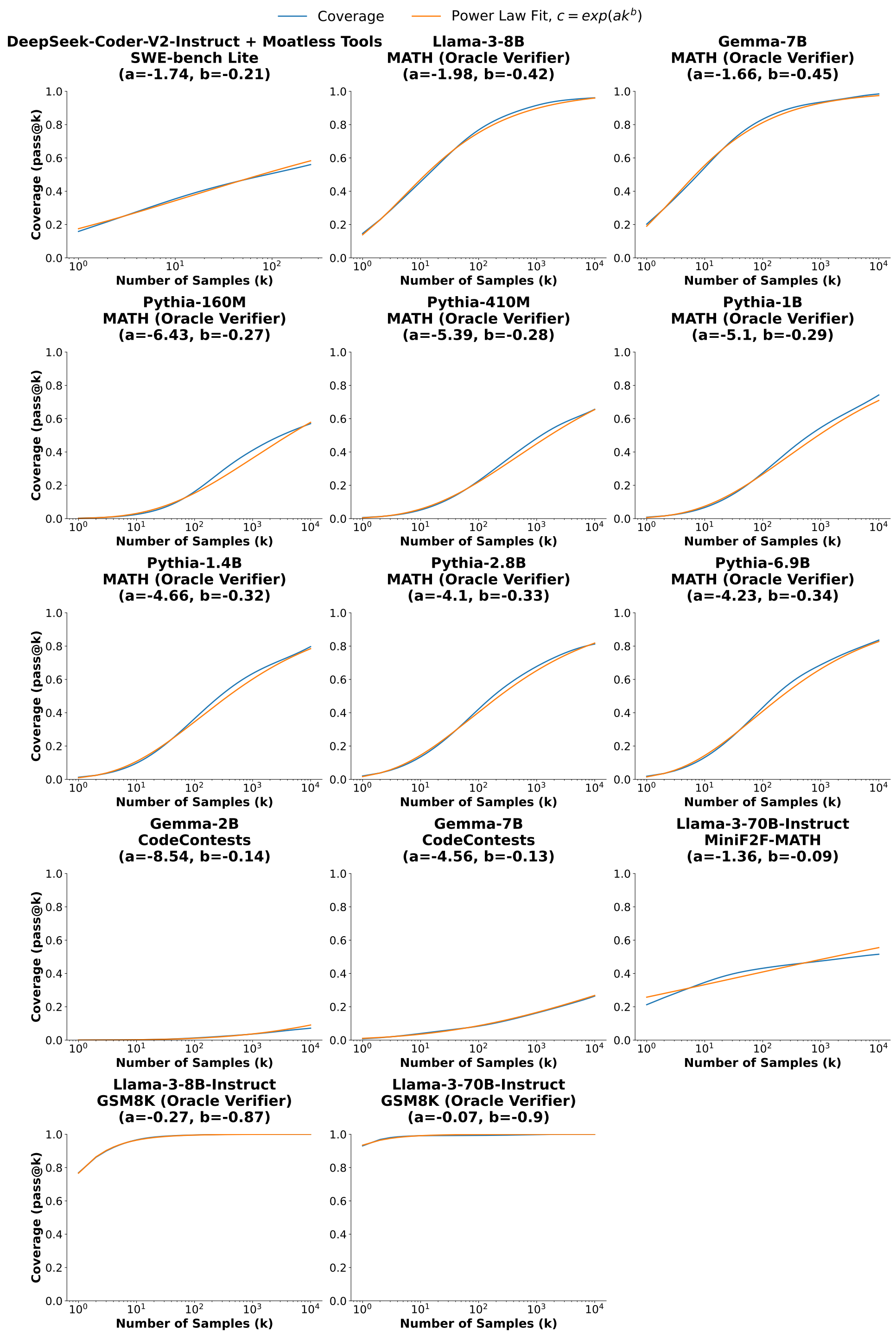


Figure C.5: Fitting exponentiated power laws to coverage curves for an expanded set of tasks and models. I show the mean and standard deviation of the error between the coverage curve and the exponentiated power law fit across 100 evenly sampled points on the log scale.

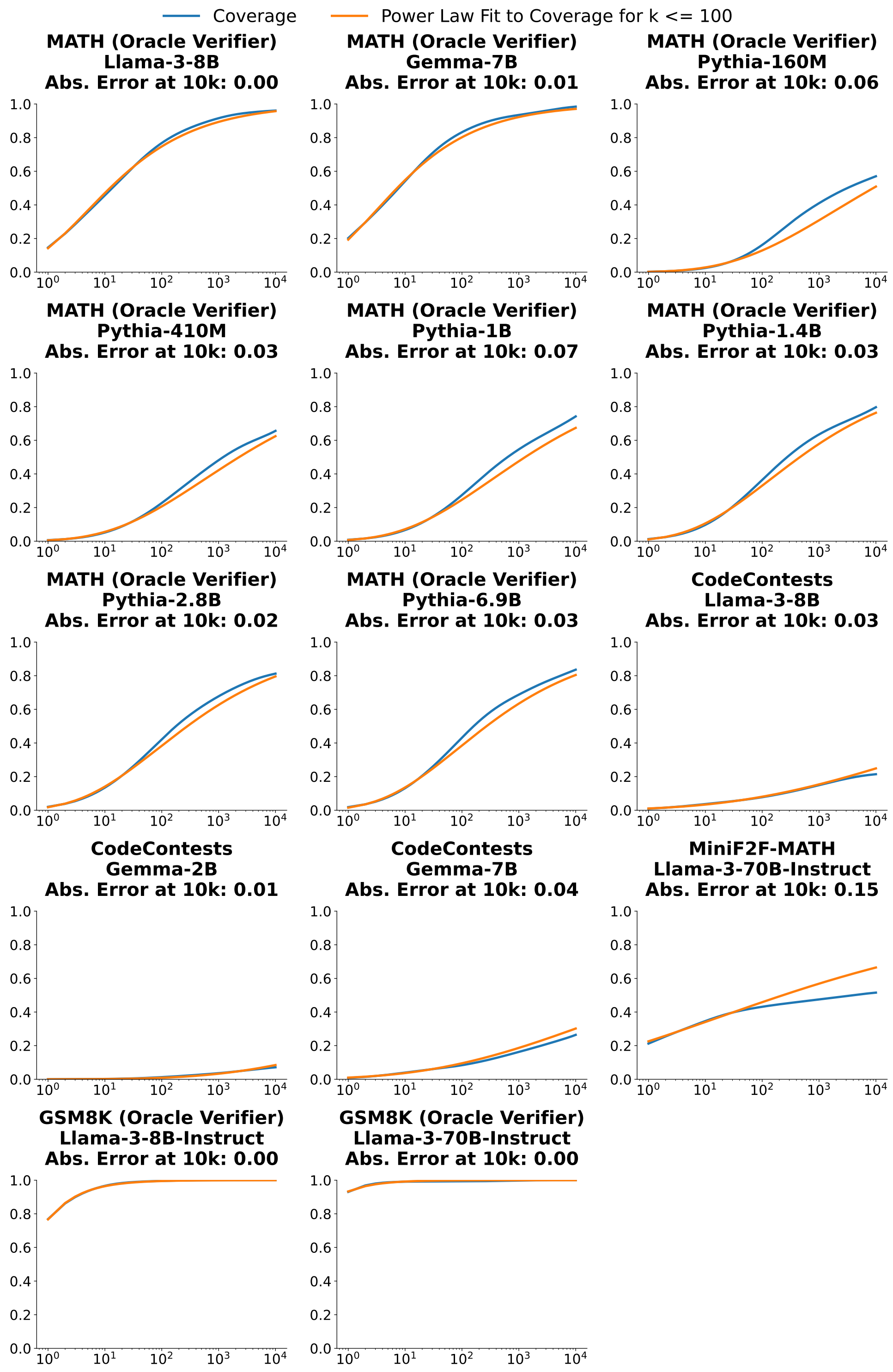


Figure C.6: Predicting coverage values for high k values by fitting a power law to the coverage curve for $k \leq 100$. Note that I recalculate the pass@k values used to do the power law fitting using a random $1k$ subset of the datapoints. I report the absolute prediction error at $10k$ samples.

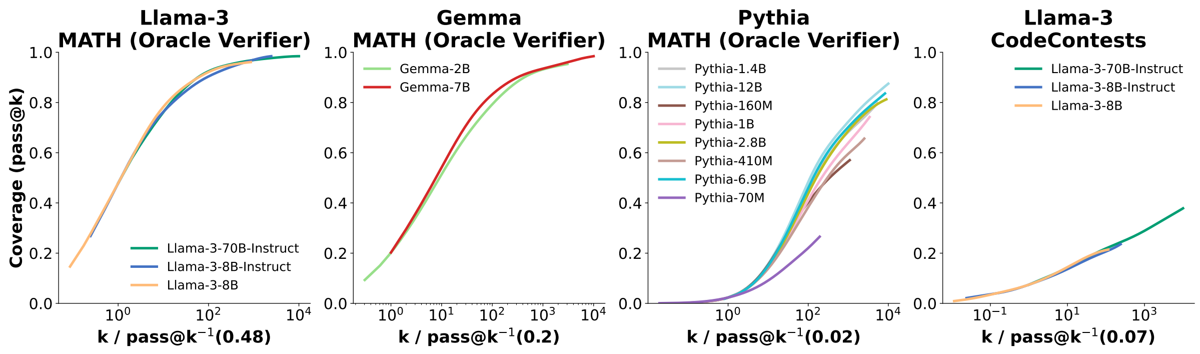


Figure C.7: Overlaying the coverage curves from different models belonging to the same family. I perform this overlay by horizontally shifting every curve (with a logarithmic x-axis) so that all curves pass through the point (1, c). I pick c to be the maximum pass@1 score over all models in the plot. I note that the similarity of the curves post-shifting shows that, within a model family, sampling scaling curves follow a similar shape.

curves of different models from the same family in Figure C.7. I do this by picking an anchor coverage value c, and shifting every curve leftward (in log-space) so that each passes through the point (1, c). This corresponds to a leftward shift by $\log(\text{pass@k}^{-1}(c))$, where $\text{pass@k}^{-1}(c)$ denotes the closest natural number k such that $\text{pass@k} = c$. I pick c to be the maximum pass@1 score over all models from the same family. These similarities demonstrate that across models from the same family, the increase in the log-sample-budget (or equivalently, the multiplicative increase in the sample budget) needed to improve coverage from c to c' is approximately constant. This can also be seen in the fit exponentiated power laws (Figure 3.4), where models from the same family on the same dataset have very similar b values.

D | GSM8K incorrect answer

As discussed in 3.5.2, I identify that a problem in the GSM8K test set (index 1042 on HuggingFace) has an incorrect ground truth solution.

Question

Johnny's dad brought him to watch some horse racing and his dad bet money. On the first race, he lost $\$5$. On the second race, he won $\$1$ more than twice the amount he previously lost. On the third race, he lost 1.5 times as much as he won in the second race. How much did he lose on average that day?

Answer

On the second race he won $\$11$ because $1 + 5 \times 2 =<< 1 + 5 * 2 = 11 >> 11$

On the third race he lost $\$15$ because $10 \times 1.5 =<< 10 * 1.5 = 15 >> 15$

He lost a total of $\$20$ on the first and third races because $15+5 =<< 15+5 = 20 >> 20$

He lost $\$9$ that day because $11 - 20 =<< 11 - 20 = -9 >> -9$

He lost an average of $\$3$ per race because $9/3 =<< 9/3 = 3 >> 3$

3

The mistake is in the second line of the answer: on the third race, Johnny's dad lost $\$16.5$, not $\$15$, meaning he made $\$11$ and lost $\$16.5 + \$5 = \$21.5$. So, the answer is an average loss of $\$3.5$ per race, not $\$3$ per race (the answer in the dataset).

E | Additional Repeated Sampling Ablations

In Figure E.1, I ablate the effect of the temperature and top-p values used for repeated sampling. In both sweeps, I sample 1k samples for the same 128 subset of the MATH dataset using Llama3-8B-Instruct. For the top-p sweep, I set temperature to 0.6 and sweep over top-p values in $\{0.5, 0.75, 0.8, 0.9, 0.95, 1.0\}$. The results are not very sensitive to p, with only top-p=0.5 being noticeably worse than the rest. For the temperature sweep, I set top-p to 1.0 and sweep over temperature values in $\{0.4, 0.6, 0.8, 1.0, 1.2, 1.4\}$. Temperatures 1.2 and 1.4 have a significantly lower coverage than the rest.

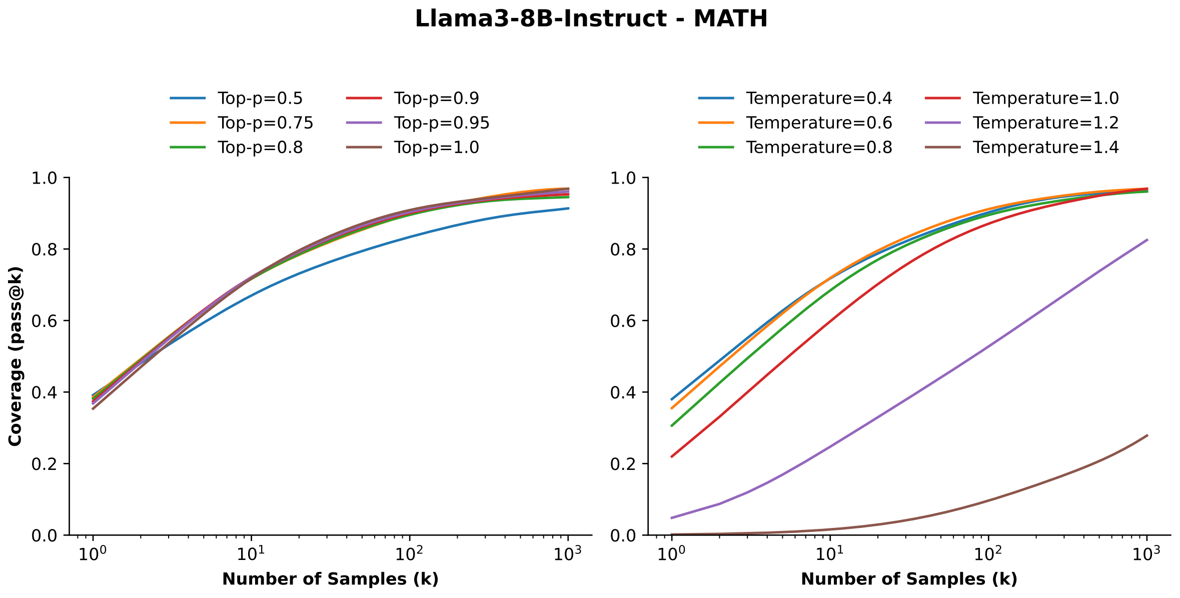


Figure E.1: Ablating the effect of top-p and temperature in repeated sampling. For the sweep over top-p values, I fix temperature to 0.6 and for the temperature sweep we fix top-p to 1.

F | FLOP Approximation for Llama Models

As Llama-3 models are dense transformers where the majority of parameters are used in matrix multiplications, I approximate inference FLOPs with the formula [95]:

$$\text{FLOPs-Per-Token}(\text{Context-Len}) \approx 2 * (\text{Num-Parameters} + 2 * \text{Num-Layers} * \text{Token-Dim} * \text{Context-Len})$$

$$\begin{aligned}\text{Total-Inference-FLOPs} \approx & \sum_{t=1}^{\text{Num-Prompt-Tokens}} \text{FLOPs-Per-Token}(t) \\ & + \sum_{t=1}^{\text{Num-Decode-Tokens}} \text{FLOPs-Per-Token}(t + \text{Num-Prompt-Tokens}) * \text{Num-Completions}\end{aligned}$$

G | CodeMonkeys with DeepSeek-V3 Results

Here, I conduct an initial evaluation of DeepSeek-V3 [36] as a potential substitute for CodeMonkeys's use of Claude 3.5 Sonnet. I reuse the same codebase context files as the primary CodeMonkeys experiments and rerun the testing and editing state machines on a 100-instance random subset of SWE-bench Verified using DeepSeek-v3. In the Figure G.1, I compare how coverage and majority voting score scale with the number of samples and iterations when compared to Claude Sonnet 3.5. I 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.

G.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 G.1, I provide the inputs, initial task, information given at each iteration, and the task at each iteration for all three state machines.

G.2 CodeMonkeys Hyperparameters

I elaborate on other experimental details for each part in Table G.2.

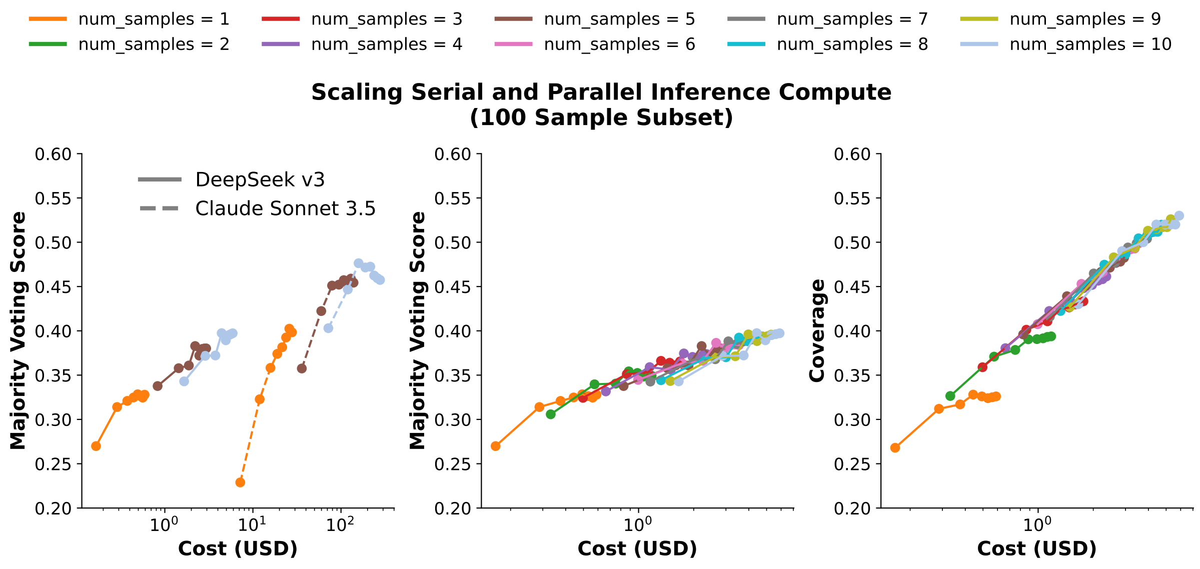


Figure G.1: **Left:** Comparing the impact of scaling the number of parallel samples and sequential iterations on majority voting score between Claude and DeepSeek-V3. Each line corresponds to a fixed amount of samples, with each dot on the line being a different maximum number of sequential iterations. 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. Note that coverage is continuing to scale with increased inference compute.

	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 G.1: Details of the Testing, Editing, and Selection State Machines.

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 G.2: CodeMonkeys hyperparameter summary.

H | Local Compute for Relevance

For the relevance stage, I run Qwen-2.5-Coder-32B-Instruct [54] locally in bf16 across 8xL40S GPUs. Each L40S has a bf16 compute throughput of 362.05 TFLOPS [81], giving the node a theoretical throughput of:

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

Since the model I run is a dense transformer with no parameter sharing, I can estimate its FLOPs per token as 2 * num_parameters [95]. Assuming a hardware utilization of 20% during inference, I 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, I use RunPod's pricing: an 8xL40S node costs \$8.24 per hour [94], yielding a total cost of:

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

I | Proving the Correctness of Attention Decomposition

I start by explicitly expressing softmax as an exponentiation followed by a normalization:

$$\frac{QK^T}{\sqrt{d}} = \frac{\exp\left(\frac{QK^T}{\sqrt{d}}\right)}{e^{Q,K}} \tag{I.1}$$

Therefore I can rewrite Equation 5.1 as:

$$\text{SDP}\left(Q, K, V\right) = \left(\frac{\exp\left(\frac{QK^T}{\sqrt{d}}\right)}{e^{Q,K}}\right) V \tag{I.2}$$

I can then expand Equation 5.5:

$$\begin{align}
&\frac{\text{SDP}\left(Q, K_1, V_1\right) e^{Q,K_1} + \text{SDP}\left(Q, K_2, V_2\right) e^{Q,K_2}}{e^{Q,K_1} + e^{Q,K_2}} \tag{I.3}\\
&= \frac{\left(\frac{\exp\left(\frac{QK_1^T}{\sqrt{d}}\right)}{e^{Q,K_1}}\right) V_1 e^{Q,K_1} + \left(\frac{\exp\left(\frac{QK_2^T}{\sqrt{d}}\right)}{e^{Q,K_2}}\right) V_2 e^{Q,K_2}}{e^{Q,K_1} + e^{Q,K_2}} \tag{I.4}\\
&= \frac{\exp\left(\frac{QK_1^T}{\sqrt{d}}\right) V_1 + \exp\left(\frac{QK_2^T}{\sqrt{d}}\right) V_2}{e^{Q,K_1} + e^{Q,K_2}} \tag{I.5}\\
&= \frac{\exp\left(\frac{Q(K_1||K_2)^T}{\sqrt{d}}\right) (V_1||V_2)}{e^{Q,K_1||K_2}} \tag{I.6}\\
&= \text{SDP}\left(Q, K_1||K_2, V_1||V_2\right) \tag{I.7}
\end{align}$$

as required. □

J | Hydragen Pseudocode

I provide PyTorch-style pseudocode implementing Hydragen attention below. I highlight that Hydragen can be implemented easily and efficiently in existing machine learning libraries, as long as there is a fast attention primitive that returns the LSE needed for softmax recombination.

```
import torch
from torch import Tensor

def attention(q: Tensor, k: Tensor, v: Tensor) -> tuple[Tensor, Tensor]:
    """
    Placeholder for some fast attention primitive
    that also returns LSEs. We use the flash-attn
    package in our implementation.

    q shape: [batch, qseq_len, qheads, dim]
    k shape: [batch, kvseq_len, kvheads, dim]
    v shape: [batch, kvseq_len, kvheads, dim]
    """
    pass

def combine_lse(
    out1: Tensor,
    lse1: Tensor,
    out2: Tensor,
    lse2: Tensor,
):
    """
    Combines two attention results using their LSEs.

    Out1/2 shape: [batch, seq_len, qheads, hdim]
    lse1/2 shape: [batch, seq_len, qheads]
    """
    max_lse = torch.maximum(lse1, lse2)

    adjust_factor1 = (lse1 - max_lse).exp()
    adjust_factor2 = (lse2 - max_lse).exp()

    new_denominator = adjust_factor1 + adjust_factor2

    aggregated = (
        out1 * adjust_factor1.unsqueeze(-1) +
        out2 * adjust_factor2.unsqueeze(-1)
    ) / new_denominator.unsqueeze(-1)

    return aggregated


def hydragen_attention(
    q: Tensor,
    prefix_k: Tensor,
    prefix_v: Tensor,
    suffix_k: Tensor,
    suffix_v: Tensor,
):
    """
    q: shape [batch, num_queries (1 during decoding), qheads, dim]

    prefix_k: shape [prefix_len, kvheads, dim]
    prefix_v: shape [prefix_len, kvheads, dim]

    suffix_k: shape [batch, suffix_len, kvheads, dim]
    suffix_v: shape [batch, suffix_len, kvheads, dim]
    """
```

```
b, nq, hq, d = q.shape

# inter-sequence batching: merge attention queries
# as if they all came from the same sequence.
batched_q = q.view(1, b * nq, hq, d)


# efficient attention over prefixes
# prefix_out: shape [1, batch * nq, hq, dim]
# prefix_lse: shape [1, batch * nq, hq]
prefix_out, prefix_lse = attention(
    batched_q,
    prefix_k.unsqueeze(0),
    prefix_v.unsqueeze(0),
)


# normal attention over suffixes
# suffix_out: shape [batch, suffix_len, hq, dim]
# suffix_lse: shape [batch, suffix_len, hq]
suffix_out, suffix_lse = attention(
    batched_q,
    suffix_k,
    suffix_v,
)

# unmerge prefix attention results and combine
# softmax denominators
aggregated = combine_lse(
    prefix_out.view(b, nq, hq, d),
    prefix_lse.view(b, nq, hq),
    suffix_out,
        suffix_lse,
    )

    return aggregated
```

K | Additional Hydragen Results

K.1 End-to-End Throughput

I expand on the end-to-end throughput experiments discussed in Section 5.3.1. I report additional results with more model sizes when generating 128 and 256 tokens. These results are displayed in Table K.1 and Table K.2 for CodeLlama-7b, Table K.3 and Table K.4 for CodeLlama-13b, and Table K.5 and Table K.6 for CodeLlama-34b, respectively. Note that in the tables where 128 tokens are generated per sequence, the "16K" column corresponds to a prefix length of 16256 tokens, while for the tables with 256 generated tokens per sequence, this corresponds to 16128 tokens (this is done to accommodate the 16384 max sequence length of the CodeLlama models).

	FlashAttention					Hydragen					vLLM (No Tokenization)					vLLM					Upper Bound (No Attention)
Batch	Prefix length					Prefix length					Prefix length					Prefix length					Prefix length
Size	1K	2K	4K	8K	16K	1K	2K	4K	8K	16K	1K	2K	4K	8K	16K	1K	2K	4K	8K	16K	All
32	2.5 ± 0.0	2.2 ± 0.0	1.8 ± 0.0	1.3 ± 0.0	0.9 ± 0.0	2.7 ± 0.0	2.7 ± 0.0	2.6 ± 0.0	2.6 ± 0.0	2.5 ± 0.0	1.7 ± 0.0	1.8 ± 0.0	1.7 ± 0.1	0.6 ± 0.0	0.4 ± 0.0	1.6 ± 0.0	1.6 ± 0.0	1.5 ± 0.0	0.6 ± 0.0	0.3 ± 0.0	3.1 ± 0.0
64	4.2 ± 0.0	3.4 ± 0.0	2.6 ± 0.0	1.7 ± 0.0	X	5.0 ± 0.0	4.9 ± 0.0	4.9 ± 0.1	4.8 ± 0.0	4.6 ± 0.0	3.5 ± 0.1	3.5 ± 0.1	2.9 ± 0.1	0.7 ± 0.0	0.4 ± 0.0	2.9 ± 0.0	2.8 ± 0.1	2.1 ± 0.2	0.7 ± 0.0	0.4 ± 0.0	5.7 ± 0.0
128	5.7 ± 0.0	4.2 ± 0.0	2.7 ± 0.0	X	X	8.6 ± 0.0	8.5 ± 0.0	8.4 ± 0.0	8.3 ± 0.0	8.0 ± 0.0	6.1	5.5	3.2	0.8	0.4	4.9	4.5	2.7	0.7	0.4	10.3 ± 0.0
256	8.1 ± 0.0	5.7 ± 0.0	X	X	X	13.3 ± 0.0	13.3 ± 0.0	13.1 ± 0.0	12.8 ± 0.0	12.3 ± 0.0	8.9	5.6	3.1	0.8	0.4	6.9	4.2	2.5	0.8	0.4	15.8 ± 0.0
512	X	X	X	X	X	19.6 ± 0.0	19.4 ± 0.0	19.1 ± 0.0	18.5 ± 0.0	17.5 ± 0.0	4.7	2.8	1.5	0.8	0.4	4.2	2.5	1.4	0.8	0.4	23.2 ± 0.0
1024	X	X	X	X	X	25.3 ± 0.0	25.1 ± 0.0	24.7 ± 0.0	23.9 ± 0.0	22.4 ± 0.0	4.9	2.8	1.5	0.8	0.4	4.2	2.5	1.4	0.7	0.4	30.1 ± 0.0
2048	X	X	X	X	X	27.9 ± 0.0	27.5 ± 0.0	26.7 ± 0.0	25.3 ± 0.0	22.8 ± 0.0	4.9	2.8	1.5	0.8	0.4	4.2	2.5	1.4	0.7	0.4	32.9 ± 0.0

Table K.1: End-to-end decoding throughput (thousands of tokens per second) with CodeLlama-7B on 8xA100 40 GB GPUs when generating 128 tokens. An x indicates the model does not have the required memory to run.

	FlashAttention					Hydragen					vLLM (No Tokenization)					vLLM					Upper Bound (No Attention)
Batch	Prefix length					Prefix length					Prefix length					Prefix length					Prefix length
Size	1K	2K	4K	8K	16K	1K	2K	4K	8K	16K	1K	2K	4K	8K	16K	1K	2K	4K	8K	16K	All
32	2.4 ± 0.0	2.2 ± 0.0	1.8 ± 0.0	1.3 ± 0.0	0.9 ± 0.0	2.6 ± 0.0	2.6 ± 0.0	2.6 ± 0.0	2.5 ± 0.0	2.4 ± 0.0	1.7 ± 0.0	1.8 ± 0.0	1.7 ± 0.0	0.6 ± 0.0	0.4 ± 0.0	1.6 ± 0.0	1.5 ± 0.0	1.5 ± 0.0	0.6 ± 0.0	0.3 ± 0.0	3.1 ± 0.0
64	3.9 ± 0.0	3.4 ± 0.0	2.5 ± 0.0	1.7 ± 0.0	X	4.8 ± 0.0	4.8 ± 0.0	4.8 ± 0.0	4.7 ± 0.0	4.5 ± 0.0	3.4 ± 0.0	3.3 ± 0.0	2.7 ± 0.0	0.7 ± 0.0	0.4 ± 0.0	2.8 ± 0.1	2.8 ± 0.0	2.3 ± 0.0	0.6 ± 0.0	0.4 ± 0.0	5.7 ± 0.0
128	5.3 ± 0.0	4.1 ± 0.0	2.7 ± 0.0	X	X	8.2 ± 0.0	8.2 ± 0.0	8.1 ± 0.0	7.9 ± 0.0	7.7 ± 0.0	6.3	5.0	2.9	0.8	0.4	4.8	4.0	2.5	0.7	0.4	10.2 ± 0.0
256	7.4 ± 0.0	X	X	X	X	12.7 ± 0.0	12.6 ± 0.0	12.5 ± 0.0	12.2 ± 0.0	11.8 ± 0.0	8.8	5.5	3.1	0.8	0.4	6.5	4.2	2.5	0.7	0.4	15.7 ± 0.0
512	X	X	X	X	X	18.4 ± 0.0	18.2 ± 0.0	18.0 ± 0.0	17.5 ± 0.0	16.6 ± 0.0	4.6	2.8	1.6	0.8	0.4	3.8	2.4	1.4	0.7	0.4	23.2 ± 0.0
1024	X	X	X	X	X	23.4 ± 0.0	23.2 ± 0.0	22.9 ± 0.0	22.2 ± 0.0	21.0 ± 0.0	4.8	2.8	1.6	0.8	0.4	3.9	2.4	1.4	0.7	0.4	30.0 ± 0.0

Table K.2: End-to-end decoding throughput (thousands of tokens per second) with CodeLlama-7B on 8xA100 40 GB GPUs when generating 256 tokens. An x indicates the model does not have the required memory to run.

	FlashAttention					Hydragen					vLLM (No Tokenization)					vLLM					Upper Bound (No Attention)
Batch	Prefix length					Prefix length					Prefix length					Prefix length					Prefix length
Size	1K	2K	4K	8K	16K	1K	2K	4K	8K	16K	1K	2K	4K	8K	16K	1K	2K	4K	8K	16K	All
32	1.7 ± 0.0	1.4 ± 0.0	1.1 ± 0.0	0.7 ± 0.0	X	2.0 ± 0.0	2.0 ± 0.0	1.9 ± 0.0	1.8 ± 0.0	1.8 ± 0.0	1.8 ± 0.0	1.8 ± 0.0	1.8 ± 0.0	0.6 ± 0.0	0.4 ± 0.0	1.6 ± 0.0	1.6 ± 0.0	1.5 ± 0.0	0.5 ± 0.0	0.3 ± 0.0	2.3 ± 0.0
64	2.9 ± 0.0	2.3 ± 0.0	1.6 ± 0.0	X	X	3.6 ± 0.0	3.6 ± 0.0	3.6 ± 0.0	3.4 ± 0.0	3.4 ± 0.0	3.5 ± 0.1	3.5 ± 0.0	2.9 ± 0.1	0.7 ± 0.0	0.4 ± 0.0	3.0 ± 0.0	2.9 ± 0.1	2.4 ± 0.0	0.6 ± 0.0	0.4 ± 0.0	4.2 ± 0.0
128	4.0 ± 0.0	2.9 ± 0.0	X	X	X	5.8 ± 0.0	5.7 ± 0.2	5.6 ± 0.0	5.6 ± 0.0	5.7 ± 0.0	5.5	4.7 ± 0.1	3.0	0.8	0.4	4.8	3.8 ± 0.1	2.6	0.7	0.4	6.8 ± 0.0
256	5.7 ± 0.0	X	X	X	X	9.6 ± 0.0	9.3 ± 0.0	9.4 ± 0.0	9.2 ± 0.0	8.8 ± 0.0	8.0	5.5 ± 0.1	3.2	0.8	0.4	6.1	4.3 ± 0.1	2.7	0.7	0.4	11.4 ± 0.0
512	X	X	X	X	X	13.4 ± 0.0	13.3 ± 0.0	13.2 ± 0.0	12.9 ± 0.0	12.3 ± 0.0	4.7	2.7 ± 0.0	1.6	0.8	0.4	4.1	2.4 ± 0.0	1.4	0.8	0.4	16.1 ± 0.0
1024	X	X	X	X	X	15.6 ± 0.0	15.5 ± 0.0	15.3 ± 0.0	14.8 ± 0.0	14.0 ± 0.0	4.9 ± 0.0	2.8 ± 0.0	1.6 ± 0.0	0.8 ± 0.0	0.4 ± 0.0	4.2 ± 0.0	2.5 ± 0.0	1.4 ± 0.0	0.7 ± 0.0	0.4 ± 0.0	18.5 ± 0.0

Table K.3: End-to-end decoding throughput (thousands of tokens per second) with CodeLlama-13B on 8xA100 40 GB GPUs when generating 128 tokens. An x indicates the model does not have the required memory to run.

	FlashAttention					Hydragen					vLLM (No Tokenization)					vLLM					Upper Bound (No Attention)
Batch	Prefix length					Prefix length					Prefix length					Prefix length					Prefix length
Size	1K	2K	4K	8K	16K	1K	2K	4K	8K	16K	1K	2K	4K	8K	16K	1K	2K	4K	8K	16K	All
32	1.7 ± 0.0	1.4 ± 0.0	1.1 ± 0.0	0.7 ± 0.0	X	1.9 ± 0.0	1.9 ± 0.0	1.9 ± 0.0	1.8 ± 0.0	1.8 ± 0.0	1.8 ± 0.0	1.7 ± 0.0	1.8 ± 0.0	0.5 ± 0.0	0.3 ± 0.0	1.6 ± 0.0	1.6 ± 0.0	1.5 ± 0.0	0.5 ± 0.0	0.3 ± 0.0	2.3 ± 0.0
64	2.8 ± 0.0	2.2 ± 0.0	1.6 ± 0.0	X	X	3.5 ± 0.0	3.5 ± 0.0	3.4 ± 0.0	3.2 ± 0.0	3.3 ± 0.0	3.4 ± 0.1	3.4 ± 0.0	2.9 ± 0.0	0.7 ± 0.0	0.4 ± 0.0	3.0 ± 0.1	2.7 ± 0.2	2.2 ± 0.1	0.6 ± 0.0	0.4 ± 0.0	4.2 ± 0.0
128	3.8 ± 0.0	2.8 ± 0.0	X	X	X	5.6 ± 0.0	5.5 ± 0.0	5.3 ± 0.0	5.4 ± 0.0	5.2 ± 0.0	5.4	4.6	3.0	0.8	0.4	4.6	3.7	2.4	0.7	0.4	6.8 ± 0.0
256	5.4 ± 0.0	X	X	X	X	8.9 ± 0.0	8.7 ± 0.0	8.8 ± 0.0	8.7 ± 0.0	8.4 ± 0.0	7.6	5.5	3.1	0.8	0.4	5.9	4.3	2.5	0.7	0.4	11.3 ± 0.0
512	X	X	X	X	X	12.3 ± 0.0	12.3 ± 0.0	12.2 ± 0.0	12.0 ± 0.0	11.4 ± 0.0	4.4	2.7	1.5	0.8	0.4	3.8	2.4	1.4	0.7	0.4	16.1 ± 0.0

Table K.4: End-to-end decoding throughput (thousands of tokens per second) with CodeLlama-13B on 8xA100 40 GB GPUs when generating 256 tokens. An x indicates the model does not have the required memory to run.

	FlashAttention					Hydragen					vLLM (No Tokenization)					vLLM					Upper Bound (No Attention)
Batch	Prefix length					Prefix length					Prefix length					Prefix length					Prefix length
Size	1K	2K	4K	8K	16K	1K	2K	4K	8K	16K	1K	2K	4K	8K	16K	1K	2K	4K	8K	16K	All
32	1.4 ± 0.0	1.4 ± 0.0	1.2 ± 0.0	1.0 ± 0.0	0.8 ± 0.0	1.4 ± 0.0	1.4 ± 0.0	1.4 ± 0.0	1.4 ± 0.0	1.4 ± 0.0	1.5 ± 0.0	1.4 ± 0.0	1.2 ± 0.0	0.5 ± 0.0	0.3 ± 0.0	1.5 ± 0.0	1.3 ± 0.0	1.1 ± 0.0	0.5 ± 0.0	0.3 ± 0.0	1.6 ± 0.0
64	2.5 ± 0.0	2.3 ± 0.1	2.1 ± 0.0	1.8 ± 0.0	1.3 ± 0.0	2.6 ± 0.0	2.6 ± 0.0	2.5 ± 0.0	2.5 ± 0.0	2.5 ± 0.0	2.6 ± 0.0	2.3 ± 0.0	1.9 ± 0.0	0.7 ± 0.0	0.4 ± 0.0	2.4 ± 0.0	2.1 ± 0.1	1.6 ± 0.0	0.6 ± 0.0	0.4 ± 0.0	2.9 ± 0.0
128	3.8 ± 0.0	3.4 ± 0.0	2.8 ± 0.0	2.1 ± 0.0	X	4.2 ± 0.0	4.1 ± 0.0	4.1 ± 0.0	4.0 ± 0.0	3.9 ± 0.0	3.8	3.0	2.3	0.8	0.4	3.4	2.7	2.0	0.7	0.4	4.4 ± 0.3
256	6.0 ± 0.0	5.3 ± 0.0	4.4 ± 0.0	X	X	6.6 ± 0.0	6.6 ± 0.0	6.5 ± 0.0	6.3 ± 0.0	5.9 ± 0.0	5.1	3.9	2.8	0.8	0.4	4.4	3.3	2.4	0.8	0.4	7.2 ± 0.2
512	7.0 ± 0.0	6.0 ± 0.0	X	X	X	8.2 ± 0.0	8.1 ± 0.0	8.0 ± 0.0	7.8 ± 0.0	7.3 ± 0.0	4.2	2.7	1.5	0.8	0.4	3.6	2.4	1.4	0.8	0.4	8.8 ± 0.1
1024	X	X	X	X	X	9.4 ± 0.0	9.2 ± 0.0	9.0 ± 0.0	8.5 ± 0.0	7.6 ± 0.0	4.3	2.8	1.6	0.8	0.4	3.7	2.5	1.4	0.8	0.4	9.9 ± 0.2
2048	X	X	X	X	X	10.4 ± 0.0	10.3 ± 0.0	10.0 ± 0.0	9.4 ± 0.0	8.5 ± 0.0	4.3	2.7	1.5	0.8	0.4	3.7	2.4	1.4	0.8	0.4	11.0 ± 0.0
4096	X	X	X	X	X	11.1 ± 0.0	11.0 ± 0.0	10.7 ± 0.0	10.2 ± 0.0	9.4 ± 0.0	4.0	2.6	1.4	0.8	0.4	3.5	2.3	1.3	0.7	0.4	11.6 ± 0.0

Table K.5: End-to-end decoding throughput (thousands of tokens per second) with CodeLlama-34B on 8xA100 40 GB GPUs when generating 128 tokens. An x indicates the model does not have the required memory to run.

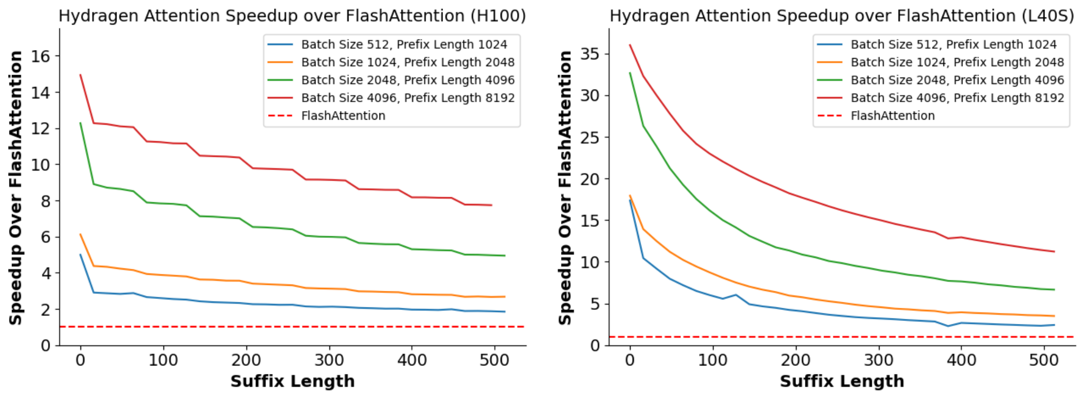


Figure K.1: Speedup of Hydragen attention over FlashAttention for various batch sizes, shared prefix lengths and suffix lengths on an H100 (left) and an L40S (right) GPU.

	FlashAttention					Hydragen					vLLM (No Tokenization)					vLLM					Upper Bound (No Attention)
Batch	Prefix length					Prefix length					Prefix length					Prefix length					Prefix length
Size	1K	2K	4K	8K	16K	1K	2K	4K	8K	16K	1K	2K	4K	8K	16K	1K	2K	4K	8K	16K	All
32	1.4 ± 0.0	1.3 ± 0.0	1.2 ± 0.0	1.1 ± 0.0	0.8 ± 0.0	1.4 ± 0.0	1.4 ± 0.0	1.4 ± 0.0	1.4 ± 0.0	1.3 ± 0.0	1.5 ± 0.0	1.4 ± 0.0	1.2 ± 0.0	0.5 ± 0.0	0.3 ± 0.0	1.5 ± 0.0	1.3 ± 0.1	1.1 ± 0.0	0.5 ± 0.0	0.3 ± 0.0	1.5 ± 0.1
64	2.5 ± 0.0	2.4 ± 0.0	2.1 ± 0.0	1.8 ± 0.0	1.3 ± 0.0	2.5 ± 0.0	2.5 ± 0.0	2.5 ± 0.0	2.5 ± 0.0	2.4 ± 0.0	2.6 ± 0.0	2.3 ± 0.0	1.8 ± 0.0	0.7 ± 0.0	0.4 ± 0.0	2.3 ± 0.1	2.0 ± 0.0	1.6 ± 0.0	0.6 ± 0.0	0.4 ± 0.0	2.8 ± 0.1
128	3.8 ± 0.0	3.4 ± 0.0	2.8 ± 0.0	2.1 ± 0.0	X	4.1 ± 0.0	4.1 ± 0.0	4.0 ± 0.0	4.0 ± 0.0	3.8 ± 0.0	3.7	3.0	2.2	0.7	0.4	3.2	2.6	2.0	0.7	0.4	4.5 ± 0.1
256	5.8 ± 0.0	5.3 ± 0.0	4.3 ± 0.0	X	X	6.5 ± 0.0	6.5 ± 0.0	6.4 ± 0.0	6.2 ± 0.0	5.8 ± 0.0	5.0	3.9	2.7	0.8	0.4	4.2	3.3	2.3	0.7	0.4	7.1 ± 0.2
512	6.8 ± 0.0	5.9 ± 0.0	X	X	X	8.0 ± 0.0	8.0 ± 0.0	7.9 ± 0.0	7.6 ± 0.0	7.2 ± 0.0	3.9	2.6	1.5	0.8	0.4	3.5	2.3	1.4	0.7	0.4	8.8 ± 0.1
1024	X	X	X	X	X	9.2 ± 0.0	9.1 ± 0.0	8.8 ± 0.0	8.3 ± 0.0	7.5 ± 0.0	3.9	2.6	1.4	0.8	0.4	3.6	2.4	1.4	0.7	0.4	9.9 ± 0.0
2048	X	X	X	X	X	10.3 ± 0.0	10.1 ± 0.0	9.8 ± 0.0	9.3 ± 0.0	8.4 ± 0.0	4.0	2.6	1.5	0.8	0.4	3.6	2.4	1.4	0.7	0.4	11.0 ± 0.0

Table K.6: End-to-end decoding throughput (thousands of tokens per second) with CodeLlama-34B on 8xA100 40 GB GPUs when generating 256 tokens. An x indicates the model does not have the required memory to run.

K.2 Microbenchmarks

I repeat the A100 microbenchmark experiment from Section 5.4 on H100 and L40S GPUs, reporting the results in Figure K.1. The L40S has the highest ratio of FLOPs to memory bandwidth of the three GPUs and therefore derives the most benefit from Hydragen's elimination of memory bottlenecks. While the compute-to-bandwidth ratio is higher on an H100 than on an A100, I measure similar speedups on both cards. This stems from the fact that the `flash-attn` package that I use is not currently optimized for Hopper GPUs, and therefore achieves a lower device utilization on an H100 vs an A100.

L | Hydragen Experimental Details

L.1 End-to-End Benchmarks

The end-to-end benchmarks only measure decoding throughput and exclude the time required to compute the prefill. I measure “decode-only” time by initially benchmarking the time required to generate one token from a given prompt and subtracting that value from the time it takes to generate the desired number of tokens. This subtraction is particularly important in order to fairly evaluate vLLM baselines, since it appears that vLLM redundantly detokenizes the prompt for every sequence in the batch at the beginning of inference (this can take minutes for large batch sizes and sequence lengths). For the “vLLM no detokenization” baseline, I disable incremental detokenization in vLLM by commenting out this line.

For all FlashAttention and No Attention datapoints, I run 10 warmup iterations and use the following 10 iterations to compute throughput. For Hydragen datapoints, I run 10 warmup and 10 timing iterations when the batch size is less than 256, and for larger batch sizes I use three warmup and three timing iterations. I observe that shorter-running Hydragen benchmarks (those with smaller batch sizes, sequence lengths, model sizes, or completion lengths) can occasionally produce longer outlier times. This seems to be related not to decoding time itself, but to variations in prefilling time before decoding. For vLLM baselines (both with and without incremental detokenization), I use three warmup and timing iterations for all batch sizes below 128, as well as for all datapoints that are used in Figures 5.2(a) and 5.2(b). The longest-running vLLM runs can take many minutes to complete a single iteration, so for baselines above a batch size of 128 that only appear in the supplementary tables of Appendix K.1, I use one warmup and one timing iteration.

L.2 Microbenchmarks

In each microbenchmark, I run 1000 iterations of warmup before reporting the mean running time across 1000 trials. Between iterations, I flush the GPU L2 cache by writing to a 128MiB tensor. I use CUDA graphs when benchmarking in order to reduce CPU overhead, which can be important since some benchmarks can complete a single iteration in tens of microseconds.

L.3 Long Document Retrieval

To demonstrate the throughput benefits of using Hydragen to answer questions about a long document, I construct a document (with 19974 tokens) that contains arbitrary facts from which question/answer pairs can be easily generated.

Prefix and Suffix Content: The content of the document is a subset of *War and Peace* [108], modified to include procedurally generated facts of the form "The dog named {name} has fur that is {color}". The questions are of the form "What color is the fur of the dog named name?", where the answer is {color}. I construct 261 questions (256 testable questions plus five for the few-shot examples) and interleave these throughout sentences of the document. When benchmarking with a greater number of questions than 256, I duplicate questions when querying the model - this is instead of adding more questions to the document in order to constrain total document length.

Model and Accelerator Choice: I choose the Yi-6B-200k model because it is small enough to fit a large KV cache in memory (important when running baselines that redundantly store the document) while also supporting a long enough context to process the document. I distribute the model across four A100-40GB GPUs in order to maximize possible KV cache size (the model only has four key/value attention heads, making the use of tensor parallelism across more GPUs difficult). The reported measurements use the mean of five timing runs after ten warmup iterations.

L.4 Hierarchical Sharing in Competitive Programming

The dataset of 120 problems that I use for this benchmark comes from the introductory difficulty split of APPS. I filter out problems that include starter code. I use two few-shot examples (2400 tokens long) that come from the training split of APPS, while all of the eval examples come from the test split. I sample 512 tokens for every completion. I run this experiment using CodeLlama-7b on eight A100-40GB GPUs. I measure the total time to run inference on all 120 questions, excluding tokenization and detokenization time.