Service Oriented Computing and Applications (2023) 17:149–159
https://doi.org/10.1007/s11761-023-00364-w

ORIGINAL RESEARCH PAPER

# Qualitative and quantitative comparison of Spring Cloud and Kubernetes in migrating from a monolithic to a microservice architecture

Yu-Te Wang[1] · Shang-Pin Ma[1] · Yue-Jun Lai[1] · Yan-Cih Liang[1]

Received: 11 February 2023 / Revised: 22 April 2023 / Accepted: 3 May 2023 / Published online: 25 May 2023


**Abstract**
The microservice architecture has several advantages over conventional monolithic architectures, such as the ability to develop and deploy services independently and flexibility in dealing with bottlenecks. Most existing works in this field have focused on methodologies by which to divide monolithic systems into microservices, paying little or no attention to the target platform or migration tools, despite the profound effects that they can have on system migration. For a monolithic and service-oriented application, the go-to frameworks and platforms for migration to a microservice architecture are Spring Cloud and Kubernetes. This paper compares the pros and cons of the two approaches and presents experiments involving a stress-testing tool to elucidate performance and scalability under various conditions.

**Keywords** Microservice · Architecture migration · Spring Cloud · Kubernetes · Scalability

## 1 Introduction

Researchers are increasingly considering the methods and tools used to facilitate the migration from a monolithic system to a microservice system (MSA). The simplicity, compactness, specificity, and individual deployment of microservices provide considerable advantages over conventional monolithic architectures, such as the ability to develop multiple services in parallel using DevOps tools [1, 2]. The modular structure also makes it possible to deal with bottlenecks [2], while providing resilience, flexibility, and fault tolerance beyond what can be achieved using a monolithic structure [3]. MSA has also been shown to reduce CPU overhead [4].

✉ Shang-Pin Ma
albert@ntou.edu.tw

Yu-Te Wang
10957032@mail.ntou.edu.tw

Yue-Jun Lai
00757125@mail.ntou.edu.tw

Yan-Cih Liang
11157037@mail.ntou.edu.tw

[1] Department of Computer Science and Engineering, National Taiwan Ocean University, Keelung, Taiwan

Unfortunately, the distributed nature of MSA also makes it inherently complex and difficult to manage [5]. Most MSA systems require a service registry to allow communication among microservices and an API gateway to control traffic and provide an interface for external users as well as load balancers to distribute traffic between multiple instances of the same service. As a result, system administrators must deal with configuration management, difficulties in identifying the nature of errors, and cascading failures. The fact that finding the root cause is often counterintuitive has led to the development of distributed tracing and centralized metrics to narrow the scope of possible errors.

Most previous papers on microservice migration have focused on the methodologies used to divide a monolithic system into a microservice system [6–10] in terms of cost and complexity [11]. The technologies used in the construction of a microservice system can have a direct effect on all subsequent maintenance, management, and expansion. Researchers have developed numerous methods to facilitate the identification of microservices during architecture migration, based on functional semantic similarity [12], business processes [13], system security and scalability [14], domain-driven design (DDD) [15], and database access [16]. However, researchers have yet to develop methods by which to analyze the target framework and platform [17].

The operating environment of an MSA system can be divided into framework and platform. Frameworks, such as Spring Cloud [18], provide a number of prebuilt tools to facilitate migration, and the container management platform. Kubernetes provides a favorable MSA operating environment [19, 20]. In this study, we compared Spring Cloud and Kubernetes in terms of cost, scalability, and monitoring tools when migrating from a Spring Boot monolith to an MSA.

The remainder of the paper is organized as follows. Section 2 briefly introduces the system used as a benchmarking example for microservice migration. In Sect. 3, we describe the target frameworks/platforms for MSA migration. In Sect. 4, we compare the two approaches from various perspectives. Experiment results pertaining to the performance and scalability of the two systems are presented in Sect. 5. Conclusions are drawn in Sect. 6.

## 2 Test system: personal data authorization system

In this study, we used Personal Data Authorization System (PDAS) [21], our previously developed monolithic application, as a benchmark for comparing microservice migration options.

### 2.1 PDAS: objective

The main objective of the proposed PDAS was to enable the purchase of personal data (such as energy usage records) from ordinary users by third-party application developers and ensure that the data holders follow through on the provision of data.

Consider an example in which the data requester (DR) is a third-party application developer seeking to provide services based on analysis of electricity power usage. After purchasing a dataset from the data sales portal, the DR is then redirected to the PDAS to enroll as a participant in a data access authorization contract. The data holder (DH) (e.g., electricity power company) signs the contract after reviewing it, whereupon the PDAS generates contracts to be signed by all participants, thereby ensuring access to the dataset.

The PDAS used government-issued certificates (Citizen Digital Certificate, MOEACA ID card) to ensure the legality of the data authorization contract.

### 2.2 PDAS: system architecture

The PDAS system was developed in Java based on the Spring Boot[1] framework. As shown in Fig. 1, the system was divided into six modules: *Email*, *User*, *Contract*, *Blockchain*, *Signature Verification*, and *External Request*. The email module handles the task of sending all notifications and reports by email. The user module is responsible for operations relating to the user (e.g., modifying user information and changing passwords). The contract module is in charge of operations relating to the contract (e.g., signing a contract and viewing contracts). The blockchain module is capable of publishing and querying the hash information of contracts on the blockchain. Note that by publishing signatures of signed contracts to a blockchain, the proposed PDAS protects the contracts from tampering. The signature verification module can verify the legality of signatures and signed contracts. The external request module is responsible for collaborating with the external DSP (Data Sales Portal) system. We also defined three restful API controllers to allow access from front-end web pages and external systems.

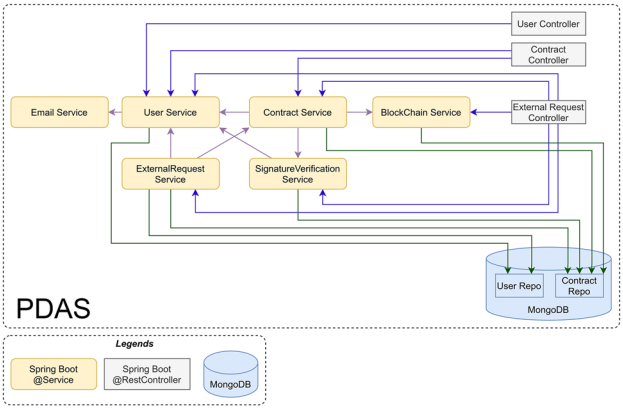


**Fig. 1** Monolithic PDAS architecture

## 3 Approaches to architecture migration

Implementing a microservice architecture involves dividing the monolithic system into small, independently deployable microservices.

The fact that the PDAS in this paper was designed as a modular system made it far easier to divide it into microservices. Our analysis of the current monolithic system revealed that the six original modules could be easily converted into microservices.

In the monolithic PDAS, we used Spring Security[2] for user authentication. Note that it was impossible to adopt this approach to the microservice PDAS, as this would have necessitated synchronizing passwords for all services. Thus, we introduced a seventh microservice (Credential) to manage token verification.

[1] https://spring.io/projects/spring-boot.

[2] https://docs.spring.io/spring-security/reference/.

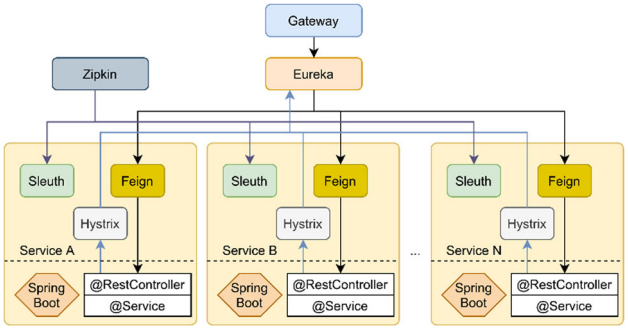


**Fig. 2** Simple Spring-Cloud-based microservice architecture example

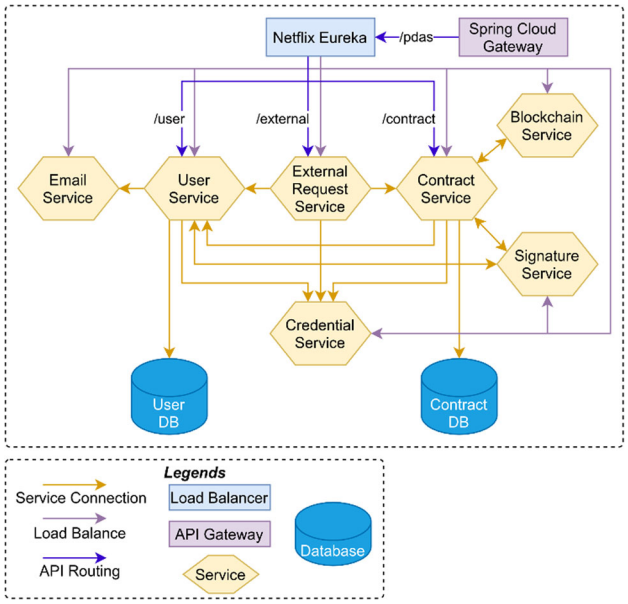


**Fig. 3** Spring-Cloud-based PDAS architecture

After working out the structure of the microservice PDAS, we had two options by which to convert the monolithic PDAS into microservices: Spring Cloud or Kubernetes. In the following sections, we compare the two approaches in the context of the PDAS.

### 3.1 Option 1: from spring boot to spring cloud

Spring Cloud is the first choice for building a microservice system using Spring Boot. The fact that Spring Cloud belongs to the same ecosystem as Spring Boot makes it intuitive to learn and adopt. It also provides numerous tools to assist in the construction of a microservice system. The Eureka service registry allows microservices to register themselves, while Feign allows them to communicate with each other. This makes it possible to redeploy a given system in various environments without altering the configuration of dependent services. Under the legacy Spring and Netflix OSS architecture, Eureka can be used with Ribbon to perform load balancing, as well as Zuul as an API gateway. Under recent architectures, Spring Cloud Gateway is used as an API gateway, while Eureka performs load balancing. Fault tolerance is dealt with using Hystrix to implement circuit breakers, timeouts, and retries with the aim of protecting the microservice from external service failure. Figure 2 presents an example of the Spring Cloud architecture.

The proposed Spring-Cloud-based PDAS used Feign for service communication, Spring Cloud Gateway for API gateway, and Eureka for service registry and load balancing (see Fig. 3). All microservices and databases were packaged within container images and deployed using Docker.

### 3.2 Option 2: from spring boot to Kubernetes

Kubernetes is a container management platform made popular by its ability to containerize any technology stacks used to build a given service. Kubernetes also provides tools to facilitate the development and deployment of microservices. Kubernetes uses various resource definitions to define the expected status of the Kubernetes cluster in auto-scheduling and dispatching tasks across multiple nodes in a cluster. Kubernetes allows third parties to customize and extend the functionality of a part or even the whole Kubernetes cluster by defining only the base specifications. Third-party developers can, for example, provide a better failover mechanism for stateful applications [22], introduce an optimized scheduler for cloud-edge computing [23], or create an auto-scaler optimized for a cloud-based microservice system [24].

Kubernetes provides an HTTP (Hypertext Transfer Protocol) interface (kube-apiserver) as well as a command line interface (kubectl) to facilitate expansion from external systems. The cluster can be modified by constructing the corresponding definition file in YAML format and then applying it using the command line interface.

Kubernetes can manage a cluster of different machines and automatically dispatch workloads across the cluster to maximize efficiency. Figure 4 presents an example of a Kubernetes multi-node cluster.

Kubernetes controls all of the resources in a cluster by applying various resource definitions, the relationships of which are shown in Fig. 5. The smallest unit is a Pod, which can contain multiple containers. The Deployment module can be used to define a template to create a Pod and control Pod replications. The Service module defines connections to Pods managed by Deployment, while simultaneously working as a load balancer. The Ingress module controls traffic going into the cluster and acts as a proxy server or API gateway. Pods can be used to auto-scale high process loads or heavy network traffic by combining Deployment with the Horizontal Pod Autoscaler (HPA). The definition of Persistent Volume (PV)

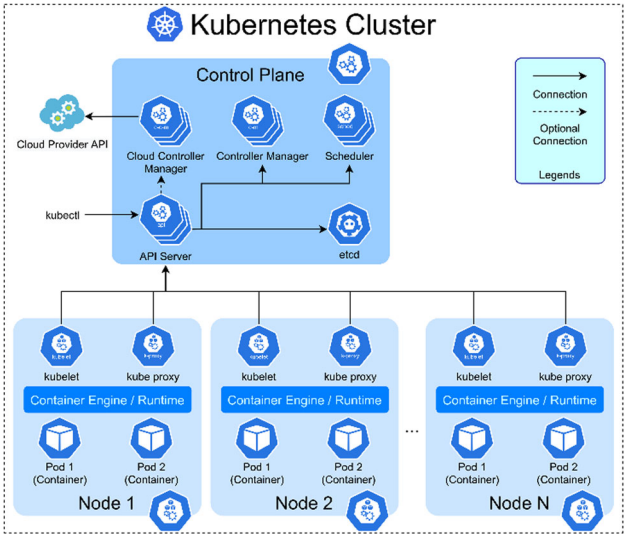


**Fig. 4** Kubernetes multi-node cluster example

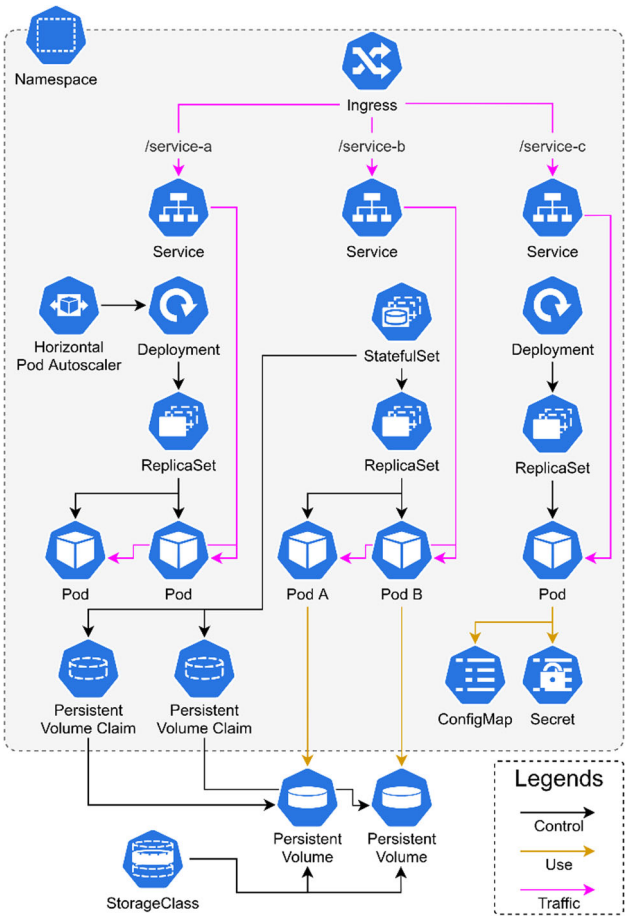


**Fig. 5** Kubernetes resources relationship

and Persistent Volume Claim (PVC) using the StatefulSet module preserves the state of the service in the cluster.

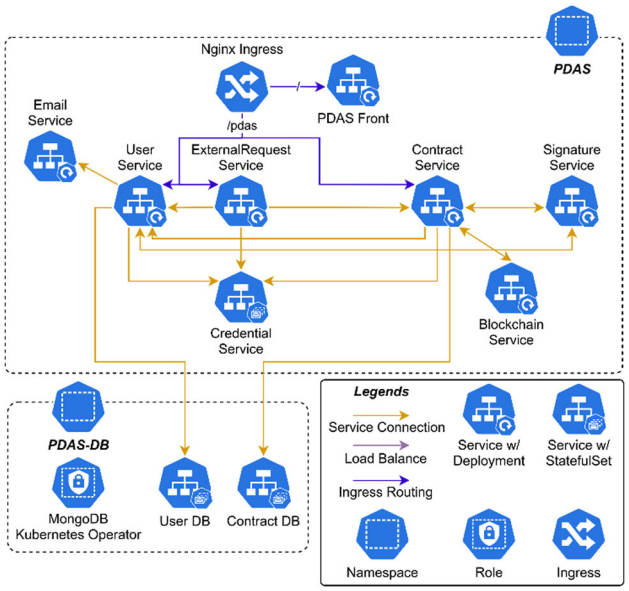


**Fig. 6** Kubernetes-based PDAS architecture

The Kubernetes-based PDAS used the Service module to perform load balancing and the Nginx Ingress module as an API gateway. In the current study, databases were managed using MongoDB Community Kubernetes Operator to create MongoDB replica sets. The system structure is shown in Fig. 6.

## 4 Comparisons from various perspectives

In the following sections, the proposed PDAS is used as a benchmark to compare the characteristics of the Spring Cloud and Kubernetes-based architectures in terms of development, deployment, fault tolerance, scalability, extensibility, and monitoring. We also examined the systems from the functionality perspective, including API gateway, service discovery, load balancing, circuit breaker, distributed tracing, and centralized metrics. The tools used by the two systems are shown in Fig. 7.

### 4.1 Development

The tools and options provided by a migration system are essential to developers. In this section, we compare the available tools and the experience of developers when using the two approaches to build a microservice system.

(1) ***Spring Cloud:*** Conversion of a Spring-Boot-based monolithic system into a Spring-Cloud-based microservice system involves dividing the monolith into multiple independently deployable microservices. The Eureka Client and Feign are required to be used by depending

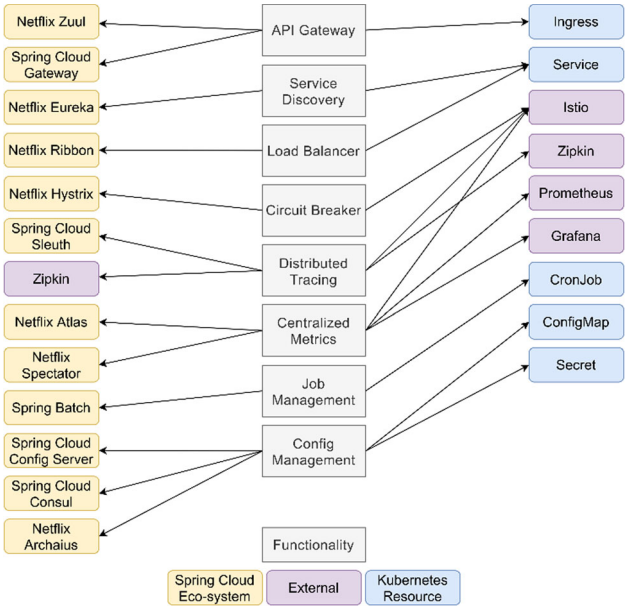


**Fig. 7** Spring Cloud Ecosystem vs. Kubernetes

service (a service that utilizes another service), thereby making it possible to call up a service endpoint of a dependent service via the declared Feign interface. Note that a Eureka server is required for service registration. Services with the Eureka Client enabled can communicate with each other after obtaining the runtime address from the Eureka server. Spring Cloud Gateway provides functionalities to create an API gateway to handle external requests.

(2) ***Kubernetes:*** Conversion into a Kubernetes-based microservice system also involves dividing the monolith into multiple parts, before setting up the Deployment and Service modules, defining the API gateway, and using Ingress to handle external requests.

The similarities between Spring Boot and Spring Cloud greatly lower the learning curve. Most of the tools from Spring Cloud are intrusive, operating as programming-layer plugins, such that most of the logic processes are implemented in Java code. The fact that Kubernetes controls containers in a nonintrusive way means that making services work as intended requires extensive configuration. Nonetheless, as a container management platform, Kubernetes supports a variety of programming languages and technologies.

Spring Cloud and Spring Boot employ the same programming style, involving the use of annotations (or decorators in other languages) to define and configure components and settings. One downside of using annotations is the fact that even when critical annotations are missing, most of the compilation will be successful (i.e., the critical annotation will simply be ignored at runtime), resulting in unexpected behaviors. The lack of error messages greatly hinders debugging. Note also that the overuse of annotations can negatively affect the readability of the program. Consider an example situation in which APIs in Spring Boot's RestController use the Swagger plugin to generate API documents and Hystrix for fault tolerance. This would require the addition of at least three annotations in front of every API definition and at least two annotations in front of every API parameter, thereby making it inconvenient to identify mistakes and impending maintenance.

The tools used to convert Spring Boot into a Spring Cloud-based system are familiar; however, each service requires extensive modification. Note also that dividing up the monolith, adding Spring Cloud dependencies, setting up Feign interfaces, and configuring Eureka settings in the Spring Boot application properties file are all intrusive processes.

Microservice systems based on Kubernetes do not require management of the service registry, thereby eliminating the need for extra dependencies for each service. After defining a given service, a unique cluster-wide service URI (Uniform Resource Identifier) is created, thereby allowing all dependent services to send requests to the same URI.

Most of the settings of tools from Spring Cloud are placed in the same location used in Spring Boot; however, some of the tools allow the assignment of settings to the program, thereby fragmenting the configuration. By contrast, Kubernetes uniformly uses the YAML format for configuration. Note, however, that as the system becomes larger, the configuration files will grow longer and/or increase in quantity.

In developing a Spring-Cloud-based system, virtual machines and container engines (e.g., Docker or containerd) can be used for testing. In developing our Kubernetes system, a range of simulators were available (e.g., Minikube or Kind).

When converting PDAS into a microservice system, we encountered fewer programming problems with Kubernetes-based systems, mainly due to nonintrusive settings.

Both microservice architectures (Spring Cloud and Kubernetes) provide the ability to deploy simple microservices independently to enable parallel development with few conflicts in terms of version control.

## 4.2 Deployment

In this section, we discuss differences in deployment using Spring Cloud or Kubernetes.

Under the Spring Cloud architecture, services communicate using Feign, such that only the Eureka server path in the configuration files must be modified to deal with different environments. Note that the Eureka server provides a simple web interface presenting basic information for registered services.

Defining a Service in a Kubernetes cluster involves registering a unique service URI in the cluster DNS, thereby allowing the hard coding of URIs for dependent services within the production profiles of a Spring Boot application. Simply applying the production profiles during deployment allows the normal invocation of dependent services.

Kubernetes provides a complete set of tools to manage containers for advanced deployments. Blue-green deployment, rolling release, and canary release can be achieved simply by modifying the deployment settings. Note, however, that similar features can be accessed under the Spring Cloud architecture by fine-tuning the load balancer and reverse proxy server.

Note that Spring Cloud does not manage databases because it only provides features built around Spring Boot applications. By contrast, Kubernetes makes it easy to manage containerized databases, usually by setting up StatefulSet. For example, MongoDB has a customized Kubernetes Operator using Custom Resource Definitions (CRDs) to describe the replica architecture for MongoDB, and controls the creation of StatefulSet automatically using a Role.

Note also that major cloud providers (e.g., Azure Kubernetes Service, Amazon Elastic Kubernetes Service, Google Kubernetes Engine, and Linode Kubernetes Engine) also provide tools that could be used to self-host a production-ready Kubernetes cluster (e.g., kubeadm or kOps).

In this study, we deployed PDAS using Spring Cloud and Kubernetes-based based on three different machines. Before initializing the Spring-Cloud-based PDAS, the delay must be manually adjusted to avoid start-up crashes associated with discrepancies in initialization timings. The deployment of Kubernetes-based PDAS was relatively smooth on all three machines without modifying any of the start-up scripts.

The fact that a microservice system is distributed means that a variety of settings must be modified to ensure unobstructed communication among services and correct processing of external requests. As a result, when implementing a simple system architecture, deploying a microservice system tends to be more expensive than deploying a monolithic system.

### 4.3 Fault tolerance

When errors occur in external services, fault tolerance mechanisms can protect the system and prevent cascading failures. In this section, we discuss the implementation of Spring Cloud with Netflix Hystrix versus Kubernetes with Istio.

Implementing a fault tolerance mechanism using Netflix Hystrix requires the inclusion of rules in front of function definitions (via annotations). Circuit breakers, timeouts, and retries can also be used to enhance fault tolerance, and fallback functions can be defined to handle exceptions.

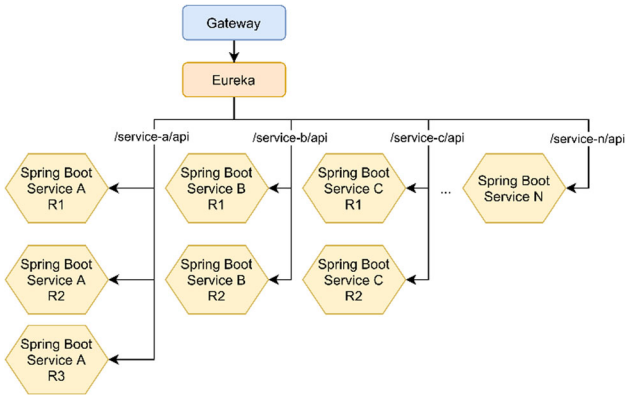


**Fig. 8** Example Spring Cloud service replication

Istio is an implementation of a service mesh in Kubernetes. It extends networking functionality using a sidecar mechanism to monitor and manage traffic in and out of every Pod. Circuit breakers based on network traffic can be defined using the Istio DestinationRule, wherein an Istio proxy in the sidecar returns HTTP 503 and prevent requests from going into the Pod in cases where the traffic has reached a predefined limit, thereby protecting the service from crashes. Retries and timeouts can also be defined using the Istio VirtualService with the Istio proxy handling retries on target services.

We observed that Hystrix and Istio differ in terms of behavior. Hystrix implements a self-protect mechanism to prevent exceptions from external services, while Istio filters traffic using a proxy layer to protect services. Note that Spring Boot applications running on Kubernetes benefit from both methods by handling errors from external services using Hystrix and protecting service from external requests by configuring Istio rules.

### 4.4 Scalability

Scalability becomes an important issue in high-load or high-traffic scenarios. In this section, we compare the creation of replications in a Spring Cloud implementation versus Kubernetes.

Under a naive containerized Spring Cloud architecture, increasing the number of replications involves manually starting a new service instance. The Eureka server can easily achieve the load balance between instances, an example of which is shown in Fig. 8. Although Spring Cloud lacks a central management tool for scalability, the scalability of Spring Cloud architecture still exceeds that of monolithic architectures in a manual way.

In a Kubernetes cluster, simply modifying the replication value in the Deployment settings makes it possible to automatically schedule the creation or termination of Pods. The Horizontal Pod Autoscaler (HPA) allows Kubernetes to automatically increase or decrease the number of replications in

accordance with predefined conditions. In the experiment section, we present two predefined scaling metrics. Note that researchers have previously demonstrated the dynamic adjustment of scaling metrics [25], and the implementation of strategies in accordance with workload patterns [24].

### 4.5 Extensibility

It is possible to add new services to the microservice system over time; however, the ease with which this can be achieved depends on system extensibility. In this section, we discuss the extensibility of microservice systems based on Spring Cloud and Kubernetes.

When adding new services under the Spring Cloud architecture, it is crucial to consider the current loading of the Eureka server and the remaining capacity of the host machine. In the event that the Eureka server is full or the host machine is nearly full, it becomes necessary to deploy multiple instances of the Eureka Server and distribute services across multiple machines. This requires a complex networking setup; however, Spring Cloud does not provide tools for network management.

When adding new services under the Kubernetes architecture, the only thing to consider is the cluster capacity. In the event that all nodes are close to being full, another machine can simply be added as an additional node in the cluster to resolve the issue. Kubernetes automatically schedules a new service for the newly created node. Some cloud providers also allow the automatic addition of new nodes to the cluster to process sudden peaks in requests.

Kubernetes manages scheduling and synchronization across all nodes in the cluster using the Kube API Server, thereby simplifying the configuration process. Taken together, it appears that Kubernetes is superior to Spring Cloud in terms of flexibility. Nonetheless, Spring-Cloud-based microservice systems are superior to monolithic architectures, which require the recompilation and redeployment of the entire monolith.

### 4.6 Monitoring

Monitoring a microservice system can provide a great deal of useful information to serve as a metric for automation, or as a reference by which to improve system performance. Spring Cloud and Kubernetes provide a wide range of monitoring tools for different situations. In this session, we compare the differences between Spring Cloud using VMAMVS (Version-based Microservice Analysis, Monitoring, and Visualization System) [26], a monitoring system previously developed by our team, and Kubernetes using Istio and the associated plugins.

VMAMVS extends Feign and Swagger plugins for Spring Boot to enable an analysis of endpoint-level dependencies and trends in failures based on historical records. VMAMVS quantifies the risk of error by combining high and low dependencies with previous error records. The Istio sidecar collects a variety of metrics and sends them to Prometheus. Grafana visualizes metrics collected in Prometheus in real time. The Kiali dashboard presents in real-time a traffic-based dependency graph as well as a history of traffic patterns and errors and traffic among services.

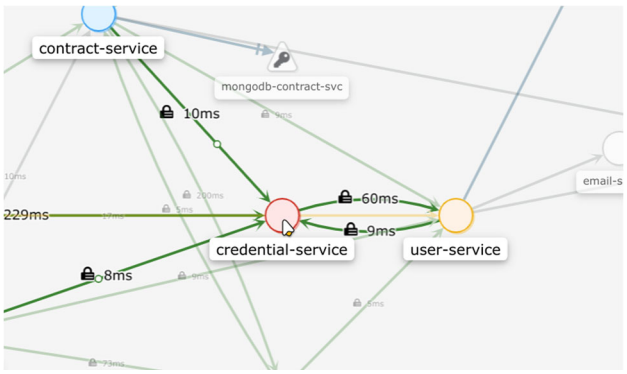


**Fig. 9** Kiali highlighting a service node with high traffic loading (red) (Color figure online)

VMAMVS and Istio both enable distributed tracing. In VMAMVS, Sleuth and Zipkin work together to record the flow of requests between services. Enabling the Zipkin plugin in Istio allows the Istio sidecar to automatically tag new requests and send records to Zipkin for further processing.

Istio is a useful tool to monitor network traffic in a Kubernetes cluster. The Kiali dashboard makes it possible to visualize traffic among services and use the direction of the request to analyze the corresponding dependencies. Kiali also makes it possible to monitor request errors (HTTP 4XX, 5XX) and traffic throughput and then notify the administrator of potentially risky services, as shown in Fig. 9. Istio can work with Prometheus and Grafana to enable the collection of data and visualization of Istio metrics.

## 5 Evaluation experiments

In addition to modularization, the most valuable feature of a microservice system is the ability to perform scaling in accordance with workload and identify bottlenecks. In this section, we compare the experimental performance results obtained using Spring Cloud and Kubernetes-based systems, mainly focusing on the feature of automatic scaling. The rationale for emphasizing the quantitative evaluation of scaling is twofold: (1) microservice systems are naturally designed to be scalable so that they can handle increased demand as the application grows. Scaling is a crucial feature to maintain performance and avoid bottlenecks as requests grow or even burst, and (2) since there are many variables involved in the autoscaling

**Table 1** Average error rates of the three systems

| Req/min | M | SC | K | SC I2 | K I2 |
|---|---|---|---|---|---|
| 30 k | – | – | – | – | 5.91% |
| 28 k | – | – | – | – | 0% |
| 25 k | – | – | – | 18.51% | 0% |
| 23 k | – | – | – | 5.05% | – |
| 20 k | – | – | 26.20% | 0% | – |
| 15 k | – | – | 0% | – | – |
| 12 k | 19.13% | 12.16% | 0% | – | – |
| 11 k | 13.99% | 5.63% | – | – | – |
| 10 k | 4.48% | 0% | – | – | – |
| 9 k | 0.00% | – | – | – | – |

*M*: monolithic, *SC*: Spring Cloud, *K*: Kubernetes, *I2*: two instances

**Table 2** Average throughput (requests/s)

| Req/min | M | SC | K | SC I2 | K I2 |
|---|---|---|---|---|---|
| 30 k | – | – | – | – | 83.6 |
| 28 k | – | – | – | – | 78.6 |
| 25 k | – | – | – | 96.4 | 76.7 |
| 23 k | – | – | – | 82.7 | – |
| 20 k | – | – | 40.5 | 78.4 | – |
| 15 k | – | – | 43.9 | – | – |
| 12 k | 46.8 | 44.6 | 39.3 | – | – |
| 11 k | 45.7 | 41.6 | – | – | – |
| 10 k | 41.1 | 39.2 | – | – | – |
| 9 k | 39.1 | – | – | – | – |

*M*: monolithic, *SC*: Spring Cloud, *K*: Kubernetes, *I2*: two instances

process, it is expected that the characteristics of the autoscaling mechanism can be quantitatively analyzed to provide a reference for the setting of autoscaling rules.

Note that performance was evaluated using the same setup for both systems. The Monolithic, Spring Cloud, and Kubernetes-based systems were run on separate Archlinux virtual machines with eight Ryzen 3700X CPUs and 32 GB of RAM. As a container runtime, we used Docker (version 20.10.6) and Kubernetes (version 1.21.0). In assessing autoscaling, we increased the number of CPUs to 16 without changing any of the other parameters. Requests were created using JMeter[3] and every test case used a ramp time of 60 s.

## 5.1 Performance

Stress testing was performed using the backend API with the highest workload in the PDAS system. We added random delays of 4–6 s to the test program to simulate complex operations without saturating the CPUs.

We first compartmentalized the monolithic PDAS using Spring Cloud or Kubernetes-based PDASs involving one or two instances of every service.

JMeter was used to gather error rates, as shown in Table 1. Overall, the error rates of the three systems were ranked as follows: Kubernetes-based PDAS < Spring-Cloud-based PDAS < monolithic PDAS. Note that the inclusion of two instances of every service improved performance, resulting in load capacities beyond those of the monolithic PDAS.

Our review of the data revealed that Spring-Cloud-based PDAS had a higher throughput (Table 2), albeit with a large number of errors. We posit that the high throughput can be attributed to request overflows resulting in errors. Besides, deploying two instances of microservices improves the average error rate. With two instances of a service, requests can be distributed between them; it means that if one service instance is experiencing high traffic or is unavailable due to some issue, the other instance can handle the requests, which eventually helps to reduce errors.

[3] https://jmeter.apache.org/.

**Table 3** Pod replication while autoscaling using CPU metrics

| Operating time (Sec) | Bl | Co | Cr | Em | Ex | Si | Us |
|---|---|---|---|---|---|---|---|
| 0 | 1/1 | 1/1 | 1/1 | 1/1 | 1/1 | 1/1 | 1/1 |
| | Initial state | | | | | | |
| 34 | 1/1 | 1/3 | 1/3 | 1/1 | 1/3 | 1/3 | 1/3 |
| | Start scaling after high loading over a period of time | | | | | | |
| 48 | 1/1 | 3/3 | 3/3 | 1/1 | 3/3 | 3/3 | 3/3 |
| | Finish scaling | | | | | | |
| 741 | 1/1 | 1/1 | 1/1 | 1/1 | 1/3 | 1/3 | 1/2 |
| | Start downscaling 5 min after loading returned to normal | | | | | | |
| 755 | 1/1 | 1/1 | 1/1 | 1/1 | 1/1 | 1/1 | 1/1 |
| | Finished state | | | | | | |

Service name abbreviation: *Bl.* Blockchain; *Co.* Contract; *Cr.* Credential; *Em.* Email; *Ex.* External; *Si.* Signature; *Us.* User

**Table 4** Metric values reported by Horizontal Pod Autoscaler (HPA)

| Time (Sec) | Co | Cr | Ex | Si | Us |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 77 | ***24.133*** | **14.866** | *5.851* | **10.339** | ***48.185*** |
| 100 | **12.112** | *6.633* | **16.583** | *6.233* | ***29.655*** |
| 337 | 1.466 | 0 | 0.160 | 0.651 | 2.012 |
| 533 | 0 | 0 | 0 | 0 | 0 |

Blockchain and Email are not shown because their values are consistently 0 throughout the testing

## 5.2 Automatic scaling

Kubernetes provides the Horizontal Pod Autoscaler (HPA) to automate the control of service replications. HPA uses a variety of monitoring metrics by which to adjust the number of replications. Note that, in this section, since Spring Cloud can only provide manual scaling, we conducted automatic scaling experiments for Kubernetes alone.

### 5.2.1 CPU

Using CPU usage as a monitoring indicator, the max CPU limit for each service was set at 100 m (10% for every core for a multi-core system), with the threshold set to 50% of the max limit. Note that replications were configured to have between 1 and 3 Pods per service.

In this experiment, we used Blockchain, Contract, Credential, Email, External, Signature, and User services in PDAS, the results of which are shown in Table 3. The traffic was configured to have 30 k requests per minute and last for exactly one minute. The recorded data is the number of running Pods and the number of initiated Pods (for example, 1/3 indicates three initiated Pods and only one running Pod).

The total error rate was 16%, and all errors (HTTP 502) appeared during the Pod creation stage of the autoscaling process. We posit that this can be attributed to the newly started Spring Boot application receiving requests during initialization. Even configuring the Spring Actuator did not resolve the issue, due to the fact that it was reported as ready before the application was fully initialized.

Autoscaling was based on the monitoring indicators associated with four types of delay, including readiness delay, metric resolution time, auto scaler sync period, and stabilization window. The readiness delay affected only newly started Pods with a default value of 30 s, the metric resolution time was set to one minute, the default auto-scaler sync period was 15 s, and the default upscaling stabilization window was 0 s. Summing up all of the delays, Kubernetes should react to the changes within one minute, even in the worse-case scenario. Multiple rounds of testing generated several results between 30 to 40 s before the HPA began scaling up the deployment.

### 5.2.2 Network traffic

Custom metrics can also be used for autoscaling. In the current study, we used Istio and the Prometheus plugin for Istio in conjunction with a Custom Metrics Server to collect and query custom indicators. We also assessed the possibility of

**Table 5** Pod replication while autoscaling using custom metrics

| Operating time (Sec) | Bl | Co | Cr | Em | Ex | Si | Us |
|---|---|---|---|---|---|---|---|
| 0 | 1/1 | 1/1 | 1/1 | 1/1 | 1/1 | 1/1 | 1/1 |
| | Initial state | | | | | | |
| 32 | 1/1 | 1/2 | 1/2 | 1/1 | 1/2 | 1/2 | 1/2 |
| | Start the first stage of upscaling after high loading over a period of time | | | | | | |
| 46 | 1/1 | 2/3 | 2/3 | 1/1 | 2/3 | 2/3 | 2/3 |
| | Second stage upscaling | | | | | | |
| 389 | 1/1 | 3/3 | 3/3 | 1/1 | 3/3 | 3/3 | 3/3 |
| | Start downscaling 1 min after loading returned to normal | | | | | | |
| 462 | 1/1 | 1/1 | 1/1 | 1/1 | 1/1 | 1/1 | 1/1 |
| | Finished state | | | | | | |

combining multiple metrics by configuring a custom indicator presenting the total number of requests in one minute divided by memory usage. After several tests, we set the scaling threshold for this custom metric at 4. The results for HPA metrics and numbers of Pods are shown in Tables 4 and 5. Note that due to the limitation of HPA, the upper limit for the number of Pods can be only three. Nevertheless, the experiment results indicated that metrics based on the monitoring of network traffic provide far finer control over autoscaling.

## 6 Conclusions

This paper compares Spring-Cloud-based and Kubernetes-based microservices from various perspectives, including overall flexibility, deployment tools, fault tolerance, scaling, extensibility, and monitoring tools. Although both microservice systems provide notable benefits over conventional monolithic systems, taken together, it appears that Kubernetes has several advantages over Spring Cloud as a monolith migration platform. The benefits include support for various technology stacks, a nonintrusive setup, superior scaling, fine control over containers, and better overall performance. The competition and cooperation relationships between Kubernetes and Spring Cloud can be sorted out as follows:

(1) Competition: Kubernetes provides a platform for deploying, scaling, and managing containerized applications, while Spring Cloud provides a set of tools for application building and deploying. Overall, Kubernetes and Spring Cloud compete for deployment and management capabilities.
(2) Cooperation: Kubernetes and Spring Cloud can also be used together in complementary ways. Spring Cloud applications can be deployed to Kubernetes clusters using tools like Kubernetes Deployment API. Developers can use the Java-based mechanism to realize service discovery and circuit breakers, and rely on Kubernetes to achieve better autoscaling and application performance.

Overall, Kubernetes is a better fit for applications that require high scalability and fault tolerance. Spring Cloud, on the other hand, is a better fit for situations where developers are familiar with service configuration and integration in a programming way.

The scientific contributions of this paper include the following:

(1) As far as we know, this paper is the first to analyze the target framework and platform when migrating a monolith to a microservice architecture.
(2) This paper advances the understanding of Kubernetes and Spring Cloud at both the conceptual and implementation levels.
(3) This paper proposes a holistic scheme to evaluate the two technologies, including the qualitative comparison dimensions and the design of quantitative experimental evaluations.

In the future, we plan to develop patterns to help organizations understand when Spring Cloud is applicable and when Kubernetes is more suitable, and devise simulation mechanisms to analyze the structure and behaviors of a microservice application before deploying it to the target framework/platform.

**Acknowledgements** This research was sponsored by the National Science and Technology Council (NSTC) in Taiwan under the grant 110-2221-E-019-039-MY3.

## Declarations

**Conflict of interest** All authors declare that they have no conflicts of interest related to the contents of this article.

## References


1. Balalaie A, Heydarnoori A, Jamshidi P (2016) Microservices architecture enables DevOps: migration to a cloud-native architecture. IEEE Softw 33(3):42–52. https://doi.org/10.1109/ms.2016.64
2. Ren Z et al. (2018) Migrating web applications from monolithic structure to microservices architecture. In: Internetware '18: the tenth Asia-Pacific symposium on internetware, 2018/09/16/ 2018, Beijing China: ACM, pp 1–10, https://doi.org/10.1145/3275219.3275230
3. Fan C, Ma S (2017) Migrating monolithic mobile application to microservice architecture: an experiment report. In: 2017 IEEE international conference on AI & mobile services (AIMS), 25–30 June 2017, pp 109–112, https://doi.org/10.1109/AIMS.2017.23
4. Gan Y, Delimitrou C (2018) The architectural implications of cloud microservices. IEEE Comput Archit Lett 17(2):155–158. https://doi.org/10.1109/LCA.2018.2839189
5. Ma S-P, Fan C-Y, Chuang Y, Liu IH, Lan C-W (2019) Graph-based and scenario-driven microservice analysis, retrieval, and testing. Futur Gener Comput Syst 100:724–735. https://doi.org/10.1016/j.future.2019.05.048
6. Gouigoux J, Tamzalit D (2017) From Monolith to microservices: lessons learned on an industrial migration to a web oriented architecture. In: 2017 IEEE international conference on software architecture workshops (ICSAW), 5–7 April 2017, pp 62–65, https://doi.org/10.1109/ICSAW.2017.35
7. Mazlami G, Cito J, Leitner P (2017) Extraction of microservices from monolithic software architectures. In: 2017 IEEE international conference on web services (ICWS), 25–30 June 2017, pp 524–531, https://doi.org/10.1109/ICWS.2017.61
8. Taibi D, Lenarduzzi V, Pahl C (2017) Processes, motivations, and issues for migrating to microservices architectures: an empirical investigation. IEEE Cloud Comput 4(5):22–32. https://doi.org/10.1109/MCC.2017.4250931
9. Balalaie A, Heydarnoori A, Jamshidi P, Tamburri DA, Lynn T (2018) Microservices migration patterns. Softw Pract Exp. https://doi.org/10.1002/spe.2608
10. Francesco PD, Lago P, Malavolta I (2018) Migrating towards microservice architectures: an industrial survey. In: 2018 IEEE international conference on software architecture (ICSA), 30 April-4 May 2018, pp 29–2909, https://doi.org/10.1109/ICSA.2018.00012
11. Almeida JF, Silva AR (2020) Monolith Migration Complexity Tuning Through the Application of Microservices Patterns. In: Jansen CA, Malavolta I, Muccini H, Ozkaya I, Zimmermann O (eds) Software architecture. Springer International Publishing, New York, pp 39–54
12. Al-Debagy O, Martinek P (2019) A new decomposition method for designing microservices. Period Polytech Electr Eng Comput Sci 63(4):274–281
13. Amiri MJ (2018) Object-aware identification of microservices. In: 2018 IEEE international conference on services computing (SCC), IEEE, pp 253–256
14. Ahmadvand M, Ibrahim A (2016) Requirements reconciliation for scalable and secure microservice (de) composition. In: 2016 IEEE 24th international requirements engineering conference workshops (REW), IEEE, pp 68–73
15. Rademacher F, Sorgalla J, Sachweh S (2018) Challenges of domain-driven microservice design: a model-driven perspective. IEEE Softw 35(3):36–43
16. Ma SP, Lu TW, Li CC (2022) Migrating monoliths to microservices based on the analysis of database access requests. In: 2022 IEEE international conference on service-oriented system engineering (SOSE), 15–18 Aug, 2022, pp 11–18, https://doi.org/10.1109/SOSE55356.2022.00008
17. Balalaie A, Heydarnoori A, Jamshidi P (2016) Migrating to cloud-native architectures using microservices: an experience report. In: Advances in service-oriented and cloud computing, Cham: Springer International Publishing, pp 201–215
18. Cosmina I, Cosmina I (2017) Spring microservices with spring cloud. In: Pivotal certified professional spring developer exam: a study guide, pp 435–459
19. Vayghan LA, Saied MA, Toeroe M, Khendek F (2018) Deploying microservice based applications with Kubernetes: Experiments and lessons learned. In: 2018 IEEE 11th international conference on cloud computing (CLOUD), 2–7 July, 2018, pp 970–973, https://doi.org/10.1109/CLOUD.2018.00148
20. Vayghan LA, Saied MA, Toeroe M, Khendek F (2019) Microservice based architecture: towards high-availability for stateful applications with Kubernetes. In: 2019 IEEE 19th international conference on software quality, reliability and security (QRS), 22–26 July, 2019, pp 176–185, https://doi.org/10.1109/QRS.2019.00034
21. Wang YT, Wu CF, Ma SP, Chen HT, Chang SY, Li CS (2020) PDAS: a digital-signature-based authorization platform for digital personal data. In: 2020 international computer symposium (ICS), 17–19 Dec, 2020, pp 513–518, https://doi.org/10.1109/ICS51289.2020.00106
22. Vayghan LA, Saied MA, Toeroe M, Khendek F (2021) A Kubernetes controller for managing the availability of elastic microservice based stateful applications. J Syst Softw 175:110924
23. Kaur K, Garg S, Kaddoum G, Ahmed SH, Atiquzzaman M (2019) KEIDS: Kubernetes-based energy and interference driven scheduler for industrial IoT in edge-cloud ecosystem. IEEE Internet Things J 7(5):4228–4237
24. Taherizadeh S, Grobelnik M (2020) Key influencing factors of the Kubernetes auto-scaler for computing-intensive microservice-native cloud-based applications. Adv Eng Softw 140:102734
25. Rossi F, Cardellini V, Presti FL (2020) Hierarchical scaling of microservices in Kubernetes. In: 2020 IEEE international conference on autonomic computing and self-organizing systems (ACSOS), 17–21 Aug, 2020, pp 28–37, https://doi.org/10.1109/ACSOS49614.2020.00023
26. Ma S, Liu I, Chen C, Lin J, Hsueh N (2019) Version-based microservice analysis, monitoring, and visualization. In: 2019 26th Asia-Pacific software engineering conference (APSEC), 2–5 Dec, 2019, pp 165–172, https://doi.org/10.1109/APSEC48747.2019.00031