Optimizing Cloudlets for Faster Feedback in LLM-Based Code-Evaluation Systems

Daniel-Florin Dosaru *, Alexandru-Corneliu Olteanu, Nicolae Țăpuș

Posted Date: 24 November 2025

doi: 10.20944/preprints202511.1744.v1

Keywords: CloudLet; automated code evaluation; OnlineJudge; GenAI

 doi:10.20944/preprints202511.1744.v1

Article

Optimizing Cloudlets for Faster Feedback in LLM-Based Code-Evaluation Systems

Daniel-Florin Dosaru *, Alexandru-Corneliu Olteanu and Nicolae Țăpuș

National University of Science and Technology Politehnica Bucharest, Romania
* Correspondence: daniel.dosaru@upb.ro

Abstract

This paper addresses the challenge of optimizing cloudlet resource allocation in a code evaluation system. The study models the relationship between system load and response time when users submit code to an an online code evaluation platform called LambdaChecker that operates a cloudlet-based processing pipeline. The pipeline includes code correctness checks, static analysis, and design-pattern detection using a local Large Language Model (LLM). To optimize the system we develop a mathematical model and apply it to LambdaChecker resource management. The proposed approach is assessed using both simulations and real contest data, focusing on improvements in average response time, resource-utilization efficiency, and user satisfaction. The results indicate that adaptive scheduling and workload prediction effectively reduce waiting times without substantially increasing operational costs. Overall, the study suggests that systematic cloudlet optimization can enhance the educational value of automated code evaluation systems by improving responsiveness while preserving sustainable resource usage.

Keywords: CloudLet; automated code evaluation; OnlineJudge; GenAI

1. Introduction

Writing maintainable, high-quality code and receiving prompt, clear feedback significantly enhance learning outcomes [1] [2] in computer science education. As students develop programming skills, rapid, reliable evaluation helps reinforce good practices, correct misconceptions early, and maintain engagement—especially in campus-wide online judge. Similarly, in the software industry, timely code review and automated testing play a crucial role in ensuring code quality, reducing defects, and accelerating development cycles, highlighting the importance of feedback-driven workflows both in education and professional practice.

This paper presents optimization strategies for managing cloudlet resources of LambdaChecker, an online code evaluation system developed at the National University of Science and Technology Politehnica Bucharest. The system was designed to enhance both teaching and learning in programming-focused courses such as Data Structures and Algorithms and Object-Oriented Programming. It does this by supporting hands-on activities, laboratory sessions, examinations, but also during a programming contests. A key advantage of LambdaChecker lies in its extensibility, allowing for the integration of customized features tailored to specific exam formats and project requirements. Moreover, the platform facilitates the collection of coding-related metrics, such as the detection of pasted code fragments in submissions, job timestamps, and the average time required to solve a task.

In its current implementation, LambdaChecker assesses code correctness through input/output testing and evaluates code quality [3] using PMD [4] - open-source static code analysis tool. PMD, which is also integrated into widely used platforms such as SonarQube, provides a set of carefully selected object-oriented programming metrics. These metrics are sufficiently simple to be applied effectively in an educational context, while still maintaining alignment with industry standards.

 doi:10.20944/preprints202511.1744.v1

In our previous work [7], building on earlier studies [5,6], we extended LambdaChecker to include support for design pattern detection using Generative AI (GenAI). We have used LLaMA 3.1 model (70.6B parameters, Q4_0 quantization) to detect design patterns in the submitted user's code. This functionality has proven particularly valuable for assistant professors during the grading process, as it facilitates the assessment of higher-level software design principles beyond code correctness and quality. This integration leverages GenAI models to enhance the reliability of automated feedback while reducing the grading workload. Nonetheless, this advancement is associated with higher resource demands and prolonged latency in code evaluation systems that lack optimization.

During a contest, the workload on our system differs significantly from regular semester activity, exhibiting a sharp increase in submission rates. While day-to-day usage remains relatively irregular and influenced by teaching assistants' instructional styles, contents periods generate highly concentrated bursts of activity. The submission rate rises steadily as users' progress with their tasks, culminating in peak workloads (such as the one represented in Figure 1 of 55 submissions per minute) during the contest. Such intense demand places substantial pressure on computational and storage resources, making it essential to optimize resource allocation. Without efficient resource management, the platform risks performance degradation or downtime precisely when reliability is most critical. Consequently, adapting the system's infrastructure to dynamically match any workload patterns ensures both cost-effectiveness during low-activity periods and robustness during high-stakes contest scenarios.

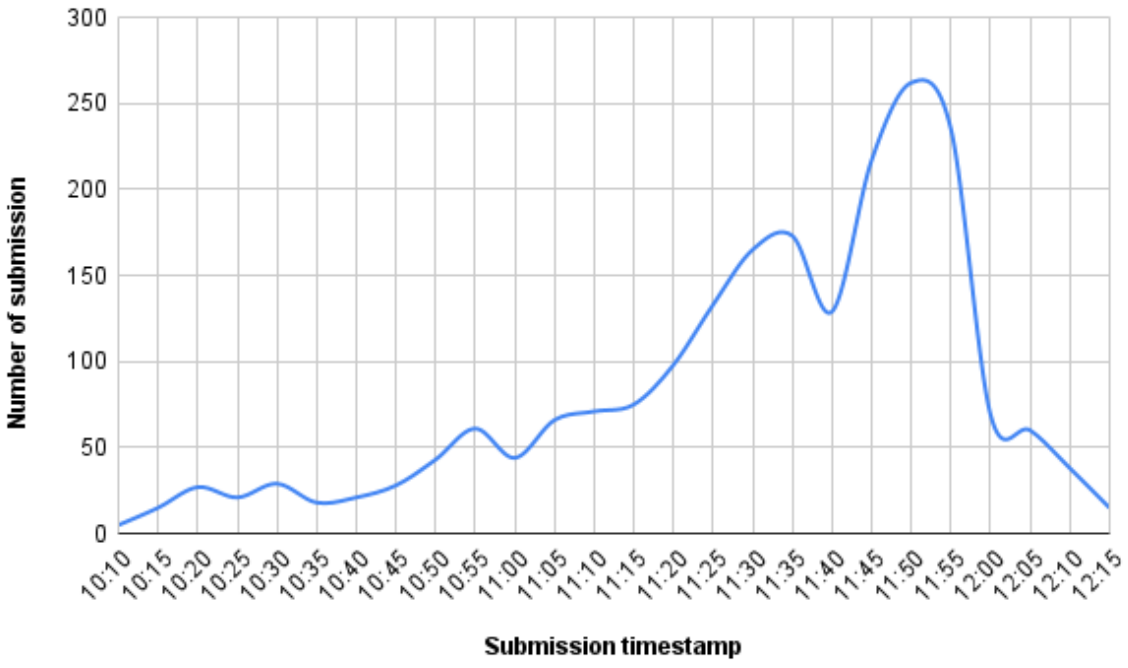

Figure 1. Frequency of submissions in 5-minute intervals during the OOP practical examination

The paper proceeds as follows: section 2 reviews the related work in the field, while section 3 describes the cluster, the LambdaChecker system and the mathematical model and optimization techniques. Section 4 applies the mathematical model in a case study - LambdaChecker during a contest. Section 5 explores the discussion and user experience implications, and section 6 concludes the paper.

2. Related work

Automated code evaluation systems, commonly known as online judges (OJs), play a central role in both competitive programming and computer science education. Early research demonstrated the

 doi:10.20944/preprints202511.1744.v1

feasibility of cloud-based architectures for scalable OJ platforms, enabling simultaneous evaluation of many users submissions [8]. Subsequent work leveraged container-based sandboxing to improve isolation and resource efficiency, allowing platforms to handle the bursty workloads typical of exams and practical assignments [9]. Surveys of automated feedback systems in education further emphasize the importance of combining correctness checking, static analysis, and pedagogically meaningful feedback to enhance learning outcomes [10,11]. These studies establish the foundational need for efficient, responsive evaluation pipelines, particularly in time-constrained assessment scenarios.

Cloudlet and edge computing approaches have been proposed to improve responsiveness and resource utilization in latency-sensitive applications. Task offloading and collaborative edge–cloud scheduling strategies demonstrate that local compute layers can significantly reduce response times while balancing load across distributed infrastructure [12]. Surveys of edge networks further highlight how the convergence of computation, caching, and communication improves performance under highly variable workloads [13]. To handle fluctuating demand, both classical and modern auto-scaling strategies have been studied, ranging from deadline- and budget-constrained cloud auto-scaling [14] to reinforcement learning–based adaptive scheduling [15]. These insights inform the design of short-, medium-, and long-term scheduling strategies for cloudlet-based code evaluation systems, such as the one presented in this study.

Analytical performance modeling provides additional guidance for managing high-volume workloads in interactive and multi-tier systems. Foundational models for predicting response times and queue lengths in multi-tier applications illustrate how arrival rates, service rates, and resource contention interact to influence system performance [19]. Interference-aware models further capture the effects of shared resource contention, which is critical when multiple submissions are processed concurrently in a virtualized environment [20]. Strategies for dynamic load balancing and quality-of-service-aware scheduling also provide mechanisms for handling bursty or time-sensitive workloads [21]. These studies support the use of queueing-based simulations and workload prediction as essential tools for optimizing cloudlet behavior in automated code evaluation pipelines.

Recent advances in large language models (LLMs) have enabled novel forms of automated code analysis and feedback. Generative AI models have been used to assess code quality, detect design patterns, and provide higher-level feedback beyond simple correctness checks [5]. Evaluations of code-oriented LLMs demonstrate their ability to classify, analyze, and predict defects across various programming languages and tasks [6]. Integrating a local LLM within a cloudlet-based evaluation pipeline allows for richer analysis—such as design-pattern detection—without sacrificing latency or data privacy, complementing traditional static analysis tools like PMD.

Finally, the pedagogical benefits of rapid feedback are well-documented. Timely, high-quality feedback significantly enhances learning outcomes by supporting iterative problem solving and reinforcing correct practices [23,24]. In computing education, shorter feedback cycles have been shown to improve student performance, reduce frustration, and increase engagement [22]. These findings underscore the educational relevance of optimizing response times in online code evaluation systems: beyond technical efficiency, faster and more consistent feedback directly contributes to better learning outcomes and improved user experience.

Govea et al. [17] explore how artificial intelligence and cloud computing can be integrated to improve the scalability and performance of educational platforms. Their work highlights strategies for efficient management of computational resources under high user loads, providing insights that are directly relevant for optimizing cloud utilization in student code evaluation systems. However, their approach is more general, targeting educational platforms broadly rather than focusing specifically on code evaluation. As a result, it lacks tailored mechanisms for handling programming assignments, such as automated testing, code similarity detection, or fine-grained feedback, which are central to dedicated student code evaluation systems.

Kim et al. [18] present Watcher, a cloud-based platform for monitoring and evaluating student programming activities. The system emphasizes isolation and convenience, providing each student with

 doi:10.20944/preprints202511.1744.v1

an independent virtual machine to prevent interference and ensure secure execution of assignments. By capturing detailed coding activity logs and enabling instructors to analyze potential cheating or errors, Watcher demonstrates how cloud-based infrastructure can support fair, scalable, and secure student code evaluation. Nevertheless, relying on dedicated virtual machines for each student can lead to higher resource usage and operational costs, potentially constraining scalability in large classes or institutions with limited cloud budgets. Furthermore, this system does not account for the substantial resource demands associated with LLM-based evaluation.

Dumitru et al. [16] present a queueing-network inspired model for scheduling "bags-of-tasks" in cloud environments by employing mean-value analysis to estimate makespan and cost under varying configurations. The mathematical modelling is useful for our problem because it provides a strong foundation in queueing theory and allows us to consider metrics like arrival rates, service rates, resource utilisation, and system throughput (for example via Little's Law). However, a key difference is that their tasks are independent and may be executed without ordering requirements, whereas in our code evaluation system we must respect the sequential submission and evaluation order of each user's jobs — thus our queueing model must account for ordering-constraints and possibly priority/precedence relations, which their model does not address.

While each of these studies provides valuable insights, none fully addresses the combined challenges of sequential job evaluation, fine-grained feedback, and cost-effective cloud utilization in the context of real-time LLM-Based code evaluation systems. Our work builds on these foundations by developing a tailored mathematical model for load and response time, specifically designed for user code evaluation estimated workload, and by analyzing optimization strategies that balance performance, cost, and user performance outcomes. This positions our approach as a bridge between theoretical cloud optimization, practical monitoring systems, and pedagogically meaningful evaluation.

3. Materials and Methods

3.1. Cluster Setup, LLM-based code evaluation tool

Owing to limited computational resources, we initially assigned all submission-processing tasks for the LambdaChecker scheduler to a dedicated compute node. This node is equipped with Intel® Xeon® E5-2640 v3 multicore processor and NVIDIA DGX H100 GPUs with 80 GB of VRAM, providing substantial memory capacity and strong computational power. This setup is ideal for large-scale models; for instance, a pair of these GPUs can efficiently run a 70B-parameter model like LLaMA 3.1, maintaining high performance and optimized memory usage.

The machine supports the entire evaluation pipeline, handling CPU-intensive tasks such as PMD static analysis and correctness testing, as well as GPU-accelerated design-pattern detection workloads. Each step increases the load on the evaluation system, with LLM-based design pattern detection being the most resource-intensive, consuming up to 70% of the total average evaluation time.

A cloudlet-based LLM offers three main advantages over a traditional cloud deployment: (1) *lower latency*, since computation occurs closer to the user and enables faster, more interactive code analysis; (2) *improved data privacy*, as sensitive source code remains within a local or near-edge environment rather than being transmitted to distant cloud servers; and (3) *reduced operational costs*, because frequent analysis tasks avoid cloud usage fees and bandwidth charges while still benefiting from strong computational resources.

User-submitted code is incorporated into a pre-defined improved prompt, tailored to the specific coding problem, to facilitate detection of particular design patterns. The large language model then performs inference on a dedicated GPU queue supported by the NVIDIA H100.

The prompt was refined to be clearer, more structured, and easier for a generative AI model to follow. It explicitly defines the JSON output, clarifies how to report missing patterns, and instructs the model to focus on real structural cues rather than class names. The rule mapping publisher–subscriber to the Observer pattern ensures consistent terminology. Overall, the prompt improves reliability and reduces ambiguity in pattern detection.

 doi:10.20944/preprints202511.1744.v1

The prompt is concise, focuses only on essential instructions, and specifies a strict JSON output format, allowing the cloudlet-hosted LLM to process the code more efficiently without unnecessary reasoning or verbose explanations.

"You analyze source code to identify design patterns. Output ONLY this JSON:

```
{
"design_patterns": [
  {
    "pattern": "<pattern_name>",
    "confidence": <0-100>,
    "adherence": <0-100>
  }
]
}
```

If no patterns are found, return {"design_patterns": []}."

The system dynamically scales across multiple computing nodes in response to the number of active submissions. It continuously monitors the average service rate, which depends on the current submissions workload, and adjusts resources in a feedback-driven loop, scaling nodes up or down as required to ensure efficient processing. The scaling mechanism is based on a mathematical model described in the following subsection.

3.2. Mathematical Model

We consider a generic queueing system and introduce the following notation:

- $L(t)$: average queue length at time t;
- $\lambda(t)$: arrival rate at time t (jobs per unit time);
- μ: service rate per machine (jobs processed per unit time);
- m: number of parallel processing machines (queues).

Even if the exact inter-arrival and service-time distributions are unknown, empirical traces in real systems typically exhibit diurnal variability, burstiness near deadlines, and transient queue-drain phases. A useful analytical approximation is to treat the system as an *M/M/m* queue with effective rates

$$\mu_{\text{eff}} = m\mu, \qquad \lambda_{\text{eff}}(t) \leq \lambda(t), \tag{1}$$

leading to an approximate expected queue length

$$L(t) \approx \frac{\lambda_{\text{eff}}(t)}{\mu_{\text{eff}} - \lambda_{\text{eff}}(t)}. \tag{2}$$

This approximation is valid whenever the stability condition

$$\lambda_{\text{eff}}(t) < \mu_{\text{eff}} \tag{3}$$

is satisfied; otherwise, the system becomes overloaded and the queue diverges.

In addition if $\lambda > \mu$, no steady-state distribution exists. A fluid approximation therefore provides insight:

$$\frac{dL}{dt} = \lambda - \mu, \qquad L(t) = (\lambda - \mu)t.$$

If arrivals are indexed deterministically so that the n-th submission arrives at

$$t_n \approx \frac{n}{\lambda},$$

 doi:10.20944/preprints202511.1744.v1

then the approximate waiting time experienced by job n-th is

$$W_n \approx \frac{L(t_n)}{\mu} = \frac{\lambda - \mu}{\lambda \mu} n. \tag{4}$$

4. Results

4.1. Case Study: LambdaChecker in a Programming Contest Increasing the Service Rate

We begin by analyzing a highly loaded contest scenario through a hypothetical M/M/1 queueing model.

We take these values from our dataset:

$$\lambda = 55 \text{ jobs/min}, \qquad \mu = 5 \text{ jobs/min}.$$

The traffic intensity is

$$\rho = \frac{\lambda}{\mu} = \frac{55}{5} = 11,$$

indicating severe overload and unbounded queue growth.

If we consider a single queue sistem we can compute for the final submission with $n = 2119$ its waiting time:

$$\begin{aligned} \lambda - \mu &= 50, \\ \lambda \mu &= 275, \\ \frac{\lambda - \mu}{\lambda \mu} &= \frac{50}{275} \approx 0.1818, \\ W_{2119} &\approx 0.1818 \cdot 2119 \approx 385.2 \text{ minutes}. \end{aligned}$$

Thus, the 2119-th job waits roughly six and a half hours before service begins.

4.2. Determining the Required Number of Machines

To prevent unbounded queue growth during peak contest activity, we model LambdaChecker as an M/M/c system with c identical processing nodes.

Peak Load Parameters

$$\lambda = 55 \text{ jobs/min}, \qquad \mu = 5 \text{ jobs/min per machine}.$$

Total Service Capacity

$$\mu_{\text{total}} = c\mu.$$

Stability Condition

$$\rho = \frac{\lambda}{\mu_{\text{total}}} \leq 1 \quad \Longrightarrow \quad \frac{\lambda}{c\mu} \leq 1.$$

Minimum Machines Required

$$c \geq \frac{\lambda}{\mu} = \frac{55}{5} = 11.$$

Thus, at least 11 machines are needed to sustain the peak arrival rate without unbounded queueing. However, $c = 11$ puts the system at critical load ($\rho = 1$), meaning small fluctuations can still cause delays. In practice, provisioning at least 12 machines yields significantly more stable performance.

 doi:10.20944/preprints202511.1744.v1

By constraining the number of jobs permitted per user and discarding any excess submissions, the system's effective load is reduced, thereby decreasing the required number of nodes. This represents just one specific instance of a broader class of resource-management strategies commonly employed to limit system load. Such methods, while effective in stabilizing resource usage, may lead to discarded submissions; thus, an appropriate balance between system robustness and user experience must be carefully maintained.

5. Discussion

We evaluated the impact of two strategies on system performance: increasing the number of processing nodes and applying queue policies to limit submissions per user. To validate and explore the queuing behavior empirically, we implemented a discrete-event simulation in Python. The script loads submission data, including user_id, arrival timestamps and evaluation durations, and then simulates processing across multiple queues. Each submission is assigned to the queue that becomes available first, and the script tracks individual waiting times and completion times. This allows us to compute statistics such as per-submission wait times and distributions of delays for different numbers of queues. The results are visualized in Figure 2 and 3 - in plots of wait times per submission providing a clear picture of how increasing the number of queues improves performance.

5.1. Effect of Scaling Machines

Figure 2 illustrates the expected waiting time $W(t)$ for different numbers of machines (processing queues). Consistent with M/M/m queueing theory, increasing the number of nodes reduces the waiting time, particularly during peak submission periods. For the original single-server setup, the traffic intensity $\rho \gg 1$, resulting in extremely long queues—our calculations show that the last submissions could experience waiting times exceeding 6 hours. Doubling or appropriately scaling the number of machines significantly reduces $W(t)$ and stabilizes the queue. While effective, this approach requires additional infrastructure, highlighting the trade-off between system performance and operational costs.

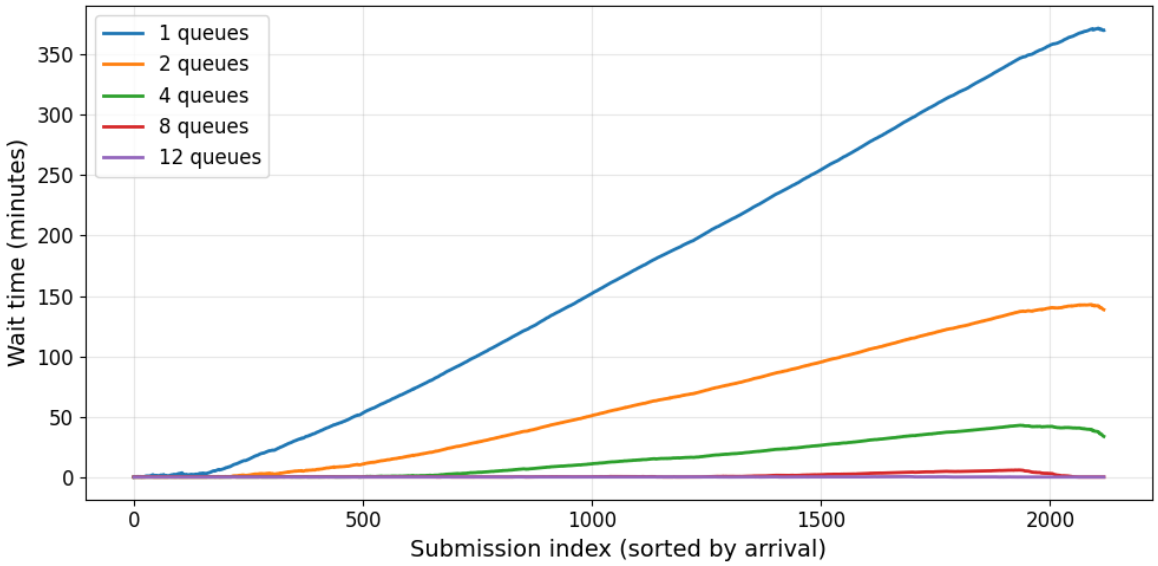

Figure 2. Per-submission wait times across varying numbers of queues, illustrating the impact of queue availability on processing delays. We prove that for our case study using 12 queues makes the system stable

In Figure 3, we include the simulation for 12 queues to verify the accuracy of our previous computations. As shown, among the 2,119 submissions, the one with the longest evaluation time took 20 seconds, and with 12 queues, the user waiting time remains below 25 seconds. Additionally, we

 doi:10.20944/preprints202511.1744.v1

included the same submission frequency from Figure 1 over the duration of the contest to visualize the submission workload on LambdaChecker during a contest with 200 participants.

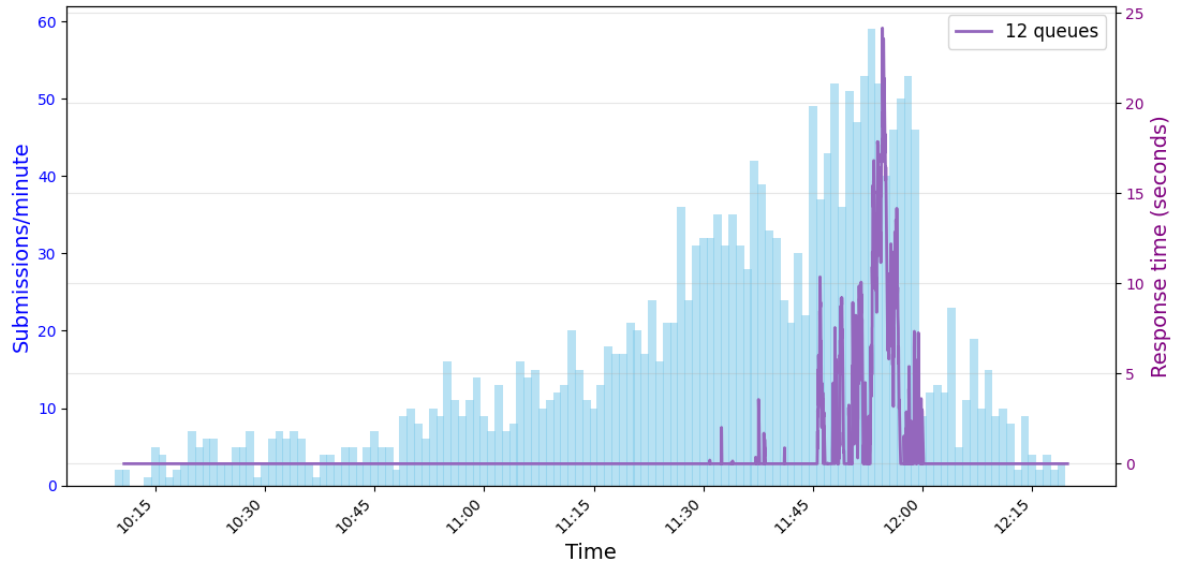

Figure 3. Submission frequency and wait times with a fixed number of 12 queues

5.2. Effect of Queue Policies

By limiting the number of jobs per user and discarding excess submissions, the system reduces load and lowers the number of required nodes. This is an example of common resource-management strategies that stabilize usage but may discard submissions, requiring a careful balance between system robustness and user experience.

5.3. Contest Insights from Code Evaluation system

Several insights arise from our analysis. First, limiting the effective arrival rate is especially impactful when the system is heavily loaded. Due to the non-linear relationship between waiting time and traffic intensity, small reductions in λ_{eff} can lead to disproportionately large decreases in $W(t)$. Second, adding new processing queues improves performance but increases operational costs, whereas queue policies provide significant benefits at minimal expense. Finally, controlling submissions not only reduces average waiting times but also stabilizes the queue, leading to more predictable processing times for users.

Beyond system metrics, these strategies have a direct impact on user experience and performance. When feedback was delayed or unavailable during the contents, participants experienced uncertainty and frustration, often leading to lower engagement and scores. In the initial run, some students waited up to five minutes to receive feedback. In contrast, when feedback was returned more quickly (e.g. in under 25 seconds), users could identify and correct errors promptly, boosting both learning and confidence. Our data show that with faster feedback, the average score increased from 53 to 69 out of 100. This demonstrates that system-level optimizations not only improve operational efficiency but also meaningfully enhance users outcomes and engagement.

5.4. Limitations

The M/M/m approximation assumes exponential inter-arrival and service times, which may not fully capture the real-world behavior of submissions. Variability in machine performance and user submission patterns may affect the observed service rate μ, and queue-limiting policies could influence user behavior in ways not captured by this static analysis. Despite these limitations, our combined approach demonstrates a practical and data-driven strategy to manage waiting times effectively, balancing system performance, stability, and operational cost.

 doi:10.20944/preprints202511.1744.v1

6. Conclusions

Our analysis shows that system performance during peak submission periods can be effectively managed through a combination of server scaling and queue control policies. Analytical calculations indicate that a single-server setup becomes severely overloaded under peak demand, leading to waiting times of several hours. Adding new processing queues reduces waiting times, though it incurs additional infrastructure costs.

Queue policies that limit the number of active submissions per user substantially lower the effective arrival rate, decreasing both the average waiting time and the variability of the queue. This approach provides significant performance improvements at minimal cost and helps stabilize the system, preventing extreme delays even during high submission rates.

The combination of server scaling and submission control yields the best results, maintaining manageable queues and near-optimal waiting times throughout peak periods. Timely feedback directly improves participant experience and performance. Delays create uncertainty and frustration, lowering engagement, while feedback within 25 seconds lets users quickly correct errors, boosting understanding, confidence, and contests results. We found that providing faster feedback boosted average scores by ≈30% on tasks of similar difficulty, highlighting how reducing wait times can improve performance outcomes. These findings demonstrate that system-level optimizations not only improve operational efficiency but also meaningfully enhance users outcomes and engagement, emphasizing the importance of balancing technical performance with participants satisfaction.

Funding: This work was supported by the National Program for Research of the National Association of Technical Universities under Grant GNAC ARUT 2023.

Institutional Review Board Statement: This study utilized fully anonymized data with no collection of personally identifiable information. Based on these characteristics, the study qualified for exemption from Institutional Review Board (IRB) review

Informed Consent Statement: Informed consent was obtained from all subjects involved in the study.

Data Availability Statement: Links to paper dataset and scripts at https://tinyurl.com/2z4tf4yc.

Acknowledgments: The authors would like to thank Emil Racec for coordinating the development of LambdaChecker.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Kuklick, L. When computers give feedback: The role of computer-based feedback and its effects on motivation and emotions. *IPN News* **2024**, *June 24*.
2. Venables, A.; Haywood, L. Programming students NEED instant feedback! *Proceedings of the Fifth Australasian Conference on Computing Education, Volume 20* **2003**, 267–272.
3. Dosaru, D.-F.; Simion, D.-M.; Ignat, A.-H.; Negreanu, L.-C.; Olteanu, A.-C. A Code Analysis Tool to Help Students in the Age of Generative AI. *European Conference on Technology Enhanced Learning* **2024**, 222–228.
4. PMD. An extensible cross-language static code analyzer. Available online: https://pmd.github.io/ (accessed November 2025).
5. Bavota, G.; Linares-Vásquez, M.; Poshyvanyk, D. Generative AI for Code Quality and Design Assessment: Opportunities and Challenges. *IEEE Software* **2022**, *39*(5), 17–24.
6. Chen, M.; Tworek, J.; Jun, H.; et al. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374* **2021**.
7. Dosaru, D.-F.; Simion, D.-M.; Ignat, A.-H.; Negreanu, L.-C.; Olteanu, A.-C. Using GenAI to Assess Design Patterns in Student Written Code. *IEEE Transactions on Learning Technologies* **2025**.
8. Lu, J.; Chen, Z.; Zhang, L.; Qian, Z. Design and Implementation of an Online Judge System Based on Cloud Computing. *IEEE 2nd International Conference on Cloud Computing and Big Data Analysis* **2017**, 36–40.
9. Singh, A.; Sharma, T. A Scalable Online Judge System Architecture Using Container-Based Sandboxing. *International Journal of Advanced Computer Science and Applications* **2019**, *10*(5), 245–252.

 doi:10.20944/preprints202511.1744.v1

10. Keuning, H.; Jeuring, J.; Heeren, B. A Systematic Literature Review of Automated Feedback Generation for Programming Exercises. *ACM Transactions on Computing Education (TOCE)* **2018**, *19*(1), 1–43.
11. Frolov, A.; Buliaiev, M.; Sandu, R. Automated Assessment of Programming Assignments: A Survey. *Informatics in Education* **2021**, *20*(4), 551–580.
12. Li, X.; Tang, W.; Yuan, Y.; Li, K. Dynamic Task Offloading and Resource Scheduling for Edge-Cloud Collaboration. *Future Generation Computer Systems* **2019**, 95, 522–533.
13. Wang, S.; Zhang, X.; Zhang, Y.; Wang, L.; Yang, J.; Wang, W. A Survey on Mobile Edge Networks: Convergence of Computing, Caching and Communications. *IEEE Access* **2020**, *8*, 197689–197709.
14. Mao, M.; Li, J.; Humphrey, M. Cloud Auto-Scaling with Deadline and Budget Constraints. *11th IEEE/ACM International Conference on Grid Computing* **2010**.
15. Zhang, Q.; He, Q.; Chen, W.; Chen, S.; Xiang, Y. Adaptive Autoscaling for Cloud Applications via Reinforcement Learning. *IEEE Transactions on Cloud Computing* **2021**, *9*(3), 1162–1176.
16. Dumitru, C.; Oprescu, A.-M.; Živković, M.; van der Mei, R.; Grosso, P.; de Laat, C. A queueing theory approach to pareto optimal bags-of-tasks scheduling on clouds. *European Conference on Parallel Processing* **2014**, 162–173.
17. Govea, J.; Ocampo Edye, E.; Revelo-Tapia, S.; Villegas-Ch, W. Optimization and scalability of educational platforms: Integration of artificial intelligence and cloud computing. *Computers* **2023**, *12*(11), 223.
18. Kim, Y.; Lee, K.; Park, H. Watcher: Cloud-based coding activity tracker for fair evaluation of programming assignments. *Sensors* **2022**, *22*(19), 7284.
19. Urgaonkar, R.; Pacifici, G.; Shenoy, P.; Spreitzer, M.; Tantawi, A. Analytic Modeling of Multi-Tier Internet Applications. *ACM Transactions on the Web* **2007**, *1*(1), 1–33.
20. Casale, G.; Serazzi, G.; Zhang, E. Interference-Aware Modeling and Prediction of Virtualized Applications. *Performance Evaluation* **2011**, *67*(11), 1128–1142.
21. Ghosh, R.; Gupta, M.; Aggarwal, N.; Puri, P. Dynamic Load Balancing and QoS in Cloud Computing. *International Journal of Computer Applications* **2010**, 167–174.
22. Bailey, D.; Jia, S. The Impact of Rapid Feedback Cycles on Student Performance in Computing Courses. *ACM SIGCSE Technical Symposium* **2021**, 1234–1240.
23. Hattie, J.; Timperley, H. The Power of Feedback. *Review of Educational Research* **2007**, *77*(1), 81–112.
24. Shute, V. Focus on Formative Feedback. *Review of Educational Research* **2008**, *78*(1), 153–189.