Model Cascading for Code: Reducing Inference Costs with Model Cascading for LLM Based Code Generation

ABSTRACT

The rapid development of large language models (LLMs) has led to significant advancements in code completion tasks. While larger models have higher accuracy, they also cost much more to run. Meanwhile, model cascading has been proven effective to conserve computational resources while enhancing accuracy in LLMs on natural language generation tasks. However, this strategy has not been used on code completion tasks, primarily because assessing the quality of code completions differs substantially from assessing natural language, where the former relies heavily on the functional correctness. To address this, we propose letting each model generate and execute a set of test cases for their solutions, and use the test results as the cascading threshold. We show that our model cascading strategy reduces computational costs while increases accuracy compared to using a single model. Furthermore, we introduce a heuristics to determine the optimal combination of the number of solutions, test cases, and test lines each model should generate, based on the budget. Ours is the first work to increase LLM code generation efficiency with model cascading.

KEYWORDS

Model Cascading, Code Completion, Large Language Models

ACM Reference Format:

. 2023. Model Cascading for Code: Reducing Inference Costs with Model Cascading for LLM Based Code Generation. In *Proceedings of ACM Conference (DAC '24).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3489517.3530518

1 INTRODUCTION

Recently, large language models (LLMs) trained on vast datasets of code have proven to be highly capable at programming tasks [7, 16, 21]. These models have been incorporated into popular AI programming assistants, e.g., GitHub Copilot [1], and have attracted millions of users. However, code completion using LLMs is expensive, since it requires *multiple* forward passes of *large* models. Moreover, prompts tend to be unique and cannot be cached. The high computational demands and large memory consumption of LLMs even during inference result in high energy consumption and massive operational costs to provide code completion as a service for millions of users [29]. Therefore, it is imperative to reduce the inference cost of code completion LLMs. methods include Prior methods to reduce LLM inference costs include quantization [17], pruning [19], knowledge distillation [12], altered attention mechanism [8], *etc.* These methods reduce inference costs for a single given model.

State-of-art code completion LLMs are typically available in multiple sizes, for instance, Codegen [21] (350M, 2B, 6B, 16B), Codegen-2 [20] (1B, 3B, 7B, 16B), StarCoder [15] (1B, 3B, 15B), representing not just a single model but a *family* of models. Smaller models have lower performance in terms of code correctness, but also lower inference-time and memory requirements. In this paper, we ask the following question: *can the capabilities of different models in a family be combined to improve the cost versus accuracy trade-offs?*

To answer this question, we propose and evaluate **model cascading for code**, which we will refer to as model cascading for short. Model cascading builds on a simple, intuitive idea: we only query big models when absolutely necessary, *i.e.*, if smaller models cannot provide correct code completions. In our implementation, prompts are sent to the smallest model first, working their way up to progressively bigger models depending on the quality of the models' responses in each stage.

We use our selected model families to further demonstrate the potential of model cascading in Table 1: Wizard-Python-V1.0's smallest 7B model achieves 53.9% accuracy on MBPP, while the largest 34B model achieves 61.8% accuracy. The additional 8% boost in accuracy comes at an estimated 10× cost. With model cascading, the hope is to route most queries to the smallest model, reserving challenging queries for the large model. Note that for some other models and datasets, notably, Codegen-Mono on APPS, smaller models are also *significantly* less performant—here model cascading might be less effective, since most queries would have to be routed to the largest model eventually.

A key challenge is deciding when to request assistance from a larger model. That is, a model must not only generate code, but also have the ability to *test* its code for correctness. One solution is for the designer to provide a test suite. However, this entails considerable design effort, and prior work in LLM-based code generation does not assume access to test suites. In model cascading, we instruct models to generate *both* code (up to a maximum of k potential solutions) and a set of test cases. We then employ a learned threshold for the success rate to decide when to seek assistance from a larger model. In this way, an initial user prompt cascades up from the smallest to the largest model, stopping when the pass rate exceeds a pre-defined learned threshold.

To the best of our knowledge, model cascading is the *first* solution that leverages the combined power of a family of black-box LLMs to improve the operational cost versus accuracy tradeoffs. Although we evaluate model families that differ in parameter size, model cascading is easily extended to other approaches that generate model families, for example, an LLM model with varying quantization levels.

Our empirical results show that the optimal cascading strategy is always better than using a single model, and in the best case, our

DAC '24, July 10–14, 2024, San Francisco, CA, USA
© 2023 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/3489517.3530518

model cascading scheme can save 49% of cost while achieving the same accuracy compared to using a single model.

Our contributions include:

- We show with experiments that model cascading with a self-testing threshold can decrease the computation cost while increase accuracy for the one-best answer in LLMs on code completion tasks.

- We propose a full heuristic pipeline to find the optimal set of parameters, in terms of lowest inference cost and highest pass@1 accuracy at a cost budget, for a given set of models on a given dataset. Our method fits the demand of an industrial code completion server. It is compatible to any set of multi-LLM system for code completion, and is fully black-box.

Table 1: Cost and Pass@1 accuracy with greedy search of all models on each dataset. Cost is estimated as $ per 1M tokens as described in Section 3.4, averaged on the three datasets. The APPS dataset refers to the introductory questions in the test set only.

LLM Family		Cost ($)	Accuracy (%)		
Name	Size		H.Eval	MBPP	APPS
Wizard-Python-V1.0	7B	2.22	56.7	53.9	16.4
	13B	7.87	64.6	55.0	21.4
	34B	23.34	72.6	61.8	23.7
Wizard-V1.0	1B	0.23	23.8	33.0	3.9
	3B	0.45	34.8	41.5	8.6
	15B	3.22	60.4	53.2	13.8
Codegen-Mono	350M	1.28	14.6	22.0	0.4
	2B	3.12	26.2	34.2	4.9
	6B	7.96	27.4	41.9	5.1
	16B	17.40	30.5	45.4	5.9

2 RELATED WORKS

LLMs for Code There is a large and growing body of work on LLM-based code generation. Recent fine-tuned LLM models for code include Code LLAMA [22], WizardCoder [18], etc. These models are typically evaluated using Pass@k metrics. That is, they generate k solutions and "pass" a test as long as at least one solution is correct. In practice, however, a developer would have to pick from the k solutions, requiring significant human effort.

Self Testing for Code Generation To assist models in ranking their own solutions, CodeT [5] has recently proposed that LLM models, in addition to solutions, generate test-cases as extra outputs, and run solutions through these test-cases to rank the best solutions. In our work, we utilize their solution-test score for cascading threshold. While CodeT's goal is only to rank-order solutions from a single model, model cascading on the other hand uses the same self-testing scheme to determine when to query the next larger model in the casade.

Iterative Code Completion Orthogonal to model cascading, methods like Chip-Chat [3] query LLMs in a chat-like way; iteratively querying the *same* model with error messages from a compiler, simulator or even feedback from a human. Repeating this procedure several times, these methods can increase model accuracy. Model cascading, in contrast, queries *different* models, starting from the smallest to the largest in sequence. Iterative feedback methods can be combined with model cascading, where the same model is repeatedly queried before moving up.

Speculative Decoding Speculative decoding is a very recent technique in the same spirit as model cascading that reduces inference time of a larger ("target") model with a smaller ("draft:) model [14]. The method first uses the smaller model to speculatively generate a certain number of tokens, let's the bigger model compare tokens with its own predictions, and has the bigger model take over when the two results diverge. Inference time is reduced because the checking procedure is faster than generating new tokens. Nonetheless, there are several limitations on this method. Firstly, it does not work with black-box models, which are increasingly common in commercial deployments. Further, speculative decoding only works for two models, and necessitates that both the draft and target models are queried. Since the draft model brings additional computation, it must be significantly smaller than the target model for the system to save time. Researchers found that the best performance is achieved when the draft model is around 20 times smaller than the target model [14]. In contrast model cascading seeks to use the small model as frequently as possible, seeking to avoid using the big(ger) models altogether.

LLM Cascading for Natural Language Finally, prior work has investigated cascading for natural language prompts. FrugalGPT [6] trains a small DistillBERT model [24] to output a score based on the query and the answer, and uses this score to determine when to query a larger model. So far, there is no published work in model cascading for code completion tasks, in part because output evaluation is more challenging.

3 METHODOLOGY

We now describe our proposed model cascading for code solution (see Figure 1 for an overview). We begin by describing our method to evaluate when to send queries to the next larger model.

3.1 Cascading Pipeline

Let there be a model family $\mathcal{M}$, with n models: $\{m_1, \ldots, m_n\}$, ordered in terms of their parameter sizes from smallest (and hence fastest) to largest. For a user prompt p, a natural language description of the function the user seeks to implement, we first ask the smallest model m_1 to generate k_1 solutions, in addition to k_1 tests, each consisting of up to l_1 test lines, i.e., the *assert* statements as shown in Figure 1. The test lines produced across all solutions are pooled together into a test suite T_1 (consisting of a total of up to $k_1 \times l_1$ tests) and used to evaluate each solution.

We adopt the same strategy to select a solution out of multiple as the prior works in self-testing [5, 27]. For each solution $s \in \mathcal{S} = \{s_1, \ldots, s_{k_1}\}$, we define $T_{pass} \subseteq T_1$, the subset of test lines that the solution passes, and we compute two measures of solution quality: $n_s = |T_{pass}|$, the number of passing test lines, and n_t, a measure of test *quality* among the passing test lines. n_t gives the highest number of solves that a passing test line has across all solutions; that is, if $solves(t, \mathcal{S})$ is the number of solutions that pass test line t, then $n_t = \max_{t \in T_{pass}}(solves(t, \mathcal{S}))$. Intuitively, n_t captures the idea that a test line (assuming it is not trivial) is higher quality if it is passed by multiple solutions. The overall solution quality is the product $n_s \times n_t$, which we use to rank solutions and decide if

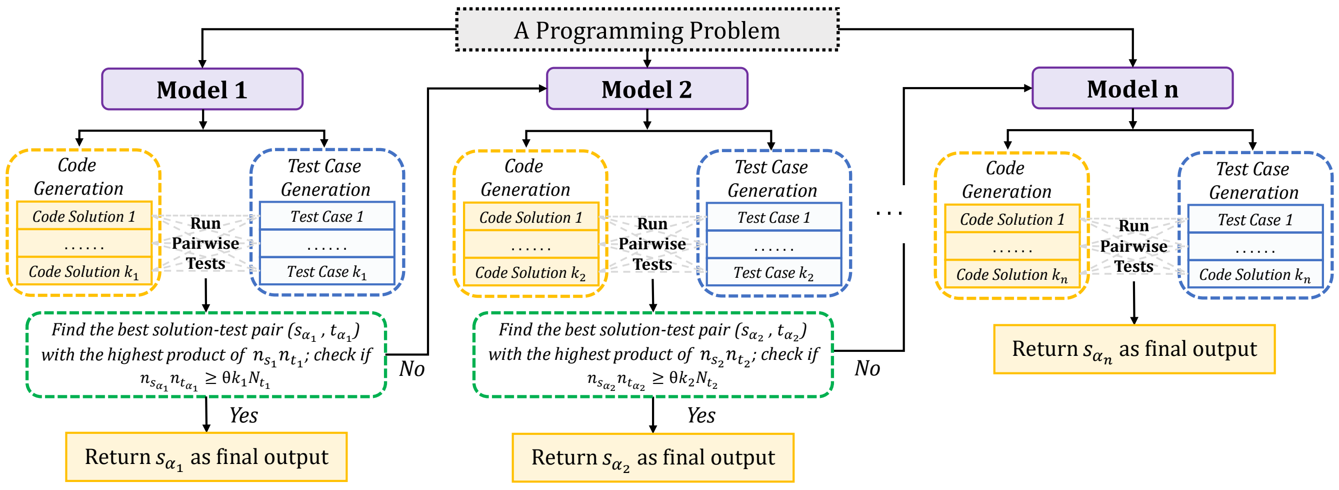
A Programming Problem
Model 1
Model 2
Model n
Code Generation
Test Case Generation
Code Solution 1
Code Solution k_1
Test Case 1
Test Case k_1
Run Pairwise Tests
Find the best solution-test pair $(s_{\alpha_1}, t_{\alpha_1})$ with the highest product of $n_{s_1}n_{t_1}$; check if $n_{s_{\alpha_1}}n_{t_{\alpha_1}} \geq \theta k_1 N_{t_1}$
No
Yes
Return s_{α_1} as final output
Code Solution k_2
Test Case k_2
Find the best solution-test pair $(s_{\alpha_2}, t_{\alpha_2})$ with the highest product of $n_{s_2}n_{t_2}$; check if $n_{s_{\alpha_2}}n_{t_{\alpha_2}} \geq \theta k_2 N_{t_2}$
Return s_{α_2} as final output
Code Solution k_n
Test Case k_n
Return s_{α_n} as final output

Figure 1: An overview of our proposed model cascading solution. Here we give an example with three models. Starting with model 1, we generate multiple code solutions, as we as multiple testcases. The best code solution and testcase is identified and scored; if the score exceeds a threshold, we move to the next solution.

we should adopt the current model's answer or proceed to a larger model; the highest scoring solution-test pair is denoted $(s_{\alpha_1}, t_{\alpha_1})$ and its score is $n_{s_{\alpha_1}} \times n_{t_{\alpha_1}}$.

To decide whether a solution is acceptable, we define a threshold parameter θ, which takes value from 0 to 1. Let there be N_{t_1} valid test lines in total. Note that $N_{t_1} \leq k_1 \times l_1$, because a model might stop generating before it reaches the maximum number of test lines l_1. If the highest solution score $n_{s_{\alpha_1}} \times n_{t_{\alpha_1}}$ is greater than or equal to $\theta \times k_1 \times N_{t_1}$, we adopt the corresponding solution-test pair $(s_{\alpha_1}, t_{\alpha_1})$ and end the pipeline. Otherwise, we repeat the above procedure with m_2: pass the same prompt to m_2, let it generate k_2 solutions and tests of l_2 test lines each, and score the solutions. We continue until we find a solution whose quality exceeds our threshold θ. When we use the largest model m_n, we skip the threshold check and directly adopt the highest-scoring solution.

3.2 Finding Optimal Parameters

We introduce the parameter selection procedure in this section. Our cascading system has mainly three sets of parameters: the number of solutions and tests generated for each model, $k_1 - k_n$; the number of test lines in each test for each model $l_1 - l_n$, and the global cascading threshold parameter θ.

We first set a range for each parameter. In our experiments, we let $k \in [-1, 0, 1, 3, 5, 10]$ for each model. When $k = -1$, it means we are skipping this model; when $k = 0$, it means we only generate one solution with greedy search, and exit with the solution as output, without generating tests; when $k = 1$, we generate one solution and one test with greedy search, and when $k > 1$, we generate k solutions and k tests with sampling, using temperature 0.8. For example, if we have 3 models in a cascading system, then $[k_1 = 3, k_2 = 5, k_3 = 0]$ and $[k_1 = 10, k_2 = -1, k_3 = 3]$ are both valid sets of parameters, but $[k_1 = 1, k_2 = 0, k_3 = 3]$ is not, because $k_2 = 0$ means we must exit on model 2, so we must have $k_3 = -1$.

Similarly, we let $l \in [0, 2, 4]$ for each model. $l = 0$ if and only if $k = 0$, which means we do not generate any code for test; when $l > 0$, we let the model generate a maximum of l lines starting with assert. Note that the model might stop the generation before it reaches l lines. In that case, we will have less than $k \times l$ test lines for the current model. Both k and l are parameters for each model in a cascading system. However, θ is a global parameter. The cascading thresholds from model 1 to model 2, and from model 2 to model 3, use the same value of θ. In our experiments, we sampled θ from $[0.0, 0.1, 0.2, \ldots 1.0]$.

Second, we sample a training set d_t out of the full dataset d to find the optimal parameters, and will test our selected combinations on the validation set d_v. In our experiments, d_t is randomly sampled, and it contains 30% of questions in d.

Third, on d_t, we run the cascading system in all the valid $[\theta, k_1 - k_n, l_1 - l_n]$ combinations as described in section 3.1, and calculate each of their cost and accuracy. Then, we make a cost-accuracy plot for all k and l combinations for each θ, similar to plots in Figure 3. We select the optimal cascading combinations by finding all the *Pareto points* in the plot—the points for which there is no other point with a higher accuracy and lower cost—and save these combinations as our optimal parameters. For comparison, we also save the optimal singular combinations by finding the Pareto points among the singular points, where there is only one model used in the system (i.e. $k_1 = -1$, $k_2 = 3$, $k_3 = -1$). We compare the difference between the Pareto points and singular Pareto points across plots, and find the best θ.

Finally, on d_v, we compare our cascading optimal combinations on the θ we selected with the singular optimal combinations.

3.3 Cascading Threshold Parameter (θ)

In this section we discuss the impact of θ on the system's cost and accuracy. As mentioned in section 3.1, θ decides when to use a larger model. When θ is close to 0, the small model's output is more likely to meet the standard and thus exit with the current output; on the other hand, when θ is close to 1, the system is more likely to use bigger models on each question. We demonstrate the impact of θ with an example in Figure 2. We used the WizardCoder-Python-V1.0 family to run full HumanEval dataset, with set values for $k_1 - k_3$, $l_1 - l_3$. As the value of θ increases, the cost increases monotonically, because using larger models more often always results in more computation. However, the highest *accuracy* is achieved at $\theta = 0.8$ rather than 1.0. The reason is that when smaller models have larger k values, they have more solutions to pick from and more tests to validate their solutions. That makes the outputs from smaller models potentially more accurate than larger models. On the other hand, a too rigorous bar would miss correct solutions from smaller models. Among the k solutions, there might be only a few of them correct, and so is the case among the $k \times l$ tests. If we set $\theta = 1.0$, it means all solutions must pass all tests, which is much unlikely for a large k.

3.4 Cost Calculation

All the models that we used are public, and thus we need to estimate the cost of running them based on the inference time they take on GPUs. To this end, we calculate the cost by multiplying the number of tokens generated by the per-token cost, similar to pricing schemes used by commercial model providers like OpenAI.

We deploy a server with multiple NVIDIA Geforce RTX 3090 GPUs, each with 24GB of VRAM. We run each model with 2^k GPUs using the minimum k that provides enough VRAM to run inference on a batch of 10 prompts. For each dataset, we randomly sample the maximum number of questions that can fit into a batch without running out of VRAM. We estimate the time to complete the inference T, and the number of non-eos tokens from all answers N_t. Finally, we look up the hourly cost to rent an RTX 3090 GPU online, c. In this case, we use the price from runpod.io where $C = \$0.44/hr$ [23]. Hence, the per-token cost on each model is $c = TN_tC$.

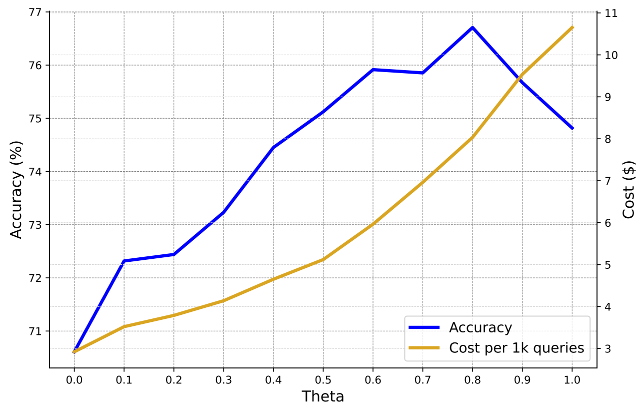

Figure 2: Cascading result using the three models in Wizard-Python-V1.0 family on full HumanEval dataset, with $k_1 = 5$, $k_2 = 3$, $k_3 = 0$, $l_1 = 4$, $l_2 = 4$, $l_3 = 4$. We sample θ value on 0, 0.1, 0.2 … 1.0. The highest accuracy is 76.7%, with a cost of $8.03 per 1k queries and $\theta = 0.8$.

Our cost statistics for all models on the HumanEval dataset are provided in Table 2. We did not include the length of the input prompt in the calculation, even though it also affects the inference time. We account for this issue by collecting the stats for each model family on each dataset, so that questions in the same dataset should have similar lengths.

Model Family	Size	N.gpu	Time (h)	Batch	Cost ($)
WizardCoder-Python-V1.0	7B	2	2.17	20	1.91
	13B	4	3.78	24	6.65
	34B	8	5.75	48	20.24
WizardCoder-V1.0	1B	1	0.31	180	0.13
	3B	1	0.58	80	0.26
	15B	2	1.97	32	1.74
Codegen-mono	350M	1	2.36	48	1.04
	2B	1	5.97	12	2.63
	6B	2	6.44	12	5.67
	16B	4	9.00	12	15.84

Table 2: Time and Cost stats per 1M tokens generated of all selected models on HumanEval dataset. We generated all completions with sampling. Time is averaged across 10 runs.

4 EXPERIMENT SETUP

4.1 Models

There are multiple open-source code generation language models, including Codegen [21], InCoder [9], StarCoder [15], Code LLAMA [22], CodeT5+ [28], WizardCoder [18], etc.

For our experiments, we chose three model families to work with: the Codegen-mono family (4 models, each having 350M, 2B, 6B, 16B parameters), the WizardCoder-V1.0 family (3 models, each having 1B, 3B, 15B parameters), and the WizardCoder-Python-V1.0 family (3 models, each having 7B, 13B, 34B parameters). The Codegen-mono family models were trained on the THEPILE, BIGQUERY and BIGPYTHON datasets. The latter two WizardCoder families were trained on from the Starcoder family and LLAMA-2 family [25], on an instructive dataset Code Alpaca [4]. Their checkpoints share the same structure as their respective foundation models. These three model families share features that are suitable for our experiments:

1. They come in families of multiple sizes spanning a relatively wide range. the larger difference in size, the more inference time we can save when adopting a smaller checkpoint's output.

2. Among each family, all checkpoints are trained from the same datasets, so that the larger checkpoints always have a higher accuracy than the smaller checkpoints. If a larger checkpoint has lower accuracy than a smaller one, we should eliminate it from the cascading system.

Performance-wise, WizardCoder-Python-V1.0 and WizardCoder-V1.0 families have the highest accuracy on HumanEval and MBPP datasets among all open-source code completion models; the Codegen-mono family also has competitive performance in the corresponding sizes.

4.2 Datasets

For our experiments, we use HumanEval [7], MBPP-sanitized [2], and APPS-test introductory level subset [11]. They have 164, 427,

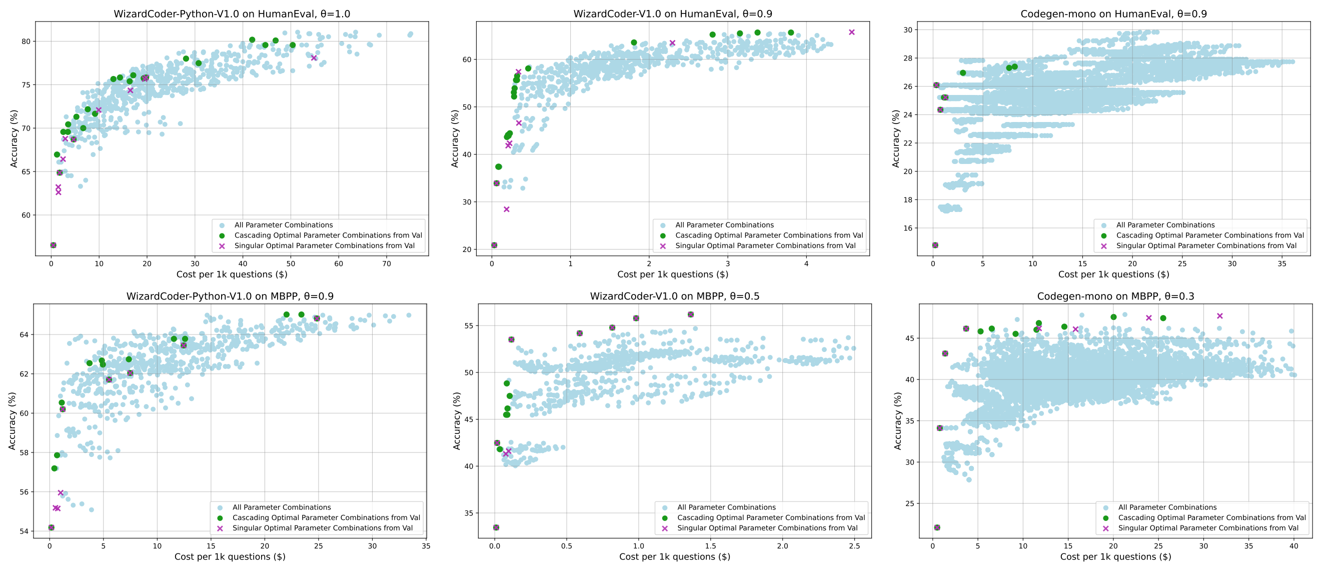

Figure 3: All $[k, t]$ combinations cost-accuracy results for WizardCoder-Python-V1.0, WizardCoder-V1.0, Codegen-mono families on HumanEval and MBPP datasets, each plot with a selected optimal θ. The first two families have 3 models, and the Codegen-mono family has 4 models. Light blue dots represent all parameter combinations; green dots represent the cascading optimal combinations; purple crosses represent single-model optimal combinations. All the marked combinations are selected from the training set.

1000 questions respectively. We show the greedy accuracy of all models in each model family on each dataset in Table 1.

In APPS-test, we only provide cascading results on the introductory level questions using WizardCoder-Python-V1.0 model family. The reason is that the accuracy of other model families on intro-level, as well as all model families on interview and competition levels, are are below 10% on average. When the test set is too challenging for the model family, the cascading system will waste time rather than save time on smaller models.

4.3 Environment

Our server is consisted of NVIDIA GeForce 3090 GPUs. We use Cuda Toolkit 11.8, and a pytorch-based python implementation from huggingface transformers [26] in version 4.31.0 with accelerate [10], which is in parallel to the WizardCoder environment. We implemented code completions with early exit and a max number of 1024 generated tokens. We did not implement vllm [13] or triton inference server, but our method is compatible with these acceleration strategies.

5 RESULTS

In Figure 3, we plot the throughput and accuracy of all the 3 model families on HumanEval and MBPP datasets. For each run, in the validation set, we randomly sampled 30% of all questions, and selected the Pareto points as optimal combinations. In the test set with the rest of the 70% questions, we find that the optimal parameter combinations from our cascading scheme are in general more accurate and less costly from single models. In the best case, as seen in WizardCoder-Python-V1.0 family on HumanEval, the

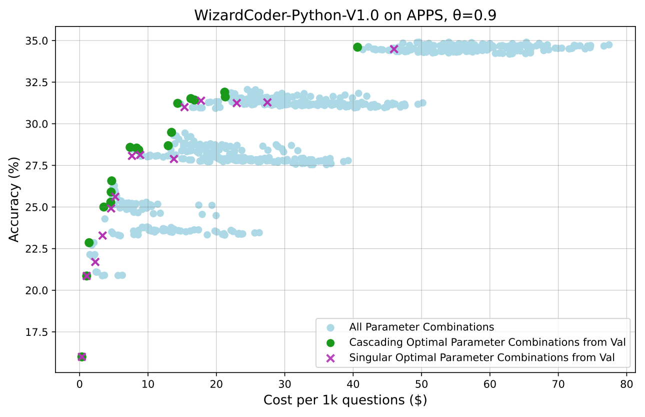

Figure 4: WizardCoder-Python-V1.0 family on APPS-test introductory level subset. The setting is the same as all plots in Figure 3.

cascading scheme can save 49% of cost while achieving the same level of accuracy as using a single model.

However, the performance varies across models and datasets. Our strongest model family, WizardCoder-Python-V1.0, has consistently stronger performance with model cascading than single models on all datasets. It still achieves visible cost savings on APPS-test even though its models' greedy search accuracy is not very high, as shown in Table 1. We speculate that the reason is the highly capable models generate high-quality tests, even when the questions are difficult.

On the other hand, model cascading did not achieve strong effect for Codegen-mono on HumanEval, and WizardCoder-V1.0 on MBPP. Several cascading optimal combinations overlapped with single model optimal combinations, and the non-overlapped ones are not saving cost while preserving accuracy. Such behavior might be due to the poor quality of smaller models in the family. When smaller models are not helpful, the optimal cascading strategy is similar to using larger models alone.

6 CONCLUSION

We proposed a full heuristic pipeline to find the optimal cascading strategy, and proved that it gains lower cost and higher accuracy for various model families and datasets. The effect is more obvious when the model family has higher quality in code generation. Our experiment focused on the python code generation tasks, but the method is compatible to all languages. In our experiments, we only attempted a global value for θ. To further examine the potential of model cascading for code completions, one can explore the results using a different θ value for each model and each $[k, t]$ combination.

REFERENCES

[1] 2021. GitHub Copilot · Your AI pair programmer. https://copilot.github.com/

[2] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021).

[3] Jason Blocklove, Siddharth Garg, Ramesh Karri, and Hammond Pearce. 2023. Chip-Chat: Challenges and Opportunities in Conversational Hardware Design. arXiv:2305.13243 [cs.LG]

[4] Sahil Chaudhary. 2023. Code Alpaca: An Instruction-following LLaMA model for code generation. https://github.com/sahil280114/codealpaca.

[5] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2023. CodeT: Code Generation with Generated Tests. In *The Eleventh International Conference on Learning Representations*. https://openreview.net/forum?id=ktrw68Cmu9c

[6] Lingjiao Chen, Matei Zaharia, and James Zou. 2023. FrugalGPT: How to Use Large Language Models While Reducing Cost and Improving Performance. arXiv:2305.05176 [cs.LG]

[7] 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. 2021. Evaluating Large Language Models Trained on Code. (2021). arXiv:2107.03374 [cs.LG]

[8] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems* 35 (2022), 16344–16359.

[9] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999* (2022).

[10] Sylvain Gugger, Lysandre Debut, Thomas Wolf, Philipp Schmid, Zachary Mueller, Sourab Mangrulkar, Marc Sun, and Benjamin Bossan. 2022. Accelerate: Training and inference at scale made simple, efficient and adaptable. https://github.com/huggingface/accelerate.

[11] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. 2021. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938* (2021).

[12] Cheng-Yu Hsieh, Chun-Liang Li, Chih-Kuan Yeh, Hootan Nakhost, Yasuhisa Fujii, Alexander Ratner, Ranjay Krishna, Chen-Yu Lee, and Tomas Pfister. 2023. Distilling step-by-step! outperforming larger language models with less training data and smaller model sizes. *arXiv preprint arXiv:2305.02301* (2023).

[13] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.

[14] Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2023. Fast Inference from Transformers via Speculative Decoding. arXiv:2211.17192 [cs.LG]

[15] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. StarCoder: may the source be with you! arXiv:2305.06161 [cs.CL]

[16] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with AlphaCode. *Science* 378, 6624 (2022), 1092–1097. https://doi.org/10.1126/science.abq1158 arXiv:https://www.science.org/doi/pdf/10.1126/science.abq1158

[17] Zechun Liu, Barlas Oguz, Changsheng Zhao, Ernie Chang, Pierre Stock, Yashar Mehdad, Yangyang Shi, Raghuraman Krishnamoorthi, and Vikas Chandra. 2023. LLM-QAT: Data-Free Quantization Aware Training for Large Language Models. *arXiv preprint arXiv:2305.17888* (2023).

[18] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. WizardCoder: Empowering Code Large Language Models with Evol-Instruct. *arXiv preprint arXiv:2306.08568* (2023).

[19] Xinyin Ma, Gongfan Fang, and Xinchao Wang. 2023. LLM-Pruner: On the Structural Pruning of Large Language Models. *arXiv preprint arXiv:2305.11627* (2023).

[20] Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. 2023. CodeGen2: Lessons for Training LLMs on Programming and Natural Languages. arXiv:2305.02309 [cs.LG]

[21] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. *arXiv preprint* (2022).

[22] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code Llama: Open Foundation Models for Code. arXiv:2308.12950 [cs.CL]

[23] RunPod. 2023. *GPU Cloud Service.* http://runpod.io

[24] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2020. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. arXiv:1910.01108 [cs.CL]

[25] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).

[26] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, Online, 38–45. https://www.aclweb.org/anthology/2020.emnlp-demos.6

[27] Weimin Xiong, Yiwen Guo, and Hao Chen. 2023. The Program Testing Ability of Large Language Models for Code. arXiv:2310.05727 [cs.CL]

[28] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, et al. 2023. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x. *arXiv preprint arXiv:2303.17568* (2023).

[29] Xunyu Zhu, Jian Li, Yong Liu, Can Ma, and Weiping Wang. 2023. A Survey on Model Compression for Large Language Models. arXiv:2308.07633 [cs.CL]