**Lab 12: Adding Docs with TechDocs (Backstage-native Docs)**

**🔹 Objective**

By the end of this lab, each trainee will:

* Add service-level documentation using Markdown and MkDocs.
* Configure TechDocs to render and serve these docs within Backstage.
* Understand how documentation becomes a first-class citizen in the developer portal.

**💡 Why This Lab Matters**

**Platform Engineering Perspective:** Documentation is one of the most underappreciated, yet most critical, components of platform engineering. This lab demonstrates how TechDocs enables **self-serve developer experience** by:

* Making documentation visible and accessible *in the same place* where developers discover and operate services.
* Eliminating the need to jump between Confluence, Google Docs, Notion, etc.
* Creating a documentation standard (Markdown + MkDocs).
* Encouraging devs to treat docs like code (version-controlled, PR-reviewed, co-located).

**📁 Lab Setup**

Ensure the following structure exists in your GitHub repo :

Customer-service/

├── catalog-info.yaml

├── mkdocs.yml

└── docs/

└── index.md

**✅ Step 1: Create Documentation Files**

**📄 Create mkdocs.yml in root:**

site\_name: customer-micro

nav:

- Home: index.md

plugins:

- techdocs-core

**📄 Create docs/index.md**

# Raman Microservice

Welcome to the \*\*customer-micro\*\* documentation.

## 🔧 Features

- Flask-based microservice

- CI/CD via GitHub Actions

- SAST, DAST, SBOM integrated

- Dockerized and Helm-compatible

- Exposes Prometheus metrics

## 🚀 Deployment

Deployed via GitHub Actions workflow.

## 🧪 Observability

Integrated with:

- Prometheus for metrics

- Loki/ELK for logs

- Grafana dashboards

## 👤 Owner

Maintained by \*\*Raman Khanna\*\*.

**✅ Step 2: Add TechDocs Annotation to Catalog**

Open your catalog-info.yaml and ensure it includes:

metadata:

annotations:

backstage.io/techdocs-ref: dir:.

This tells Backstage to look for docs locally inside the repo.

**✅ Step 3: Commit and Push Changes**

git add mkdocs.yml docs/

git commit -m "Add TechDocs documentation"

git push

**⚙️ Step 4: Confirm TechDocs Configuration in Backstage**

Ensure the following block is in your app-config.yaml:

techdocs:

builder: local

generator:

runIn: docker

publisher:

type: local

backend:

baseUrl: http://localhost:7007

✉️ If you use an external publisher (like GCS or S3), configuration will vary.

**✅ Step 5: View Docs in Backstage UI**

1. Open your Backstage portal
2. Navigate to: Catalog → raman-micro
3. Click on the **"View TechDocs"** button (top-right corner)

You should see your Markdown rendered beautifully inside the UI. 🎉

**📚 Summary for Trainees**

|  |  |
| --- | --- |
| **What You Did** | **Why It Matters** |
| Wrote Markdown Docs | Consistent and readable documentation format |
| Added mkdocs.yml | Defines the structure and metadata for docs |
| Used TechDocs annotation | Connects your repo to Backstage TechDocs |
| Rendered Docs in Backstage | Centralized access in developer portal |

**🔹 Post-Lab Discussion Questions**

1. How does co-locating docs with code improve your dev workflow?
2. Can TechDocs replace existing tools like Confluence?
3. How might you enforce documentation as part of service creation?

**📈 Platform Engineering Takeaways**

* Enables **self-serve documentation** without tribal knowledge.
* Bridges the gap between **code ownership** and **service discoverability**.
* Makes documentation versioned, reviewed, and owned just like code.

**Lab 13 – Add Prometheus Metrics & Grafana Dashboards to customer-order-service**

**🎯 Goal**

Enable your customer-order-service Flask microservice to **expose runtime metrics**, have them **scraped by Prometheus**, visualized in **Grafana dashboards**, and **linked inside Backstage** for developer observability.

**🔍 Why This Lab Matters in Platform Engineering**

As part of an **Internal Developer Platform (IDP)**:

1. **Self-service Observability**  
   Developers should **not** need to manually set up monitoring for every new service. The **Golden Path** should include *instrumentation by default*.
2. **Faster Troubleshooting**  
   Exposing metrics via Prometheus allows:
   * Real-time visibility into service health.
   * Quick detection of anomalies without SSH-ing into servers.
3. **Backstage Integration**  
   Linking **Grafana dashboards** directly inside Backstage means:
   * One-click access to metrics per service.
   * A single pane of glass for developers and SREs.
4. **SLO / SLA Tracking**  
   Metrics collected here can power Service Level Objectives (SLOs) and error budgets.

**📦 Lab Outcomes**

By end of this lab, you will:

* ✅ Have /metrics endpoint in your microservice.
* ✅ See metrics scraped by Prometheus.
* ✅ Build a Grafana dashboard.
* ✅ Link the dashboard in Backstage **catalog-info.yaml**.

**🧩 Steps**

**Step 1 – Add Prometheus Client to Flask**

1. **Install client library**:

bash

CopyEdit

pip install prometheus\_client

Add to requirements.txt:

nginx

CopyEdit

prometheus\_client

1. **Instrument your Flask app** (app.py):

python

CopyEdit

from flask import Flask, Response

from prometheus\_client import Counter, Histogram, generate\_latest, CONTENT\_TYPE\_LATEST

import time

app = Flask(\_\_name\_\_)

# Metrics

REQUEST\_COUNT = Counter("request\_count", "Total number of requests")

REQUEST\_LATENCY = Histogram("request\_latency\_seconds", "Request latency in seconds")

@app.route("/")

def home():

REQUEST\_COUNT.inc()

with REQUEST\_LATENCY.time():

time.sleep(0.1) # Simulate processing

return "Hello from Raman Micro!"

# Expose metrics

@app.route("/metrics")

def metrics():

return Response(generate\_latest(), mimetype=CONTENT\_TYPE\_LATEST)

if \_\_name\_\_ == "\_\_main\_\_":

app.run(host="0.0.0.0", port=5000)

**Step 2 – Run and Test Metrics Endpoint**

bash

CopyEdit

python app.py

Check in browser or terminal:

bash

CopyEdit

curl http://localhost:5000/metrics

You should see output like:

nginx

CopyEdit

# HELP request\_count Total number of requests

# TYPE request\_count counter

request\_count 2.0

**Step 3 – Create Prometheus Config**

Create prometheus.yml:

yaml

CopyEdit

global:

scrape\_interval: 10s

scrape\_configs:

- job\_name: 'raman-micro'

static\_configs:

- targets: ['host.docker.internal:5000']

This scrapes metrics from your Flask app every 10 seconds.

**Step 4 – Run Prometheus & Grafana in Docker**

Create a shared network:

bash

CopyEdit

docker network create observability-net

**Run Prometheus**:

bash

CopyEdit

docker run -d \

--name prometheus \

--network observability-net \

--add-host=host.docker.internal:host-gateway \

-v "$(pwd)/prometheus.yml":/etc/prometheus/prometheus.yml \

-p 9090:9090 \

prom/prometheus

**Run Grafana**:

bash

CopyEdit

docker run -d \

--name grafana \

--network observability-net \

-p 3001:3000 \

grafana/grafana

**Step 5 – Check Prometheus Target**

Go to:

arduino

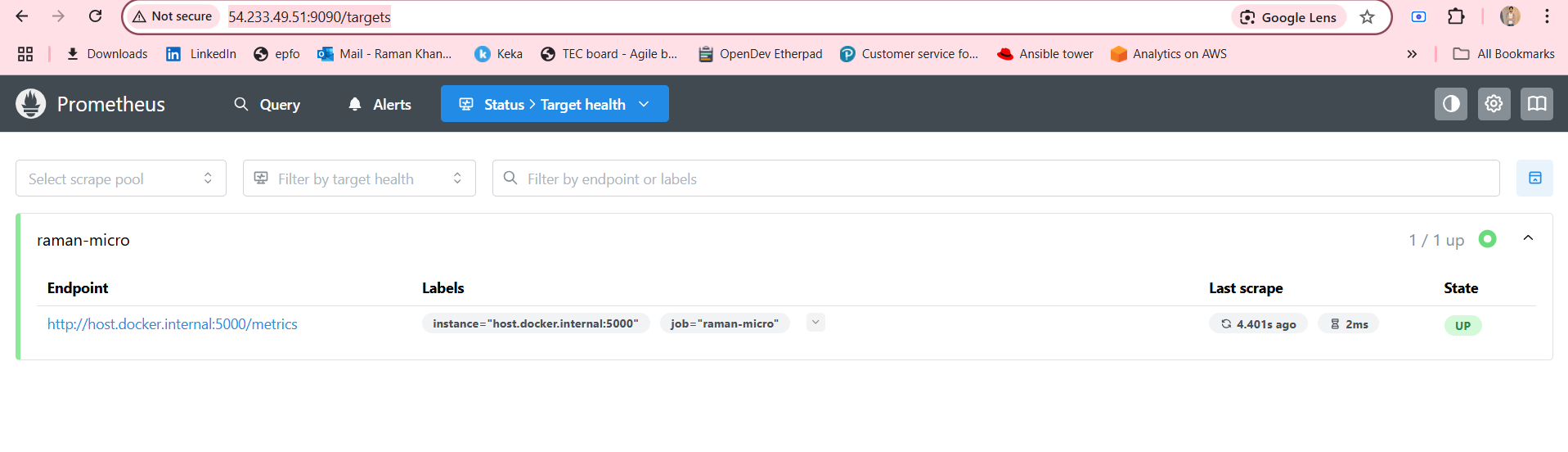
CopyEdit

http://<your-server-ip>:9090/targets

You should see:

CopyEdit

raman-micro – UP



**Step 6 – Configure Grafana**

1. Open Grafana:

cpp

CopyEdit

http://<your-server-ip>:3001

(Default login: admin/admin)

1. Go to:

sql

CopyEdit

⚙️ → Data Sources → Add Data Source → Prometheus

Set URL:

arduino

CopyEdit

http://prometheus:9090

Click **Save & Test** → Should say “Data source is working”.

**Step 7 – Create Grafana Dashboard**

1. Go to:

sql

CopyEdit

Dashboards → New → New Dashboard

**Panel 1 – Total Requests**:

* Query:

prometheus

CopyEdit

request\_count\_total

* Title: **Total Requests**
* Visualization: Time series

**Panel 2 – Average Latency**:

* Query:

prometheus

CopyEdit

rate(request\_latency\_seconds\_sum[1m]) / rate(request\_latency\_seconds\_count[1m])

* Title: **Average Latency**
* Visualization: Time series

Save as:

CopyEdit

raman-micro-dashboard



**Step 8 – Link Dashboard in Backstage**

Edit catalog-info.yaml of your service:

yaml

CopyEdit

apiVersion: backstage.io/v1alpha1

kind: Component

metadata:

name: raman-micro

description: Flask microservice with metrics

annotations:

github.com/project-slug: ramannkhanna2/raman-micro

grafana/dashboard-url: http://<your-server-ip>:3001/d/abc123/raman-micro-dashboard

spec:

type: service

owner: user:raman

lifecycle: production

Now, inside Backstage **service page**, you can show a **Grafana link** to view metrics.

**📌 Key Platform Engineering Takeaways**

* **Golden Path for Observability** → Any new microservice should have metrics from day 1.
* **Self-service Monitoring** → Devs can see service health without raising infra tickets.
* **One-Stop Developer Portal** → Backstage becomes the central hub for code, builds, and monitoring.
* **Promotes SRE Practices** → Enables tracking of error budgets, latency SLIs, and uptime SLOs.

**Lab 14 – Integrate Grafana Dashboards into Backstage for customer-order-svc**

**🎯 Goal**

Link your **Grafana dashboards** for the customer-order-svc microservice directly into Backstage so developers, SREs, and platform teams can:

* See service metrics inside the **Backstage service entity page**.
* Open Grafana dashboards with **one click**.
* Easily filter dashboards for the specific service using annotations and tags.

**🔍 Why This Lab Matters in Platform Engineering**

In **Lab 12**, you instrumented your service with **Prometheus metrics** and visualized them in **Grafana**.  
However, without **Backstage integration**, developers have to:

* Remember the Grafana URL.
* Search for the correct dashboard manually.
* Navigate across multiple tools.

By embedding Grafana into **Backstage**:

* **Single Pane of Glass** → Developers see code, builds, deployments, and metrics in one place.
* **Golden Path Observability** → Every scaffolded service can automatically link to its dashboard.
* **Faster Incident Response** → No context switching during production issues.
* **Developer Self-Service** → No tickets needed to access performance data.

This is a key **Platform Engineering principle**:

“Developers should spend time shipping features, not finding dashboards.”

**📦 Lab Outcomes**

By the end of this lab:

* ✅ Grafana community plugin installed in Backstage.
* ✅ Backstage proxy configured for Grafana API access.
* ✅ Service entity page shows **Grafana Dashboard Card**.
* ✅ customer-order-svc dashboard linked via annotations.

Two community versions available:

* Official Backstage community version: **@backstage-community/plugin-grafana**
* Roadie fork by K‑Phoen: **@k-phoen/backstage-plugin-grafana**

📦 For most users, the **community version** is ideal:

<https://github.com/backstage/community-plugins/blob/main/workspaces/grafana/plugins/grafana/docs/setup.md>

root@ip-172-31-14-172:~/backstage-app/ramanapp/packages/app# yarn add @backstage-community/plugin-grafana

➤ YN0000: · Yarn 4.4.1

➤ YN0000: ┌ Resolution step

➤ YN0000: └ Completed in 0s 789ms

➤ YN0000: ┌ Post-resolution validation

➤ YN0060: │ @testing-library/react is listed by your project

* To vierify if plugin installed :

root@ip-172-31-14-172:~/backstage-app/ramanapp/packages/app# cat package.json

* U shud see the plugin there .
* Go to Grafana and generate a service account and token

**🔧 Step 2: Configure Proxy and Grafana URL (if Grafana requires auth or is remote)**

Edit **app-config.yaml** (at project root):

proxy:

### Example for how to add a proxy endpoint for the frontend.

### A typical reason to do this is to handle HTTPS and CORS for internal services.

# endpoints:

# '/test':

# target: 'https://example.com'

# changeOrigin: true

'/grafana/api':

target: http://54.233.49.51:3001

headers:

# Only needed if your Grafana API requires an auth token

Authorization: Bearer glsaRxAGZz5qCqX8j8dqflkgbut #grafana sa token

grafana:

domain: http://54.233.49.51:3001

# If you're using Grafana's new unified alerting:

unifiedAlerting: false

* Expose the plugin to Backstage:
* Create a file named plugins.tsx in ~/backstage-app/ramanapp/packages/app/src
* // packages/app/src/plugins.tsx
* // other plugins...
* export { grafanaPlugin } from '@backstage-community/plugin-grafana';
* **for testing if ur able to reach Grafana ui from api :**

curl -H "Authorization: Bearer glsa\_RxAGZz5qCqX8j8dqflnr8oi" http://54.233.49.51:3001/api/search?tag=raman-micro

**🧩 Step 3: Update Service Entity Layout to Include Dashboard Card**

[**https://github.com/backstage/community-plugins/blob/main/workspaces/grafana/plugins/grafana/docs/dashboards-on-component-page.md**](https://github.com/backstage/community-plugins/blob/main/workspaces/grafana/plugins/grafana/docs/dashboards-on-component-page.md)

Edit **packages/app/src/components/catalog/EntityPage.tsx** (or wherever you layout service pages):

**Display dashboards on a component page**

Adding the EntityGrafanaDashboardsCard component to an entity's page will display a list of dashboards related to that entity.

// packages/app/src/components/catalog/EntityPage.tsx

import { EntityGrafanaDashboardsCard } from '@backstage-community/plugin-grafana';

// ...

const overviewContent = (

<Grid container spacing={3} alignItems="stretch">

<Grid item md={6}>

<EntityAboutCard variant="gridItem" />

</Grid>

<Grid item md={6}>

{/\* Grafana alert card start \*/}

<EntityGrafanaDashboardsCard />

{/\* Grafana alert card end \*/}

</Grid>

<Grid item md={4} xs={12}>

<EntityLinksCard />

</Grid>

<Grid item md={8} xs={12}>

<EntityHasSubcomponentsCard variant="gridItem" />

</Grid>

</Grid>

);

* on Grafana ui add the tag to the "count" dashboard "customer-micro" so that backstage plugin can find the dashboard of urs in Grafana ..

**🔁 Step 4: Ensure Your catalog-info.yaml Has the Grafana Annotation**

root@ip-172-31-14-172:~/raman-micro# cat catalog-info.yaml

apiVersion: backstage.io/v1alpha1

kind: Component

metadata:

name: raman-micro

description: Flask microservice demo with Prometheus & Grafana

tags:

- flask

- python

- prometheus

- demo

annotations:

github.com/project-slug: ramannkhanna2/raman-micro

backstage.io/techdocs-ref: dir:.

grafana/dashboard-url: http://54.233.49.51:3001/d/1f416562-37ea-4ef1-8df3-d6d1381b7f89/count

grafana/dashboard-selector: customer-micro

#grafana/dashboard-selector: 'tag=raman-micro'

#grafana/tag-selector: raman-micro

spec:

type: service

lifecycle: production

owner: dev-team

system: raman-platform

This ensures the plugin can pick up the correct dashboard to display.

**🌀 Step 5: Restart Backstage & Refresh Entity**

bash

CopyEdit

cd ~/backstage-app/raman-app

yarn dev

OR

Yarn start

* Than again register ur service on backstage .

https://github.com/ramannkhanna2/raman-micro/blob/main/catalog-info.yaml

→ Go to **Catalog → raman-micro**, then click the “Refresh” button (⋮ menu).

You should see a **“Grafana”** card with your dashboard preview and alerts (if any).

**✅ Summary: What You’ve Set Up**

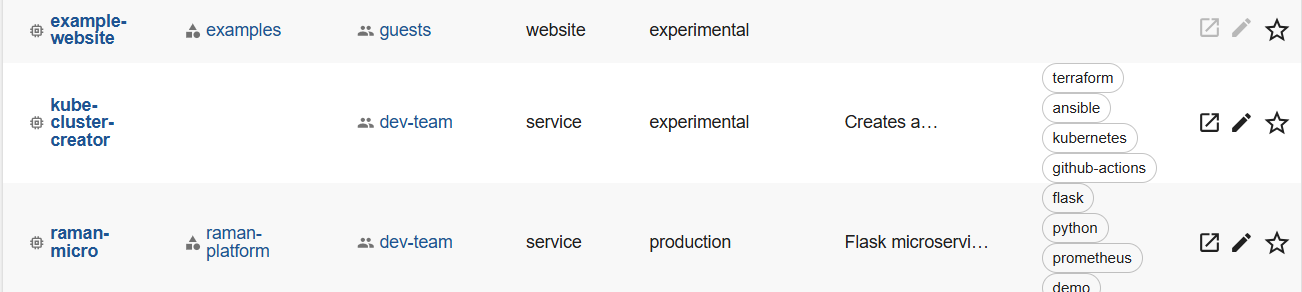
* 🎯 Installed the **Grafana plugin**
* 🔧 Configured proxy and domain in app-config.yaml
* 🛠️ Added **Grafana cards** to service page layout
* ✅ Provided dashboard URL in your service metadata
* 🔄 Restarted Backstage and refreshed

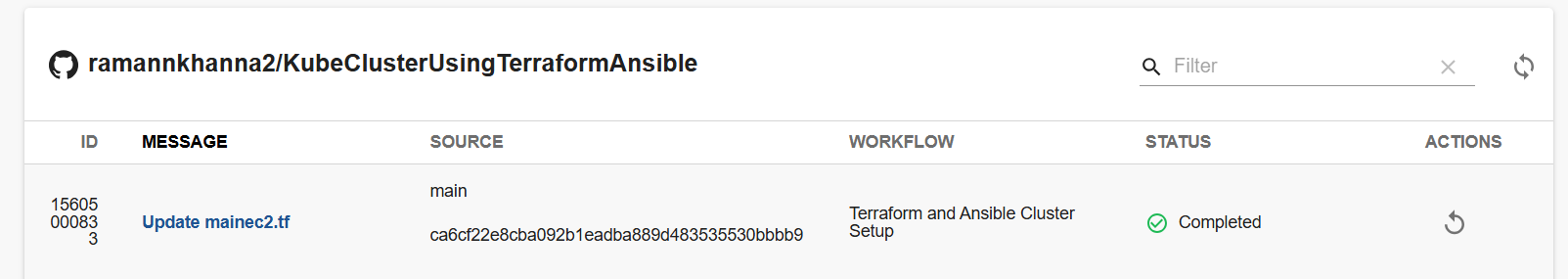
**📌 Platform Engineering Value Recap**

* **Developer Experience (DX)** → Metrics are part of the developer workflow, not an afterthought.
* **Golden Path Compliance** → Every service follows the same monitoring standard.
* **Self-Service Observability** → Devs don’t rely on ops teams to view metrics.
* **Reduced MTTR** → Troubleshooting happens faster with in-context metrics.
* **IDP Maturity** → This makes your Backstage portal a *true* single entry point for both code & operational insights.

**Lab 15: Integrate KubeClusterUsingTerraformAnsible into Backstage for One-Click Cluster Creation**

* Already have the setup of kubeadm cluster creation in <https://github.com/ramannkhanna2/KubeClusterUsingTerraformAnsible.git>
* Thers the catalog-info.yaml inside .
* Register it as a component in backstage with below url :
* <https://github.com/ramannkhanna2/KubeClusterUsingTerraformAnsible/blob/main/catalog-info.yaml>
* Now u shud see your github actions one click workflow to setup kubeadm 3 node Kubernetes cluster .





**Lab 16 : Gatekeeper + GitOps(argoCD) + Backstage Integration**

**📦 Pre-Requirements**

| **Component** | **Status** | **Notes** |
| --- | --- | --- |
| 🧠 Kubeadm cluster | ✅ Ready | Already provisioned by you |
| 🔐 OPA Gatekeeper | ✅ We'll install it | Policy enforcement |
| 📁 ArgoCD | ⏳ To be installed | GitOps sync engine |
| 🐙 GitHub Repo | ✅ You have it | Your raman-micro or Helm app |
| 🎛️ Backstage | ✅ Running | We’ll add ArgoCD plugin |

**✅ Phase 1: Enforce Gatekeeper Policies (Cluster Level)**

* Increase capacity of all nodes to 2 vcpus , 4 gb ram atlease for all nodes

**🧱 Step 1.1: Install Gatekeeper**

kubectl apply -f https://raw.githubusercontent.com/open-policy-agent/gatekeeper/release-3.12/deploy/gatekeeper.yaml

Verify:

bash

CopyEdit

kubectl get pods -n gatekeeper-system

Wait for all pods (controller, audit) to be Running.

* Create a github repo named kube+gatekeeper+argo and clone it to add all files in it

**🚫 Step 1.2: Create ConstraintTemplate: Block Untrusted Image Registry**

root@ip-172-31-27-88:~/kube-gatekeeper-argo# cat template-image-policy.yaml

apiVersion: templates.gatekeeper.sh/v1beta1

kind: ConstraintTemplate

metadata:

name: k8sallowedrepos

spec:

crd:

spec:

names:

kind: K8sAllowedRepos

targets:

- target: admission.k8s.gatekeeper.sh

rego: |

package k8sallowedrepos

violation[{"msg": msg}] {

container := input.review.object.spec.containers[\_]

not startswith(container.image, "docker.io/raman")

msg := sprintf("container image '%v' is not from allowed registry", [container.image])

}

Apply:

bash

CopyEdit

kubectl apply -f template-image-policy.yaml

|  |  |
| --- | --- |
| template-image-policy.yaml | Defines *how* to validate container images |

|  |  |
| --- | --- |
| constraint-image-policy.yaml | Activates enforcement for real objects like Pods |

**⚙️ Step 1.3: Apply Constraint: Only Allow docker.io/raman**

# constraint-image-policy.yaml

apiVersion: constraints.gatekeeper.sh/v1beta1

kind: K8sAllowedRepos

metadata:

name: allowed-docker-raman

spec:

match:

kinds:

- apiGroups: [""]

kinds: ["Pod"]

parameters: {}

bash

CopyEdit

kubectl apply -f constraint-image-policy.yaml

**🔐 You now have:**

**✅ A real-time Kubernetes admission control policy:**

❌ Blocks **all Pods** that do **not** use images from docker.io/raman

**💡 Step 1.4: Add Another Constraint — Require CPU & Memory Limits**

# template-limits-required.yaml

apiVersion: templates.gatekeeper.sh/v1beta1

kind: ConstraintTemplate

metadata:

name: k8srequiredresources

spec:

crd:

spec:

names:

kind: K8sRequiredResources

targets:

- target: admission.k8s.gatekeeper.sh

rego: |

package k8srequiredresources

violation[{"msg": msg}] {

container := input.review.object.spec.containers[\_]

not container.resources.limits.cpu

msg := "Missing CPU limit"

}

violation[{"msg": msg}] {

container := input.review.object.spec.containers[\_]

not container.resources.limits.memory

msg := "Missing memory limit"

}

bash

CopyEdit

kubectl apply -f template-limits-required.yaml

You’ve now successfully created a **second Gatekeeper ConstraintTemplate**:

bash

CopyEdit

✅ constrainttemplate.templates.gatekeeper.sh/k8srequiredresources created

This policy will enforce that **all Pods must define both cpu and memory limits**, which is a **crucial best practice** for Kubernetes multi-tenancy and resource fairness.

Constraint:

yaml

CopyEdit

# constraint-limits.yaml

apiVersion: constraints.gatekeeper.sh/v1beta1

kind: K8sRequiredResources

metadata:

name: require-cpu-memory-limits

spec:

match:

kinds:

- apiGroups: [""]

kinds: ["Pod"]

bash

CopyEdit

kubectl apply -f constraint-limits.yaml

✅ This means your cluster will now **block any Pod that doesn’t define both cpu and memory limits** for every container.

**🧪 Optional Test: Validate Enforcement**

Here’s a **negative test** to confirm Gatekeeper is enforcing the rule:

yaml

CopyEdit

# no-limits.yaml

apiVersion: v1

kind: Pod

metadata:

name: no-limits

spec:

containers:

- name: busybox

image: docker.io/raman/busybox:latest

command: ["sleep", "3600"]

Try to apply:

bash

CopyEdit

kubectl apply -f no-limits.yaml

You should see:

pgsql

CopyEdit

Error from server (Forbidden): admission webhook "validation.gatekeeper.sh" denied the request: [Missing CPU limit, Missing memory limit]

**✅ Current Setup Already Ensures**

| **Layer** | **Tool** | **Enforces Policy?** |
| --- | --- | --- |
| Kubernetes | OPA Gatekeeper | ✅ Denies at admission |

Your cluster will now:

* ❌ Reject any pod without resources.limits
* ❌ Reject any image not starting with docker.io/raman

Which means:

**No manifest can ever be deployed to your cluster if it violates the rules.**

**✅ Goal: GitOps + Backstage Integration for raman-micro**

We’ll now:

1. 🔁 Use **ArgoCD** to GitOps-deploy your k8s/\*.yaml to the cluster
2. 🔍 Hook that into **Backstage** so devs can visually see:
   * Sync status
   * Health
   * Drift

**🚀 Phase 3: GitOps with ArgoCD**

**🔧 Prerequisites Recap**

| **Requirement** | **Status** |
| --- | --- |
| ArgoCD installed (namespace argocd) | ✅ TobeDone |
| raman-micro GitHub repo | ✅ Exists |
| Manifests under k8s/ | ✅ TobeDone |
| Gatekeeper enforcing image + limits | ✅ Active |

Let's create a proper k8s/ directory in your raman-micro GitHub repo with fully compliant Kubernetes manifests that:

* ✅ Use docker.io/raman/raman-micro image
* ✅ Include CPU & memory limits
* ✅ Pass Gatekeeper constraints
* ✅ Are deployable via ArgoCD

**📁 Folder Structure to Add to Your Repo**

markdown

CopyEdit

raman-micro/

└── k8s/

├── deployment.yaml

└── service.yaml

**📄 k8s/deployment.yaml**

yaml

CopyEdit

apiVersion: apps/v1

kind: Deployment

metadata:

name: raman-micro

labels:

app: raman-micro

spec:

replicas: 1

selector:

matchLabels:

app: raman-micro

template:

metadata:

labels:

app: raman-micro

spec:

containers:

- name: raman-micro

image: docker.io/raman/raman-micro:latest

ports:

- containerPort: 5000

resources:

limits:

cpu: "250m"

memory: "256Mi"

**📄 k8s/service.yaml**

yaml

CopyEdit

apiVersion: v1

kind: Service

metadata:

name: raman-micro

spec:

selector:

app: raman-micro

ports:

- protocol: TCP

port: 80

targetPort: 5000

type: NodePort

**✅ What to Do Next**

**🔧 Step 1: Add These Files to Your GitHub Repo raman-micro**

git add .

git commit -m "Add k8s manifests for ArgoCD sync"

git push origin main

* **Install argocd CLI**

Run the following on your Ubuntu (or Amazon Linux) system:

bash

CopyEdit

# Download latest ArgoCD CLI (v2.11.3 as of June 2025)

VERSION=$(curl -s https://api.github.com/repos/argoproj/argo-cd/releases/latest | grep tag\_name | cut -d '"' -f 4)

curl -sSL -o argocd "https://github.com/argoproj/argo-cd/releases/download/${VERSION}/argocd-linux-amd64"

chmod +x argocd

sudo mv argocd /usr/local/bin/

Verify install:

bash

CopyEdit

argocd version

**✅ Yes, the ArgoCD UI runs as a pod in your Kubernetes cluster.**

When you installed ArgoCD using:

kubectl create namespace argocd

kubectl apply -n argocd -f https://raw.githubusercontent.com/argoproj/argo-cd/stable/manifests/install.yaml

This deployed all ArgoCD components — including the **UI (argocd-server)** — into your cluster.

**📦 ArgoCD Core Components (All Run as Pods)**

You can check with:

bash

CopyEdit

kubectl get pods -n argocd

* You will notice pods will not get created due to the gatekeeper policy

❌ Problem: Gatekeeper is Blocking ArgoCD Pods

**✅ Disable the policy temporarily ; will activate again after argo cd installation.**

Only if you're still testing.

bash

CopyEdit

Kubectl get constraint

kubectl delete k8srequiredresources.constraints.gatekeeper.sh require-cpu-memory-limits

kubectl delete k8sallowedrepos.constraints.gatekeeper.sh allowed-docker-raman

* Again install argocd :

kubectl delete -n argocd -f https://raw.githubusercontent.com/argoproj/argo-cd/stable/manifests/install.yaml

kubectl apply -n argocd -f https://raw.githubusercontent.com/argoproj/argo-cd/stable/manifests/install.yaml

* Kubectl get pods -n argocd
* Now again enable gatekeeper constraints :

root@ip-172-31-27-88:~/kube-gatekeeper-argo# k apply -f constraint-image-policy.yaml

k8sallowedrepos.constraints.gatekeeper.sh/allowed-docker-raman created

root@ip-172-31-27-88:~/kube-gatekeeper-argo# k apply -f constraint-limit.yaml

k8srequiredresources.constraints.gatekeeper.sh/require-cpu-memory-limits created

root@ip-172-31-27-88:~/kube-gatekeeper-argo# k get constraints

NAME ENFORCEMENT-ACTION TOTAL-VIOLATIONS

k8sallowedrepos.constraints.gatekeeper.sh/allowed-docker-raman

NAME ENFORCEMENT-ACTION TOTAL-VIOLATIONS

k8srequiredresources.constraints.gatekeeper.sh/require-cpu-memory-limits

* To expose argocd :

root@ip-172-31-27-88:~/kube-gatekeeper-argo# k edit -n argocd svc argocd-server

service/argocd-server edited

root@ip-172-31-27-88:~/kube-gatekeeper-argo# k get svc -n argocd argocd-server

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

argocd-server NodePort 10.99.164.110 <none> 80:30786/TCP,443:31194/TCP 4m51s

Let’s now help you **log into the UI** — ArgoCD uses:

* **Username:** admin
* **Password:** stored in a Kubernetes **secret** (base64-encoded)

**✅ Step-by-Step: Get ArgoCD Admin Credentials**

**Get the admin password :**

kubectl -n argocd get secret argocd-initial-admin-secret -o jsonpath="{.data.password}" | base64 -d && echo

* **Login argocd using cli as well :**

root@ip-172-31-27-88:~/kube-gatekeeper-argo# argocd login 15.228.172.48:30786 --username admin --password V9kGvXX0SH1Fw7Fn --insecure

'admin:login' logged in successfully

Context '15.228.172.48:30786' updated

**📦 Step 3.1: Create ArgoCD App for raman-micro**

Replace YOUR\_GITHUB\_URL if needed.

bash

CopyEdit

argocd app create raman-micro \

--repo https://github.com/ramannkhanna2/raman-micro.git \

--path k8s \

--dest-server https://kubernetes.default.svc \

--dest-namespace default \

--sync-policy automated \

--self-heal \

--auto-prune

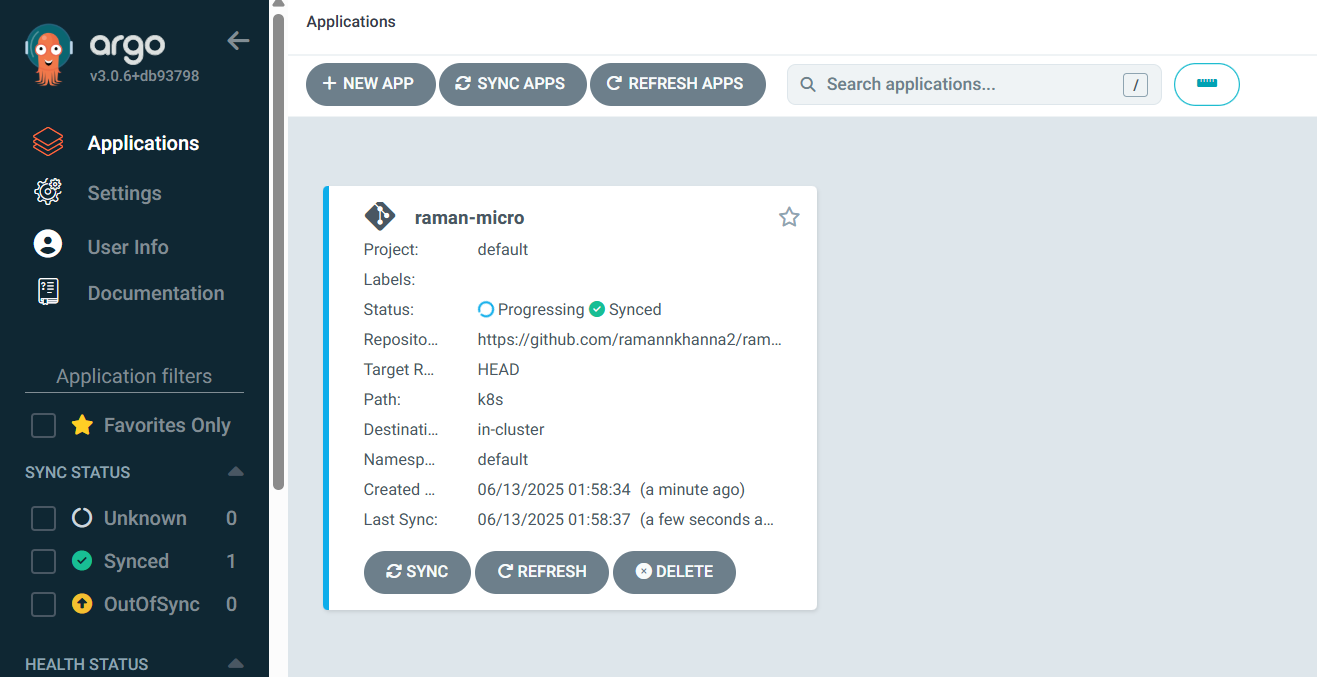
Explanation:

* --path k8s: we’re using plain manifests, not Helm
* --sync-policy automated: auto-deploys when Git changes
* --self-heal: ensures drifted resources are reconciled
* --auto-prune: deletes removed resources from Git

root@ip-172-31-27-88:~/kube-gatekeeper-argo# k get application -n argocd

NAME SYNC STATUS HEALTH STATUS

raman-micro Synced Progressing



**🔄 Step 3.2: Trigger a Sync (optional)**

bash

CopyEdit

argocd app sync raman-micro

* **Check if argo cd setup is working fine by making changes on raman-micro/k8s github repo and se eif its automatically replicating on the kubecluster using argi cd ..**
* **If fails check for the right image from dockerhub and update in raman-micro github repo i.e : ramann123/raman-micro:latest**

**🧭 Phase 4: Backstage + ArgoCD Plugin Integration**

* <https://backstage.io/plugins/>
* https://roadie.io/backstage/plugins/argo-cd/?utm\_source=backstage.io&utm\_medium=marketplace&utm\_campaign=argo-cd

Install the plugin into Backstage.

cd /root/backstage-app/ramanapp/packages/app

yarn add @roadiehq/backstage-plugin-argo-cd

Add proxy config to the app-config.yaml file

proxy:

'/argocd/api':

target: https://<your-argocd-instance>/api/v1/

changeOrigin: **true**

# only if your argocd api has self-signed cert

secure: **false**

headers:

Cookie:

$env: ARGOCD\_AUTH\_TOKEN

For above generate token and save as env :

bash

CopyEdit

argocd account generate-token

Add argoCD widget to your overview page

// packages/app/src/components/catalog/EntityPage.tsx

import {

EntityArgoCDOverviewCard,

isArgocdAvailable

} from '@roadiehq/backstage-plugin-argo-cd';

const overviewContent = (

<Grid container spacing={3} alignItems="stretch">

...

<EntitySwitch>

<EntitySwitch.Case if={e => Boolean(isArgocdAvailable(e))}>

<Grid item sm={4}>

<EntityArgoCDOverviewCard />

</Grid>

</EntitySwitch.Case>

</EntitySwitch>

...

</Grid>

);

Add annotation to the yaml config file of a component

metadata:

annotations:

argocd/app-name: <your-app-name>

Get and provide **ARGOCD\_AUTH\_TOKEN** as env variable in following format

ARGOCD\_AUTH\_TOKEN='argocd.token=<token>'

**🔄 Step 4.5: Restart Backstage**

bash

CopyEdit

cd ~/backstage-app/raman-app

yarn start

Go to:  
🧭 Backstage → raman-micro → ArgoCD tab

✅ You should now see:

* Sync Status
* Drift
* Health
* Last sync timestamp
* Commit SHA

**🎉 You Now Have:**

| **Capability** | **Tool** | **Status** |
| --- | --- | --- |
| GitOps deployment | ArgoCD | ✅ |
| Admission policy enforcement | Gatekeeper | ✅ |
| Visual deployment status | Backstage | ✅ |
| Real-time sync & drift detection | ArgoCD UI | ✅ |