Release Management Plan
CA2 Project
Enterprise Architecture in Deployment

AINUL HABIB

Abstract

This is a Release Management Plan for Blood Pressure Monitoring Web Application. The Plan covers architecture updates for the creation of loosely coupled microservices, and continuous build, test, and release plans as part of the Automated DevOps Cycle.

Introduction

The Blood Pressure Monitoring Web Application, previously deployed will undergo architectural transformation to a microservice architecture, supporting de-coupled and independent services. Release, Delivery, and Deployment solution will embrace DevOps techniques to deliver Continuous Build, Test, Release, and Deployment pipeline. The Release plan will leverage from Blue-Green Deployment strategy to accomplish zero downtime in deployment.

Using a microservices approach, the team is promising to deliver a highly scalable, available solution, accommodating high traffic as business need expands.

The overall solution, which includes both Microservices and DevOps approach, will be robust, reliable, cost-effective, and scalable, satisfying expanding business requirements.

The approach also promotes vulnerability checks during build and deployment, which will guarantee the software in production is of higher quality and risk-free.

Microservices: Design and Decomposition

Current Application Design

The current Application is designed in 2-Tier service. A Front-End UI Service and a Backend Web Service, with Mongo Database working as a persistence data store. Current solution design does follow basic Microservices techniques, it is de-coupled and independent to some extent. To fully leverage a better Microservices Design, more de-coupling and decomposition will be needed.

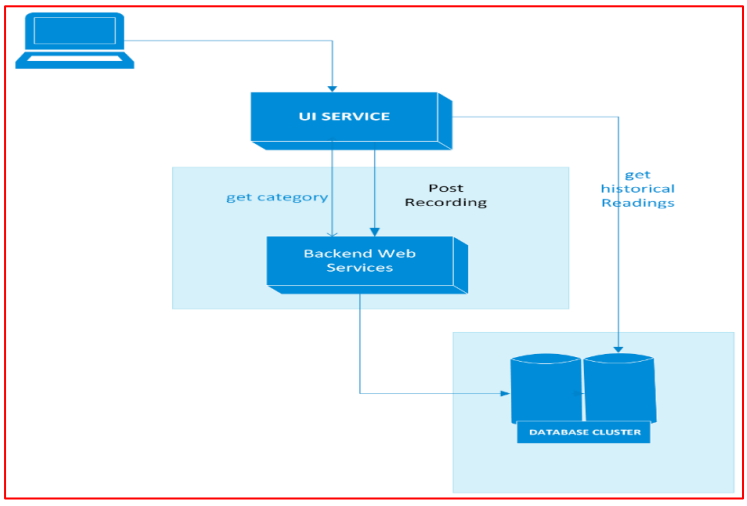


Figure 1: Current Design

Architecture Changes

1. Frontend UI service should not communicate with Database directly.
    a. The UI Service should delegate all data query to backend services.

    b. Exposure to the complexity of database connection, data structure and access to credentials should be limited to few backend Microservices only.
2. Separate Microservice for calculating the Category.
    a. Calculation of Category is an independent business process.
    b. This reduces the load on Backend services.
    c. It enhances the de-coupling making the solution more independent.
3. Separate Microservice to record Blood Pressure Readings.
    a. Enhance flexibility in accepting reading from other sources.
    b. Separate channel for writing to the database.
4. Backend Service is transformed to Data Service only responsible to read historical data from the database.
    a. It Reduces the responsibility of Data Service, restricting it to only read historical data from the database.
    b. It increases flexibility, there is a separation of services and channels for reading and writing to the database.

The flow results in Microservices designed by separation of concern. Each Microservices has its own responsibility, lifecycle, and can scale independently. The Independence and de-coupled nature of Microservices increases testability, it can be patched and upgraded without downtime.

Each microservices is built, deployed, and tested independently. All Microservices is packaged as independent services, deployed as one solution.

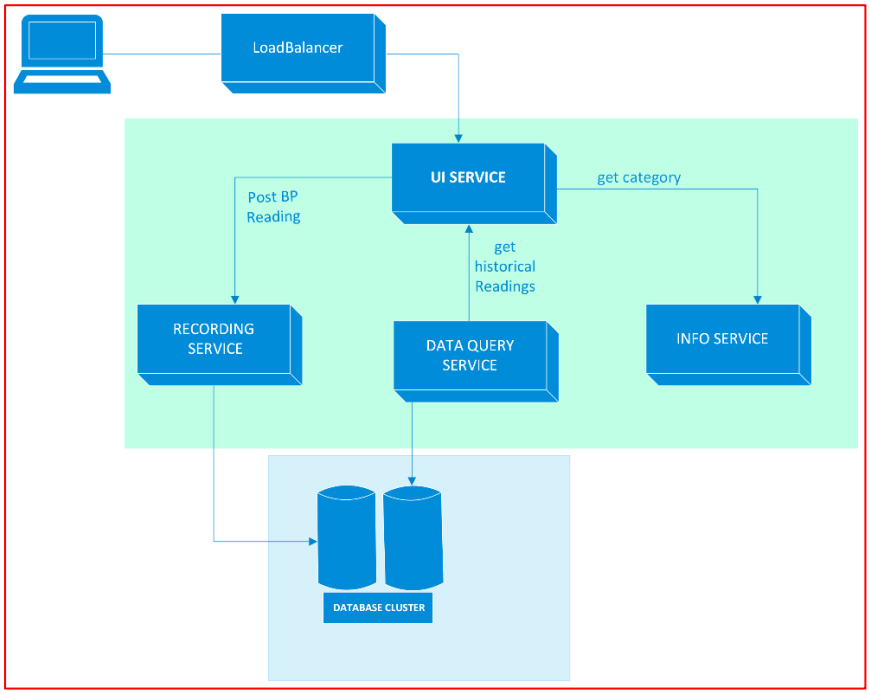


Figure 2: Updated Microservices

Implementation

The new design consists of 3 Microservices and a UI Service

1. Info Service: Calculates the category of Blood Pressure.
2. Recording Service: Saves the Blood Pressure Reading to Database.
3. Data Services: Reads the historical Blood Pressure data from Database.
4. UI Service: Provides the User Interface, accepting BP reading using a Web Application. It processes by calling the above 3 microservices.

Technology Stack

To reduce the technology stack, it is better to stick with one programming language. NodeJS is perfectly suitable for all purposes. It is powerful, fast, scalable, robust, functional, and reactive.

Code Updates

Choosing NodeJS reduces the technology stack, which will also reduce the complexity in build and testing infrastructure. NodeJS is reactive by design and has many libraries which support REST communication and E2E Testing. We also cannot ignore the fact that NodeJS has larger community support and are actively releasing new libraries. As such transforming all the functions of the previous Blood Pressure Monitoring backend Java-based Web Services to a NodeJS application will be good for the product.

Express JS

For REST-based communication we Express JS is now considered the most accepted module for NodeJS applications. It supports all JSON-based content types by default, reducing the conversion complexity.

Cypress

Cypress is an automated Test Tool, with the capability to test E2E (End To End), Components, and User Interface. Various reporters can be added to report the result.

Cloud Infrastructure

The application is delivered as a cloud solution. This affects how the application is packaged, released, and deployed. The process also requires the handling of credentials and runtime properties. The application must be deployed as a solution with complete automation from build to deployment.

Cloud Deliverables

Application is delivered as a cloud solution on a Kubernetes cluster rather than composing them on an independent virtual machine(s). Kubernetes is a vendor-agnostic solution, which can run on any cloud provider. It also creates its private network with its own isolated, security networking, and scalability rules. Kubernetes scales the application unit, across its cluster which is composed of its nodes. This reduces the effort in provisioning new VMs for scaling up.

For deploying the application in a Kubernetes cluster:

- Individual Microservice applications must be delivered as a Docker Image.
- Manifest files for each Microservices, containing deployment and networking of Docker images as Pods in Kubernetes cluster.
- Packaging and grouping the manifests as Helm Charts, published to a docker registry.
- All Credentials and Environment properties must be injected at runtime.
- All deployable artifacts (i.e. docker image and helm charts) must be published to the cloud provider’s Container Registry.
- Application should be disposable, it can be destroyed and started without any side-effect.
- All deployment, configuration, scaling and networking requirements should be configured in a manifest file.

Helm Chart

A Helm chart can be thought of as a Kubernetes package. Charts contain the declarative Kubernetes resource files required to deploy an application. Similar to an RPM, it can also declare one or more dependencies that the application needs in order to run.

(Andrew Block, JUNE 2020)

Advantages of Helm Chart:

- Helm chart provided reusable templates for Kubernetes deployment, which allowed us to update the configuration during the deployment phase for different environments.
- Helm provides a dependency option, which allows us to integrate with another dependent application deployment. Using Helm dependency, we now package and deploy the entire solution as one single chart or deployment.
- Helm also manages application version and lifecycle (Install, Upgrade and Rollback and Uninstall)

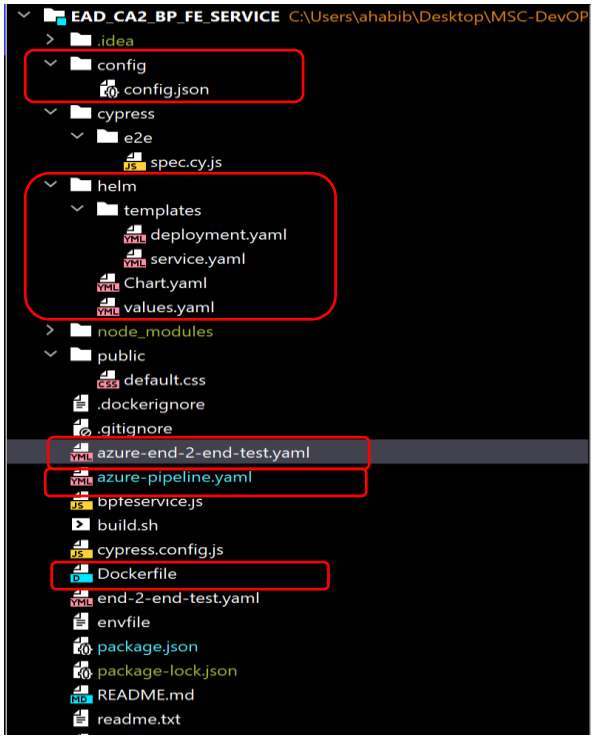
Code Repositories

Each Microservice has its repository. This allows us to control the build, versioning, and release independently.

Repository Name	Github	Purpose
EAD_CA2_BP_INFO_SERVICE	link	Business function, which accepts blood pressure readings and computes the Level.
EAD_CA2_BP_DATA_SERVICE	link	Provides historical blood pressure readings based on ID like email
EAD_CA2_BP_RECORDING_SERVICE	link	Service which records the blood pressure reading to the database
EAD_CA2_BP_FE_SERVICE	link	Front End UI service Blood Pressure reading
EAD_CA2_BP_DEPLOYMENT	link	Provides the Solution as Helm Chart packaging all the above services as dependencies. The repository also contains deployment pipeline code for Mongo Database and Ingress-nginx configuration.

Code Structure

The code is structured keeping in mind the NodeJS requirements, NPM requirements, Helm packaging, and Cypress Test. The Helm folder contains the Kubernetes manifest, services, and value files. The code repository also contains Azure Test and Build Pipeline file, which is executed in the Azure DevOps pipeline. Cypress Test specification is available in the “cypress” folder.



azure-end-2-end-test.yaml : Contains azure pipeline code for building and testing docker image.

azure-pipeline.yaml: Contains azure pipeline code for creation of docker image and Helm chart. Both Helm chart and docker image is published to Azure Container Registry

Figure 3:Source Code Structure

Architectural Challenges

High Scalability and Availability

The application is deployed in the Kubernetes cluster as a POD. A pod is a basic deployment unit, which is orchestrated by the Kubernetes engine to distribute and scale in their clustered nodes. Kubernetes will handle scalability, and maintain SLA, security, communication, and high availability.

Each Microservices application is deployed with a minimum of 2 replicas, to support High Availability. Kubernetes manages the load balancing between the replicas.
Kubernetes will ensure:

- Each replica resides on multiple nodes.
- Maintain the SLA, always keeping the minimum replica-sets in the cluster.

Networking, Exposing Services

Microservices deployed in Kubernetes expose their container port, using "service". Kubernetes assign a virtual IP to that service, which is known as "ClusterIP". The service with virtual IP and the Port number becomes a communication endpoint for the application.

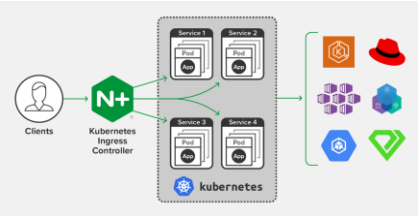
All Microservices configure their services as default, i.e. "ClusterIP". This restricts pod accessibility and communication within the cluster.

The external traffic will be proxied to the Front-End UI Service using the "Ingress-Nginx" Kubernetes Load-Balancer.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: bprecordingsvc
spec:
  type: ClusterIP
  ports:
    - name: http
      port: 30256
      targetPort: 30256
  selector:
    app: bprecording
```

Figure 4 : Sample Service YAML

Nginx Ingress Controller (expose service to Web)



An Ingress Controller is a Kubernetes controller that is deployed manually to the cluster, most often as a Daemon Set or a Deployment object that runs dedicated Pods for handling incoming traffic load balancing and smart routing. It is responsible for processing the Ingress objects (which specify that they especially want to use the Ingress Controller) and dynamically configuring real routing rules. A commonly used Ingress controller for Kubernetes is ***ingress-nginx***

(McKendrick, 2022)

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: bp-web-app-ingress
  annotations:
    nginx.ingress.kubernetes.io/ssl-redirect: "false"
    nginx.ingress.kubernetes.io/use-regex: "true"
    nginx.ingress.kubernetes.io/rewrite-target: /$2
spec:
  ingressClassName: nginx
  rules:
    - http:
        paths:

          - path: /live(/|$)(.*)
            pathType: Prefix
            backend:
              service:
                name: bpfesvc
                port:
                  number: 32137

          - path: /(.*)
            pathType: Prefix
            backend:
              service:
                name: bpfesvc
                port:
                  number: 32137
```

Ingress-Nginx controller scans all namespaces for resources of the kind "Ingress", which contains routing information based on the path prefix. The routing information maps the URL's path prefix with a Kubernetes service name and port, thus routing traffic to the respective service in the backend.

Figure 5 : Ingress YAML file

Deploying Ingress Controller

Ingress controller is not available by default in many Cloud Infrastructure, it can be deployed using minimum effort. The details are provided in Installation Guide - NGINX Ingress Controller (kubernetes.github.io) and Create an ingress controller in Azure Kubernetes Service (AKS) - Azure Kubernetes Service | Microsoft Learn

A dedicated Ingress-Nginx deployment pipeline is available in code repository
https://github.com/AINULX00159358/EAD_CA2_BP_DEPLOYMENT/blob/main/LoadBalancer/ingress/ingress-nginx-deployment.yaml

Ingress Controller is delivered as Helm chart, which starts an Ingress Nginx POD and provisions a Service of type "LoadBalancer". When a service is exposed as type "LoadBalancer" in the cluster, Kubernetes assign a Public IP to this service, which is visible as "EXTERNAL-IP".

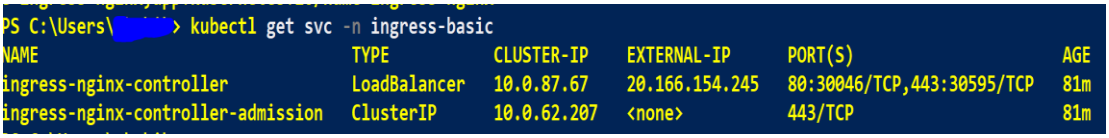


Figure 6: ingress-nginx-controller

External Traffic is received on the Public IP by the Ingress controller. The routing rules prepared using configuration will route the traffic to the configured service based on the path prefix.

DevSecOps Delivery Pipeline

Delivery Pipeline

The Blood Pressure Web Application is divided into 4 independent deliverable services or applications. All applications are NodeJS Applications, it will use NPM (Node Package Manager) to assemble their dependencies at runtime. Dockerfile is provided in every repository to package a Docker image and pushed to ACR (Azure Container Registry).

Finally, the application is packed as a Helm Chart. The helm chart contains the Kubernetes Manifest file that defines the pod (, Kubernetes basic processing unit). Pod will download and compose a docker image on startup. Helm chart also contains the Service YAML file, which specifies the ports to be exposed, and a service name.

Azure Pipeline

Azure DevOps provides the infrastructure for developing and running the Pipeline. It has options to connect to ACR, AKS, and external GitHub Repository. The pipeline has credentials, authorizations, subscriptions and service connections to facilitate the build, release, and deployment of the application.

Build Stages

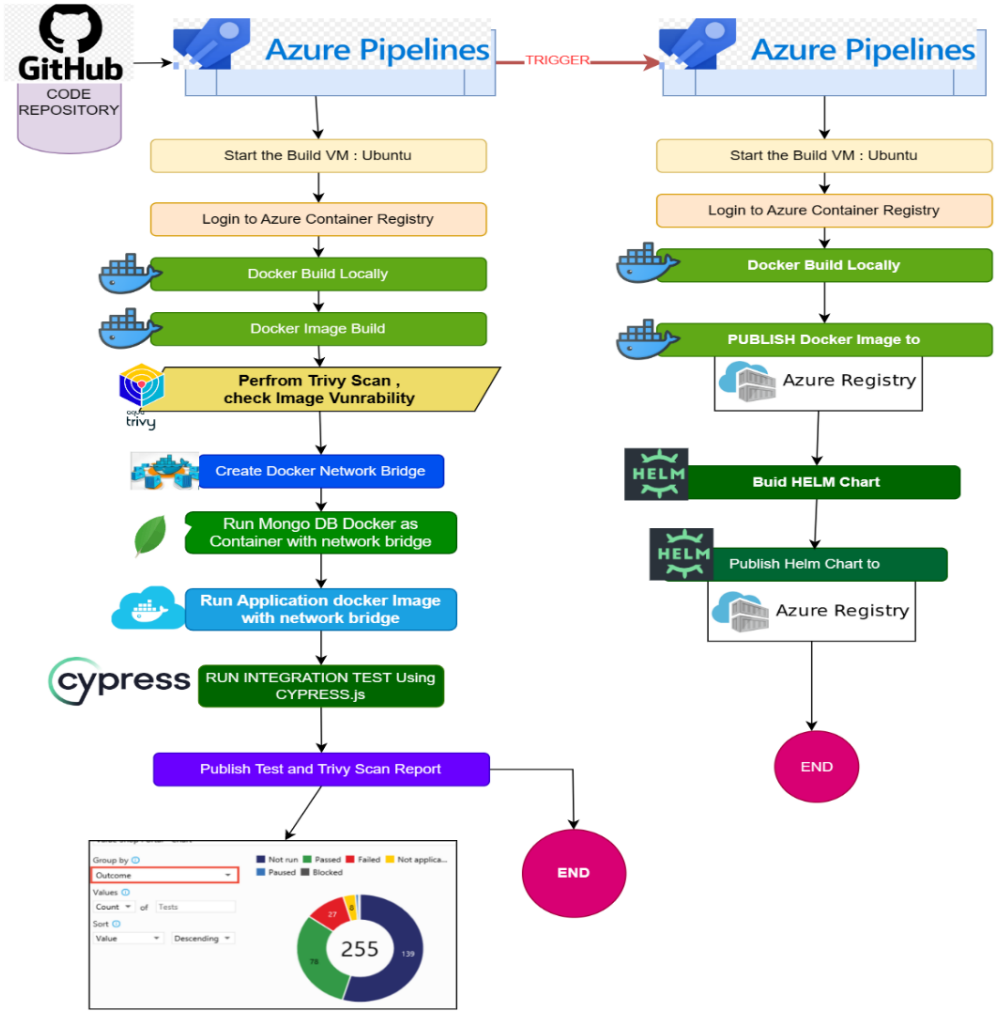


Figure 7 : Build Stages

Build Pipeline

The build and release pipeline is stored as a YAML file in our code repository. Azure Pipeline provisions these files, and executes them when code change is triggered.

List of all Build and Release pipelines

Name	Pipeline File	Purpose
Data-Service-Test	azure-end-2-end-test.yaml	Build Docker image, Check Vulnerability, Integration Test for bpDataService app
Data-Service-Build	azure-pipeline.yaml	Package Docker Image and Helm Chart to Azure Container Registry for bpDataService app (*triggered by Data-Service-Test success)*
INFO-SERVICE-TEST	azure-end-2-end-test.yaml	Build Docker image, Check Vulnerability, Integration Test for bpInfoService app
INFO-SERVICE-BUILD	azure-pipeline.yaml	Package Docker Image and Helm Chart to Azure Container Registry for bpInfoService app (*triggered by INFO-SERVICE-TEST success)*
Recording-service-test	azure-end-2-end-test.yaml	Build Docker image, Check Vulnerability, Integration Test for bpRecordingService app
Recording-service-build	azure-pipeline.yaml	Package Docker Image and Helm Chart to Azure Container Registry for bpDataService app (*triggered by Recording-Service-Test success)*
FE-Service-Test	azure-end-2-end-test.yaml	Build Docker image, Check Vulnerability, Integration Test for bpFEService app
FE-Service-Build	azure-pipeline.yaml	Package Docker Image and Helm Chart to Azure Container Registry for bpFEService app (*triggered by FE-Service-Test success)*
SOLUTION-BUILD	helm-build-pipeline.yaml	Package all above applications as Helm Chart Dependency and publish a common Helm Chart to Azure Container Registry (*triggered by FE-Service-Test success)*

Table 1: Build Pipelines

Solution Build

The “SOLUTION-BUILD” pipeline prepares the Helm charts for the solution, packing all microservices as its dependent Charts, and a common value file. The Helm chart is published to Azure Container Registry as “charts/bpwebapp”.

The Helm chart from the Solution-build also packages a service manifest for the Mongo Database service of type “ExternalName”.

```
appVersion: "1.0.0"
dependencies:
  - name: bpdataservice
    repository: "oci://ainhabacr.azurecr.io/charts"
    version: "1.0.0"
  - name: bpinfoservice
    repository: "oci://ainhabacr.azurecr.io/charts"
    version: "1.0.0"
  - name: bprecordingservice
    repository: "oci://ainhabacr.azurecr.io/charts"
    version: "1.0.0"
  - name: bpuiservice
    repository: "oci://ainhabacr.azurecr.io/charts"
    version: "1.0.0"
```

Figure 8 : Helm Chart Dependency

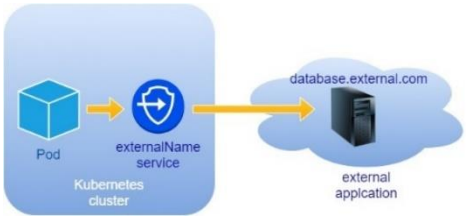


Figure 9 : Service Type as External Name

Services of type ExternalName can be used to proxify applications that are not actually running on your cluster, while still keeping the Service as a layer of abstraction that can be updated at any time.

(Raul, 2021)

Mongo Database service is deployed in the “mongo” namespace. Application in other namespaces like “blue” or “green” can access the service FQDN. But instead using an ExternalName allows us to update the service dynamically without updating the running application consuming it.

```
apiVersion: v1
kind: Service
metadata:
  name: mongodb-service
spec:
  type: ExternalName
  externalName: mongodb-service.mongo.svc.cluster.local
  ports:
  - port: 27017
```

Figure 10 : MongoDB ExternalName Service

Packaging the Solution

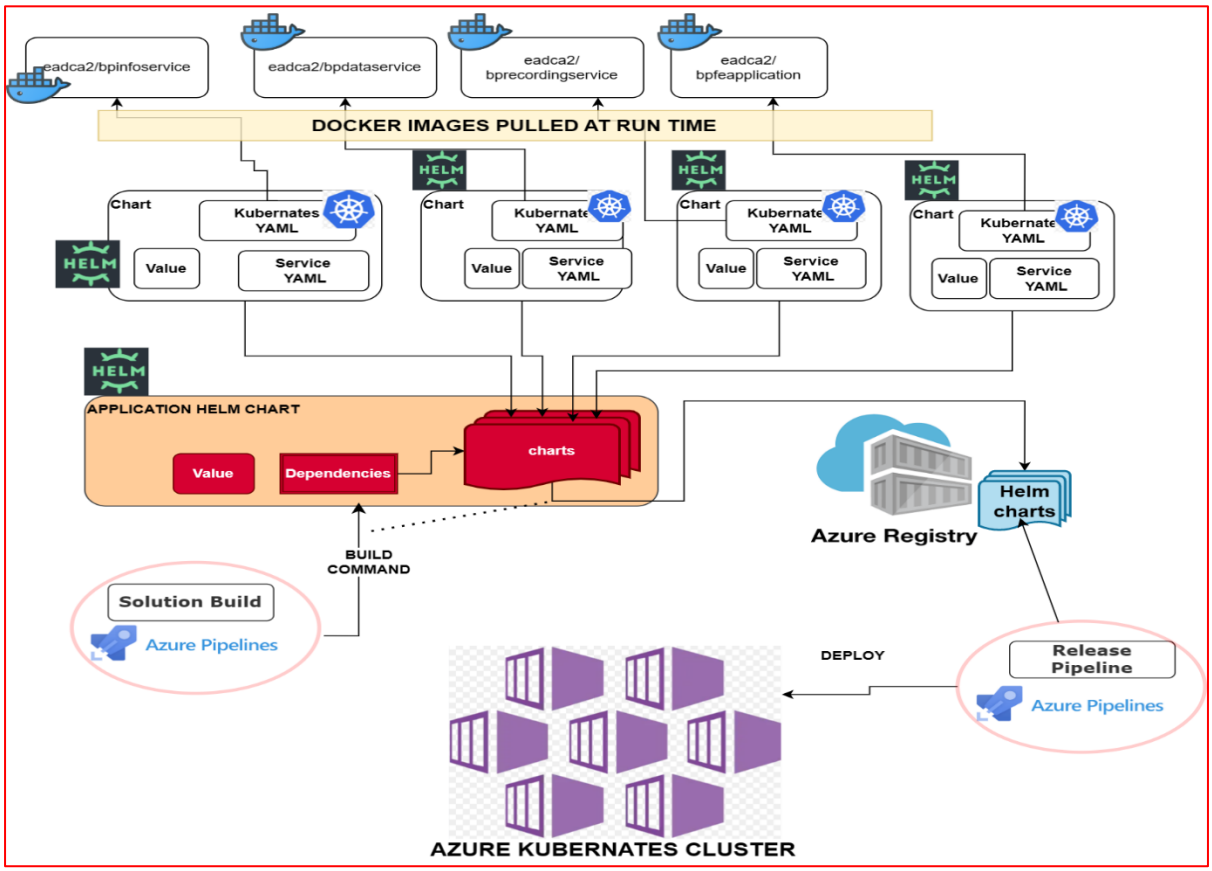


Figure 11: Helm Chart Packaging Solution

Vulnerability Scan

Software vulnerability scan must be carried on the deployable artifacts and container images, before releasing the software for production.

Trivy Scan is an open-source tool that can scan docker images for various vulnerabilities and breaches. The scanner is integrated into the Azure Build Pipeline stages so that vulnerability scanning is done in every build before publishing to the Azure Container Registry.

The vulnerability report will be published in the build pipeline.

To use Trivy Scan, it is first installed on the build VM machine. After the docker image is built, the Trivy scan is executed for the vulnerability scan. The image build is passed as a parameter to the Trivy scanner.

```
- script: |
    sudo apt-get install rpm
    wget -q https://github.com/aquasecurity/trivy/releases/download/v$(trivyVersion)/trivy_$(trivyVersion)_Linux-64bit.deb
    sudo dpkg -i trivy_$(trivyVersion)_Linux-64bit.deb
    mkdir $(ScanResultsPath)
    trivy image --scanners vuln --severity CRITICAL,HIGH -o $(ScanResultsPath)/results.table bprecordingservice:CA2_TEST_V1
  displayName: 'Scan using Trivy scan'
```

Figure 12: Trivy Scanner script

Trivy Scan, by default, exits with exit code of 0. If the build must be broken in case of Critical and High vulnerabilities, "*--exit-code 1*" option will be added to the Trivy scan command.

Automation in E2E Testing

Cypress provides an E2E Test framework. It is capable of testing application that uses Web UI or REST interface. It is specifically designed for NodeJS applications capable of doing live testing of UI components. The reporting tool is added to consume to test result and publish it to the test pipeline. Cypress Test process is integrated into the Azure Build Pipeline, which provides the capability to perform End to End Test during the build process, before pushing the docker image to ACR.

To run the Cypress test, the NodeJS application should be running. The application's docker image is executed in the local docker environment, along with its dependencies. Cypress Test will perform an "End to End" test, calling the application's exposed endpoints.

Cypress test is executed using NPM command, with reporting format as JUNIT. The test report is piped to a file, which is published to the Build Pipeline Result.

```
bash: |
  echo "Running Cypress Test"
  node ./bpfeservice &
  sleep 1
  export CYPRESS_APPURL=http://127.0.0.1:32137/
  npx cypress run --browser chrome --reporter junit --reporter-options "mochaFile=./cypress/results/TEST-bpfeservice-Result.xml,toConsole=true"
displayName: 'Cypress Test'
continueOnError: false

task: PublishTestResults@2
displayName: 'Publish Test Results ead-ca2-bp-fe-service'
inputs:
  testResultsFiles: '**/TEST-*.xml'
  failTaskOnFailedTests: true
  testRunTitle: 'FE-SERVICE-E2E-TEST'
```

Figure 13 : Cypress E2e Testing pipeline config

Sample Test Report

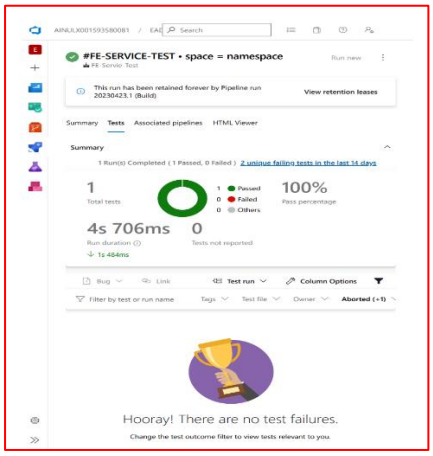


Figure 14: Sample test report

Deployment Pipeline

The Blood Pressure Web Application is deployed to the Azure Kubernetes cluster. The deployment will require “Service Connection”, which acts like authorization for the pipeline to deploy on Azure Kubernetes. Each steps involving connection to Azure Kubernetes Cluster require either Azure resource subscription or Service Connection per namespace.

The Release Pipeline deploys the Helm Chart created and pushed by the build pipeline to the Kubernetes cluster.

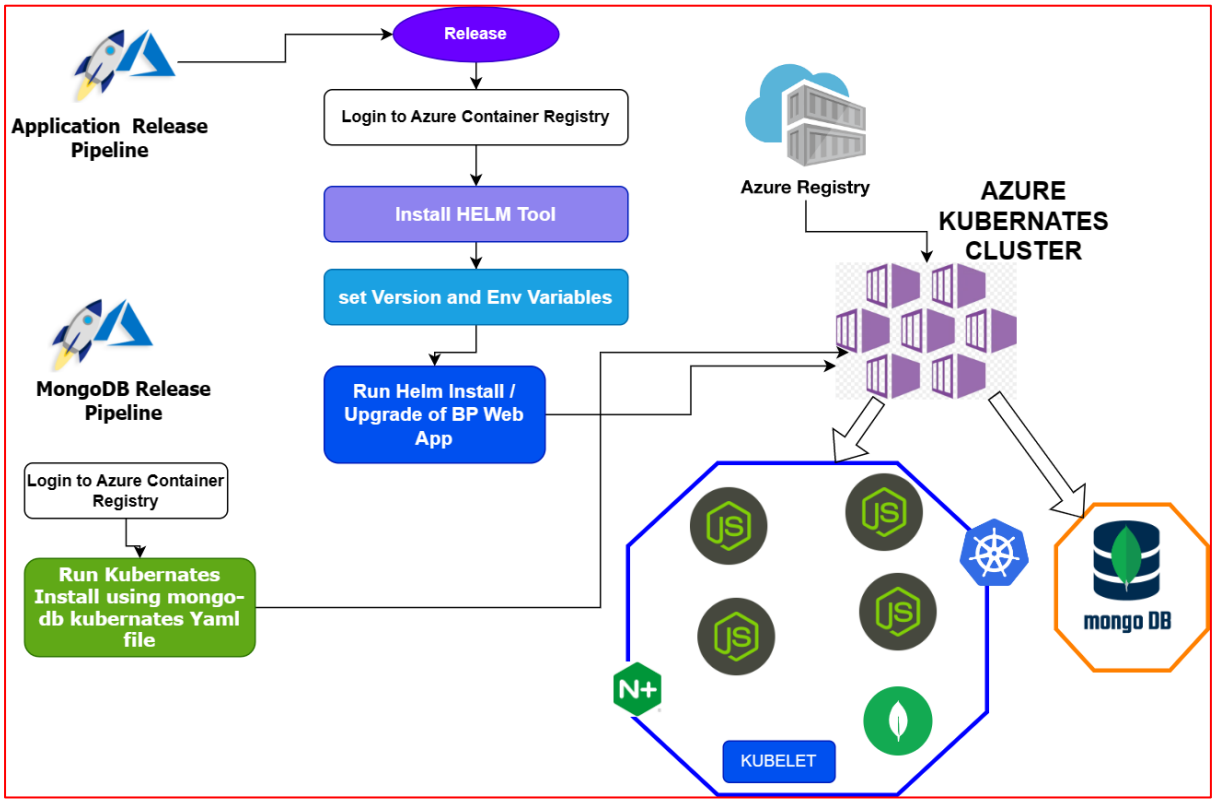
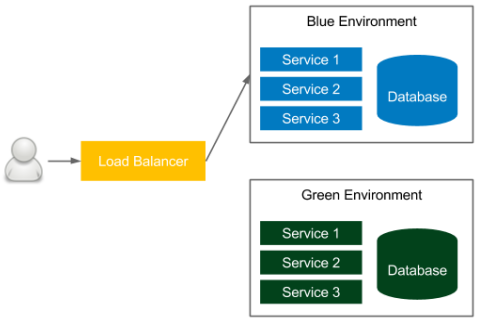


Figure 15 : Deployment Pipeline high-level view

Deployment Strategy (Blue Green Deployment)

The deployment process starts by creating a Release Branch. The docker image and helm chart version are tagged by REL-*version-number.* (REL stands for Release). All deployments are triggered manually using a dedicated release pipeline for different namespaces. All security and authorization are injected into the pipeline, reducing any user entry.

Blue Green Deployment Strategy



*Blue-green deployment is a technique to reduce the downtime associated with the release. It concerns having two identical production environments, one called **green**, the other called **blue**.*

If we want to make a new release, then we deploy everything to the green environment and, at the end of the release process, change the load balancer to the green environment. (Leszko, 2017)

To accomplish Blue Green Deployment, the Blood Pressure Application is deployed in two different namespaces, blue and green. Nginx Ingress configuration will be used to expose the application to public IP. As per the norm, an application that is marked as blue is considered Live. The green namespace is reserved for application staging.

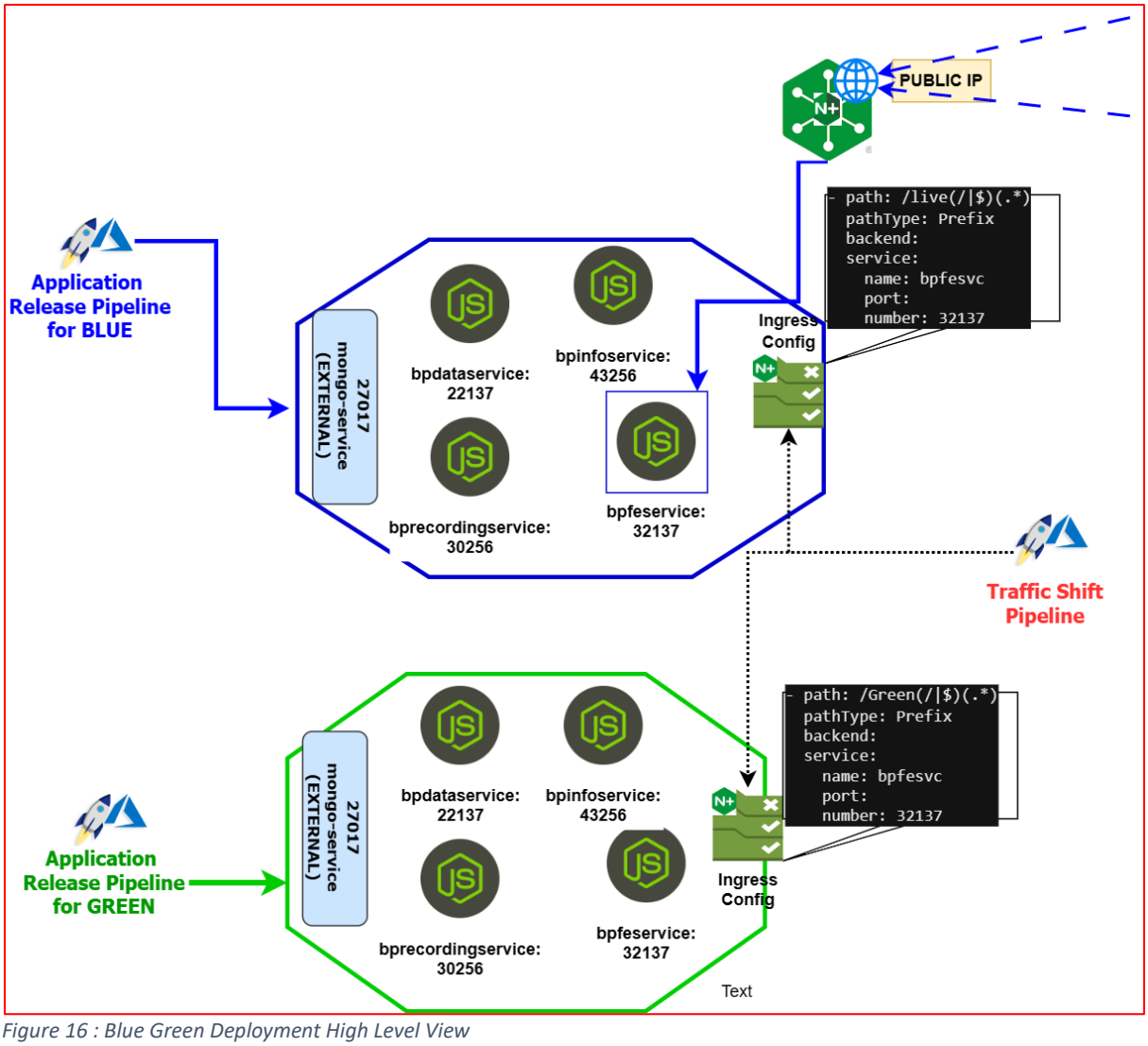


Figure 16 : Blue Green Deployment High Level View

Azure Release Pipelines

Link to Releases: https://dev.azure.com/AINULX001593580081/EAD_CA2/_release

Deployment Pipelines

Release Pipeline Name	JSON File
Application Deployment BLUE	Application Deployment BLUE.json
Application Deployment GREEN	Application Deployment GREEN.json

Table 2 : Deployment Pipeline

Blue Green Pipelines

Release Pipeline Name	JSON File
BlueToGreen	BlueToGreen.json
GreenToBlue	GreenToBlue.json

Table 3 : Blue Green Traffic Pipelines

Versioning, Upgrade Rollback Procedure

Application will follow Semantic Versioning rule, which uses 3 parts of version number, i.e. "MAJOR.Minor.Patch". Applications are deployed and upgraded using Helm. Every deployment is an Helm upgrade command, which will updated currently deployed application if there is any change in the artifacts.

After upgrade, Helm does a rolling restart of the *upgraded* PODs. There is no service downtime due restart, because minimum 2 replicas are maintained per microservices.

Since Blue-Green deployment strategy is followed, new version of application is first deployed on Staging site (Green). Application is made live only after successful completion of E2E (End-To-End) Test.

Helm rollbacks the application to the last version, which is triggered after failure of E2E test.

Distributing Database Credential

Database credentials is safely distributed in using a Kubernetes secrets. Kubernetes provide a kind called "secrets" which is specifically designed to hold confidential data. Confidential data is injected to pods deployment using volume and persistence volume claim.

The secret is deployed using deployment file. There is one secrets per namespace. During Deployment of applications the secrets is passes as volume mount to a "mountPath" *"/etc/mongo-secrets",* which is read by application at runtime.

```
apiVersion: "v1"
kind: "Secret"
metadata:
  name: "mongo-secrets"
type: "Opaque"
#  username = base64 encoded "mongo". password = base64 encoded "1f2d1e2e67df"
data:
  MONGO_USERNAME: bW9uZ28K
  MONGO_PASSWORD: MWYyZDF1MmU2N2RmCg==
  MONGO_CREDENTIAL: bW9uZ286MWYyZDF1MmU2N2RmCg==
#  mongo:1f2d1e2e67df
```

Figure 17: Mongo Secrets yaml

To secure Mongo Database, credentials is provided using security file, which is volume mounted during deployment. The security file is embedded in pod volume from secret named "mongo-secrets", created in earlier steps.

```
spec:
  volumes:
    - name: mongo-secrets
      secret:
        secretName: mongo-secrets
        items:
          - key: MONGO_USERNAME
            path: MONGO_USERNAME
            mode: 0444
          - key: MONGO_PASSWORD
            path: MONGO_PASSWORD
            mode: 0444
  containers:
  - image: mongo
    name: mongodb
    ports:
    - containerPort: 27017
    volumeMounts:
      - name: mongo-secrets
        mountPath: /etc/mongo-secrets
        readOnly: true
    env:
      - name: MONGO_INITDB_ROOT_USERNAME_FILE
        value: /etc/mongo-secrets/MONGO_USERNAME
      - name: MONGO_INITDB_ROOT_PASSWORD_FILE
        value: /etc/mongo-secrets/MONGO_PASSWORD
```

Figure 18 : MongoDB deployment with secrets

Cluster Role is created to control access to secrets created, safeguarding against accidental modification. The cluster role is then added to specific service account.

```
kubectl create clusterrole mongosecretadmin --verb=get --verb=list --verb=create --verb=update --resource=secret --namespace=mongo
kubectl create clusterrolebinding crbmongosecret --role=mongosecretadmin --serviceaccount=ainhab-serviceaccount --namespace=mongo
```

Figure 19: Cluster Role and Role Binding for mongo secrets

Monitoring and Logging

Kubernetes Cluster Metrics

Kubernetes has a dedicated service, called “kube-state-metrics”, which listens to the Control Plane and generates metrics about resources. These metrics provide valuable insight about the health of the cluster, resource utilization and deployed artefacts. Alerts is also configured using the Kubernetes monitoring and metrics.

Azure provides in-house monitoring tools for Kubernetes, which analyse logs, generate metrics, and configure alerts. External cloud-based monitoring and logging solution is popular in enterprise world. Some popular examples are Datadog, Splunk, Dynatrace, etc. These solutions requires a light weight agent or jobs in the cluster, which scrapes logs and metrics to the cloud server.

Monitoring solution is also provided using Prometheus and Grafana. Both are added using Azure Portal to the monitoring workspace.

Prometheus Integration

Prometheus is deployed from GitHub using Helm Command. This deploys Prometheus “Node Exporter”, “Operator” and “State-Metrics”. Prometheus server is exposed as “Load Balancer” using a Configuration YAML file https://github.com/AINULX00159358/EAD_CA2_BP_DEPLOYMENT/blob/main/monitoring/prometheus-loadbalancer.yaml

This exposes the Prometheus server to external traffic, using a public IP passed in the YAML file.

Add Grafana using Azure Portal

Azure portal provides a deployment Wizard to add the resource named “*Azure Managed Grafana*". The wizard deploys a basic Grafana web application, using subscription ID, Resource Group and location. The basic deployment creates a Grafana workspace with a public endpoint.

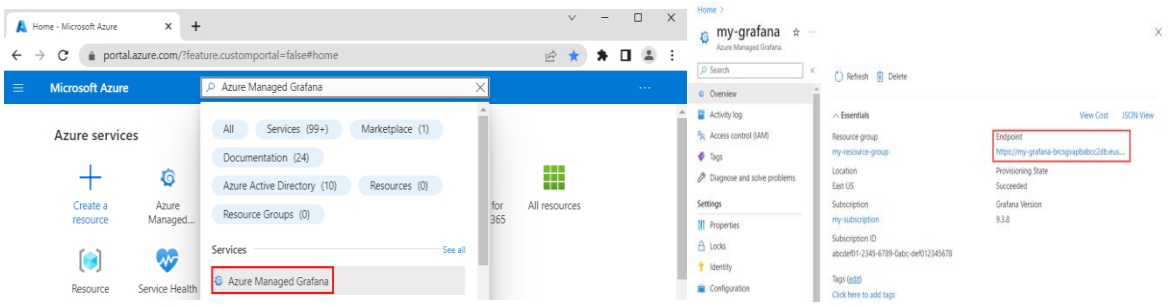


Figure 20: Add Azure Managed Grafana

Connect to Prometheus server deployed in previous steps with Grafana.

Prometheus is added as a “Data Source” in Grafana using Prometheus Server Load Balancer IP, exposed in previous steps. After Prometheus “Data Source” is synced, Kubernetes Dashboards starts populating the metrics after few seconds.

Prometheus Agents collect all the metrics from Kubernetes “metric-server” and “control pane”, scrapes it to Prometheus server. Grafana pulls these metrics to populate the graphs and user interfaces.

Some screen shorts of the Dashboard are provided below

Figure 21: Grafana Kubernetes Dashboard

Future Improvements

- Secure Mongo database credentials using Vault (e.g. HashiCorp)
- Use Datadog as Enterprise-wide external Logging and Monitoring Solutions

Cost Estimates

Service category	Service type	Region	Description	Estimated monthly cost
Compute	Virtual Machines	North Europe	3 A4 (8 Cores, 14 GB RAM) x 730 Hours (Pay as you go), Linux, (Pay as you go); 2 managed disks – E30, 10,000 transaction units; Inter Region transfer type, 10000 GB outbound data transfer from North Europe to UK South	$1,424.70
Compute	Azure Kubernetes Service (AKS)	North Europe	Standard; Cluster management for 2 clusters; 3 F4s (4 vCPUs, 8 GB RAM) x 730 Hours (Pay as you go), Linux; 1 managed OS disk – E40	$794.54
Developer tools	Azure DevOps		10 Basic Plan license users, 10 Basic + Test Plans license users, Free tier - 1 Microsoft Hosted Pipeline(s), 11 Self Hosted Pipeline(s), 10 GB Artifacts, 0 VUMs	$716.00
DevOps	Azure Monitor	North Europe	Managed Prometheus, Grafana, Metrics collection , etc	$407.53
Containers	Azure Container Registry	North Europe	Standard Tier, 3 registries x 30 days, 30 GB Extra Storage, Container Build - 1 CPUs x 1 Seconds - Inter Region transfer type, 5 GB outbound data transfer from North Europe to East Asia	$62.69
Networking	IP Addresses	North Europe	Standard (ARM), 1 Static IP Addresses X 730 Hours, 0 Public IP Prefixes X 730 Hours	$3.65
Support			Support	$560.00
			Total PER MONTH	**$3,970.00**

Conclusion

The Automated deployment pipeline for the Blood Pressure Web Application has all DevSecOps features, which build, test, scans and deploys Microservices application to Azure Kubernetes Cluster. The process automates the build, packaging, and release of Microservices to Container Registry. The Automated Deployment pipelines deploys applications as Blue-Green deployment, with Zero-Downtime. Monitoring of Microservices is provided using Prometheus and Grafana services provided as SaaS.

List of Figures

Figure 1: Current Design 1
Figure 2: Updated Microservices 2
Figure 3:Source Code Structure 4
Figure 4 : Sample Service YAML 5
Figure 5 : Ingress YAML file 5
Figure 6: ingress-nginx-controller 6
Figure 7 : Build Stages 7
Figure 8 : Helm Chart Dependency 8
Figure 9 : Service Type as External Name 8
Figure 10 : MongoDB ExternalName Service 8
Figure 11: Helm Chart Packaging Solution 8
Figure 12: Trivy Scanner script 9
Figure 13 : Cypress E2e Testing pipeline config 9
Figure 14: Sample test report 10
Figure 15 : Deployment Pipeline high-level view 10
Figure 16 : Blue Green Deployment High Level View 11
Figure 17 : MongoDB deployment with secrets 12
Figure 18: Mongo Secrets yaml 12
Figure 19: Cluster Role and Role Binding for mongo secrets 13
Figure 20: Add Azure Managed Grafana 13
Figure 21: Grafana Kubernetes Dashboard 14

List of Tables

Table 1: Build Pipelines 7
Table 2 : Deployment Pipeline 12
Table 3 : Blue Green Traffic Pipelines 12

References

Andrew Block, A. D., JUNE 2020. *Learn Helm.* s.l.:PACKT.

Leszko, R., 2017. *Continuous Delivery with Docker and Jenkins.* s.l.:Packt.

McKendrick, N. K. a. R., 2022. *The Kubernetes Bible: The Definitive Guide to Deploying and Managing Kubernetes Across Major Cloud Platforms.* s.l.:Packt.

Raul, A., 2021. *Cloud Native with Kubernetes: Deploy, configure, and run modern cloud native applications on Kubernetes.* s.l.:Packt.