



Architecture Design for Enhancing Direct Client-to-Microservice Communication in an E-Participation Platform for Local Administrative Organizations in Suratthani Province, Thailand

Kanjana Phuakkhong¹, Chaiwat Phuakkhong², Anak Chaisamut³, and Somkid Sinwittayarak^{1*}

¹ Faculty of Science and Technology, Suratthani Rajabhat University, 84100, Thailand

² College of Politics and Government, Suan Sunandha Rajabhat University, Bangkok, 10300, Thailand

³ Faculty of Humanities and Social Sciences, Suratthani Rajabhat University, 84100, Thailand

* Correspondence: somkid@sru.ac.th

Citation:

Phuakkhong, K.; Phuakkhong, C.; Chaisamut, A.; Sinwittayarak, S. Architecture design for enhancing direct client-to-microservice communication in an e-Participation platform for Local Administrative Organizations in Suratthani Province, Thailand. *ASEAN J. Sci. Tech. Report.* **2026**, 29(9), e262659. <https://doi.org/10.55164/ajstr.v29i9.262659>

Article history:

Received: December 3, 2025

Revised: May 15, 2026

Accepted: June 1, 2026

Available online: July 26, 2026

Publisher's Note:

This article is published and distributed under the terms of the Thaksin University.

Abstract: Public participation is essential for governance in a democratic system. Local Administrative Organizations in Thailand typically provide websites and social media platforms such as Facebook Pages and the Line application for e-Participation. However, there is a lack of specific applications designed for e-Participation that ensure secure, efficient communication and support rapid information exchange. This research aimed to design and evaluate a direct client-to-microservice communication architecture for an e-Participation platform in Local Administrative Organizations in Surat Thani Province, Thailand. We adopted a microservice architecture based on Domain-Driven Design (DDD), along with an API-first design and a database-per-service pattern. A message broker was adopted for asynchronous inter-service communication. We implemented JSON Web Tokens (JWT) with AES-256 encryption for system security and used Apache JMeter for performance testing. The performance testing followed ISO/IEC 25010 standards under varying loads to assess throughput, response time, and resource utilization. The performance testing showed that the system throughput could handle 16.67 requests per second at 5,000 concurrent users, with zero error rate. The time behavior testing demonstrated that the system maintained an average response time of 41-46 milliseconds, well below 1,500 ms. The resource utilization test showed that the CPU peak value fell within the expected range, indicating that the system is highly stable, unaffected by CPU spikes, and efficient in memory use. Overall, the system successfully passed all performance criteria. This architecture enabled real-world usage of scalable, secure, and efficient e-Participation platforms for Local Administrative Organizations.

Keywords: Microservice architecture; direct client-to-microservice; e-Participation; local administrative organizations

1. Introduction

Citizens' electronic participation in the operations of Local Administrative Organizations in Thailand is crucial because these organizations not only implement various local development projects but also serve as the government entities closest to local communities. Local Administrative Organizations in Thailand normally provide websites, Facebook pages, and Line as electronic channels for e-participation. The United Nations (UN) has defined three levels

of electronic participation [1] (1) e-Information, which involves providing valuable information to the public through electronic channel; (2) e-Consultation which involves gathering suggestions and opinions on important issues, and (3) e-Decision making, which involves engaging citizens in the decision-making process to collaboratively create policies that truly reflect the needs and aspirations of the community. Local citizens engage with the websites, Facebook pages, and Line platforms of Local Administrative Organizations at a moderate level, primarily to disseminate information rather than foster meaningful citizen engagement. There is still a lack of specific applications to promote electronic participation that foster meaningful public engagement. Additionally, the growing use of public mobile applications to access services provided by Local Administrative Organizations has become an important trend, leading to increased interaction, monitoring, and follow-up of requested services. Therefore, the application delivered must be high-performing and responsive, ensuring quick access to vital information and services. Applications designed to boost public participation must prioritize performance and security, scalability, coupling, and the capacity to manage high user demand effectively [2]. Microservice architecture (MSA) is a modern software approach that divides functionality into modular, independently deployable services aligned with business domains, and it is widely used in system design today. The system is divided into smaller, independent components, each operating autonomously and focusing on its own functionality and internal communication [3]. MSA is suitable for building an efficient communication platform between citizens and government agencies [4]. Microservice architecture can be used to implement the public participation platforms proposed in this research. This architecture ensures scalability to accommodate high user volumes across multiple local government agencies while enabling seamless version updates and operational modifications without disrupting core data systems.

This research aimed to design a direct client-to-microservice communication architecture for an e-Participation platform tailored to Local Administrative Organizations and to evaluate the performance of data communication within the system. We adopted a direct communication design between clients and microservices to ensure fast, efficient data exchange. To address security concerns, the system was designed to use JWT authentication with AES-256 encryption. Our research contributes to software architecture design by creating a system that delivers both high performance and secure data use. It also enables the public to effectively participate in the operations of local administrative organizations through electronic means. The architectural design of the e-Participation platform, based on a microservice paradigm that prioritizes data integrity and operational efficiency, encompasses several fundamental components, such as component decomposition of services into smaller, autonomous units, which is a pivotal aspect of the design, along with essential elements such as communication protocols, database administration, security measures, and implementation methodologies. We conducted a comprehensive review of the existing literature as follows.

1) E-Participation: Electronic Participation (e-Participation) refers to the use of Information and Communication Technology (ICT) by citizens to engage with government agencies [5]. It is one aspect of e-Government [6] that encompasses information provision, public consultation, and decision-making via digital platforms [7]. Some studies employ microservice architecture in the design of e-Participation platforms, with the scope of each platform and the corresponding architectural approaches summarized in Table 1.

2) Microservice architecture (MSA) is a software architecture design that emphasizes breaking down complex systems into small modules, independently deployable services [10]. It was introduced in 2014 to address the challenges of legacy systems, including increasing complexity, high dependency, and tight coupling. It directly addresses the limitations of monolithic applications by offering a service-oriented architectural design that decomposes system functionality into independent microservices that communicate via lightweight mechanisms such as RESTful APIs [11]. There are several key benefits of microservice architecture, including improved agility, modifiability, scalability, deployability, and fault tolerance, as well as easier system maintenance [2, 12, 14].

Table 1. Scope of e-Participation and corresponding microservice architectural approaches

Scope of e-Participation	The aim of adopting a microservice architecture
The civic engagement platform development in collaboration with DPRD Jawa Barat in Indonesia for reporting public issues, fostering discussions, and coordinating governmental responses. Integrating community feedback mechanisms aims to enhance participatory governance by promoting more structured collaboration between citizens and their government [4].	Microservices architecture was employed to improve backend scalability and maintainability. API testing ensures seamless microservice functionality by validating interface reliability and consistency through stress testing to assess scalability and performance, and by identifying optimization opportunities to enhance the system's responsiveness under fluctuating workloads.
The development of modernizing regulations for Philippine e-government [8].	The system achieves scalability and security while handling high transaction volumes and regional variations through a microservices architecture. It also proves particularly effective, delivering measurable improvements in service delivery and client satisfaction. The comparative study showed that the system enhanced security, transparency, and operational efficiency.
Implementing Interoperability for e-Government Services [9].	API-driven microservices architecture provides an interoperable framework for efficient data sharing across e-government platforms.

3) Domain-Driven Design (DDD) is a model-driven approach in software development that focuses on the use of abstract models as the primary artifacts throughout the software development lifecycle [23]. DDD is a popular methodology for guiding architectural decisions, especially in microservice architecture design. In the DDD approach, the appropriate size of a microservice can range from one per entity to one per bounded context. The application of DDD is a critical success factor in developing microservice-based applications [24], as it enables services to be easily replaceable, encapsulates implementation details, and ensures their use is limited to well-defined smart endpoints [14].

4) Application Programming Interfaces (APIs) are essential for communication in microservice architecture because the system is broken down into small services, each with its own distinct functions. There are two types of communication in microservice architecture design: communication among internal system services and communication between the system and client applications. Effective communication design supports system reliability by ensuring consistency and enabling functions to operate without failure, while also promoting scalability to handle the growing workload from users and transactions and facilitating information exchange. Communication is achieved through Application Programming Interfaces, and the Representational State Transfer (REST) API is one of the most widely adopted mechanisms, enabling convenient interaction between internal system microservices and external applications [25]. There are two techniques to utilize APIs during the design of a communication system. The first approach begins with code writing and implementation, together with an API designed and documented based on the existing code. On the other hand, the API-first design approach emphasizes designing and defining APIs before writing code. It requires creating detailed API specifications using OpenAPI or Swagger. The API-first approach enables clear communication between services and external client applications and promotes easier collaboration among development teams [26-27].

5) Direct client-to-microservice architecture, in this approach, client web applications and mobile applications interact directly with individual microservices via their exposed API endpoints, bypassing the API gateway. A direct client-to-microservice architecture requires clients to handle service discovery, authentication, and communication; therefore, clear, well-defined API contracts are essential [24, 28]. The comparison of the advantages and disadvantages between the direct client-to-microservice and API Gateway pattern is shown in Table 3.

Table 2. Comparison of architectural approaches in software development

Architectural approaches	Monolithic	Centralized REST	Microservice
Characteristics	It is a traditional software design in which the user interface, business logic, and data access are tightly integrated. This approach offers simplicity and straightforward deployment.	It is a client-server architecture that uses Representational State Transfer (REST) to centralize data management and resource logic on the server, while delegating user interface and state management to the client. Therefore, this design can efficiently handle web-based interactions, providing the flexibility and ease of integration required for modern and cross-platform environments.	Breaks down complex systems into small services. Therefore, an application consists of many loosely coupled, independently deployable services.
Scalability	It is typically less scalable due to its tightly integrated nature. Scaling often involves vertical scaling, which can be cost-prohibitive and less efficient as the application grows [15-16].	It offers more straightforward horizontal scaling options, which can be beneficial for handling increased load [15], and can improve scalability compared to monoliths by decoupling some components. However, it may not achieve the same level of scalability as microservices due to potential bottlenecks in the centralized components [17].	It excels at scalability, enabling horizontal scaling in which individual services can be scaled independently based on demand. This architecture supports high-volume processing and is well-suited to large, dynamic systems. [18-20].
Performance	It provides fast response times under low load and with few concurrent users [16], and it is beneficial for applications that do not require frequent updates or high scalability [20].	The resource utilization of this architecture depends on the framework used (e.g., Express.js vs. Laravel)	It is excellent in scalability and fault isolation by allowing independent scaling of services and better handling of high-volume transactions, faster response times, and lower error rates under high-load conditions [21-22]

Table 2. Comparison of architectural approaches in software development (continue)

Architectural approaches	Monolithic	Centralized REST	Microservice
Maintainability	It is relatively easy to maintain; however, as the system grows, it may become increasingly complex, rendering updates and bug fixes more difficult and time-consuming [18-19].	It offers moderate maintainability improvements over monoliths, but still requires careful management of dependencies and interfaces to avoid complexity [17].	It enhances maintainability by allowing independent updates and deployments of services. This modularity supports parallel development and reduces the risk of system-wide failures [15, 20].
Deployment Complexity	It involves the straightforward deployment of the entire application in a single unit. However, this simplicity can lead to longer deployment times and increased downtime during updates [17-18].	Deployment complexity is generally lower than in a microservice architecture.	Deployment is complex and requires sophisticated orchestration frameworks. Each service must be deployed independently [15, 20].

Table 3. The comparison of the advantages and disadvantages between the direct client-to-microservice and API Gateway pattern

	API gateway pattern	Direct the client to the microservice pattern.
Advantages	It uses an API Gateway as a centralized entry point for client requests, thereby simplifying service-to-service communication and improving security enforcement across microservices architectures [29]. It improves scalability by distributing traffic efficiently and supporting load balancing [30]. With centralized service management, security can be improved through authentication and enhanced load balancing. The client can interact more simply through a single entry point for multiple services, thus improving development efficiency [31].	This approach is a simple architecture that is easy to design and implement. It is also enabled for fine-grained interactions, easier debugging, and monitoring, and suits small microservice-based applications. The direct communication between clients and microservices eliminates the need for intermediary layers, potentially reducing latency [31] and improving performance, as clients can quickly access the specific service they need [33]. It improved scalability and enhanced fault isolation. Clients communicate directly with specific microservices, enabling faster response times and reducing latency. Additionally, it facilitates technological flexibility, as different microservices can be developed and deployed independently, enabling quicker updates and innovations [34].
Disadvantages	API Gateways can become a single point of failure if not properly managed [33]. Because of the centralized entry point, it can become a performance bottleneck, especially under high load conditions. Managing an API Gateway involves configuring routing, security policies, and rate limiting, which can become complex and error-prone, especially in large systems with numerous services. This can lead to increased latency, as requests must pass through the gateway before reaching the intended service [32].	The weakness of this approach is increased client-side complexity and an expanded security attack surface. Direct client access to microservices can expose them to security vulnerabilities, as each service must handle authentication, authorization, and other security measures independently. It can lead to increased operational overhead and potential service outages if not managed properly. Direct communication requires clients to handle API versioning and compatibility, which can lead to consumer lock-in and make it difficult to evolve APIs without breaking existing clients [35-36].

6) JSON Web Tokens (JWT) are an open standard (RFC 7519) for safely passing claims in space-constrained environments, providing a compact and self-contained way to securely transmit information between parties as a JSON object [37]. The usefulness of JWT includes authentication and authorization, federated identity, client-side sessions (particularly for stateless sessions), and client-side secrets. JWTs are widely used in security mechanisms for authentication in microservice architectures [38] because they provide a stateless approach that enables efficient, scalable, and secure verification of user identities across distributed services.

7) Advanced Encryption Standard (AES-256) is a specification for encrypting electronic data using a symmetric block cipher. It uses the same key for both encryption and decryption, meaning the sender and receiver share the same secret encryption key. AES-128 bit provides generally adequate security and protection, while AES-256 provides stronger security and supports high performance and efficiency [39].

2. Methodology

We adopted the Design Science Research (DSR) methodology, emphasizing problem identification, definition of solution objective, artifact design and development, demonstration of functionality, artifact evaluation, and communication of results and insights throughout the entire e-Participation Platform development project. In this research paper, we focused on the architectural design stage of the e-Participation platform, as there was a lack of dedicated applications for electronic participation in the Local Administrative Organization that required high performance and fast access to information and services from the organization. The overview diagram of this research methodology is illustrated in Figure 1.

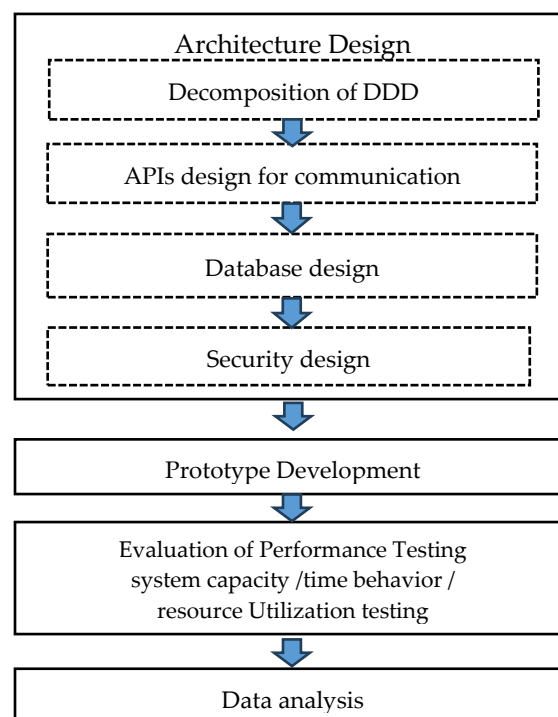


Figure 1. The process of research methodology

To develop the research methodology for enhancing direct client-to-microservice architecture in an E-Participation Platform. We defined the research question (RQ1) as: **Does direct client-to-microservice architecture design meet predefined performance success criteria?** This question aims to evaluate the performance of the proposed system architectures.

2.1 Services Decomposition

In microservice architecture, service decomposition should be based on Domain-Driven Design (DDD) principles. In this research, we focused on the business domain of electronic participation from the public to the local administrative organization. A combination of Domain-Driven Design and business capability analysis is one of the most widely used strategies for decomposing an application into microservices, as DDD helps determine how many microservices a system should comprise. Each service is responsible for a specific task and operates independently. In this research, the strategic design phase of Domain-Driven Design was employed to break down the major tasks of an electronic participation system. This was achieved through data collection, interviews, and consultations with relevant stakeholders and domain experts, including the chief and Local Administrative Organization officers responsible for public information services, as well as local citizens. This process helped identify the system's functional requirements. It revealed that the development of an electronic participation platform, aligned with its key levels of e-information, e-consultation, and e-decision-making, involved three primary contexts: 1) user context, 2) news management context, and 3) main service context, with one supporting context being 4) system notification management context. In the strategic phase of DDD, we design the context map based on the relationships among the contexts. The relationship between the user context and the news management context is a partnership that enables Local Administrative officials to create and publish news for citizens through the application. At the same time, citizens can also submit news reports via the application. The user context and the main service context work together, allowing citizens to access various local government services, including submitting complaints and petitions and using e-services. Similarly, the main service context maintains a partnership with the news management context, as it both creates and consumes news content while providing various services. The notification context is related to the user context, news management context, and main service context as a conformist (CF). It served as an upstream (U) context for sending SMS alerts and status updates for various events. In contrast, the other contexts act as downstream (D) consumers, expected to conform to their outputs. These relationships are illustrated in Figure 2

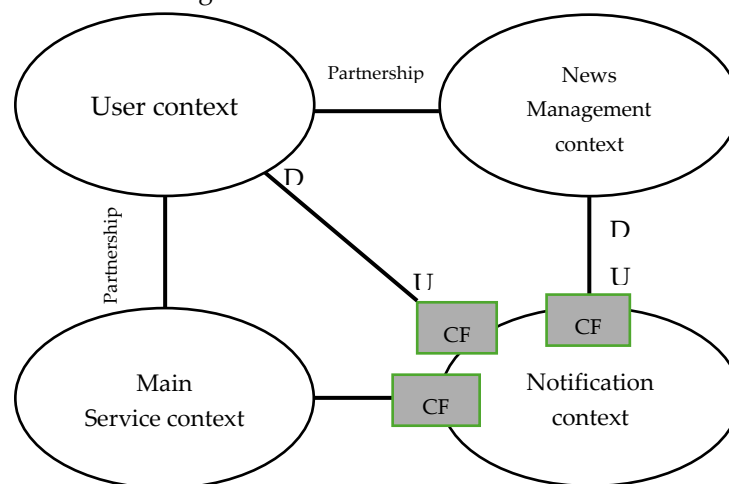


Figure 2. Context mapping of e-Participation Platform

After context mapping, we translated the identified contexts into corresponding services within the microservice architecture. Each sub-task within a context becomes an operational module within a microservice. Interactions between components in related contexts constituted communication between microservices.

2.2 Communication design

In microservice architecture, communication between services and external client applications is essential and depends on the nature and requirements of the task. The e-Participation platform in this study is not highly complex; however, it requires fast and efficient communication between users and the application. Local Administrative Organizations aim to deliver information and services to the public promptly while enabling citizens to access various public services through digital platforms. To achieve simplicity and high performance in external communication, we adopted the Direct Client-to-Microservice communication pattern.

According to the context mapping defined in the DDD design, the user service, news service, and main service were designated as public services, allowing direct access by external users. In contrast, the notification service was implemented as a private service dedicated to internal communication and coordination among microservices, and was therefore not accessible to external users. We adopted an API-first approach at the initial stage of API design to facilitate communication among microservices. The functional roles of each service were defined before developing other application components. A RESTful API was utilized because REST employs the HTTP request–response mechanism. RESTful APIs are widely used and well-suited for both web and mobile communication. They emphasize loose coupling, which is a core principle of microservice architecture. RESTful APIs enable fast and efficient client–server communication by exchanging data in JSON format. The proposed communication design method is described in more detail below:

1) Defined the functional requirements of four service offerings.

1.1) The user service managed all users within the system, including local citizens and local officials from thirty-one Local Administrative Organizations in Suratthani Province.

1.2) The news management service is responsible for creating and publishing news articles and official announcements from Local Administrative Organizations for citizens. On the other hand, it also allowed citizens to create news content and submit it to the system. This service included submodules for news content management, with associated images, and news report summarization.

1.3) The main service was responsible for managing various requests submitted by citizens to Local Administrative Organizations. It includes e-services, complaint submissions, grievance notifications, and status updates at each stage of the process.

1.4) The Notification Service was responsible for sending short message service (SMS) notifications, operational updates, and real-time news alerts.

The three core services (user, news management, and main service) were designed with a focus on CRUD (Create, Read, Update, and Delete) operations.

2) In this research, we created an API specification by defining API endpoints and HTTP methods using Swagger UI specification language. There were two types of API implementations: public APIs, which were directly accessed by client applications, and backend APIs, which facilitated communication among internal backend services. We carefully defined RESTful APIs to facilitate communication between services and support end-to-end use cases. Tables 4–7 present the communication design of the microservice architecture using RESTful APIs for each of the four microservices.

Table 4. Design of microservice architecture with RESTful APIs for the user microservice

APIs Endpoints	HTTP Method	Action
/user/users/{userId}	GET	Get the user by ID
	DELETE	Delete user by ID
/user/users/requestOTP	GET	Get OTP for initial authentication.
/user/users/sumUserPerMonth	GET	Get the summary user per month.
/user/users	GET	Retrieve all users
	POST	Create user
	PATCH	Update user
/user/lgo	GET	Retrieve all Local Administrative Organizations
/user/lgo/{id}	GET	Get Local Administrative Organization by ID
	DELETE	Delete Local Administrative Organization by ID
/user/users/register/user	POST	Register user
/user/users/register/verifyOTP	POST	Verify OTP
/user/users/login/user	POST	User logs in to the system
/user/users/upload/profile	POST	User upload profile
/user/lgo	POST	Create Local Administrative Organization
	PATCH	Update Local Administrative Organization
/user/users/user/update/approve	PATCH	Update approved user
/user/users/user/update/status	PATCH	Update user status

Table 5. Design of microservice architecture with RESTful APIs for news management microservice

APIs Endpoints	HTTP Method	Action
/news/news/gets/{type}	GET	Get news by type
/news/news/gets/{uuidNews}	GET	Get news
/news/news/summary/getAll	GET	Retrieve news summary
/news/news/{uuidNews}/like	GET	Get Like with news
/news/news/{uuidNews}/comment	GET	GET comment in the news
/news/files	GET	Retrieve news image file
/news/news	POST	Create news
/news/news/{uuidNews}/approve	POST	Approve news from citizens.
/news/news/{uuidNews}/ like	POST	Create a Like for news
/news/news/{uuidNews}/comment/ {commentId}/like	POST	Create a Like for a news comment.
/news/news/{uuidNews}/	PUT	Edit news
/news/news/{uuidNews}/	DELETE	Delete news
/news/news/{uuidNews}/comment/ {commentId}	DELETE	Delete news comment

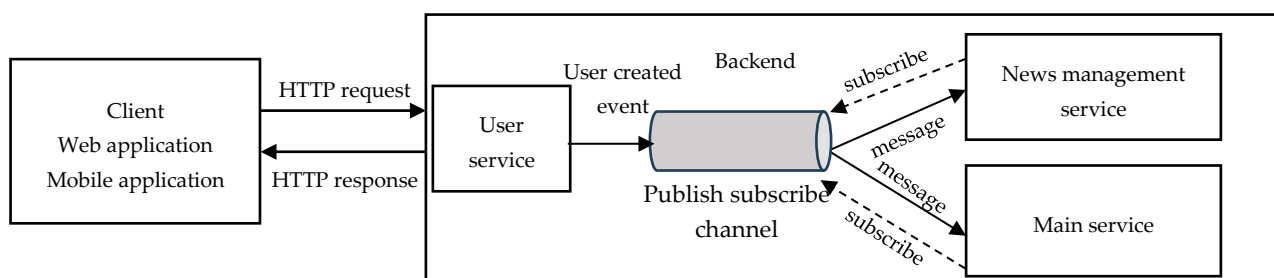
Table 6. Design of Microservice Architecture with RESTful APIs for the Main Service Microservice

APIs Endpoints	HTTP Method	Action
/main/e-service/summary/getAll	GET	Retrieve e-service summary
/main/e-service/findBy/{id}	GET	Get e-service by ID
/main/e-service/byNo/{eServiceNo}	GET	Get e-service by Number
/main/e-service/type/gets	GET	Retrieve e-service by type
/main/e-service	POST	Create e-service
/main/e-service/event	POST	Create an e-service event
/main/e-service/pagination	POST	Crete pagination for e-services
/main/e-service/{id}	PATCH	Update e-service by ID
	DELETE	Delete the e-service by ID
/main/report-problem/summary/getAll	GET	Retrieve the summary of problems
/main/report-problem/{id}	GET	Get problems by ID
/main/report-problem/ reportProblemType/getAll	GET	Retrieve all problem types
/main/report-problem	POST	Create problem details
/main/report-problem/update-event	POST	Create problem status
/main/report-problem/pagination	POST	Crete paging of problems
/main/report- problem/update/{problemId}	PATCH	Update the problem by ID
/main/report-problem/{report ProblemId}	DELETE	Delete problem by ID
/main/report-problem/event/{eventId}	DELETE	Delete the event of the problem
/main/complain/{id}	GET	Get a complaint by ID
/mail/complain/summary/getAll	GET	Retrieve the complaint summary
/main/complain/type/get All	GET	Retrieve complaints by type
/main/complain/create	POST	Create complain
/main/complain/update_event	POST	Create a complaint event
/main/complain/pagination	POST	Create a page of complaints
/main/complain/{complainId}	DELETE	Delete the complaint by ID
/main/complain/event/{eventId}	DELETE	Delete the event of complaint

Table 7. Design of Microservice Architecture with RESTful APIs for Notification Microservice

APIs Endpoints	HTTP Method	Action
/one-signal	POST	Create notification
/one-signal/sendNotification	POST	Send a notification to the user
/one-signal/user/{id}/onesignal/{onesignalId}	DELETE	Delete notification

We utilized Redis as a message broker to decouple services in cross-service transactions. The system supports services that operate independently while enabling asynchronous communication among microservices, allowing each service to execute without waiting for others to complete. In our direct client-to-microservice architecture, we implemented an event bus within a message broker using a publish-subscribe (pub-sub) messaging pattern, which is particularly well-suited for handling cross-service transactions. Figure 3 illustrates an example of publish-subscribe communication, in which the User Service publishes a UserCreated event to a topic. The News Management Service and Main Service subscribe to this topic and respond to the event accordingly.

**Figure 3.** Publish-subscribe communication in microservices.

2.3 Security design

We were concerned about security in the microservices architecture due to the direct client-to-service communication design. Data transmission security is vital, especially during the user authentication process. We designed a robust data protection mechanism using JSON Web Token (JWT); however, standard JWT implementations do not provide sufficient protection for the token data, which may contain sensitive government information that requires high levels of security and reliable communication. We implemented the Advanced Encryption Standard (AES) with a 256-bit key in Cipher Block Chaining (CBC) mode, using PKCS#7 padding during token generation to ensure secure and complete transmission of information, as illustrated in Figure 4.

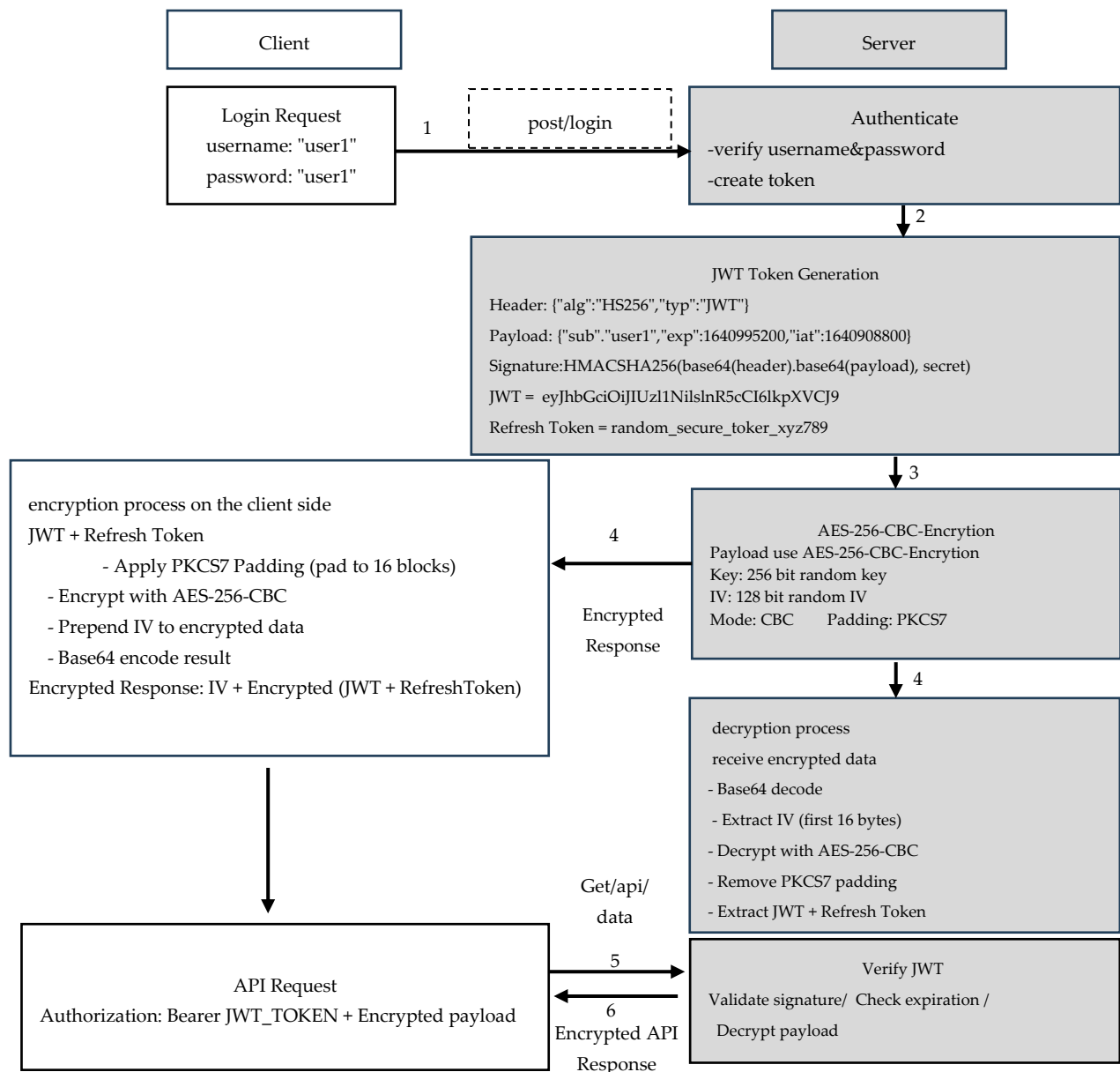


Figure 4. Authentication process of JWT with AES-256

The workflow for creating JWT tokens integrated with AES-256 encryption on the server is as follows:

- 1) The process begins when the client sends a login request, including username and password, to the server.
- 2) The server receives the request and verifies the username and password. If the credentials are correct, the server proceeds to generate the token as follows:
 - 2.1) Create the header containing:
 - alg: The encryption algorithm used
 - type: The token type, which is "JWT"
 - 2.2) Create the payload containing:
 - sub: user ID
 - exp: expiration date
 - iat: token issuance date
 - 2.3) Create the signature by:

3) Concatenating the header, payload, and server secret, then applying HMASHA256 encryption to generate the signature. The server then applies AES-256-CBC encryption as follows:

- key: 256-bit random key (32 bytes)
- IV: 128-bit random key (16 bytes)
- Mode: CBC (Cipher Block Chaining)
- Padding: PKCS7

4) This results in both encryption and decryption processes:

Encryption yields:

- Apply PKCS7 padding (pad to 16-byte blocks)
- Encrypt with AES-256-CBC
- Prepend IV to the encrypted data
- Base64 encode the result

Encrypted Response: IV + Encrypted (JWT + RefreshToken)

Decryption yields:

- Base64- decode
- Extract IV (first 16 bytes)
- Decrypt with AES-256-CBC
- Remove PKCS7 padding
- Extract JWT + Refresh Token

5) The API request data includes:

- Authorization: Bearer JWT_TOKEN + encrypted payload

6) The JWT verification process includes:

- Validate the signature
- Check expiration
- Decrypt the payload

Authorization is also essential in an e-Participation platform. We defined authorization based on user access rights within the system, including permissions that restrict access to API endpoints and roles, which are groups of permissions associated with users. We assigned permissions to roles rather than to individual users as follows.

1) The super administrator held the highest level of system privileges, enabling full control over user management across all levels.

2) Local administrative officers were responsible for managing citizens' approval, posting news, approving news, and approving requests for e-service usage.

3) Local administrative executives could view reports and participation statistics from the system.

4) Citizens in each local administrative area could register, post news, read news, access e-services, and submit complaints or problem reports.

2.4 Database design

Databases play a vital role in information systems, and database design is a critical aspect of microservice architecture. The database-per-service pattern was adopted in this architecture design. It means that each microservice manages its own database and may also exchange information with other services. This database design pattern supports loose coupling, a core principle of microservice architecture, as each microservice maintains its own independent database that can scale without affecting other services. The key advantage of the database-per-service pattern is its isolation: changes to the schema of one database do not affect the others. The one-microservice, one-database approach can eliminate single points of failure. When a problem occurs, it affects only one database. It also enhances system resilience, a key characteristic of microservice architecture. Based on the results of DDD and the API endpoint and HTTP method communication design from the previous stage, the system was structured accordingly. Four relational databases were created using MariaDB: user, news, main service, and notification. We adopted MariaDB as the DBMS because of its clear data modeling capabilities and support for well-defined schema from the early design phase. The structure and schema of relational databases support data consistency and accuracy by

enforcing relationships between tables and play a crucial role in managing essential data within each microservice. In this research, database design comprised four distinct databases, each responsible for managing essential data within its respective microservices.

1) The user database stores information related to system users, including Local Administrative Organizations, local administrative officers, and local citizens. The tables in the user database comprised user, sao (Local Administrative Organization), sao_organization_structure, user_token, user_socket, and user_password.

2) The main service database handled the functions of Local Administrative Organizations, including managing complaints, grievances, and e-service requests. Tables in the main service database comprised complain, complain_process, complain_status, report_problem, report_problem_type, report_problem_process, file, report_problem_status, e_service, e_service_type, and e_service_list.

3) The news database stored information about news and announcements from local authorities, as well as citizen-submitted news. Its main tables included: news, file, comments, and likes.

4) The notification database was responsible for handling messaging and alert functions. Tables in the notification database included sms_log, one_signal, and settings.

This modular database structure supported service independence, scalability, and efficient data management, aligning with microservice architectural principles.

2.5 Implementation and testing

The selected tools and techniques for implementing this architecture are as follows:

1) Angular Framework v.16 served as the front-end framework for building single-page, dynamic web applications for the e-Participation platform, enabling direct interaction with multiple microservices to deliver a highly interactive user experience. Ionic provided a robust and efficient way to build cross-platform mobile applications.

2) TypeScript and Node.js were used for backend development.

3) JSON Web Tokens were used for authentication and authorization, combined with AES-256 encryption.

4) Swagger was used for generating API documentation.

5) POSTMAN was used for API testing.

6) Apache JMeter was used for performance testing based on a predefined test plan. We measure system capacity through throughput, assess time behavior via average response time, and evaluate resource utilization.

7) Cloudflare was used as a content delivery network (CDN) for security and web content delivery.

8) Docker containers were used to provide containerization for supporting the four microservices and APIs in the prototype system.

3. Results and Discussion

The result of designing a direct client-to-microservice communication architecture for the e-Participation platform tailored to local government organizations is illustrated in Figure 5.

The system architecture diagram in Figure 5 illustrates the direct interaction between the client and the application, based on a microservice architecture. The client can directly access the microservice system via both mobile and web applications. This process was initiated when users logged in to request data retrieval or other service operations. The API layer was designed within each microservice to handle and transmit client requests. RESTful APIs enable communication between client applications and microservices. We utilized JSON Web Token (JWT) and AES-256 encryption within the authentication service layer. Both were employed to verify access rights and provide secure authentication and data protection. Upon successful authentication, the user's request was routed to the appropriate microservices: User Service, News Management Service, Main Service, and Notification Service. Each microservice maintained its own database. We utilized an in-memory cache, a high-speed storage mechanism that accessed data in the computer's main memory to accelerate data retrieval and improve application performance. Communication between microservices in the system was facilitated by a message broker that enabled asynchronous message delivery and supported efficient concurrent task handling.

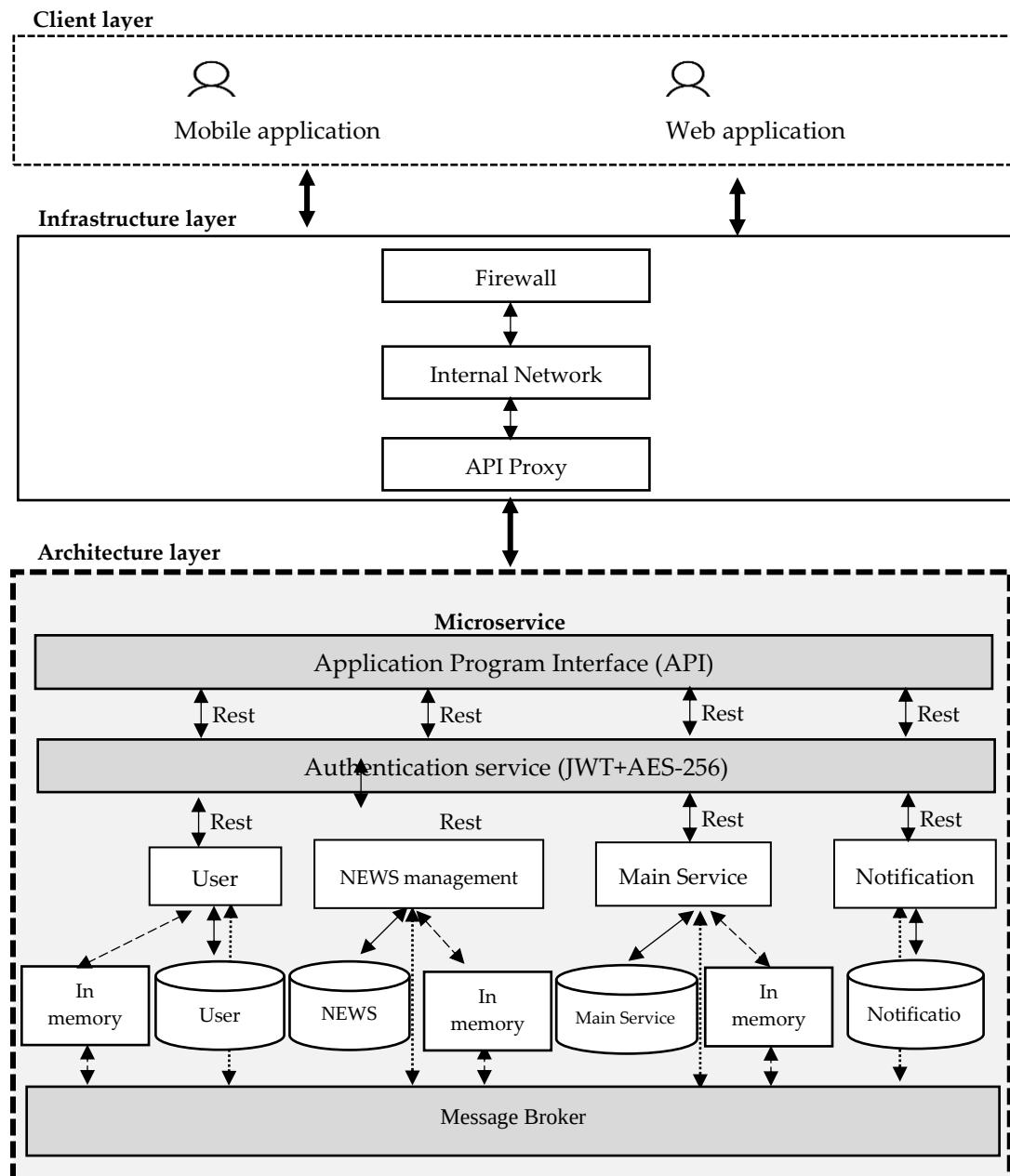


Figure 5 System architecture diagram of the e-Participation platform in Local Administrative Organizations

In this research, performance testing results were measured using Apache JMeter and aligned with the ISO/IEC 25010 standard, specifically the performance efficiency criterion, which refers to how efficiently the software operates. The workload modeling and experimentation were set up to assess the system's performance under both realistic and extreme conditions. We modeled the workload using the target population and standard capacity-planning heuristics. The estimation of Peak Concurrent Users (CCU_{peak}) [40][41] was formulated as follows:

$$CCU_{peak} = P \times AR \times DAR \times PHC \quad (1)$$

P (Population): The total target population of this study was 427,723 citizens from thirty-one Local Administrative Organizations in Suratthani Province [42].

AR (Active Ratio): The proportion of the population expected to register for and use the e-Participation platform, estimated at 20% (0.20). This value accounts for the "Digital Divide" and the varying levels of e-Government adoption across age groups [43].

DAR (Daily Active Ratio): The percentage of registered users who access the platform on a typical day, estimated at 6% (0.06), which is a standard benchmark for public service utility applications [40].

PHC (Peak Hour Concurrency): The fraction of daily active users (DAU) who are online simultaneously during the busiest hour, set to 10% (0.10) per industry-standard capacity planning rules for web traffic distribution [41].

Based on the mathematical model above, this research established four incremental testing scenarios to observe system behavior under different operational conditions:

1. Normal Load (Baseline): 500 CCU represented the system's typical day-to-day operating condition during non-peak periods with standard user engagement. This baseline was derived by rounding the estimated peak concurrent users (approximately 513) obtained from $CCU_{(peak)} = 427,723 \times 0.20 \times 0.06 \times 0.10$

2. Elevated Load: 1,000 CCU, doubling the baseline load, represented a moderate increase in traffic, such as during local government announcements or the release of new public information.

3. Peak Load (Event Day): 2,000 CCU was derived from the model by assuming that the Daily Active Ratio (DAR) increased to 24% during high-engagement events, yielding an estimated peak concurrency of approximately 2,053 CCU. This scenario represented the maximum expected traffic during critical government activities.

4. Stress Load (Beyond Limit): 5,000 CCU was selected to push the architecture beyond its anticipated maximum (2.5× the peak load). This level was used to identify bottlenecks, evaluate system resilience, and verify stability under extreme, unforeseen traffic spikes.

The performance testing results are categorized into three aspects: system capacity, time behavior, and resource utilization, as illustrated in Tables 8-11.

Table 8. Results of system capacity testing using the API HTTP method GET

Concurrent users (CCU)	Throughput (req/sec)	Error rate (%)	Duration	Expected throughput
500	11.18	0.00%	5 mins	≥ 1.5 req/sec, Error $< 0.5\%$
1000	13.38	0.00%	5 mins	≥ 3.0 req/sec, Error $< 0.5\%$
2000	14.74	0.00%	5 mins	≥ 6.0 req/sec, Error $< 1\%$
5000	16.67	0.00%	5 mins	≥ 15.0 req/sec, Error $< 1\%$

The results presented in Table 8 demonstrate the system's processing capability (Throughput) and absolute stability under varying concurrent user loads. A key finding was that the system maintained an error rate of 0.00% across all testing tiers. The throughput was recorded at 11.8 requests/sec for 500 concurrent users (CCU) and increased to 13.38 requests/sec at 1,000 CCU, 14.74 requests/sec at 2,000 CCU, and 16.67 requests/sec at 5,000 CCU. This increasing trend provides empirical evidence of the system's scalability. Furthermore, the performance aligned with the theory proposed in [44], which posits that system scaling need not be strictly linear to achieve robust growth. This increasing trend demonstrates the system's scalability, as evidenced by its accelerated growth and improved efficiency under concurrent load. Throughput remained within the expected range, and actual performance consistently exceeded expectations, particularly under high concurrent user loads.

$$\text{Expected_Throughput} = \text{Baseline_Throughput} \times (\text{Load_Ratio}^{0.7}) \times \text{Efficiency_Factor} \quad (2)$$

Baseline throughput refers to the actual throughput measured during the baseline test.

Load_Ratio is the ratio of the load increase compared to the baseline.

Efficiency_Factor is the value that indicates the reduction in efficiency as the load increases.

Table 9. Results of time behavior testing using the API HTTP method GET

Concurrent Users (CCU)	Ramp-Up (sec)	Avg. Response Time (ms)	95 th Percentile	Expected Response Time
500	45	41	52	< 300 ms
1000	75	46	55	< 525 ms
2000	135	41	48	< 900 ms
5000	315	44	53	< 1500 ms

The average response time results presented in Table 9 indicate that the system exhibited excellent performance, maintaining consistently low response times throughout the testing period. The average response time ranged from 41 to 46 milliseconds, which represents outstanding performance according to industry standards [45]. 95% of users received very fast responses from the system, with the 95th percentile response time ranging from 48 to 55 milliseconds, well below the expected response time. These test results demonstrate a highly efficient system architecture design.

$$\text{Expected Response Time} = \text{Base_Response_Time} + (\text{Load_Factor} \times \text{Degradation_Coefficient}) \quad (3)$$

Base_Response_Time is the baseline response time (for 100 users)

Load_Factor is the ratio of the load increase

Degradation_Coefficient is the coefficient of degradation (0.1–0.3)

Table 10. Results of Resource Utilization testing using the API HTTP method POST

Concurrent Users (CCU)	Ramp-Up (sec)	CPU % Usage (AVG/Peak)	Memory (MB)	Response Time (ms)	Expected CPU and Memory
500	45	12.1% / 17.0%	1373MB / 1381 MB	136	CPU < 15%, Memory < 1.8GB
1000	75	15.6% / 23.5%	1396 MB / 1398 MB	111	CPU < 20%, Memory < 2 GB
2000	135	20.2% / 26.1%	1395 MB / 1406 MB	107	CPU < 25%, Memory < 2.2GB
5000	315	24.6% / 31.0%	1400 MB / 1412 MB	104	CPU < 50%, Memory < 3GB

The resource utilization results presented in Table 10 are divided into two components: CPU performance and Memory performance. CPU utilization showed a logarithmic growth pattern. It started at 12.1% with 500 users. It then increased to 15.6% with 1,000 users, 20.2% with 2,000 users, and 24.6% with 5,000 users. CPU usage does not increase rapidly as the number of users grows, indicating that the system was well-designed and efficient in managing and allocating resources. Peak CPU utilization ranged from 17.0% to 31.0% and remained within the expected thresholds, indicating that the system was highly stable and unaffected by CPU spikes. Peak CPU utilization ranged from 17.0% to 31.0% and remained within the expected thresholds, indicating that the system was highly stable and unaffected by CPU spikes. The expected CPU threshold was determined using a data-driven approach to established performance standards, based on empirical baseline data and industry-standard scaling models [46].

$$\text{Expected CPU} = \text{Baseline_CPU} \times (1 + (\log(\text{Load_Ratio} + 1) / \log(2)) \times 0.4) + \text{Safety_Margin} \quad (4)$$

Baseline_CPU is the actual CPU utilization value measured from the baseline test.

Load_Ratio is the ratio of load increase compared to the baseline, calculated from

$$\text{Load_Ratio} = \text{Target_Users} / \text{Baseline_Users} \quad (5)$$

$\log(\text{Load_Ratio} + 1) / \log(2)$ is a logarithmic function that transforms the increase in load into a logarithmic growth pattern.

Coefficient 0.4 (Growth Coefficient) is a multiplier that controls the level of CPU increase according to the logarithmic pattern.

Safety_Margin is a compensation value that accounts for unexpected fluctuations.

Memory usage was highly efficient, increasing only slightly from 1,373 MB at 500 users to 1,400 MB at 5,000 users. It increased only 2% even though the number of users increased tenfold. This memory usage pattern indicates an efficient architectural design. Memory usage remained at only 45–75% of the expected memory value.

$$\text{Expected_Memory} = \text{Baseline_Memory} \times (\text{Load_Ratio}^{0.7}) + \text{Safety_Margin} \quad (6)$$

Baseline_Memory is the actual memory utilization value measured from the baseline test.

Load_Ratio is the ratio of load increase compared to the baseline.

$$\text{Load_Ratio} = \text{Target_Users} / \text{Baseline_Users} \quad (7)$$

The exponent 0.7 (Sublinear Scaling Exponent) represents the sublinear growth pattern of memory usage. Safety_Margin is the compensation value used to accommodate fluctuations in memory usage.

To evaluate system performance under real-world network latency conditions, we simulated three distinct network scenarios: high-speed (Fiber/5G), average (4G/LTE), and poor (3G/Unstable). The Characters Per Second (CPS) parameter in Apache JMeter was used to emulate bandwidth throttling and to assess its impact on system latency under a load of 2,000 concurrent users. The test results are summarized in Table 11.

Table 11. Comparative Performance Testing under Variable Network Conditions

Network Profile	Latency (ms)	Bandwidth	Avg. Response Time (ms)	Success Rate (%)
HighSpeed (Fiber/5G)	0 - 5	Unlimited	39	100.00%
Average (4G/LTE)	20 - 50	10 Mbps	44	100.00%
Poor (3G/Unstable)	100 - 300	1 Mbps	114	100.00%

The results of network latency simulation indicate that the average response time increased with higher latency and tighter bandwidth constraints, but overall performance remained robust. Specifically, the average response time increased from 39 ms under high-speed connectivity (Fiber/5G) to 44 ms under a moderate network profile (4G/LTE), and reached 114 ms under severely constrained conditions with only 1 Mbps bandwidth (3G/Unstable). Notably, the system maintained a 100.00% transaction success rate across all scenarios. These empirical findings suggest that the proposed architecture demonstrated high system resilience and an effective connection-pooling mechanism, enabling rapid resource release and preventing slow-client bottlenecks even in unstable network environments.

Although this study adopts a direct client-to-microservice architecture, the experimental results demonstrated that the proposed system achieved both high performance and strong security. These findings are discussed in the following subsections. Performance is widely recognized as one of the most critical quality attributes in microservices architecture design [2]. The performance can be enhanced through tactics such as resource placement and allocation, load balancing, containerization, and profiling. However, while prior studies have addressed these tactics broadly, limited attention has been given to the performance implications of resource placement and allocation in real-world applications. To address this gap, this research evaluated system efficiency using ISO/IEC 25010, with particular emphasis on three performance aspects: system capacity, time behavior, and resource utilization, which indicate the system's functional efficiency and resource management [47]. The design findings demonstrated that the architecture achieved a throughput of 16.67 requests per second with 5,000 concurrent users and met the expected throughput threshold. This outcome validates the effectiveness of microservice decomposition in distributing system load and ensuring responsiveness. In addition, the API design in this research played a crucial role in determining system throughput, as REST APIs are known for fast response times and high throughput [48]. The system maintained an average response time of 41-46 milliseconds, which was lower than the expected response time and highlighted the API's responsiveness and efficiency. In this research, we used an in-memory cache and Redis as a message broker for inter-service asynchronous communication, which significantly reduced response time [49] by reducing inter-service coupling and improving throughput [50].

Furthermore, CPU resource utilization did not increase rapidly as the number of users increased from 500 to 5,000. Peak CPU utilization ranged from 17.0% to 31.0%, which remained within expected thresholds. There was also only a slight increase in memory usage, from 1,381 MB at 500 users to 1,412 MB at 5,000 users, indicating that the system could handle intensive workloads without compromising stability. This outcome reflects the benefits of microservice architecture, in which applications are decomposed into smaller independent services that can be scaled individually based on demand. To answer RQ1, we confirm that the proposed architecture met the critical performance quality attributes, including system capacity, time behavior, and resource utilization.

Security also represents one of the most critical quality attributes in microservices architecture design [2]. We used a direct client-to-microservice architecture and set up the infrastructure layer to ensure system security. A firewall acted as a security barrier to block malicious software, while an API proxy served as an intermediary for API requests, handling security and caching. Furthermore, a considerable body of this research has examined security mechanisms, with particular emphasis on the application of JSON Web Tokens (JWT) and AES-256 encryption. The resource utilization results for the HTTP POST method in Table 10 highlighted the architecture's ability to handle the security overhead associated with AES-256 encryption before database storage. Although this process is inherently more complex than standard data retrieval (HTTP GET), the total response time remained remarkably low, ranging from 104 to 136 milliseconds. Compared with the results in Table 9, the overhead increases by approximately 60-95 milliseconds, accounting for both encryption time and write latency. These findings confirm that the integrated security mechanisms do not compromise the user experience, as the system maintains robust processing efficiency even under peak load conditions of 5,000 concurrent users (CCU), with peak CPU utilization reaching only 31.0%. These results indicate that the system was sufficiently stable while ensuring data security. We implemented AES-256 encryption for the e-Participation system because the application required high-speed performance and stronger data security, consistent with Singh (2022) [51], who found that AES-256 provides stronger security than AES-128 when higher security requirements are needed. AES provides fast encryption and decryption, resulting in lower latency than Rivest-Shamir-Adleman (RSA) and elliptic curve cryptography (ECC) [52][53]. AES also provides high throughput because it is ideal for encrypting large volumes of data quickly [53], with efficient CPU and memory usage, which contributes to its widespread adoption across various applications [54]. The adoption of RESTful APIs and the integration of AES-256 encryption significantly enhanced the protection of sensitive information, thereby reducing the risk of data leakage. Although the direct client-to-microservice architecture increases the attack surface, the evaluated architecture was deployed on a single-node server and therefore subject to a hardware-level single point of failure (SPOF). The software layer was explicitly engineered to tolerate overload conditions. Results in Table 10 show that the system maintained normal operation under a workload of 5,000 concurrent users, during which incoming requests exceeded the maximum available processing threads. The system avoided deadlock and out-of-memory conditions by employing graceful degradation, temporarily rejecting excess requests with HTTP 503 (Service Unavailable) or by returning timeout responses. This mechanism prevents cascading failures and enables automatic recovery once the workload returns to nominal levels, without requiring a system restart.

Much prior research has demonstrated the effectiveness of enhancing system performance and security in microservice architecture design. However, limited research has examined the integration of these technologies within direct client-to-microservice communication for e-Participation platforms in Local Administrative contexts. This study sought to address this gap by ensuring compliance with security requirements and maintaining high performance in such contexts. Nevertheless, in real-world deployments of e-Participation platforms, several factors should be considered, including internet connectivity, heterogeneous client devices, and organizational constraints. Therefore, future research should extend the evaluation to field deployments, usability, and user satisfaction.

This study was designed and developed as a prototype for deployment within thirty-one Local Administrative Organizations in Suratthani Province, which serve an estimated target population of approximately 427,723 citizens. The evaluated system capacity of 5,000 concurrent users (CCU) was sufficient to cover the anticipated peak load for this target user group. However, the primary technical limitation of the proposed prototype lies in its scalability. If extended to regional or national deployment with significantly higher user demand, the current single-node server architecture may encounter resource constraints. Therefore, future work should focus on evolving the architecture toward a distributed system by integrating an API Gateway for centralized request routing and employing load balancers to enable horizontal scaling across multiple servers, with proper management to support large-scale expansion and sustained high-load operation.

4. Conclusions

This study demonstrated the feasibility of a direct client-to-microservice communication architecture design. The combination of DDD, API-first design, JWT with AES-256 encryption, and a message broker proposed in the architecture provided a robust foundation for ensuring both performance and security in an e-Participation platform for Local Administrative Organizations. The use of direct client-to-microservice communication exposed services more directly to external clients. Therefore, careful authentication, role-based access control, and encrypted transport layers were necessary. Furthermore, the database-per-service pattern upholds service independence and aligns with the system's modular requirements. The findings of this research suggest that adopting this architectural model can enable local governments to deploy flexible, secure, and efficient e-participation platforms that enhance interaction, monitoring, and follow-up on various services requested by local civilians.

5. Acknowledgments

Author Contributions: Conceptualization, KP, CP. and AC.; methodology, KP and CP.; software, KP and SS.; validation, KP, CP, AC and SS.; formal analysis, KP, CP, AC and SS.; investigation, KP, CP and AC.; resources, KP, CP, AC and SS.; data curation, KP and SS.; writing—original draft preparation, KP and SS.; writing—review and editing, KP and SS.; visualization, KP and SS.; supervision, KP and CP.; project administration, KP.; funding acquisition, KP. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the National Research Council of Thailand (NRCT), grant number N33A660025.

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study, in the collection, analysis, or interpretation of data, in the writing of the manuscript, or in the decision to publish the results.

References

- [1] United Nations. *E-Participation Index*; UN E-Government Knowledgebase. <https://publicadministration.un.org/egovkb/en-us/About/Overview/E-Participation-Index> (accessed Dec 1, 2024).
- [2] Li, S.; Zhang, H.; Jia, Z.; Zhong, C.; Zhang, C.; Shan, Z.; Shen, J.; Babar, M. A. Understanding and Addressing Quality Attributes of Microservices Architecture: A Systematic Literature Review. *Inf. Softw. Technol.* **2021**, *131*, 106449. <https://doi.org/10.1016/j.infsof.2020.106449>
- [3] Thönes, J. Microservices. *IEEE Softw.* **2015**, *32*(1), 116–116. <https://doi.org/10.1109/MS.2015.11>.
- [4] Isnaeni, R. M.; Utama, N. I.; Suakanto, S. Backend Development of a Microservice-Based Website Application for Public Issue Reporting: Case Study in People Representative Council. *J. La Multiapp* **2024**, *5*(2), 63–77. <https://doi.org/10.37899/journallamultiapp.v5i2.1148>
- [5] Medaglia, R. eParticipation Research: Moving Characterization Forward (2006–2011). *Gov. Inf. Q.* **2012**, *29*, 346–360. <https://doi.org/10.1016/j.giq.2012.02.010>
- [6] Blanc, D. L. *E-Participation: A Quick Overview of Recent Qualitative Trends*; United Nations Department of Economic and Social Affairs, 2020. https://www.un.org/esa/desa/papers/2020/wp163_2020.pdf (accessed Jan 10, 2025)
- [7] Medaglia, R.; Zheng, L. Mapping Government Social Media Research and Moving It Forward: A Framework and a Research Agenda. *Gov. Inf. Q.* **2017**, *34*(3), 496–510. <https://doi.org/10.1016/j.giq.2017.06.001>
- [8] Maurel, A. L.; Telen, M. A. E.; Uy, U. L. F.; Lorilla, F. M. A. Enhancing e-Government through Microservices: The Development and Impact of the NTC-EDGE System. *Mindanao J. Sci. Technol.* **2024**, *22* (Suppl. 1). <https://doi.org/10.61310/mjst.v22iS1.2221>
- [9] Puspitasari, N.; Budiman, E.; Sulaiman, Y. N.; Firdaus, M. B. Microservice API Implementation for E-Government Service Interoperability. *J. Phys.: Conf. Ser.* **2021**, *1807*(1), 012005. <https://doi.org/10.1088/1742-6596/1807/1/012005>
- [10] Baresi, L.; Garriga, M. Microservices: The Evolution and Extinction of Web Services? In *Microservices: Science and Engineering*; Springer: Cham, Switzerland, 2020; pp 3–28. https://doi.org/10.1007/978-3-030-31646-4_1

- [11] Levcovitz, A.; Terra, R.; Valente, M. T. Towards a Technique for Extracting Microservices from Monolithic Enterprise Systems. *arXiv* **2016**, arXiv:1605.03175.
- [12] Soldani, J.; Tamburri, D. A.; Van Den Heuvel, W.-J. The Pains and Gains of Microservices: A Systematic Grey Literature Review. *J. Syst. Softw.* **2018**, *146*, 215–232. <https://doi.org/10.1016/j.jss.2018.09.082>
- [13] Kapferer, S.; Zimmermann, O. Domain-Specific Language and Tools for Strategic Domain-Driven Design, Context Mapping and Bounded Context Modeling. In *Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development*; SCITEPRESS, 2020; pp 299–306. <https://doi.org/10.5220/0008910502990306>
- [14] Velepucha, V.; Flores, P. A Survey on Microservices Architecture: Principles, Patterns and Migration Challenges. *IEEE Access* **2023**, *11*, 88339–88358. <https://doi.org/10.1109/ACCESS.2023.3305687>
- [15] Gite, K.; Gunjal, A.; Shinde, V. M. Monolithic vs. Microservices Architecture: A Comparative Study of Software Development Paradigms. *Int. Sci. J. Eng. Manage.* **2025**, *4*(6), 1–9. <https://doi.org/10.55041/ISJEM04468>
- [16] Blinowski, G.; Ojdowska, A.; Przybyłek, A. Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation. *IEEE Access* **2022**, *10*, 20357–20373. <https://doi.org/10.1109/ACCESS.2022.3152803>
- [17] Kambhammettu, A. Microservices vs. Monoliths: Choosing the Right Architecture. *J. Multidiscip.* **2025**, *5*(7), 629–636.
- [18] Padvekar, A. A.; Gupta, V. B. Comparative Analysis of Monolithic vs. Distributed Architecture. *Int. J. Adv. Res. Sci. Commun. Technol.* **2024**, 433–442. <https://doi.org/10.48175/IJAR SCT-18946>
- [19] Tupsakhare, P. Monolithic vs. Microservices: Analyzing Architecture Paradigms in Modern Software Development. *J. Sci. Eng. Res.* **2019**, *6*(2), 299–303.
- [20] Nagraj, A. Microservices vs. Monoliths in Financial Applications: A Comparative Analysis for Scalable Architectures. *J. Econ. Manage. Res.* **2021**, *2*, 2–6. [https://doi.org/10.47363/JESMR/2021\(2\)288](https://doi.org/10.47363/JESMR/2021(2)288)
- [21] Kumar, V. Microservice-Driven Performance Optimization in Large-Scale Transaction Processing Systems. *Int. J. Comput. Exp. Sci. Eng.* **2025**, *11*(4). <https://doi.org/10.22399/ijcesen.4461>
- [22] Wahyudin, A.; Anisyah, A.; Kusumawardani, S. M. Migrating Monolithic to Microservices: Comparative Performance Testing Evaluation Between Architectures. *Sci. J. Inf.* **2025**, *12*(4), 796–816. <https://doi.org/10.15294/sji.v12i4.28499>
- [23] Rademacher, F.; Sachweh, S.; Zündorf, A. Towards a UML Profile for Domain-Driven Design of Microservice Architectures. In *Software Engineering and Formal Methods*; Cerone, A.; Roveri, M., Eds.; Lecture Notes in Computer Science; Springer International Publishing: Cham, Switzerland, 2018; Vol. 10729, pp 230–245. https://doi.org/10.1007/978-3-319-74781-1_17
- [24] Newman, S. *Building Microservices*, 1st ed.; O'Reilly Media: Sebastopol, CA, 2015.
- [25] Lercher, A.; Glock, J.; Macho, C.; Pinzger, M. Microservice API Evolution in Practice: A Study on Strategies and Challenges. *J. Syst. Softw.* **2024**, *205*, 112110. <https://doi.org/10.1016/j.jss.2024.112110>
- [26] Richardson, C. *Microservices Patterns: With Examples in Java*; Manning Publications: Shelter Island, NY, 2018.
- [27] Dudjak, M.; Martinović, G. An API-First Methodology for Designing a Microservice-Based Backend as a Service Platform. *Inf. Technol. Control* **2020**, *49*(2), 206–223. <https://doi.org/10.5755/j01.itc.49.2.23757>
- [28] Dragoni, N.; Lanese, I.; Larsen, S. T.; Mazzara, M.; Mustafin, R.; Safina, L. Microservices: Yesterday, Today, and Tomorrow. In *Present and Ulterior Software Engineering*; Mazzara, M.; Meyer, B., Eds.; Springer: Cham, Switzerland, 2017; pp 195–216. https://doi.org/10.1007/978-3-319-67425-4_12
- [29] Joshi, P. K. Microservices and API Gateways: The Backbone of Modern Application Platforms. *Int. J. Core Eng. Manage.* **2019**, *6*(2), 177–183. <https://doi.org/10.5281/zenodo.13917563>
- [30] Yerram, S. AWS API Gateway: A Research-Based Study on Cloud-Native Proxying and Integration. *Int. J. Innov. Res. Creat. Technol.* **2025**, *10*(6), 1–4. <https://doi.org/10.5281/zenodo.17409762>
- [31] Zhao, J. T.; Jing, S. Y.; Jiang, L. Z. Management of API Gateway Based on Micro-Service Architecture. In *J. Phys.: Conf. Ser.*; IOP Publishing, 2018; 1087, p 032032. <https://doi.org/10.1088/1742-6596/1087/3/032032>
- [32] Ochuba, N. A.; Kisina, D.; Owoade, S.; Uzoka, A. C.; Gbenle, T. P.; Adanigbo, O. S. Systematic Review of API Gateway Patterns for Scalable and Secure Application Architecture. *Int. J. Front. Multidiscip. Res.* **2021**, *2*, 94–100. <https://doi.org/10.54660/IJFMR.2021.2.1.94-100>
- [33] Bhole, A. Reducing Inter-Service Communication Latency in Microservices. *Indian Sci. J. Res. Eng. Manage.* **2024**, *8*(12), 1–6. <https://doi.org/10.55041/IJSREM39350>

- [34] Obuse, E.; Erigha, E. D.; Okare, B. P.; Uzoka, A. C.; Owoade, S.; Ayanbode, N. Optimizing Microservice Communication with gRPC and Protocol Buffers in Distributed Low-Latency API-Driven Applications. *Iconic Res. Eng. J.* **2020**, *4*(3), 250–268.
- [35] Eyerman, S.; Hur, I. Efficient Asynchronous RPC Calls for Microservices: DeathStarBench Study. *arXiv* **2022**, arXiv:2209.13265. <https://doi.org/10.48550/arXiv.2209.13265>
- [36] Nomula, V. R. Optimizing Financial Systems with Microservices Architecture. *Int. J. Comput. Eng. Technol.* **2024**, *15*(5), 229–236.
- [37] Peyrott, S. *JWT Handbook*; Auth0, 2024. <https://auth0.com/resources/ebooks/jwt-handbook/thankyou> (accessed Aug 10, 2025).
- [38] Yarygina, T.; Bagge, A. H. Overcoming Security Challenges in Microservice Architectures. In *Proceedings of the 2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)*; IEEE: Washington, DC, 2018; pp 11–20. <https://doi.org/10.1109/SOSE.2018.00011>
- [39] Adrianto, S.; Herlambang, B. A.; Renaldy, R. Optimizing Customer Data Security in Water Meter Data Management Based on RESTful API and Data Encryption Using AES-256 Algorithm. *J. Appl. Inform. Comput.* **2025**, *9*(3), 876–882. <https://doi.org/10.30871/jaic.v9i3.9358>
- [40] Molyneaux, I. *The Art of Application Performance Testing: From Strategy to Tools*, 2nd ed.; O'Reilly Media, Inc.: Sebastopol, CA, 2014.
- [41] Menascé, D. A.; Almeida, V. A. F. *Capacity Planning for Web Services: Metrics, Models, and Methods*; Prentice Hall Professional: Upper Saddle River, NJ, 2002.
- [42] Bureau of Registration Administration, Department of Provincial Administration. *Civil Registration Population Statistics (Monthly)*. <https://stat.bora.dopa.go.th/stat/statnew/statMONTH/statmonth/mainpage> (accessed Dec 1, 2024).
- [43] Carter, L.; Bélanger, F. The Utilization of E-Government Services: Citizen Trust, Innovation and Acceptance Factors. *Inf. Syst. J.* **2005**, *15*(1), 5–25. <https://doi.org/10.1111/j.1365-2575.2005.00183.x>
- [44] Hassan, S.; Bahsoon, R.; Buyya, R. Systematic Scalability Analysis for Microservices Granularity Adaptation Design Decisions. *Softw.: Pract. Exp.* **2022**, *52*(10), 2096–2131. <https://doi.org/10.1002/spe.3069>
- [45] Alnuhait, H. A.; Alzyadat, W.; Althunibat, A.; Kahtan, H.; Zaqaibeh, B.; Al-Khawaja, H. A. Web Application Performance Assessment: A Study of Responsiveness, Throughput, and Scalability. *Int. J. Adv. Appl. Sci.* **2024**, *11*(9), 214–226. <https://doi.org/10.21833/ijaas.2024.09.023>
- [46] Jugo, I.; Kirinić, V.; Vukovac, D. P. Analysis and Evaluation of Web Application Performance Enhancement Techniques. In *Current Trends in Web Engineering*; Springer International Publishing: Cham, Switzerland, 2014; pp 24–37. https://doi.org/10.1007/978-3-319-08245-5_3
- [47] Hussein, S.; Lahami, M.; Torjmen, M. Assessing the Quality of Microservice and Monolithic-Based Architecture: A Systematic Literature Review. *Oper. Res. Eng. Sci.: Theory Appl.* **2024**, *7*(2), 417–446.
- [48] Lawi, A.; Panggabean, B. L. E.; Yoshida, T. Evaluating GraphQL and REST API Services Performance in a Massive and Intensive Accessible Information System. *Computers* **2021**, *10*(11), 138. <https://doi.org/10.3390/computers10110138>
- [49] Iurchenko, A. Optimization of Microservices Architecture Performance in High-Load Systems. *Am. J. Eng. Technol.* **2025**, *7*(05), 123–132. <https://doi.org/10.37547/tajet/Volume07Issue05-10>
- [50] Weerasinghe, S.; Perera, I. Optimized Strategy for Inter-Service Communication in Microservices. *Int. J. Adv. Comput. Sci. Appl.* **2023**, *14*(2). <https://doi.org/10.14569/IJACSA.2023.0140233>
- [51] Singh, A. Testing the Impact of Encryption on Network Performance. *Int. J. Sci. Res. Eng. Manage.* **2022**, *6*(9), 1–10. <https://doi.org/10.55041/IJSREM16483>
- [52] Karacali, H.; Cebel, E.; Donum, N. Performance Analysis of Encryption Algorithms in Named Pipe Communication for Linux Systems. *Eurasia Proc. Sci. Technol. Eng. Math.* **2024**, *27*, 214–227. <https://doi.org/10.55549/epstem.1518789>
- [53] Asare, S.; Yaokumah, W.; Gyebi, E.; Abdulai, J.-D. Evaluating the Impact of Cryptographic Algorithms on Network Performance. *Int. J. Cloud Appl. Comput.* **2022**, *12*(1), 1–15. <https://doi.org/10.4018/IJCAC.309937>
- [54] Özer, E.; Aydos, H. Performance and Security of AES, DES, and RSA in Hybrid Systems: An Empirical Analysis of Triple Encryption. *Int. J. Comput. Exp. Sci. Eng.* **2024**, *10*(4). <https://doi.org/10.22399/ijcesen.694>