Computer Science, Engineering and Technology
Vol: 3(2), June 2025
REST Publisher; ISSN: 2583-9179
Website: https://restpublisher.com/journals/cset/
DOI: https://doi.org/10.46632/cset/3/2/12

Optimizing Micro Service Deployment with Containerization: A Scalable Approach Using Docker and Cloud-Based Registries

*** Prashanth Chevva**
Ferris State University, Michigan.
**Corresponding author Email: prashanthch6034@gmail.com*

Abstract: *The evolution of cloud computing has necessitated efficient deployment strategies for microservices, with container- ization emerging as a key solution. This paper explores the role of Docker in streamlining microservice deployment, focusing on the use of multi-stage builds, lightweight containerization, and cloud-based registries for scalable and efficient application man- agement. By leveraging Docker, microservices achieve enhanced portability, reduced resource overhead, and simplified depen- dency management. Furthermore, the paper discusses the lim- itations of standalone containerized applications and highlights the need for orchestration platforms like Kubernetes to ensure resilience, dynamic scaling, and automated service discovery. Through a practical implementation, this study demonstrates the benefits of containerized microservices and provides insights into optimizing cloud-native deployments.*

1. INTRODUCTION

The motivation behind selecting this project stems from the increasing prominence of microservices architecture in modern software development. As with many architectural paradigms that have emerged over time, microservices have gained significant traction due to their potential to enhance scalability, flexibility, and maintainability in complex systems. However, determining the suitability of this architecture for a given problem requires a comprehensive understanding of both the underlying technologies and the conceptual principles that govern its implementation. A structured approach has been taken to explore various dimensions of microservices architecture. The initial phase involved familiarization with foundational concepts, including the definition and core principles of microservices. Subsequently, practical implementation was undertaken, beginning with virtualization and containerization, followed by an in- depth study of container orchestration techniques. Further exploration extended to key areas such as service state management and inter-service communication, incorporating both synchronous and asynchronous communication mechanisms. This iterative approach provided valuable insights into how microservices operate in real-world scenarios and the challenges associated with their deployment and maintenance

2. SCOPE AND PURPOSE

An extensive review of literature, online resources, and industry practices has identified three fundamental aspects that are integral to microservices architecture: independent deployability, communication strategies, and business domain logic. Each of these aspects encapsulates multiple core principles essential for designing and managing microservices effectively. Independent deployability is a critical attribute of microservices, as it enables each service to be developed, updated, and deployed autonomously without impacting other components. This characteristic is deeply rooted in principles such as encapsulation, information hiding, and effective state management. Understanding these principles facilitates the design of services that can evolve independently while maintaining system integrity. Inter-service communication plays a pivotal role in ensuring seamless interactions between microservices. The effective- ness of a microservices-based system is heavily influenced by both the communication style—whether synchronous or asynchronous—and the technological stack employed, including HTTP-based APIs, Remote Procedure Calls (RPC), and messaging queues. Properly structured communication pat- terns contribute to system

responsiveness, reliability, and fault tolerance. The conceptualization of business domain logic within mi- croservices architecture requires careful consideration. Rather than being driven by technological constraints, the definition of service boundaries is guided by domain-driven design principles. A well-structured microservices ecosystem ensures that each service is responsible for a distinct business func- tion, avoiding overlaps and redundancies. This approach not only enhances modularity but also simplifies maintenance and scalability. A recurring theme across all these aspects is scalability, which extends beyond mere technological advancements. Scal- ability influences how microservices are orchestrated, man- aged across distributed teams, and integrated into broader enterprise systems. By effectively leveraging scalable archi- tectures, organizations can ensure that development processes align with business growth, leading to sustainable and efficient software systems. The subsequent sections will delve deeper into these considerations and their practical implications.

3. EXPERIMENTAL IMPLEMENTATION

To gain a practical understanding of microservices architecture and its associated technologies, various fundamental components were developed as experimental implementations. These components were chosen based on their common occurrence in modern software applications, with a particular emphasis on user management and authentication functionalities.

Containerization: The implementation process commenced with an in-depth exploration of Docker, a widely used platform for containerization. Containers play a vital role in microservices architecture by isolating application dependencies from the underlying infrastructure, thereby simplifying deployment and ensuring consistency across environments. By encapsulating essential elements such as the operating system, runtime, libraries, and storage, containerized applications maintain a high degree of portability and scalability. Unlike traditional virtualization, where each instance re- quires a separate guest operating system, containerization enables multiple isolated environments to run on a shared kernel, significantly reducing resource overhead. This approach enhances efficiency and accelerates deployment processes. Advanced tools such as Docker Swarm and Kubernetes further extend the capabilities of containerization by enabling the orchestration and management of distributed containerized applications across multiple physical or virtual servers. As a result, containerization serves as the foundation for the independent deploy ability of microservices. To demonstrate these principles, a minimal web service was implemented to handle HTTP POST requests for user profile creation. The next step involved transforming this service into a fully containerized application. A multi-stage Docker build process was employed to containerize the microservice efficiently. The initial stage utlized a base image that included all necessary Go tools and dependencies for compilation. Once the build process was completed, the compiled binary was transferred to a lightweight Alpine Linux-based image, ensuring that the final container remained as minimal as possible while containing all required functionalities. This technique optimizes resource usage by separating the build environment from the execution environment. Since the service is designed to run on port 4000, the Docker configuration explicitly exposes this port. For deployment readiness, the generated container image was uploaded to a container registry, allowing seamless acess across various deployment platforms, including Kuber- netes. Container registries serve as centralized repositories for storing, versioning, and distributing container images. These repositories can be hosted on local servers or cloud-based platforms, offering additional features such as access control and security policies. Given that many cloud-based registries require a paid subscription, the Google Cloud Artifact Registry was selected due to its free-tier availability, making it a suitable choice for experimental purposes. The process of integrating Docker with Google Cloud involved overcoming multiple authentication and authorization challenges due to the stringent security policies in place. After consulting documentation and installing the necessary authentication tools, successful interaction with the Google Cloud Artifact Registry was achieved, allowing container images to be pushed and pulled as needed. Since the web service required a database for user data per- distance, an additional containerized instance of PostgreSQL was configured. Using Docker Compose, both the application container and the database container were linked, enabling seamless interaction between the two services. Ensuring data persistence across container restarts posed a challenge, as containerized environments are typically ephemeral. To ad- dress this issue, a volume mapping strategy was implemented, allowing data to be stored in a designated directory on the host machine. By specifying volume configurations in the Docker Compose file, the PostgreSQL container was able to retain its state across multiple executions, ensuring data integrity. This phase of implementation provided valuable insights into the fundamental principles of containerized microservices and highlighted practical challenges associated with service or- chestration and data persistence. The subsequent sections will further explore these challenges and additional refinements applied to enhance the overall architecture.

Container Limitations: The implemented solution has demonstrated stable performance in handling requests from both direct interactions and tools such as curl and Swagger. While this approach is effective for non-critical applications, test environments, or controlled deployments with a limited number of users, it presents several constraints when applied to large-scale, high-availability services that must remain operational around the clock. The following challenges highlight the inherent limitations of containerized applications in such scenarios: Scalability: A major constraint of standalone containers is the lack of dynamic scaling. Since the container is confined to a single server, it cannot be replicated, nor can its resources be adjusted in response to varying workloads. The only available option is manual intervention, such as migrating the container to a more capable machine or upgrading the existing hardware. However, these solutions do not provide real-time adaptability and may lead to temporary service disruptions. An ideal system would allow resource allocation to be adjusted automatically based on demand, optimizing both performance and cost. Updates: Deploying an updated version of the service requires stopping the current container, applying the changes, and restarting it. This approach not only introduces downtime but also lacks robust failover mechanisms. If an issue arises during an update, the affected version may remain accessible to users until manual intervention is performed. A more efficient strategy would enable seamless updates with zero downtime, ensuring that faulty deployments can be detected and rolled back automatically without affecting service avail- ability. Error Handling: While container platforms provide basic restart policies in case of software failures, managing faults at a more granular level remains a challenge. A robust system must be capable of identifying various failure conditions, such as unresponsiveness, timeouts, or unexpected errors, rather than relying solely on application crashes as failure indicators. Additionally, restarting a container incurs a delay, leaving the service temporarily unavailable, which is undesirable for mission-critical applications. Networking: When multiple microservices need to interact and exchange data, default container networking capabilities are often insufficient. Docker employs a virtualized internal network, which provides only limited control over routing and connectivity. Configuring network rules manually can be cumbersome, especially when dealing with dynamic services that may change IP addresses or migrate across hosts. Further- more, a clear distinction is necessary between public endpoints accessible to users and internal communication channels that must remain private for security and efficiency. Deployment Complexity: Managing and maintaining in- dividual containers manually across multiple servers is labor- intensive. Ensuring that all instances remain operational, prop- erly updated, and error-free requires continuous oversight. A centralized system for orchestrating deployments across multiple nodes would significantly enhance efficiency by au- tomating the distribution, monitoring, and recovery of services. To address these challenges, an orchestration solution is required. Orchestration software automates the deployment, scaling, and management of containerized applications, mak- ing it an essential component in distributed systems. While orchestration extends beyond microservices, it is particularly valuable in ensuring reliability, efficiency, and maintainability in large-scale architectures. Among the available orchestration tools, Kubernetes has emerged as a leading platform due to its comprehensive set of features for managing containerized workloads. Given the constraints observed in the current implementation, Kuber- netes was selected as the next step in refining the architecture. A local cluster was set up using Minikube, allowing for an initial configuration of the necessary components to deploy and manage services effectively.

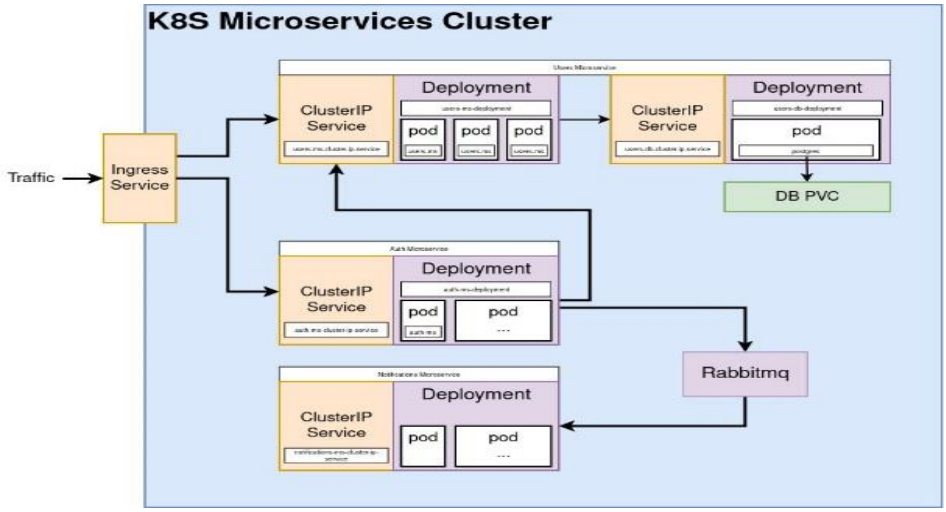

FIGURE 1. Architecture of Kubernetes cluster for the practical implementation

Users Microservice: The Users Microservice is responsible for handling user- related operations through multiple HTTP handlers, following the REST (Representational State Transfer) architectural style. These handlers are exposed on port 4000. POST: /users-ms/users – Accepts a JSON payload con- taining user details such as name, surname, email, and other relevant attributes for database storage. GET: /users-ms/users/id – Retrieves the user record as- sociated with the provided identifier. GET: /users-ms/users – Returns a list of all stored users. Query parameters such as email, name, or surname can be included to filter results. PUT: /users/id – Updates the details of an existing user identified by the given ID using the provided data. DELETE: /users/id – Removes the user entry corre- sponding to the specified ID. A critical design aspect of this microservice is the man- agement of its internal state, which must remain persistent to ensure data availability across multiple requests and service restarts. This is typically achieved by integrating a database. For microservices to be independently deployable while maintaining horizontal scalability, each instance must function autonomously without sharing an embedded state. Running multiple instances of a microservice enhances reliability, as failures in individual cases can be handled through automated recovery mechanisms while traffic is dynamically redirected to operational instances. Embedding a dedicated database within each microservice instance introduces significant challenges in maintaining data consistency. If each instance were to operate with an independent database, synchronization issues could arise, leading to data fragmentation and integrity concerns. To mitigate this, a single shared database is deployed as a separate entity, accessible by all instances of the Users Microservice. This ensures that all the cases operate on a unified dataset without conflicts. Further details on this approach are discussed in the following section. The Kubernetes (K8s) setup for this microservice involves defining a YAML configuration file that specifies three key components: a deployment object, a node port service, and a cluster IP service. The deployment object retrieves the container image from Google Cloud and orchestrates the creation of multiple pods based on the specified configuration. The node port service links port 4000 of a pod to port 30400 of the host, enabling direct access to the service for testing and debugging. Meanwhile, the cluster IP service assigns an internal network port (4000) to allow other components within the Kubernetes cluster to interact with the microservice. As pods are created, terminated, and restarted—whether for fault recovery, updates, or scaling—their associated IP addresses are dynamically reassigned. Relying on static IP configurations for communication between microservices is therefore impractical in an orchestrated environment. To ad- dress this challenge, Kubernetes employs a selector-based service discovery mechanism. Labels assigned to pods allow Kubernetes services to dynamically bind to the appropriate instances, while an internal DNS system resolves label-based bindings to real-time IP addresses. This abstraction facilitates seamless connectivity between microservices deployed across multiple nodes, ensuring robust and efficient service orches- tration.

Users DB Microservice: This microservice is designed to provide persistent storage for user-related data, supporting the Users Microservice. The deployment object utilizes a PostgreSQL image sourced from the official registry (Docker Hub) and is configured using envi- ronment variables to define database credentials, including the database name, user, and password. To maintain consistency and prevent duplication, the number of replicas is set to 1, ensuring a single active database instance. Similar to other components, a node port service is imple- mented to facilitate direct access to the database pod for de- bugging and testing, whereas a cluster IP service is responsible for internal communication between different services in the cluster. The Users Microservice is granted access through pre- defined environment variables, eliminating the need for manual IP configuration due to the service discovery mechanism. The database deployment is identified with the label ”users-db,” allowing the Users Microservice to establish a connection through the endpoint ”users-db-cluster-ip-service.” Data persistence is maintained through the integration of a PersistentVolume object, linked via a PersistentVolumeClaim. The volume access mode is defined as ReadWriteOncePod, ensuring that it remains accessible exclusively to a single node and the designated database instance. This database setup does not inherently support horizontal scaling, as additional resources must be allocated manually to improve performance. However, scalability is achieved at the architectural level, where different services manage their own databases, distributing the load more effectively. In cases where a database demands a more scalable approach, multiple strategies can be considered. One option involves leveraging a managed cloud database service, allowing automatic scaling based on demand while reducing maintenance overhead. Though this approach may introduce additional costs, it simplifies database scaling, par- ticularly for teams with limited resources. Alternatively, a distributed database solution can be integrated within the Kubernetes environment or managed through external systems. Notably, Zalando has developed an open-source PostgreSQL Operator for Kubernetes, offering advanced features such as automatic scaling, rolling updates, load balancing, and point- in-time recovery. While this implementation was not incorpo- rated in the current project due to its scope, it remains a viable option for future enhancements.

Auth Microservice: To facilitate inter-service communication, the Auth Mi- croservice has been developed. This microservice includes a single handler (POST: /auth-ms/auth/login) that accepts a username and password, verifying the credentials against the stored user data. If authentication is successful, a token is generated and returned as a response. The implementation fol- lows a token-based authentication model, where users initially authenticate with credentials and subsequently use a token for authorization. The authentication process involves sending a GET request to the Users Microservice, passing the user's email. Upon receiving a valid response, the Auth Microservice compares the provided password with the stored hashed password. If the credentials match, a randomly generated token is issued. The interaction between these services relies on Kubernetes' internal service discovery mechanism, avoiding direct IP ad- dress configuration and instead utilizing label-based service resolution. Security considerations play a crucial role in password management. Storing passwords in plaintext poses a significant risk; therefore, bcrypt is employed to hash passwords upon receipt. The authentication process involves comparing the hashed password stored in the database with the hashed version of the user-provided input, ensuring a secure validation mechanism. Additionally, adherence to microservice bound- aries is maintained by preventing direct database access from the Auth Microservice. Instead, all user-related queries are routed through the Users Microservice, reinforcing modularity and separation of concerns. The authentication request operates synchronously, meaning the Auth Microservice must wait for a response before pro- ceeding. While synchronous communication introduces poten- tial latency, it remains essential for authentication workflows, where immediate validation is required before granting access. To explore asynchronous processing, an event-driven mech- anism is incorporated for user notifications. Whenever a successful login occurs, an event is triggered to notify the user without blocking the authentication process. This is achieved using RabbitMQ, where a message is published to an exchange in a mailbox-style communication model. A separate Notification Microservice listens to the message queue and processes notifications independently. This approach ensures decoupling of authentication from notification handling, im- proving system responsiveness while allowing scalable event driven operations.

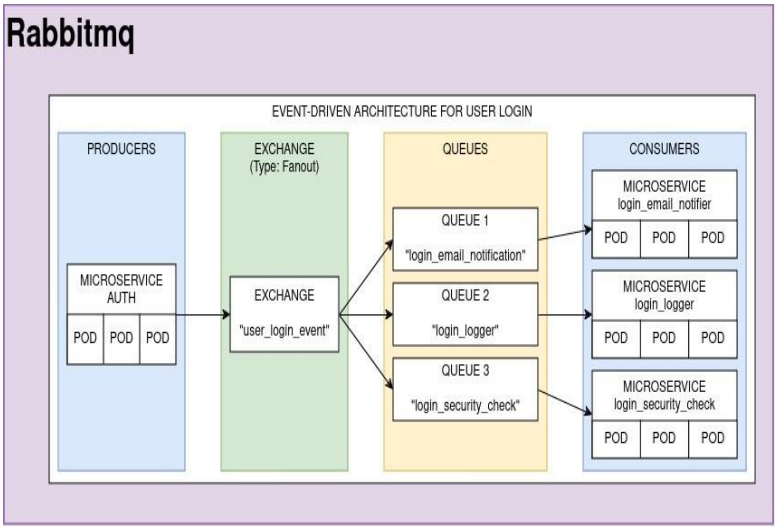

FIGURE 2. RabbitMQ scheme with producer, exchanges, queues, and consumers.

Asynchronous Communication: Asynchronous messaging allows a microservice to initiate a request and continue executing other tasks without being blocked while waiting for a response. In this implementation, event-driven asynchronous communication is leveraged to dis- patch notifications upon user login. Since email delivery can introduce latency, this approach ensures that the microservice remains responsive by processing other operations concur- rently rather than waiting for completion. One of the fundamental principles in microservice architecture is minimizing interdependencies between components. Asynchronous communication reduces temporal coupling by eliminating the requirement for immediate responses between the authentication and notification microservices. Temporal coupling occurs when a service depends on another to complete an action in real time. By adopting an asynchronous event-driven strategy, interactions between these services be- come more flexible, improving scalability and resilience. RabbitMQ serves as the message broker, facilitating event- driven communication within the Kubernetes cluster. It is deployed using a cluster operator, which enables automated management of message queues and event routing. Within the authentication microservice, each successful login generates a message containing login

details, which is then published to an exchange. The exchange functions as an intermediary, forwarding messages to various queues based on predefined rules such as routing keys or topics. In this case, the exchange is labeled “user login event” and follows a “fanout” routing strategy, ensuring that every bound queue receives a copy of the message. To maintain persistence, the exchange is marked as “durable,” allowing it to be recreated automatically in case of service restarts. The queues are defined on the consumer side within the Notification Microservice. Since different notification methods may be required depending on the event type and the preferred communication channel, a structured queue-naming convention is adopted. The exchange names indicate the event type (e.g.,”user login event”,” new user event”), while queues are designated based on the notification method (e.g.,”lo- gin email notification”,”login sms notification”). This structured approach standardizes event handling and facilitates the extension of the notification system.

Notification Microservice: The primary function of this microservice is to process login events and send email notifications informing users of new access attempts. Upon initialization, the service es- tablishes a connection with RabbitMQ and subscribes to the ”user login event” exchange. A dedicated queue named ”user login event” is created and bound to the exchange. Since the queue is defined as non-exclusive, multiple instances of the microservice can be deployed concurrently, enabling parallel message processing. Once activated, the microservice continuously listens for messages in the queue and processes them sequentially. The email-sending operation is simulated using a delay mechanism, ensuring that messages are acknowledged only after success- ful processing. Manual acknowledgments are employed to confirm message consumption, preventing premature message removal before email dispatch is finalized. To enhance scalability, multiple instances of the Notification Microservice are deployed, each functioning as an independent consumer. Since all instances share access to the queue, horizontal scaling can be achieved effectively by distributing the workload among multiple consumers. This design allows the system to dynamically adapt to increased notification demands while maintaining efficient processing of messages.

Update Strategies: Kubernetes simplifies image updates by modifying the specified image version within the deployment configuration. When no explicit strategy is defined, a rolling update mechanism is applied by default, utilizing the preset values for maxSurge and maxUnavailable, both set at 25% of the desired number of replicas. This approach ensures a controlled transition by incrementally replacing old Pods with new ones. The maxUnavailable parameter dictates the highest number of Pods that can be offline during the update process. For instance, in a deployment with four replicas and a maxUnavailable value of 25%, only one Pod can be out of service at any given moment. Conversely, the maxSurge setting allows additional Pods to be created beyond the target number. If a deployment consists of four replicas and maxSurge is set at 25%, up to one extra Pod may be provisioned temporarily, reaching a total of five active instances. These values can be specified in absolute numbers instead of percentages, influencing the overall speed of the update. In this project, the default rolling update strategy is applied to the Auth, Users, and Notifications microservices. However, the Users DB microservice follows a different approach by implementing the recreate strategy. Since running multiple replicas of the same database instance is not feasible (maxReplicas = 1), the existing instance is terminated before a new one is deployed when an update occurs. Alternative update strategies, such as Blue-Green and Ca- nary deployments, exist but are not utilized in this setup. Using the latest tag for container images within a Ku- bernetes environment is generally discouraged due to chal- lenges related to stability and reproducibility. Without ex- plicit versioning, tracking deployments or rolling back to previous states becomes difficult, leading to inconsistencies across different stages of development, testing, and production. This lack of predictability complicates debugging efforts and hinders the ability to maintain controlled and well-documented deployments.

Health Checks: A key aspect of container orchestration involves detecting failures and responding appropriately to maintain service reli- ability. Kubernetes provides configurable health check probes to assess the status of running applications and take corrective measures when necessary. A liveness probe is used to verify whether a containerized application remains in a functional state. The control plane continuously monitors the defined probe and restarts a con- tainer if it encounters failures. Typically, an HTTP request (e.g., GET or POST) is sent to a designated endpoint inside the container, and if no response is received, the container is restarted. Additional parameters can be configured, such as an initial delay before the first probe execution and the frequency of subsequent checks. For this implementation, a liveness probe is defined for the Users microservice. The probe performs a GET request to the ”/users-ms/healthcheck” endpoint at five-second intervals. An initial delay of five seconds is also included to prevent premature failure detection while the service establishes its connection to the database. Without this delay, Kubernetes might incorrectly identify the pod as unhealthy, potentially leading to unnecessary restarts while the microservice is still initializing.

Documentation and Examples: Maintaining accurate documentation is essential when working with distributed architectures, as microservices introduce complexity through numerous endpoints and varied interactions. To systematically track API endpoints, required parameters, and expected responses, an OpenAPI specification file is created. OpenAPI provides a standardized format using YAML syntax to describe API structures, enabling consistent documentation across services. These specification files can be integrated with tools such as Swagger, which facilitates API testing and visualization through an interactive dashboard. To validate this approach, a Swagger Docker container is deployed, allowing real-time interaction with all microservice endpoints. Ideally, every microservice should maintain an independent OpenAPI specification. This practice ensures that developers working on different services can reference the necessary re- quest formats and response structures, streamlining integration and development efforts.

4. EXPERIMENTS

Throughout the development process, various experiments were conducted to validate the expected behavior of Kubernetes components in addition to ensuring that microservices functioned correctly. To examine the impact of deploying multiple instances of the notification microservice as RabbitMQ queue consumers, a sequence of login requests was generated through the authen- tication microservice. By logging the messages processed by each instance, it was observed that the workload was evenly distributed among the consumers, with each message being handled by only one instance. This confirmed that scaling message consumption horizontally is an effective method for increasing throughput within a queue-based system. To validate the implementation of the liveness probe applied to the users microservice, request logs from different pods were monitored. The probe was found to be executing as ex- pected, sending requests at five-second intervals. A controlled failure was introduced by forcing the microservice to crash randomly. Upon failure detection, the liveness probe identified the unresponsive service and triggered a restart, demonstrating the probe's ability to maintain availability through automated recovery. To evaluate the rolling update mechanism, a microservice was updated with a newer container image within Kubernetes. Using the Minikube dashboard, the update process was moni- tored as Pods were progressively terminated and replaced. The transition followed the expected rolling update configuration, ensuring that service availability was maintained throughout the deployment process.To investigate the functionality of Kubernetes Horizontal Pod Autoscaler (HPA), resource usage and active replicas were observed through both the command line (kubectl get hpa users-ms-deployment --watch) and the Minikube dashboard. Based on Kubernetes documentation guidelines, an additional pod was deployed using an Alpine image configured to generate continuous HTTP requests to a designated microservice endpoint. A specialized load-testing endpoint was designed to increase computational demand by calculating Fibonacci sequences. While unrelated to typical user management operations, this approach was chosen after observing that standard endpoint requests did not lead to autoscaling activation. Despite sustained resource load, no additional replicas were created by the autoscaler. Monitoring tools reported CPU usage levels remaining below 0.1%, suggesting that the virtu- alized Minikube environment may not be accurately reflect- ing resource consumption. This behavior implies potential limitations in resource measurement within local Kubernetes clusters and raises the need for further testing in a distributed production environment.

5. SELF-ASSESSMENT

Project critique: One of the initial challenges encountered in this project was identifying a compelling implementation scenario, as most potential use cases appeared too generic or unengaging. Rather than attempting to develop an overly complex or novel solu- tion, the focus was placed on fundamental components com- monly found in cloud-based applications—user management, authentication, and notifications. This decision enabled an in- depth exploration of microservice orchestration without being sidetracked by intricate application logic. While the resulting implementation consists of relatively straightforward services, these provided a valuable foundation for experimenting with various configurations and gaining a deeper understanding of microservice interactions.

Learnings: Through hands-on experimentation, it became evident that managing microservices presents unique challenges. Several key aspects were explored, including update deployment strategies, scalability mechanisms, failure recovery, and ser- vice discovery. While Kubernetes abstracts many underlying complexities, its adoption does not eliminate the need for thoughtful architectural decisions. Compared to a manual orchestration approach, Kubernetes significantly reduces the operational burden, yet an understanding of distributed system design remains crucial for effectively leveraging its capabilities. Although the learning curve for Kubernetes and Docker can be steep, these

technologies simplify many of the difficulties associated with distributed computing. The primary challenge lies in acquiring the necessary expertise to use them efficiently. Once you are familiar with these tools, configuration and deployment become much more intuitive. Beyond the technical aspects, architectural considerations play a crucial role in microservice design. Decisions regarding the level of service decoupling, appropriate communication models (synchronous vs. asynchronous), and concurrency strategies require a comprehensive understanding of distributed systems. Without this foundational knowledge, working with Kubernetes would have been significantly more error-prone and challenging. An essential takeaway from this implementation was the importance of careful planning in microservice architectures. Services must be designed to be modular, independent, and aligned with well-defined business domains. The choice of technology is secondary to ensuring that services interact seamlessly while remaining loosely coupled. For instance, during the implementation phase, considerable thought was given to determining how user authentication should be handled—whether credential verification should be managed by the authentication service alone or involve inter- actions with the user service. Similar deliberations arose when designing the notification microservice, particularly regarding its ability to support future extensions for additional event types and communication channels. The most valuable insights gained from this process extend beyond the specifics of Kubernetes or container orchestration. While technical knowledge can be acquired relatively quickly through documentation and online resources, developing an architectural mindset requires experience and deliberate analy- sis. Understanding how to structure microservices to maximize efficiency and maintainability is a skill that will be beneficial for future projects. Another realization from this project was that microservices, despite their advantages, can introduce additional complexity, particularly in smaller development teams. As the number of services grows, maintaining an overview of all interactions becomes increasingly difficult, making system management more challenging for a single developer. In contrast, microservices architectures are better suited for organizations where different teams manage separate services while collaborating on integration points. Under such a struc- ture, developers can concentrate on specific domains while ensuring interoperability through well-documented interfaces. This approach reduces the cognitive load on individual con- tributors and enhances the maintainability of the system. Ultimately, microservices and Kubernetes are neither inher- ently beneficial nor detrimental technologies. Their suitability depends on the context in which they are applied. Completing this project has provided a clearer perspective on the advan- tages and limitations of these tools, enabling more informed architectural decisions in the future.

6. CONCLUSION

This research project provided a comprehensive exploration of microservice orchestration using Kubernetes, focusing on fundamental aspects such as update strategies, health checks, scalability mechanisms, and architectural planning. The prac- tical implementation of core microservices, including authen- tication, user management, and notifications, served as a foundation for gaining hands-on experience in configuring and managing distributed systems. Through various experiments, key observations were made regarding the behavior of Kubernetes components under dif- ferent conditions. The rolling update strategy demonstrated its effectiveness in ensuring minimal downtime, while the use of health probes enabled automatic failure detection and recovery. Additionally, the exploration of horizontal scaling revealed po- tential challenges when operating in virtualized environments, emphasizing the importance of real-world infrastructure for performance validation. A significant takeaway from this study was the realization that microservice architectures demand careful planning be- yond technological implementation. The process of design- ing modular, decoupled services highlighted the necessity of defining clear service boundaries and communication models to ensure maintainability and scalability. While Kubernetes provides powerful automation capabilities, achieving optimal results requires a solid understanding of distributed system design principles. Moreover, the study underscored the trade-offs associated with microservices, particularly in terms of management complexity. While beneficial in large-scale applications with multiple development teams, smaller projects may experience increased overhead due to the need for extensive service coordination and documentation. Overall, this research reinforced the importance of balanc- ing technological adoption with architectural best practices. The insights gained from this study not only deepen the understanding of Kubernetes-based microservice orchestration but also offer valuable guidance for future projects involving distributed computing and cloud-native development.

REFERENCES

[1]. Vassallo, S., & Camilleri, K. (2017). Microservices: A software architectural style. Proceedings of the 2017 International Conference on Cloud Computing and Big Data Analysis (ICCCBDA), 82-87. Available at: https://scholar.google.com/scholarlookup? Title Microservices %

3A+A+software+architectural+style&author=Vassallo%2C+S.&author= Camilleri%2C+K.

[2]. Burns, B., Oppenheimer, D., & Wilkes, J. (2016). Kubernetes: A system for running and managing containerized applications. Communications of the ACM, 59(3), 50-59. Available at: https: //scholar.google.com/scholar lookup? title=Kubernetes%3A+A+system+ for +running +and+ managing+ containerized+ applications & author= Burns%2C+B.&author=Oppenheimer%2C+D.&author=Wilkes%2C+J.

[3]. Richardson, C. (2018). Microservices patterns: With examples in Java. Manning Publications. Available at: https://scholar.google.com/scholarlookup? title=Microservices+patterns%3A+With+examples+in+Java& author=Richardson%2C+C.

[4]. Duan, Z., & Wang, Y. (2019). Scaling Kubernetes for large scale systems: Challenges and opportunities. IEEE Access, 7, 22482- 22491. Available at: https://scholar.google.com/scholar lookup? title= Scaling+Kubernetes+for+large+scale+systems%3A+Challenges+and+ opportunities&author=Duan%2C+Z.&author=Wang%2C+Y.

[5]. Bass, L., Weber, I., & Zhu, L. (2015). DevOps: A software architect's perspective. Addison-Wesley. Available at: https://scholar.google.com/scholarlookup?title=DevOps%3A+A+ software+architect%27s+perspective&author=Bass%2C+L.&author= Weber%2C+I.&author=Zhu%2C+L.

