Ref. Ares(2025)830928 - 03/02/2025

DOME

(Distributed Open Marketplace for Europe)

D4.3 DOME operation report 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-D.4.3-V0.1

Project Document Date

26.12.2024

Deliverable Type and Security

REPORT, PUBLIC

Main editor

ACK Cyfronet

Contributors

FICODES, OUTSCALE, ENG, IONOS

Log Table

Version	Date	Change	Author/partner
V1.0	02.01.2025	Final Version	ACK CYFRONET

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

Table of Contents

1 Introduction 6
- 1.1 Executive Summary 6
- 1.2 Intended audience 6
- 1.3 Structure of the document 7
- 1.4 Related documents and resources 7

2 Monitoring and alerting infrastructure for DOME operations 8
- 2.1 Monitoring System Architecture 8
- 2.2 Cloud Infrastructure Monitoring 10
  - 2.2.1 Grafana Dashboards 11
- 2.3 Applications Monitoring 14
  - 2.3.1 Grafana Dashboards 16
- 2.4 User Services Monitoring 17
  - 2.4.1 Grafana Dashboards 19
- 2.5 Alerting System 19
- 2.6 Infrastructure Alerting 21
- 2.7 Applications Alerting 23

3 Installation and Setup 24
- 3.1 Installation of kube-prometheus-stack 24

List of figures

Figure 1 - Monitoring System Architecture

Figure 2 - Standard Cluster Monitoring Dashboard

Figure 3 - Standard Namespace Monitoring Dashboard

Figure 4 - ArgoCD Application Monitoring Dashboard

Figure 5 - User Services Monitoring Dashboard

Figure 6 - Kube-prometeus-stack Components

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

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

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

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

1 Introduction

1.1 Executive Summary

This document addresses developers and service providers contributing to the DOME ecosystem and its associated federated marketplace. It is particularly pertinent for entities offering services within DOME, a pivotal initiative in the advancement of cloud computing capabilities within the European digital landscape. The European Commission's strategic objective to accelerate cloud adoption across both public and private sectors within the European Union is being realized through the development of a pan-European cloud services marketplace. DOME is positioned to fulfill this objective, serving as a trusted portal for the provisioning of Cloud and Edge services, alongside other software and data processing services, under the auspices of significant EU programs such as the Digital Europe Programme and Horizon 2020.

Architecturally aligned with Gaia-X principles and leveraging open standards, DOME aims to enhance the proliferation of reliable cloud and edge services throughout Europe. It achieves this by establishing a federated system of marketplaces interconnected with a distributed digital catalogue of services. This architecture connects service consumers with providers in a decentralized manner. These marketplaces are designed to be adaptable to specific cloud or platform providers, thus facilitating the seamless integration of vertical services, such as those tailored for smart city initiatives, and specialized services, including Artificial Intelligence (AI) offerings.

The DOME ecosystem is predicated on the utilization of common open standards for both service documentation and access, mediated through a unified, distributed catalogue. This standardization ensures interoperability and facilitates the discovery and utilization of services within the federated marketplace. The architecture is designed to support a diverse array of services and providers, while maintaining a consistent and reliable user experience.

1.2 Intended audience

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

- Owner of the DOME components and services; those who intended to develop some software components and offer their services for DOME
- Potential new partners; those who are intended to add their services to the DOME shared catalogue
- Owner of the open-source marketplace which will be develop and integrate with the DOME ecosystem

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

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 three main chapters, each addressing different aspects of the DOME monitoring and alerting infrastructure infrastructure:

- **Chapter 1: Introduction** - Provides a summary of the document, intended audience, and an overview of its structure and related resources.
- **Chapter 2: Monitoring and alerting infrastructure for DOME operations** - Describes the purpose, architecture, and core functionalities of the monitoring and alerting system of DOME.
- **Chapter 3: Installation and Setup** - explains how to setup the monitoring and alerting infrastructure from scratch.

Each chapter is designed to give a comprehensive understanding of the respective topics, ensuring that the reader has all the necessary information to use and deploy the DOME monitoring and alerting infrastructure.

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

2 Monitoring and alerting infrastructure for DOME operations

2.1 Monitoring System Architecture

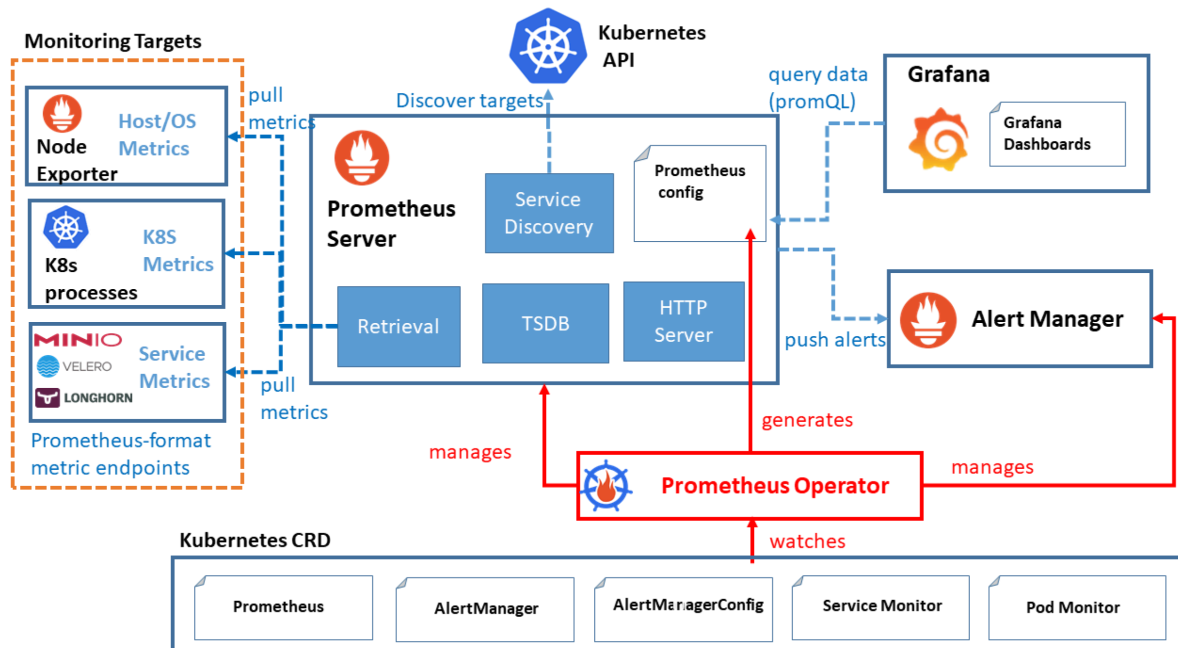


Fig. 1 - Monitoring System Architecture

The monitoring system uses The `kube-prometheus-stack` helm chart to deploy and manage the lifecycle of its components. This is the most standard way to realise the Prometheusdriven monitoring on a kubernetes cluster.

Architectural Overview of the `kube-prometheus-stack` Helm Chart for Kubernetes Monitoring

The `kube-prometheus-stack` Helm chart represents a comprehensive and integrated solution for monitoring Kubernetes clusters, predicated on the Prometheus ecosystem. This architecture facilitates the automated deployment, configuration, and management of critical monitoring components within a Kubernetes environment.

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

Core Architectural Components and Their Functional Roles:

The `kube-prometheus-stack` comprises an assembly of components, each fulfilling a distinct function within the monitoring pipeline:

1. **Prometheus Operator:**
    - **Orchestration and Management Layer:** The Prometheus Operator serves as the central control plane for the deployment, acting as an orchestrator for Prometheus, Alertmanager, and associated components.
    - **Kubernetes-Native Configuration:** It uses Kubernetes Custom Resource Definitions (CRDs) — such as `Prometheus`, `Alertmanager`, `ServiceMonitor`, `PodMonitor`, `Probe`, and `PrometheusRule` - to declaratively define the desired state of the monitoring infrastructure.
    - **State Reconciliation:** The Operator actively monitors the Kubernetes API server for changes to these CRDs and applies a reconciliation loop to align the observed state with the defined desired state.

2. **Prometheus:**
    - **Time-Series Data Acquisition and Storage:** Prometheus is the core of the monitoring system, handling metrics collection, storage, and query processing. It functions as a time-series database, storing metrics scraped from configured targets.
    - **Operator-Mediated Deployment:** Prometheus instances are deployed and managed by the Operator, adhering to the configurations defined in the `Prometheus` CRD, encompassing aspects such as resource allocation, scaling, and high availability.
    - **Target Discovery:** Prometheus employs a variety of service discovery mechanisms to dynamically identify metric endpoints. Within the `kube-prometheus-stack`, Kubernetes service discovery is extensively utilized, primarily configured through `ServiceMonitor` and `PodMonitor` CRDs, which abstract the underlying Kubernetes API objects.

3. **Alertmanager:**
    - **Alert Management and Notification:** Alertmanager acts as the central hub for handling alerts generated by Prometheus instances. It provides robust mechanisms for alert aggregation, inhibition, routing, and notification.
    - **Alert Processing and Deduplication:** It receives alerts triggered by rules defined in `PrometheusRule` CRDs and performs deduplication, grouping of related alerts, and suppression of notifications based on predefined configurations.
    - **Operator-Driven Configuration:** The Operator manages the lifecycle and configuration of Alertmanager instances, instantiated based on the specifications provided in the `Alertmanager` CRD.

4. **Grafana:**

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

- **Data Visualization and Analysis:** Grafana offers a sophisticated platform for the visualization and exploration of metrics data stored in Prometheus. It empowers users to create interactive dashboards, define thresholds for visual alerts, and analyze trends.
- **Integrated Solution Component:** the `kube-prometheus-stack` typically includes a pre-configured Grafana instance for operational convenience, readily connected to Prometheus as a data source.
- **Automated Dashboard Provisioning:** The chart supports automated provisioning of community-curated dashboards tailored for Kubernetes monitoring, enhancing immediate observability.

5. **kube-state-metrics:**

   - **Kubernetes State Metrics Exposition:** This dedicated agent functions as a listener to the Kubernetes API server, generating metrics that represent the state and health of various Kubernetes resources (e.g., Pods, Deployments, Nodes).
   - **Stateless Metrics Provider:** It operates without persistent storage, solely exposing metrics in a format consumable by Prometheus for scraping.

6. **prometheus-node-exporter:**

   - **Host-Level Metrics Collection:** This agent, typically deployed as a Kubernetes DaemonSet to ensure cluster-wide coverage, is responsible for collecting hardware and operating system-level metrics from each node in the cluster (e.g., CPU utilization, memory consumption, disk I/O, network bandwidth).

System usage:

By deploying the kube-prometheus-stack, the administrator/developer can use Prometheus to collect time-series data on cluster resources (e.g., CPU usage, memory consumption, disk I/O) and Grafana to visualise this data in real-time dashboards. Alerts can be configured using Alertmanager to notify administrators about resource saturation, pod crashes, or node failures.

It can trigger auto-scaling policies or alert the team when resources are near capacity. Horizontal Pod Autoscaler (HPA) can be integrated with Prometheus to automatically scale pods based on these metrics.

With ServiceMonitor and PodMonitor, Prometheus can scrape custom application metrics exposed by application containers. Grafana can be used to create dashboards that track the performance of individual services and their interdependencies.

The kube-prometheus-stack can be used to monitor security metrics like unauthorised access attempts, container vulnerabilities, and security policy violations using custom Prometheus rules. Alerts from Alertmanager can notify security teams in real time of any suspicious activities, and Grafana dashboards can provide detailed insights for audits.

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

2.2 Cloud Infrastructure Monitoring

Since the Kubernetes cluster composes almost the entire infrastructure of DOME project, the monitoring of cloud infrastructure focuses on monitoring Kubernetes clusters.

2.2.1 Grafana Dashboards

The `kube-prometheus-stack` Helm chart, beyond deploying the core monitoring infrastructure, provisions a curated set of Grafana dashboards. These dashboards provide pre-configured visualizations of key metrics, offering significant operational advantages for monitoring Kubernetes clusters. Moreover, DOME incorporates additional dashboards created and open-sourced by LGT Bank in Liechtenstein and published here: https://github.com/onzack/grafana-dashboards

Those dashboards exhibit the following key features:

Comprehensive Kubernetes Resource Monitoring:

- The dashboards provide a holistic view of Kubernetes cluster resources, including:
    - **Cluster-level metrics:** Overall CPU, memory, and network utilization across the entire cluster.
    - **Node-level metrics:** Detailed metrics for individual nodes, including resource usage, disk I/O, and network statistics, facilitating the identification of resource bottlenecks at the node level.
    - **Namespace-level metrics:** Aggregated metrics for specific namespaces, enabling resource usage tracking and allocation per project or team.
    - **Pod-level metrics:** Granular insights into individual pod resource consumption, facilitating the optimization of application performance.
    - **Persistent Volume metrics**: Detailing information on the status and health of persistent storage.

Kubernetes API Server Monitoring:

- Dedicated dashboards offer insights into the performance and health of the Kubernetes API server, a critical component for cluster operation. These metrics include:
    - **API request latency:** Quantifying the response time of the API server to various requests.
    - **API request rates:** Tracking the volume of requests handled by the API server.
    - **Workqueue depth and latency:** Exposing performance aspects of internal API server work queues.
    - **etcd performance:** As etcd is often the backend for API server, dashboards might expose etcd performance and health metrics, such as read/write latency.

Control Plane Monitoring:

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

- Dashboards offer insights into the performance and health of the Kubernetes control plane components. These metrics include:
  - **Scheduler latency:** Monitoring the time taken for the scheduler to schedule pods.
  - **Controller Manager performance:** Tracking the performance of various controllers.

Comprehensive list of dashboards used in DOME project for cloud infrastructure monitoring that enables analysis of cloud infrastructure status from different angles:

Core Kubernetes Service Dashboards:

- CoreDNS
- Etcd
- Kubernetes / API server
- Kubernetes / Controller Manager
- Kubernetes / Proxy
- Kubernetes / Kubelet
- Kubernetes / Scheduler

Kubernetes Resource Utilization Dashboards:

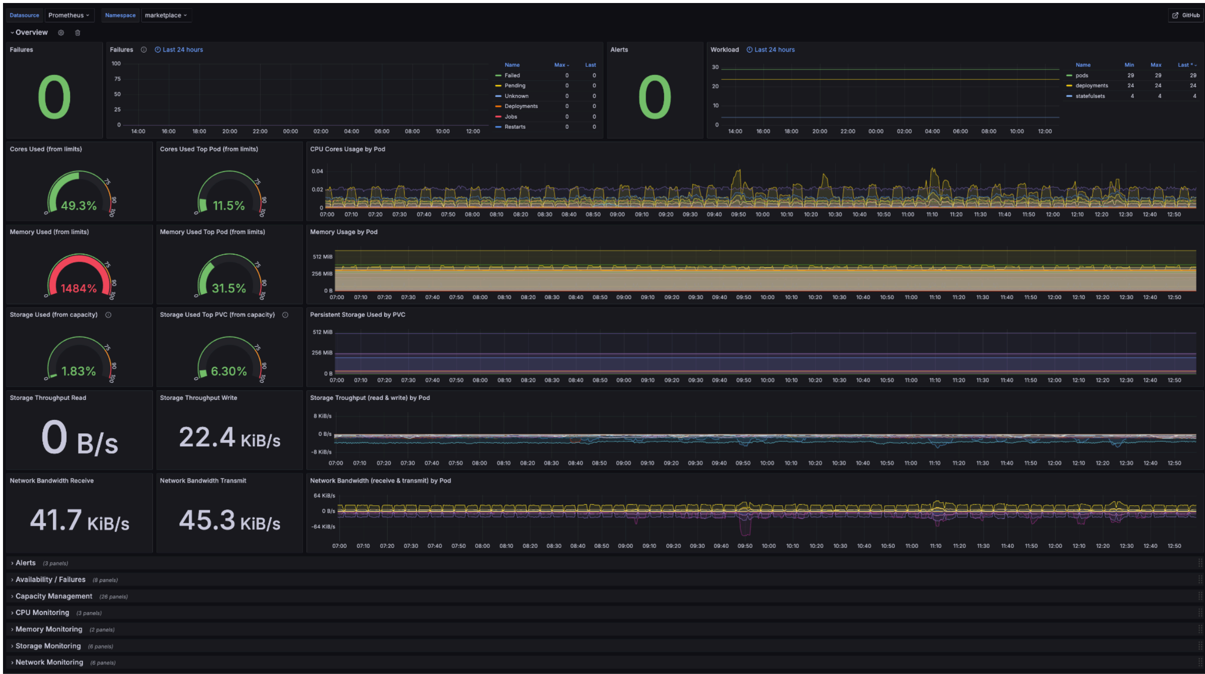
- Kubernetes / Compute Resources / Cluster
- Kubernetes / Compute Resources / Namespace (Pods)
- Kubernetes / Compute Resources / Namespace (Workloads)
- Kubernetes / Compute Resources / Node (Pods)
- Kubernetes / Compute Resources / Pod
- Kubernetes / Compute Resources / Workload
- Kubernetes / Networking / Cluster
- Kubernetes / Networking / Namespace (Pods)
- Kubernetes / Networking / Namespace (Workload)
- Kubernetes / Networking / Pod
- Kubernetes / Networking / Workload
- Kubernetes / Persistent Volumes

Comprehensive multi-angle dashboards developed by LGT Bank in Liechtenstein:

- Standard Cluster Monitoring
  A screenshot of part of the dashboard:

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

- Standard Namespace Monitoring
  A screenshot of part of the dashboard (for marketplace namespace)



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

2.3 Applications Monitoring

the DOME application, monitoring is tightly integrated with ArgoCD as part of the GitOps methodology. ArgoCD ensures that the desired application state, as defined in Git, is continuously applied to the Kubernetes cluster, while also monitoring the health of deployed applications in real-time. This health monitoring goes beyond simple pod readiness checks, evaluating the overall state of all resources associated with an application. By aggregating the health of all components, ArgoCD provides a unified health status for each DOME component, defined as a single ArgoCD application, offering a centralized point of health monitoring for the entire application. Here's a breakdown of how ArgoCD's application health monitoring works:

1. Health Assessment Framework:

Argo CD employs a hierarchical health assessment framework. The overall health of an application is determined by the aggregated health of its individual resources. This framework consists of the following components:

- **Resource-Specific Health Checks:** ArgoCD has built-in logic to assess the health of standard Kubernetes resources (Deployments, ReplicaSets, Services, StatefulSets, etc.). These checks vary depending on the resource type.
- **Custom Health Checks (Lua Scripts):** For non-standard resources or to implement custom health logic, ArgoCD allows defining custom health checks using Lua scripts.
- **Health Status:** Each resource is assigned a health status based on the checks performed. The possible health statuses are:
    - **Healthy:** The resource is functioning as expected.
    - **Progressing:** The resource is in the process of being deployed or updated and is not yet healthy, but is making progress.
    - **Degraded:** The resource is operational but has some issues that may require attention.
    - **Suspended:** The resource's reconciliation is paused.
    - **Missing:** The resource is expected to exist but is not found in the cluster.
    - **Unknown:** The health of the resource cannot be determined.

2. Built-in Health Checks for Kubernetes Resources:

Argo CD has predefined health checks for common Kubernetes resources. Here are some examples:

- **Deployment:**
    - **Healthy:** The number of available replicas matches the desired number of replicas, and all pods are ready.
    - **Progressing:** The deployment is being updated (e.g., rolling update), and the new ReplicaSet is being scaled up while the old one is being scaled down.
    - **Degraded:** The number of available replicas is less than the desired number, or pods are not ready.
- **ReplicaSet:**
    - **Healthy:** The number of ready replicas matches the desired number of replicas.
    - **Progressing:** The ReplicaSet is being scaled up or down.
    - **Degraded:** The number of ready replicas is less than the desired number.

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

- **StatefulSet:**
  - **Healthy:** The number of ready replicas matches the desired number of replicas, and all pods are ready. Each pod that should be updated is updated.
  - **Progressing:** Pods are being created or updated sequentially, according to the StatefulSet's update strategy.
  - **Degraded:** The number of ready replicas is less than the desired number, or pods are not ready, or pods are not updated to the latest version.
- **Service:**
  - **Healthy:** The service has at least one endpoint (i.e., it's routing traffic to at least one pod).
  - **Degraded:** The service has no endpoints.
- **Ingress:**
  - **Healthy:** The ingress has a load balancer configured and is ready to accept traffic.
  - **Degraded:** The ingress does not have a load balancer configured.
- **PersistentVolumeClaim:**
  - **Healthy:** The PVC is bound to a PersistentVolume.
  - **Degraded:** The PVC is not bound.

3. Custom Health Checks (Lua):

For resources not covered by built-in health checks, or for situations requiring specific health logic, ArgoCD allows defining custom health checks using Lua scripts.

- **Configuration:** Custom health checks are defined in the `resource.customizations` section of the ArgoCD ConfigMap (`argocd-cm`).
- **Script Structure:** The Lua script receives the resource object as input and must return a table containing:
  - `status`: The health status (one of the values listed above).
  - `message`: An optional message providing more details about the health status.

4. Health Assessment Aggregation:

Argo CD aggregates the health statuses of individual resources to determine the overall health of an application:

- **Healthy:** All resources within the application are healthy.
- **Progressing:** One or more resources are progressing, and no resources are degraded, suspended, or missing.
- **Degraded:** One or more resources are degraded.
- **Suspended:** One or more resources are suspended and no resources are degraded or missing.
- **Missing:** One or more resources are missing.
- **Unknown:** The health status of one or more resources is unknown, and no resources are degraded or missing.

5. Health Status in the ArgoCD UI and CLI:

The health status of applications and resources is prominently displayed in the ArgoCD UI and can also be accessed using the ArgoCD CLI.

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

- **UI:** The UI provides a visual representation of health status using icons and colors, making it easy to quickly assess the state of applications.
- **CLI:** The `argocd app get` command displays the overall health status of an application, and the `argocd app list` command lists the health status of all applications. `argocd app manifests` can be used to get detailed information about the health of each resource.

Benefits of ArgoCD's Health Monitoring:

- **Early Issue Detection:** Continuous health monitoring allows for the early detection of problems, enabling prompt intervention and minimizing downtime.
- **Automated Rollouts and Rollbacks:** ArgoCD can be configured to automatically roll back deployments if the application's health degrades, ensuring application stability.
- **Improved Reliability:** By monitoring the health of all resources, ArgoCD helps ensure the overall reliability of deployed applications.
- **Simplified Troubleshooting:** The detailed health information provided by ArgoCD simplifies troubleshooting and helps pinpoint the root cause of issues.
- **Customizable Health Checks:** The ability to define custom health checks using Lua scripts provides flexibility to tailor health monitoring to specific application

2.3.1 Grafana Dashboards

A dedicated Grafana dashboard presents the real-time state of the ArgoCD application in terms of synchronization and health. The data for this dashboard is gathered by Prometheus from dedicated ArgoCD monitoring endpoints.

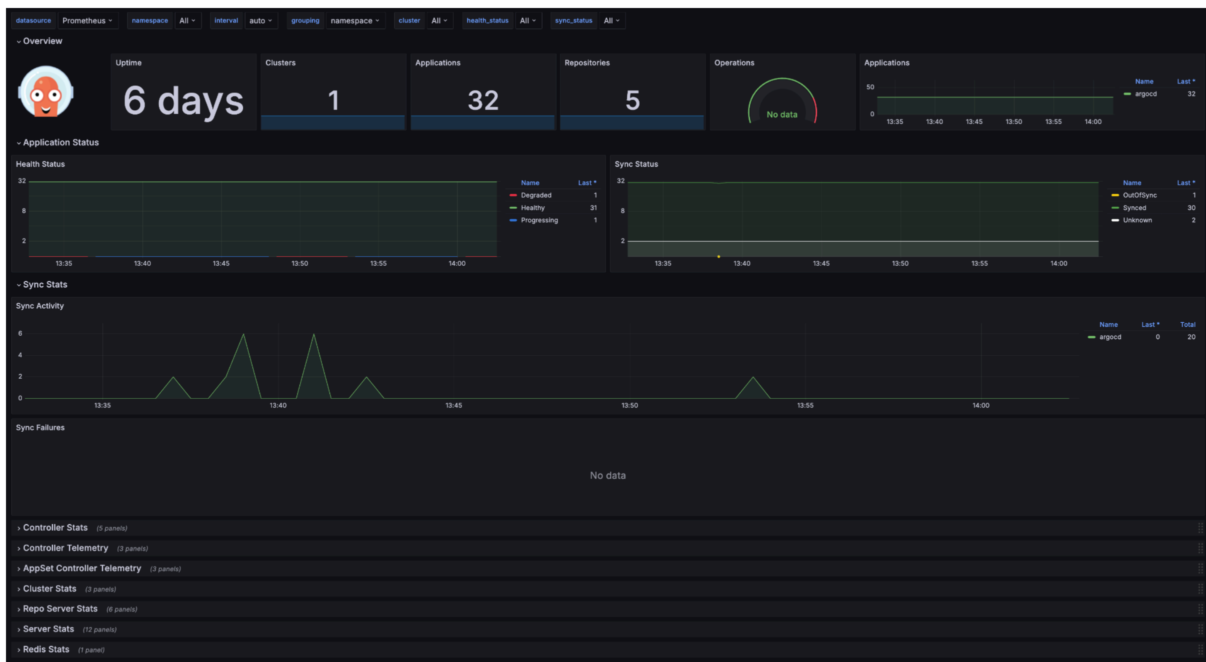


Fig 4 - ArgoCD Application Monitoring Dashboard

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

2.4 User Services Monitoring

The Prometheus Blackbox Exporter, an official project under the Prometheus umbrella, serves as a versatile tool for **probing** endpoints over various protocols, such as HTTP, HTTPS, DNS, TCP, and ICMP. Its primary function is to facilitate **external, black-box monitoring** of services, effectively emulating the perspective of an end-user or external client.

The core purpose of the Blackbox Exporter is to assess the **availability, performance, and correctness** of services by actively probing them and exposing metrics reflecting the results of these probes. This approach allows for the detection of issues that might not be apparent through internal, white-box monitoring alone. It is particularly valuable for:

- **End-to-End Monitoring:** Validating the entire service delivery chain, from the load balancer down to the application and underlying infrastructure.
- **Service Level Agreement (SLA) Monitoring:** Measuring and verifying metrics critical to SLA adherence, such as uptime and response time.
- **External Dependency Monitoring:** Ensuring the availability and performance of external services that an application relies upon.
- **Certificate Validity Monitoring:** Tracking the expiration of SSL/TLS certificates to prevent service disruptions due to expired certificates.
- **DNS Resolution Validation:**

Architectural Description:

The Blackbox Exporter is designed as a **standalone, stateless application** that exposes metrics in the Prometheus exposition format. It operates based on a modular architecture, comprising the following key components:

1. **Configuration:**
    - The Blackbox Exporter is configured via a YAML configuration file.
    - This file defines **modules** and **targets**.
    - **Modules** specify the type of probe to be performed (e.g., `http_2xx`, `tcp_connect`, `dns_lookup`), along with associated parameters like timeouts, valid status codes, and regular expressions for content validation.
    - **Targets** are the actual endpoints to be probed (e.g., `https://www.example.com`, `8.8.8.8:53`).
2. **Web Server:**
    - The exporter runs an embedded web server that listens for HTTP requests on a designated port (default: 9115).
    - The `/probe` endpoint is used to initiate probes.
3. **Probe Execution Logic:**
    - When a probe request is received, the exporter parses the request parameters, specifically `target` and `module`.
    - It then retrieves the corresponding module configuration.

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

  - Based on the selected module, it dynamically instantiates a **prober** responsible for executing the specific probe type.

4. **Probers:**

  - The Blackbox Exporter includes a set of built-in probers for common protocols, such as:
    - **`http`:** Performs HTTP requests (GET, POST) and validates response status codes, headers, and body content.
    - **`tcp`:** Establishes TCP connections and optionally sends/receives data.
    - **`icmp`:** Sends ICMP echo requests (pings).
    - **`dns`:** Performs DNS lookups and validates the results.
  - Each prober encapsulates the logic required to execute the specific probe type and gather relevant metrics.

5. **Metrics Exposition:**

  - The prober returns the results of the probe, including metrics such as:
    - **`probe_success`:** A boolean (gauge) indicating whether the probe was successful (1) or not (0).
    - **`probe_duration_seconds`:** A gauge representing the duration of the probe in seconds.
    - **`probe_http_status_code`:** (HTTP specific) The HTTP status code returned by the server.
    - **`probe_ssl_earliest_cert_expiry`:** (HTTPS specific) A gauge representing the timestamp of the earliest certificate expiry in the certificate chain.
    - **`probe_dns_lookup_time_seconds`:** (DNS specific) A gauge representing the duration of the DNS lookup in seconds.
    - Other protocol-specific metrics, including content length, phase timings, SSL handshake durations, etc.
  - These metrics are then exposed in the Prometheus exposition format via the web server's `/metrics` endpoint.

Operational Workflow:

1. **Configuration:** The Blackbox Exporter is configured with modules defining probe types and parameters.
2. **Prometheus Configuration:** Prometheus is configured to scrape the Blackbox Exporter's `/metrics` endpoint at a defined interval. Additionally, Prometheus is configured to send requests to the `/probe` endpoint, specifying the `target` and `module` to probe.
3. **Probe Execution:** When Prometheus sends a request to the `/probe` endpoint (e.g., `/probe?target=https://www.example.com&module=http_2xx`), the exporter executes the corresponding probe.
4. **Metrics Collection:** The results of the probe, including various metrics, are exposed on the `/metrics` endpoint.

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

5. **Metrics Storage and Analysis:** Prometheus scrapes these metrics and stores them in its time-series database, enabling analysis, visualization, and alerting based on the collected data.

2.4.1 Grafana Dashboards

A dedicated Grafana dashboard presents the real-time data on availability, performance, and correctness defined DOME services. The data for this dashboard is gathered by Prometheus from Prometheus Blackbox Exporter.

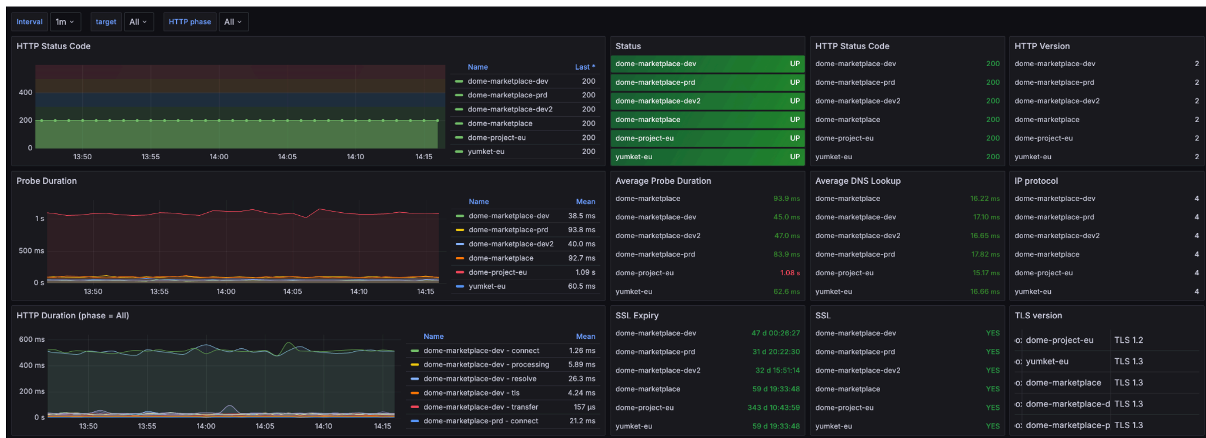


Fig 5 - User Services Monitoring Dashboard

2.5 Alerting System

The DOME alerting system uses components of Prometheus, which is a system designed to generate alerts based on time-series data, enabling proactive identification and resolution of operational issues. It comprises two primary, decoupled components: **Prometheus** and the **Alertmanager**. This separation of concerns allows for independent scalability and development. The system operates through a well-defined workflow involving rule evaluation, alert generation, notification routing, and alert management.

Components of the Alerting System:

1. **Prometheus:**
    - **Rule Evaluation Engine:** The core of alert generation resides within Prometheus servers. They are responsible for evaluating **alerting rules** defined in configuration files or dynamically managed through a rule management API.
    - **Alerting Rules:** These rules are written in PromQL (Prometheus Query Language) and define conditions based on time-series data that, when met, trigger an alert. Rules specify:
        - `expr`**:** A PromQL expression that is evaluated at regular intervals.

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

        - `for`: A duration that the expression must evaluate to true for before an alert is fired. This helps to prevent spurious alerts caused by transient fluctuations.
        - `labels`: A set of key-value pairs that are added to the generated alert. These can be used to add more context and meta-data about the alert. These are usually used for routing, grouping or inhibiting other alerts.
        - `annotations`: A set of key-value pairs that are added to the generated alert. These provide additional, human-readable information about the alert. These are information that are useful for the person who is reading the alert, like runbook links, description about what the alert means, graph links, etc.
    - **Alert State Management:** Prometheus servers track the state of each alerting rule, transitioning between `inactive`, `pending`, and `firing` states based on the evaluation of the rule's expression and the `for` duration.

2. **Alertmanager:**

    - **Alert Aggregation and Deduplication:** Alertmanager receives alerts sent by Prometheus servers. It deduplicates alerts based on their labels, suppressing redundant notifications for the same issue.
    - **Grouping:** Alertmanager groups related alerts into a single notification. This reduces alert fatigue by aggregating alerts that likely stem from the same root cause. Grouping is configurable based on labels.
    - **Routing:** Alertmanager routes alerts to different receivers (e.g., email, Slack, PagerDuty, OpsGenie) based on configurable routing policies. Routing configurations define which alerts should be sent to which receivers based on their labels.
    - **Inhibition:** Alertmanager can be configured to inhibit certain alerts if other, related alerts are already firing. This prevents notification storms when a high-level problem triggers a cascade of lower-level alerts.
    - **Silencing:** Alertmanager provides a mechanism to silence alerts for a specific duration, typically used during maintenance windows or to temporarily mute alerts for known issues.
    - **High Availability:** Alertmanager can be deployed in a high-availability configuration, ensuring that alert notifications are not lost due to a single point of failure.

Operational Workflow:

The alerting process follows these steps:

1. **Rule Evaluation:** Prometheus servers periodically evaluate alerting rules against the collected time-series data.
2. **Alert State Transition:**
    - If a rule's expression evaluates to true and remains true for the specified `for` duration, the alert transitions from `inactive` to `pending`.
    - After the `for` duration, the alert transitions from `pending` to `firing`.
    - If the rule's expression evaluates to false, the alert transitions back to `inactive`.

3. **Alert Forwarding:** When an alert enters the `firing` state, the Prometheus server sends it to the configured Alertmanager instances.
4. **Alert Processing (Alertmanager):**
    - **Deduplication:** Alertmanager deduplicates the incoming alert based on its labels. If an identical alert is already present, it is ignored.
    - **Grouping:** The alert is added to a group based on the configured grouping rules.
    - **Inhibition:** Alertmanager checks if any inhibition rules apply to the alert. If so, and the inhibiting alert is firing, the alert is suppressed.
    - **Routing:** The alert is routed to the appropriate receiver(s) based on the routing configuration.
5. **Notification Delivery:** Alertmanager sends a notification to the configured receiver(s) (e.g., email, Slack, PagerDuty). The notification typically includes the alert's labels, annotations, and a link back to the Alertmanager UI.
6. **Silencing:** Users can silence alerts via the Alertmanager UI or API, preventing notifications for a specified duration.
7. **Alert Resolution:** Once the underlying issue is resolved, the alerting rule in Prometheus will eventually evaluate to false, causing the alert to transition back to the `inactive` state. Alertmanager will automatically mark resolved alerts as such.

Key Architectural Considerations:

- **Decoupling:** The separation of Prometheus and Alertmanager allows for independent scaling and development, contributing to a more robust and flexible architecture.
- **Statelessness (Prometheus):** Prometheus servers are largely stateless, simplifying operations and facilitating horizontal scaling.
- **Declarative Configuration:** Alerting rules and routing configurations are defined declaratively, enabling version control and automated management.
- **Extensibility:** Alertmanager supports a wide range of integrations through its receiver system, allowing for seamless integration with various notification and incident management platforms.
- **High Availability:** Both Prometheus and Alertmanager can be deployed in highly available configurations, ensuring the reliability of the alerting pipeline.

Alerts are published to dedicated DOME discord channels and a subset of them (particularly related to User Service Monitoring is forwarded to dedicated mailing lists).

2.6 Infrastructure Alerting

The DOME infrastructure alerting system relies heavily on infrastructure monitoring data, with alerts being defined based on time-series telemetry. As of now, DOME delivers over 270 alert rules, aggregated by the specific infrastructure components, ensuring comprehensive monitoring and proactive issue detection across various system layers. Key alert rules include:

Alertmanager:

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

Manages and routes alerts generated by Prometheus, including aggregation, suppression, and notifications.

Config Reloaders:

Monitors the process of reloading configurations in the system, ensuring new changes take effect correctly.

etcd:

Monitors the health and performance of etcd, the distributed key-value store used by Kubernetes for storing cluster data.

Pod Owner:

Tracks the ownership and health of pods in Kubernetes, ensuring the right resources are associated with the right pods.

Kube Apiserver Availability:

Monitors the availability and responsiveness of the Kubernetes API server, the central management point for the cluster.

Kube Apiserver Burnrate:

Tracks the rate of resource consumption (e.g., CPU, memory) by the Kubernetes API server, helping identify potential performance bottlenecks.

Kube Apiserver Histogram:

Collects and visualizes metrics related to API server performance, such as request duration and status codes.

Kube Apiserver SLOs (Service Level Objectives):

Monitors the performance of the API server based on predefined SLOs, ensuring the server meets service-level agreements.

Kube Prometheus:

Monitors Kubernetes components using Prometheus, collecting metrics on system health and resource usage.

Kube Prometheus Node Recording:

Records node-level metrics such as CPU, memory, and disk usage for analysis and alerting in Prometheus.

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

Kube Scheduler Rules:

Monitors the behavior of the Kubernetes Scheduler, which assigns pods to nodes in the cluster.

Kube State Metrics:

Exposes metrics about the state of Kubernetes resources (e.g., pods, deployments, nodes) for Prometheus to scrape.

Kubelet:

Tracks the health and status of the Kubelet on each node, ensuring containers are running correctly.

Kubernetes Resources:

Monitors resources such as pods, deployments, services, and namespaces within Kubernetes to ensure they are healthy and meet performance requirements.

Kubernetes Storage:

Monitors storage resources (e.g., Persistent Volumes) within Kubernetes, ensuring availability

These rules provide alerts about failures of key components of the infrastructure.

2.7 Applications Alerting

The generic application alerting rules are derived from infrastructure monitoring metrics and help track issues within application containers, such as violations of resource limits or failures indicated by error codes. These rules specifically monitor:

Kubernetes Apps:

Monitors the health and resource usage of Kubernetes applications, ensuring containers run as expected within the defined resource constraints.

Container CPU Usage:

Tracks CPU utilization within containers, alerting if the usage exceeds predefined limits or indicates resource contention.

Container Memory Cache:

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

Monitors memory cache usage within containers to ensure they are not consuming excessive memory that could affect system performance.

Container Memory RSS (Resident Set Size):

Measures the RSS of container memory, indicating how much physical memory the container is using, which helps in detecting memory over-consumption.

Container Memory Swap:

Tracks the usage of swap memory by containers. High swap usage can indicate that containers are consuming more memory than available and are forced to swap to disk.

Container Memory Working Set:

Monitors the working set of memory used by containers, which is the memory actively used by the application. This helps detect if memory usage is increasing beyond expected levels.

Container Resource Usage:

General monitoring of resource usage (CPU, memory, disk, network) within containers, ensuring they remain within acceptable limits and perform as expected.

3 Installation and Setup

3.1 Installation of kube-prometheus-stack

The DOME Gitops repository provides the dedicated aggregate ArgoCD application that sets up the monitoring infrastructure with a single command on a kubernetes cluster (example from the production environment)

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: monitoring
  namespace: argocd
  labels:
    purpose: monitoring
spec:
  destination:
    namespace: argocd
```

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

```
    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
```

This follows the app-of-apps ArgoCD pattern and install in turn all the ArgoCD application stored under applications_prd/monitoring:

```
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-infrastr
ucture-secret.yaml,standard-cluster-monitoring-recording-rules.yaml,onzack-namespace-m
onitoring-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:
```

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

```
- CreateNamespace=true
- ServerSideApply=true
```

This application setups kube-prometheus-stack Helm chart providing it with dedicated values.yaml and additional Grafana dashboards. The detailed list of components installed this chart:

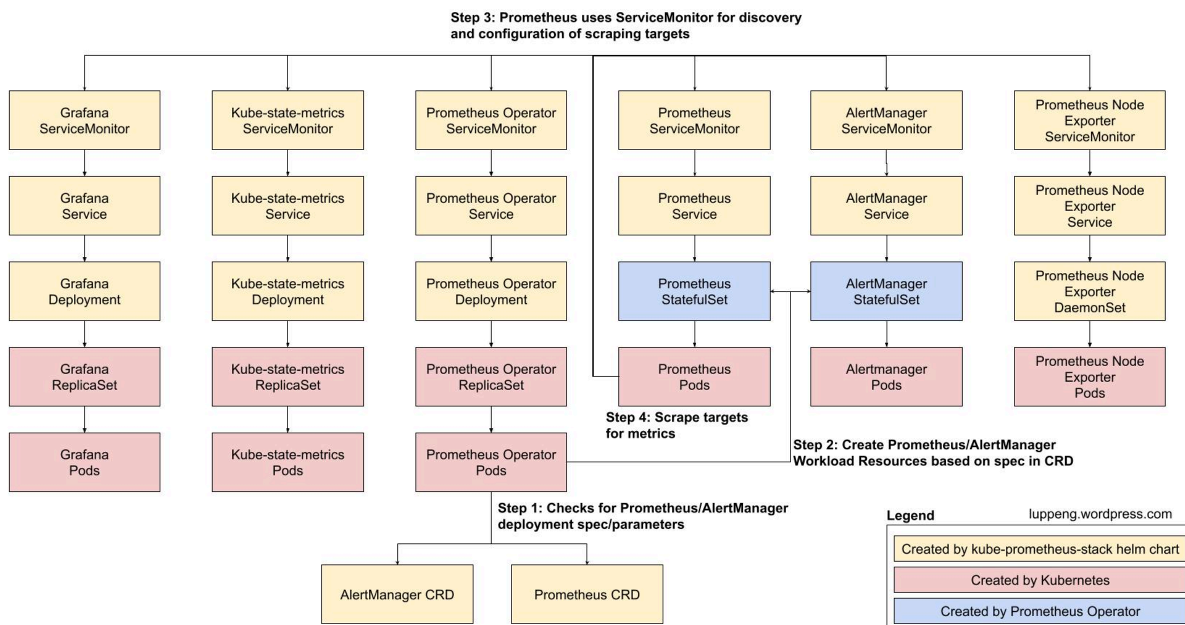


Fig 6 - Kube-prometeus-stack Components

The installation procedure requires only to invoke a single command:

```
kubectl create -f applications_sbx/monitoring.yaml
```

The certificates required by the components are automatically generated by the cert-manager component and the necessary

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