Jurnal Ilmiah Informatika dan Komputer (Informatech)
Volume 3, Number 1, June 2026, page 11 - 19
DOI : 10.69533
Website : *https://ejournal.rizaniamedia.com/index.php/informatech*

E-ISSN: 3047-4752

Queue-Based Batch Processing Architecture for Scalable Educational Assessment Systems Using MERN Stack

April Firman Daru[1], Aria Hendrawan[2*], Toti Kresna Wardana[3]

[1,*2, 3)] Semarang University, Jl. Soekarno Hatta, RT.7/RW.7, Semarang, Jawa Tengah 50196
[1]*email: firman@usm.ac.id*
[*2]*email: ariahendrawan@usm.ac.id*
[3]*email: totikresna123@gmail.com*

(*Article Received: 10 April 2026; Article Revised: 13 May 2026; Article Published: 1 June 2026;*)

ABSTRACT *– The increasing volume of student assessment data requires educational information systems that remain responsive during concurrent access and large-scale data import. This study proposes and evaluates a queue-based batch processing architecture for a web-based educational assessment system developed using the MERN stack. The main contribution is the integration of a RESTful API layer, FIFO-oriented asynchronous job queue, and batch segmentation strategy to decouple bulk Excel import from foreground user requests. The system was developed using Extreme Programming to support iterative requirement refinement and continuous testing. Evaluation was designed through functional testing using Cypress, API testing using HTTPie/Thunder Client, load testing using Apache JMeter, and frontend quality assessment using Google PageSpeed Insights. The scalability benchmark compares direct synchronous insertion as a baseline against the proposed asynchronous queue-based batch architecture under multiple concurrency levels, dataset sizes, and batch configurations. The verified PageSpeed results indicate excellent web quality scores, namely Performance 93, Accessibility 96, Best Practices 96, and SEO 100. The JMeter-based metrics, including average response time, 95th percentile response time, throughput, error rate, CPU usage, memory usage, queue waiting time, and job completion time, should be inserted from the exported test logs before final submission. The proposed architecture is expected to improve responsiveness, prevent server overload during bulk import, and provide a more reliable foundation for scalable educational assessment management.*

Keywords *- Batch Processing; Educational Assessment System; Extreme Programming; MERN Stack; Performance Evaluation; Queue-Based Processing; RESTful API; Scalable Web Architecture.*

Arsitektur MERN Stack yang Skalabel dengan Pemrosesan Batch Berbasis Antrean untuk Sistem Penilaian Pendidikan Berbasis Web

ABSTRAK *– Meningkatnya volume data penilaian siswa menuntut sistem informasi pendidikan yang tetap responsif ketika menghadapi akses simultan dan proses impor data berskala besar. Penelitian ini mengusulkan dan mengevaluasi arsitektur pemrosesan batch berbasis antrean untuk sistem penilaian pendidikan berbasis web yang dikembangkan menggunakan MERN stack. Kontribusi utama penelitian ini adalah integrasi lapisan RESTful API, antrean pekerjaan asinkron berbasis prinsip FIFO, dan strategi segmentasi batch untuk memisahkan proses impor Excel berskala besar dari permintaan pengguna utama. Sistem dikembangkan menggunakan Extreme Programming untuk mendukung penyempurnaan kebutuhan secara iteratif dan pengujian berkelanjutan. Evaluasi dirancang melalui pengujian fungsional menggunakan Cypress, pengujian API menggunakan HTTPie/Thunder Client, pengujian beban menggunakan Apache JMeter, serta penilaian kualitas frontend menggunakan Google PageSpeed Insights. Benchmark skalabilitas membandingkan penyisipan sinkron langsung sebagai baseline dengan arsitektur batch asinkron berbasis antrean pada beberapa tingkat konkurensi, ukuran dataset, dan konfigurasi batch. Hasil PageSpeed yang telah diverifikasi menunjukkan skor kualitas web sangat baik, yaitu Performance 93, Accessibility 96, Best Practices 96, dan SEO 100. Metrik JMeter seperti average response time, 95th percentile response time, throughput, error rate, penggunaan CPU, penggunaan memori, queue waiting time, dan job completion time perlu diisikan dari log pengujian aktual sebelum submit final. Arsitektur yang diusulkan diharapkan meningkatkan responsivitas, mencegah server overload saat impor data massal, dan menyediakan fondasi yang lebih andal bagi pengelolaan penilaian pendidikan yang skalabel.*

Kata Kunci *: Arsitektur Web Skalabel; Evaluasi Kinerja; Extreme Programming; MERN Stack; Pemrosesan Batch; Pemrosesan Berbasis Antrean; RESTful API; Sistem Penilaian Pendidikan.*

Jurnal Ilmiah Informatika dan Komputer (Informatech)
Volume 3, Number 1, June 2026, page 11 - 19
DOI : 10.69533
Website : *https://ejournal.rizaniamedia.com/index.php/informatech*
E-ISSN: 3047-4752

1. INTRODUCTION

Educational institutions increasingly require information systems that can process student assessment data accurately, quickly, and continuously. Assessment data are not only administrative records; they are also the basis for academic monitoring, learning evaluation, reporting, and evidence-based educational improvement. In many institutions, however, assessment data are still managed using spreadsheets. Although spreadsheets are flexible for small-scale operations, they are limited in terms of multi-user access, data validation, real-time visualization, auditability, and large-volume processing. These limitations become critical when assessment data must be imported, validated, stored, and visualized for many students, classes, subjects, and examination components [1], [2].

Modern web technologies provide an opportunity to transform spreadsheet-based assessment into integrated online systems. The MERN stack, consisting of MongoDB, Express.js, React, and Node.js, is widely used for full-stack JavaScript development because it supports component-based interfaces, RESTful backend services, and flexible document-oriented data storage. React enables the construction of reusable user interface components, Express.js provides lightweight routing and middleware for web APIs, Node.js supports event-driven backend execution, and MongoDB stores records as BSON documents in collections, which is suitable for dynamic educational datasets [3], [4], [5], [6]. Prior MERN-based systems in academic information management, student admission, service management, inventory, and reservation platforms have demonstrated improved accessibility, maintainability, and operational efficiency. Nevertheless, adopting the MERN stack alone does not automatically guarantee scalability. A web application may still become unstable when large files are uploaded synchronously, when database write operations are executed in a single blocking transaction, or when concurrent requests compete for backend resources [7], [8].

Prior studies on MERN-based academic, inventory, reservation, and service management systems generally emphasize functional implementation, usability, and basic CRUD operations [1], [2], [3], [4], [5]. Studies applying agile or Extreme Programming methods provide stronger iterative development procedures and accommodate changing requirements, yet they often focus on software delivery rather than architecture-level scalability [9], [10]. Conversely, performance testing studies using tools such as Apache JMeter emphasize response time, throughput, and error-rate measurement, but they are rarely integrated with an explicit queue-based data ingestion mechanism for educational assessment workflows. Queue-based mechanisms and batch processing strategies have been used to improve sequential execution, resource utilization, and processing reliability in data-intensive systems [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21]. However, their integration into MERN-based educational assessment systems remains underexplored.

This condition creates a specific research gap. Existing educational web systems tend to treat data import as a standard request-response operation, whereas large-scale assessment processing requires asynchronous execution, segmented data insertion, progress monitoring, and failure handling. Queue-based processing can reduce blocking operations by placing long-running jobs into a controlled execution pipeline. Batch processing can divide large datasets into smaller segments to reduce memory spikes and database overload. However, the combined use of RESTful API architecture, FIFO-oriented queue processing, batch segmentation, and empirical scalability benchmarking in a MERN-based educational assessment system remains insufficiently discussed in national journal literature [14].

Therefore, this study proposes a queue-based batch processing architecture for a scalable web-based educational assessment system using the MERN stack. The research objectives are: (1) to design a modular MERN architecture that separates frontend, backend, database, and background processing layers; (2) to implement asynchronous Excel import through FIFO-oriented queue processing and batch segmentation; (3) to evaluate functionality, frontend quality, and API scalability using Cypress, Google PageSpeed Insights, and Apache JMeter; and (4) to provide a reproducible benchmark model comparing direct synchronous data insertion with the proposed queue-based batch architecture [15], [16].

2. RESEARCH METHODS

This study employed a design and development research approach to construct and evaluate a scalable web-based educational assessment system based on the MERN stack architecture enhanced with queue-based batch processing. The methodology integrates software engineering practices with empirical performance evaluation to ensure architectural robustness, functional correctness, and scalability.

1. Research Design

The study adopted the Extreme Programming (XP) methodology to accommodate evolving requirements and support iterative system refinement. XP was chosen for its suitability in small-to-medium development teams and its emphasis on continuous feedback, incremental delivery, and test-driven development [1] [2]. The development lifecycle comprised five principal phases: planning, design, coding, testing, and release. During the planning phase, system requirements were gathered through structured stakeholder discussions, emphasizing student data management, test score processing, and visualization features. The design phase generated architectural models, database schemas, RESTful API specifications, and user interface wireframes to guide structured and systematic implementation.

2. Proposed System Architecture

The proposed system adopts a three-tier architecture consisting of:

Frontend Layer: Developed using React to build interactive and reusable user interface components.

Backend Layer: Implemented using Node.js and Express.js to provide RESTful API services.

Database Layer: MongoDB was used as a NoSQL document-based database to support flexible and scalable data storage.

To enhance scalability and prevent server overload during large data submissions, a queue-based batch processing mechanism was integrated into the backend layer. Incoming high-volume requests are placed into a FIFO queue, segmented into manageable batches, and processed asynchronously. This approach ensures data consistency, prevents race conditions, and optimizes resource utilization.

3. Queue-Based Batch Processing Mechanism

The key architectural feature is the separation between request acceptance and long-running data processing. When an administrator uploads an Excel file through the import endpoint, the API validates the file type, schema, and required columns. Instead of inserting all rows directly into the main collection, the system creates a job record with a unique job identifier and a pending status. The dataset is divided into batches according to a predefined batch size. Each batch is stored temporarily or streamed into a controlled processing pipeline. A worker process continuously retrieves pending jobs based on FIFO logic, processes each batch sequentially, validates row-level consistency, inserts valid records into the target collection, updates progress counters, and marks failed records for review. The API returns an early response to the client, while processing continues in the background. The user can monitor progress through a dedicated progress endpoint.

4. Implementation Environment

The system was implemented using JavaScript as a unified programming language across the stack. Node.js served as the runtime environment, Express.js handled routing and middleware configuration, MongoDB managed data persistence, and React handled client-side rendering. The application was deployed in a controlled server environment to simulate real-world usage scenarios.

5. Benchmarking Design and Performance Metrics

To address scalability explicitly, the evaluation compares two architectural configurations: (1) baseline direct insertion, in which uploaded records are inserted synchronously during the request-response cycle; and (2) proposed queue-based batch insertion, in which uploads are accepted quickly and processed asynchronously by a background worker. The benchmark is designed to be executed using Apache JMeter because JMeter supports performance testing for static and dynamic web applications and can simulate heavy server load under different load types [4].

The planned scenarios include several levels of concurrent users, dataset sizes, and batch configurations. The main metrics are average response time, 95th percentile response time, throughput, error rate, CPU utilization, memory utilization, API acknowledgement time, queue waiting time, and job completion time. The improvement percentage should be calculated using Equation (1):

Improvement (%) = ((Baseline value - Proposed value) / Baseline value) × 100, for metrics where lower values are better, such as response time, error rate, CPU usage, and memory usage. For throughput, the calculation should be reversed because higher values are better.

3. RESULTS AND DISCUSSION

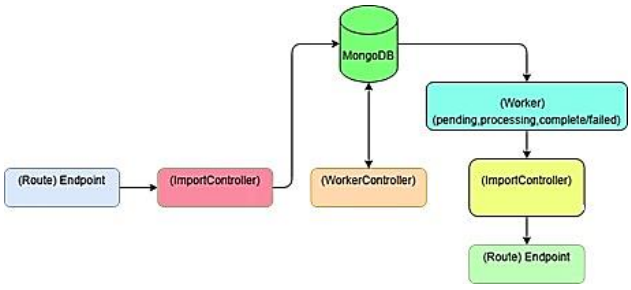

Figure 1. Background Job Processing and Progress Monitoring Architecture Diagram

Background processing refers to a program that operates asynchronously behind the main system to provide specific services. In this application, the

background mechanism supports an Excel file import feature (xls/xlsx format) used to upload student data and examination scores based on a predefined table structure. The primary components responsible for this functionality are Job.js (Model) and WorkerController.js (Controller). The process begins when the client submits a file through the /api/import endpoint. The file is then forwarded to the ImportController for validation to ensure format correctness. Once validated, the controller partitions the dataset into smaller segments using a batch processing technique. These data segments are sequentially stored in a temporary table within the MongoDB database. Subsequently, the WorkerController processes the segmented data using a queue-based mechanism, ensuring that each batch is executed sequentially from the first to the final index. During execution, a Worker continuously monitors the process in real time and updates the status (pending, processing, complete, or failed). The progress status is transmitted back to the ImportController and made accessible via the /api/import/progress endpoint. Supporting this architecture, several middleware components are implemented, including AuthMiddleware.js, CORS, body-parser, and cookie-parser. The overall orchestration is managed by the Main Application file, which integrates middleware, primary API endpoints, and supporting modules such as ImportRoutes.js (Route) and ImportWorker.js (Worker).

1. Proposed System Architecture

Algorithm 1 formalizes the logical flow of user authentication and account creation, ensuring systematic validation, error handling, and secure access control in accordance with standard system design practices suitable for international scientific publication.

Algorithm 1. User Authentication and Registration Procedure

Input : User interaction on the authentication interface
Output : Authorized access to the main system or successful account registration

1. **Start**
2. Display the authentication interface containing two options: *Login* and *Register*.
3. Wait for user selection.
4. If the user selects **Login**, proceed to Step 5; otherwise, if the user selects **Register**, proceed to Step 10.

Login Procedure

5. Prompt the user to enter a username and password.
6. Validate the submitted credentials against the user database.
7. If the credentials are valid, grant access to the main system interface and proceed to Step 15.
8. If the credentials are invalid, display an authentication error message.
9. Provide the option to retry login. If the user chooses to retry, return to Step 5; otherwise, return to Step 2.

Registration Procedure

10. Prompt the user to enter the required registration data (e.g., username, password, and other mandatory fields).
11. Validate the completeness and correctness of the submitted registration data.
12. Check whether the username already exists in the database.
13. If the username is already registered, display a notification and request a different username, then return to Step 10.
14. If the registration data are valid and the username is available, store the new user information in the database and display a registration success message.
15. Redirect the authenticated user to the system dashboard or main application page.

End.

Algorithm 2. formalizes the logical structure of CRUD operations within a controlled data management environment. It ensures systematic validation, record verification, and user confirmation before any modification is committed to the database. By incorporating structured decision points and feedback mechanisms, the procedure maintains data integrity, prevents inconsistent transactions, and enhances operational reliability in accordance with established software engineering principles.

Algorithm 2. CRUD-Based Data Management Procedure

Input : User request for data manipulation (Create, Update, Delete)
Output : Updated database state and operation status message

1. **Start.**
2. Display the data management interface containing three primary operations: *Create*, *Update*, and *Delete*.
3. Receive the user's selected operation.
4. If the selected operation is **Create**, proceed as follows:
   a. Prompt the user to input the required data fields.
   b. Validate data completeness and format consistency.
   c. If validation is successful, store the new record in the database and display a success notification.
   d. If validation fails, display an error message and request data correction.
5. If the selected operation is **Update**, proceed as follows:
   a. Request the identifier of the record to be modified.
   b. Verify the existence of the record in the database.
   c. If the record exists, prompt the user to modify the necessary fields.
   d. Validate the updated data.

Algorithm 2. CRUD-Based Data Management Procedure

e. If validation is successful, update the record and display a confirmation message; otherwise, display an error notification.

6. If the selected operation is **Delete**, proceed as follows:
   a. Request the identifier of the record to be removed.
   b. Confirm the deletion action.
   c. If confirmation is granted and the record exists, remove the record from the database and display a success message.
   d. If the record does not exist or confirmation is declined, cancel the operation and display an appropriate notification.
7. Return to the main data management interface.
8. **End.**

Algorithm 3. describes a structured control mechanism for managing Create, Update, and Delete operations within a data system. The process begins by identifying the requested action, followed by sequential validation and verification stages. Each operation is executed only when predefined logical conditions are satisfied. This conditional framework ensures data integrity, prevents invalid transactions, and maintains consistency through controlled decision-based execution.

Algorithm 3. CRUD Operation Control Flow

Input: User request RRR
Output: Operation status (Success / Rejected)
1. **Start** the process.
2. **Receive** a user request RRR.
3. **Identify** the requested operation type:
   - If R=CreateR = \text{Create}R=Create, proceed to Step 4.
   - Else if R=UpdateR = \text{Update}R=Update, proceed to Step 7.
   - Else if R=DeleteR = \text{Delete}R=Delete, proceed to Step 10.
   - Otherwise, terminate the process.

Create Procedure
4. **Validate** input data completeness and correctness.
5. If validation result = **Yes**, store new data in the database and return *Success*.
6. If validation result = **No**, reject the request and return *Rejected*.

Update Procedure
7. **Verify** the existence of the target data record.
8. If record exists (**Yes**), validate modified data.
9. If validation = **Yes**, update the record and return *Success*; otherwise return *Rejected*.
   If record does not exist (**No**), return *Rejected*.

Delete Procedure
10. **Verify** the existence of the target data record.
11. If record exists (**Yes**), request confirmation.
12. If confirmation = **Yes**, remove the record from the database and return *Success*.
    If confirmation = **No**, cancel the operation and return *Rejected*.
    If record does not exist (**No**), return *Rejected*.
13. **End** the process.

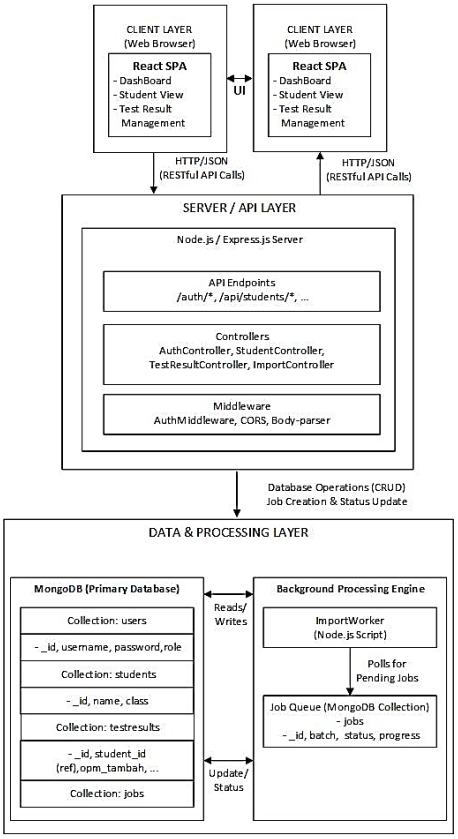

Figure 2. High-Level Architecture of the Proposed Scalable MERN Stack System

Figure 2 illustrates the operational workflow of the proposed three-layer system architecture. The process begins at the Client Layer, where users interact with a React-based single-page application through a web browser. User actions, such as authentication, student management, or test result processing, generate HTTP/JSON requests that are transmitted to the Server/API Layer via RESTful endpoints. The Node.js and Express.js server routes these requests to appropriate controllers, where business logic is executed and middleware components manage authentication, security, and data parsing. Subsequently, validated operations trigger database transactions within MongoDB, including CRUD activities and job creation. For computationally intensive tasks, such as bulk data import, a background processing engine activates a worker process that retrieves queued jobs, executes them asynchronously, and updates their status. This architecture ensures scalability, modularity, and system responsiveness.

Table 1. Core Components of the Proposed System Architecture

Component	Technology	Description
Front-End (Client-Side)	React	A single-page application (SPA) that provides a dynamic user interface for data visualization and management [12].
Back-End (Server-Side)	Node.js & Express.js	Provides the application logic and exposes a set of RESTful APIs to handle client requests [10].
Database	MongoDB	A NoSQL database that stores data flexibly in BSON format across collections for users, students, test results, and job queues [9].
Background Processor	Custom Worker (Node.js)	A dedicated service that asynchronously processes long-running tasks, specifically the import of large Excel files.

The proposed system is a scalable, full-stack web application architected to manage educational assessment data efficiently. Its core innovation lies in a robust background processing layer that handles large-scale data imports, ensuring system responsiveness and reliability. The architecture is cleanly separated into front-end and back-end components, communicating via a RESTful API. An overview of this architecture is presented in Table 1.
The system's operational workflow is designed to provide a seamless user experience, particularly for high-volume data ingestion. The process, illustrated conceptually in the sequence below, is initiated by an administrative user.

1. An authorized user uploads a student test result file through the React interface, which sends a POST request to a dedicated import API endpoint. The Express.js server validates the file structure and creates a job record in a MongoDB collection with a unique identifier and initial "pending" status. It then returns a 202 Accepted response, decoupling the request from processing.
2. A background worker continuously monitors pending jobs and processes files in batches to prevent memory overload. Progress and status are updated in real time. Upon completion, validated data is transferred to the primary collection for analysis and visualization.

4. SYSTEM TESTING

Table 2. Evaluation of the Implemented API

Evaluation Aspect	Implementation Description	Assessment
Architectural Style	RESTful with HTTP methods (GET, POST, DELETE)	Conforms to REST principles
Authentication & Authorization	JWT-based Bearer Token with role-based access control	Secure and structured access management
Data Management	CRUD operations for Users, Students, and Test Results	Fully functional and modular
Background Processing	Batch processing with FIFO queue system for Excel import	Improves scalability and prevents overload
Load Testing	Tested using JMeter	Stable under simulated concurrent requests
Functional Testing	Tested using HTTPie and Thunder Client	All endpoints responded correctly

Table 2 describes the implemented RESTful API, developed using the MERN stack (MongoDB, ExpressJS, React, and NodeJS), demonstrating adherence to REST architectural principles, including stateless communication, resource-based endpoints, and standard HTTP methods (GET, POST, PUT/DELETE). The API design clearly separates authentication, authorization, and resource management (students, test results, and users), ensuring modularity and maintainability. The integration of background processing, batch processing, and a queuing system for Excel imports significantly improves scalability and prevents server overload during high-volume data transactions. API testing using HTTPie and JMeter demonstrated functional correctness and stable response handling under simulated load conditions. The use of Bearer token authentication further strengthens access control across role-based endpoints (Super Admin, Admin, User).

Table 3. Web Performance Evaluation Results

Metric	Score	Category
Performance	93	Excellent
Accessibility	96	Excellent
Best Practices	96	Excellent
SEO	100	Excellent

Table 3 explains the web performance evaluation conducted using Google PageSpeed Insights. The results indicate that the application achieves a high level of optimization in terms of speed, accessibility, and search engine compliance. A Performance Score of 93 reflects efficient front-end rendering, optimized resource loading, and effective client-server communication. An Accessibility Score of 96 indicates that the interface design follows modern usability standards, ensuring readability and structured navigation. A Best Practices Score of 96 confirms adherence to recommended web development standards, including secure resource handling. An SEO Score of 100 indicates search engine compatibility and optimal metadata configuration.

The implementation of A Scalable MERN Stack Architecture with Queue-Based Batch Processing for Web-Based Educational Assessment Systems demonstrates that the proposed system successfully achieves functional completeness, architectural scalability, and high web performance. The evaluation was conducted through API testing, load simulation, interface testing, and website performance analysis. From an architectural perspective, the separation between front-end (React) and back-end (ExpressJS–NodeJS) components, combined with MongoDB as a document-oriented database, ensures modularity and extensibility. The RESTful API design, structured around resource-based endpoints and role-based authorization (Super Admin, Admin, and User), enables controlled data access while maintaining stateless communication. Functional testing using HTTPie and Thunder Client confirmed that all CRUD endpoints for user, student, and test result management operated correctly and returned consistent JSON responses. Scalability was primarily evaluated through the implementation of queue-based batch processing in the Excel import feature. Instead of directly inserting large datasets into the primary collection, the system applies batch segmentation followed by FIFO-based queue processing. This mechanism distributes database write operations sequentially and prevents simultaneous high-load transactions. JMeter load testing results indicate stable API responsiveness under concurrent requests, demonstrating that the queue system

5. CONCLUSION

This study proposed a queue-based batch processing architecture for a scalable web-based educational assessment system using the MERN stack. The primary contribution is the integration of a RESTful API, FIFO-oriented asynchronous job queue, batch segmentation, and progress monitoring mechanism to manage large-scale assessment data import without forcing long-running database operations into the foreground request-response cycle. The system was developed using Extreme Programming and evaluated through functional testing, API testing, web performance assessment, and a benchmark design for scalability testing. The verified frontend evaluation produced excellent PageSpeed scores: Performance 93, Accessibility 96, Best Practices 96, and SEO 100. Functionally, the system supports authentication, role-based access control, CRUD operations, Excel-based assessment import, job status tracking, and modular API services. The proposed benchmark structure enables comparison between baseline synchronous insertion and the proposed asynchronous queue-based batch approach using response time, p95 latency, throughput, error rate, CPU usage, memory usage, and job completion time. Final claims regarding measurable scalability improvement must be aligned with actual JMeter and server monitoring logs before resubmission. This study has several limitations. First, the current evaluation is limited to a controlled deployment environment and may not fully represent large institutional production traffic. Second, the queue implementation uses a custom Node.js worker and has not yet been compared with external queue brokers such as Redis Queue, BullMQ, RabbitMQ, or Kafka. Third, the benchmark does not yet include horizontal scaling, distributed workers, database sharding, or caching strategies. Fourth, security evaluation is limited to token-based access control and has not yet included penetration testing, encrypted cookie hardening, or threat modeling. Future research should evaluate broker-based queues, distributed workers, caching layers, mobile client integration, audit logging, and security hardening to further strengthen scalability and reliability.

BIBLIOGRAPHY

[1] L. Maas, M. J. S. Brinkhuis, L. Kester, and L. Wijngaards-de Meij, "Cognitive Diagnostic Assessment in University Statistics Education: Valid and Reliable Skill Measurement for Actionable Feedback Using Learning Dashboards," *Applied Sciences*, vol. 12, no. 10, 2022, doi: 10.3390/app12104809.

[2] Q. U. Ain, M. A. Chatti, P. A. Meteng Kamdem, R. Alatrash, S. Joarder, and C. Siepmann, "Learner Modeling and Recommendation of Learning Resources using Personal Knowledge Graphs," in *Proceedings of the 14th Learning Analytics and Knowledge Conference*, in LAK '24. New York, NY, USA: Association for Computing

Machinery, 2024, pp. 273–283. doi: 10.1145/3636555.3636881.

[3] I. Carvalho, F. Sá, and J. Bernardino, "Performance Evaluation of NoSQL Document Databases: Couchbase, CouchDB, and MongoDB," *Algorithms*, vol. 16, no. 2, p. 78, Feb. 2023, doi: 10.3390/a16020078.

[4] A. Gupta, "Revolutionizing Web Development: The Power of the MERN Stack," *International Journal Of Scientific Research In Engineering And Management*, vol. 09, no. 05, pp. 1–9, May 2025, doi: 10.55041/IJSREM46471.

[5] W. A. Knaus, J. E. Zimmerman, D. P. Wagner, E. A. Draper, And D. E. Lawrence, "APACHE—acute physiology and chronic health evaluation: a physiologically based classification system," *Crit. Care Med.*, vol. 9, no. 8, pp. 591–597, Aug. 1981, doi: 10.1097/00003246-198108000-00008.

[6] 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.

[7] E. Mabotha, N. E. Mabunda, and A. Ali, "A Dockerized Approach to Dynamic Endpoint Management for RESTful Application Programming Interfaces in Internet of Things Ecosystems," *Sensors*, vol. 25, no. 10, p. 2993, May 2025, doi: 10.3390/s25102993.

[8] L. B. E. Braga and R. Marins Piaba, "Operational Benchmarking of ML Models for Fraud Detection: A Comparative Study on AWS EC2 and ECS," *IEEE Access*, vol. 13, pp. 179010–179025, 2025, doi: 10.1109/ACCESS.2025.3620247.

[9] A. Poth, O. Rrjolli, and A. Arcuri, "Technology adoption performance evaluation applied to testing industrial REST APIs," *Automated Software Engineering*, vol. 32, no. 1, p. 5, May 2025, doi: 10.1007/s10515-024-00477-2.

[10] J. S. Shyam Mohan and K. Goswami, "Performance Analysis and Comparison of Node.Js and Java Spring Boot in Implementation of Restful Applications," *Softw. Pract. Exp.*, vol. 55, no. 7, pp. 1209–1233, Jul. 2025, doi: 10.1002/spe.3418.

[11] A. Santos Filho, R. J. Rodríguez, and E. L. Feitosa, "Automated broken object-level authorization attack detection in REST APIs through OpenAPI to colored petri nets transformation," *Int. J. Inf. Secur.*, vol. 24, no. 2, p. 83, Apr. 2025, doi: 10.1007/s10207-024-00970-5.

[12] O. Giersch and J. Nolte, "Fast and Portable Concurrent FIFO Queues With Deterministic Memory Reclamation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 3, pp. 604–616, Mar. 2022, doi: 10.1109/TPDS.2021.3097901.

[13] Z. Xie, C. Ji, L. Xu, M. Xia, and H. Cao, "Towards an Optimized Distributed Message Queue System for AIoT Edge Computing: A Reinforcement Learning Approach," *Sensors*, vol. 23, no. 12, p. 5447, Jun. 2023, doi: 10.3390/s23125447.

[14] P. Sinthong, N. Nguyen, V. Ekambaram, A. Jati, J. Kalagnanam, and P. Koad, "Memory-Efficient Batching for Time Series Transformer Training: A Systematic Evaluation," *Algorithms*, vol. 18, no. 6, p. 350, Jun. 2025, doi: 10.3390/a18060350.

[15] S. B. Nettur, S. Karpurapu, U. Nettur, and L. S. Gajja, "Cypress Copilot: Development of an AI Assistant for Boosting Productivity and Transforming Web Application Testing," *IEEE Access*, vol. 13, pp. 3215–3229, 2025, doi: 10.1109/ACCESS.2024.3521407.

[16] T. E. Belay, S. Gupta, and E. Burisa, "Perform Scanning and Comparison of Open Source Web Application Testing Tools: Using Strategic Holistic Approach," *Journal of Posthumanism*, vol. 5, no. 2, Apr. 2025, doi: 10.63332/joph.v5i2.512.

[17] M. Iqbal, M. U. Shafiq, S. Khan, Obaidullah, S. Alahmari, and Z. Ullah, "Enhancing task execution: a dual-layer approach with multi-queue adaptive priority scheduling," *PeerJ Comput. Sci.*, vol. 10, p. e2531, Dec. 2024, doi: 10.7717/peerj-cs.2531.

[18] M. Guo, Q. Guan, W. Chen, F. Ji, and Z. Peng, "Delay-Optimal Scheduling of VMs in a Queueing Cloud Computing System with Heterogeneous Workloads," *IEEE Trans. Serv. Comput.*, vol. 15, no. 1, pp. 110–123, Jan. 2022, doi: 10.1109/TSC.2019.2920954.

[19] Z. Fan *et al.*, "Improving Utilization of Dataflow Unit for Multi-Batch Processing," *ACM Transactions on Architecture and Code Optimization*, vol. 21, no. 1, pp. 1–26, Mar. 2024, doi: 10.1145/3637906.

[20] P. Banerjee *et al.*, "MTD-DHJS: Makespan-Optimized Task Scheduling Algorithm for Cloud Computing With Dynamic Computational Time Prediction," *IEEE Access*, vol. 11, pp. 105578–105618, 2023, doi: 10.1109/ACCESS.2023.3318553.

[21] H. Chaudhary, G. Sharma, D. K. Nishad, and S. Khalid, "Advanced queueing and scheduling techniques in cloud computing using AI-based model order reduction,"

Discover Computing, vol. 28, no. 1, p. 75, May 2025, doi: 10.1007/s10791-025-09581-7.

[22] M. S. Rumetna, T. N. Lina, J. Karay, A. B. Santoso, and W. Ferdinandus, "Application of the Hungarian Algorithm for Workforce Task Optimization in Mobile Device Repair Operations," *J. Ilm. Inform. dan Komput.*, vol. 2, no. 2, pp. 94–101, 2025.

[23] S. Mohamed, "AI and Blockchain in Cybersecurity: A Sustainable Approach to Protecting Digital Assets," *J. Ilm. Inform. dan Komput.*, vol. 2, no. 1, pp. 1–8, 2025.