Large Language Models for Serverless Function Generation: An Investigation on FaaS Performance

Xinghan Chen
kirito20@uw.edu
University of Washington
Tacoma, Washington, USA

Robert Cordingly
rcording@uw.edu
University of Washington
Tacoma, Washington, USA

Ling-Hong Hung
lhhung@uw.edu
University of Washington
Tacoma, Washington, USA

Wes Lloyd
wlloyd@uw.edu
University of Washington
Tacoma, Washington, USA

Abstract

The rapidly expanding use of large language models (LLMs) for code generation introduces promising opportunities and challenges in cloud computing, particularly regarding runtime performance. Previous research has focused primarily on LLMs in non-cloud environments, focusing on the largest, resource-intensive LLMs while neglecting mid-range LLMs (1.5–8 billion or 32–70 billion parameters) that can operate on consumer-grade hardware. In this work, we address this gap by investigating both large-scale publicly accessible LLMs and smaller locally hosted LLMs (e.g. those that can be deployed on small clusters or local machines) for serverless function code generation. We investigate serverless function runtime performance, and also cloud hosting costs for code generated by different LLMs. Our study examines LLM-generated solutions for three code generation prompt categories (e.g. common benchmarks, interview questions, and custom problems that LLMs may not have ever seen). We analyze how prompt characteristics, such as token count and metadata, impact serverless function runtime performance in the cloud for each solution.

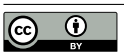
CCS Concepts

• **Software and its engineering** → **Genetic programming**; *Error handling and recovery*; *Software performance*; • **Computer systems organization** → **Cloud computing**.

Keywords

Large language models (LLMs), FaaS, Serverless computing, Code generation, Cloud Computing, Prompt engineering, Performance evaluation

ACM Reference Format:
Xinghan Chen, Robert Cordingly, Ling-Hong Hung, and Wes Lloyd. 2025. Large Language Models for Serverless Function Generation: An Investigation on FaaS Performance . In *11th International Workshop on Serverless Computing (WoSC11 '25), December 15–19, 2025, Nashville, TN, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3774899.3775016



WoSC11 '25, Nashville, TN, USA

ACM ISBN 979-8-4007-2302-5/25/12
https://doi.org/10.1145/3774899.3775016

1 Introduction

Large Language Models (LLMs) have quickly become a cornerstone of modern software development, capable of automatically generating code from natural language prompts. This capability enables opportunities for productivity gains for cloud-native and serverless development. However, trade-offs between LLM size, correctness, function execution cost, and runtime performance remain underexplored, especially in serverless computing environments where runtime and memory utilization directly impact cost. Most existing research has focused on performance metrics in general-purpose settings, often focusing on large LLMs such as GPT-4 or Llama-70B. In contrast, in this paper we investigate code generated using both large and small/medium-sized LLMs (LLMs that can be deployed on-premises or on edge hardware) and compare runtime performance, correctness, and hosting costs. We evaluate publicly available and locally hosted LLMs and analyze performance of their serverless function code solutions on Function-as-a-Service (FaaS) deployments for a set of coding tasks.

1.1 Research Questions

We investigate the following research questions:
RQ-1: (Generated Code Quality) What differences in code quality (syntactic correctness, functional correctness, and static analysis metrics) are observed when using smaller vs. larger LLMs to generate serverless function code for common algorithms, interview-style questions, and potentially unseen custom problems?
RQ-2: (Performance and Cost) What runtime performance and cloud execution cost differences occur on serverless FaaS platforms when using smaller vs. larger LLMs to generate serverless function code for common algorithms, interview-style algorithms, and custom problems potentially unseen by the LLMs when deployed on a public FaaS platform with standard resource configurations?

2 Background and Related Work

Large-scale LLMs have shown great ability in automated code generation in a variety of domains. AlphaCode has shown that large-scale LLMs can solve competitive programming tasks by generating a large number of candidate solutions and filtering for correctness [23]. However, AlphaCode's approach is computationally expensive and requires extensive infrastructure.

Recent research has challenged the idea that larger LLMs with a higher number of parameters are always better for code generation. Hassid et al. show that with a fixed compute budget (i.e. GPU time), medium-sized LLMs (e.g. 13 billion parameters) can outperform larger LLMs like CodeLlama-70B, because inferencing using smaller LLMs is faster enabling multiple inferencing attempts to produce multiple code results where the user can then choose the best code [17]. This suggests a practical strategy for code generation: perform more attempts with smaller LLMs in an attempt to improve success rates and resource efficiency.

At the same time, LLM code generation is increasingly used in serverless and cloud-native environments. Arun et al. evaluated the ability of LLMs to regenerate real-world serverless functions and architectural components and found that while LLMs generate functionally correct code, ensuring consistent code quality remains a challenge without access to a real execution environment [4].

In addition to pure code generation, recent research has also considered evaluation and efficiency improvements for code generated by LLMs. Liu et al. propose Differential Performance Evaluation (DPE), a framework to benchmark runtime performance under computational stress [24]. Similarly, PerfCodeGen iteratively optimizes LLM-generated code using a runtime feedback loop, achieving notable improvements compared to static reasoning [33]. EFFIBENCH further provided a benchmark to evaluate the runtime efficiency of LLM generated code for 1,000 LeetCode-style problems compared to the best human-written solutions [18]. Mercury is another tool that targets code efficiency by providing an evaluation benchmark, which combines correctness with a runtime percentile ranking of known solutions [13]. It contains 1,889 Python problems, each with a different test case, and derives runtime distributions based on historical commit results.

Furthermore, studies such as [7, 14] emphasize the need for more comprehensive evaluation frameworks that balance performance, scalability, and deployment feasibility. Coignion et al. confirmed these concerns with an empirical study that showed that, in some cases, LLM-generated solutions were not only correct, but also faster than human-written solutions [9].

Simultaneously, research has studied how to serve LLM inference efficiently using serverless backends. ServerlessLLM demonstrated techniques for low-latency model inference on serverless platforms [15], while LLMaaS explored trusted serverless substrates for LLM serving [6]. A recent effort surveyed challenges and opportunities for scalable AI inference using serverless computing [36]. Operational safety has come into focus where Wen et al. used LLMs to detect AWS serverless misconfigurations, pointing to secure-by-default deployment pipelines when automating application delivery [37].

Despite the above advances, relatively few studies have evaluated LLMs in live, cloud-deployed, serverless environments. Most studies have focused on evaluating generated source code offline using static analysis rather than by deploying code on the cloud and performing live performance evaluations.

3 Methodology

3.1 LLM Selection and Grouping

Table 1 summarizes the LLMs we evaluated. We selected LLMs that span three dimensions: (i) model size/class (Small, Medium, Large), (ii) deployment mode (Cloud vs. Local), and (iii) inference style (traditional next-token vs. dedicated reasoning/thinking models). We also selected diverse LLMs from various major vendors. This coverage allowed us to study not only raw code quality but also operational trade-offs of LLMs that emerge when using models that are hosted locally vs. by large cloud vendors.

Regarding the model size and class, "Large" models typically yield higher code quality but incur higher latency and cost; smaller models are faster (on the same compute resources) and cheaper, enabling more samples under a fixed budget.

With respect to deployment mode, LLMs deployed to the "Cloud" abstract hardware and autoscaling, while "Local" models (such as those run via Ollama on the Local server cluster) expose controllable, reproducible runtimes and are attractive for data locality or cost reasons [32].

Reasoning LLMs allocate more inference budget to multi-step deliberation and tool-choice planning, while "traditional" LLMs generate code in a single-pass style. We keep vendor defaults for "thinking" depth to reflect realistic usage [5].

Our set of models studied in this paper: (1) provide coverage of realistic choices. Practitioners commonly choose between small/fast and large/accurate, and between cloud convenience and local control; our LLMs cover these trade-offs. (2) include reasoning vs. sampling approaches. We directly compare deliberate reasoning approaches to traditional decoding under identical prompts and budget rules. (3) investigate vendor diversity. Cross-vendor results help separate modeling advances from platform effects (rate limits, context size, tokenizer, and pricing tiers listed in Table 1). (4) provide an operational study. By deploying generated functions to AWS Lambda to compare performance, our evaluation is widely relevant to developers and practitioners [34].

3.2 Prompts and Characteristics

Table 3 summarizes the ten code generation prompts used. Our prompts span a variety of use cases including: (i) data analytics (Healthcare 1/2; Sales Analysis), (ii) algorithmic (Perfect Number, Prime Number, Median of Two Arrays), (iii) kernel performance (PageRank*), (iv) image processing (Thumbnailer*), and (v) LeetCode-style problems (Minimal Cost Split, Car Collision), to probe the I/O utilization, library reliance, and compute intensity under FaaS of code generated using different LLMs.

Healthcare 1/2 and Sales Analysis accept CSV inputs and require schema parsing and aggregation. These prompts stress file I/O and computation. PageRank* and Thumbnailer* include four variants each (long vs. short; with vs. without libraries). "Nolib" prompts force the LLM to produce a full implementation without the use of external libraries, while the "long" prompt provided the whole structure of the program, and "short" only sent the basic feature request into the LLM. Minimal Cost Split, Prime/Perfect Numbers, and Median of Two Arrays consist of CPU-bound logic with minimal I/O.

The input-token length spans from 236–405 tokens (Thumbnailer*) and 254–367 (PageRank*) to 3,300 tokens (Healthcare 1/2). We use token count to normalize prompting cost across models and check whether longer, more-detailed prompts reduce syntax/logic defects. Given that tokenization differs by vendor, our reported counts are based on OpenAI's GPT4 tokenizer.

Table 1: LLMs used for Serverless Function Code Generation

Vendor	Model	Class	Runtime	Size	Style	Tokens
OpenAI	gpt-4o [27]	Large	Cloud	~1.8T	traditional	128,000
OpenAI	gpt-4o-mini [27]	Small	Cloud	8B	traditional	128,000
OpenAI	o1-mini [28]	Small	Cloud	Small	reasoning	128,000
OpenAI	o1 [28]	Large	Cloud	Large	reasoning	200,000
OpenAI	o3-mini [29]	Small	Cloud	Small	reasoning	200,000
DeepSeek	deepseek-reasoner [12]	Large	Cloud	671B/32B	reasoning	131,072
DeepSeek	deepseek-r1:8b [12]	Small	Local	8B	reasoning	131,072
DeepSeek	deepseek-r1:32b [12]	Medium	Local	32B	reasoning	131,072
Alibaba	qwen2.5-coder:7b [1]	Small	Local	7B	traditional	131,072
Facebook	llama3.3:70b [25]	Medium	Local	70B	traditional	8,192
Xai	grok-3-beta [38]	Large	Cloud	314B	traditional	131,072
Xai	grok-3-mini-beta [38]	Small	Cloud	Small	traditional	131,072
Google	gemini-2.5-pro-exp-03-25 [16]	Large	Cloud	Large	traditional	1,000,000
Google	gemini-2.0-flash [16]	Large	Cloud	Large	traditional	1,000,000

Table 2: Release Date, Updates, and Descriptions of Models

Model	Original	Update	Description
gpt-4o	05/13/24	03/26/25	Multimodal "omni" model handling text, images and audio.
gpt-4o-mini	07/18/24	–	Smaller, cheaper GPT-4o variant for high-volume or low-cost use.
o1-mini	09/12/24	05/25	Mini version of OpenAI's first "reasoning model," optimized for speed with chain-of-thought reasoning.
o1	12/05/24	05/25	Reasoning-focused model using chain of thought to solve complex math, science, and coding problems.
o3-mini	01/31/25	02/06/25	Compact o3 model with low/medium/high reasoning modes for technical tasks and fast responses.
deepseek-reasoner	05/28/24	05/28/25	Reasoning model with chain-of-thought separation to improve logical problem-solving.
deepseek-r1:8b	01/20/25	05/28/25	8B distilled R1 model.
deepseek-r1:32b	01/20/25	05/28/25	32B distilled R1 model.
qwen2.5-coder:7b	12/07/24	–	7B parameter version of Qwen2.5 Coder for code generation, repair and reasoning across languages.
llama3.3:70b	12/07/24	–	70B parameter open-source model with long context support.
grok-3	02/17/25	03/25	xAI's largest model.
grok-3-mini	02/17/25	03/25	Smaller Grok 3 variant trading accuracy for speed, offering similar reasoning modes.
gemini-2.5-pro	03/25/25	06/17/25	Gemini 2.5 Pro with advanced reasoning, and 1M context.
gemini-2.0-flash	01/30/25	04/17/25	Fast Gemini 2.0 optimized for speed.

Table 3: Summary of Code Generation Prompts

Prompt Name	Description	Token Count	Source
Health care 1	Healthcare payment records analysis from a CSV input	3,306	Original
Health care 2	State funding reduction calculation form a CSV input	3,244	Original
Sales Analysis	Daily sales income and order analysis	751	Original
Perfect number	Generation of perfect numbers up to n	382	Original
Prime number	Generation of prime numbers up to n	353	Original
Pagerank*	PageRank calculation on graph	367/352/261/254	SeBS [35]
Thumbnailer*	Image thumbnail generation	405/368/255/236	SeBS [35]
Minimal Cost Split	Minimum cost array splitting	308	LeetCode [22]
Car collision	Car fleet collision time computation	682	LeetCode [20]
Median of two array	Median calculation for sorted arrays	452	LeetCode [21]

* *Indicates prompts with four variants (long, short, no-library-short and no-library-long), for this prompt token count record all for variant in the order of long, short, no-library-short and no-library-long. All the token count is based on OpenAI GPT-4o & GPT-4o mini tokenalizer [31].*

3.3 Benchmarking Environment

Our research was conducted on AWS Lambda in the us-west-2 (Oregon) region. Functions were provisioned with 3008 MB memory, the minimum size that guarantees full access to 2 vCPUs [2]. Allocating fewer resources results in fractional CPU time shares, potentially degrading performance and increasing runtime variance. To ensure consistent and accurate benchmarks, we concentrated exclusively on evaluating runtime performance on Intel Xeon Platinum 8259CL CPUs, identified by a 2.50 GHz base frequency and 36,608 KB cache when inspecting the serverless function's Linux proc file system, from the 'stat' file. To determine the actual CPU type, we matched base frequency and cache size to CPUs with matching clock speed and cache sizes on publicly available Amazon EC2 instances.

To systematically analyze workload performance and resource utilization, we used the Serverless Application Analytics Framework (SAAF) [11, 10], to collect CPU, memory, and I/O utilization metrics. Each LLM-generated function was deployed using SAAF's function template, ensuring standard metric collection. Metric averages were calculated using 100 warm executions per function, excluding cold-start runs for consistency.

3.4 Code Generation

We systematically generated code using multiple LLMs described in Table 1. We provided each LLM with clearly defined prompts for each programming task as described in Table 3 and an initial common prompt to specify background and context adapted from Cursor [3]. Cursor is an integrated development environment (IDE) that interfaces with LLMs to assist in coding tasks. Cursor is an enhanced version of Visual Studio Code with intelligent code completion, AI-powered chat, and code generation capabilities. Our evaluation investigated code generated for our ten distinct prompts spanning diverse computational problems; eight prompts had a single variant, and two prompts featured four variants to evaluate LLM robustness.

Each prompt was provided only once to each LLM. We captured the initial LLM outputs and made no prompt refinements or additional queries. Our evaluation included LLMs hosted in cloud environments and LLMs executed locally on the Hyak compute cluster. To orchestrate experiments and analyze data, we leveraged a custom Jupyter Notebook [19], OpenAI's Python API library [30], and Ollama [26].

3.5 Static and Functional Evaluation

To complement functional and performance evaluations, we performed static analysis of all generated functions using two widely adopted Python code assessment tools: Radon and Pylint. Radon reports code metrics such as cyclomatic complexity, maintainability index, and Halstead difficulty. Pylint provides style and linting scores reflecting conformance to Python best practices. Together, these tools give a holistic view of readability, maintainability, and potential long-term costs beyond simple correctness.

Functional correctness was assessed by executing LLM-generated functions on AWS Lambda using a predefined set of test cases. Outputs were systematically compared against reference implementations to categorize failures. Functions that fail due to syntactic

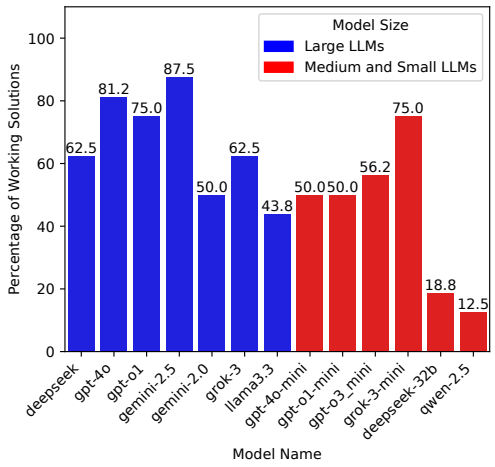


Figure 1: % of Functionally Correct Functions by LLM

errors were marked as SE (syntax error), while those that produce incorrect logical output were classified as FD (Functional Defect).

3.6 Performance Evaluation

We executed each generated function (from specific prompt-LLM combinations) 100 times under strictly controlled conditions. The runtime environment explicitly filtered executions to use only Intel Xeon Platinum 8259CL CPUs at 2.5 GHz, removing runtime variability from AWS Lambda's provisioning of CPUs [8]. Cold-start executions were also excluded to ensure that performance testing was also performed with serverless functions in the warm state, which has been shown to improve consistency.

Each function ran using a fixed AWS Lambda configuration with 3008 MB memory and 2 vCPUs configured with a maximum runtime of 5 minutes. For LLM generated functions with exceptionally slow serverless runtime (i.e. approximately 10 times slower than typical), we adopted smaller task configurations (e.g. generate 500 prime numbers instead of 1,000) to normalize function runtimes to approximately 5 seconds. Reducing the task size also enabled some functions to finish before the timeout value, which would otherwise have timed out.

4 Results and Evaluation

4.1 Generated Code Correctness

To answer RQ-1, we evaluated the syntactic and functional correctness of the serverless functions generated by each LLM for 16 distinct prompts. Functions that failed due to syntax errors were classified as 'SE' (Syntax Error) and functions with logical errors were classified as FD (Functional Defect). The code for these functions is syntactically correct, and the code can be run, but the code produces incorrect results. Functions were then benchmarked by executing each function 100 times under controlled conditions. We report results for running functions in the warm state using only the Intel Xeon Platinum 8259CL CPU to ensure consistent results. For consistency, we removed profiling data for functions that ran on other CPUs on AWS Lambda from our data.

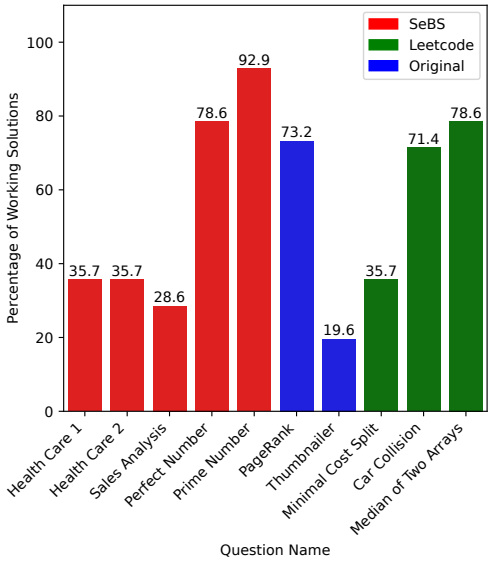


Figure 2: % of Functionally Correct Functions by Prompt

Our evaluation shows that large LLMs, including GPT-4o and Gemini-2.5, produced more functionally correct implementations for a broad range of coding tasks. In contrast, small LLMs such as DeepSeek-8b and Qwen-2.5 struggled, especially when encountering complex prompts, resulting in higher rates of syntax and logic errors. As shown in Figure 1, the best LLM achieved near perfect accuracy on all tasks, while mid-size and local LLMs exhibited variable outcomes. Certain models such as Grok-3-mini and GPT-o1-mini produced correct results on simpler prompts, but failed on prompts requiring algorithmic reasoning or non-standard data processing. Figure 2 further demonstrated that correctness varied, not only by model, but also by task complexity.

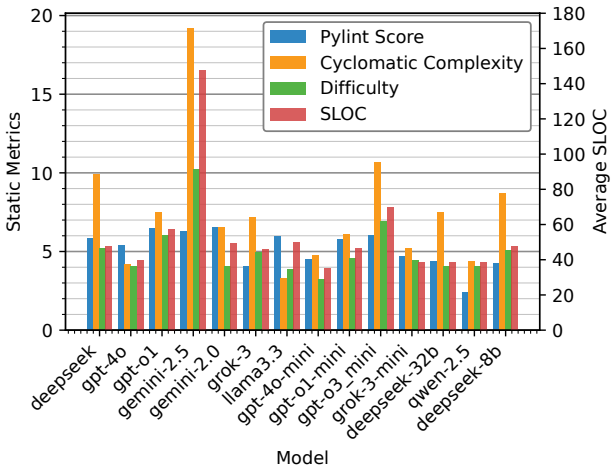


Figure 3: Static Analysis of Serverless Functions by LLM

Table 4: Runtime Percentage Difference Compared to Reference Function Code Implementation

Task	deepseek	gpt-4o	gpt-o1	gemini-2.5	gemini-2.0	grok-3	llama3.3	gpt-4o-mini	gpt-o1-mini	gpt-o3-mini	grok-3-mini	deepseek-32b	qwen-2.5
Health Care 1	SE	2.84*	**0.00***	20.36*	SE	**-39.21***	FD	FD	SE	SE	FD	**-15.77***	SE
Health Care 2	**0.00***	10.75*	SE	15.14*	SE	**-37.02***	FD	FD	FD	SE	**-9.85***	SE	FD
Sales Analysis	SE	FD	**0.00***	9.14*	SE	SE	FD	FD	5.26*	SE	**-5.35***	SE	SE
Perfect Number	**-0.62**	**0.00**	0.76	0.50	0.20	3.35	10215.34*	10345.47*	3.04	0.32	0.06	FD	FD
Prime Number	**-46.79**	417.96*	455.44*	**0.00**	**-2.14**	861.35*	1.86	820.99*	4.89	5.09	0.23	815.11*	5750.05*
PageRank Long	1126.45	75612.85*	618.63	**466.22**	FD	5768.82*	FD	10552.53*	SE	1290.66	326882.06*	FD	FD
PageRank Short	21.39*	**0.00**	51.61	-0.27*	51.08*	22.37*	24.48*	**-0.77**	51.82*	264.90*	0.47*	21.07*	FD
PageRank Long Nolib	3606.50*	491.38	32077.09*	3603.06*	6083.39*	1771.68*	**-84.37***	16776.33*	SE	1716.38	67083.12*	FD	93038.03*
PageRank Short Nolib	613.08*	671.25*	773.52*	732.16*	115124.79*	9895.98	1578.18*	FD	20220.56*	**246.22***	76997.49*	FD	FD
Thumbnailer Long	FD	FD	**0.17***	FD	FD	SE	SE	FD	FD	SE	SE	FD	FD
Thumbnailer Short	FD	FD	FD	FD	FD	SE	FD	FD	FD	FD	FD	FD	FD
Thumbnailer Long Nolib	SE	12.77*	40.45*	**0.00***	SE	SE	FD	25.65*	-8.33*	SE	**-9.48***	FD	SE
Thumbnailer Short Nolib	**43.57***	78.26*	FD	285.58*	FD	FD	FD	FD	FD	FD	75.61*	SE	FD
Minimal Cost Split	SE	21072.83	FD	**0.00**	49650.71*	SE	FD	FD	FD	15506.58	26055.37	FD	SE
Car Collision	619.15*	399.62*	437.23*	676.97*	**0.00***	65.92*	420.24*	**-2.49***	634.29*	469.38*	SE	SE	FD
Median of Two Arrays	29.56	48.48	**0.00**	35.14	53.28	49.76	113.17	46.88	13846.91*	**-7.35**	45.89	FD	FD
AVG Runtime Diff	601.23	7601.46	2871.24	417.43	21370.16	1836.30	1752.70	4820.57	4344.81	2165.80	41426.30	273.47	49394.04

*: Function does not work in a reasonable time limit or failed in an extended test, the comparison is based on a smaller test data.
SE (Syntax Error): Functions that have syntax errors and could not been executed, FD (Functional defect): Functions producing incorrect logical outputs.
deepseek-8b is not shown because it produced no functionally correct code. Best performing LLM results for each task are shown in **bold**.

For each prompt–LLM pair, we report three static metrics: Complexity, the cyclomatic complexity of generated functions/methods from radon (lower is better); Difficulty, the Halstead difficulty measure by radon estimating effort to write/understand code (lower is better); and Score, the Pylint global evaluation on a 0–10 scale (higher is better). Metric averages were calculated for all tasks to compare relative trends for small vs. large models, local vs. cloud execution, and reasoning vs. traditional inference styles. Results are visualized in Figure 3. This figure demonstrates that larger cloud-hosted models (e.g., GPT-4o, Gemini-2.5) generally produce code with higher Pylint scores—averaging 5.80 compared to 4.58 for smaller models, indicating cleaner, more maintainable implementations. A two-sample t-test confirmed that this difference is statistically significant at the 5% level (t = 2.20, p = 0.0503). In contrast, smaller and mid-scale local models tended to generate more verbose or inconsistent structures, reflected in their lower linting scores and reduced maintainability.

These static analysis results in Figure 3 align with our runtime findings: models that generate simpler, more idiomatic code often achieve lower execution variance and fewer functional defects. However, static analysis also highlights trade-offs not visible at runtime: some reasoning-oriented models produced correct and efficient code but with reduced maintainability, suggesting that reasoning depth does not guarantee stylistic clarity.

Smaller models were, on average, 3.23 × slower than larger cloud-hosted models when executing their generated code. Per-prompt analysis in Figure 4 and Figure 6 revealed that runtime disparities were highly task-dependent. Compute-intensive prompts such as Perfect Number, Prime Number, and PageRank magnified these gaps, with smaller models running up to 6.39 times slower. In contrast, I/O-bound tasks like Healthcare 1/2 and Sales Analysis exhibited minimal differences (0.95x), as CSV parsing and aggregation dominated total runtime regardless of code quality. Thumbnailer exhibited another edge case: some models generated inefficient decode–resize loops, leading to extreme slowdowns. These results underscore that LLM performance must be considered jointly with task requirements, as the task mediates whether code differences translate into observable runtime effects.

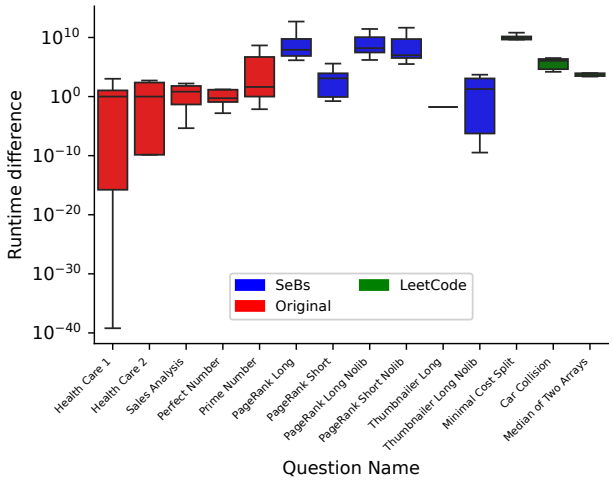


Figure 4: Runtime % Difference by Question Compared to Reference Implementation (log scale)

4.2 Function Performance and Execution Cost

To investigate RQ-2, we evaluated the cost of executing LLM generated serverless function code on AWS Lambda using AWS Lambda's billing model (e.g. $0.00001667 per GB/sec with x86_64). We compared hosting costs for code generated by 14 different LLMs for all of our code generation prompts shown in table 3. Figure 7 shows the hypothetical execution cost for running each LLM generated function 1,000,000 times using 3008 MB memory on AWS Lambda for our pagerank and prime number function generation prompts. We found that larger LLMs provided greater cost savings than smaller LLMs. When comparing the least expensive function generated by a large vs. small/medium LLM, for pagerank we observed a 30.9% savings, and for prime number generation, a 59.4% savings.

In terms of performance, larger LLMs generated serverless functions with consistently better runtime performance, as reflected in their shorter average execution times across multiple prompt

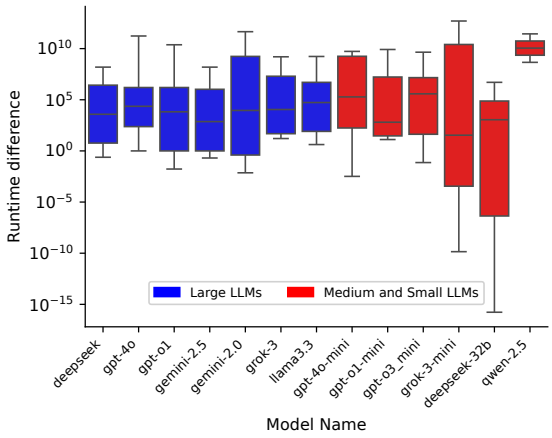


Figure 5: Runtime % Difference by LLM vs. Reference (log scale)

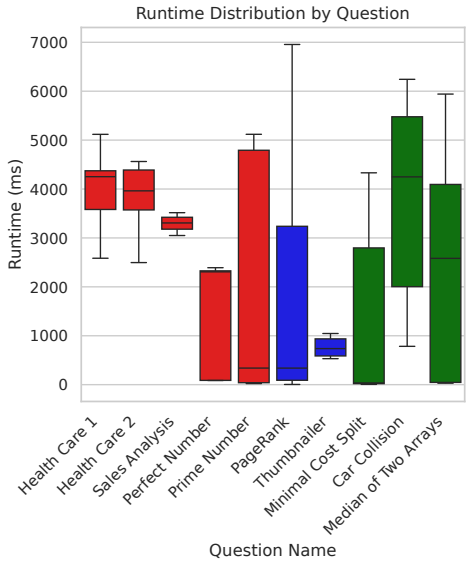


Figure 6: Runtime Distribution by Question

scenarios. This is supported by Table 4, which shows that the functions generated by GPT-4o and Gemini-2.5 often approached or had lower runtime when compared to the function reference implementations. In contrast, some small and mid-size LLMs produced code that was hundreds to thousands of percent slower than the reference implementations in Figure 5, in several cases exceeding AWS Lambda's maximum runtime limit.

These results show the importance of choosing an appropriately sized LLM based on task complexity to balance LLM capability, correctness, and generated code execution efficiency in serverless computing environments. Although smaller LLMs may offer lower inference costs, function correctness and slower runtime could lead to increased debugging time, higher costs, or code quality problems.

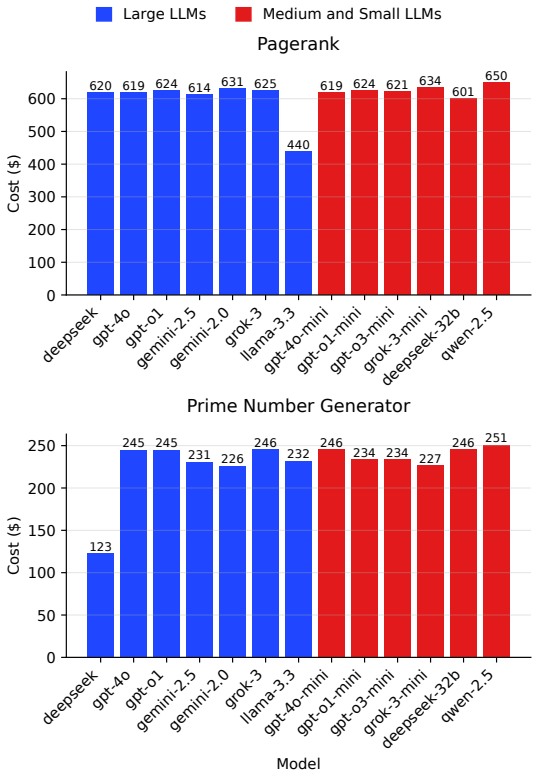


Figure 7: Cost of 1m Executions for Each Model and Question

5 Conclusion

This study examined LLMs for serverless code generation across the dimensions of correctness, code quality, cost, and performance. We found that larger cloud-hosted models produced more correct and maintainable functions than smaller models, which struggled with complex tasks **(RQ-1)**. Benchmarking on AWS Lambda showed that larger models also generated more efficient and cost-effective code, while code from smaller models incurred runtime and cost penalties **(RQ-2)**. Overall, our findings highlight the importance of evaluating LLMs not only for correctness, but also for their operational impact on serverless software development.

References

[1] Alibaba. 2024. Qwen2.5-coder-7b-instruct. https://huggingface.co/Qwen/Qwen2.5-Coder-7B-Instruct. Accessed: 2025-04-16. (2024).
[2] Amazon Web Services. 2025. Aws lambda documentation. https://docs.aws.amazon.com/lambda/. Accessed: 2025-05-05. (2025).
[3] Anysphere Inc. 2025. Cursor: the ai code editor. Accessed: 2025-05-05. (2025). https://www.cursor.com/.
[4] Shrikara Arun, Meghana Tedla, and Karthik Vaidhyanathan. 2025. LLMs for Generation of Architectural Components: An Exploratory Empirical Study in the Serverless World. In *2025 IEEE 22nd Intl. Conf. on Software Architecture (ICSA)*, 25–36.
[5] Dave Bergmann. 2025. What is a reasoning model? https://www.ibm.com/think/topics/reasoning-model. Accessed: 2025-10-10. IBM, (2025).
[6] Zinuo Cai and et al. 2024. LLMaaS: Serving Large Language Models on Trusted Serverless Computing Platforms. *IEEE Transactions on Artificial Intelligence*.
[7] Liguo Chen and et al. 2024. A survey on evaluating large language models in code generation tasks. *arXiv preprint arXiv:2408.16498*.

[8] Xinghan Chen and et al. 2024. Predicting ARM64 Serverless Functions Runtime: Leveraging function profiling for generalized performance models. In *2024 IEEE/ACM 17th Intl. Conf. on Utility and Cloud Computing (UCC)*, 63–72.

[9] Tristan Coignion, Clément Quinton, and Romain Rouvoy. 2024. A performance study of llm-generated code on leetcode. In *Proceedings of the 28th Intl. Conf. on Evaluation and Assessment in Software Engineering*, 79–89.

[10] Robert Cordingly and et al. 2021. Enhancing observability of serverless computing with the serverless application analytics framework. In *Companion of the ACM/SPEC Int. Conf. on Perf. Engineering*, 161–164.

[11] Robert Cordingly, Wen Shu, and Wes J Lloyd. 2020. Predicting performance and cost of serverless computing functions with SAAF. In *2020 IEEE Intl Conf on Cloud and Big Data Computing*, 640–649.

[12] DeepSeek. 2024. Deepseek reasoner r1 series: 8b, 32b, 671b. https://huggingface.co/deepseek-ai/DeepSeek-R1. Accessed: 2025-04-16. (2024).

[13] Mingzhe Du and et al. 2024. Mercury: A code efficiency benchmark for code large language models. *arXiv preprint arXiv:2402.07844*.

[14] Xueying Du and et al. 2024. Evaluating large language models in class-level code generation. In *Proceedings of the IEEE/ACM 46th Intl. Conf. on Software Engineering*, 1–13.

[15] Yao Fu and et al. 2024. ServerlessLLM: Low-Latency Serverless Inference for Large Language Models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 135–153.

[16] Google. 2024. Gemini 2.0 flash and gemini 2.5 series. https://blog.google/technology/google-deepmind/. Accessed: 2025-04-16. (2024).

[17] Michael Hassid and et al. 2024. The larger the better? improved llm code-generation via budget reallocation. *arXiv preprint arXiv:2404.00725*.

[18] Dong Huang and et al. 2024. Effibench: Benchmarking the efficiency of automatically generated code. *Advances in Neural Information Processing Systems*, 37, 11506–11544.

[19] Thomas Kluyver et al. 2016. Jupyter Notebooks–a publishing format for reproducible computational workflows. In *Positioning and power in academic publishing: Players, agents and agendas*. IOS press, 87–90.

[20] LeetCode. 2023. Count collisions on a road. https://leetcode.com/problems/count-collisions-on-a-road/. Accessed: 2025-04-16. (2023).

[21] LeetCode. 2023. Median of two sorted arrays. https://leetcode.com/problems/median-of-two-sorted-arrays/. Accessed: 2025-04-16. (2023).

[22] LeetCode. 2023. Minimum cost to split an array. https://leetcode.com/problems/minimum-cost-to-split-an-array/. Accessed: 2025-04-16. (2023).

[23] Yujia Li and et al. 2022. Competition-level code generation with alphacode. *Science*.

[24] Jiawei Liu and et al. 2024. Evaluating language models for efficient code generation. *arXiv preprint arXiv:2408.06450*.

[25] Meta. 2023. Llama 3.3 70b instruct. https://huggingface.co/meta-llama/Llama-3.3-70B-Instruct. Accessed: 2025-04-16. (2023).

[26] Ollama Inc. 2024. Ollama: run large language models locally. https://ollama.com. Accessed: 2025-05-05. (2024).

[27] OpenAI. 2023. Gpt-4o and gpt-4o mini. https://openai.com/index/hello-gpt-4o/. Accessed: 2025-04-16. (2023).

[28] OpenAI. 2023. Introducing openai o1 and o1 mini. https://openai.com/o1/. Accessed: 2025-04-16. (2023).

[29] OpenAI. 2023. Introducing openai o3 and o4 mini. https://openai.com/index/introducing-o3-and-o4-mini/. Accessed: 2025-04-16. (2023).

[30] OpenAI. 2025. Openai python api. https://github.com/openai/openai-python. Accessed: 2025-05-05. (2025).

[31] OpenAI. [n. d.] Tokenizer – openai api. https://platform.openai.com/tokenizer. Accessed: 2025-09-04. ().

[32] ollama organization. 2025. Ollama/ollama: get up and running with openai gpt-oss, deepseek-r1, gemma 3 and other models. https://github.com/ollama/ollama. Accessed: 2025-10-10; latest release version 0.12.3. (2025).

[33] Yun Peng and et al. 2024. PerfCodeGen: Improving Performance of LLM Generated Code with Execution Feedback. *arXiv preprint arXiv:2412.03578*.

[34] Joel Scheuner and Philipp Leitner. 2020. Function-as-a-service performance evaluation: a multivocal literature review. *Journal of Systems and Software*, 170, 110708.

[35] SeBS Project. 2020. Serverless benchmark suite (sebs). https://github.com/parasj/sebs. Accessed: 2025-04-16. (2020).

[36] Li Wang, Yankai Jiang, and Ningfang Mi. 2024. Advancing serverless computing for scalable ai model inference: Challenges and opportunities. In *Proceedings of the 10th Intl. Workshop on Serverless Computing*, 1–6.

[37] Jinfeng Wen and et al. 2025. LLM-Based Misconfiguration Detection for AWS Serverless Computing. *ACM Transactions on Software Engineering and Methodology*.

[38] xAI. 2024. Grok-3 and grok-3 mini beta. https://x.ai/news/grok-3. Accessed: 2025-04-16. (2024).