DOME

(Distributed Open Marketplace for Europe)

D4.1 Real deployment scenarios and DOME validation (v1)

Project full title

A Distributed Open Marketplace for Europe Cloud and Edge Services

Contract No.

101084071

Strategic Objective

DIGITAL-2021-CLOUD-AI-01-DS-MARKETPLACE-CLOUD

Project Document Number

DOME-D4.1

Project Document Date

4.7.2024

Deliverable Type and Security

DEM - PU

Main editor

DEMX

Contributors

ION, ACK, BEIA

This project has received funding from the European Union's Digital Europe Programme under grant agreement No 101084071.

Log Table

Version	Date	Change	Author/partner
V01	26/05/2024	DEMX contribution	Demetrix
V02	25/06/2024	DEMX Update	Demetrix
V02	03/07/2024	CYFRONET contribution	Cyfronet
V02	04/07/2024	IONOS revision	Ionos
VFinal	04/07/2024	Final check for submission	ENG

Table of Contents

List of figures

Acronyms

Acronym	Definition
DOME	Distributed Open Marketplace for Europe
AI	Artificial Intelligence
CA	Consortium Agreement
CI/CD	Continuous Integration/Continuous Deployment
CSP	Cloud Service Provider
DSA	Digital Services Act
IaaS	Infrastructure-as-a-Service
IAM	Identity and Access Management
IPR	Intellectual Property Rights
ISSP	Information Society Service Provider
OSS	Open-Source Software
PaaS	Platform-as-a-Service
BAE	Business API Ecosystem
DLT	Distributed Ledger Technology
PEP/PDP	Policy Enforcement Point/Policy Decision Point
API	Application Programming Interface
JSON	JavaScript Object Notation
UUID	Universally Unique Identifier
SLA	Service Level Agreement
VC	Verifiable Credential
VP	Verifiable Presentation

K8s	Kubernetes
NGINX	Engine-X (web server, reverse proxy server)
DBaaS	Database as a Service
S3	Simple Storage Service
SSD	Solid-State Drive
HDD	Hard Disk Drive
ACME	Automated Certificate Management Environment
IP	Internet Protocol
DNS	Domain Name System

1 Introduction

1.1 Executive Summary

Cloud computing is identified as a central piece of Europe's digital future, giving European businesses and public organizations the data processing technology required to support their digital transformation.

The European Commission thereby stepped up its efforts to support cloud uptake in Europe as part of its strategy, notably with the pledge to facilitate "the set-up of a cloud services marketplace for EU users from the private and public sector". DOME will materialize the envisioned online marketplace, providing the means for accessing trusted services, notably cloud and edge services, building blocks deployed under the Common Services Platform and more generally any software and data processing services developed under EU programmes such as the Digital Europe Programme, Horizon 2020 or Horizon Europe Relying on Gaia-X concepts and open standards, DOME will provide the finishing touch to the technical building that the Digital Europe Program is creating for boosting the development and adoption of trusted Cloud and Edge services in Europe. It will provide the single point for enabling customers and service providers to meet each other in a trustful manner.

DOME will take the form of a federated collection of marketplaces connected to a shared digital catalog of cloud and edge services. Each of the federated marketplaces will be independent or connected to the offering of a given cloud providers which, in turn, can be classified as cloud IaaS providers or cloud platform providers (each of which provide a platform targeted to solve the integration of vertical data/application services from a given vertical domain, like smart cities or smart farming, or the integration of certain type of data/application services, e.g., AI services). DOME will rely on the adoption of common open standards for the description of cloud and edge services and service offerings as well as their access through a shared catalog.

This document describes the real deployment scenarios and validation processes for the DOME Marketplace. It outlines the technological stack, architecture, functionalities, and the on-boarding process for service providers. Furthermore, it details the environments used in deployment, the DevOps methodologies applied, and the operations and maintenance procedures. The aim is to provide a comprehensive guide that ensures a seamless setup and operational management of the DOME Marketplace, promoting a reliable and efficient digital marketplace for cloud and edge services across Europe.

1.2 Intended audience

This document is meant to provide valuable information, guidance, and references to:

- **Owner of the DOME Marketplace (DOME Operator)**: Individuals or entities responsible for managing the DOME Marketplace, who need to gather all required software artifacts and documentation to initiate the deployment of the DOME software platform.
- **DOME Project Partners**: Particularly federated marketplaces and providers who are part of the DOME project and are looking to deploy and run local instances of the DOME platform for development and testing purposes.
- **Potential Federated Marketplaces and Providers Outside the Consortium**: Organizations or individuals outside the DOME consortium who wish to gain knowledge about the open-source technical foundation of the DOME platform and consider deploying their own instances.

This document aims to equip each group with the necessary information and guidelines to ensure the successful deployment, integration, and management of the DOME Marketplace.

1.3 Structure of the document

This document is divided into four main chapters, each addressing different aspects of the DOME Marketplace deployment and validation:

- **Chapter 1: Introduction** - Provides a summary of the document, intended audience, and an overview of its structure and related resources.
- **Chapter 2: DOME Marketplace Demonstrator** - Describes the purpose, architecture, core functionalities, and the onboarding process for service providers. It includes detailed information on the initial setup, registration, account creation, catalog integration, provider dashboard, and available documentation and support.
- **Chapter 3: Deployment** - Details the technological stack used for the DOME Marketplace, including GitHub Repository, GitHub Actions, ArgoCD, and Helm. It also covers the different environments (Sandbox, Integration, Production) and their management, as well as the DevOps methodologies applied.
- **Chapter 4: Operations and Maintenance** - Discusses the tools and processes for the ongoing operations and maintenance of the DOME Marketplace. It includes guidelines for setup, cluster creation, GitOps setup, and various tools like ArgoCD, Helm, and Cert-Manager.

Each chapter is designed to give a comprehensive understanding of the respective topics, ensuring that the reader has all the necessary information to successfully deploy and manage the DOME Marketplace.

1.4 Related documents and resources

Following is a list of valuable links to relevant documents and resources:

- EU Digital Identity Wallet Architecture and Reference Framework
- DSBA Technology Convergence: Discussion Document
- Handbook for Roll-Out and Operational Environments
- DOME D3.2 - DOME Technical Infrastructure (v1)[1]

[1] Public deliverables and other resources of the project will be available in the DOME website https://dome-marketplace.eu/resources/

2 Marketplace Demonstrator

2.1 Overview

Purpose

The DOME Marketplace Demonstrator aims to illustrate the core functionalities and capabilities of the DOME Marketplace. It serves as a practical example to showcase how the marketplace operates, highlighting its potential benefits for service providers and end-users. The demonstrator is designed to provide a comprehensive understanding of the marketplace's architecture, functionalities, and onboarding process.

Target Audience

The primary users of the DOME Marketplace Demonstrator include:

- **Service Providers**: Entities looking to offer their cloud and edge services through the marketplace.
- **End-Users**: Customers and businesses that will utilize the services aggregated within the DOME Marketplace.
- **Developers and Technical Teams**: Individuals responsible for integrating services and ensuring the smooth operation of the marketplace.

2.2 Architecture

The architecture of the DOME Marketplace is based on a federated model, integrating multiple independent marketplaces into a unified digital catalog. This structure promotes decentralization and ensures scalability and interoperability across various cloud and edge services.

Key Subsystems:

- **Trust and Identity and Access Management (IAM) Framework**: Ensures secure operations through authentication, authorization, and digital identity management using Verifiable Credentials (VCs) and Verifiable Presentations (VPs).
- **Decentralized Persistence Layer**: Manages data storage and persistence for transactional data, service catalogs, and ledgers.
- **Central Marketplace and Value-added Services**: Acts as the main interface for service listing and procurement, enhanced by additional functionalities such as certification, payment systems, and customer support.

2.3 Functionalities

Core Features

The core functionalities of the DOME Marketplace include:

- **Service Registration and Listing**: Allows service providers to register and list their services within the marketplace.
- **Service Procurement**: Enables end-users to browse, select, and procure services through a user-friendly interface.
- **Catalog Management**: Facilitates the management of service offerings, including updates and modifications.
- **Transaction Management**: Ensures secure and transparent transactions using blockchain technology.
- **Customer Support**: Provides AI-powered support systems, including chatbots and ticketing for enhanced user experience.

User Interface

The user interface of the DOME Marketplace is designed to be intuitive and user-friendly, ensuring seamless interactions for both service providers and end-users. Key interactions and workflows are streamlined to enhance user experience and operational efficiency.

2.4 On Boarding process

Initial Setup for Providers

The onboarding process for service providers involves several key steps to ensure a smooth integration into the DOME Marketplace:

Legal Onboarding Procedure

1. **Initial Contact and Registration:**
    - Interested companies initiate contact via the DOME marketplace portal.
    - Register an account and submit a preliminary application for initial review.
2. **Submission of Documentation:**
    - Provide required legal documentation, including company registration, proof of identity, and relevant licenses.
    - Complete and submit the DOME onboarding form, detailing the services to be offered.
3. **Verification and Approval:**
    - The DOME onboarding team reviews the submitted documents.
    - Additional information or clarifications may be requested as needed.
    - Upon successful verification, the company is approved to proceed to the next step.
4. **Service Listing Setup:**
    - Approved companies set up their service listings within the DOME marketplace.

    - Ensure that all service details, pricing, and terms of service are accurately represented.
5. **Final Review and Activation:**
    - The onboarding team conducts a final review of the service listings to ensure compliance.
    - Once approved, the services are made live on the marketplace and available to users.

Technical Onboarding Procedure

1. **Initial Consultation:**
    - Technical partners and service providers engage in an initial consultation with the DOME technical team.
    - Discuss integration requirements, expectations, and technical specifications.
2. **Provisioning of Access Credentials:**
    - Partners receive necessary access credentials, such as API keys, LEAR credential, sandbox environment access, and configuration files.
3. **Component Integration:**
    - Follow the integration guide to configure and integrate technical components into the DOME environment.
    - Utilize provided documentation, such as the DOME Service Provider Guide and the DOME GitOps Integration Documentation.
4. **Testing and Validation:**
    - Conduct thorough testing within the sandbox environment.
    - Validate the integration to ensure that compatibility and performance standards are met.
5. **Deployment and Monitoring:**
    - Upon successful validation, deploy the integrated components into the production environment.
    - Continuous monitoring and support are provided to ensure operational stability .

Documentation and Support

Comprehensive documentation, FAQs, and integration guides are available to assist providers. Support channels include:

- Technical support;
- Webinars and workshops;
- Personalized onboarding sessions.

3 Deployment

3.1 Technological stack

GitHub Repository

Code has been managed and stored using Github: Github is an online developer platform that allows developers to create, store, manage and share their code. It is based on GIT software for code management and version control, and offers, on top of this, access control, bug tracking, software feature requests, task management, continuous integration and wikis for every project.

Github use is widespread over developers community, commonly for open source projects. It has been chosen for DOME for of the following features:

- GIT usage: GIT is widespread as code versioning tool, and is known and used by the vast majority of software developers;
- CI/CD features: GitHub Actions (described below) allows implementation of CI workflows, and it's been used as the basis for Gitops approach;

GitHub Actions

GitHub Actions is a feature offered by GitHub that enables CI/CD workflows for building, testing, and deploying applications starting from the code stored on Github.

It allows Developers, Devops and Gitops to define custom automated pipelines for the various stages of software build, test, release and deployment.

Pipelines are easily defined through yaml files that allow jobs and steps definition. Pipeline definitions can be inherited by other pipelines allowing reuse of jobs and steps.

Github Actions has been chosen for the following reasons:

- It is fully integrated with Github;
- It provides a wide library of templates for various common steps (such as Git checkout and common kubectl actions). These templates are ready to use and allow yaml files complexity reduction, bringing developers to focus on custom steps;

ArgoCD

Argo CD is a declarative, GitOps continuous delivery tool for Kubernetes.

Argo CD follows the GitOps pattern of using Git repositories as the source of truth for defining the desired application state. Kubernetes manifests can be specified in several ways:

- customize applications
- helm charts
- jsonnet files
- Plain directory of YAML/json manifests

Any custom config management tool configured as a config management plugin

Argo CD automates the deployment of the desired application states in the specified target environments. Application deployments can track updates to branches, tags, or pinned to a specific version of manifests at a Git commit.

Argo CD has been chosen for the following reasons:

- It has full compatibility with Kubernetes;
- It allows several ways to specify Kubernetes manifests, allowing applications developers to define their manifest as they prefer;
- It has a nice graphical interface that allows Gitops managers to easily monitor and act on deployed applications;

Helm

Helm is a package manager for Kubernetes.applications. Helm Charts help Devops define, install, and upgrade even the most complex Kubernetes application.

Charts are easy to create, version, share, and publish so they allow a huge simplification of all of the processes tied to application deployment.

Helm has been chosen as the preferred way to deploy DOME applications for the following reasons:

- It allows complexity management: Charts describe even the most complex apps, provide repeatable application installation, and serve as a single point of authority;
- Updates are easier with in-place upgrades and custom hooks;
- Sharing is simple, as Charts are easy to version, share, and host on public or private servers;
- Rollbacks to older versions are simplified with the helm rollback functionality;

3.2 Environments

Environment Management

In DOME Gitops has been applied a clear separation of duties and purposes for each environment.

This has been done with the implementation of a consistent deployment pipeline across all environments.

The environment management procedure relies on a regular synchronization of integration and production environments to reflect the latest changes.

Comprehensive backup and rollback strategies have been implemented to maintain stability.

Environments are represented in the Github repository as different folders on the same Git branch, instead of relying on a different branch for each environment: this choice has been done following the best practices indicated here:

- https://codefresh.io/blog/stop-using-branches-deploying-different-gitops-environments/
- https://codefresh.io/blog/how-to-model-your-gitops-environments-and-promote-releases-between-them/

The main advantages of this approach are the following:

- Promotion between different environments on different branches, using pull requests and merges between different branches is problematic. On the other side, promotion between different environments on the same branch are easy as a copy-and-paste;
- On a different branch per environment approach people are tempted to include environment specific code and create configuration drift. This is avoidable on single branch approach, especially if non-environment-dependant code is factored outside of the env folders;
- As soon as you have a large number of environments, maintenance of all environments on different branches gets quickly out of hand. With a single branch approach this problem is mitigated;
- The branch-per-environment model goes against the existing Kubernetes ecosystem – see GitOps Kubernetes for more info;

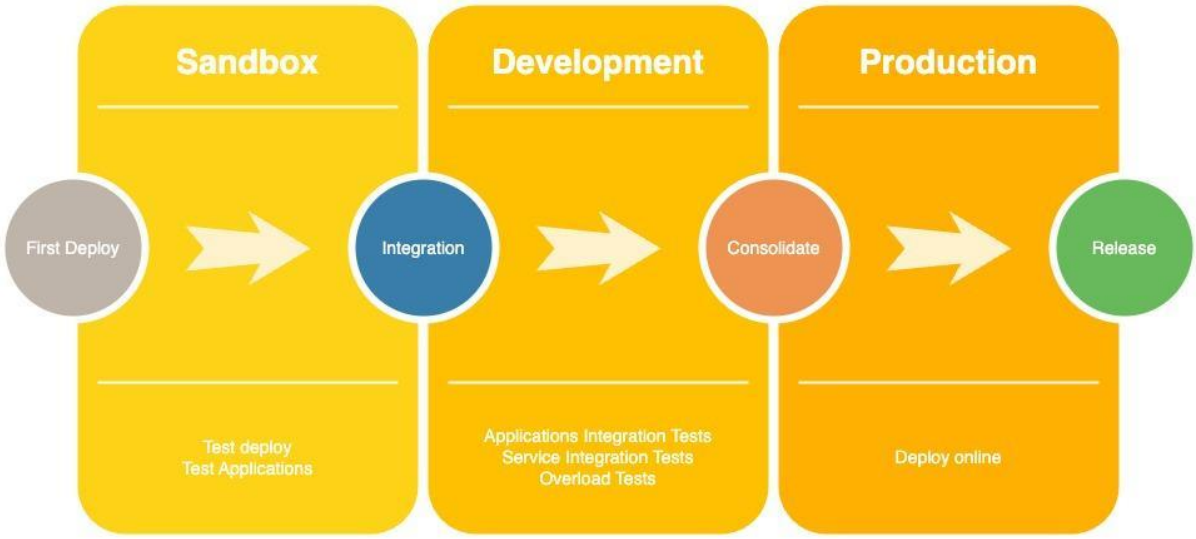

Fig. 1 - DOME processing deployment

Sandbox Environment

A sandbox environment acts as a playground for DevOps. It is a dedicated space used for experimenting with new tools, configurations, and technologies, making it an ideal and safe environment for testing scripts and automation processes without impacting other critical environments.

One of the primary purposes of a sandbox environment is to facilitate innovation and continuous improvement. In this environment, developers and DevOps engineers can try out new software, updates, or configuration changes without the risk of causing disruptions or issues in live production systems. This isolation ensures that any potential problems or bugs can be identified and resolved early in the development process.

Furthermore, a sandbox environment is crucial for testing the integration of new DevOps tools and practices before porting to the Dev environment for integration testing. For instance, teams can simulate deployments, validate the functionality of CI/CD pipelines, and test the effectiveness of monitoring and configuration management tools. This allows for thorough validation and optimization of processes before they are implemented in production environments.

Additionally, sandbox environments are invaluable for training and onboarding new team members. New hires can familiarize themselves with the existing infrastructure, learn best practices, and gain hands-on experience in a risk-free setting. This practical exposure accelerates the onboarding process and ensures that new team members are well-prepared to contribute effectively to production environments.

In summary, a sandbox environment is essential for fostering experimentation, innovation, and continuous learning within DOME DevOps teams. It provides a safe and isolated space for testing and development, helping to maintain the stability and integrity of production environments while promoting the adoption of new tools and practices.

Integration Environment

An integration environment is specifically dedicated to testing the integration of new components within a system. This environment plays a crucial role in the development process as it is used to verify updates, changes, and new features before they reach the production stage. By ensuring compatibility and stability, the integration environment helps maintain the integrity of existing systems while incorporating new elements.

One of the primary functions of the integration environment is to validate that new components work seamlessly with the current infrastructure. This includes checking for any conflicts, ensuring that data flows correctly between systems, and verifying that all integrated components perform as expected under various scenarios. This rigorous testing helps to identify and address any issues that may arise from new updates or changes.

The integration environment is critical for catching and resolving issues early in the development cycle. By thoroughly testing in this dedicated space, teams can detect potential problems before they impact the production environment, reducing the risk of downtime, bugs, and other issues that could affect end-users. This proactive approach to quality assurance ensures that only stable and compatible updates are deployed to production.

Moreover, the integration environment supports continuous integration practices, allowing developers to frequently merge code changes into a shared repository. Automated tests run in this environment help to identify integration issues quickly, facilitating smoother and more reliable code integration. This practice not only enhances collaboration among team members but also accelerates the development process by providing immediate feedback on the integration status.

In summary, an integration environment is essential for verifying the compatibility and stability of new components with existing systems. Its usage is critical for early issue detection and resolution, ensuring that updates and changes do not compromise the production environment. By supporting continuous integration and rigorous testing, the integration environment contributes significantly to the overall quality and reliability of the software development lifecycle.

Production Environment

The production environment is the live environment where the application is accessible to end-users. This environment represents the final stage in the deployment pipeline, where the software operates in real-world conditions and serves actual customers. As such, it is designed to be highly stable and optimized for peak performance and stringent security.

Given the critical nature of the production environment, it undergoes rigorous preparation and configuration to ensure it meets the highest standards of reliability and efficiency. Performance tuning is a key focus, with careful attention given to resource allocation, load balancing, and response times to ensure a seamless user experience. Security measures are also paramount, including the implementation of robust authentication, encryption, and intrusion detection systems to protect sensitive data and prevent unauthorized access.

In the production environment, changes are applied very cautiously and minimally to avoid disruptions. Any updates or modifications undergo thorough testing and validation in lower environments, such as staging or integration environments, before being deployed to production. This careful approach minimizes the risk of introducing bugs or issues that could impact the end-users.

Continuous monitoring is a fundamental aspect of managing the production environment. Advanced monitoring tools and practices are employed to track various metrics, including system performance, uptime, error rates, and user activity. This real-time monitoring allows for the rapid detection and resolution of any issues that may arise, ensuring the application remains available and reliable at all times.

In summary, the production environment is the live, end-user-facing environment that prioritizes stability, performance, and security. Changes are applied minimally and cautiously to maintain service continuity, and continuous monitoring is conducted to ensure high availability and reliability. This meticulous approach ensures that the application delivers a consistent and dependable experience to its users.

3.3 DevOps Methodologies

The integration pipeline serves as the backbone of the CI/CD workflow, connecting code repositories with testing environments and deployment mechanisms. Its primary objective is to enhance collaboration among development and operations teams by automating the integration and validation of code changes, thereby reducing the risk of errors and ensuring the consistent delivery of reliable software. Fig. X below depicts the overall approach to the integration for the DOME platform:

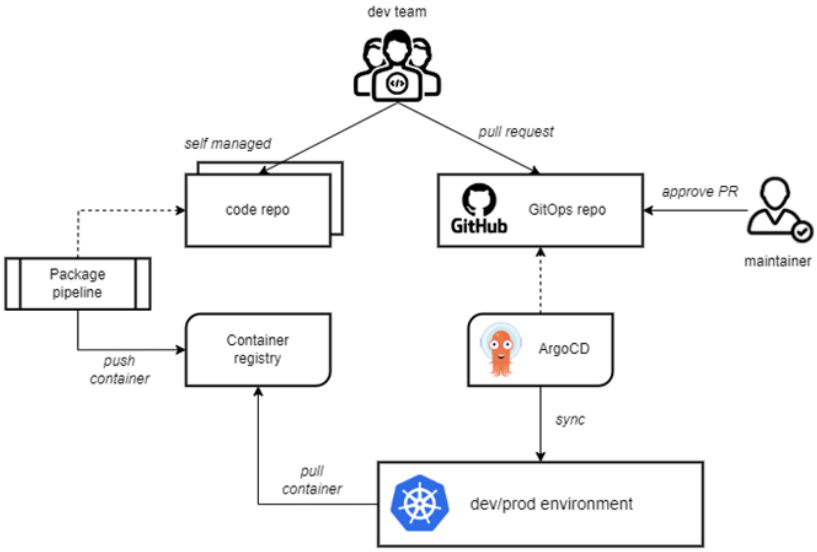

Fig. 2 - DOME Kubernetes Pipeline

In the integration approach designed for DOME, the team intending to integrate its component needs to adhere to a structured set of steps:

1. Fork the repository and create a new branch
2. Add component manifest files
3. Add ArgoCD application
4. Create a Pull Request and wait for merge

The following paragraphs detail the steps to integrate a new application into DOME.

Configuration of kubectl to access the remote cluster

Before starting the integration process, you need to configure **kubectl** in order to access the remote cluster locally; to do this, follow the steps below:

1. Request the cluster administrator to create a service account, providing the name of your organization and the namespace within which your application will be deployed. You will be provided with a configuration file to access the cluster
2. Modify the **KUBECONFIG** environment variable by adding the path to the configuration file
3. Verify that the new context has been added by executing the following command:

```
kubectl config get-contexts
```

4. Switch the context by executing the following command:

```
kubectl config use-context <name of the context>
```

At this point, you should be able to run kubectl commands on the remote cluster.

Notes

- Ensure that the username of the new context has not already been used in other contexts; if so, modify the user name in the configuration file.
- The **service account** provided to you will have **limited permissions**; you will be able to view all resources within your namespace but will only have write access to secrets.

Fork the repository and create a new branch

Team members begin by forking the main GitOps repository and creating a new branch for their specific integration work. This facilitates isolated development and changes.

After forking and subsequently cloning the project, navigate to the project directory to proceed with the rest of the guide.

Repository structure

The GitOps repository has the following structure:

```
applications/
   └── app_1.yaml
   └── app_2.yaml
   . . .
applications_dev/
   └── app_1.yaml
   └── app_2.yaml
```

```
ionos/
   └── app_1/
         └── Chart.yaml
         └── values.yaml
         . . .
ionos_dev/
   └── app_1/
         └── Chart.yaml
         └── values.yaml
         . . .
```

For each environment, two directories are defined:

- `applications_<env>`: it will host the environment-specific Argocd applications (see Add ArgoCD application)
- `ionos_<env>`: it will host application manifest files (see Add component manifest files)

N.B. The **`applications`** and **`ionos`** directories are currently **reserved** for the demo environment. For the continuation of the guide, we will use the “dev” environment, which serves as a playground where teams can test the integration of their components within the DOME ecosystem.

Add component manifest files

The team incorporates their integration by adding either a Helm chart or plain Kubernetes manifests to a properly named folder under the designated directory. This directory serves as a centralised location for all integrations.

First, create a directory for your application by executing the following commands:

```
cd ionos_dev/
mkdir <name of you application>
```

Then put your application manifest files or Helm charts inside this directory. Once done, the structure should look like the following:

```
ionos_dev/
   └── your-app/
         └── deployment.yaml
         └── service.yaml
         └── ingress.yaml
         └── secret.yaml
         └── configmap.yaml
```

or, if you are using Helm:

```
ionos_dev/
   └── your-app/
         └── templates/
               └── secret.yaml
               └── configmap.yaml
         └── Chart.yaml
         └── values.yaml
```

You don't have manifest files nor Helm charts? Follow this guide.

Add secrets

Using GitOps, means every deployed resource is represented in a git-repository. While this is not a problem for most resources, secrets need to be handled differently. We use the bitnami/sealed-secrets project for that.

To encrypt secrets from your local machine to the remote cluster, it is necessary to install kubeseal. Once installed, you can create an encrypted secret as follows:

1. Create a plain secret manifest file named `<secret name>-plain-secret.yaml` (**IMPORTANT**: Make sure the file name ends with *-plain-secret.yaml* so it will be ignored when pushing to repository)

   ```
   apiVersion: v1
   kind: Secret
   metadata:
     name: <secret name>
     namespace: <app namespace>
   data:
     <key>: <base64 encoded value>
   ```

2. Seal the secret by running the following command:

   ```
   kubeseal -f <secret name>-plain-secret.yaml -w <secret name>-sealed-
   secret.yaml --controller-namespace sealed-secrets  --controller-name
   sealed-secrets
   ```

   Or using the script SealSecret:

   Windows PowerShell

   ```
   .\scripts\SealSecret.ps1 -secretPath <path to plain secret>
   ```

   Shell

```
# chmod +x ./scripts/SealSecret.sh
./scripts/SealSecret.sh <path to plain secret>
```

Note: If you are using Helm charts, make sure to place the sealed secret file under the directory *your-app/templates*.

Add ArgoCD application

Now that the manifest files for your component are ready, the next step is to create the application for ArgoCD.

First, navigate to the "application" directory:

```
cd applications_dev/
```

and create a file name *your-app.yaml* with the following content:

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: <app name>
  namespace: argocd
  labels:
    purpose: <app purpose>
spec:
  destination:
    namespace: <app namespace>
    server: https://kubernetes.default.svc
  project: default
  source:
    path: ionos_dev/<app name>
    repoURL: https://github.com/DOME-Marketplace/dome-gitops
    targetRevision: HEAD
  syncPolicy:
    automated:
      prune: true
      selfHeal: true
```

If your application is complex and consists of multiple components but you still want to manage it as a single ArgoCD application, you can use the following approach:

1. During the step "Add component manifest files", create a directory for each sub-component (i.e. *ionos_dev/your-app/sub-component-1*, *ionos_dev/your-app/sub-component-2* etc)
2. Create the directory *applications/your-app*

3. Inside *applications_dev/your-app*, create an ArgoCD application file for each sub-component, ensuring that it points to the respective subfolder under *ionos_dev/your-app*

```
source:
  path: ionos_dev/<app name>/<sub-component name>
  repoURL: https://github.com/DOME-Marketplace/dome-gitops
  targetRevision: HEAD
```

4. Create an ArgoCD application file for your entire application and make it point to applications/your-app

```
source:
  path: applications_dev/<app name>
  repoURL: https://github.com/DOME-Marketplace/dome-gitops
  targetRevision: HEAD
```

You can use the `marketplace` application or `dome-trust` as a reference.

Create a Pull Request

Upon completing the changes, the team initiates a pull request from their branch to the main one. This PR serves as a formal request for the integration to be reviewed and merged. Team members must wait for the review process to be completed.

Once the pull request is approved and merged into the main branch, the GitOps pipeline automatically triggers the deployment process. This involves synchronizing the desired state of the cluster with the changes introduced in the merged pull request. The deployment is executed based on the Helm chart or manifest configurations added to the repository.

Create K8s manifest files for your application

This guide explains how to create the manifest files necessary to deploy your application on a Kubernetes cluster.

For demonstration purposes, we will use a sample docker-compose file describing a simple application with a MySql database and convert its services into proper manifest files.

```
version: '3'
services:
  mysql:
    image: mysql:8.2
    ports:
      - "3406:3306"
```

```
    environment:
      MYSQL_ROOT_PASSWORD: secret
      MYSQL_DATABASE: my-app-db
      MYSQL_USER: username
      MYSQL_PASSWORD: secret
    volumes:
      - 'mysql-data:/var/lib/mysql'
  my-app:
    image: registry/my-app:1.0.0
    ports:
      - "8080:8080"
    environment:
      SERVER_PORT: 8080
      DB_HOST: mysql:3306
      DB_DATABASE: bookstack
      DB_USERNAME: my-app-db
      DB_PASSWORD: secret
    volumes:
      - 'app-data:/home/app/data'
volumes:
  mysql-data:
  app-data:
```

Use available Helm Charts whenever possible

If your application relies on a database or other third-party software, you should use their corresponding Helm charts. In this repository, you can find several examples of using Helm charts to deploy databases and other tools. Here are some examples:

- MySql
- MongoDb
- Kafka

N.B. Remember to add required secrets within the templates folder

Manifest files

You need to create the following resource files for each service in your docker-compose. So the first step is to create the folder structure for your application within the proper environment directory.

Here is the folder structure for the demo application:

```
ionos_<env>/
  └── app-name/
    └── mysql/
      └── templates/
      └── Chart.yaml
      ...
    └── my-app/
      └── deployment.yaml
   …
```

N.B. The resources described below are those required for deploying the sample application. Other applications may require additional resources or may not need all those described.

Deployment

A Deployment provides declarative updates for Pods and ReplicaSets. You describe a desired state in a Deployment, and the Deployment Controller changes the actual state to the desired state.

Here is the deployment file for the demo application:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
  namespace: my-app
  labels:
    app.kubernetes.io/instance: my-app
    app.kubernetes.io/name: my-app
spec:
  replicas: 1
  selector:
    matchLabels:
      app.kubernetes.io/instance: my-app
      app.kubernetes.io/name: my-app
  template:
    metadata:
      labels:
        app.kubernetes.io/instance: my-app
```

```
    app.kubernetes.io/name: my-app
spec:
  containers:
    - name: my-app
      image: registry/my-app:1.0.0
      env:
        - name: SERVER_PORT
          value: 8080
        - name: DB_HOST
          value: "<name of mysql service>:3306"
        - name: DB_DATABASE
          value: my-app-db
        - name: DB_USERNAME
          value: username
        - name: DB_PASSWORD
          valueFrom:
            secretKeyRef:
              name: my-app-secret
              key: DB_PASSWORD
      ports:
        - containerPort: 8080
          name: http
          protocol: TCP
      volumeMounts:
        - mountPath: /home/app/data
          name: my-app-storage
  volumes:
  - name: my-app-storage
    persistentVolumeClaim:
      claimName: my-app-pvc
```

Notes

- The `<name of MySql service>` should be set in the corresponding Helm Chart within the values file
- The value of `claimname` must match the one used in the Persistent Volume Claim file.

Persistent Volume Claim

A **PersistentVolumeClaim** (**PVC**) is a request for storage by a user. Claims can request specific size and access modes.

Here is the PVC file for the demo application:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-app-pvc
  namespace: my-app
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
```

Service

A Service is a method for exposing a network application that is running as one or more Pods in your cluster.

Here is the service file for the demo application:

```
apiVersion: v1
kind: Service
metadata:
  name: my-app
  namespace: my-app
spec:
  type: ClusterIP
  selector:
    app.kubernetes.io/instance: my-app
    app.kubernetes.io/name: my-app
  ports:
    - name: http
      port: 8080
      protocol: TCP
      targetPort: 8080
```

Ingress

A Ingress is an API object that manages external access to the services in a cluster, typically HTTP. Ingress may provide load balancing, SSL termination and name-based virtual hosting.

Here is the ingress file for the demo application:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-app-ingress
  namespace: my-app
  annotations:
    cert-manager.io/cluster-issuer: letsencrypt-dev-issuer
    nginx.ingress.kubernetes.io/ssl-passthrough: "true"
    nginx.ingress.kubernetes.io/backend-protocol: "HTTP"
spec:
  ingressClassName: nginx
  rules:
  - host: my-app.dome-marketplace-dev.org
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: my-app
            port:
              number: 8080
  tls:
  - hosts:
    - my-app.dome-marketplace-dev.org
    secretName: my-app-tls-secret
```

Notes

- Hosts are automatically created and managed by External-DNS
- The value of `backend.service.name` must match the name of the Service
- The value of `backend.service.port.number` must match the one defined in the Service

Secret

A Secret is an object that contains a small amount of sensitive data such as a password, a token, or a key. Using a Secret means that you don't need to include confidential data in your application code.

Here is the secret file for the demo application:

```
apiVersion: v1
kind: Secret
metadata:
  name: my-app-secret
  namespace: my-app
stringData:
  DB_PASSWORD: "secret"
```

N.B. Remember, ***plain secrets should never be pushed to the repository***. You should first encrypt them using SealedSecret as described here.

4 Operations and Maintenance

4.1 Tools

Setup

The GitOps approach aims for a maximum of automation and will allow to reproduce the full setup. For more information about GitOps, see:

- RedHat Pros and Cons - Link
- ArgoCD - https://argo-cd.readthedocs.io/en/stable/
- FluxCD - https://fluxcd.io/

Preparation

In order to set up the DOME-Marketplace, its recommended to install the following tools before starting

- ionosctl-cli to interact with the Ionos-APIs
- jq as a json-processor to ease the work with the client outputs
- kubectl for debugging and inspecting the resources in the cluster
- kubeseal for sealing secrets using asymmetric cryptography

Cluster creation

Execute the following steps:

1. In order to create the cluster, login to Ionos:

```
ionosctl login
```

2. A Datacenter as the logical bracket around the nodes in your cluster has to be created:

```
export DOME_DATACENTER_ID=$(ionosctl datacenter create --name DOME-Production -o json | jq -r '.items[0].id')

# wait for the datacenter to be "AVAILABLE"

watch ionosctl datacenter get -i $DOME_DATACENTER_ID

export DOME_DATACENTER_ID=db088b6a-4c61-4a47-8fe9-a4f1c5f916cc
```

3. Create the Kubernetes Cluster and wait for it to be "ACTIVE":

```
export DOME_K8S_CLUSTER_ID=$(ionosctl k8s cluster create --name DOME-Production-K8S -o json | jq -r '.items[0].id')

watch ionosctl k8s cluster get -i $DOME_K8S_CLUSTER_ID

export DOME_K8S_CLUSTER_ID=5821f7a6-e4ee-48c8-9404-741fc0ae281d
```

4. Create the initial nodepool inside your cluster and datacenter:

```
export DOME_K8S_DEFAULT_NODEPOOL_ID=$(ionosctl k8s nodepool create --cluster-id $DOME_K8S_CLUSTER_ID --name default-pool --node-count 2 --ram 32768 --storage-size 40 --datacenter-id $DOME_DATACENTER_ID --cpu-family "INTEL_SKYLAKE" -o json | jq -r '.items[0].id')

# wait for the pool to be available

watch ionosctl k8s nodepool get --nodepool-id $DOME_K8S_DEFAULT_NODEPOOL_ID --cluster-id $DOME_K8S_CLUSTER_ID

export DOME_K8S_DEFAULT_NODEPOOL_ID = 24ca1894-6b36-4e3f-9332-4bb00c3b9891
```

5. Following the recommendations from the Ionos-FAQ, we also dedicate a specific nodepool for the ingress-controller

```
export DOME_K8S_INGRESS_NODEPOOL_ID=$(ionosctl k8s nodepool create --cluster-id $DOME_K8S_CLUSTER_ID --name ingress-pool --node-count 1 --datacenter-id $DOME_DATACENTER_ID --cpu-family "INTEL_SKYLAKE" --labels nodepool=ingress -o json | jq -r '.items[0].id')

# wait for the pool to be available

watch ionosctl k8s nodepool get --nodepool-id $DOME_K8S_INGRESS_NODEPOOL_ID --cluster-id $DOME_K8S_CLUSTER_ID

export DOME_K8S_INGRESS_NODEPOOL_ID=8668accc-b521-4c57-a283-33907af85726
```

6. Retrieve the kubeconfig to access the cluster:

```
ionosctl k8s kubeconfig get --cluster-id $DOME_K8S_CLUSTER_ID > dome-k8s-config.json

# Exporting the file path to $KUBECONFIG will make it the default config for kubectl.

# If you work with multiple clusters, either extend your existing config or use the file inline with the --kubeconfig flag.

export KUBECONFIG=$(pwd)/dome-k8s-config.json
```

Gitops Setup

Even though the cluster creation was done on Ionos, the following steps apply to all Kubernetes installations(tested version is 1.26.7). It's not required to use Ionos for that.

In order to provide GitOps capabilities, we use ArgoCD. To set up the tool, we need 2 manual steps to deploy ArgoCD, as it's described by the manual.

1. Create a namespace for argocd. For easier configuration, we use argo's default argocd

   ```
   kubectl create namespace argocd
   ```

2. Deploy argocd with extensions enabled:

   ```
   kubectl apply -k ./extension/ -n argocd
   ```

From now on, every deployment should be managed by ArgoCD through Applications.

Namespaces

To properly separate concerns, the first application to be deployed will be the namespaces. It will create all namespaces defined in the ionos/namespaces folder.
Apply the application via:

```
kubectl apply -f applications/namespaces.yaml -n argocd
```

Sealed Secrets

Using GitOps, means every deployed resource is represented in a git-repository. While this is not a problem for most resources, secrets need to be handled differently. We use the bitnami/sealed-secrets project for that. It uses asymmetric cryptography for creating secrets and only decrypts them inside the cluster. The sealed-secrets controller will be the first application deployed using ArgoCD. Since we want to use the Helm-Charts and keep the values inside our git-repository, we get the problem of ArgoCD only supporting values-files inside the same repository as the chart(as of now, there is an open PR to add that functionality → PR#8322). In order to workaround that shortcoming, we are using "wrapper charts". A wrapper-chart does consist of a Chart.yaml with a dependency to the actual chart. Besides that, we have a values.yaml with our specific overrides. See the sealed-secrets folder as an example.

Apply the application via:

```
kubectl apply -f applications/sealed-secrets.yaml -n argocd
# wait for it to be SYNCED and Healthy
watch kubectl get applications -n argocd
```

Once it's deployed, secrets can be "sealed" via:

```
kubeseal -f mysecret.yaml -w mysealedsecret.yaml --controller-namespace sealed-secrets --controller-name sealed-secrets
```

Ingress controller

In order to access applications inside the cluster, an Ingress-Controller is required. We use the NGINX-Ingress-Controller here.

Apply the application via:

```
kubectl apply -f applications/ingress.yaml -n argocd
# wait for it to be SYNCED and Healthy
watch kubectl get applications -n argocd
```

External-DNS

In order to automatically create DNS entries for Ingress-Resources, External-DNS is used.

External-DNS watches the ingress objects and creates A-Records for them. To do so, it needs the ability to access the AWS APIs.

Execute instructions from file /docs/EXTERNAL_DNS_IONOS.md.md

Cert-Manager

In addition to the dns-entries, we also need valid certificates for the ingress. The certificates will be provided by Lets encrypt. To automate creation and update of the certificates, Cert-Manager is used.

1. In order to follow the ACME protocol, Cert-Manager also needs the ability to create proper DNS entries. Therefore we have to provide the AWS account used by External-DNS, too.

    i. Put key and key-id into the following template:

```
apiVersion: v1
  kind: Secret
  metadata:
    name: aws-access-key
    namespace: cert-manager
  stringData:
    key: "THE_KEY"
     keyId: "THE_KEY_ID"
```

ii. Seal the secret and commit the sealed secret. :warning: Never put the plain secret into git.

2. Update the issuer with information about the hosted zone managing your domain and commit it.

3. Apply the application:

```
kubectl apply -f applications/cert-manager.yaml -n argocd
# wait for it to be SYNCED and Healthy
watch kubectl get applications -n argocd
```

Update ArgoCD

ArgoCD provides a nice GUI and a command-line tool to support the deployments. In order for them to work properly, an ingress and auth-mechanism need to be configured.

Ingress

Since ArgoCD is already running, we also use it to extend itself, by just providing an ingress-resource pointing towards its server. That way, we will get a proper URL automatically through the previously configured External-DNS and Cert-Manager.

Auth

To seamlessly use ArgoCD, we use Githubs Oauth to manage users for ArgoCd together with those accessing the repo.

1. Register ArgoCd in Github, following the documentation
2. Put the secret into:

```
apiVersion: v1
kind: Secret
metadata:
labels:
  app.kubernetes.io/part-of: argocd
name: github-secret
namespace: argocd
stringData:
clientSecret: THE_CLIENT_SECRET
```

3. Seal and commit it.
4. Configure the organizations to be allowed in the configmap
5. Configure user-roles in the rbac-configmap
6. Apply the application

```
kubectl apply -f applications/argocd.yaml -n argocd
# wait for it to be SYNCED and Healthy
watch kubectl get applications -n argocd
```

Login to our ArgoCD.

Deploy a new application

In order to deploy a new application, follow the steps:

1. Fork the repository and create a new branch.
2. (OPTIONAL) Add a new namespace to the namespaces
3. Add either the helm-chart(see External-DNS as an example) or the plain manifests(see ArgoCD as an example) to a properly named folder under /ionos
4. Add your application to the /applications folder.
5. Create a PR and wait for it to be merged. The application will be automatically deployed afterwards.

4.2 Monitoring

The monitoring stack consists of:

- Prometheus - open-source monitoring solution with metrics and alerting -
- Loki - a horizontally-scalable, highly-available, multi-tenant log aggregation system inspired by Prometheus
- Grafana - a multi-platform open-source analytics and interactive visualisation web application

additionally, for monitoring security threats, we use:

- Falco - a cloud-native runtime security tool for Linux operating systems

The examples in this section will use DOME prod environment manifests, depending on the environment the manifests differ only by “prd/dev/sbx” suffix in the directory paths.

Deployment

The deployment of all the monitoring tools is done in accordance with GitOps practices described in this document. For each of the DOME environments, there is an argocd “Monitoring” Application that bootstraps the monitoring stack:

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: monitoring
  namespace: argocd
  labels:
    purpose: monitoring
spec:
  destination:
    namespace: argocd
    server: https://kubernetes.default.svc
  project: default
  source:
    path: applications_prd/monitoring
    repoURL: https://github.com/DOME-Marketplace/dome-gitops
    targetRevision: HEAD
```

```
  syncPolicy:
    automated:
      prune: true
      selfHeal: true
```

From the `applications_prd/monitoring` directory, the argocd Applications manifests of the monitoring tools are ingested.

Prometheus-Grafana Deployment

To deploy Prometheus and Grafana we use kube-prometheus-stack Helm chart and configure it using values.yaml. The argocd Application manifest for kube-prometheus-stack:

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: kube-prometheus-stack
  namespace: argocd
  finalizers:
  - resources-finalizer.argocd.argoproj.io
spec:
  project: default
  sources:
  - repoURL: https://prometheus-community.github.io/helm-charts
    targetRevision: 60.2.0
    chart: kube-prometheus-stack
    helm:
      releaseName: kube-prometheus-stack
      valueFiles:
      - $values/ionos_prd/kube-prometheus-stack/values.yaml
  - repoURL: https://github.com/DOME-Marketplace/dome-gitops
    targetRevision: HEAD
    ref: values
  - repoURL: https://github.com/DOME-Marketplace/dome-gitops
    targetRevision: HEAD
    path: ionos_prd/kube-prometheus-stack
```

```
    directory:
      include:     '{kube-prometheus-stack-sso-secret.yaml,kube-prometheus-
stack-discord-webhook-infrastructure-secret.yaml,standard-cluster-
monitoring-recording-rules.yaml,onzack-namespace-monitoring-recording-
rules.yaml}'
  - repoURL: https://github.com/DOME-Marketplace/dome-gitops
    targetRevision: HEAD
    path: ionos_common/grafana-dashboards
  - repoURL: https://github.com/DOME-Marketplace/dome-gitops
    targetRevision: HEAD
    path: ionos_prd/grafana-dashboards
  destination:
    namespace: kube-prometheus-stack
    server: https://kubernetes.default.svc
  syncPolicy:
    automated:
      prune: true
      selfHeal: true
    syncOptions:
    - CreateNamespace=true
    -                                                    ServerSideApply=true
```

The manifest specifies multiple sources:

- `ionos_common/grafana-dashboards` directory contains definitions of Grafana dashboard common for all DOME environments
- `ionos_prd/grafana-dashboards` directory contains definitions of Grafana dashboard specific to prod environment

The collection of static manifests is added as a separate source:

- `kube-prometheus-stack-sso-secret.yaml` - the single sign-on secret necessary to use Github authorization when logging to Grafana
- `kube-prometheus-stack-discord-webhook-infrastructure-secret.yaml` - the discord secret needed to post alert notifications to Discord channel
- `standard-cluster-monitoring-recording-rules.yaml`, `onzack-namespace-monitoring-recording-rules.yaml` - dedicated recording rules required by custom Grafana dashboards

The last source specifies the Helm values file. The file for the prod environment can be found here.

Loki-Promtail Deployment

The Promtail component gathers logs from individual containers and Loki aggregates them for further processing. The argocd Application manifests of both components are listed below. The value files for Loki can be found here here, the Promtail default values proved to be sufficient.

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: loki-distributed
  namespace: argocd
  finalizers:
  - resources-finalizer.argocd.argoproj.io
spec:
  project: default
  sources:
  - repoURL: https://grafana.github.io/helm-charts
    targetRevision: 0.79.0
    chart: loki-distributed
    helm:
      releaseName: loki-distributed
      valueFiles:
      - $values/ionos_prd/loki-distributed/values.yaml
  - repoURL: https://github.com/DOME-Marketplace/dome-gitops
    targetRevision: HEAD
    ref: values
  destination:
    namespace: loki-distributed
    server: https://kubernetes.default.svc
  syncPolicy:
    automated:
      prune: true
      selfHeal: true
```

```
    syncOptions:
    - CreateNamespace=true


apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: promtail
  namespace: argocd
  finalizers:
  - resources-finalizer.argocd.argoproj.io
spec:
  project: default
  sources:
  - repoURL: https://grafana.github.io/helm-charts
    targetRevision: 6.16.0
    chart: promtail
    helm:
      releaseName: promtail
      valueFiles:
      - $values/ionos_prd/promtail/values.yaml
  - repoURL: https://github.com/DOME-Marketplace/dome-gitops
    targetRevision: HEAD
    ref: values
  destination:
    namespace: promtail
    server: https://kubernetes.default.svc
  syncPolicy:
    automated:
      prune: true
      selfHeal: true
    syncOptions:
    - CreateNamespace=true
```

Falco Deployment

To deploy Falco a single manifest is needed and so far the default Helm values of the Falco Helm Chart proved to be sufficient:

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: falco
  namespace: argocd
  finalizers:
  - resources-finalizer.argocd.argoproj.io
spec:
  project: default
  sources:
  - repoURL: https://falcosecurity.github.io/charts
    targetRevision: 4.5.0
    chart: falco
    helm:
      releaseName: falco
      valueFiles:
      - $values/ionos_prd/falco/values.yaml
  - repoURL: https://github.com/DOME-Marketplace/dome-gitops
    targetRevision: HEAD
    ref: values
  destination:
    namespace: falco
    server: https://kubernetes.default.svc
  syncPolicy:
    automated:
      prune: true
      selfHeal: true
    syncOptions:
    - CreateNamespace=true
```

5 Conclusion

In conclusion, this document has detailed the deployment scenarios and validation processes for the DOME platform, providing comprehensive guidelines and best practices for various stakeholders. The DOME project, through its federated marketplace model, aims to enhance the integration and interoperability of cloud and edge services across Europe.

The deployment scenarios outlined in this document offer a step-by-step approach to setting up and managing local instances of the DOME platform. This includes detailed instructions on using GitOps principles for automation, configuring Kubernetes clusters, and managing secrets securely. The document also emphasizes the importance of using Helm charts and Kubernetes manifests to streamline the deployment process, ensuring that all necessary components are correctly configured and deployed.

Validation of the DOME platform is critical to ensure its robustness and reliability. The document highlights various validation strategies, including functional testing, performance evaluation, and security assessments. These validation steps are essential to guarantee that the platform meets the required standards and can handle the demands of real-world applications.

The DOME project partners, federated marketplaces, and potential external providers can leverage the information in this document to deploy and validate their instances of the DOME platform effectively. By following the guidelines and recommendations provided, stakeholders can ensure a smooth and successful deployment, contributing to the overall success of the DOME initiative.

Overall, the DOME platform represents a significant step forward in creating a distributed, open marketplace for cloud and edge services in Europe. The deployment and validation guidelines provided in this document are integral to achieving the project's goals and ensuring its long-term sustainability and impact.