

Design and Implementation of a Secure, Scalable and Observable Cloud-Native DevOps Architecture for Microservices Deployment on Microsoft Azure

Nader Amri

elamrineder100@gmail.com

Higher Institute of Computer Science of Kef

Helmi Jbili

Higher Institute of Computer Science of Kef

Sofien Mhatli

Higher Institute of Computer Science of Kef

Shiraz Ouertani

Higher Institute of Computer Science of Kef

Adel Aldalbahi

King Faisal University

Article

Keywords: Cloud computing, DevOps, microservices, Azure Kubernetes Service, Terraform, Docker, CI/CD, Kubernetes, observability, cloud security

Posted Date: October 27th, 2026

DOI: <https://doi.org/10.21203/rs.3.rs-11489182/v1>

License:  This work is licensed under a Creative Commons Attribution 4.0 International License.

[Read Full License](#)

Additional Declarations: No competing interests reported.

Design and Implementation of a Secure, Scalable and Observable Cloud-Native DevOps Architecture for Microservices Deployment on Microsoft Azure

Microservices Application Development and Cloud Infrastructure Deployment

Nader Amri^{1,*}, Helmi Jbili¹, Dr Sofien Mhatli¹, Shiraz Ouertani¹, and Adel Aldalbahi²

¹Higher Institute of Computer Science of Kef, Le Kef, Tunisia

²Department of Electrical Engineering, King Faisal University, Ahsa, Saudi Arabia

*elamrineder100@gmail.com

Helmi Jbili contributed as Microservices Application Developer. Dr Sofien Mhatli supervised the work as Professional Supervisor, Cloud and Cybersecurity Expert.

Contributor: Shiraz Ouertani, Enseignante Universitaire | IT Expert. Role: Professional Co-Supervisor. Certifications: Azure Administrator Associate · CCNA Security · ANSSI · SFC™. Areas of expertise: Cloud/Azure · Cybersecurity · DevOps · Cloud-Native · Monitoring.

Corresponding author: Nader Amri (elamrineder100@gmail.com).

E-mail addresses: Nader Amri, elamrineder100@gmail.com; Helmi Jbili, helmijbili.ai@gmail.com; Dr Sofien Mhatli, sofiene.mehtli@isikef.u-jendouba.tn; Adel Aldalbahi, aaldalbahi@kfu.edu.sa; Shiraz Ouertani, shiraz.wertany@gmail.com.

This work was supported by the Scientific Research Deanship of the Higher Institute of Computer Science of Kef, Tunisia, under Amri Nader end of study project. The project earned the highest distinction during the final project defense.

ABSTRACT

Cloud-native architectures have become a practical response to the limitations of manually deployed monolithic applications, particularly when scalability, availability, security, and deployment reproducibility are required. This paper presents the design and implementation of an automated cloud deployment architecture for a salon reservation microservices application. The proposed architecture combines Docker-based containerization, Terraform-based Infrastructure as Code, Azure Kubernetes Service orchestration, PostgreSQL Flexible Server, Azure DevOps CI/CD pipelines, TLS-secured ingress exposure, horizontal pod autoscaling, and observability through Azure Monitor, Container Insights, Prometheus, and Grafana. The work transforms an initially manual and weakly automated deployment process into a reproducible cloud-native workflow. Functional validation, fault-tolerance testing, scalability testing, alerting verification, and load-oriented ingress rate-limiting tests demonstrate the operational feasibility of the proposed solution. The results show that combining Infrastructure as Code, Kubernetes orchestration, CI/CD automation, and monitoring provides a reliable foundation for deploying and operating microservices in a secure and scalable cloud environment.

Index Terms—Cloud computing, DevOps, microservices, Azure Kubernetes Service, Terraform, Docker, CI/CD, Kubernetes, observability, cloud security.

Introduction

Modern software systems increasingly require rapid delivery, high availability, elastic scalability, and secure operation. Traditional deployment practices, especially those based on manual infrastructure provisioning and manual application releases, often introduce operational delays, configuration inconsistencies, and a higher probability of human error. These limitations become more visible when applications evolve toward distributed microservices, where each service may have its own runtime, database, deployment lifecycle, and operational requirements.

In this context, cloud-native DevOps practices provide a structured approach for automating the complete lifecycle of an application. Containerization improves portability and reproducibility, Infrastructure as Code (IaC) automates infrastructure provisioning, Kubernetes enables orchestration and self-healing, CI/CD pipelines automate software delivery, and monitoring tools improve visibility into the runtime state of the platform.

This paper presents a complete cloud-native deployment architecture implemented for a salon reservation application composed of multiple microservices. The project was conducted in the context of YohaTech, a Tunisian technology company working in cloud, DevOps, cybersecurity, and digital transformation¹. The main objective is to design and validate an automated, secure, observable, and scalable architecture capable of deploying microservices reliably on Microsoft Azure.

The main contributions of this work are as follows:

- the design of a layered cloud architecture for a microservices application deployed on Azure Kubernetes Service (AKS);
- the implementation of infrastructure automation using Terraform;
- the containerization and Kubernetes deployment of application services;
- the integration of an Azure DevOps CI/CD pipeline for automated image build, registry push, and redeployment;
- the integration of security, autoscaling, observability, and alerting mechanisms;
- the validation of the architecture through functional, resilience, scalability, alerting, and load-oriented tests.

1 Related Work and Technological Background

1.1 Microservices and Cloud-Native Architectures

Microservices decompose an application into a set of independently deployable services. Compared with monolithic systems, this architectural style supports independent scaling, improved maintainability, and service-level deployment autonomy. However, it also introduces operational complexity because networking, configuration, service discovery, persistence, and monitoring must be managed consistently across services.

Cloud-native architectures address these challenges by combining containers, orchestration platforms, automated delivery pipelines, and observability mechanisms². Kubernetes has become a widely adopted orchestration platform because it provides declarative deployment, self-healing, service abstraction, and scaling mechanisms. Managed Kubernetes services, such as AKS, reduce the operational burden of maintaining the control plane while preserving Kubernetes primitives.

1.2 DevOps, CI/CD, and Infrastructure as Code

DevOps aims to improve collaboration between development and operations while automating the software delivery lifecycle³. CI/CD pipelines reduce manual deployment effort by automating build, test, image publication, and deployment stages. In parallel, Infrastructure as Code tools such as Terraform allow infrastructure resources to be described declaratively and provisioned reproducibly. This approach improves traceability, reduces configuration drift, and supports repeatable environment creation. A comparative study of deployment alternatives guided the final selection of Azure, AKS, Terraform, and Azure DevOps because this stack provides managed orchestration, native CI/CD integration, reproducible provisioning, and direct compatibility with the project environment.

1.3 Security and Observability in Cloud Deployments

Security in cloud-native deployments requires layered controls, including encrypted communications, secret management, network segmentation, and controlled ingress exposure. In Kubernetes environments, TLS certificates, Kubernetes Secrets, ingress controllers, and rate-limiting policies contribute to reducing exposure and mitigating abuse. Observability complements security and reliability by enabling metric collection, visualization, alerting, and incident detection⁴. Tools such as Azure Monitor and Container Insights provide visibility into cluster and application behavior.

2 Research Methodology

Following the structure of the reference paper template, this study adopts an applied engineering research methodology. The methodology is organized around problem analysis, architecture design, implementation, and validation.

2.1 Problem Identification

The initial environment relied on partially manual deployment and infrastructure configuration practices. The main limitations identified were:

- manual infrastructure deployment without IaC;
- lack of standardized cloud resource configuration;
- absence of a CI/CD pipeline for build and deployment automation;
- limited scalability and lack of orchestration;
- insufficient observability and alerting mechanisms;

- insufficiently optimized security and isolation mechanisms.

2.2 Architecture Design

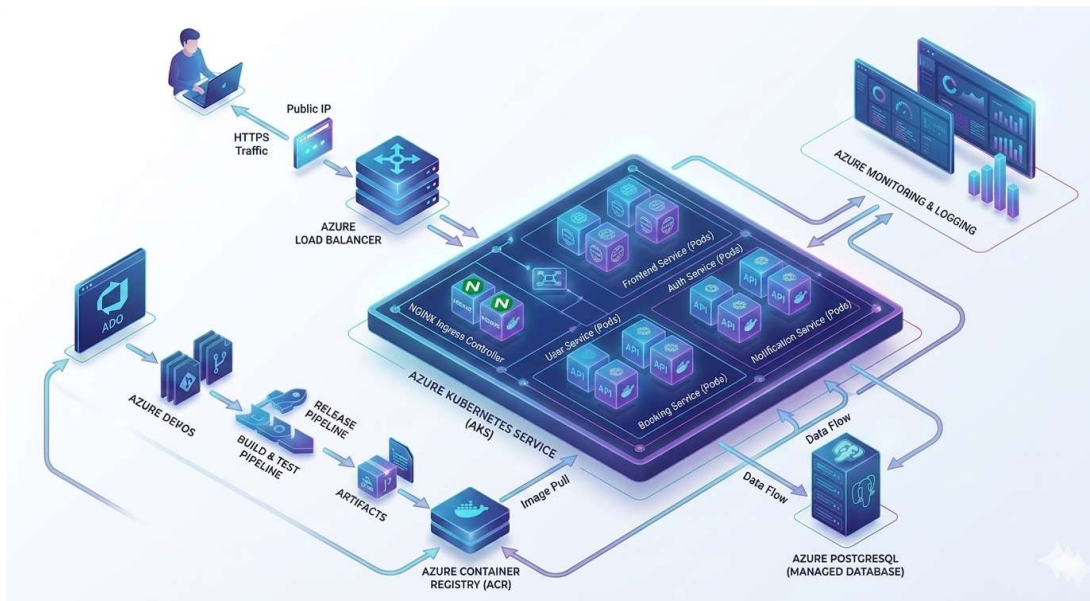


Figure 1 Global architecture of the proposed cloud-native solution.

The proposed solution was designed as a layered cloud-native architecture. The application services are containerized using Docker and deployed on AKS. Azure Container Registry (ACR) stores Docker container images. Terraform provisions the cloud infrastructure, including the resource group, networking components, AKS cluster, ACR, PostgreSQL Flexible Server, and monitoring-related resources. PostgreSQL Flexible Server provides managed relational storage. Azure DevOps automates the CI/CD workflow. Ingress-NGINX exposes the application and routes HTTP/HTTPS traffic to the appropriate services. TLS certificates secure external communications. Observability is implemented using Container Insights, Azure Monitor, Prometheus, and Grafana.

2.3 Implementation Workflow

The implementation workflow consisted of the following steps:

- 1) preparation and local validation of the microservices application;
- 2) containerization of backend services and frontend components;
- 3) definition and provisioning of Azure infrastructure using Terraform;
- 4) installation and configuration of Azure CLI and kubectl;
- 5) Kubernetes deployment using YAML manifests for namespaces, secrets, deployments, services, ingress, HPA, certificates, and issuers;
- 6) configuration of PostgreSQL Flexible Server, schemas, and initial data;
- 7) creation of an Azure DevOps pipeline connected to the GitHub repository;
- 8) activation of monitoring, metrics visualization, and alert rules;
- 9) validation through operational tests.

3 Proposed Cloud-Native Architecture

The proposed solution is organized as a complete cloud-native deployment chain that starts from the AuraCut microservices application and ends with an observable, scalable, and secure runtime environment on Microsoft Azure. The solution separates the application layer, infrastructure layer, containerization and orchestration layer, CI/CD layer, and security and observability

layer. This layered organization clarifies the role of each technology: Docker packages the services, Terraform provisions Azure resources, AKS runs and scales the workloads⁵, Azure DevOps automates delivery, PostgreSQL stores application data, and monitoring tools provide operational visibility.

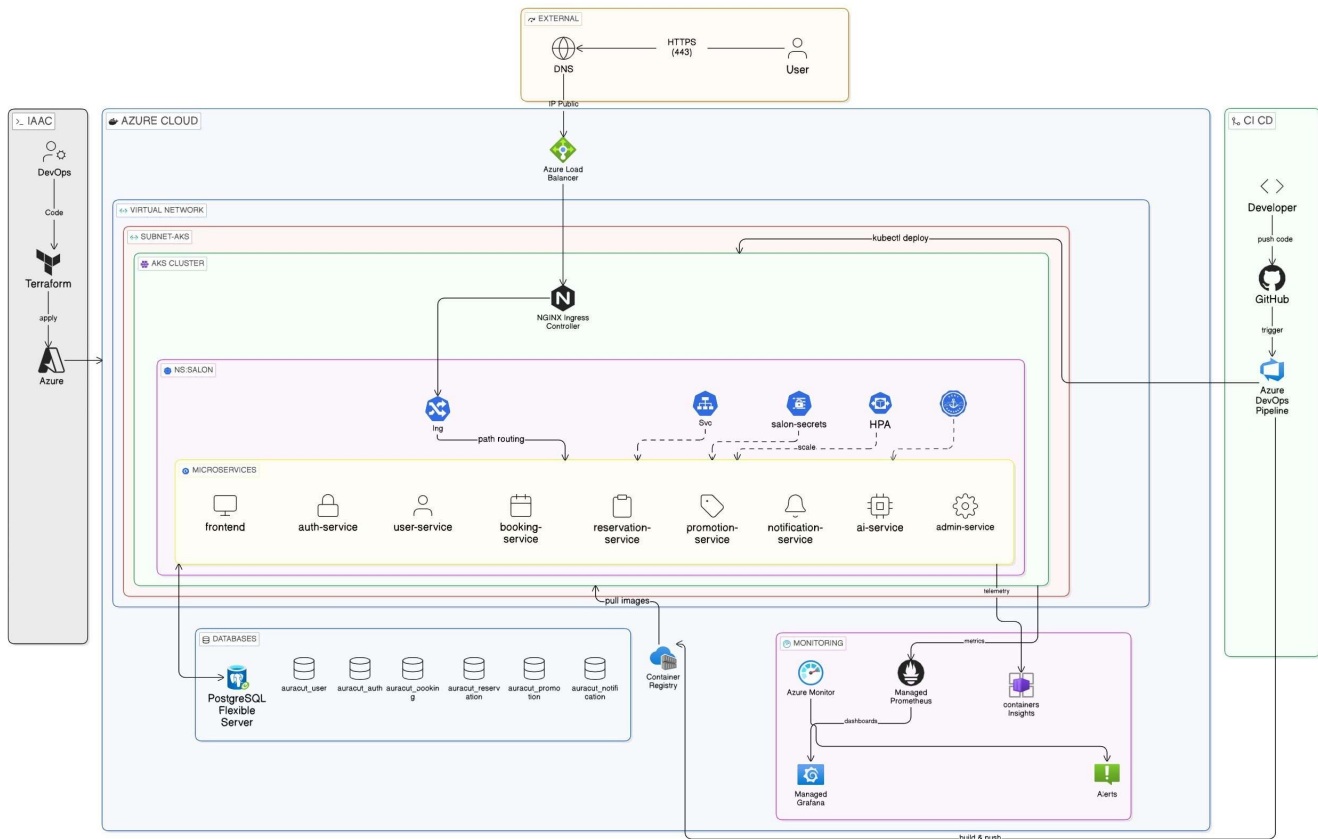


Figure 2 Final architecture view used for the cloud deployment design.

3.1 Application Layer

The target application is AuraCut, a salon reservation platform designed to manage authentication, users, bookings, reservations, notifications, promotions, administration, and intelligent assistant features. Before focusing on the cloud infrastructure, the application itself was developed and validated as a functional microservices platform. The frontend provides the user interface and communicates with backend services through REST APIs. Each microservice is designed to be independently deployable, and each service may manage its own data.

The application is organized around several functional services. The authentication service manages identity and access logic, the user service manages user-related operations, the booking and reservation services support salon appointment workflows, and complementary services support notifications, promotions, administration, and assistant-oriented features. This separation makes it possible to build, deploy, scale, and monitor services independently instead of redeploying the whole application as a monolithic system.

Before presenting the cloud-native deployment solution, it is necessary to clarify the internal application architecture. AuraCut follows a microservices-oriented design in which the frontend consumes backend APIs through a gateway or routed service endpoints. Each backend service encapsulates a specific business capability and communicates with the persistence layer according to its functional responsibility. This architecture improves maintainability because changes in authentication, booking, users, notifications, or administration can be implemented and deployed with limited impact on the other services. It also

prepares the application for containerization, Kubernetes deployment, independent scaling, and service-level monitoring.

The microservices application development work was carried out by Helmi Jbili. This phase included the design and implementation of the application services, the preparation of backend APIs, the connection between the frontend and backend components, and the validation of the functional behavior before cloud deployment. This application-development stage is important because the cloud-native pipeline described in this paper depends on having services that can be packaged, versioned, deployed, and monitored independently.

Aura Cut — Microservices Architecture

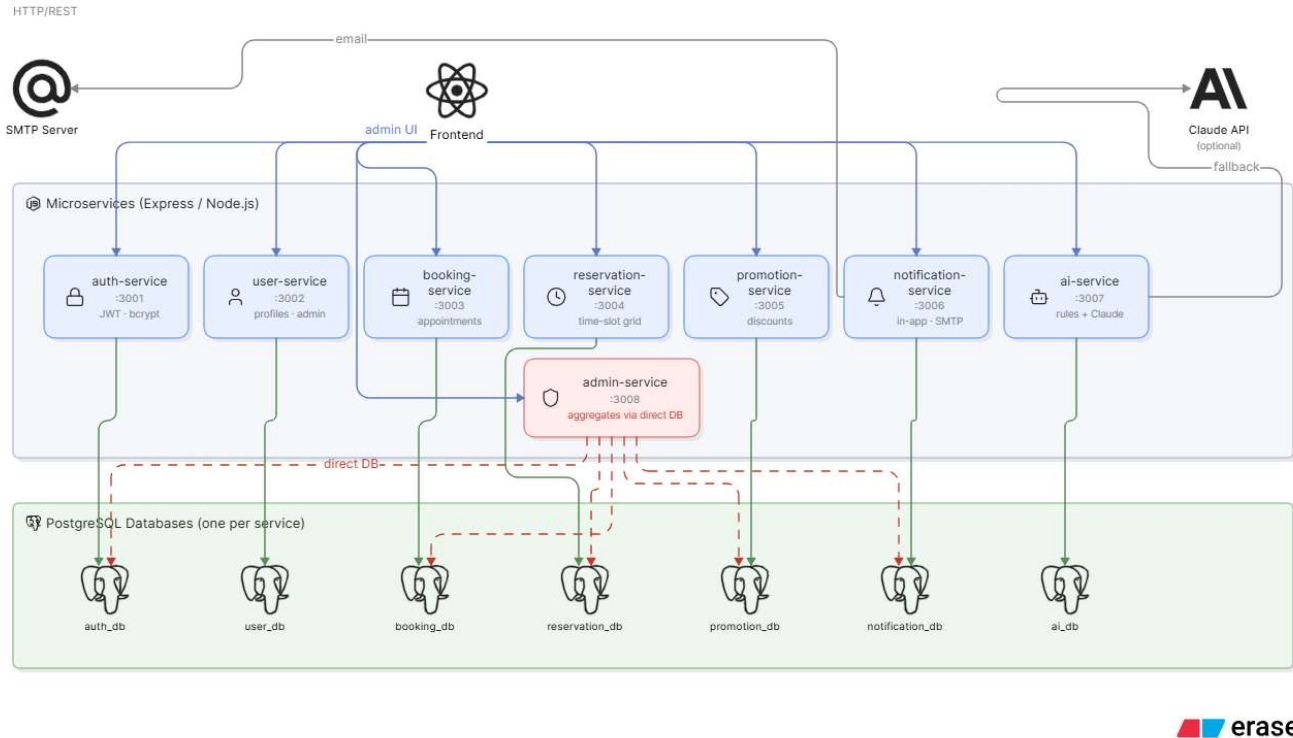


Figure 3 Microservices architecture of the AuraCut application before cloud-native deployment.

3.2 Infrastructure Layer

The infrastructure layer is provisioned with Terraform⁶. The project defines configuration files such as `main.tf`, `variables.tf`, `terraform.tfvars`, `outputs.tf`, `providers.tf`, and `versions.tf`. Terraform enables reproducible creation and update of cloud resources, reducing manual configuration errors and improving deployment traceability.

The Terraform workflow describes the infrastructure lifecycle. Configuration files are written and validated, the execution plan is reviewed, and the infrastructure is applied to Azure. This process provisions the resource group, networking layer, AKS cluster, ACR, PostgreSQL Flexible Server, and monitoring-related resources in a controlled and repeatable manner⁷.

3.3 Containerization and Orchestration Layer

Docker is used to package the services into portable images⁸. The backend services are based on Node.js, while the frontend is served through NGINX.

Kubernetes manifests define the runtime state of the services in AKS⁹. The deployment includes namespaces, secrets, deployment objects, services, ingress routing, HPA resources, and certificate-related resources.

The containerization phase converts each microservice into a Docker image that can be executed consistently across local and cloud environments. Docker Desktop was used to verify that the containers were correctly built and running before pushing images to Azure Container Registry and deploying them on AKS.

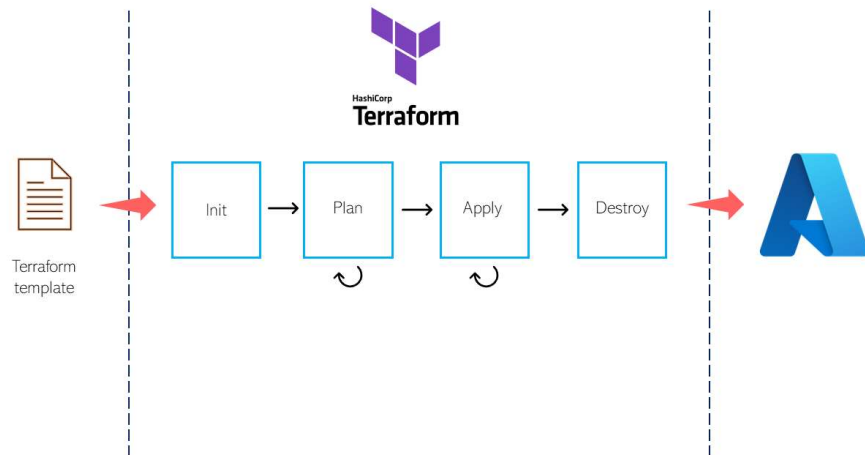


Figure 4 Terraform workflow for reproducible infrastructure provisioning on Microsoft Azure.

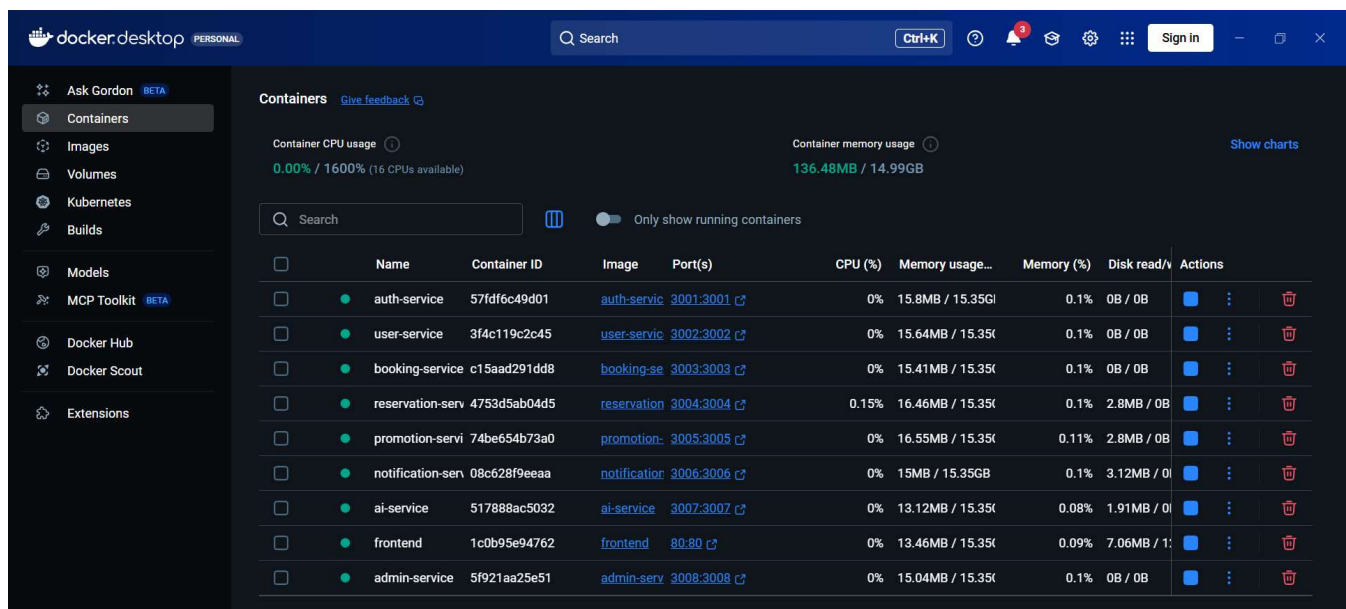


Figure 5 Docker Desktop verification showing containerized microservices running locally.

The containerization workflow packages each backend service and the frontend into Docker images. The images are first verified locally using Docker Desktop, then published to ACR and consumed by Kubernetes manifests. This workflow is essential for ensuring that the same application artifact can be tested locally and deployed consistently in the cloud.

3.4 CI/CD Layer

The CI/CD layer is implemented using Azure DevOps and Azure Pipelines¹⁰. The pipeline is connected to the GitHub repository and uses a YAML pipeline file. Its role is to automate image construction, push images to ACR, and trigger Kubernetes redeployment. This reduces manual release effort and improves consistency between development and deployment workflows.

The CI/CD workflow starts when a change is pushed to the GitHub repository. Azure DevOps then triggers the pipeline, builds the container image, pushes it to Azure Container Registry, and refreshes the Kubernetes deployment in AKS. This workflow supports repeatable delivery and reduces the operational risk associated with manual deployment commands¹¹. Continuous deployment concepts also support the automated promotion of changes after successful validation¹².

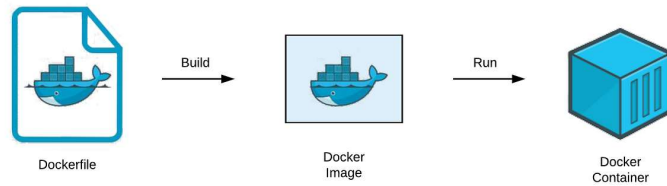


Figure 6 Containerization workflow used to package microservices into portable Docker images.

```

Dockerfile X
auth-service > Dockerfile > FROM
1 FROM node:20-alpine
2 WORKDIR /app
3 COPY package*.json ./
4 RUN npm ci --omit=dev
5 COPY . .
6 RUN addgroup -S app && adduser -S app -G app
7 USER app
8 EXPOSE 3001
9 CMD ["node", "server.js"]
10
  
```

Figure 7 Dockerfile used to package the authentication microservice.

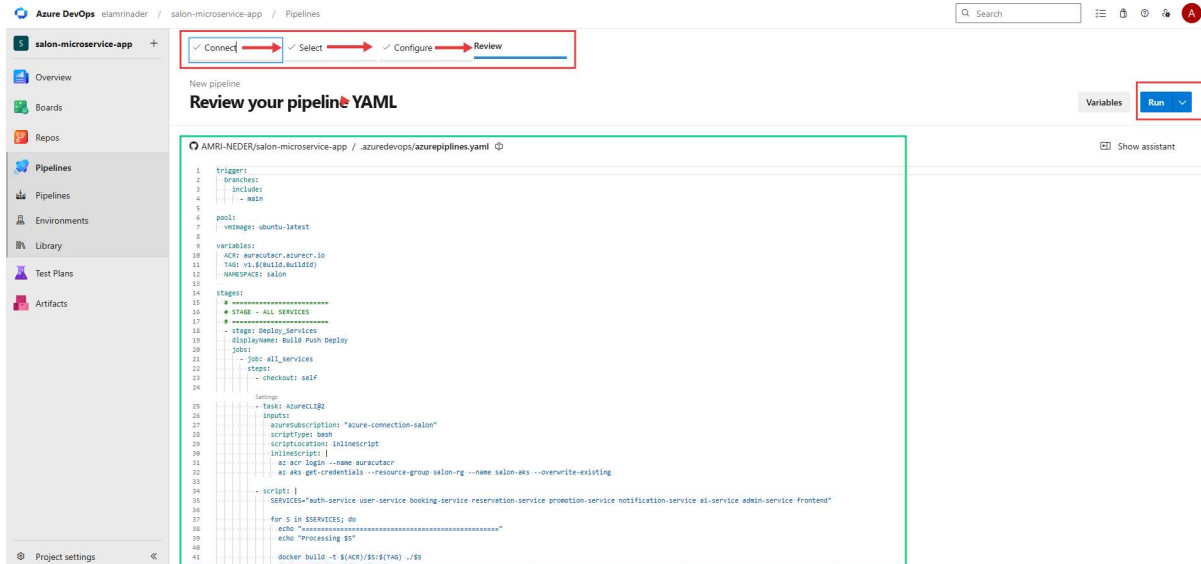


Figure 8 Manual execution of the Azure DevOps pipeline during validation.

3.5 Security and Observability Layer

Security mechanisms include TLS-based HTTPS access, Kubernetes Secrets, certificate management, and ingress rate-limiting. Observability is implemented through Container Insights, Prometheus, Grafana, and alert rules. Azure Monitor supports cloud-level monitoring¹³, while Container Insights provides container-oriented cluster visibility¹⁴. These tools enable monitoring of namespace-level and pod-level metrics, as well as alerting when abnormal restart counts or high CPU/memory consumption are detected.

The security configuration was validated at both configuration and runtime levels. Sensitive connection parameters were externalized from the application code and injected through Kubernetes Secrets, while TLS resources were used to secure public

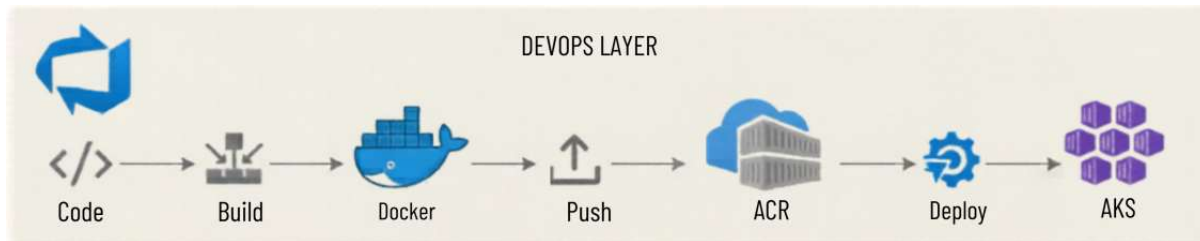


Figure 9 CI/CD workflow from source-code update to image build, ACR push, and deployment on AKS.

access. This design reduces direct exposure of credentials in manifests and supports a clearer separation between application logic, deployment configuration, and operational security controls.

4 Implementation and Experimental Validation

4.1 Infrastructure Provisioning

Terraform was initialized, validated, planned, and applied to deploy the infrastructure. The Terraform configuration describes the cloud resources required by the solution, including the Azure resource group, AKS cluster, container registry, networking components, and database-related services. This Infrastructure as Code approach makes the infrastructure reproducible, auditable, and easier to redeploy than a manual configuration process.

The infrastructure validation step confirmed that Terraform produced the expected resources and outputs. The execution plan was reviewed before deployment, and the apply phase was used to create the target infrastructure on Azure. This step demonstrates the feasibility of reproducible infrastructure deployment and provides the foundation on which the Kubernetes application layer is installed.

```
PS C:\Users\amrin\OneDrive\Bureau\PFE\salon-microservice-app\terraform> terraform plan
Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the
following symbols:
+ create

Terraform will perform the following actions:

# azurerm_container_registry.acr will be created
+ resource "azurerm_container_registry" "acr" {
+   admin_enabled           = false
+   admin_password         = (sensitive value)
+   admin_username         = (known after apply)
+   data_endpoint_host_names = (known after apply)
+   encryption             = (known after apply)
+   export_policy_enabled   = true
+   id                     = (known after apply)
+   location               = "francecentral"
+   login_server            = (known after apply)
+   name                   = "auracutacr"
+   network_rule_bypass_option = "AzureServices"
+   network_rule_set        = (known after apply)
+   public_network_access_enabled = true
+   resource_group_name     = "salon-rg"
+   sku                    = "Basic"
+   tags                    = {
+     "environment" = "dev"
+     "managed_by"  = "terraform"
+     "project"     = "salon"
+   }
+   trust_policy_enabled     = false
+   zone_redundancy_enabled = false
+ }

# azurerm_kubernetes_cluster.aks will be created
+ resource "azurerm_kubernetes_cluster" "aks" {
```

Figure 10 Terraform execution plan used to preview the Azure infrastructure deployment.

4.2 Kubernetes Deployment

After infrastructure deployment, the AKS environment was configured using Azure CLI and kubectl. Kubernetes resources were applied to deploy the application services, expose them internally, and route external traffic through ingress. DNS association was performed using a DuckDNS domain, and the application was validated first through HTTP and then through HTTPS.

```

PS C:\Users\amrin\OneDrive\Bureau\PFE\salon-microservice-app\terraform> terraform apply

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:
+ create

Terraform will perform the following actions:

# azurerm_container_registry.acr will be created
+ resource "azurerm_container_registry" "acr" {
  + admin_enabled           = false
  + admin_password          = (sensitive value)
  + admin_username          = (known after apply)
  + data_endpoint_host_names = (known after apply)
  + encryption              = (known after apply)
  + export_policy_enabled   = true
  + id                     = (known after apply)
  + location                 = "francecentral"
  + login_server             = (known after apply)
  + name                     = "auracutacr"
  + network_rule_bypass_option = "AzureServices"
  + network_rule_set         = (known after apply)
  + public_network_access_enabled = true
  + resource_group_name      = "salon-rg"
  + sku                      = "Basic"
  + tags                     = {
    + "environment" = "dev"
    + "managed_by"  = "terraform"
    + "project"     = "salon"
  }
  + trust_policy_enabled     = false
  + zone_redundancy_enabled = false
}

# azurerm_kubernetes_cluster.aks will be created

```

Figure 11 Terraform infrastructure deployment using the apply command.

The HTTPS exposure relied on cert-manager resources, including Issuer and Certificate manifests, while ingress annotations supported TLS routing and request rate limiting.

The deployment scenario followed a progressive validation approach. First, the cluster connection and namespace resources were verified with kubectl. Second, deployment and service manifests were applied for the microservices. Third, ingress routing and HTTPS certificates were checked to confirm that external traffic reached the correct service endpoints.

```

PS C:\Users\amrin\OneDrive\Bureau\PFE\salon-microservice-app> kubectl get pods -n salon

```

NAME	READY	STATUS	RESTARTS	AGE
admin-service-574d884567-nnggv	1/1	Running	0	23s
ai-service-65fc9657df-sk6lj	1/1	Running	0	23s
auth-service-8b578c4c9-njvt8	1/1	Running	0	23s
booking-service-5659957c4c-8zs5t	1/1	Running	0	23s
frontend-7964d78c85-hj7dv	1/1	Running	0	22s
notification-service-7649b99fc6-qrvqk	1/1	Running	0	22s
promotion-service-d9cf8b767-562jt	1/1	Running	0	22s
reservation-service-d998d9484-k9vkb	1/1	Running	0	21s
user-service-8c664db6b-cqh4q	1/1	Running	0	21s

Figure 12 Kubernetes deployment verification after applying service manifests.

4.3 Database Configuration

PostgreSQL Flexible Server was configured to host the application databases. Schemas and seed data were applied to initialize the microservice databases, and table creation and inserted records were verified after initialization. The pgcrypto extension was enabled to support cryptographic operations at the database level.

The database of the authentication service was used as an example to validate the database initialization workflow. This example shows how the initial database was created in PostgreSQL, how the first tables were generated, and how initial data was inserted before connecting the service to the Kubernetes deployment. The same principle can be reused by the other microservices that require persistence.

The database scenario was validated by creating the required schema, checking the generated tables, and inspecting representative inserted data. This step ensured that the deployed services had a consistent persistence layer before functional and scalability tests were executed. It also confirmed that the migration and initialization work prepared during development could be reproduced in the managed PostgreSQL environment.

List of databases									
Name	Owner	Encoding	Locale Provider	Collate	Ctype	ICU Locale	ICU Rules	Access privileges	
auracut_auth	Nader	UTF8	libc	en_US.utf8	en_US.utf8				
auracut_booking	Nader	UTF8	libc	en_US.utf8	en_US.utf8				
auracut_notification	Nader	UTF8	libc	en_US.utf8	en_US.utf8				
auracut_promotion	Nader	UTF8	libc	en_US.utf8	en_US.utf8				
auracut_reservations	Nader	UTF8	libc	en_US.utf8	en_US.utf8				
auracut_user	Nader	UTF8	libc	en_US.utf8	en_US.utf8				
azure_maintenance	azuresu	UTF8	libc	en_US.utf8	en_US.utf8				
azure_sys	azuresu	UTF8	libc	en_US.utf8	en_US.utf8				
postgres	azure_pg_admin	UTF8	libc	en_US.utf8	en_US.utf8				
template0	azure_pg_admin	UTF8	libc	en_US.utf8	en_US.utf8			=c/azure_pg_admin +	
								azure_pg_admin=CTc/azure_pg_admin	
template1	azure_pg_admin	UTF8	libc	en_US.utf8	en_US.utf8			=c/azure_pg_admin +	
								azure_pg_admin=CTc/azure_pg_admin	

Figure 13 PostgreSQL database created successfully.

```

auracut_booking=> \c auracut_auth
psql (16.12, server 14.22)
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, compression: off)
You are now connected to database "auracut_auth" as user "Nader".
auracut_auth=> CREATE TABLE IF NOT EXISTS users (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  name VARCHAR(120) NOT NULL,
  email VARCHAR(200) NOT NULL UNIQUE,
  password VARCHAR(255) NOT NULL,
  role VARCHAR(20) NOT NULL DEFAULT 'user'
  CHECK (role IN ('user','admin')),
  created_at TIMESTAMPTZ NOT NULL DEFAULT NOW()
);

CREATE INDEX IF NOT EXISTS idx_users_email ON users(email);
CREATE INDEX IF NOT EXISTS idx_users_role ON users(role);
CREATE TABLE
CREATE INDEX
CREATE INDEX

```

Figure 14 Schema initialization step for the managed PostgreSQL database.

```

auracut_auth=> \dt
List of relations
Schema | Name | Type | Owner
-----+-----+-----+-----
public | users | table | Nader
(1 row)

```

Figure 15 Database table verification for the AuraCut authentication service.

```

auracut_auth=> SELECT * FROM users;

```

id	name	email	password	role	created_at
c2adf3a7-a2c8-48bd-b75d-c0dfa9d37445	Admin	elamrineder100@gmail.com	\$2b\$10\$scMif8MqCa5zXP.h4bjuw.78D6K1ztbxZCC9MUasJsfpuF10p4HiUO	admin	2026-05-10 02:33:47.090713+00

Figure 16 Verification of inserted authentication-service data after initialization.

4.4 CI/CD Validation

The Azure DevOps pipeline was connected to the GitHub repository and executed to validate the automated delivery chain. A commit was pushed to trigger a new pipeline execution, and the creation of updated pods in the cluster confirmed that the redeployment workflow was operational.

In the validation scenario, the repository acted as the single entry point for delivery. A source-code change triggered the Azure DevOps pipeline, which built the image, pushed it to ACR, and refreshed the Kubernetes workload. The observed pod update confirmed that the deployment state in AKS followed the pipeline execution rather than a manual redeployment step.

4.5 Monitoring and Alerting Validation

Container Insights provided an initial view of the AKS cluster. Prometheus and Grafana were then enabled for more detailed metrics visualization. The project included dashboards for namespace and pod-level monitoring. Alerting rules were configured to detect abnormal pod restarts and high CPU or memory usage.

The monitoring scenario was implemented in three stages. First, Container Insights was enabled to obtain a managed view of cluster nodes, namespaces, pods, CPU, memory, and restart indicators. Second, Prometheus was activated as a metrics source in

order to collect Kubernetes and application-level measurements with finer operational detail. Third, Grafana was connected to the Prometheus data source and used to visualize namespace-level and service-level dashboards. This separation is useful because Azure Monitor gives an integrated cloud-provider view, while Prometheus and Grafana provide flexible dashboards for operational investigation.

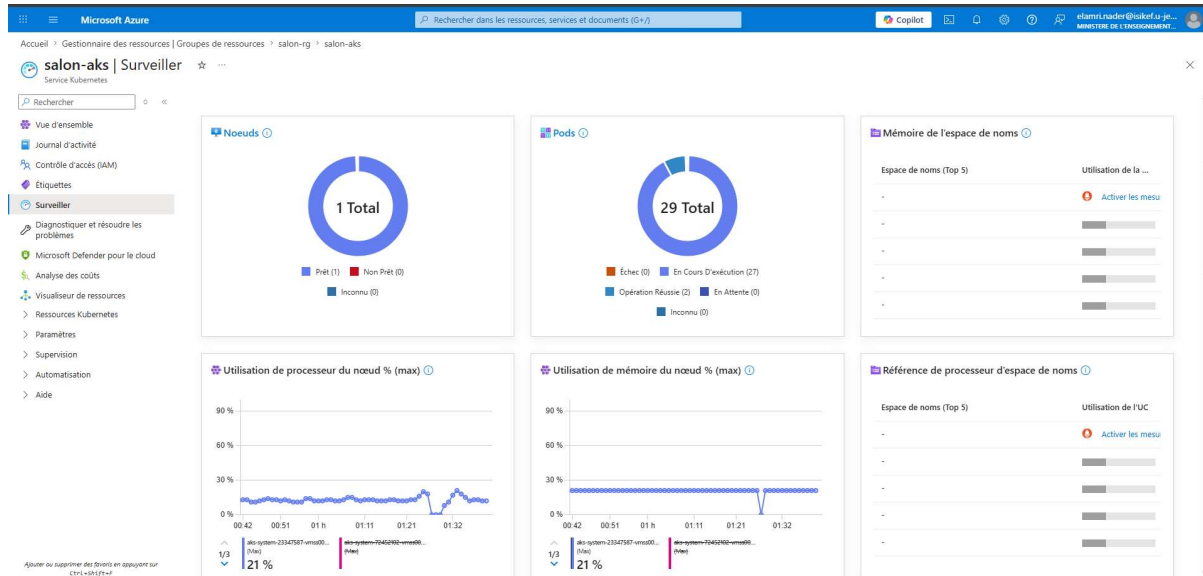


Figure 17 Container Insights view used for managed AKS monitoring.

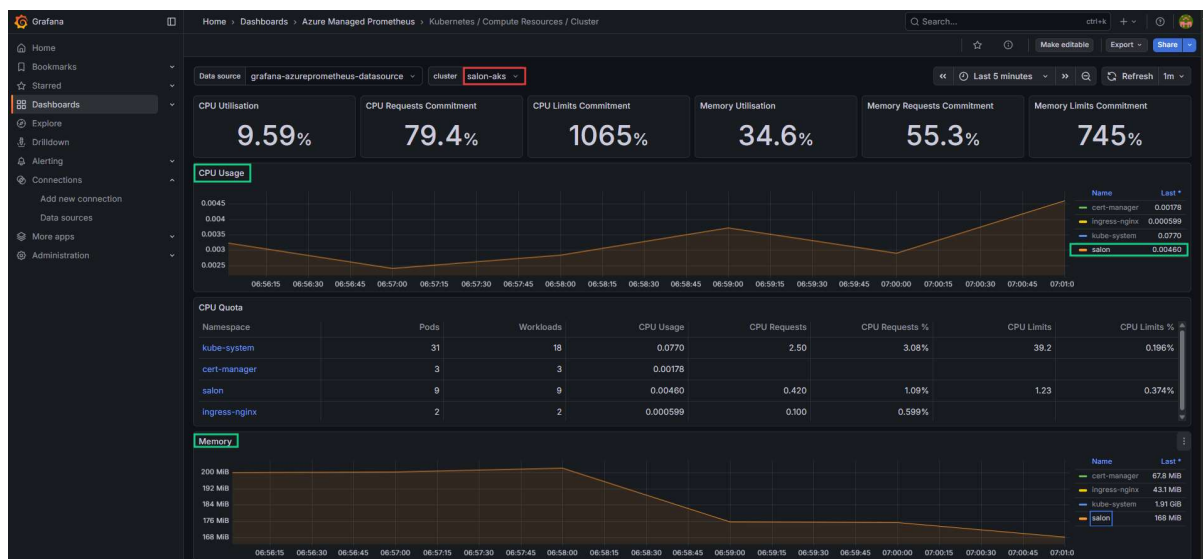


Figure 18 Grafana visualization of namespace-level metrics.

Alerting was configured after the dashboard validation. The alert rule creation step defined the monitored condition, the evaluation behavior, and the notification target. A controlled restart scenario was then used to trigger the alert path and verify that the administrator received the expected notification. This scenario validates not only metric collection but also the complete detection and communication chain required during an operational incident.

4.6 Operational Tests

The architecture was validated through several operational tests:

- **Functional access test:** the application was accessed through the configured domain to verify service availability.

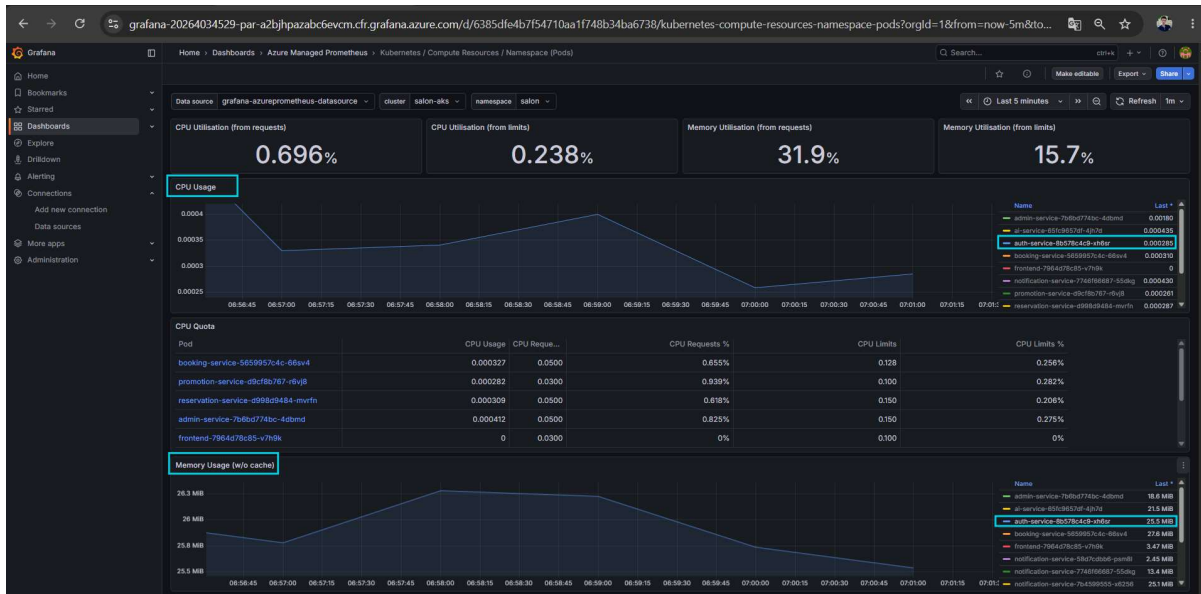


Figure 19 Grafana visualization of service-level metrics.

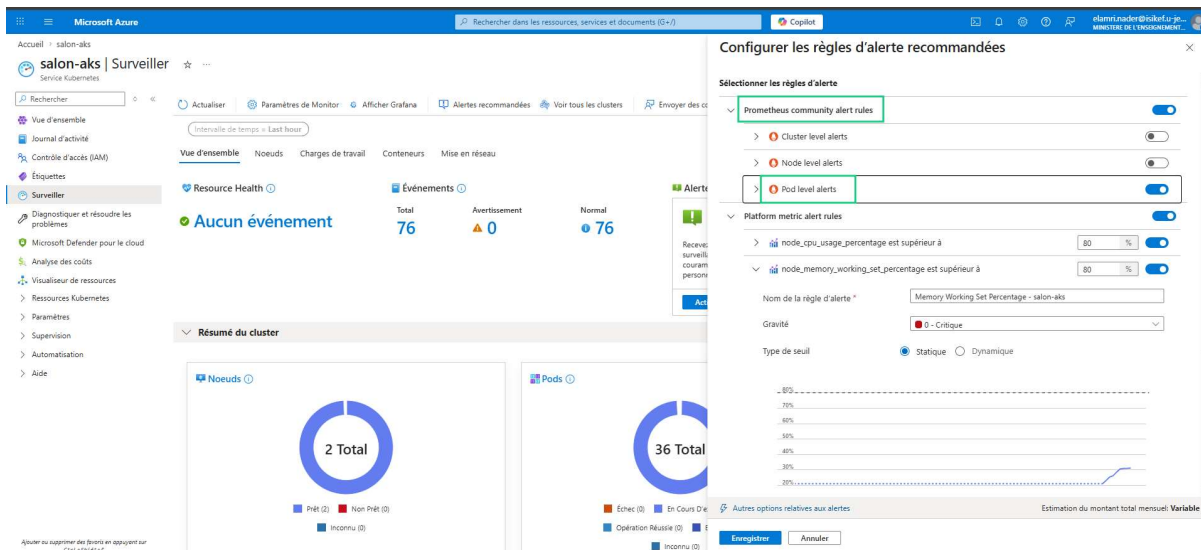


Figure 20 Creation of an alert rule for abnormal runtime conditions.

```
PS C:\Users\amrin\OneDrive\Bureau\PFE\salon-microservice-app> kubectl rollout restart deployment admin-service -n salon
deployment.apps/admin-service restarted
PS C:\Users\amrin\OneDrive\Bureau\PFE\salon-microservice-app> kubectl rollout restart deployment admin-service -n salon
deployment.apps/admin-service restarted
```

Figure 21 Controlled restart scenario used to trigger the alert rule.

- **Fault-tolerance test:** a pod was manually deleted, and Kubernetes automatically recreated it to preserve the desired replica count.
- **Scalability test:** load was generated inside a service container, and HPA increased the number of replicas up to three pods.
- **Alerting test:** a pod restart scenario was used to validate alert notification reception.
- **Rate-limiting test:** repeated requests were sent to verify ingress-based request limitation and temporary source IP blocking.

Each test scenario was executed to verify one operational property of the platform. The functional access scenario validated

Table 1 Operational test scenarios and acceptance criteria.

Scenario	Action	Acceptance criterion
Access	Open the public domain through ingress.	Application pages are reachable through HTTPS.
Recovery	Delete a running pod manually.	Kubernetes recreates a pod automatically.
Scaling	Generate CPU load inside a service.	HPA increases the replica count.
Alerting	Trigger a restart condition.	Notification is received by the administrator.
Protection	Send repeated requests to ingress.	Excessive requests are limited or blocked.

end-user reachability through the public domain. The fault-tolerance scenario validated Kubernetes reconciliation by deleting a pod and checking that the desired state was restored. The scalability scenario generated resource pressure and verified that HPA increased the replica count¹⁵. The alerting scenario forced a restart condition and checked that the monitoring chain produced a notification. The rate-limiting scenario generated repeated requests to confirm that ingress rules reduced abusive traffic.

Table 1 summarizes the test scenarios, their objectives, and their acceptance criteria. The purpose of this table is to make the validation reproducible: each scenario starts from a known action, observes the platform reaction, and accepts the result only if the observed behavior matches the expected operational property.

```
PS C:\Users\amrin\OneDrive\Bureau\PFE\salon-microservice-app> kubectl delete pod admin-service-6c6774ff49-7gzqv -n salon
pod "admin-service-6c6774ff49-7gzqv" deleted from salon namespace
PS C:\Users\amrin\OneDrive\Bureau\PFE\salon-microservice-app> kubectl get pods -n salon
```

NAME	READY	STATUS	RESTARTS	AGE
admin-service-6c6774ff49-6pgws	1/1	Running	0	13s
admin-service-6c6774ff49-7gzqv	1/1	Terminating	0	2m16s
ai-service-557bcc7685-zzvm8	1/1	Running	0	2m31s
auth-service-c85645f47-cbk4f	1/1	Running	0	3m56s
booking-service-8495c4bf8f-dmpxn	1/1	Running	0	3m28s
frontend-d775c5869-8wfnf	1/1	Running	0	65s
notification-service-756f9887bb-c58ch	1/1	Running	0	2m47s
promotion-service-5b64ff6d89-zwmgn	1/1	Running	0	3m
reservation-service-f4797755b-fnsps	1/1	Running	0	3m15s
user-service-69f4d6ccc6-8p6np	1/1	Running	0	3m43s

Figure 22 Fault-tolerance scenario showing Kubernetes recovery after pod deletion.

```
PS C:\Users\amrin\OneDrive\Bureau\PFE\salon-microservice-app> kubectl get pods -n salon
```

NAME	READY	STATUS	RESTARTS	AGE
admin-service-6c6774ff49-6pgws	1/1	Running	0	65s
ai-service-557bcc7685-zzvm8	1/1	Running	0	3m23s
auth-service-c85645f47-cbk4f	1/1	Running	0	4m48s
booking-service-8495c4bf8f-dmpxn	1/1	Running	0	4m20s
frontend-d775c5869-8wfnf	1/1	Running	0	117s
notification-service-756f9887bb-c58ch	1/1	Running	0	3m39s
promotion-service-5b64ff6d89-zwmgn	1/1	Running	0	3m52s
reservation-service-f4797755b-fnsps	1/1	Running	0	4m7s
user-service-69f4d6ccc6-8p6np	1/1	Running	0	4m35s

```
PS C:\Users\amrin\OneDrive\Bureau\PFE\salon-microservice-app> kubectl exec -it user-service-69f4d6ccc6-8p6np -n salon -- sh
/app $ while true; do ;; done
```

Figure 23 Scalability scenario used to generate load and validate HPA behavior.

```
PS C:\Users\amrin\OneDrive\Bureau\PFE\salon-microservice-app> kubectl get hpa -n salon -w
```

NAME	REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS	AGE
admin-service-hpa	Deployment/admin-service	cpu: 2%/80%	1	3	1	121m
ai-service-hpa	Deployment/ai-service	cpu: 1%/80%	1	3	1	121m
auth-service-hpa	Deployment/auth-service	cpu: 2%/80%	1	3	1	121m
booking-service-hpa	Deployment/booking-service	cpu: 2%/80%	1	3	1	121m
frontend-hpa	Deployment/frontend	cpu: 0%/80%	1	3	1	121m
notification-service-hpa	Deployment/notification-service	cpu: 3%/80%	1	3	1	121m
promotion-service-hpa	Deployment/promotion-service	cpu: 3%/80%	1	3	1	121m
reservation-service-hpa	Deployment/reservation-service	cpu: 2%/80%	1	3	1	121m
user-service-hpa	Deployment/user-service	cpu: 302%/80%	1	3	3	121m

Figure 24 Horizontal scaling result showing three service replicas.

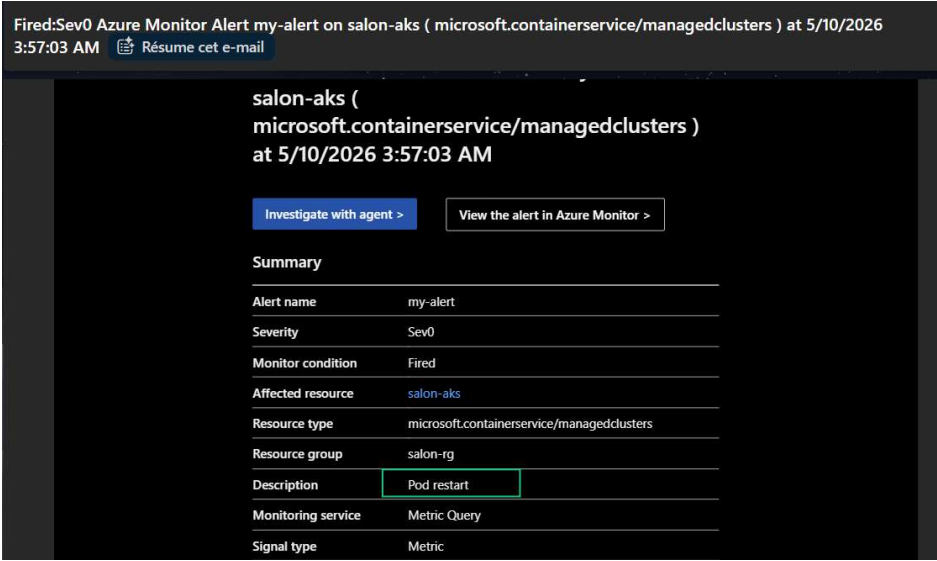


Figure 25 Alerting scenario confirming notification reception after the restart condition.

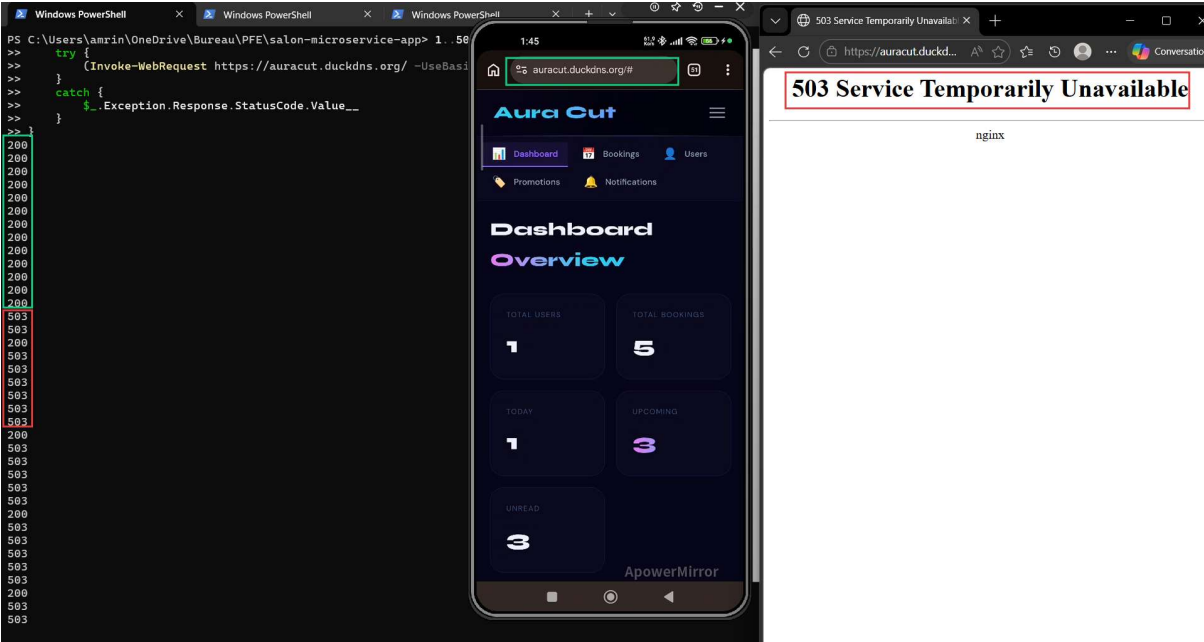


Figure 26 Rate-limiting and load-oriented scenario used to verify ingress protection.

5 Results and Discussion

The implemented architecture demonstrates that combining Docker, Terraform, AKS, Azure DevOps, PostgreSQL Flexible Server, and monitoring tools can transform a manual deployment process into a reproducible cloud-native workflow. The functional test confirmed that the application was reachable through the configured domain with an observed response time below one second during normal access tests. The CI/CD redeployment workflow completed in approximately five minutes from image build to pod refresh. The fault-tolerance test confirmed Kubernetes self-healing behavior, with the deleted pod replaced in less than one minute. The scalability test confirmed that HPA can increase the number of replicas in response to resource pressure, using an 80% CPU utilization threshold and scaling the service up to three replicas. During validation, CPU and memory consumption remained within acceptable operational limits for the tested workload. The alerting test confirmed the ability of the monitoring system to notify administrators. The rate-limiting test demonstrated an additional protection layer against excessive requests and saturation attempts.

Nevertheless, several limitations remain. The current evaluation is primarily functional and operational, and the quantitative indicators were collected in a limited validation context rather than through a long-duration production benchmark.

Table 2 Summary of validation tests.

Validation aspect	Observed outcome
Functional access	Application available through the configured domain, with response time below one second during normal access tests.
CI/CD deployment	Automated redeployment completed in approximately five minutes from build to pod refresh.
Fault tolerance	Deleted pod automatically recreated by Kubernetes in less than one minute.
Scalability	HPA used an 80% CPU threshold and increased replicas up to three pods during load simulation.
Resource usage	CPU and memory remained within acceptable limits during the validation workload.
Alerting	Notification received after alert condition.
Ingress protection	Excessive requests from one source were temporarily blocked.

Future work should include controlled benchmarking, security testing, cost analysis, throughput evaluation, and comparison with alternative deployment strategies.

5.1 Lessons Learned and Future Improvements

The implementation confirmed several practical lessons for cloud-native DevOps projects. First, the infrastructure layer should be automated before the application deployment layer because AKS, ACR, networking, database, and monitoring resources form the operational foundation of the platform. Second, local container validation remains important even when the final target is Kubernetes, because image build errors and runtime dependency issues are easier to diagnose before pushing images to the registry. Third, observability must be integrated early rather than added only after deployment, since metrics and alerts are required to evaluate resilience, scaling, and runtime stability.

The project also showed that screenshots and execution traces are useful validation artifacts. Pipeline captures demonstrate the delivery path from repository to cluster, Terraform captures document the infrastructure lifecycle, database captures confirm persistence initialization, and test-scenario captures provide evidence for recovery, scaling, alerting, and ingress protection. These artifacts make the experimental validation easier to audit and reproduce.

Several improvements can extend the current work. The CI/CD workflow can include automated unit tests, container vulnerability scanning, image signing, and policy-based deployment gates. The Kubernetes layer can be strengthened with network policies, pod security standards, resource quotas, and stricter secret rotation. The observability layer can be extended with distributed tracing, structured log aggregation, and service-level objectives. Finally, the evaluation can be expanded with longer load tests, cost measurements, backup-and-restore validation, and multi-environment promotion from development to staging and production.

6 Conclusion

This paper presented the design and implementation of a secure, scalable, and observable cloud-native deployment architecture for a microservices application. The solution integrates Docker for containerization, Terraform for infrastructure provisioning, AKS for orchestration, Azure DevOps for CI/CD automation, PostgreSQL Flexible Server for managed data storage, and observability tools for monitoring and alerting. The validation confirms the feasibility of deploying and operating the microservices application using automated and reproducible DevOps practices. Future research should extend the evaluation with quantitative benchmarking, advanced security analysis, cost optimization, and broader DevSecOps integration across the application lifecycle.

Acknowledgment

This work was supported by the Deanship of Scientific Research, Vice Presidency for Graduate Studies and Scientific Research, King Faisal University, Saudi Arabia, under Grant No. KFU263257.

References

1. Yohatech. Yohatech–Official Website. Accessed: Apr. 6, 2026. [Online]. Available: <https://yohatech.com/>
2. B. M. Harve, D. M. Bidkar, M. S. Krishnappa, G. Pandey, V. Jayaram, P. K. Veerapaneni, and G. Mehta, “The cloud-native revolution: Microservices in a cloud-driven world,” in *Proc. Int. Conf. Intelligent Cybern. Technol. Appl.*, 2024. [Online]. Available: <https://ieeexplore.ieee.org/document/10913359>
3. SoftFluent. DevOps: Definition, Roles, Principles, and Benefits. Accessed: Apr. 18, 2026. [Online]. Available: <https://www.softfluent.fr/blog/devops-c-est-quoi-roles-principes-avantages/>
4. O. Mart, C. Negru, F. Pop, and C. Dobre, “Observability in Kubernetes cluster: Automatic anomalies detection using Prometheus,” in *Proc. IEEE Int. Conf. High Perform. Comput. Commun.*, 2020. [Online]. Available: <https://ieeexplore.ieee.org/document/9407871>
5. Microsoft. What is Azure Kubernetes Service (AKS)? Accessed: Mar. 3, 2026. [Online]. Available: <https://learn.microsoft.com/azure/aks/>
6. HashiCorp. Terraform Documentation. Accessed: Apr. 2, 2026. [Online]. Available: <https://developer.hashicorp.com/terraform/docs>
7. Microsoft. Azure Database for PostgreSQL Flexible Server. Accessed: Mar. 18, 2026. [Online]. Available: <https://learn.microsoft.com/azure/postgresql/flexible-server/>
8. Docker. Docker Documentation. Accessed: Apr. 10, 2026. [Online]. Available: <https://docs.docker.com/>
9. Kubernetes. Kubernetes Documentation: Overview. Accessed: Feb. 24, 2026. [Online]. Available: <https://kubernetes.io/docs/concepts/overview/>
10. Microsoft. What is Azure DevOps? Accessed: May 6, 2026. [Online]. Available: <https://learn.microsoft.com/azure/devops/>
11. JFrog. Continuous Integration: Definition, Benefits, and Best Practices. Accessed: Apr. 18, 2026. [Online]. Available: <https://jfrog.com/fr/learn/devops/continuous-integration/>
12. Cloud Native Glossary Authors. Continuous Deployment. Accessed: Apr. 18, 2026. [Online]. Available: <https://glossary.cncf.io/fr/continuous-deployment/>
13. Microsoft. Azure Monitor Overview. Accessed: Jun. 12, 2026. [Online]. Available: <https://learn.microsoft.com/azure/azure-monitor/overview>
14. Microsoft Learn. Azure Monitor Overview–Azure Monitor. Accessed: Apr. 18, 2026. [Online]. Available: <https://learn.microsoft.com/fr-fr/azure/azure-monitor/fundamentals/overview>
15. A. A. Khaleq and I. Ra, “Development of QoS-aware agents with reinforcement learning for autoscaling of microservices on the cloud,” in *Proc. IEEE Int. Conf. Autonomic Comput. Self-Organizing Syst. Companion (ACSOS-C)*, 2021, pp. 13–19. [Online]. Available: <https://ieeexplore.ieee.org/document/9599204/>

Author Biographies

Nader Amri. I received my bachelor’s degree in computer engineering from the Higher Institute of Computer Science of Kef, Le Kef, Tunisia, in 2026. I am a cloud enthusiast with a strong interest in DevOps, cloud-native architectures, Kubernetes, Azure services, infrastructure automation, and secure application deployment. My work focuses on transforming manually deployed applications into reproducible, scalable, and observable cloud platforms.

Helmi Jbili contributed to the microservices application development of the AuraCut platform. His work covered the development of the application services, backend API logic, frontend integration, and functional preparation required before the containerization and cloud deployment phases. His contribution provided the application foundation on which the Docker, Kubernetes, and CI/CD deployment workflow was implemented.

Dr Sofien Mhatli supervised this work as an international IT, cloud, and cybersecurity expert. He is an IT and cloud trainer with certifications and expertise including VCP-DCV, AWS Solutions Architect Associate, Microsoft Azure Administrator, Azure DevOps, Microsoft 365 administration, Fortinet, VMware, Ericsson-LG, Huawei, modern workspace, cloud security, and cybersecurity engineering. With more than 16 years of experience in IT, cloud, telecommunications, and cybersecurity, he has supported enterprises in technological transformation, trained more than 2000 professionals, and deployed innovative solutions in more than 15 countries. His mission can be summarized as training, securing, and transforming organizations through modern cloud, DevOps, hybrid infrastructure, and cybersecurity practices.

Adel Aldalbahi is a Saudi academic and innovation leader, currently serving as Dean of the College of Engineering at King Faisal University. He holds a Ph.D. in Electrical Engineering from the New Jersey Institute of Technology and is recognized for his expertise in telecommunications, IoT, AI, and strategic program management.

Shiraz Ouertani Enseignante Universitaire | IT Expert. Professional Co-Supervisor of this work. Certifications: Azure Administrator Associate · CCNA Security · ANSSI · SFC™. Areas of expertise: Cloud/Azure · Cybersecurity · DevOps · Cloud-Native · Monitoring.