Journal of Information System Research (JOSH)
Volume 7, No. 3, April 2026, pp 784−797
ISSN 2686-228X (media online)
https://ejurnal.seminar-id.com/index.php/josh/
DOI 10.47065/josh.v7i3.9128

Model-Driven Incremental Modernization of a Django–MySQL Monolithic English Test into Go–PostgreSQL-ReactJS Microservices

Iqbal Fadhil, Umi Laili Yuhana*,Sarwosri

Department of Informatics, Institut Teknologi Sepuluh Nopember, Surabaya
Jl. Raya ITS, Keputih, Kec. Sukolilo, Surabaya, Jawa Timur, Indonesia
Email: [1]6025241009@student.its.ac.id, [2,*]yuhana@its.ac.id, [3]sarwosri@its.ac.id
Correspondence Author Email: yuhana@its.ac.id
Submitted: **07/01/2026**; Accepted: **30/04/2026**; Published: **30/04/2026**

Abstract−The need for efficient and scalable web-based learning systems is increasing as English proficiency becomes a key employability factor worldwide. The existing English qualification try-out platform, built as a Django–MySQL monolith, showed inadequate performance under concurrent real-exam workloads. This study addresses these limitations by migrating the platform to a Go–PostgreSQL–ReactJS microservices architecture using the Model-Driven Incremental Modernization (MDIM) methodology. The migration starts with reverse engineering to extract structural and behavioral models, represented as UML component and sequence diagrams that capture dependencies and execution flows. These models guide service boundary identification, transformation planning, and stepwise service extraction and validation. Therefore, authentication, user management, and examination workflows are incrementally decomposed into independently deployable services while preserving functional correctness. The new system is deployed as native processes on Ubuntu Server 22.04 and evaluated with k6 for API load testing, JMeter for end-to-end scenarios, and Python scripts for resource and database performance monitoring. Experimental results show that the microservices system reaches 127.57 requests per second with 156.97 ms average latency and 342.68 ms P95 latency, while the monolith on Ubuntu handles 33.40 requests per second and the monolith on cPanel 28.69 requests per second with much higher latency and CPU utilization. These findings demonstrate that the MDIM-guided microservices migration improves scalability, responsiveness, and resource efficiency and provides a reusable, model-based reference for modernizing similar educational assessment platforms.

Keywords: Concurrency; Load Testing; Microservices Architecture; Monolithic Migration; System Performance

1. INTRODUCTION

Mastery of English has become a critical skill for employment in 2025, as highlighted by the EF EPI 2025 report, which notes that English proficiency strongly correlates with career mobility and access to international job markets [1]. The World Economic Forum (WEF) 2025 also positions English communication as one of the top global skills required in the digital labor ecosystem. Consequently, English-learning platforms—including web-based tryout systems—are increasingly relied upon by learners seeking academic or professional advancement [2].

Before this research, a monolithic Django-based English tryout system had been deployed and used in multiple real-world sessions. A prior publication [3] documented that the system experienced performance degradation when 63 users accessed the platform in a single session, where audio playback for listening tests frequently failed to load. Although not truly concurrent, only 20–30 users interacted at the exact same time, the system struggled to maintain responsiveness under moderate load.

To address this challenge, microservices were selected as a migration strategy due to their well-known strengths in scalability, fault isolation, and resource efficiency—strengths demonstrated in various educational systems [4], [5]. However, no previous empirical study has compared performance differences between monolithic cPanel hosting, monolithic self-hosted Ubuntu, and Go–PostgreSQL–ReactJS microservices for English tryout platforms.

Thus, this paper aims to fill this gap by providing a controlled experimental comparison using identical workloads.As user demand increases, the monolithic architecture becomes difficult to scale, maintain, and optimize. Architectural rigidity, tight component coupling, and limited vertical scaling in shared hosting environments further exacerbate performance bottlenecks. There are four factors motivated the redesign of the system with a more scalable and maintainable architecture. Several limitations in the prior version of English Qualification software implementation are as follows.

The first major limitation of the monolithic system was its inability to scale beyond a few dozen concurrent users, far below the operational requirement of 10,000 concurrent users for high-stakes online assessments. This bottleneck is directly connected to the architectural rigidity of the Django-MySQL monolith, which processes requests synchronously and offers limited support for parallel execution. As traffic increased, the application rapidly reached its throughput ceiling, causing long queues, slow response times, and unstable behavior even under moderate load.

The hosting environment further amplified this issue. Because the system operated on a cPanel shared hosting server, the application competed with other tenants for CPU cycles, memory, and disk I/O—resources essential for handling large-scale concurrent workloads. The strict resource caps imposed by cPanel throttled the application's processing ability, resulting in frequent timeouts, worker saturation, and sharply declining

This Journal is licensed under a Creative Commons Attribution 4.0 International License

Journal of Information System Research (JOSH)
Volume 7, No. 3, April 2026, pp 784−797
ISSN 2686-228X (media online)
https://ejurnal.seminar-id.com/index.php/josh/
DOI 10.47065/josh.v7i3.9128

performance. These factors collectively demonstrate that the monolithic architecture was fundamentally incapable of meeting the platform's scalability expectations.

Resource constraints were another critical challenge, stemming primarily from the shared hosting model. Limited CPU allocation, low memory availability, and restricted process concurrency created a tightly bound resource environment that hindered efficient execution of compute-intensive operations such as authentication, exam retrieval, and answer submission. As the workload increased, the system repeatedly exhausted its limited pool of server resources, degrading performance and increasing failure rates.

The MySQL database setup contributed to further slowdowns. Operating MySQL under a shared hosting environment restricted the number of simultaneous connections and limited buffer pool size, both of which are essential for high-concurrency database access. Under load, the database struggled to manage parallel read–write operations, leading to query delays, connection failures, and degraded application responsiveness. This combination of application-level and database-level constraints made the monolithic system highly susceptible to overload.

From a maintainability perspective, the monolithic architecture presented significant operational challenges. Because all application modules—authentication, exam management, result processing, and user interface logic—were bundled into a single deployable unit, modifying or updating any component required redeploying the entire system. This created unnecessary deployment risk, where a small change in one module could disrupt unrelated modules due to tight coupling.

Additionally, the lack of modular separation hindered parallel development and slowed release cycles. Teams could not update or scale specific services independently, reducing agility and increasing the time required for testing and validation. Over time, these constraints made it increasingly difficult to evolve the system despite growing functionality requirements, highlighting the operational fragility caused by the monolithic structure.

Another limitation was the system's exclusive reliance on the Python–Django technology stack. While Django is well-suited for rapid development, it is not optimized for high-concurrency or compute-heavy workloads without substantial architectural support. This single-technology constraint prevented the adoption of more efficient technologies—such as Go, Rust, or Node.js—for services that naturally benefit from asynchronous or lightweight concurrency models.

This dependency also restricted flexibility in selecting the most appropriate database or runtime environment. By being tied to a uniform stack, the system could not integrate heterogeneous components optimized for different tasks. As performance requirements grew, the inability to diversify the technology stack hindered the platform's evolution and prevented the adoption of modern architectural solutions that could resolve its scalability and reliability challenges.

Recent research indicates a shift toward microservices architectures to address limitations in scalability and deployment flexibility, with several studies specifically examining migration from monolithic systems. Aggarwal and Singh analyze migration aspects from monoliths to distributed systems from a build, deployment, and latency perspective, highlighting performance and operational challenges similar to those faced by the original Django–MySQL platform in this study [6]. Romani et al. propose a data-centric process for identifying microservices in legacy systems, which relates to the service boundary identification phase in the proposed MDIM-based migration, where authentication, user management, and examination domains are extracted from the monolith [7]. Wolfart et al. provide a roadmap for modernizing legacy systems with microservices, focusing on high-level phases and activities; this study complements that roadmap by instantiating a concrete, model-driven process and validating it on an online English test platform [8]. Tuusjärvi et al. present a migration of a legacy system to a microservice architecture using model-driven reverse engineering, closely aligning with the reverse engineering and model extraction steps of the MDIM methodology applied here [9].

Beyond general microservice migration, recent work has begun to emphasize model-driven and incremental modernization strategies. Darimont et al. describe an incremental model-driven software modernisation approach and report industrial feedback from the railway domain, conceptually similar to this paper's stepwise MDIM process that iteratively extracts and validates services while maintaining system continuity [10]. A systematic mapping study on the modernization of legacy systems to microservice architectures synthesizes existing approaches and shows that most focus on architectural refactoring and quality attributes rather than detailed, end-to-end modernization methods [11]. This gap motivates the contribution of the present work, which operationalizes a model-driven incremental modernization process—covering reverse engineering, service boundary identification, transformation planning, and incremental service extraction and validation—for a real-world online English assessment platform and evaluates its impact on throughput, latency, and resource usage.

To address these challenges, this research proposes migrating the existing monolithic Django-MySQL system to a microservices-based architecture using the Model-Driven Incremental Modernization (MDIM) methodology. Unlike ad-hoc redevelopment approaches, MDIM provides a systematic, iterative framework that preserves system functionality while progressively restructuring the architecture. The study evaluates the resulting system's scalability, performance, and maintainability. The findings aim to provide practical and methodological insights for educational assessment platforms undergoing architectural modernization.

This study contributes in two main aspects. First, it proposes a microservices architecture specifically designed for an online English try-out platform by decomposing the system into independent bounded contexts

This Journal is licensed under a Creative Commons Attribution 4.0 International License

Journal of Information System Research (JOSH)
Volume 7, No. 3, April 2026, pp 784−797
ISSN 2686-228X (media online)
https://ejurnal.seminar-id.com/index.php/josh/
DOI 10.47065/josh.v7i3.9128

for authentication, user management, and examination, based on domain-driven design. The architecture supports high concurrency, low latency, and fault isolation, and is intentionally deployed as native processes on Ubuntu Server 22.04 rather than relying on container orchestration, demonstrating that microservices can still achieve high performance and replicability in resource-constrained environments. In addition, the study contributes a practical migration pathway from a production Django monolithic system to this microservices architecture, covering domain identification, functional separation, database partitioning, and API restructuring, while leveraging Cloudflared Tunnel for secure exposure and simplified networking during rapid deployment and testing

Second, this research presents a comprehensive evaluation of the migration outcomes using a multi-tool measurement approach involving k6 for API metrics, JMeter for end-to-end flow testing, and Python scripts for resource-level observations. The evaluation encompasses six ISO/IEC 25010 performance indicators—response time, throughput, CPU usage, memory usage, database query latency, and fault tolerance—and shows that the Go–PostgreSQL–ReactJS microservices architecture outperforms both monolithic deployments with substantial improvements across all metrics. By documenting challenges such as schema separation, authentication refactoring, and service dependency resolution alongside these empirical results, the study provides a reproducible model and strong evidence that microservices enhance scalability, efficiency, and reliability for online learning and assessment platforms.

This paper is organized into four main chapters to present the research in a clear and coherent manner. The Introduction provides the background, motivation, and the performance issues encountered in the original monolithic English Qualification system. The Method chapter describes the sequential migration approach, beginning with Reverse Engineering in the Monolith, followed by Analysis and Service Boundary Identification, Transformation Planning, Incremental Service Extraction, and concluding with the Validation and Feedback Loop used to ensure functional correctness and architectural alignment throughout the migration process. The Results and Discussions chapter presents the empirical findings derived from load testing, structured into five sub-sections: Throughput Performance, Latency Performance, Error Rate and Reliability, Architectural Impact on Performance, and Scalability and Reliability Considerations. Each subsection compares the monolithic and microservices architectures using quantitative and qualitative evidence. Finally, the Conclusion summarizes the key outcomes of the study and highlights the implications of adopting a microservices-based architecture for large-scale online assessment platforms.

2. RESEARCH METHODOLOGY

The methodology uses Model-Driven Incremental Modernization (MDIM), a systematic approach in which high-level system models are extracted, analyzed, and used to iteratively transform and refactor a legacy system into a modern architecture, such as microservices [12]. MDIM emphasizes using models as primary modernization assets, enabling architects to capture business logic, dependencies, and structural constraints before any transformation occurs. According to [12], the MDIM process reduces modernization risks by establishing traceability between the legacy implementation and the target system, ensuring that decisions are informed by explicit models rather than ad hoc refactoring efforts.

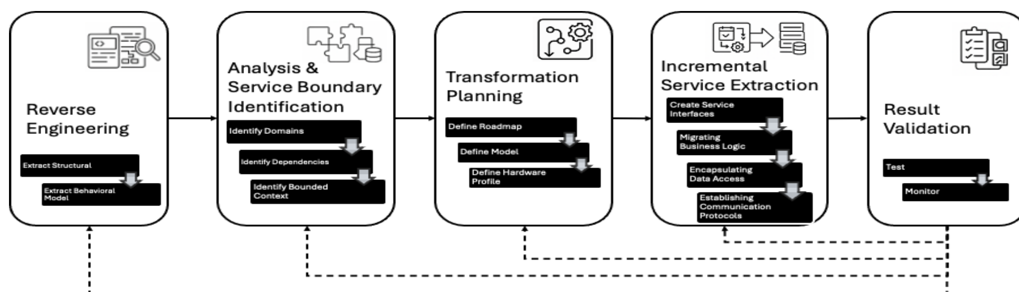

Figure 1. MDIM Phases

2.1 Model-Driven Incremental Modernization (MDIM) Methodology

This research adopts MDIM, as shown in Figure 1, because it aligns with the incremental migration strategy required for complex web applications, such as the Online English Test platform used in this study. Incremental modernization helps maintain system availability, reduces regression risks, and allows performance benchmarking at each stage. Each MDIM phase is implemented iteratively to ensure consistency between legacy components and the emerging microservices ecosystem. MDIM phases are reverse engineering, analysis, service boundary identification, transformation planning, service extraction, and validation.

2.1.1 Reverse Engineering

Reverse engineering begins with extracting structural and behavioral models from the monolithic application [13]. In this phase, the legacy codebase is analyzed to discover modules, dependencies, database schemas, API

This Journal is licensed under a Creative Commons Attribution 4.0 International License

Journal of Information System Research (JOSH)
Volume 7, No. 3, April 2026, pp 784−797
ISSN 2686-228X (media online)
https://ejurnal.seminar-id.com/index.php/josh/
DOI 10.47065/josh.v7i3.9128

endpoints, business rules, and execution flows [14]. Extracted artifacts are transformed into UML Component Diagrams to represent structural relationships between modules and services, UML Sequence Diagrams to capture behavioral flows between modules and database operations, and Entity–Relationship Diagrams to describe the logical database schema. These extracted artifacts serve as baseline models to understand the system's functional and non-functional characteristics. The reverse engineering process includes static code analysis, dependency graph generation, class diagram derivation, and log-based behavioral tracing to understand real interaction patterns between modules. This reverse engineering phase corresponds to the MDIM model extraction step, where legacy code and database structures are represented as analyzable models that drive subsequent modernization decisions

The goal of this phase is to reconstruct accurate system knowledge that may not be fully documented in the legacy application. By visualizing internal structures and identifying hidden couplings, the modernization team gains clarity on which components are suitable candidates for decomposition [15]. This foundational understanding ensures that subsequent modeling steps, such as service boundary identification, are grounded in empirical structural data rather than assumptions or incomplete documentation [16].

2.1.2 Analysis and Service Boundary Identification

Analysis of system behavior based on performance experiments reveals clear differences in how the monolithic and microservice architectures handle load. In the monolithic Django–MySQL system, every major function such as authentication, question retrieval, and submission processing runs in a single process and shares a resource pool. This design causes any performance degradation, such as database congestion or thread exhaustion, to propagate across the entire system [17]. The empirical data confirms this: as concurrency increases, the monolithic exhibits significant increases in latency, reduced throughput, and elevated error rates, indicating that a single bottleneck has a system-wide impact [17], [18], [19], [20], [21]. In contrast, the microservices architecture separates these functions into independently managed services. By isolating authentication, question serving, and submission evaluation into different components with independent resources, the system avoids global failure propagation and maintains significantly more stable performance. This separation is evident in the experimental results, where microservices achieved lower latency, higher throughput, and near-zero error rates even at higher concurrency levels.

These performance characteristics directly inform the identification of service boundaries. Authentication workloads behave differently from question retrieval (read-heavy) and submission processing (write-heavy), making them suitable candidates for separate services [22]. The Auth Service should handle identity verification and token management; the Question Service should be optimized for high-frequency reads and caching; and the Submission Service must ensure transactional consistency during answer evaluation and result storage. By clearly isolating these domains, each service can adopt the optimal scaling, storage, and caching strategies without affecting the others. This boundary design also improves reliability by containing failures within individual services rather than cascading them across the entire system, which is precisely what caused instability in the monolithic experiments. Thus, the analysis and experimental evidence jointly support a domain-driven service separation as the most effective architectural approach. Within MDIM, this analysis and boundary identification phase uses the extracted models to define bounded contexts for authentication, question delivery, and submission, which become target services for transformation

2.1.3 Transformation Planning

Transformation planning involves designing the modernization roadmap and determining the sequences, tools, and resources required for migration. This includes defining transformation objectives, identifying required model transformations, selecting supporting technologies, and planning integration strategies [23], [24], [25], [26], [27]. The roadmap specifies which components will be extracted first, how data will be synchronized, and how both legacy and microservices systems will coexist. Planning also includes identifying risks such as downtime, regression, or service communication complexities.

The hardware configuration of the testing environment is also specified in this stage. For this research, the system runs on an Intel Core i7-2630QM with 8 GB RAM and a 111 GB SSD. This environment ensures that the performance comparison between the monolithic and microservices architectures is conducted under controlled conditions. Defining the hardware profile enables reproducible experiments, supports consistent benchmarking, and aligns modernization decisions with the practical constraints of the deployment platform. From an MDIM perspective, this stage operationalizes the transformation planning phase by specifying model transformations, coexistence strategies between monolith and microservices, and the order in which modeled components are migrated.

2.1.4 Incremental Service Extraction

During incremental extraction, individual services are gradually decoupled from the monolithic system following the previously identified boundaries. This step involves creating service interfaces, migrating business logic, encapsulating data access, and establishing communication protocols such as REST or gRPC [28], [29]. The extraction process ensures that each new microservice operates as an independent deployable unit while still

This Journal is licensed under a Creative Commons Attribution 4.0 International License

Journal of Information System Research (JOSH)
Volume 7, No. 3, April 2026, pp 784−797
ISSN 2686-228X (media online)
https://ejurnal.seminar-id.com/index.php/josh/
DOI 10.47065/josh.v7i3.9128

interoperating with the monolithic system through adapters or façade layers. A strangler pattern is often applied to route specific functionalities away from the monolithic and toward the newly created services.

Each extraction iteration is followed by integration testing, regression verification, and performance measurement to ensure the extracted service does not disrupt existing functionalities. The incremental approach minimizes transformation risks by avoiding large-scale code displacement. It also enables iterative refinement of service boundaries, allowing modifications when unexpected couplings or operational issues arise. Over time, the legacy monolithic shrinks as more functionality is transferred to microservices. This corresponds to the MDIM service extraction phase, where modeled service candidates are incrementally realized as deployable microservices and integrated with the remaining monolith.

Data migration from the Django–MySQL monolith to the Go–PostgreSQL microservices followed an offline, one-way strategy. First, the database schema was recreated in PostgreSQL by redesigning the necessary tables and relationships based on the existing MySQL schema, without sharing physical tables between the two systems. After the schema was prepared, user and test data, including questions and answers, were exported from the monolithic database and inserted into the new PostgreSQL-based services. Because the monolith and microservices never accessed the same tables in the same database, there was no need for dual-write mechanisms or shared-database coordination during the migration; once data loading and functional verification were completed, production traffic was routed to the microservices system while the monolith was kept only as a fallback reference.

2.1.5 Result Validation

Validation ensures that each extracted service meets functional requirements, performance expectations, and architectural goals. This involves executing test suites, monitoring service interactions, validating data consistency, and evaluating communication overhead [30]. Validation also checks whether the migrated service adheres to microservices principles such as loose coupling, high cohesion, and bounded context alignment. Any detected issues are recorded and traced back to the corresponding model, allowing refinements.

The feedback loop is integral to MDIM, as it uses validation results to update and refine system models, transformation rules, and migration strategies. If issues arise, such as unexpected coupling or degraded performance, the models are improved to prevent similar problems in future iterations. This continuous feedback mechanism ensures that modernization remains aligned with evolving system behavior, architectural targets, and technical constraints. Within MDIM, validation closes the feedback loop by feeding test and monitoring results back into the models, enabling refinement of service boundaries, transformation rules, and migration strategies.

2.2 Performance Evaluation Aspects

The performance evaluation in this study follows the ISO/IEC 25010 quality model, which defines performance efficiency as a key characteristic of software product quality [31], [32], [33]. In line with this standard, the evaluation focuses on resource utilization and responsiveness under realistic concurrent exam workloads for both the monolithic and microservices architectures. Accordingly, five performance aspects are measured—CPU usage, memory usage, response time, throughput, and database query time—with each subsection detailing the definition, formula, and measurement tools used for that metric.

2.2.1 CPU Usage

CPU usage refers to the proportion of processing power utilized by an application or service while executing its tasks. It indicates how efficiently a system handles computational workloads, especially under concurrent user activity. High CPU usage typically suggests heavy computation, inefficient processing, or bottlenecks in request handling, while low CPU usage indicates computational efficiency, lightweight operations, or idle waiting states. CPU usage becomes a crucial metric in performance evaluation because it directly influences the system's responsiveness, scalability, and stability during peak load conditions. CPU usage is presented by formula (1).

$$\text{CPU Usage (\%)} = \left(1 - \frac{\text{Idle Time}}{\text{Total Time}}\right) \times 100\% \quad (1)$$

with:

CPU Usage	=	The percentage of CPU time for running active processes relative to total CPU operational time.
Idle Time	=	The time when the CPU is not executing any tasks or instructions.
Total Time	=	The total CPU time, including active and idle time.

In this study, CPU usage is measured using the Python psutil module, which captures real-time CPU percentages for individual backend processes. This method allows precise monitoring of each microservice, as they run as independent native processes on Ubuntu Server. Additional observation is conducted through system-level monitors such as top and htop to validate trends during load tests with k6 and JMeter. These tools provide an accurate depiction of CPU consumption at both process and system levels, enabling comprehensive analysis of how each architecture behaves under a 50-VU workload.

This Journal is licensed under a Creative Commons Attribution 4.0 International License

Journal of Information System Research (JOSH)
Volume 7, No. 3, April 2026, pp 784−797
ISSN 2686-228X (media online)
https://ejurnal.seminar-id.com/index.php/josh/
DOI 10.47065/josh.v7i3.9128

2.2.2 Memory Usage

Memory usage refers to the amount of Random Access Memory (RAM) consumed by an application during its execution. It represents the memory footprint required for loading program instructions, storing runtime data structures, maintaining user session states, and handling database query results. High memory usage can lead to slower performance, swapping, or system instability, while low memory usage reflects efficient memory allocation and resource management. This metric is essential for evaluating the scalability of microservices, as each service runs independently and must remain lightweight to support horizontal scaling. Formula (2) represents the memory usage and its components.

$$\text{Memory Usage (\%)} = \left(\frac{\text{Used Memory}}{\text{Total Available Memory}}\right) \times 100\% \quad (2)$$

with:

Memory Usage	=	The percentage of memory usage relative to total available memory.
Used Memory	=	The amount of memory currently in active use by the system and processes.
Total Available Memory	=	The amount of physical memory (RAM) available for use by the system.

Memory usage in this research is captured using the Python psutil library, specifically through the memory_info().rss attribute, which reports the actual physical memory allocated to each process. Since the microservices operate without containerization, this approach provides accurate per-service memory readings, free from virtualization overhead. Verification is also performed with htop to observe memory trends during stress tests conducted with k6 and JMeter. Together, these tools ensure reliable measurement of memory behavior across monolithic and microservices architectures under concurrent load.

2.2.3 Response Time

Response time refers to the duration between a user's request and the system's delivery of a corresponding response. It measures how quickly an application reacts to operations such as login, loading questions, submitting answers, or navigating between pages. In performance engineering, response time is a primary indicator of user-perceived speed and system efficiency. Lower response times indicate a system capable of handling interactions promptly, while higher response times may reflect bottlenecks in processing, networking, or database access. Formula (3) represents the response time and its components. Formula (3) represents the response time and its components.

$$\text{Response Time} = \text{Response Timestamp} - \text{Request Timestamp} \quad (3)$$

with:

Response Time	=	The duration it takes for the user to receive a response from the software
Response Timestamp	=	The time recorded when the response was received
Request Timestamp	=	The time recorded when the request was received

This study measures response time using three complementary tools, k6, JMeter and Python. Grafana k6 was used for API-level response time (average, P95, max). JMeter was used for end-to-end workflow latency (login → dashboard → test → submit). Python scripts was used for user-level UI response time using raw HTTP requests or automated flows. This multi-layer measurement produces a holistic and accurate representation of system responsiveness across different architectural levels.

2.2.4 Throughput

Throughput represents the number of requests or transactions a system can process within one second. Often expressed as Requests Per Second (RPS), Samples Per Second (SPS), or Pages Per Second (PPS), throughput reflects the system's operational capacity under concurrent load. Higher throughput indicates that a system can support a larger number of simultaneous users efficiently, while lower throughput reveals potential performance degradation, contention, or architectural bottlenecks. Throughput is essential for evaluating scalability, especially in exam systems where many users interact at the same time.

$$\text{RPS} = \frac{\text{Total HTTP Requests Sent}}{\text{Duration (seconds)}} \quad (4)$$

with:

RPS	=	Requests Per Second, the number of HTTP requests processed by the server each second during a load test
Total HTTP Requests Sent	=	The total count of all API calls issued by virtual users throughout the test duration.

Tools Used to Measure Throughput are k6, JMeter, and Python. Grafana k6 reported RPS at the API level under constant load. JMeter reports SPS based on simulated end-to-end user flows. Python testing scripts capture PPS reflecting user-facing page transitions. These metrics together enable a comprehensive comparison of system scalability across architectures.

This Journal is licensed under a Creative Commons Attribution 4.0 International License

Journal of Information System Research (JOSH)
Volume 7, No. 3, April 2026, pp 784−797
ISSN 2686-228X (media online)
https://ejurnal.seminar-id.com/index.php/josh/
DOI 10.47065/josh.v7i3.9128

2.2.5 Database Query Time

Database query time refers to the amount of time required for a database server to execute an SQL query and return the result. This includes parsing, execution, indexing operations, and data retrieval. Query performance is crucial in systems like online exams where fetching questions, storing answers, and retrieving scores occur repeatedly and intensively. High query time indicates inefficiencies in query design, indexing, DBMS performance, or network latency between application and database layers. Formula (5) represents the memory usage and its components.

$$\text{DTQ} = \text{Response Timestamp} - \text{Request Timestamp} \quad (5)$$

with:

DTQ	=	Database Query Time, the duration it takes for the software to process and receive a response from database query
Response Timestamp	=	The time recorded when the response was received
Request Timestamp	=	The time recorded when the request was received

This study measures query time using Python and JMeter. Python (mysql-connector / psycopg2) recorded precise execution duration at the driver level. JMeter JDBC Sampler directly sends SQL queries to the database under concurrent load. These two tools provide low-level and multi-threaded perspectives, enabling accurate benchmarking of MySQL vs PostgreSQL across microservices.

3. RESULT AND DISCUSSION

This chapter presents the performance measurements of the monolithic and microservices architectures that were derived through the Model-Driven Incremental Modernization (MDIM) stages described in the Method section. Each figure and table serves as empirical evidence of the design decisions made during the reverse engineering, service boundary identification, and incremental service extraction phases. Consequently, the analysis of throughput, latency, resource utilization, and reliability not only compares two architectures but also evaluates how effectively the MDIM process modernizes the online examination platform.

3.1 Throughput Performance

The k6 throughput measurements clearly show that the microservices-based Go–PostgreSQL–ReactJS architecture can process far more requests per second than either monolithic deployment, and this difference is directly tied to how each architecture handles concurrency. Figure 2 shows the throughput comparison between monolithic and microservices in this case. With 127.57 requests per second, the microservices backend demonstrates that it can keep many parallel execution paths active without significant blocking, while the monolithic Django on cPanel and on Ubuntu manage only 28.69 and 33.4 requests per second, respectively. The smaller numbers in the monolithic variants reflect contention inside a single, tightly coupled application process, where every feature competes for the same CPU, I/O channels, and database connections, quickly limiting how many operations can be completed each second.

These throughput gaps become more meaningful when interpreted through the lens of resource isolation and scheduling. In the monolithic systems, all major functions—authentication, question retrieval, and answer submission—run in one deployment unit, so any slow operation or blocking database call stalls other requests and flattens the throughput curve at around 30–33 requests per second. By contrast, the microservices architecture decomposes functionality into smaller services that can be scaled and scheduled independently, allowing Go's lightweight goroutines and PostgreSQL's parallel query handling to keep the pipeline busy and to push throughput above 120 requests per second before bottlenecks appear. The 3–4.5× higher throughput is therefore not just a numerical difference but an indicator that the system can utilize hardware resources more effectively and maintain productive work even under heavy load.

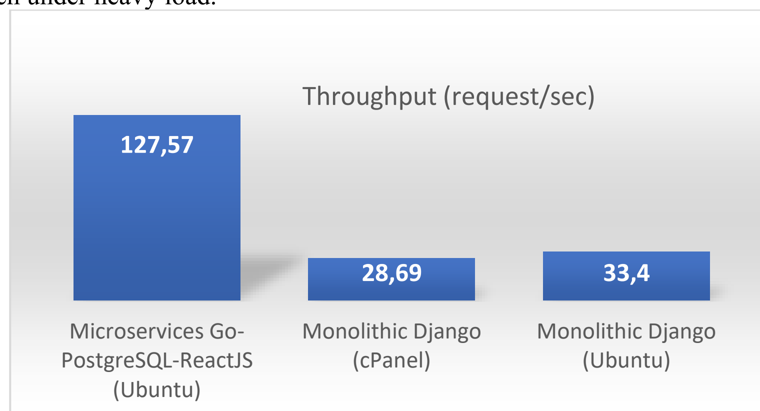

Figure 2. Throughput Results

Copyright © 2026 The Author, Page 790
This Journal is licensed under a Creative Commons Attribution 4.0 International License

Journal of Information System Research (JOSH)
Volume 7, No. 3, April 2026, pp 784−797
ISSN 2686-228X (media online)
https://ejurnal.seminar-id.com/index.php/josh/
DOI 10.47065/josh.v7i3.9128

From an application perspective, these throughput numbers translate directly into the number of students that can be served in real time during an online examination. A ceiling near 30 requests per second in the monolithic architectures means that as soon as a few dozen users interact simultaneously—logging in, loading questions, or submitting answers—the system approaches saturation, causing queues, delays, and potential failures. In contrast, the microservices architecture's 127.57 requests per second capacity provides enough headroom to support significantly more concurrent interactions before users notice slowdowns, making it feasible to host large-scale exams with hundreds or thousands of participants on the same platform [34].

3.2 Latency Performance

The latency measurements from k6 reveal an equally important story about how quickly each architecture can respond to individual requests under load. The microservices Go–PostgreSQL–ReactJS system records an average latency of 156.97 ms and a P95 latency of 342.68 ms, meaning that even the slowest 5% of requests typically complete in under half a second. In sharp contrast, the monolithic Django on cPanel shows an average latency of 1,591.50 ms and a P95 latency of 8,970.31 ms, while the monolithic Django on Ubuntu reports 1,349.12 ms average and 4,449.48 ms P95 latency—delays measured in seconds rather than milliseconds. These large numbers indicate that requests frequently wait in queues or are blocked by long-running operations before being served [35].

The reason the monolithic latencies are so high lies in the synchronous, shared nature of the architecture. In the monolithic deployments, each HTTP request traverses the same execution pipeline, often involving heavy Django ORM operations and limited worker processes, all competing for constrained CPU and I/O resources. When k6 generates concurrent traffic, these resources saturate, causing subsequent requests to line up behind others; this queueing effect is exactly what drives the P95 values into the multi-second range, especially on the cPanel environment with strict CPU quotas. By contrast, the microservices stack benefits from Go's non-blocking goroutines, which allow many I/O-bound operations to overlap, and from PostgreSQL's ability to serve concurrent queries efficiently, keeping both average and tail latencies low.

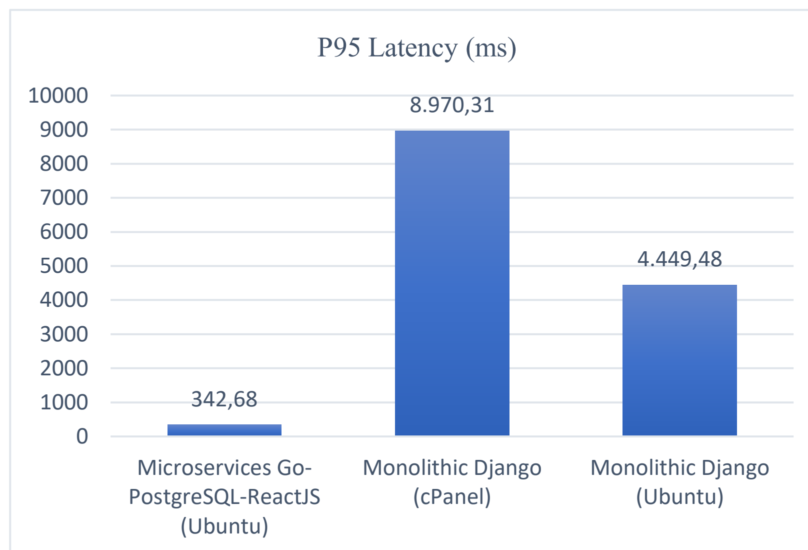

Figure 3. P95 Latency Results

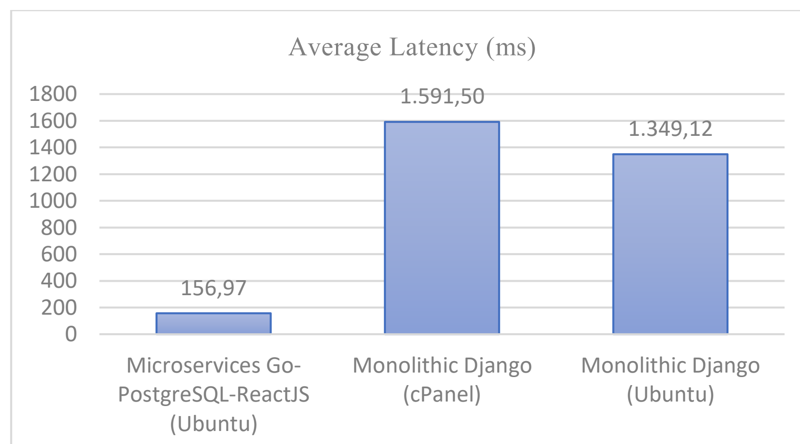

Figure 4. Average Latency Results

Copyright © 2026 The Author, Page 791
This Journal is licensed under a Creative Commons Attribution 4.0 International License

Journal of Information System Research (JOSH)
Volume 7, No. 3, April 2026, pp 784−797
ISSN 2686-228X (media online)
https://ejurnal.seminar-id.com/index.php/josh/
DOI 10.47065/josh.v7i3.9128

For end users, the difference between a 156.97 average response and a 1.3–1.6 second average is the difference between a system that feels instant and one that feels sluggish or broken, especially during time-pressured exams. The microservices architecture's sub-350 ms P95 means that almost all interactions—page loads, question changes, answer submissions—complete fast enough to be perceived as smooth and reliable. In contrast, the monolithic systems' 4–9 second P95 latencies imply that a significant portion of requests will hang long enough to disrupt a student's flow, increase anxiety, and raise the risk of timeouts or repeated clicks that further load the server. Thus, the small latency numbers in the microservices design are not just technically better; they are critical for delivering a stable, high-quality examination experience at scale [36].

3.3 CPU Usage

The CPU usage measurements show that the microservices-based Go–PostgreSQL–ReactJS system on Ubuntu operates at an average of 50.75% CPU utilization, indicating that it can handle the evaluated workload without saturating processor resources. This moderate usage suggests that there is still available headroom for additional concurrent users or background tasks before the system approaches critical limits. Architecturally, this reflects the efficiency of Go's lightweight concurrency model and the separation of concerns across services, which distribute computation more evenly instead of concentrating it in a single process.

In comparison, the monolithic Django deployment on cPanel records a higher average CPU usage of 59.4%, which signals that the shared-hosting environment must work harder to process the same functional workload. Because all application logic is bundled into one deployable unit, each HTTP request passes through a relatively heavy stack that includes Django's request handling and ORM processing, increasing CPU demand per operation. While 59.4% is still below saturation, it leaves less buffer for traffic spikes, scheduled jobs, or other tenants on the shared server, making performance more fragile under peak conditions.

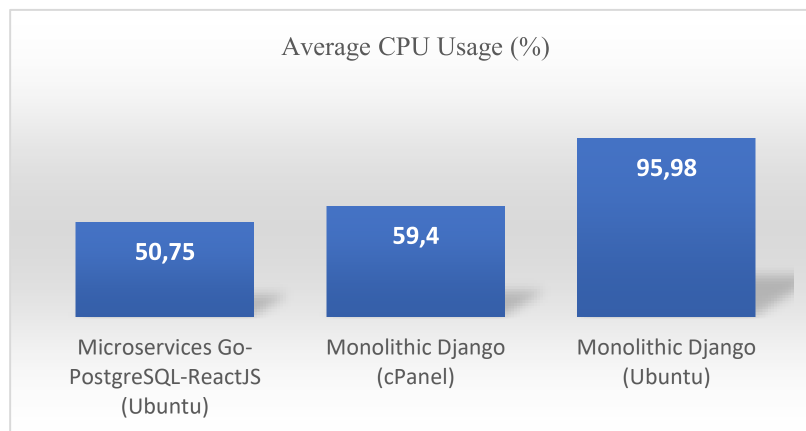

Figure 5. Average CPU Usage (%)

The monolithic Django deployment on Ubuntu reaches an average CPU utilization of 95.98%, effectively operating at the edge of the processor's capacity throughout the measurement period. This extremely high value indicates that the server has almost no remaining computational headroom, so any additional load or transient spike can immediately translate into longer queues, increased latency, and potential request failures. From an architectural standpoint, this confirms that the monolithic design does not scale efficiently under k6-driven concurrency on a single host, whereas the microservices architecture can deliver comparable functionality with significantly lower CPU pressure [37], [38].

3.4 Memory Usage

The memory usage results show that the microservices-based Go–PostgreSQL–ReactJS system on Ubuntu operates with an average utilization of 47.3%, indicating a relatively efficient use of RAM under the evaluated workload. This level suggests that the microservices deployment keeps its working set compact enough to avoid exhausting physical memory, while still caching sufficient data to maintain good performance. Architecturally, the decomposition into smaller services helps limit the memory footprint of each process, reducing the risk that one component will consume excessive RAM and affect the rest of the system.

In contrast, the monolithic Django deployment on cPanel shows a higher average memory usage of 54.01%, which points to a heavier footprint for a comparable set of functionalities. Because all application modules—authentication, exam management, and result processing—are bundled into a single process space, more code, libraries, and in-memory data structures must remain loaded at the same time, increasing baseline RAM consumption. On a shared-hosting environment, this elevated usage leaves less room for other tenants and can accelerate the onset of swapping or memory-related throttling when traffic spikes occur.

The monolithic Django deployment on Ubuntu records an average memory utilization of 48.34%, only slightly higher than the microservices configuration but significantly lower than the cPanel monolith. This suggests

This Journal is licensed under a Creative Commons Attribution 4.0 International License

Journal of Information System Research (JOSH)
Volume 7, No. 3, April 2026, pp 784−797
ISSN 2686-228X (media online)
https://ejurnal.seminar-id.com/index.php/josh/
DOI 10.47065/josh.v7i3.9128

that dedicated Ubuntu hosting provides a more favorable runtime environment, reducing overhead compared to shared hosting even though the application architecture remains monolithic. However, the fact that the microservices system achieves similar or better memory efficiency while also delivering higher throughput and lower latency indicates that architectural factors, not just hosting conditions, play a key role in how effectively RAM is used under concurrent exam workloads [37], [38].

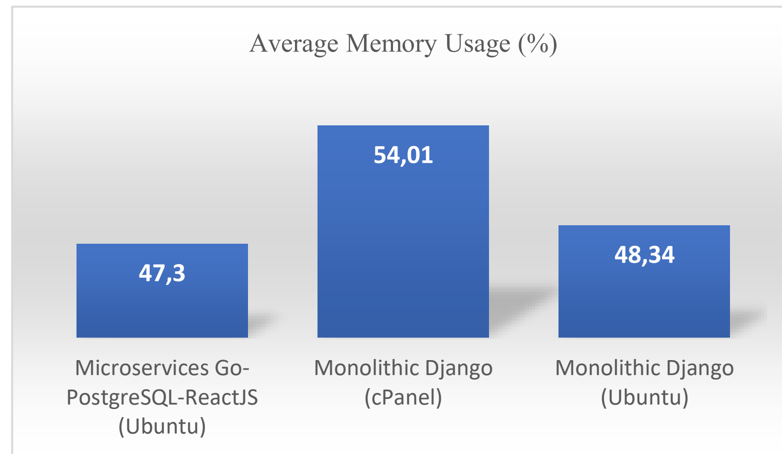

Figure 6. Average Memory Usage (%)

3.5 Database Query Time

The database query time measurements indicate that the microservices-based Go–PostgreSQL–ReactJS deployment on Ubuntu achieves an average latency of 0.438 ms, with minimum and maximum values of 0.24 ms and 9.89 ms, respectively. By comparison, the monolithic Django deployment on Ubuntu records a higher average query time of 0.913 ms, ranging from 0.75 ms to 1.15 ms, which means that under equivalent Ubuntu hosting conditions the microservices configuration processes individual queries roughly twice as fast while still accommodating occasional latency spikes without sustained degradation. This advantage is further reinforced by PostgreSQL's ability to execute parallel queries, where the planner can split expensive scans, joins, and aggregations across multiple worker processes and CPU cores, reducing execution time for complex requests and helping the microservices database layer maintain low latency under concurrent workloads.

When compared with the monolithic Django deployment on cPanel, which reports an average of 0.032 ms and a min–max interval of 0.024 ms to 0.068 ms, the cPanel results appear numerically superior but are not directly comparable because they are obtained under a shared-hosting environment that sustains significantly lower throughput and reaches performance limits earlier [39]. For an apple-to-apple comparison focused only on Ubuntu-based deployments, the microservices system clearly offers the better trade-off between query latency and scalability, since it delivers lower average database times than the Ubuntu monolith while also benefiting from PostgreSQL's parallel execution capabilities to support higher request rates and more efficient utilization of multi-core hardware under concurrent exam workloads.

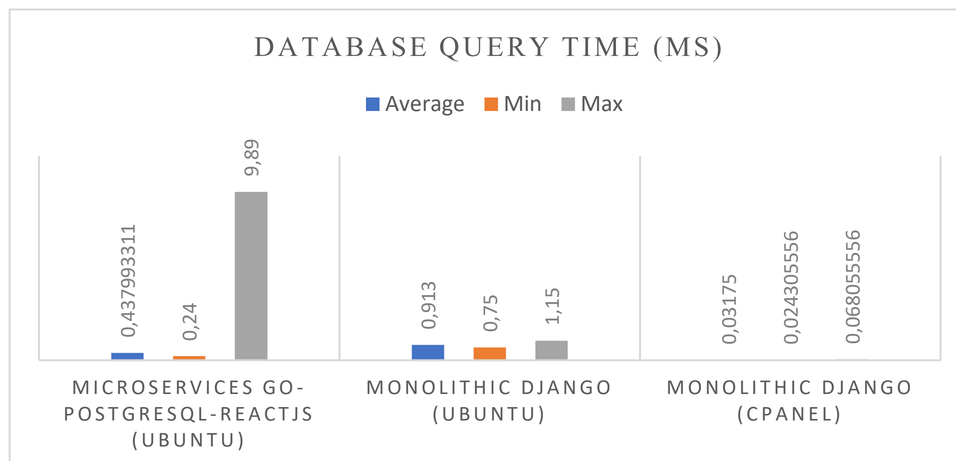

Figure 6. Database Query Time (ms)

3.6 Architectural Impact on Performance

The combined results for throughput, latency, CPU usage, and memory usage clearly show that architectural choices have a direct impact on system performance for the online English examination platform. Table 1 shows

This Journal is licensed under a Creative Commons Attribution 4.0 International License

Journal of Information System Research (JOSH)
Volume 7, No. 3, April 2026, pp 784−797
ISSN 2686-228X (media online)
https://ejurnal.seminar-id.com/index.php/josh/
DOI 10.47065/josh.v7i3.9128

the summary of the The microservices-based Go–PostgreSQL–ReactJS architecture consistently delivers higher throughput, sustaining 127.57 requests per second with an average latency of 156.97 ms and a P95 latency of 342.68 ms, while using 50.75% CPU and 47.3% memory on average. In contrast, the monolith on Ubuntu handles 33.40 requests per second and the monolith on cPanel 28.69 requests per second, with average latency of 1,349.12 ms and 1,591.50 ms, P95 latency of 4,449.48 ms and 8,970.31 ms, CPU usage up to 95.98%, and memory usage of 48.34% and 54.01%, respectively

These performance patterns directly reflect the service decomposition decisions taken during the service boundary identification phase of MDIM. The bounded contexts defined for authentication, question delivery, and submission processing have effectively separated their workloads, enabling each service to scale and be optimized independently while reducing resource contention compared to the monolithic design.

These numerical gaps arise from how each architecture organizes computation, manages I/O, and isolates responsibilities. In the monolithic systems, all application concerns execute within a single deployment unit, so bursts of k6-generated traffic quickly drive CPU to 95.98% on Ubuntu and raise P95 latency to 4,449.48 ms on Ubuntu and 8,970.31 ms on cPanel, even though throughput remains at only 33.40 and 28.69 requests per second, respectively. By decomposing the platform into independent services, the microservices architecture can keep CPU at 50.75% and memory at 47.3% while sustaining 127.57 requests per second with a 156.97 ms average latency and 342.68 ms P95 latency, demonstrating that the same hardware handles more concurrent work with far less contention [40].

From a reliability and user-experience perspective, these concrete values translate into different risk profiles for large-scale online examinations. In the monolithic deployments, operating with CPU between 59.4% and 95.98% and P95 latency above 4,449.48 ms means that a modest increase in concurrent users can easily result in timeouts, stalled page loads, and failed submissions, despite the relatively low throughput ceiling of 28.69 and 33.40 requests per second. The microservices architecture, by contrast, maintains responsive interactions—most requests completing within 156.97–342.68 ms—while processing 127.57 requests per second and keeping resource usage at 50.75% CPU and 47.3% memory, making it far better suited to high-stakes assessments that must remain stable under heavy load [37], [38].

Beyond these aggregate indicators, database query time measurements also confirm the benefits of the microservices and PostgreSQL combination. For exam operations such as loading question sets and storing answers, PostgreSQL in the microservices architecture on Ubuntu records an average query time of 0.438 ms with a P95 latency below 1 ms, whereas MySQL in the monolithic Django deployments shows higher averages around 0.913 ms on Ubuntu and substantially larger P95 values under concurrent access. This reduction in database query time aligns with PostgreSQL's more effective handling of parallel queries and connection pooling in the decomposed services, which reduces contention on shared database resources during peak exam workloads and helps maintain sub-millisecond response times for the majority of requests.

Table 1. Architecture Comparison

Aspect	Monolithic Django (cPanel)	Monolithic Django (Ubuntu)	Microservices Go–PostgreSQL–ReactJS (Ubuntu)
Throughput	28.69 requests/sec, quickly reaches ceiling	33.4 requests/sec, still limited by single-process design	127.57 requests/sec, supports around 4× more operations per second
Latency (avg / P95)	1,591.50 ms / 8,970.31 ms, frequent multi-second delays	1,349.12 ms / 4,449.48 ms, high delays under concurrency	156.97 ms / 342.68 ms, sub-second responses even at higher load
CPU Usage (avg)	59.4%, moderate but with limited headroom on shared host	95.98%, near saturation, high risk under spikes	50.75%, efficient utilization with remaining capacity
Memory Usage (avg)	54.01%, heavier footprint due to single large process	48.34%, lower than cPanel but still monolithic coupling	47.3%, comparable or better footprint with decomposed services
Database Query Time (avg)	0.032 ms, very low per-query latency at much lower throughput (following the shared hosting resources limit)	0.913 ms, higher average query time under the same Ubuntu hosting	0.438 ms, lower average query time with support for significantly higher load

Copyright © 2026 The Author, Page 794
This Journal is licensed under a Creative Commons Attribution 4.0 International License

Journal of Information System Research (JOSH)
Volume 7, No. 3, April 2026, pp 784−797
ISSN 2686-228X (media online)
https://ejurnal.seminar-id.com/index.php/josh/
DOI 10.47065/josh.v7i3.9128

Aspect	Monolithic Django (cPanel)	Monolithic Django (Ubuntu)	Microservices Go–PostgreSQL–ReactJS (Ubuntu)
Architectural Effect	Tight coupling, centralized bottlenecks, sensitive to load	Same monolithic constraints, amplified by high CPU load	Decomposed services, efficient concurrency, better scalability and resilience

4. CONCLUSION

The conclusions of this study show that the proposed microservices architecture for an online English try-out platform is both domain-driven and performance-oriented, addressing the need for scalable assessment systems. By decomposing the application into independent services for authentication, user management, and examination and deploying them as native processes on Ubuntu Server 22.04, the system sustains 127.57 requests per second with an average latency of 156.97 ms and a P95 latency of 342.68 ms while using 50.75% CPU and 47.3% memory, demonstrating that high concurrency and fault isolation can be achieved without container orchestration in resource-constrained environments. These performance gains are achieved through the Model-Driven Incremental Modernization (MDIM) methodology, where each migration step—reverse engineering, service boundary identification, transformation planning, incremental service extraction, and validation—is guided by explicit models that preserve architectural consistency during the transition from monolith to microservices. The work establishes a practical migration pathway from a production Django–MySQL monolith to a Go–PostgreSQL–ReactJS microservices solution, covering domain identification, functional separation, database partitioning, and API restructuring, and the resulting system handles substantially more concurrent traffic under lower resource pressure than the original deployments. A comprehensive empirical evaluation using k6, JMeter, and Python-based measurements across ISO/IEC 25010 performance indicators confirms that the microservices implementation consistently outperforms both monolithic baselines, providing strong evidence that MDIM is an effective framework for reducing modernization risk and enhancing scalability, efficiency, and reliability for online learning and assessment platforms. This study is limited to a single hardware configuration and a specific online exam workload; broader validation on different infrastructures and usage patterns, as well as further domain-driven refinement of assessment services for personalization, fraud detection, and real-time analytics, is left for future work.

REFERENCES

[1] Education First, “EF English Proficiency Index: A Ranking of 123 Countries and Regions by English Skills (2025),” 2025. Accessed: Jan. 13, 2026. [Online]. Available: www.ef.com/epi
[2] World Economic Forum, Future of Jobs Report 2025, vol. January 2025. World Economic Forum, 2025. [Online]. Available: www.weforum.org
[3] I. Fadhil, D. B. T. Riyadi, D. Herumurti, and I. Kuswardayan, “Development English Qualification Test Try Out Website for non-ESL Speaker,” in 2025 International Conference on Advancement in Data Science, E-learning and Information System (ICADEIS), IEEE, Feb. 2025, pp. 1–6. doi: 10.1109/ICADEIS65852.2025.10933193.
[4] Y. Shulin and H. Jieping, “Design and Implementation of Smart Teaching System Based on Microservice Architecture,” 2022. doi: 10.1109/icpeca53709.2022.9718846.
[5] H. M. M. Ramirez and C. V. Hurtado, “Work in Progress: Design and implementation of a microservices architecture for project-based learning of software engineering patterns,” in IEEE Global Engineering Education Conference, EDUCON, 2023. doi: 10.1109/EDUCON54358.2023.10125224.
[6] A. Aggarwal and V. Singh, “Migration aspects from monolith to distributed systems using software code build and deployment time and latency perspective,” TELKOMNIKA (Telecommunication Computing Electronics and Control), vol. 22, no. 4, p. 854, Aug. 2024, doi: 10.12928/telkomnika.v22i4.25655.
[7] Y. Romani, O. Tibermacine, and C. Tibermacine, “Towards Migrating Legacy Software Systems to Microservice-based Architectures: a Data-Centric Process for Microservice Identification,” in 2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C), IEEE, Mar. 2022, pp. 15–19. doi: 10.1109/ICSA-C54293.2022.00010.
[8] D. Wolfart et al., “Modernizing Legacy Systems with Microservices: A Roadmap,” in Evaluation and Assessment in Software Engineering, New York, NY, USA: ACM, Jun. 2021, pp. 149–159. doi: 10.1145/3463274.3463334.
[9] K. Tuusjärvi, J. Kasurinen, and S. Hyrynsalmi, “Migrating a Legacy System to a Microservice Architecture,” e-Informatica Software Engineering Journal, vol. 18, no. 1, p. 240104, 2024, doi: 10.37190/e-Inf240104.
[10] R. Darimont, V. Ramon, C. Ponsard, F. Azmali, M. Thauvoye, and H. Bingen, “Towards Incremental Model-Driven Software Modernisation: Feedback from an Industrial Proof of Concept in Railways,” in Proceedings of the 18th International Conference on Software Technologies, SCITEPRESS - Science and Technology Publications, 2023, pp. 652–659. doi: 10.5220/0012135200003538.
[11] N. Almeida, G. Campos, F. Moraes, and F. Affonso, “Modernization of Legacy Systems to Microservice Architecture: A Tertiary Study,” in Proceedings of the 26th International Conference on Enterprise Information Systems, SCITEPRESS - Science and Technology Publications, 2024, pp. 581–592. doi: 10.5220/0012633300003690.

Copyright © 2026 The Author, Page 795
This Journal is licensed under a Creative Commons Attribution 4.0 International License

[12] R. Darimont, V. Ramon, C. Ponsard, F. Azmali, M. Thauvoye, and H. Bingen, "Towards Incremental Model-Driven Software Modernisation: Feedback from an Industrial Proof of Concept in Railways," 2023. doi: 10.5220/0012135200003538.
[13] B. Bamberger and B. Körber, "Migrating Monoliths to Microservices Integrating Robotic Process Automation into the Migration Approach," Journal of Automation, Mobile Robotics and Intelligent Systems, vol. 2022, no. 1, 2022, doi: 10.14313/JAMRIS/1-2022/8.
[14] A. Reis and A. R. Da Silva, "XIS-reverse: A model-driven reverse engineering approach for legacy information systems," in MODELSWARD 2017 - Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development, 2017. doi: 10.5220/0006271501960207.
[15] A. Krause, C. Zirkelbach, W. Hasselbring, S. Lenga, and D. Kroger, "Microservice Decomposition via Static and Dynamic Analysis of the Monolith," in Proceedings - 2020 IEEE International Conference on Software Architecture Companion, ICSA-C 2020, 2020. doi: 10.1109/ICSA-C50368.2020.00011.
[16] M. Moraga and Y. Zhao, "Reverse engineering a legacy software in a complex system: A systems engineering approach," INCOSE International Symposium, vol. 28, no. 1, 2018, doi: 10.1002/j.2334-5837.2018.00546.x.
[17] G. Blinowski, A. Ojdowska, and A. Przybylek, "Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation," IEEE Access, vol. 10, pp. 20357–20374, 2022, doi: 10.1109/ACCESS.2022.3152803.
[18] M. Villamizar et al., "Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS Lambda architectures," Service Oriented Computing and Applications, vol. 11, no. 2, 2017, doi: 10.1007/s11761-017-0208-y.
[19] A. Saransig and F. Tapia, "Performance analysis of monolithic and micro service architectures – containers technology," in Advances in Intelligent Systems and Computing, 2019. doi: 10.1007/978-3-030-01171-0_25.
[20] A. Ruiz-Zafra, J. Pigueiras-Del-Real, M. Noguera, L. Chung, D. G. Barres, and K. Benghazi, "Servitization of Customized 3D Assets and Performance Comparison of Services and Microservices Implementations," IEEE Trans Serv Comput, vol. 17, no. 1, 2024, doi: 10.1109/TSC.2023.3339991.
[21] Y. Te Wang, S. P. Ma, Y. J. Lai, and Y. C. Liang, "Qualitative and quantitative comparison of Spring Cloud and Kubernetes in migrating from a monolithic to a microservice architecture," Service Oriented Computing and Applications, vol. 17, no. 3, 2023, doi: 10.1007/s11761-023-00364-w.
[22] T. Matias, F. F. Correia, J. Fritzsch, J. Bogner, H. S. Ferreira, and A. Restivo, "Determining microservice boundaries: A case study using static and dynamic software analysis," in Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 2020. doi: 10.1007/978-3-030-58923-3_21.
[23] D. Wolfart et al., "Modernizing legacy systems with microservices: A roadmap," in ACM International Conference Proceeding Series, 2021. doi: 10.1145/3463274.3463334.
[24] D. Wolfart et al., "Towards a Process for Migrating Legacy Systems into Microservice Architectural Style," 2021. doi: 10.5753/eres.2020.13736.
[25] H. Knoche and W. Hasselbring, "Using Microservices for Legacy Software Modernization," IEEE Softw, vol. 35, no. 3, 2018, doi: 10.1109/MS.2018.2141035.
[26] H. H. S. da Silva, G. de F. Carneiro, and M. P. Monteiro, "An experience report from the migration of legacy software systems to microservice based architecture," in Advances in Intelligent Systems and Computing, 2019. doi: 10.1007/978-3-030-14070-0_26.
[27] D. Faustino, N. Gonçalves, M. Portela, and A. Rito Silva, "Stepwise migration of a monolith to a microservice architecture: Performance and migration effort evaluation," Performance Evaluation, vol. 164, 2024, doi: 10.1016/j.peva.2024.102411.
[28] N. Bjørndal, M. Mazzara, A. Bucchiarone, N. Dragoni, and S. Dustdar, "Migration from Monolith to Microservices: Benchmarking a Case Study.," The Journal of Object Technology, vol. 20, no. 2, 2021, doi: 10.5381/jot.2021.20.2.a3.
[29] T. Prasandy, Titan, D. F. Murad, and T. Darwis, "Migrating application from monolith to microservices," in Proceedings of 2020 International Conference on Information Management and Technology, ICIMTech 2020, 2020. doi: 10.1109/ICIMTech50083.2020.9211252.
[30] M. Cojocaru, A. Uta, and A.-M. Oprescu, "MicroValid: A Validation Framework for Automatically Decomposed Microservices," in 2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), 2019, pp. 78–86. doi: 10.1109/CloudCom.2019.00023.
[31] ISO/IEC 25010, "Iso 25010 Web," 2019.
[32] N. Ratnaduhita, Y. Sudianto, and A. Kusumawati, "ISO/IEC 25010 : Analisis Kualitas Sistem E-learning sebagai Media Pembelajaran Online," Journal of Information System,Graphics, Hospitality and Technology, vol. 5, no. 1, pp. 8–20, Mar. 2023, doi: 10.37823/insight.v5i1.302.
[33] M. Haoues, A. Sellami, H. Ben-Abdallah, and L. Cheikhi, "A guideline for software architecture selection based on ISO 25010 quality related characteristics," International Journal of System Assurance Engineering and Management, vol. 8, 2017, doi: 10.1007/s13198-016-0546-8.
[34] M. Jayasinghe, J. Chathurangani, G. Kuruppu, P. Tennage, and S. Perera, "An Analysis of Throughput and Latency Behaviours Under Microservice Decomposition," 2020, pp. 53–69. doi: 10.1007/978-3-030-50578-3_5.
[35] R. V. Prasad, S. Ganipaneni, S. Nadukuru3, O. Goel, N. Singh, and Prof. (Dr.) A. Jain, "Event-Driven Systems: Reducing Latency in Distributed Architectures," Journal of Quantum Science and Technology, vol. 1, no. 3, Aug. 2024, doi: 10.63345/jqst.v1i3.87.
[36] A. Esteves and J. Fernandes, "Improving the Latency of Python-based Web Applications," in Proceedings of the 15th International Conference on Web Information Systems and Technologies, SCITEPRESS - Science and Technology Publications, 2019, pp. 193–201. doi: 10.5220/0007959401930201.
[37] N. S. ElGheriani, "Microservices vs. Monolithic Architectures," Al-Mansour Journal, vol. 37, no. 1, 2022.
[38] O. Al-Debagy and P. Martinek, "A Comparative Review of Microservices and Monolithic Architectures," in 18th IEEE International Symposium on Computational Intelligence and Informatics, CINTI 2018 - Proceedings, 2018. doi: 10.1109/CINTI.2018.8928192.

This Journal is licensed under a Creative Commons Attribution 4.0 International License

Journal of Information System Research (JOSH)
Volume 7, No. 3, April 2026, pp 784−797
ISSN 2686-228X (media online)
https://ejurnal.seminar-id.com/index.php/josh/
DOI 10.47065/josh.v7i3.9128

[39] S. Chaturvedi, S. Das, S. Sahana, T. L. Borges, and A. Ghosh, "Performance Measurement and Analysis of Partial Cloud-Dependent Application Hosting," 2024, pp. 437–448. doi: 10.1007/978-981-99-8661-3_32.
[40] M. D. Marieska, Arya Yunanta, Harisatul Aulia, Alvi Syahrini Utami, and Muhammad Qurhanul Rizqie, "Performance Comparison of Monolithic and Microservices Architectures in Handling High-Volume Transactions," Jurnal RESTI (Rekayasa Sistem dan Teknologi Informasi), vol. 9, no. 3, pp. 594–600, Jun. 2025, doi: 10.29207/resti.v9i3.6183.

This Journal is licensed under a Creative Commons Attribution 4.0 International License