https://doi.org/10.31891/csit-2023-2-5
UDC 004.93

Oleksii MAKIEIEV, Natalia KRAVETS
Kharkiv National University of Radio Electronics

STUDY OF METHODS OF CREATING SERVICE-ORIENTED SOFTWARE SYSTEMS IN AZURE

The modern development of service-oriented software systems is accompanied by the wide use of cloud technologies, which affect the competitiveness of companies and their systems, which provide opportunities to expand the client base thanks to the coverage of several regions of the city or country.

The advantage of cloud services is availability in any part of the world where there is an Internet connection. Cloud providers provide a large volume of services for various needs: such as hosting, deployment of containers, file storage, databases, etc.

In particular, all the most popular cloud providers offer several options for creating service-oriented software systems, including both standard technologies and proprietary developments. This paper compares the methods of creating service-oriented software systems based on the Azure cloud platform: Azure Container Apps, Azure Kubernete Service, and Azure Red Hat OpenShift. The subject area of technologies for the implementation of service-oriented application architecture is considered, and criteria for the analysis of methods for implementing applications with such an architecture are proposed. A software solution for comparing methods of creating service-oriented applications based on the Azure cloud platform was designed and developed. The developed software system provides an opportunity to rent scooters, bicycles and cars.

The purpose of the study is a comparative analysis of the methods of creating service-oriented software systems based on Azure services, and the subject of the study is a software solution implemented using these methods.

The purpose of this work will be the development of a software system that will provide an opportunity to rent scooters, bicycles and cars. Using this system, we will investigate the deployment of this system on certain services from Azure.

The results of this research on Azure services: Azure Container Apps, Azure Kubernete Service and Azure Red Hat OpenShift can be used when creating a new software system, when expanding an existing software system, when transferring software system components from other platforms to the Azure platform using these services.

Keywords: service-oriented software system, cloud technologies, Azure cloud, Azure Container Applications, Azure Kubernetes Service, Azure Red Hat Open Shift, docker, web-services.

Олексій МАКЄЄВ, Наталія КРАВЕЦЬ
Харківський національний університет радіоелектроніки

ДОСЛІДЖЕННЯ МЕТОДІВ СТВОРЕННЯ СЕРВІСНО-ОРІЄНТОВАНИХ ПРОГРАМНИХ СИСТЕМ У AZURE

Сучасний розвиток сервісно-орієнтованих програмних систем супроводжується широким використанням хмарних технологій, які впливають на конкурентоспроможності компаній та їх систем, що надають можливості в розширенні клієнтської бази завдяки охопленню декількох областей міста чи країни.

Перевагою хмарних сервісів є доступність в будь-який точці світу, де є підключення до Інтернету. Хмарні провайдери надають великий обсяг сервісів для різних потреб: таких як хостинг, розгортання контейнерів, файлове сховище, бази даних тощо.

Зокрема всі найпопулярніші хмарні провайдери пропонують кілька варіантів створення сервісно-орієнтованих програмних систем, включаючи як стандартні технології так і власні розробки. У даній роботі виконане порівняння методів створення сервісно-орієнтованих програмних систем на базі хмарної платформи Azure: Azure Container Apps, Azure Kubernete Service та Azure Red Hat OpenShift. Розглянуто предметну область технологій реалізації сервісно-орієнтованої архітектури застосунків, запропоновано критерії для аналізу методів реалізації застосунків із такою архітектурою. Спроектовано та розроблено програмне рішення для порівняння методів створення сервісно-орієнтованих застосунків на базі хмарної платформи Azure. Розроблена програмна система надає можливість брати в оренду самокати, велосипеди та автомобілі.

Метою дослідження є порівняльний аналіз методів створення сервісно-орієнтованих програмних систем на базі сервісів Azure, а предметом дослідження – програмне рішення, яке реалізоване за допомогою цих методів.

Отримані результати даного дослідження над Azure сервісами: Azure Container Apps, Azure Kubernete Service та Azure Red Hat OpenShift, можна буде використовувати при створені нової програмної системи, при розширенні існуючої програмної системи, при перенесенні компонентів програмної системи з інших платформ на Azure платформу використовуючи дані сервіси.

Ключові слова: сервісно-орієнтована програмна система, хмарні технології, хмара Azure, програми-контейнери Azure, сервіс Kubernetes Azure, Azure Red Hat OpenShift, докер, веб-сервіси.

Introduction

Service-oriented architecture [1] is a software architectural pattern that uses a modular approach to software development based on the use of distributed, loosely coupled replaceable components equipped with standardized interfaces to interact using standardized protocols.

Service-oriented architecture is not tied to a specific technology. It can be implemented using a wide range of technologies, including technologies such as Web services, message-oriented middleware, enterprise service bus, microservices.

This architecture can include the following elements, as can be Application frontend, services, service bus or message broker, data storage. These elements can be loosely connected services that interact through a strictly defined interface with each other.

When implementing a service-oriented architecture, it is necessary to attach to certain principles, namely:

- ensuring compatibility;
- weak interdependence;
- abstraction;
- degree of detailing.

Ensuring contiguity means that any system will be able to run the service regardless of the underlying platform or programming language. For example, business processes can use services written in C#, Java, and Python.

The principle of weak interdependence suggests that services should be weakly connected, which should have as few external dependencies as possible. That way, if you change a service, it won't affect client applications and other services that use that service.

The principle of abstraction means that users do not need to know the logic of the service code or the details of the implementation. For them, services should be like a black box.

The principle of the degree of detail means that the services should have an appropriate size and scope. Developers can use multiple services to create a composite service to execute complex logic.

Service-oriented architecture has a number of advantages, namely:

- reduction of market entry time;
- effective service;
- improved adaptability.

Efficient maintenance makes it easier to create, update, and debug small services than larger blocks of code in monolithic applications.

Improved adaptability makes it possible to modernize your programs effectively and without unnecessary costs.

For the study, service-oriented software for renting vehicles, namely scooters, bicycles, and cars, will be developed to investigate the deployment system on certain services from Azure.

Related Works

At the current time, we have some technologies for implementing service-oriented architecture:

- Microservice architecture[2];
- RESTful Web Services[3];
- GraphQL.

Microservice architecture is a specific implementation of a service-oriented architecture (SOA) that focuses on breaking down a large, monolithic system into smaller, independent services that can be developed, deployed, and scaled independently. Microservices are designed to be loosely coupled and communicate with each other through APIs.

In a service-oriented architecture, microservices can provide a number of benefits, including:

- Greater flexibility: Microservices allow for faster and more frequent deployments, as individual services can be developed and deployed independently of each other.
- Better scalability: Because microservices can be scaled independently, it's easier to handle changes in demand for specific services without impacting the rest of the system.
- Improved fault tolerance: By isolating each service, it's easier to handle failures in one service without impacting the rest of the system.

However, implementing a microservice architecture in a service-oriented architecture can also introduce some challenges. For example:

- Increased complexity: With more services, there is greater complexity in managing service-to-service communication, monitoring, and testing.
- Data consistency: Maintaining consistency across multiple services can be challenging, as each service may have its data store.
- Service discovery: In a microservice architecture, services need to be discoverable by other services, which can be challenging to manage.

Representational State Transfer (REST) is an architectural style that defines a set of constraints for building web services. RESTful web services conform to these constraints and are designed to be simple, lightweight, and scalable. They use HTTP methods such as GET, POST, PUT, and DELETE to perform CRUD (create, read, update, delete) operations on resources. In a service-oriented architecture, RESTful web services can be used to facilitate communication between services by providing a standard way for services to interact with each other. Each service can expose a RESTful API that other services can use to access its functionality.

Some benefits of using RESTful web services in a service-oriented architecture include:

- Scalability: RESTful web services are designed to be scalable, making it easy to handle changes in demand for specific services.

- Interoperability: Because RESTful web services use standard HTTP methods, they can be accessed by a wide variety of clients and platforms.
- Simplicity: RESTful web services are easy to understand and can be implemented using simple, lightweight frameworks.

However, implementing RESTful web services in a service-oriented architecture can also introduce some challenges. For example:

- Data consistency: Maintaining consistency across multiple services can be challenging, as each service may have its own data store.
- Service discovery: Services need to be discoverable by other services, which can be challenging to manage.
- Security: RESTful web services need to be secured against unauthorized access and attacks such as SQL injection.

GraphQL is a query language and runtime for APIs that was developed by Facebook in 2015. It provides a more flexible and efficient way for clients to request data from servers, allowing clients to specify exactly what data they need and reducing the number of round trips to the server. In a service-oriented architecture, GraphQL can be used to facilitate communication between services by providing a single API endpoint that aggregates data from multiple services.

One of the main benefits of using GraphQL in a service-oriented architecture is that it allows for greater decoupling between services. Because each service can expose its own GraphQL schema and resolvers, other services can query that service's data without needing to know the details of its internal implementation. This makes it easier to change or replace individual services without impacting the entire system.

However, implementing GraphQL in a service-oriented architecture can also introduce some challenges. For example, it requires careful consideration of data ownership and access control, as well as potential performance implications due to the increased number of round trips to the server. Additionally, it may require additional development effort to create and maintain the GraphQL schema and resolvers for each service.

The software system will be implemented using RESTful web services because web services are easy to implement using modern frameworks, web services can be used in different systems because they support HTTP methods, and web services can support scalability. Using the RESTful web services the system will be implemented for renting scooters, bikes, and cars. This system will be used in the Azure services to analyze methods for creating a service-oriented software systems.

Methods of creating service-oriented software systems in Azure

In Azure we choice three services for creating container applications:

- AzureContainerApps[4];
- AzureKubernetesService[5];
- AzureRedHatOpenShift[6].

Azure Container Apps is a service provided by Microsoft Azure that allows you to run and manage containerized applications without having to worry about the underlying infrastructure. It simplifies the process of deploying and managing applications in a containerized environment.

Azure Container Apps can be used in a variety of scenarios, as you can see on the Fig. 2, including:

- Running web applications: You can use Azure Container Apps to run web applications built on any technology stack, including Node.js, Python, PHP, .NET, and more.
- Running microservices: Azure Container Apps can be used to run microservices-based applications, allowing you to easily deploy and manage each service as a containerized application.
- Running batch processing jobs: You can use Azure Container Apps to run batch processing jobs, such as data processing or image rendering, in a containerized environment.
- Running AI and machine learning workloads: Azure Container Apps can be used to run machine learning models and other AI workloads in a containerized environment, providing a scalable and reliable platform for your applications.
- Running IoT workloads: Azure Container Apps can be used to run IoT workloads, such as data ingestion and processing, in a containerized environment, allowing you to easily scale your applications as needed.

Applications built on top of Azure Container Apps can scale dynamically based on the following characteristics:

- HTTP traffic;
- event-driven processing;
- CPU or memory load.

Azure Container Apps allows the execution of application code packaged in any container and is independent of the runtime environment or programming model.

Azure Container Apps manages automatic horizontal scaling using a set of declarative scaling rules. When a container application scales, new instances of the container application are created on demand. These instances are known as replicas. When you first create a container application, the scale rule is set to zero.

Azure Kubernetes Service (AKS) is a managed container orchestration service provided by Microsoft Azure. It allows you to deploy, manage, and scale containerized applications using the open-source Kubernetes orchestration engine.

Kubernetes is the de facto open source platform for orchestrating containers, but typically requires a lot of cluster management overhead. AKS helps manage much of the overhead by reducing the complexity of deployment and management tasks. AKS is designed for users and companies who want to build scalable applications using Docker and Kubernetes using the Azure architecture.

You can create an AKS cluster using the Azure Command Line Interface (CLI), the Azure portal, or Azure PowerShell. Users can also create template-based deployment options using Azure Resource Manager templates.

The main benefits of AKS are flexibility, automation, and reduced management costs for administrators and developers.

Some of the key features of AKS include:

- Automatic scaling: AKS can automatically scale your cluster based on the demand for your applications. This allows you to handle sudden increases in traffic without having to manually add more resources.
- High availability: AKS provides built-in high availability features, such as automatic node replacement and node redundancy, which ensure that your applications remain available even in the event of node failures.
- Security: AKS provides a secure environment for your containerized applications, with features such as network security policies, private cluster access, and integration with Azure Active Directory.
- Integration with other Azure services: AKS integrates with other Azure services, such as Azure Container Registry and Azure Monitor, which allows you to easily manage your containerized applications and monitor their performance.

An AKS deployment also spans two resource groups. One group is just a Kubernetes[7] service resource and the other is a node resource group. A node resource group contains all the infrastructure resources associated with the cluster. A service principal or managed identity is required to create and manage other Azure resources.

A Kubernetes module encapsulates a container and how packages are assembled into nodes. Kubernetes node can contain different pods. For example, Front-end, Back-end, undrelated pods, etc.

Azure Red Hat OpenShift (ARO) is a fully managed container platform offered jointly by Microsoft and Red Hat. It provides a Kubernetes-based platform for developing, deploying, and managing containerized applications as you can see the high-level architecture of Azure Red Hat OpenShift.

Running containers in production with Kubernetes requires additional tools and resources. This often involves juggling image registries, storage management, networking solutions, and logging and monitoring tools, all of which must be versioned and tested together.

Building container-based applications requires even greater integration with middleware, frameworks, databases, and CI/CD[8] tools. Azure Red Hat OpenShift brings it all together in a single platform, making it easier for IT teams while giving teams what they need to get things done. In addition, the Microsoft Azure Red Hat OpenShift service allows you to deploy fully managed OpenShift clusters.

Some of the key features of ARO include:

- Automatic scaling: ARO can automatically scale your cluster based on the demand for your applications. This allows you to handle sudden increases in traffic without having to manually add more resources.
- High availability: ARO provides built-in high availability features, such as automatic node replacement and node redundancy, which ensure that your applications remain available even in the event of node failures.
- Enterprise-grade security: ARO provides a secure environment for your containerized applications, with features such as network security policies, private cluster access, and integration with Azure Active Directory.
- Integration with other Azure services: ARO integrates with other Azure services, such as Azure Container Registry and Azure Monitor, which allows you to easily manage your containerized applications and monitor their performance.
- Support for legacy applications: ARO includes support for running legacy applications, such as databases or other stateful workloads, within the platform, making it easier to modernize your applications.

Design and development of the software solution

Designing an architecture for a service-oriented software system is a complex and important process with a complex subject area. Considering the subject area of the transport rental software system, a multi-tier architecture was chosen for the backend of the web application, and Flux was chosen for the client application.

The server and client application will communicate using the REST API, and the authorization mechanism will also be used. A relational database will be used to store data about transport, users and other subject models.

A relational database will be used to store data about transport, users and other object models.

All services that will be developed will be able to create a Docker image, so that they can then be placed in Docker containers, as shown in Fig. 1.

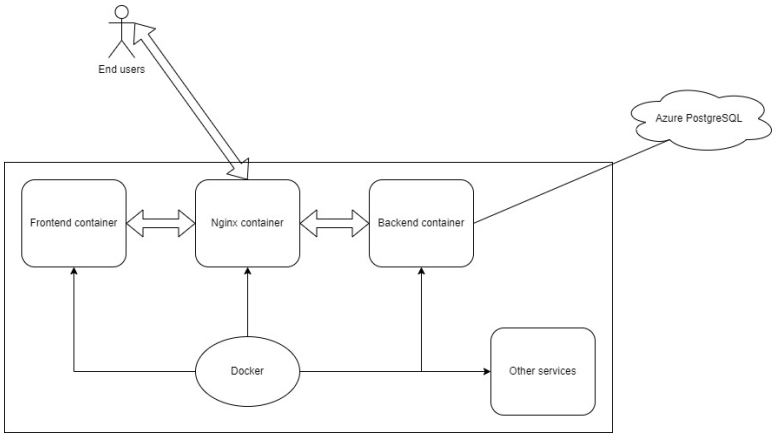

Fig. 1. The structure of Docker containers

The server part and the client part will be in Docker[9] containers, and nginx will handle requests.

Nginx is a web server used as a reverse proxy, load balancer, mail proxy and HTTP cache. Nginx[10] will redirect the request from the end user to the desired container.

Creation of Docker images will be done via Dockerfile.

A Dockerfile is a text document that contains all the commands that a user can call at the command line to build a Docker image. The Dockerfile will include the core files for building and configuring the project.

Building Docker images can be done locally for local testing and through a CI/CD process. If this is done through CI/CD, then in this case, the creation of Docker images should be done as the last step to check the service for correct execution of the main functionality.

To carry out planned studies of Azure services, it is necessary to build a prototype of a software system using design patterns, technologies, and certain architectural styles. This software system will be responsible for renting bikes, scooters, and cars and providing them in Ukrainian cities.

Backend: On the server side, we use the SOA architecture as shown in Fig. 2. The architecture contains 2 services: core service, mail service. These services communicate through the Azure Service Bus using the queue. Also, these services use the same database, Azure PostgreSQL. Each service is responsible for specific business logic. The core service includes the primary logic for renting transports. The mail service is responsible for sending emails to users to notify users, sending verification codes after registration, etc. Communication between services also uses protobuf to standardize sending and receiving data. Protocol Buffers (Protobuf) is a free and open-source cross-platform data format used to serialize structured data. It is useful in developing programs to communicate with each other over a network or for storing data. Also, communication between services and clients uses the API Gateway to hide requests to the services and provide user-friendly API to end users.

To start developing the business logic, it is first necessary to design the use cases of the system for different users using UML.

During the development of the system, a UML diagram was created, this diagram represents the roles of the users of the system, as well as information about the functionality available to each role, thanks to which you can get an idea of what the end user can get.

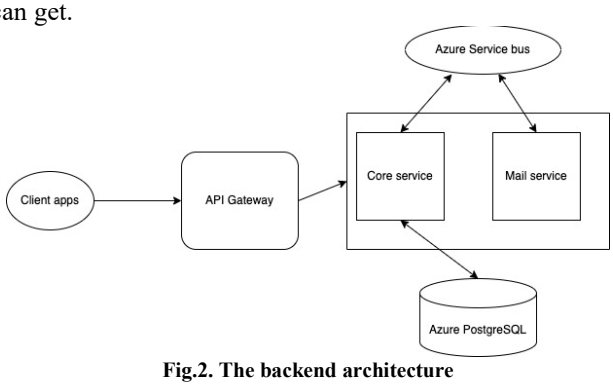

Fig.2. The backend architecture

Fig.3shows a capability diagram for a vehicle rental software system.

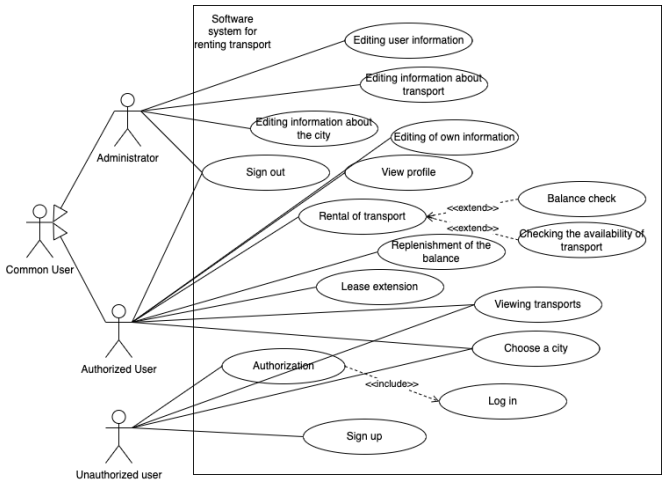

Fig. 3. UML for the software system

All system users are divided into two main roles:
- authorized user;
- unauthorized user.

This software system includes 2 roles of an authorized user:
- system administrator;
- authorized user.

Each role has its own restrictions on the use of functionality. A user who has the role of an unauthorized user can authorize in the system, that is, register or log in to the system. You can also view transport and select a city for further transport selection.

A user with the role of an authorized user can view transport, view a city, rent a vehicle, view his profile, edit his data and top up his balance.

A user with the role of an administrator manages user data, transport and cities.

We will use PostgreSQL[11] for data storage. The PostgreSQL database is compatible with several major programming languages and protocols, including C, C++, Go, Perl, Python, Java, .Net, Ruby, ODBC, and Tcl. This means that users will be able to work in the language they know best without the risk of system conflicts. To work with this database, we will use Azure Database for PostgreSQL server.

The services will be built on a multi-tier architecture to divide the logic. One of the biggest advantages of a multi-tier architecture is that new functionality can be easily added to the system. Making changes to one of the system levels will not affect the modification of the system components in any way if the interaction goes through interfaces and isolation of the model from other components.

In the component diagram, as shown in Fig. 4, you can see parts for each service and how services communicate with each other.

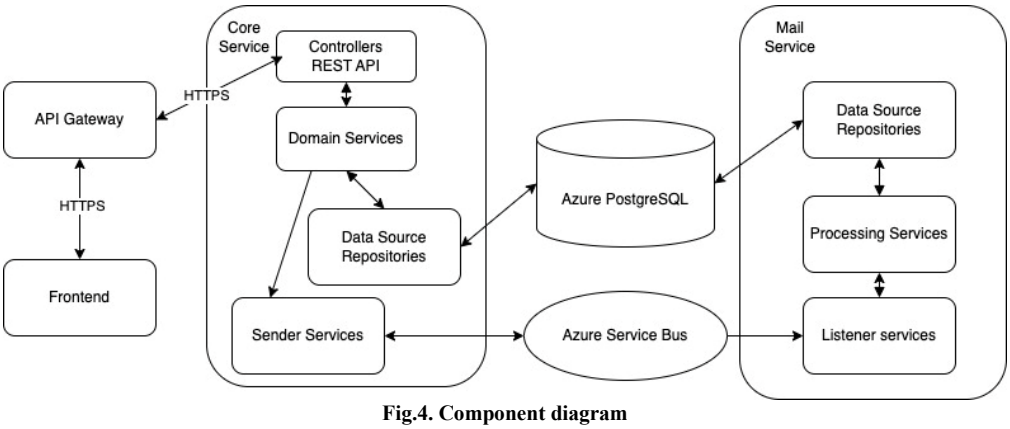

Fig.4. Component diagram

The core service contains four layers: controllers, domain services, data source, and sender services. Domain services can communicate with repositories and sender services. The sender services send messages to the Azure Service Bus, and others receive these messages. In this case, the Mail service contains the listener services to receive notifications from the Azure Service Bus. After getting messages, it starts processing them through the processing services responsible for providing a specific logic for sending mail to users. The data source layer includes repositories and communicates with Azure PostgreSQL.

The mail service contains three layers: listener services, processing services and data source. The listener services receive messages from the Azure Service Bus and start processing them using the processing services. The processing services define a pattern for sending emails. It can be password confirmation, suggestions of the day/week, new updates.

For implementing core and mail services we choice the Java and the Spring Framework. This framework provides rapid development for writing the server part of the system. To make the communication of levels as simple as possible, the principle of injection through the Spring Framework, namely the Spring Boot and Spring Core modules, will be used. For services to work on different devices using the required versions of the libraries, the Gradle automatic assembly system was chosen. Also, thanks to the Gradle configuration, it is possible to break the structure of the server application into separate modules that will be responsible for separate levels of the architecture.

Frontend: To implement the client part, the TypeScript programming language and the React library were chosen. TypeScript will provide us with data typing, which will facilitate the scaling of this software system, and thanks to the React library, we will be able to create components that will be used in the implementation of the Flux architecture. The Flux architecture includes components such as Dispatcher, Store, React Views, and Action Creators.

The Flux architecture imposes restrictions on the flow of data, in particular, excluding the possibility of updating the state of components themselves. This approach makes the flow of data predictable and makes it easier to trace the causes of possible errors in the software.

Analysis of methos for creating a service-oriented software system in Azure using a software solution

After implementing the software system, we are going to start compare Azure services using the scales:

- cost and pricing;
- features and functionality;
- speed of deployment;
- support container registries;
- monitoring and logging support;

Cost and pricing

For calculating we used the official pricing calculator - https://azure.microsoft.com/en-us/pricing/calculator/ and calculated prices for 1 month. After researching we collected the main characteristics of each service and used the results to create table 1 with the cost and pricing for Azure services.

Table 1

Cost and pricing services

Service name	Azure Container Apps	Azure Kubernetes Services	Azure Red Hat OpenShift
vCPU	4	4	4
Memory	16 GB	16 GB	16 GB
Requests per month	60 million	-	-
Nodes	-	4	4
Temporary storage	-	150 GB	-
Master nodes	-	-	8 vCPU, 32 GB RAM
Price per month	771.04$	744.60$	2545.87$

Features and functionality

After researching we determined some features and functionality for Azure Container Apps, Azure Kubernetes Service and Azure Red Hat OpenShift and represented them in table 2.

Table 2

Features and functionality

	Service names		
Azure Container Apps	1.Docker image support 2. Auto-scaling 3.Integration with various Azure services 4.Network Configuration 5. Security protection	6. Monitoring and logging 7. Support for multiple programming languages 8. Scaling services 9. Deployment speed 10. Versioning	11.Support for different types of containers 12. Cluster Management 13. Backup and Restore 14.Support for different operating systems
Azure Kubernetes Services	1. Optimized Kubernetes Management 2. Auto-scaling 3. Docker image support 4. Integration with AzureMonitor 5. Integration with Azure Active Directory 6. Security protection 7. Integration with Azure DevOps 8. Support for multiple programming languages	9. Scaling services 10. Deployment speed 11. Versioning 12. Support for different types of containers 13. Cluster Management, 14. Backup and Restore 15. Automatic recovery 16. Remote access 17. Integration with Azure Policy	18. Flexibility 19. Cross-platform support20. Integration with Azure Arc 21. Integration with Azure Policy for Kubernetes 22. Network management 23. Integration with Azure Private Link 24. Support for different types of accounts
Azure Red Hat OpenShift	1. Kubernetes-based container orchestration	15. Automated testing and quality assurance	28. Kubernetes Operators for automated application management

	2. Red Hat Enterprise Linux operating system 3. Integration with Azure services 4. Enterprise-grade security 5. High availability and disaster recovery options 6. Scalability and elasticity 7. Monitoring and logging 8. Resource utilization tracking and optimization 9. Automated container builds and deployments 10. Multi-cluster management 11. Application templates and deployment patterns 12. Developer tools and SDKs 13. Integrated development environment integration 14. Application lifecycle management	16. Continuous integration and deliverypipeline 17. Git integration and version control 18. Application scaling and load balancing 19. Networking and service mesh capabilities 20. Role-based access control 21. Compliance and audit logging 22. Integration with external identity providers 23. Customizable policies and quotas 24. Resource tagging and management 25. Secure image registry and distribution 26. Integration with third-party registries 27. Application portability across hybrid cloud environments	29. Full-stack observability with Prometheus and Grafana 30. Distributed tracing with Jaeger 31. Logging with Elasticsearch, Fluentd, and Kibana 32. Integration with Azure DevOps for end-to-end software development 33. Container-native storage 34. Data persistence options 35. Serverless computing with Azure Functions 36. Artificial intelligence and machine learning services 37. Edge computing and IoT integration 38. Compatibility with Red Hat OpenShift ecosystem and marketplace 39. Flexible pricing and billing options 40. Enterprise-level support and service level agreements

Support container registries

After getting results we can represent table 3 for showing the number of container registries that Azure services support.

Table 3

Support container registries

Service name	Azure Container Apps	Azure Kubernetes Services	Azure Red Hat OpenShift
Docker Hub	+	+	+
Azure Container Registry	+	+	+
GitHub Container Registry	+	+	+
Amazon Elastic Container Registry	+	+	+
Google Cloud Registry	+	+	+
Harbor Registry	+	+	+
JFrog Container Registry	+	+	+
Quay.io	+	+	+
Red Hat Quay	+	+	+
IBM Cloud Container Registry	+	+	+
GitLab Container Registry	+	+	+
Artifactory		+	+
Quay Enterprise		+	+
VMware Harbor Registry		+	+
Oracle Cloud Infrastructure Registry			+
Azure Stack Hub Container Registry			+

Monitoring and logging support

After researching we determined monitoring and logging features for Azure Container Apps, Azure Kubernetes Service and Azure Red Hat OpenShift. Monitoring and logging features help to monitor and diagnose the state of the app to improve performance. Azure Monitor for monitoring and analyzing metrics, logs, and traces of distributed applications. Azure Log Analytics to collect, analyze and visualize logs from various sources in Azure, including container logs. Azure Application Insights for monitoring and analyzing the performance of applications running in containers. Kubernetes Dashboard for visualizing the status of a Kubernetes cluster and its components. Azure Log Analytics to collect, analyze and visualize logs from various sources in Azure, including container and Kubernetes cluster logs. Prometheus for collecting, monitoring, and analyzing Kubernetes and container metrics. Grafana for visualizing monitoring data from Prometheus and other sources. Kibana for visualizing logs collected using Elasticsearch and Logstash. Elasticsearch for storing and indexing logs from various sources. Fluent to collect and forward logs from containers to Elasticsearch or other storage. Jaeger for tracing distributed applications and identifying problems in the interaction between application components. OpenShift Console for visualizing the state of the cluster and its components. Istio for managing network traffic between application components and protecting against security threats between microservices. Other descriptions of these technologies you can read in the previous paragraphs.

After getting results for monitoring and logging we can represent table 4 for showing the number of monitoring and logging functionality that Azure services support.

Table 4

Monitoring and logging support

Servicename	Azure Container Apps	Azure Kubernetes Services	Azure Red Hat OpenShift
Azure Monitor	+	+	+
Azure Log Analytics	+	+	+
Azure Application Insights	+		
Kubernetes Dashboard		+	
Prometheus		+	+
Grafana		+	+
Kibana		+	+
Elasticsearch		+	+
Fluentd		+	+
Jaeger		+	+
OpenShift Console			+
Istio.OpenShift Console			+

Speed of deployment

For comparing speed of deployment we pushed our system to GitHub repositories to deploy them on the different services. For deploying we will use pipelines to deploy new images to Azure Container Apps, Azure Kubernetes Service and Azure Red Hat OpenShift. We will run 10 times pipelines for each services. In the case for Azure Red Hat OpenShift, we will use Argo CD to configure GitOps processors, one of which is the image deployment. We got average results for each service: Azure Container Apps is 80 sec, Azure Kubernetes Service is 207 sec and Azure Red Hat OpenShift is 220 sec.

Analysing results of deployments, we can create diagram to visualize the results as shown in Fig. 5.

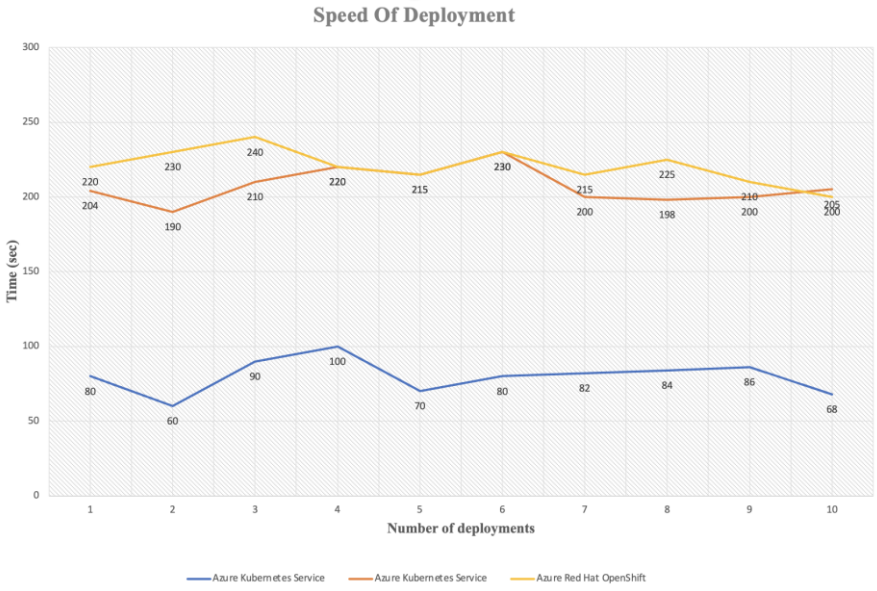

Fig. 5. Speed of deployments

After analyzing methods of creation, we can create a table with results as shown in table 5.

Table 5

Results of analyzing methods

Services/Scale	Cost and pricing ($)	Features and functionality (amount)	Speed of deployment (sec)	Support container registries (amount)	Monitoring and logging support (amount)
Azure Container Apps	771.04	14	80	11	3
Azure Kubernetes Service	744.60	24	204	14	9
Azure Red Hat OpenShift	2545.87	40	220	16	10

Conclusions

The article explored the study of methods for creating service-oriented software systems in Azure. Based on the results of the experiment, the following conclusions can be drawn. In terms of cost and pricing, Azure Container Apps is the most cost-effective option, while Azure Red Hat OpenShift is the most expensive. However, the cost difference is largely due to the advanced features and capabilities offered by Azure Red Hat OpenShift, which may be necessary for more complex applications or large-scale deployments.

In terms of features and functionality, Azure Red Hat OpenShift offers the most advanced set of features, including large-scale cluster support, advanced security support, and integrated CI/CD tooling. Azure Kubernetes Service offers a comprehensive set of features and functionality, while Azure Container Apps provides a simpler, more streamlined option for containerized application deployment.

In terms of support for container registries, all three services offer support for a wide range of container registries, with Azure Container Apps and Azure Kubernetes Service offering similar options and Azure Red Hat OpenShift offering a slightly wider range of options.

In terms of monitoring and logging support, Azure Kubernetes Service offers the widest range of tools, while Azure Red Hat OpenShift offers advanced security features and integrations.

In terms of speed of deployment, all three services offer fast and efficient deployment of containerized applications, with Azure Red Hat OpenShift offering the most advanced features for pipeline automation and integrated CI/CD tooling.

Ultimately, the best choice for a particular project will depend on the specific needs and requirements of the application being developed, as well as factors such as budget, development team experience, and existing infrastructure. After receiving and analyzing the results, we can see such prospects as choosing an Azure service for a certain budget and business needs for building a new system.

References

1. Jain K., Aggarwal P. A Systematic Review of Service-Oriented Architecture: Benefits, Challenges and Road Ahead // International Journal of Computer Applications, 2018. Vol. 182. No. 24. 31-40 с.
2. Sivaramakrishnan, K., & Gokulakrishnan, R. A survey on microservices architecture: Challenges and benefits. In: 2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI), 2019. 369-373 с.
3. Xu, J., Wang, Q., & Liu, D. Design and Implementation of a RESTful Web Service for Intelligent Agriculture. In: 2019 12th International Symposium on Computational Intelligence and Design (ISCID), 2019. 67-70 с.
4. Azure Container Apps. URL: https://azure.microsoft.com/en-us/products/container-apps.(дата звернення: 10.02.2023)
5. Azure Kubernetes Service. URL: https://azure.microsoft.com/en-us/products/kubernetes-service.(дата звернення: 10.02.2023)
6. Azure Red Hat OpenShift. URL: https://azure.microsoft.com/en-us/products/openshift.(дата звернення: 10.02.2023)
7. Лукса М. Kubernetes в дії / пер. з англ. А. Мартиненко. К.: Видавництво Маннінг, 2018. 372 с.
8. CI/CD for containers / URL: https://learn.microsoft.com/en-us/azure/architecture/solution-ideas/articles/cicd-for-containers. (дата звернення: 15.02.2023)
9. Liu, J., Li, L., Ma, J., & Wang, J. Docker-Based Microservice Architecture of Financial Logistics Management System. In: Proceedings of the 2nd International Conference on Modern Educational Technology and Management Science. Atlantis Press, 2021. 299-304 с.
10. Тоттен Р. Основи Nginx: крок за кроком до володіння базовими можливостями Nginx в реальних додатках / пер. з англ. К. Батигіна. - М.: Видавництво "Пакт", 2019.194 с.
11. Thomas, S. M., & Riggs, S. PostgreSQL 12 High Availability Cookbook: Over 100 recipes to design a highly available server with the advanced features of PostgreSQL. Packt Publishing, 2020. 462 с.

Oleksii Makieiev **Олексій Макєєв**	Bachelor, Kharkiv National University of Radio Electronics, e-mail: oleksii.makieiev@nure.ua https://orcid.org/0009-0002-7909-4793	бакалавр, Харківський національний університет радіоелектроніки
Natalia Kravets **Наталя Кравець**	Ph.D, Associate Professor of Department of Software Engineering, Kharkiv National University of Radio Electronics, e-mail: natalia.kravets@nure.ua https://orcid.org/0000-0002-6753-3333, ResearcherID: B-7312-2019	кандидат технічних наук, доцент кафедри програмної інженерії, Харківський національний університет радіоелектроніки