Available at https://journalenrichment.com/index.php/jr/

Enrichment: Journal of Multidisciplinary Research and Development

Construction and Optimizing Domain Driven Design Principles and Twelve Factor Application Methodology for Software Backend Architecture Risky

Risky Kurniawan*, Rangga Sanjaya
Universitas Adhirajasa Reswara Sanjaya, Indonesia
Email: riskykurniawan15@gmail.com*

ABSTRACT

Software architecture plays a crucial role in software development; however, finding an architecture that remains continuously adaptable, employs a declarative format, supports scaling, and is suitable for modern cloud platforms is challenging. Additional challenges often arise due to business and technical obstacles during development. This study aims to design a software architecture that supports continuous development, enables declarative deployment, and incorporates clean contracts based on the twelve-factor app methodology. The goal is to create an architecture applicable to modern cloud platforms, supporting scalability and leveraging domain-driven design to eliminate business and technical constraints. The research follows a waterfall methodology, starting with analysis, followed by system design, implementation, testing, and delivery. The findings indicate that the proposed architecture exhibits continuous development characteristics and adopts the twelve-factor app principles, resulting in a descriptive format and clean contracts. The architecture is also compatible with modern cloud platforms and supports scaling. Additionally, the use of domain-driven design enhances resource allocation effectiveness and helps eliminate business and technical limitations.

Keywords: Software Architecture, Backend, Domain-Driven Design, Twelve-Factor App

INTRODUCTION

Technological advancements in building information systems have resulted in the development of web service technologies such as SOAP (XML), RestApi (JSON), and gRPC (Protobuf) (Marii & Zholubak, 2022; Sharma, 2023; Yellavula, 2020). These technologies improve the efficiency of message exchange and enable new data communication methods. As a result, microservices-based architectures have been adopted for their increased flexibility and streamlined workflows (Ait Said et al., 2024; Raj & David, 2021).

Software deployment methods have undergone a transformation, transitioning from SCP (file transfer) to CI/CD (Continuous Integration and Continuous Delivery) with a container-based approach (Shah et al., 2020). CI/CD is widely recognized for its ability to mitigate the risk of software malfunction post-deployment (Thatikonda, 2023; Karunarathne et al., 2024; Saieva & Kaiser, 2022; Throner et al., 2023). In this context, software architecture assumes a pivotal role in software development. One of the key challenges is to design a robust software architecture that remains adaptable and can accommodate various changes. In the contemporary landscape, software is often delivered as a service, such as web applications or Software as a Service (SaaS) (Raghavan R et al., 2020; Ali et al., 2020; Li & Kumar, 2022; Pervin, 2021; Lopez-Viana et al., 2020). The twelve-factor app methodology is particularly well-suited for building applications as a service, leveraging a declarative format, clean contract usage with operating systems, compatibility with modern cloud platforms, minimizing

differences between development and production environments, and enabling upgrades without significant changes (Telang, 2022; Sangapu et al., 2022; Indrasiri & Suhothayan, 2021; Stüber & Frey, 2021).

Domain-driven design, a methodology that has emerged over the past two decades, is a powerful tool in software architecture for developing information systems (Oukes et al., 2021; Özkan et al., 2021; Hofer & Schwentner, 2021; Vural & Koyuncu, 2021; Singjai et al., 2021). It is highly regarded for its effectiveness in developing systems with complex business processes, providing tools to prevent miscommunication between business teams and software development teams. The twelve-factor app methodology, with its emphasis on practicality and adaptability, serves as a guiding light for architectural development, making it particularly suitable for projects with high complexity (Khan et al., 2021; Kosińska et al., 2023).

It is challenging to find a software architecture that is ready to use, can embrace technological changes, and allows for the practical implementation of new technologies. Many architectures still need a declarative format, which can significantly impact time and cost, especially when new developers are involved. Additionally, many architectures must be optimized for modern cloud platform technologies, allowing them to scale up without requiring changes to development practices.

While the adoption of Domain-Driven Design (DDD) for enhancing modularity and clarity in complex software systems is well-supported—Özkan et al. (2023) highlight DDD's benefits in defining clean domain models, bounded contexts, and facilitating decomposition in microservices architectures─yet they note the lack of empirical or implementation-focused validation in real-world cloud-native services. Similarly, research on the Twelve-Factor App methodology emphasizes its effectiveness in guiding cloud-native, SaaS-oriented architectures (e.g., Heroku's principles ensuring portability, declarative setup, and scalable deployment), but does not examine how it integrates with DDD practices for backend application boundary definitions.

The author is interested in designing a software architecture with backend application boundaries using domain-driven design principles and the twelve-factor app methodology. The aim is to adapt to and practically accept technological changes and be compatible with modern cloud platform technologies. The contribution lies in providing a practical, ready-to-use architectural blueprint that simplifies onboarding, supports evolving technologies, and streamlines development cost and time—benefiting software architects and development teams working in dynamic cloud-native ecosystems.

METHOD

This research aims to design a backend software architecture using the waterfall software development method in Figure 1, which is part of the System Development Life Cycle (SDLC). The process begins with an analysis of the software's needs and objectives, followed by designing the system or software architecture. The design is then implemented in code, tested, and delivered.

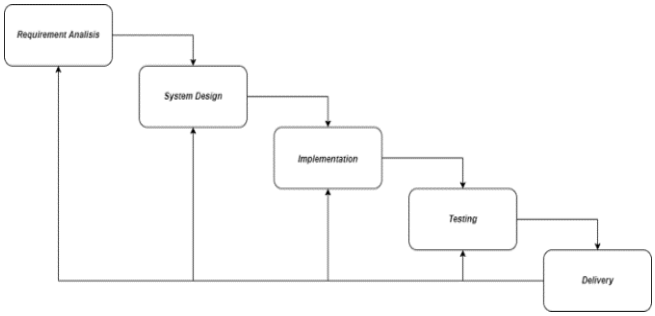

Figure 1. Method Research

Requirement Analysis

Requirement analysis involves gathering relevant data from literature reviews and technological advancements. The next step is to analyze the criteria for the software architecture, which must meet the ease of implementing new technologies, a declarative format for automation setup, clean contracts for operating system use, compatibility with modern cloud platforms, ease of deployment, ability to scale up without significant changes in tooling, architecture, or development practices, and minimize software development barriers by eliminating the gap between business and technical aspects.

Our strategy for meeting these criteria involves creating a software architecture that allows easy integration and detachment of technology without over-reliance on individual components. A key aspect of this strategy is adhering to the methodology of the twelve-factor app, a set of principles for building modern, scalable, and maintainable software-as-a-service applications. This standardization and presentation as a service, whether web applications or SaaS (Software as a Service), ensures the architecture's usability as either monolithic or microservices-based services and compatibility with modern cloud platforms.

The next step in our proposal is for the architecture to implement the principles of domain-driven design. This software development approach is not just about writing code, but about developing a domain model that comprehends the domain's processes and rules. It is centered on collaboration between technical experts and domain experts, ensuring that the software architecture aligns with the business needs and requirements.

System Design (Design)

The system design phase involves designing the architecture layer framework and modelling the software architecture system. The architecture layer framework in Figure 2 is modified from the Traditional N-layered architecture framework, which consists of presentation, business, and data layers. The architecture layers include application layers, interface layers, and domain layers. Application layers are used for program initialization, process setup, and managing application scalability. Interface layers contain user methods for accessing the application and interacting with software users. Domain layers contain services, models, and repository layers, handling business process logic. The layered architecture structure separates responsibilities and enhances application scalability. All components are easily detached and reattached, allowing the architecture to adapt to practical technology.

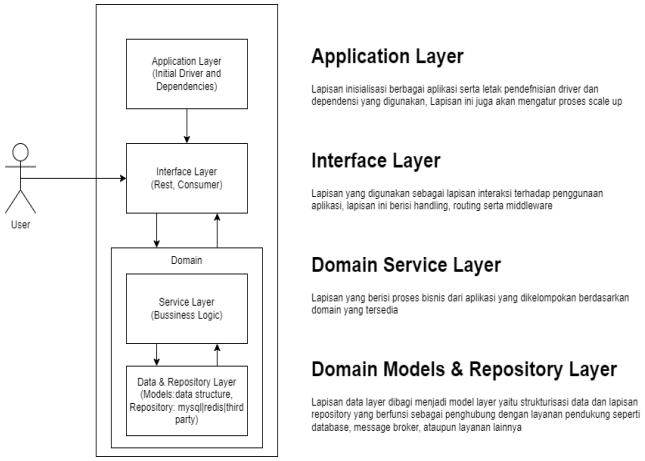

Figure 2. Architectural Layer Design

During the system modelling phase, the software architecture is modelled using Unified Modeling Language (UML), a widely used standard language in the industry. UML can define requirements, conduct analysis, design system workflows, and illustrate architecture in object-oriented programming. This approach ensures that the architecture can be easily detachable and reattached, allowing the architecture to adapt to practical technology.

Implementation

The implementation process involves creating a software architecture by writing it in a specific programming language, such as Go (Golang). The outcome of the implementation is a base code that can be directly utilized for software development.

Testing

This study focuses on software testing to minimize mistakes and flaws in new architectures. Black box or functional testing assesses functionality based on input and output behavior without examining the codebase's structure. It involves running each component in the application and observing its compliance with the system design. Load testing assesses the architecture's resilience under varying conditions and surges in software users. The main benchmarks for load testing are response time and error rate. This study uses Apache JMeter software for load testing, which serves as performance testing to ensure the software functions as expected under various circumstances. Both methods ensure that the software functions as expected under various circumstances.

Delivery

The next stage is delivery once the testing process is complete and the architecture is deemed satisfactory. This phase involves distributing the software as a repository while including a license in the program's codebase. Each delivery process will always produce a new version of the software. In this study, the type of license to be used is MIT, allowing users to utilize the software for commercial purposes, modification, distribution, and personal use. Semantic versioning will be employed using the major, minor, and patch format (1.0.0).

RESULTS AND DISCUSSION

Architecture Overview

The software's architecture concept in Figure 3 is intended for building backend applications. It is built using the Go programming language, which was chosen for its advantages, particularly in handling concurrency. This allows the language to execute programs using multiple threads and non-blocking operations.

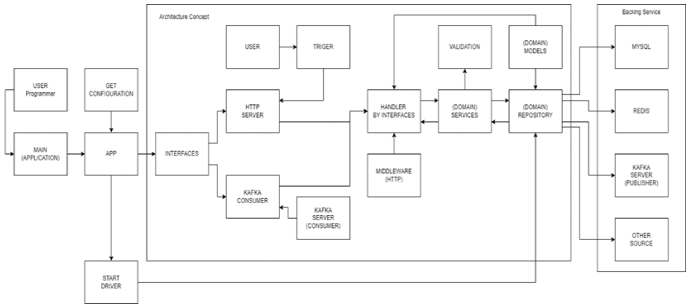

Figure 3. Architectural Overview

The software architecture developed comprises two distinct programs packaged within one architectural type. The first application, ELSA (Electronic Smart Assistant), is designed to handle all forms of administrative processes. The second application, Xarch (X Architecture), operates business services based on a domain and offers various options. The software architecture is designed to facilitate the reuse of each component and minimize dependency on various existing components. This approach aims to simplify the integration of new technologies, both supporting and data communication methods.

The architecture overview includes several architecture blocks with different functions and purposes. The first block is the application layer, which is the program's initiation layer and handles configuration reading and driver initialization. Next is the interfaces block, which provides a gateway for interaction between software service users and specific methods and technologies, consisting of HTTP server services and consumers in its current implementation. The next block is the domain, initialized according to domain-driven design principles. The final block is backing services, which support the software architecture.

Architectural Characteristics

Architectural characteristics describe the expectations of software architecture at both operational and technical levels, encompassing its core purpose, quality standards, and division of responsibilities among different units. Continuous development is a crucial characteristic of our architecture, allowing developers to evolve and implement new technologies. Our architecture is designed to embody continuous development, with services divided into small, loosely coupled units, enabling developers to upgrade the existing architecture by integrating or removing technologies efficiently.

Multiple applications or mono repos are unique characteristics of our architecture. They allow developers to create one or more applications within the same architectural scope or codebase. This high reusability characteristic is particularly beneficial when there are similarities in business processes. Reusability is essential, as the classification and separation of processes into smaller components enable the reuse of previously defined processes.

The developed architecture supports horizontal scale-out processes within the application's interface block. Developers can scale by providing arguments for interfaces to be

scaled according to the definitions in the application block. However, the architecture does not currently support automatic scaling. Our architecture offers flexibility, adaptability, and a strong foundation for software development.

Architectural Layer Framework and Rules

The architecture adopted the Traditional N-layered architecture design, consisting of three distinct yet interconnected layers: the application, interfaces, and domain.

The application layer defines programs to be executed based on interactions initiated by developers, facilitating the initiation of projects and creating applications within the same codebase.

The interface layer initializes input and output unit services technologies, limiting developers' ability to provide instructions in the form of business processes or logic sequences. The application can only accept requests and display responses generated by the domain layer's processing units. Handlers contain instructions for retrieving parameters or request bodies, which are forwarded to the next layer according to the predefined payload contract for the target domain.

The domain layer is divided into sub-layers, including domain services, domain data and repository, and domain data and model. The services layer prohibits direct interaction with support services and dependencies, while the domain data and repository layer should not contain business processes or mathematical calculations. The domain layer is built according to the requirements and plans established by domain experts or individuals who deeply understand the purpose of the application.

In practice, the interfaces layer cannot use the domain layer directly, and developers must register the domain in the "domain.go" file to use it by all interfaces.

Architectural Modelling

Architectural modelling is employed to provide a visual representation of the developed architecture. The modelling will be created using Unified Modeling Language (UML), illustrated through a use case diagram in Figure 4.

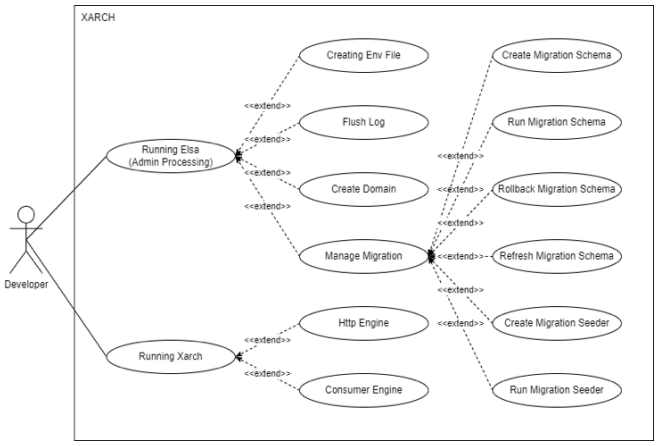

Figure 4. Architectural Usecase Modeling

Communication Flow

An overview of the communication flow in Figure 5 begins with the interfaces component, which is then passed through middleware and handlers according to the previously defined routing. In the subsequent process, incoming requests are processed in the domain service, and if supporting services are needed, the interaction continues to the domain repository.

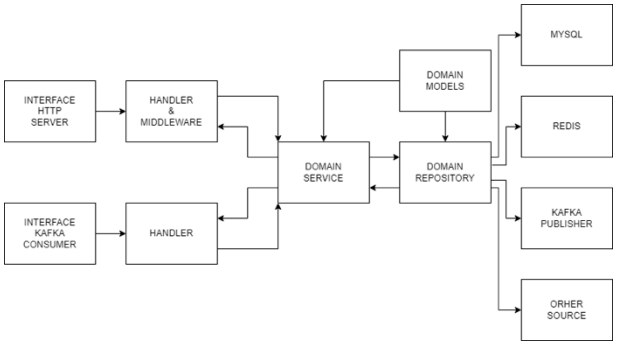

Figure 5. Architectural Communication Flow

Folder Structure

The folder structure in Figure 6 is an essential aspect of the architecture. It should effectively represent the function and purpose of all the files within it to prevent confusion when developers need to manage and utilize the files.

```
.
├──app                  # Application layer contains all program in architecture
|   ├──elsa             # Elsa Application (tools)
|   ├──xarch            # Xarch Application (services)
|   └──app.go           # Program for handling switching program application
├──config               # All configuration in application
├──domain               # Domain Layers is containing a models, repository and services
|   ├──domain_name      # Name of domain
|   |   ├──models       # Contains many file for modeling data
|   |   ├──repository   # All file to comunicated all backing services
|   |   └──services     # Services file to processing a bussiness application
|   └──domain.go        # For publishing a domain and inject all dependencies to domain
├──driver               # Dir to defined and registered all driver (backing services)
├──helpers              # Dir to contains all helpering program
|   ├──bcrypt           # Hash and compare bcrypt
|   ├──env              # Program to get environtment to specifics format data type
|   ├──errors           # Defined a error code to communicated domain and interfaces
|   ├──jwt              # Jwt token helper
|   ├──mail             # Mail sender helper
|   └──md5              # Converted plain text to MD5
├──interfaces           # Interfaces Layer contains all engine method for user access a application
|   └──xarch            # Group interfaces by app
|       ├──consumer     # Consumer Engine
|       |   ├──engine   # Define a engine script for running a consumer interfaces
|       |   ├──handlers # All handler in consumer interfaces
|       |   └──consumer.go # Routing and defined all job in consumer interfaces
|       └──http         # Http Engine
|           ├──engine   # Define a engine script for running a http server interfaces
|           ├──entities # Define to entities models contains a response formater
|           ├──handlers # All handler in http interfaces
|           ├──middleware # All handler middleware in http interfaces
|           └──routers  # Routing http server
├──logger               # Contains a logger program and dir to save all log file
├──migration            # Migration layers can acessed with elsa program
|   ├──schema           # Containing script schema in sql format
|   ├──seeder           # Containing script seeder in sql format
|   └──migrate.go       # For registered schema and seeder
└──main.go              # Main Application
```

Figure 6. Folder Structure

Collaboration Mechanism

The collaboration mechanism in a developed architecture is flexible, allowing developers to choose their methods during the software development process. However, it must adhere to domain-driven design principles, establishing a communication mechanism using ubiquitous language to prevent miscommunication. The architecture divides processes into minor components within different layers, offering advantages for collaboration and specific allocation of human resources. For instance, an infrastructure engineer can manage the application layer, while a software or backend engineer and domain experts can manage the interface and domain services layers. A database administrator or data engineer can manage

the domain data and repository layer. The use of ubiquitous language and the division of human resources aim to achieve efficiency and eliminate barriers between business and technical aspects.

Implementation of The Twelve-Factor App Methodology

1) Codebase: The first principle involves adopting a codebase for revision control. All changes should be easily traceable. The developed architecture has implemented revision control using the GitHub platform.
2) Dependencies: A packaging system or dependency management is necessary for efficiently distributing applications. This system helps automate the setup of supporting libraries. The architecture has implemented dependency management using Go modules. As a result, developers only need to enter the command "go get ./…" in the terminal during installation.
3) Config: This principle requires the application to store all configurations in a file. There are no strict rules regarding file type or format. The architecture follows the principle of storing configurations in the environment, enabling developers to modify settings in the ".env" file. The architecture reads configuration data from the ".env" file and the operating system and machine. This approach streamlines deployment, especially in distributed environments such as containers. Developers can set configurations in environment variables within virtual machines, containers, or operating systems.
4) Backing Services: In this principle, supporting services must be outside the application. When a change occurs from a local service to a publicly distributed third-party service, adjustments should be possible without changes to the codebase. The designed architecture adheres to this principle, allowing changes in credentials or hosts through the provided environment.
5) Build, Release, Run: In this principle, the codebase should compile into a bundle that developers can execute, followed by a release with a specific code and the ability to run it. This principle is directly implemented by the programming language used. In Go, applications can be directly built into binary files or executables. Additionally, the architecture includes a Dockerfile to facilitate compilation into a container image. The release mechanism is performed manually.
6) Processes: In this principle, applications should be stateless, meaning they should not retain data in any form. If an application needs to store data, it should do so only temporarily or for a single process. This approach maintains data consistency, especially during scaling through application replication. The developed architecture adheres to a stateless approach, with all data stored in backing services rather than being retained within the application.
7) Port Binding: This principle requires an application to export services through specific ports. An application should run independently and not rely on runtime injection from the execution environment. Applications should bind to specific ports and listen to all incoming requests on those ports. This principle has developed the architecture, with a machine state in each interface layer. Port binding can be easily adjusted according to configurations in the application environment.
8) Concurrency: In this principle, applications should be able to scale out horizontally and be reliable based on specific services with high workloads. This principle has been applied in the developed architecture. Developers can quickly scale out any desired service. In

implementation, there are two interfaces: HTTP and consumer workers running in the background. The ease of scaling out is based on the layered design and the application of domain-driven architecture, which prevents excessive dependency on any single component. Additionally, scaling out is easily achieved due to the stateless nature of the architecture. In scaling out, developers only need to rerun the program with the appropriate flags for the desired service. By default, when the program is run with the command "go run main.go xarch," the program initializes one HTTP server and one consumer. If the consumer has an excessive workload and needs to scale out, developers can rerun the program with the specific engine interfaces flag, such as "go run main.go xarch -engine=consumer." To scale out the HTTP server, developers must set up the port environment, and the scaling process can begin. Scaling out the HTTP server creates two services with the same process but different port values. For efficiency, developers are advised to use a load balancer.

9) Disposability: This principle relates to application performance. An application should be able to start up quickly and shut down gracefully. Graceful shutdown involves delaying the application termination process. Rejecting incoming requests and completing requests already in the process helps minimize data corruption caused by sudden process termination. During graceful shutdown, the application should ensure that all instances of other services are correctly terminated. For example, when an application has a database connection, that connection must be closed when the application is stopped to avoid zombie connections. This principle has been implemented in the developed architecture, which can start up very quickly, completing the process in 1.1206623 seconds. The architecture has also implemented graceful shutdown during application shutdown, including all machines in the interfaces and when closing all drivers.
10) Dev/prod parity: This principle historically addresses substantial gaps between development and production conditions. It cannot yet be implemented in the designed architecture, which remains a codebase and is not tied to a production environment. However, it can be used when developers use the architecture for their own application development.
11) Logs: In this principle, applications must implement a logging system. The architecture has implemented logging using a JSON format for log output. In the architecture, there are two log writing processes. The first process displays logs in a console for developers to view directly. The second process saves logs to a file, allowing developers to process stored data to review events that have occurred quickly. In the architecture, developers can define log levels such as info, error, warning, fatal, and panic. Logging in the architecture can also be integrated with other applications for log management services, such as ELK Stack.
12) Admin Processes: This is a regulatory process that can be run separately for administrative purposes, such as database migration, tools, or other commands. Admin processes are typically run in a shell or console. The architecture includes an admin process service called ELSA. Developers can run the program whether the main application is running or not. Developers can interact with ELSA by typing "go run main.go elsa ${argument}".

Black Box Testing

Black box testing is a method used to test the functionality of the developed architecture. Testing in the research is divided into two types based on all available use cases. The first test is conducted to evaluate the functionality of the ELSA program, while the next test assesses the architecture program. The following table presents the testing results for the architecture:

Table 1. Black Box Testing ELSA

No	Scenarios	Result
1	Automatic creation of env files	Success
2	Cleaning of entire log files	Success
3	Creation of a new domain	Success
4	Create a migration schema	Success
5	Run the schema migration pipeline	Success
6	Execute schema migration specifics	Success
7	Run the rollback command of the entire schema migration	Success
8	Run schema migration specific rollback commands	Success
9	Run the refresh migration schema command	Success
10	Runs the migration scheme specific refresh command	Success
11	Create a seeder scheme	Success
12	Performs the entire seeder migration	Success
13	Execute schema migration specifics	Success

Table 2. Black Box Testing XARCH

No	Scenarios	Result
1	Running all engine interfaces	Success
2	Running specific engine interfaces	Success
3	Checking the graceful shutdown feature	Success
4	Checking the logging system	Success

Load testing is used to evaluate the performance of the developed architecture. This stage involves creating testing scenarios along with expectations or goals for the testing. Below is a table outlining the scenarios and results of the testing conducted on the architecture:

Table 3. Load Testing

NO	Attribute	Value
1	*Software Tools*	*JMeter* 5.4.3
2	*Objective*	The testing is conducted to evaluate the architecture's performance based on the percentage of successful and failed outcomes and to assess the behavior of dependencies, such as the database.
3	*Scenario*	Send 1000 requests to the health endpoint within 10 seconds, or 100 per second. The health endpoint is chosen because it includes initialization checks on dependencies and connections.
4	*Goals*	Achieve an average response time of less than 1 second with an error rate as low as possible, up to a maximum tolerance of 5% of the total requests.
5	*Result Load Test*	In performance testing, the architecture demonstrated the ability to handle all incoming requests with an error rate of 0%. The results obtained are as follows: 1. Average Response Time: 222 ms 2. Min Response Time: 34 ms 3. Max Response Time: 1,215 ms 4. Standard Deviation: 269.00 5. Error: 0% 6. Throughput: 92.8 requests/second 7. Received Data: 104.58 KB/second 8. Sent Data: 11.23 KB/second Average Bytes: 1,154.3

6	*Driver performance analyst.*	The driver's performance testing revealed 1,000 connection interactions (equal to the number of requests) and a maximum of 25 open connections. These results are due to the use of connection pool technology.

CONCLUSION

Based on the research results, the architectural design supports continuous development by dividing services into small, independent units, enabling rapid integration of new technologies. The implementation of Twelve-Factor App methodology principles provides a descriptive format that enhances automation, supported by revision control and isolated dependencies for streamlined installation. This approach reduces operating system reliance and improves efficiency. The architecture is deployable on modern cloud platforms using methods such as Docker files, with its stateless nature facilitating scalability. Additionally, applying domain-driven design principles by breaking down components according to business domains enhances resource management, performance, and communication through ubiquitous language, minimizing miscommunication. Future research should explore advanced automation, incorporate emerging technologies such as serverless and edge computing, and strengthen security and compliance. Further refinement of domain-driven design principles, performance optimization, user experience improvements, collaboration mechanisms, real-world case studies, benchmarking, and examining the environmental impact of cloud deployments will enhance the architecture's scalability, efficiency, and sustainability in line with evolving technological demands.

REFERENCES

Akinsola, J. E., Ogunbanwo, A. S., Okesola, O. J., Odun-Ayo, I. J., Ayegbusi, F. D., & Adebiyi, A. A. (2020). Comparative analysis of software development life cycle models (SDLC). In Intelligent Algorithms in Software Engineering: Proceedings of the 9th Computer Science On-line Conference 2020 (pp. 310–322). Springer International Publishing.

Ali, M. B., Wood-Harper, T., & Ramlogan, R. (2020). The role of SaaS applications in business IT alignment: A closer look at value creation in service industry. United Kingdom Academy for Information Systems.

Ait Said, M., Belouaddane, L., Mihi, S., & Ezzati, A. (2024). A systematic framework to enhance reusability in microservice architecture. International Journal of Computing and Digital Systems, 16(1), 189–203.

Khan, O. M. A., Siddiqui, N., Oleson, T., & Fussell, M. (2021). Embracing microservices design: A practical guide to revealing anti-patterns and architectural pitfalls to avoid microservices fallacies. Packt Publishing Ltd.

Karunarathne, M. A. W., Wijayanayake, W. M. J. I., & Prasadika, A. P. K. J. (2024). DevOps adoption in software development organizations: A systematic literature review. In 2024 4th International Conference on Advanced Research in Computing (ICARC) (pp. 1–7). IEEE.

Indrasiri, K., & Suhothayan, S. (2021). Design patterns for cloud native applications. O'Reilly Media, Inc.

Kosińska, J., Baliś, B., Konieczny, M., Malawski, M., & Zieliński, S. (2023). Towards the observability of cloud-native applications: The overview of the state-of-the-art. IEEE Access.

Li, B., & Kumar, S. (2022). Managing software-as-a-service: Pricing and operations. Production and Operations Management, 31(6), 2588–2608.

Lopez-Viana, R., Diaz, J., Díaz, V. H., & Martinez, J.-F. (2020). Continuous delivery of customized SaaS edge applications in highly distributed IoT systems. IEEE Internet of Things Journal, 7(10), 10189–10199.

Marii, B., & Zholubak, I. (2022). Features of development and analysis of REST systems. Advances in Cyber-Physical Systems, 7(2).

Oukes, P., Andel, M. V., Folmer, E., Bennett, R., & Lemmen, C. (2021). Domain-driven design applied to land administration system development: Lessons from the Netherlands. Land Use Policy, 104.

Özkan, O., Babur, Ö., & Brand, M. V. D. (2021). Refactoring with domain-driven design in an industrial context: An action research report. Empirical Software Engineering, 28(4).

Pargaonkar, S. (2023). A comprehensive research analysis of software development life cycle (SDLC) Agile & Waterfall model: Advantages, disadvantages, and application suitability in software quality engineering. International Journal of Scientific and Research Publications, 13(8).

Pervin, H. (2021). Software as a service and security. World Journal of Advanced Research and Reviews, 11(3), 327–331.

Raj, P., & David, G. S. S. (2021). Engineering resilient microservices toward system reliability: The technologies and tools. In Cloud Reliability Engineering.

Raghavan, S. R., Jayasimha, K. R., & Nargundkar, R. V. (2020). Impact of software as a service (SaaS) on software acquisition process. Journal of Business & Industrial Marketing, 35(4), 757–770.

Saieva, A., & Kaiser, G. (2022). Update with care: Testing candidate bug fixes and integrating selective updates through binary rewriting. Journal of Systems and Software, 191.

Sangapu, S. S., Panyam, D., & Marston, J. (2022). The definitive guide to modernizing applications on Google Cloud: The what, why, and how of application modernization on Google Cloud. Packt Publishing Ltd.

Shah, S. D. A., Gregory, M. A., Li, S., & Fontes, R. D. R. (2020). SDN enhanced multi-access edge computing (MEC) for E2E mobility and QoS management. IEEE Access, 8, 77459–77469.

Sharma, S. (2023). Modern API development with Spring 6 and Spring Boot 3: Design scalable, viable, and reactive APIs with REST, gRPC, and GraphQL using Java 17 and Spring Boot 3. Packt Publishing Ltd.

Singjai, A., Zdun, U., & Zimmermann, O. (2021). Practitioner views on the interrelation of microservice APIs and domain-driven design: A grey literature study based on grounded theory. In 2021 IEEE 18th International Conference on Software Architecture (ICSA) (pp. 1–10). IEEE.

Stüber, M., & Frey, G. (2021). A cloud-native implementation of the simulation as a service-concept based on FMI. In Modelica Conferences (pp. 393–402).

Telang, T. (2022). Microservices architecture. In Beginning cloud native development with MicroProfile, Jakarta EE, and Kubernetes: Java DevOps for building and deploying microservices-based applications. Apress.

Thatikonda, V. K. (2023). Beyond the buzz: A journey through CI/CD principles and best practices. European Journal of Theoretical and Applied Sciences, 1(5), 334–340.

Throner, S., Abeck, S., Petrovic, P., & Hütter, H. (2023). A DevOps approach to the mitigation of security vulnerabilities in runtime environments. In 2023 IEEE International Conference on Service-Oriented System Engineering (SOSE) (pp. 106–113). IEEE.

Vural, H., & Koyuncu, M. (2021). Does domain-driven design lead to finding the optimal modularity of a microservice? IEEE Access, 9, 32721–32733.

Yellavula, N. (2020). Hands-on RESTful web services with Go: Develop elegant RESTful APIs with Golang for microservices and the cloud. Packt Publishing Ltd.

Hofer, S., & Schwentner, H. (2021). Domain storytelling: A collaborative, visual, and agile way to build domain-driven software. Addison-Wesley Professional.