Matrik: Jurnal Manajemen, Teknik Informatika, dan Rekayasa Komputer
Vol. 25, No. 1, November 2025, pp. 189∼204
ISSN: 2476-9843, accredited by Kemenristekdikti, Decree No: 10/C/C3/DT.05.00/2025
DOI: 10.30812/matrik.v25i1.5545

Middleware Development for Heterogeneous Databases on Multi-Architecture Systems Small Medium Enterprise

Christopel H. Simanjuntak[1], Musfiah[1], Muhammad Bahit[2], Cristovani W. Lohonauman[1], Stenly B. Dodie[1], Khamla Nonalinsavath[3]
[1]Politeknik Negeri Manado, Manado, Indonesia
[2]Universitas Lambung Mangkurat, Banjarmasin, Indonesia
[3]National University of Laos, Vientiane, Laos

Article Info

Article history:

Received August 16, 2025
Revised October 14, 2025
Accepted November 21, 2025

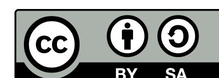
Keywords:

Application Programming Interface;
Information Systems;
Sales Forecasting.

ABSTRACT

The fisheries industry is highly complex, requiring information technology to support data recording, information management, and sales forecasting. At Kampunglawo, a sales information system has been developed to manage transactions and track shipments of fishery products. However, the sales forecasting systems were developed separately, with different architectures and underlying data structures, necessitating data duplication and restructuring and resulting in inefficiencies. The objective of this research is to develop an Application Programming Interface (API) that connects the two systems, enabling data sharing without redundancy. The methods used were literature review, system requirements analysis, design, implementation, functional and performance testing, and evaluation. The results of this research show that the developed API can synchronize data between the sales and forecasting systems with high efficiency. Testing showed that for 1, 10, and 50 synchronized data sets, the server response ratio was 1:1,057:1,869, with an increase in processing time of only 41.7% for the largest data volume. The conclusion of this research shows that using APIs can reduce processing time and eliminate the need for data restructuring, thereby increasing the efficiency of the company's information system integration.

Copyright ©2025 The Authors.
This is an open access article under the CC BY-SA license.



Corresponding Author:

Christopel H. Simanjuntak,
Electrical Engineering Department,
Politeknik Negeri Manado, Manado, Indonesia,
Email: simanjuntak@polimdo.ac.id

How to Cite:
C. H. Simanjuntak, M. Musfiah, M. Bahit, C. W. Lohonauman, S. B. Dodie, and K. Nonalinsavath, "Middleware Development for Heterogeneous Databases on Multi-Architecture Systems Small Medium Enterprise", *MATRIK: Jurnal Manajemen, Teknik Informatika, dan Rekayasa Komputer*, Vol. 25, No. 1, pp. 189-204, November, 2025.
This is an open access article under the CC BY-SA license (https://creativecommons.org/licenses/by-sa/4.0/)

1. INTRODUCTION

The utilization of digital information systems in the fisheries sector has a strategic role to improve distribution efficiency and accuracy in decision-making [1]. Digitalization of the system can support the tracking of catch shipments and provide projections of sales potential based on historical data and environmental parameters [2, 3]. Due to various needs, the systems built are not always the same, so differences in data processing, application components, and system development are inevitable [4].

The fisheries industry is one of the strategic sectors for food security and maritime economy, especially in Indonesia. Along with the increasing market demand and the complexity of the seafood supply chain, fisheries businesses, both large companies and MSMEs, are developing information systems that can encourage quick and precise decision-making, from the sales process to predicting the results of both fish and sales data [5]. This factor makes the development of multi-platform-based systems [6] such as desktop, mobile, and web-based, have become an important and urgent need so that information can be accessed anytime and anywhere, both from collectors and at the company management scale [7].

In their implementation, information systems generally use two main types of databases: relational (SQL) databases [8] to handle structured data, and non-relational (NoSQL) databases for other unstructured data, such as fishing weather or measuring package delivery, etc [9, 10]. Relational databases (SQL) such as MySQL or PostgreSQL are commonly used for structured transaction data processing [11]. In contrast, NoSQL, such as MongoDB, is more suitable for managing large amounts of data that do not always have a fixed format [12]. However, not all systems can be merged due to the time of system creation and cost reasons for the merger, so other ways are needed so that, even though different platforms, data exchange between systems is still established [13]. Although each type of database has its own advantages, integration between them is still a challenge, especially when used in cross-platform systems [14].

To bridge the communication between separate databases with different system architectures efficiently and securely, an Application Programming Interface (API) is needed [15] that can manage data flow effectively. Good API development allows cross-platform applications to access data, such as sales data and prediction data, without having to know the technical details of each database [16, 5]. In addition, APIs can be used to simplify the data synchronization process, maintain information consistency, and increase system response speed [17]. API Gateway is proven to optimize control over data traffic from various databases, in terms of security and access speed [18]. In modern systems running across multiple platforms, the main challenge is ensuring consistent, fast, and secure data access. APIs must be designed efficiently to support maximum performance for each application that exchanges data [19]. Another challenge is whether the required data is stored in databases that have different structures and different system architectures [17]. APIs must be built to have good resilience and reliability so that the two systems that will communicate with each other are not interrupted in carrying out each existing function [18].

Research conducted [20] uses GvdsSQL, a unified access technology for heterogeneous databases in wide-area environments. This system builds adaptive middleware for various DBMSs and extracts metadata to form a unified access model. GvdsSQL provides request resolution and multi-level caching mechanisms so that queries to multiple relational and non-relational databases can be served through a single, high-performance, centralized interface. While it provides a powerful, unified approach to integrating heterogeneous databases, the research relies on metadata from various DBMSs, which is often inconsistent and of varying quality.

Research conducted [20] designed a unified query platform as middleware that enables standardized query execution against various NoSQL data stores (key-value, document, column, graph) through a single query mechanism. The main focus of this research is how the middleware hides the differences in data models and query languages to facilitate the use of a single interface layer. This research has the weakness that unifying query mechanisms in the middleware can overabstract the unique features of each NoSQL type, preventing the optimal utilization of specific capabilities, such as nested document structures, column operations, or graph traversal.

Study [21] proposes a middleware model, SQL-to-NoSQL Query Translation (SQL-No-QT), that serves as an intermediary layer between legacy RDBMS-based applications and MongoDB. The middleware aims to translate basic SQL operations (SELECT, INSERT, UPDATE, DELETE, including some joins) into MongoDB document queries, then returns the results to the application as if they came from a relational database. The weakness of this research on the SQL-No-QT middleware model, which translates basic SQL operations into MongoDB document queries, remains limited in handling the complexity of SQL features, especially multi-table join operations with strong relational dependencies.

Research by [22] evaluated the role of API Gateway in improving access speed and security in multi-platform systems. While this solution improves data traffic management, the study did not synchronize datasets between two systems with different structures and architectures. [23] introduces middleware based on a service-oriented architecture (SOA) to provide integrated access to heterogeneous database systems. This middleware allows applications to interact with multiple different data systems without needing to know the internal details of each (e.g., database type, location, data format). However, SOA-based middleware can introduce overhead, degrading performance under high loads.

Based on previous research, this study offers the novelty of developing middleware for heterogeneous database integration that not only provides unified access but also addresses the major weaknesses of the unified access, unified query, and SQL–NoSQL translation models identified. Unlike previous research [20] which is highly dependent on metadata quality, this study proposes an integration mechanism that does not rely entirely on uniform metadata. Instead, it utilizes dynamic mapping strategies and adaptive abstractions that are more tolerant of schema differences. In addition, this study does not adopt a rigid query unification approach as in the previous study [24] which results in the loss of typical NoSQL features; instead, the proposed middleware accommodates database-specific features by providing an adaptation layer that preserves each DBMS's native capabilities. Compared to research [21] which can only translate basic SQL operations to MongoDB and cannot handle joins.

This research was conducted at Tampunglawo, a company engaged in fisheries that exports fish from North Sulawesi to various cities in Indonesia. In this company, a sales system has been developed to assist companies in selling their marine products. This system uses a tracking system for shipping goods to customers, built with the Flask architecture and MongoDB as the NoSQL database, and is developed in Python. While in development, a prediction system was developed to forecast the company's sales trend, enabling stakeholders to make timely, informed decisions to increase sales [7, 25, 26]. This prediction system was developed with an Apache server architecture with a PHP backend [27]. This system is embedded with a structured database that is used for prediction. The developed sales prediction system really needs data from the sales information system, which is developed on a different database and even a different architecture [17]. As a consequence, to operate the prediction system, it is necessary to duplicate and restructure the data so that the prediction system's database can recognize and process the existing records. This process is time-consuming and labor-intensive, as it requires personnel with sufficient expertise in the two distinct database structures employed by the respective systems. Moreover, this issue reduces organizational productivity, even though both systems were originally developed to enhance the company's overall efficiency. Due to this problem, a way is needed to bridge the two systems without disrupting the running system [28].

This research aims to design and develop an API [29, 15] that can help synchronize data across SQL and NoSQL databases in this company's multi-architecture system. With this approach, it is expected that data communication between systems is much easier and can support communication between two systems. APIs (Application Programming Interfaces) facilitate the secure, standardized exchange of data between systems. One commonly used API approach is the RESTful API due to its simplicity and ability to handle HTTP protocol-based requests [19]. The developed API must be evaluated using appropriate testing methodologies. Previous studies have evaluated aspects such as source code, functionality, and query-based performance [25–27]. However, no prior research has examined performance testing in the context of dataset synchronization between two heterogeneous systems. Therefore, one objective of this study is to conduct a direct performance evaluation during dataset synchronization across the two systems.

2. RESEARCH METHOD

In this study, the research workflow is structured into seven systematic stages to ensure methodological rigor and comprehensive outcomes. The process begins with an extensive literature review to establish the theoretical foundation and identify relevant gaps pertinent to the research objectives. A detailed analysis of system requirements follows this to define functional and non-functional specifications. Subsequently, the design phase focuses on developing an Application Programming Interface (API) architecture aligned with the identified requirements. The API is then implemented using the selected technologies and development frameworks. After implementation, a series of testing and evaluation activities is conducted to assess performance, reliability, and compliance with the expected criteria. The final stage involves compiling the research results into a comprehensive report. The overall research workflow is illustrated in Figure 1, which outlines each stage.

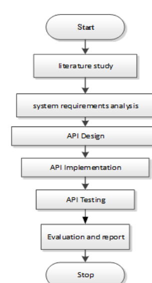


Figure 1. Research steps

2.1. Literature study

In the literature review phase, the researcher collected, mapped, and analyzed scientific references relevant to the research focus: the development of middleware for heterogeneous database integration in multi-architecture systems, specifically for fisheries MSMEs in North Sulawesi. The main focus of the literature study covers three groups of materials. First, fundamental theories and concepts regarding heterogeneous database integration, covering the characteristics of relational (SQL) and non-relational (NoSQL) database management systems, data storage models, differences in schema structures, and cross-platform data synchronization mechanisms. Literature related to interoperability, data federation, and middleware models for heterogeneous systems is also discussed.

Second, the researchers reviewed prior research and solutions on API development as middleware for system integration. This study covered RESTful API, GraphQL, and RPC-based API architectures, including their advantages and limitations in multi-architecture environments. The literature review also included an analysis of best practices for building APIs capable of handling differences in data structures, serialization formats (JSON, XML, BSON), authorization mechanisms (OAuth, JWT), and endpoint standardization to enable consistent operations across two databases.

Third, the study focused on frameworks, supporting technologies, and tools potentially used in the research, such as Node.js, Express.js, Spring Boot, Python Flask/FastAPI, or Golang as middleware development platforms. The researchers also reviewed database technologies such as MySQL, PostgreSQL (SQL), MongoDB, Cassandra, and Firebase (NoSQL), including differences in performance, transaction models, and compatibility with the middleware being developed. Literature on multi-platform architecture and integration challenges in small- and medium-scale systems such as fisheries MSMEs is crucial given the research context, which focuses on North Sulawesi Fisheries MSMEs. Furthermore, the researchers reviewed middleware development models implemented in the small- and medium-sized industry sector across various regions to gain insight into the common problems faced by MSMEs.

The results of this literature review were systematically analyzed to identify research gaps, map the strengths and weaknesses of existing approaches, and determine the most appropriate methodology and technology for the research. Thus, the results of the literature review provide an understanding of relevant theories and technologies, serving as a basis for formulating system requirements, designing middleware, and selecting an appropriate API architecture to support data integration in North Sulawesi fisheries MSMEs.

2.2. System requirements analysis

System requirements analysis is a system requirements analysis process that aims to comprehensively identify all functional and non-functional requirements that must be met by middleware development. In this study, the analysis was conducted by considering the characteristics of the fisheries MSME system in North Sulawesi, which uses two different types of databases: SQL and NoSQL, and has a heterogeneous system architecture. Researchers identified functional requirements, including the middleware's ability to perform two-way data synchronization, convert data structures between SQL and NoSQL, handle data reads and writes, and provide an API accessible to both systems.

In addition to functional aspects, non-functional requirements were also analyzed to ensure the middleware can operate stably in the MSME's operational environment. Non-functional requirements identified include data security, system reliability, synchronization process performance, API response time, and scalability. The middleware must maintain data integrity during transfer, protect access with authentication and authorization, and minimize the risk of data corruption due to differences in structure or access load. This analysis also includes the need for middleware compatibility with the hardware and software used by fisheries MSMEs, which generally have limited resources. The results of the requirements analysis are then used as a basis for designing the API architecture, selecting the most appropriate technology, and ensuring that the middleware built can effectively and securely bridge system differences.

2.3. API design

API design is a crucial phase in middleware development because it determines how communication, data processing, and system integration will be performed. At this stage, researchers designed the structure and architecture of an API capable of bridging two systems with different architectures and databases, namely SQL and NoSQL. The design process began by defining endpoints that represent the main functions required by fisheries MSMEs, such as managing production data, transactions, inventory, or distribution. Each endpoint was designed following RESTful principles to ensure the API is easily accessible, stateless, and interoperable across platforms.

In addition to endpoints, the design phase also determined the data exchange format, such as JSON, chosen for its flexibility and compatibility with both types of databases. Researchers also defined a data transformation mechanism to address the differ-

ences in schema structure between structured SQL and the more flexible NoSQL. This included designing data mapping, schema normalization, and data type conversion to ensure consistency between the two systems during synchronization.

Security aspects were also a crucial part of API design. The API design also defines authentication and authorization mechanisms, for example through the use of token-based authentication like JWT, to ensure that endpoint access is only granted by administrators. Furthermore, the API design includes error handling, standard HTTP status codes, and a uniform response structure to simplify integration management and analysis.

The API design also defines the API's internal communication flow, including request handling, routing, a business logic layer, and a database abstraction layer that connects the middleware to two heterogeneous databases. This design ensures that the API functions efficiently, does not disrupt existing MSME operational systems, and can adapt to evolving data synchronization needs. The API design serves as the foundation for middleware development, ensuring that the API implementation runs optimally and securely and integrates data from systems and databases with varying characteristics within the fisheries MSME ecosystem in North Sulawesi.

2.4. API implementation

API implementation is the realization phase of the previously formulated API design. This phase involves translating the entire architecture, endpoint structure, and data communication mechanisms into program code that can be integrated with systems using SQL or NoSQL databases. API implementation begins by building a development environment compatible with the selected technologies, using Node.js and Express.js, and establishing connections to both databases. Researchers implemented a database abstraction layer that serves as an intermediary between the API and both MySQL and MongoDB, enabling the API to interact with both relational and non-relational databases without disrupting the structure or operation of either system.

Each designed endpoint is implemented as a function that handles requests, processes business logic, validates data, and forwards the request to the relevant database. Implementation includes creating data transformation and schema-mapping mechanisms to ensure that SQL data can be correctly converted to a NoSQL-compatible format and vice versa. Furthermore, transaction control and data consistency management are implemented to address differences in the structure and storage properties of the two databases, particularly when bidirectional synchronization is required.

Each API operation was tested modularly during implementation to ensure no disruption to the existing fisheries MSME systems. Furthermore, researchers ensured that the middleware could run in resource-constrained environments, given that many MSMEs lack robust IT infrastructure. The implementation process also included performance adjustments, such as query optimization, index settings, and caching, if necessary, to ensure the API delivered good response times as data volumes increased. The API implementation resulted in functional, secure, and compatible middleware with heterogeneous system architectures, ensuring optimal integration of SQL and NoSQL systems within the operational environments of fisheries MSMEs in North Sulawesi.

2.5. API testing

API testing is a crucial process in this research to ensure that the developed middleware functions correctly, stably, and efficiently in integrating two heterogeneous database systems. Testing is conducted using two main approaches: functional testing and performance testing. In functional testing, each API endpoint is tested to verify that its implemented functions perform as specified. This testing includes verifying the API's ability to read, write, update, and delete data on both systems, as well as ensuring that data transformation and synchronization between SQL and NoSQL are accurate and consistent. Data integrity validation is performed at this stage to ensure that no data is lost, duplicated, or corrupted during synchronization. Performance testing is conducted to evaluate the API's ability to handle real-world workloads in the operational environment of a fisheries MSME. This testing includes measuring API response times, the middleware's ability to handle high request volumes, and the total time required to synchronize data across systems.

Performance testing also helps identify potential bottlenecks that could hinder the integration process, enabling optimizations before the middleware is fully operational. Through this series of structured tests, researchers ensured that the API not only functioned as intended but also delivered optimal, secure, and stable performance when integrating data into the multi-architecture system of fisheries MSMEs in North Sulawesi. These test results then served as the basis for evaluating and refining the API in the next phase.

2.6. 2.6 Evaluation and report

The evaluation and reporting phase is the final phase of this research, serving to comprehensively assess the effectiveness, reliability, and performance of the developed middleware and to document the entire research process. The evaluation is based on API testing results, both functional and performance, to ensure that the middleware can integrate SQL- and NoSQL-based systems consistently, accurately, and efficiently. At this stage, researchers assess the extent to which the API meets the functional requirements established in the analysis phase, including two-way synchronization capabilities, the management of data structure differences, and the accuracy of information transformation between platforms. Furthermore, non-functional aspects such as response time, system stability, error resilience, and the API's ability to operate within the limited infrastructure of MSMEs are also part of the evaluation.

The evaluation process involves reviewing performance benchmarking results, such as latency levels, the volume of processable data, and total synchronization time. Using these parameters, researchers analyze potential bottlenecks, data inconsistencies, and system weaknesses that could affect the middleware's future scalability.

After analyzing all evaluation results, the research provides recommendations for further development, including improving the synchronization algorithm, strengthening API security, and adapting the middleware for larger MSMEs. The evaluation and report stage not only assesses the success of the developed solution but is also important for developing a better integration system in the future.

3. RESULT AND ANALYSIS

3.1. System Development

There are two systems developed in the company, which have different architectures and data structures even though the data used is the same. The 2 systems are the sales information system which is used in making sales every day and the prediction system which is used to see predictive data for the future using sales data contained in the sales information system. The sales system can be seen in Figure 2.

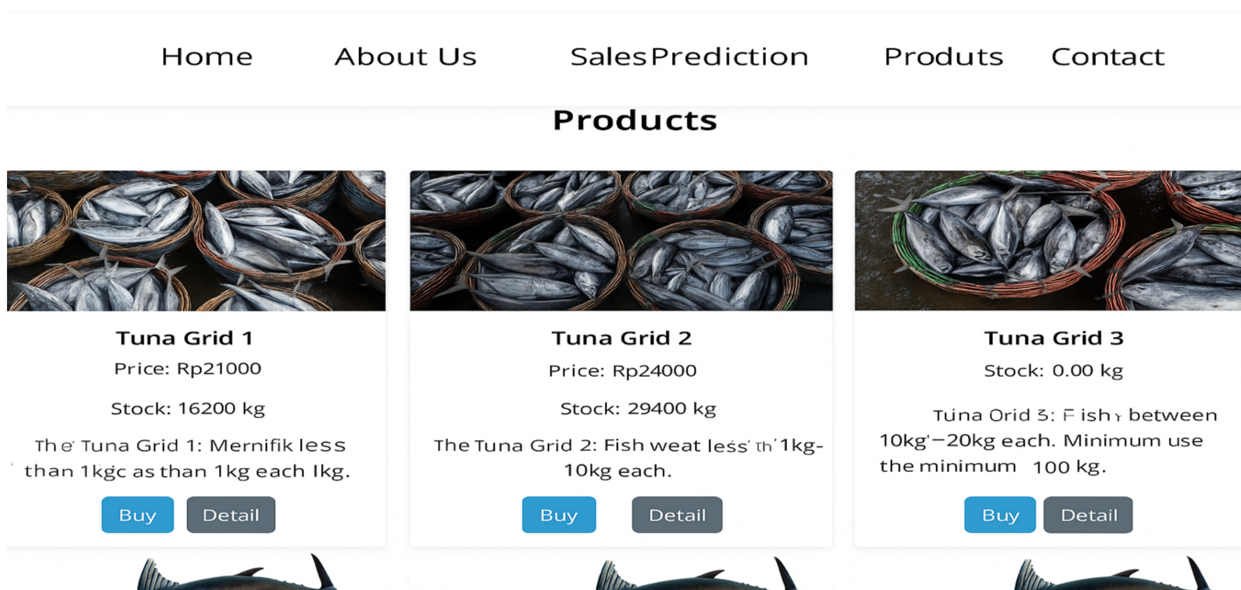


Figure 2. Display for sales system

The sales system is built on the Flask framework and uses NoSQL in data handling due to the delivery tracking feature embedded in the product delivery courier—sales prediction system, where this system runs on an Apache server by handling data using MySQL. The prediction system uses SQL-based data because it is used to see future sales prediction data. With structured data, it makes it easier for prediction linear regression algorithm to run well. The prediction system for product sales can be seen in Figure 3.

Figure 3. Prediction system for product sales

3.2. Implementation of API on the system

Research was conducted with the fetchData() main function, which is used to construct a response structure consisting of three elements, namely status: HTTP code (200 signifying success), message: status message, and data: containing an array of sales prediction data in kilograms. The response structure is then converted into JSON format using the json_encode() function and sent with the Content-Type: application/json header setting. The API flowchart for the prediction system can be seen in Figure 4.

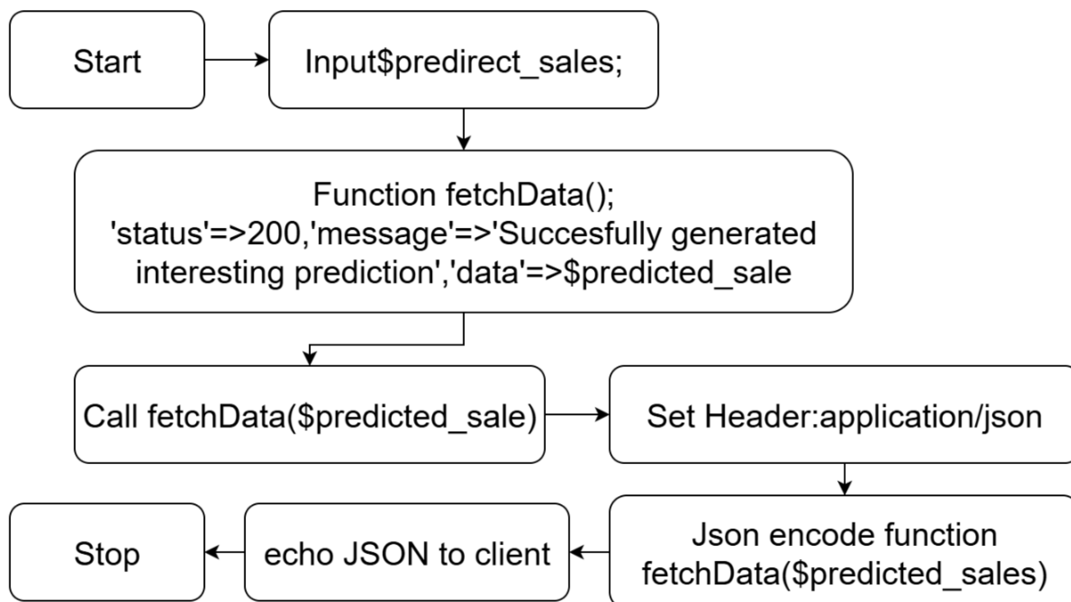


Figure 4. API flowchart for the prediction system

In the Sales System, an API was created to allow the prediction system to pull transaction data from the sales database automatically. This API is built with the Flask framework and provides transaction data based on the time range specified by the from and to parameters. The resulting data is packaged in JSON format for easy access and processing by the prediction system. The API pseudocode for the sales system can be seen in.

```
start
Function: get_orders(from_date, to_date)
Input:
- from_date:(format: YYYY-MM-DD)
- to_date:(format: YYYY-MM-DD)
proccess:
1. Create a query
2. If from_date and to_date are
available:
-Create a query filter based on
the field 'order_date'
-order_date must be in between
from_date hour 00:00:00 and
to_date hours 23:59:59
3. Retrieve data from the database
(orders_collection) according to
filter query and convert the
result to list form
4. For each order data:
- Change the value of field_id field value to a string
- convert to JSON
Output:
- JSON response:
{
status: 200,
message: "Berhasil menarik
data prediksi",
data: [list data order]
}
stop
```

As shown in the pseudocode, the main function used in this API is get_orders(). The initial input is to enter the from_date and to_date data according to the format. Next in the process, a query filter is run based on the order_date that has been created. Then based on the query function, the API retrieves data from the database and converts it into a list with the condition that the field_id value must be changed to a string. Furthermore, all retrieved data is converted to JSON format. If all processes are complete, the output is a JSON-based response. This implementation allows efficient data exchange between two systems built with different technologies, namely Python and PHP.

API development begins on the prediction system. In the prediction system, the implementation process begins with creating a native PHP API that provides sales prediction data in JSON format. This API packages the data from the linear regression algorithm's previously run prediction results into a format that can be directly accessed and utilized by the sales system.

This API is then accessed by the sales system through the JavaScript fetch API, without re-storing the data. The fetched data is displayed as a line graph using the Chart.js library. This allows real-time, interactive visualization of prediction data, as shown in Figure 5.

From Figure 4, it can be concluded that the implementation of the API in a Linear Regression-based sales prediction system has proven to be able to simplify data integration between different database systems. Visualizing prediction results through an interactive interface improves the effectiveness of managerial decision-making.

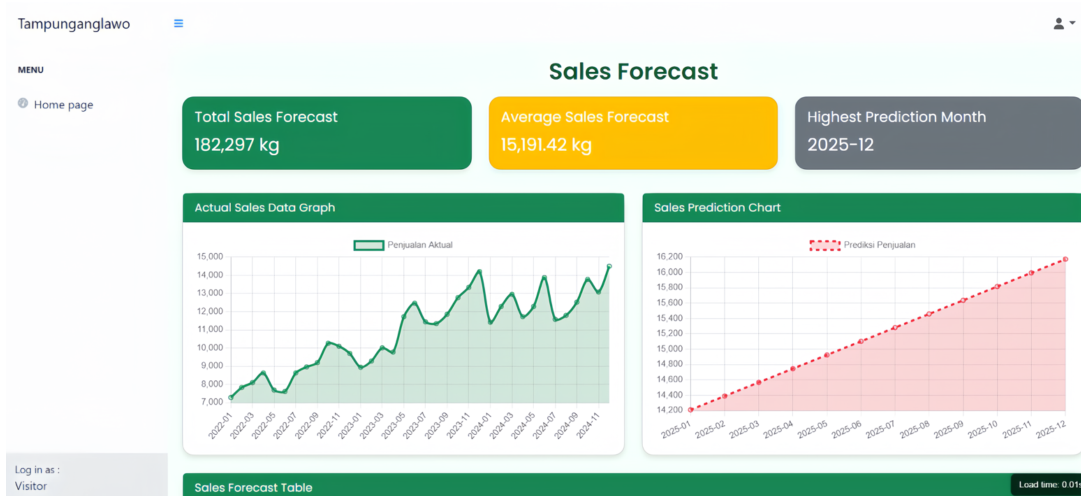


Figure 5. Visualization of sales forecasting

3.3. API Function Testing

In functionality testing, the embedded API is run to see if the input and output are correct or not. This is done by accessing the API generated by the prediction or sales system. This way, we can determine whether the resulting output matches the desired results. The results of the prediction system API testing are shown in Table 1.

Table 1. Testing on the prediction system API

Tested function	Input	Output	Result
Access API on sales Prediction system	Access api.php file	Display JSON data	Data displayed
Fetch data via API	Using the fetch() function	Retrieval of prediction data displayed on the sales system prediction chart	The graph appears according to the data

Table 1 shows the functional testing results of the API developed for the sales prediction system. This testing aimed to ensure that the API functioned as intended, both in sending and displaying prediction data to the sales system. The first scenario aimed to ensure that the API could provide a response according to specifications when accessed directly. The api.php API is the primary endpoint that sends prediction data calculated using the Linear Regression algorithm from a MySQL database. The test results showed that the API responded correctly, producing output in JSON format containing prediction values such as month, total sales, and growth rate. This indicates that the connection between the server, database, and data processing logic was running normally without errors. Success at this stage proves that the API has fulfilled its basic function as a data provider for other external systems. The second scenario tested the integration between the prediction system and the sales system using a real-time data communication approach. The sales system, built using the Flask framework (Python) and the MongoDB database, invoked data from the prediction system's API using the fetch() function. The test results showed that the prediction data was successfully retrieved and visualized using Chart.js, without the need for re-storage in the sales system database. The graph that appears shows the prediction trend that matches the calculation results in the prediction system, which means that communication between the systems was carried out correctly and efficiently. In the next scenario, the fetch () function is activated where the desired output is that the data stored in the prediction system can also be displayed to the sales system in the form of graphs. The results of the 2 scenarios show that all scenarios produce appropriate results. Furthermore, testing from the sales system side. In the sales system, the test is carried out by synchronizing sales data to the prediction system database via API. The results of the data synchronization in the sales prediction system can be seen in Figure 6.

Figure 6 shows the interface display of the data synchronization results in the sales prediction system after the API integration was successfully implemented. This page is part of the transaction management module, where sales data from the sales system (Flask–MongoDB) has been automatically synchronized into the prediction system that has been developed.

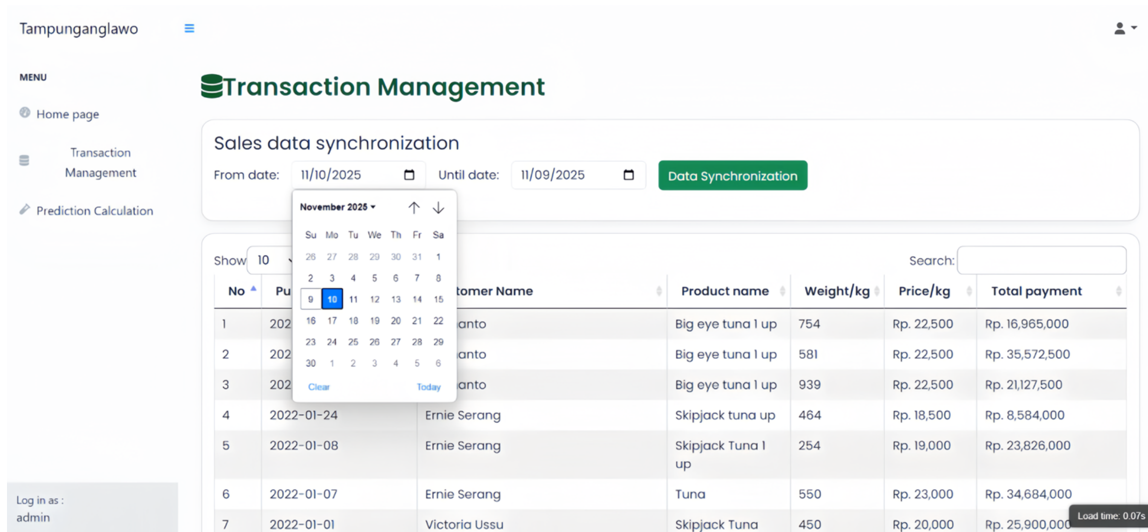


Figure 6. Data synchronization results on sales prediction system

Table 2 describes the function testing of the API in the sales system. In the sales system, 2 conditions are tested, namely access to the prediction system API and synchronization of sales data. In the first condition, the input given from the sales system API is access to prediction data based on the existing time where the desired output is JSON data containing prediction data. In the second condition, is to synchronize where there is a synchronize button in the prediction system which retrieves sales data to be used in the prediction system. In this case the API successfully provides sales data without duplication to the prediction system. This way, testing on 2 conditions produces the right results according to the input given.

Table 2. Function Testing On Sales System API

Tested Function	Input	Output	Result
Access Prediction API	Access prediction data based on time unit	JSON containing prediction data	JSON Data
Synchronize to prediction system	Click the synchronize button in the prediction system	Sales data is saved without duplication	Data is saved

Table 2 shows the test results of the main API functions in the sales system after successful integration with the prediction system. The first test was conducted to ensure that the data communication process between the two systems ran as expected, both in terms of predictive data retrieval and sales data synchronization. This function aimed to test whether the sales system could retrieve predicted sales data from the prediction system via the API with specific time parameters. When the user selects a time period, the sales system sends a request to the API endpoint in the prediction system. The test results showed that the request was successfully processed and produced output in JSON format containing the results of the prediction algorithm calculations (Linear Regression). This JSON data was then displayed on the sales system interface, such as in a prediction graph or report table. The success of this function demonstrated that the API can function bidirectionally, not only sending data to the prediction system but also receiving analysis data from the system in real time. The second test was conducted to verify the API's ability to synchronize sales data from the Flask–MongoDB system to the Apache–MySQL prediction system. When the user clicked the synchronize button, the API sent all the most recent transaction data to the prediction system. On the prediction system side, the data is stored with automatic checks to avoid duplicate entries (for example, based on transaction ID or timestamp). Test results show that the synchronization process is stable and accurate new data is successfully stored in the prediction system without any reproduction of pre-existing data.

3.4. API Performance Testing

In this test, the API in the prediction system will be tested to synchronize data from the sales system. This is made to see the performance of the API in synchronizing the required data. This is done because the prediction system has a repetitive frequency of pulling sales data based on data from the sales system. The synchronization scheme can be seen in Figure 7. In this test, synchronization is carried out on several sales data sets in order to see several aspects, namely the amount of data, the speed of the server in responding, and the amount of data sent.

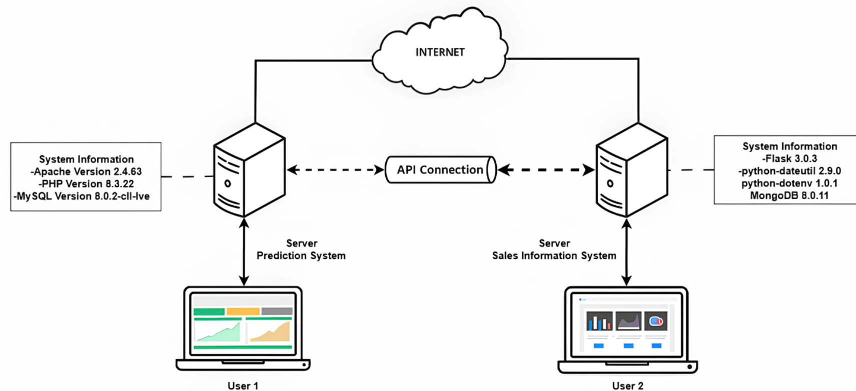


Figure 7. System testing topology scheme

Figure 7 depicts the topology scheme of the system testing (system topology testing) used to test communication and data integration between two systems of different architectures, namely: the Prediction System and the Sales System which are connected via the Application Programming Interface (API). This topology testing proves that the system is able to run in an integrated manner in multi-platform and multi-database playback, thus supporting the efficiency of data-based decision making in the company. An example of observation in this test can be seen in Figure 8. The system is synchronized and then inspected for information related to the data that has been synchronized. The measurement method involves the user synchronizing a certain amount of data from the database to the system. Then, several criteria are inspected, such as the amount of data, the server's response time, and the total time required for synchronization.

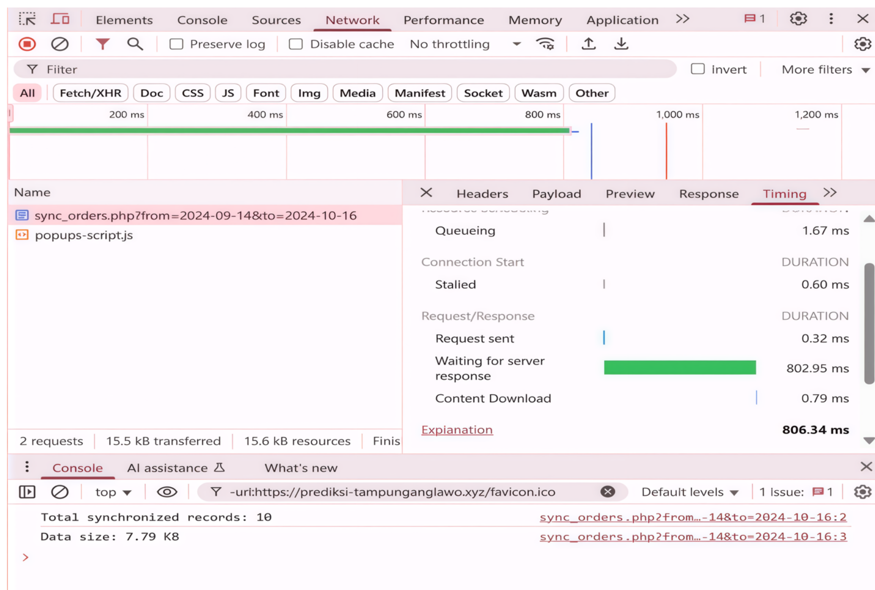


Figure 8. Data synchronization via API

From Figure 8, it can be seen that the results of data synchronization performance testing via the API using developer tools were used to measure the API's performance in sending and receiving data between systems, especially between the sales system and the prediction system. From the test results, it is known that the request status is 200 OK (meaning the API is running normally

without errors), the data size (payload) is approximately 6.8 KB, and the total synchronization time is ± 806 ms. These values indicate that the API can synchronize small to medium-sized datasets in under 1 second, which is very fast and efficient for web-based transactions. In addition, the Timing Breakdown section shows the process stages that go through data Queuing & Waiting with the waiting time before the sending process begins, Request Sent & Waiting (TTFB) with the time required for the server to respond for the first time, and Content Download with the time to download the response results from the server. From the test results, most of the time is spent waiting for the server response, indicating that the main processing time occurs on the server-side API, not in the data download process.

After the synchronization test, the results are divided by the amount of data the API performance evaluation measured across varying levels of synchronization. Three performance indicators were assessed: data size, server response time, and total synchronization time. The API Performance test results are shown in Table 3.

Table 3. Pembagian data untuk Training dan Testing

Amount of Data	Data Size	Server Response	Timing
1 data	0.9 kb	759.64 ms	763.68 ms
10 data	7.79 kb	802.95 ms	806.34 ms
50 data	36.9 kb	1.42 s	1.83 s

From Table 3 above, it can be seen that for a single record (0.9 kb), the server response time was 759.64 ms, with a total synchronization time of 763.68 ms. When the dataset increased to 10 records (7.79 kb), the server response time rose slightly to 802.95 ms, and the synchronization time reached 806.34 ms. At the largest tested volume of 50 records (36.9 kb), the server response time was 1.42 seconds, and the total synchronization time was 1.83 seconds. The results demonstrate that, despite the fivefold increase in data size from 10 to 50 records, the total synchronization time increased only marginally. Specifically, synchronizing 50 records took just 1.83 seconds, representing only a 41.7% increase over the smallest dataset, rather than the expected linear growth of approximately 50 times. This finding indicates that the developed API exhibits efficient scalability and minimizes synchronization overhead even as data volumes increase. These outcomes provide empirical evidence that the API can maintain stable performance across heterogeneous systems while ensuring both data integrity and time efficiency. Such performance efficiency is particularly advantageous in real-world applications, where large-scale data synchronization is required without compromising system productivity.

4. CONCLUSION

This research successfully developed an Application Programming Interface (API) to integrate two systems with different architectures and databases, namely a Flask–MongoDB (NoSQL)-based sales system and an Apache MySQL (SQL)-based sales prediction system at the Tampunglawo fisheries company in North Sulawesi. The test results showed that: (1) In functional testing, the API functioned well on both systems. The prediction system could send Linear Regression calculation results in JSON format, while the sales system could retrieve and display the prediction data in real time using Chart.js. (2) In synchronization testing, the API could transfer sales data from MongoDB to MySQL without duplication. (3) In performance testing, the API demonstrated high efficiency with an average response time of under 1 second for small to medium data. Even for the most significant data volume (50 records, 36.9 KB), synchronization time increased by only 41.7% compared to the smallest, indicating efficient performance scaling and high stability. The results of this study demonstrate that RESTful APIs are effective for connecting heterogeneous systems, improving business process efficiency, and reducing time and effort in data synchronization. The implications of these findings are twofold. First, the developed API significantly reduces the time and manual effort traditionally required for data restructuring, thereby enhancing overall system productivity. Second, the API demonstrates the potential to support large-scale data synchronization tasks without substantial performance degradation, making it highly suitable for Tampunglawo company environments where real-time or near-real-time interoperability between systems is critical. In future system development, server optimization is required to further reduce data synchronization time between the two operational systems. Another potential development involves the integration of cloud-based services to facilitate system management within the Tampunglawo enterprise. In addition, future research should focus on extending the API beyond its current application for MSMEs such as Tampunglawo, toward broader adoption by other MSMEs or larger enterprises with diverse business segments.

5. ACKNOWLEDGEMENTS

We would like to thank all our friends at Politeknik Negeri Manado, Manado, Indonesia, and Universitas Lambung Mangkurat, Banjarmasin, Indonesia, for their support with this project.

6. DECLARATIONS

AI USAGE STATEMENT

During the preparation of this work, the authors used ChatGPT (OpenAI) to improve the language and clarity of the manuscript. After using this tool, the authors reviewed and edited the content as needed and took full responsibility for the publication's content.

AUTHOR CONTIBUTION

Christopel H. Simanjuntak, Musfiah, Muhammad Bahit, Cristovani W. Lohonauman, Stenly B. Dodie, and Khamla Nonalinsavath contributed jointly to conceptualization, methodology, validation, and data interpretation. Their contributions also include writing the initial manuscript, developing the system, visualizing results, and conducting experiments.

COMPETING INTEREST

The authors declare that there are no conflicts of interest.

REFERENCES

[1] T. Taipalus, "Database management system performance comparisons: A systematic literature review," *Journal of Systems and Software*, vol. 208, p. 111872, Feb. 2024, https://doi.org/10.1016/j.jss.2023.111872.

[2] A. Krasnikov, E. Romanova, and O. Kireeva, "Development of fishery information system for production processes organization," *BIO Web of Conferences*, vol. 83, p. 03005, 2024, https://doi.org/10.1051/bioconf/20248303005.

[3] R. Kurniawan, "Application of Random Forest Algorithm on Credit Risk Analysis," *Procedia Computer Science*, vol. 245, pp. 740–749, 2024, https://doi.org/10.1016/j.procs.2024.10.300.

[4] G. E. Mushi, P.-Y. Burgi, and G. Di Marzo Serugendo, "State of Agricultural E-Government Services to Farmers in Tanzania: Toward the Participatory Design of a Farmers Digital Information System (FDIS)," *Agriculture*, vol. 14, no. 3, p. 475, Mar. 2024, https://doi.org/10.3390/agriculture14030475.

[5] I. M. Putrama and P. Martinek, "Heterogeneous data integration: Challenges and opportunities," *Data in Brief*, vol. 56, p. 110853, Oct. 2024, https://doi.org/10.1016/j.dib.2024.110853.

[6] C. Li, Y. Chen, and Y. Shang, "A review of industrial big data for decision making in intelligent manufacturing," *Engineering Science and Technology, an International Journal*, vol. 29, p. 101021, May 2022, https://doi.org/10.1016/j.jestch.2021.06.001.

[7] S. Benjelloun, M. E. M. El Aissi, Y. Lakhrissi, and S. El Haj Ben Ali, "Data Lake Architecture for Smart Fish Farming Data-Driven Strategy," *Applied System Innovation*, vol. 6, no. 1, p. 8, Jan. 2023, https://doi.org/10.3390/asi6010008.

[8] C. A. Győrödi, D. V. Dumşe-Burescu, D. R. Zmaranda, R. Ş. Győrödi, G. A. Gabor, and G. D. Pecherle, "Performance Analysis of NoSQL and Relational Databases with CouchDB and MySQL for Application's Data Storage," *Applied Sciences*, vol. 10, no. 23, p. 8524, Nov. 2020, https://doi.org/10.3390/app10238524.

[9] J. I. Janjua, T. A. Khan, S. Zulfiqar, and M. Q. Usman, "An Architecture of MySQL Storage Engines to Increase the Resource Utilization," in *2022 International Balkan Conference on Communications and Networking (BalkanCom)*. Sarajevo, Bosnia and Herzegovina: IEEE, Aug. 2022, pp. 68–72, https://doi.org/10.1109/BalkanCom55633.2022.9900616.

[10] A. Rehman, S. Naz, and I. Razzak, "Leveraging big data analytics in healthcare enhancement: Trends, challenges and opportunities," *Multimedia Systems*, vol. 28, no. 4, pp. 1339–1371, Aug. 2022, https://doi.org/10.1007/s00530-020-00736-8.

[11] J. Chaudhary, V. Vyas, and C. K. Jha, "Qualitative Analysis of SQL and NoSQL Database with an Emphasis on Performance," in *IOT with Smart Systems*, J. Choudrie, P. Mahalle, T. Perumal, and A. Joshi, Eds. Singapore: Springer Nature Singapore, 2023, vol. 312, pp. 155–165, https://doi.org/10.1007/978-981-19-3575-6_18.

[12] E. Lupu, A. Olteanu, and A. D. Ionita, "Concurrent Access Performance Comparison Between Relational Databases and Graph NoSQL Databases for Complex Algorithms," *Applied Sciences*, vol. 14, no. 21, p. 9867, Oct. 2024, https://doi.org/10.3390/app14219867.

[13] Sasmoko, Y. Indrianti, S. R. Manalu, and J. Danaristo, "Analyzing Database Optimization Strategies in Laravel for an Enhanced Learning Management," *Procedia Computer Science*, vol. 245, pp. 799–804, 2024, https://doi.org/10.1016/j.procs.2024.10.306.

[14] I. vSuvster and T. Ranisavljevic, "Optimization of MySQL database," *Journal of Process Management and New Technologies*, vol. 11, no. 1-2, pp. 141–151, 2023, https://doi.org/10.5937/jouproman2301141Q.

[15] M. Boyd, L. Vaccari, M. Posada, and D. Gattwinkel, "An Application Programming Interface (API) framework for digital government," *Publications Office of the European Union*, 2020.

[16] L. Vaccari, M. Posada, M. Boyd, and M. Santoro, "APIs for EU Governments: A Landscape Analysis on Policy Instruments, Standards, Strategies and Best Practices," *Data*, vol. 6, no. 6, p. 59, Jun. 2021, https://doi.org/10.3390/data6060059.

[17] M. A. Ali and S. M. Salih, "Impact of Application Programming Interfaces (APIs) Economy on Digital Economics in Saudi Arabia," *Sustainability*, vol. 17, no. 9, p. 4104, May 2025, https://doi.org/10.3390/su17094104.

[18] P. Gowda and A. N. Gowda, "Best Practices in REST API Design for Enhanced Scalability and Security," *Journal of Artificial Intelligence, Machine Learning and Data Science*, vol. 2, no. 1, pp. 827–830, Feb. 2024, https://doi.org/10.51219/JAIMLD/priyanka-gowda/202.

[19] I. R. D. Muhammad and I. V. Paputungan, "Development of Backend Server Based on REST API Architecture in E-Wallet Transfer System," *Jurnal Sains, Nalar, dan Aplikasi Teknologi Informasi*, vol. 3, no. 2, pp. 79–87, Jan. 2024, https://doi.org/10.20885/snati.v3.i2.35.

[20] J. Shang, L. Xiao, Z. Wu, J. Yang, Z. Xiao, J. Wang, Y. Zhang, X. Chen, J. Wang, and H. Li, "GvdsSQL: Heterogeneous Database Unified Access Technology for Wide-Area Environments," *Electronics*, vol. 13, no. 8, p. 1521, Apr. 2024, https://doi.org/10.3390/electronics13081521.

[21] B. Namdeo and U. Suman, "A Middleware Model for SQL to NoSQL Query Translation," *Indian Journal of Science and Technology*, vol. 15, no. 16, pp. 718–728, Apr. 2022, https://doi.org/10.17485/IJST/v15i16.2250.

[22] S. Bhatt, "Best Practices for Designing Scalable REST APIs in Cloud Environments," *Journal of Sustainable Solutions*, vol. 1, no. 4, pp. 48–71, Oct. 2024, https://doi.org/10.36676/j.sust.sol.v1.i4.26.

[23] Y. Mesmoudi, M. Lamnaour, Y. El Khamlichi, A. Tahiri, A. Touhafi, and A. Braeken, "A Middleware based on Service Oriented Architecture for Heterogeneity Issues within the Internet of Things (MSOAH-IoT)," *Journal of King Saud University - Computer and Information Sciences*, vol. 32, no. 10, pp. 1108–1116, Dec. 2020, https://doi.org/10.1016/j.jksuci.2018.11.011.

[24] H. Valentine and B. Kabaso, "Design and Development of a Unified Query Platform as Middleware for NoSQL Data Stores," *International Journal of Advanced Computer Science and Applications*, vol. 15, no. 7, 2024, https://doi.org/10.14569/IJACSA.2024.0150762.

[25] F. F. Yudan and M. Arief Virgy, "Implementasi Open Government Data oleh Pemerintah Kota Bandung," *Jurnal Transformative*, vol. 7, no. 1, pp. 128–153, Mar. 2021, https://doi.org/10.21776/ub.transformative.2021.007.01.6.

[26] Y. Ikhwani, A. Ramadhan, M. Bahit, and T. H. Faesal, "Single elimination tournament design using dynamic programming algorithm," *MATRIK : Jurnal Manajemen, Teknik Informatika dan Rekayasa Komputer*, vol. 23, no. 1, pp. 113–130, Nov. 2023, https://doi.org/10.30812/matrik.v23i1.3290.

[27] A. Niarman, Iswandi, and A. K. Candri, "Comparative Analysis of PHP Frameworks for Development of Academic Information System Using Load and Stress Testing," *International Journal Software Engineering and Computer Science (IJSECS)*, vol. 3, no. 3, pp. 424–436, Dec. 2023, https://doi.org/10.35870/ijsecs.v3i3.1850.

[28] G. Locicero, A. D. Maria, S. Alaimo, and A. Pulvirenti, "MASFENON: Implementing a multi-agent simulation framework for interconnected networks with distributed programming," *Procedia Computer Science*, vol. 255, pp. 73–82, 2025, https://doi.org/10.1016/j.procs.2025.02.262.

[29] K. N. Markert, G. Da Silva, D. P. Ames, I. Maghami, G. P. Williams, E. J. Nelson, J. Halgren, A. Patel, A. Santos, and M. J. Ames, "Design and implementation of a BigQuery dataset and application programmer interface (API) for the U.S. National Water Model," *Environmental Modelling & Software*, vol. 179, p. 106123, Aug. 2024, https://doi.org/10.1016/j.envsoft.2024.106123.

[This page is intentionally left blank.]