## 1. Overview

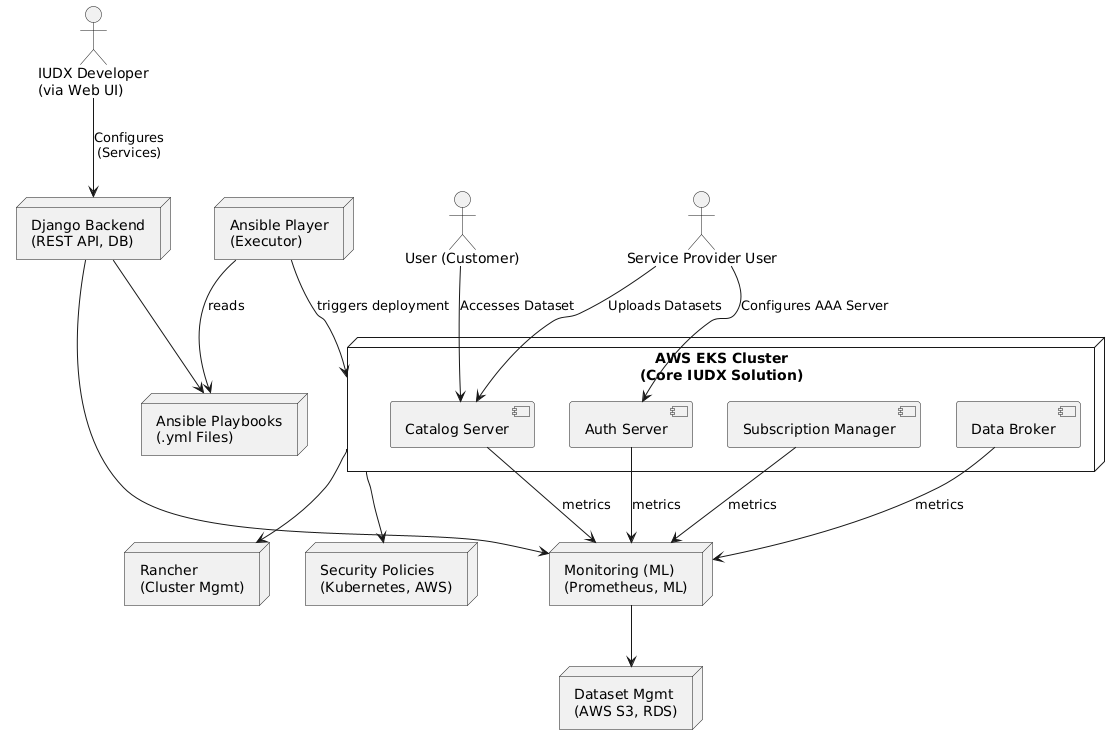
The IUDX Deployment Configuration Service is a web-based platform designed to streamline the deployment, configuration, monitoring, and management of IUDX components in a Kubernetes environment hosted on AWS. It enables service providers to configure components, manage security policies, monitor deployment statistics, and handle datasets via an intuitive web interface. The system leverages Rancher for Kubernetes management, Ansible for automation, Django for the web application, and machine learning for predictive analytics, ensuring scalability, security, and ease of use.

## 2. System Requirements

* Web Interface: A Django-based web application for user interaction, configuration, and management.
* Deployment Automation: Ansible playbooks integrated with Rancher to deploy and configure Kubernetes resources (e.g., DBs, AAA servers).
* Monitoring: Real-time monitoring of Kubernetes cluster metrics (nodes, load, alerts) with ML-based overload prediction.
* Security Policies: Configuration of Kubernetes-native security policies and integration with AWS security services.
* Dataset Management: Upload, update, and delete datasets stored in a managed database or object storage.
* Technologies: Python, Django, Kubernetes, Rancher, Ansible, AWS (EKS, S3, RDS), Machine Learning (e.g., scikit-learn, TensorFlow).

## 3. System Architecture

The system is divided into five core components, each addressing specific functionality. The architecture is deployed on AWS EKS, managed by Rancher, with Ansible automating configurations and Django powering the web interface.



**Figure 01: Architecture Diagram**

## 4. Component Details

### 4.1 Web Interface with Customer

A Django-based web interface allows users to configure IUDX components via forms. Configurations are saved to a PostgreSQL database (via AWS RDS) or exported as YAML files for version control.

**Web Form / Wizard**:

Developer fills a form with:

Org name

Required DBs (MongoDB/MySQL/etc)

Datasets to preload

AAA needs (OAuth, ImmunDB, etc.)

Resource quotas

Store it in a backend DB or generate a YAML/JSON spec.

**YAML or JSON Upload Option** (for advanced users):

Let them upload configuration like:

yaml

CopyEdit

org: acme-corp

dbs:

- mongodb

- immundb

aaa: true

quotas:

cpu: 2

memory: 4Gi

datasets:

- name: traffic\_data

file: traffic.csv

**Store Configurations in DB**  
You should persist this configuration (Postgres or any RDBMS) for tracking, auditing, and re-provisioning.

**Frontend** : HTML, Tailwind CSS, JavaScript for dynamic forms.

**Backend** : Django REST Framework for API endpoints, PostgreSQL for storage.

**Features** : Form-based input for component selection (e.g., DBs, AAA servers), configuration preview, and export to YAML.

**Storage** : Configurations stored in AWS RDS; YAML files in AWS S3 for backup and GitOps integration.

### 4.2 Deployment Automation

This component uses Ansible playbooks to deploy and configure IUDX components in an AWS EKS cluster managed by Rancher. It reads configurations from the database or YAML files.

**Tools** : Ansible for automation, Rancher for Kubernetes cluster management, AWS EKS for hosting.

**Workflow**:

* User submits configuration via web interface.
* Django backend saves configuration to RDS or generates YAML.
* Ansible playbooks (invoked via Django) deploy resources (e.g., PostgreSQL, Keycloak for AAA) using Kubernetes manifests.
* Rancher manages cluster scaling and health.

**Security**: Uses Kubernetes Secrets for sensitive data, integrated with AWS Secrets Manager.

### 4.3 Monitoring and Statistics

A monitoring component provides real-time deployment statistics and predictive analytics for overload detection using machine learning.

**Tools** : Prometheus for metrics collection, Grafana for visualization, Python (scikit-learn/TensorFlow) for ML-based predictions.

**Features**:

**Real-time metrics :** Number of nodes, CPU/memory usage, pod status.

**Alerts** : Configurable thresholds for resource usage.

**ML Model** : Predicts potential overload based on historical load patterns.

**Integration** : Prometheus deployed via Rancher, with data fed to a Django API for web display.

### 4.4 Security Policy Configuration

Users can configure Kubernetes-native security policies and AWS security services via the web interface.

**Tools** : Kubernetes Network Policies, OPA/Gatekeeper, AWS IAM, AWS WAF.

**Features** :

* Define network policies to restrict pod communication.
* Configure role-based access control (RBAC) for cluster resources.
* Integrate with AWS IAM for user authentication.
* Apply WAF rules for web application protection.

**Workflow** : Policies configured via forms, stored in RDS, and applied via Ansible playbooks.

### 4.5 Dataset Management

A component for uploading, updating, and deleting datasets, stored in AWS S3 or RDS.

**Tools**: AWS S3 for file storage, AWS RDS for metadata, Django for API and UI.

**Features**:

* Upload datasets via web interface (supports CSV, JSON, etc.).
* Update or delete datasets with version control.
* Metadata management in PostgreSQL for search and retrieval.

**Security**: AWS S3 bucket policies and IAM roles for access control.

## 5. Technology Stack

| **Component** | **Technologies** |
| --- | --- |
| Web Interface | Django, HTML, Tailwind CSS, JavaScript |
| Deployment | Ansible, Rancher, Kubernetes (AWS EKS) |
| Monitoring | Prometheus, Grafana, Python (scikit-learn/TensorFlow) |
| Security | Kubernetes Network Policies, OPA/Gatekeeper, AWS IAM, AWS WAF |
| Dataset Management | AWS S3, AWS RDS, Django |
| Backend | Python, Django REST Framework, PostgreSQL |
| Infrastructure | AWS (EKS, RDS, S3, Secrets Manager), Rancher |

## 6. Implementation Considerations

**Scalability**: Use AWS EKS auto-scaling and Rancher for cluster management to handle varying loads.

**Security**: Implement Kubernetes Secrets, AWS Secrets Manager, and OPA/Gatekeeper for policy enforcement. Disable SSL inspection for self-signed certificates.

[](<https://docs.redhat.com/en/documentation/red_hat_ansible_automation_platform/2.3/html/red_hat_ansible_automation_platform_planning_guide/platform-system-requirements>)

**Storage**: Use PersistentVolumeClaims (PVCs) for stateful applications like PostgreSQL, with AWS EBS for persistence.[](<https://blog.jetbrains.com/pycharm/2024/03/deploying-django-apps-in-kubernetes/>)

**Monitoring**: Deploy Prometheus and Grafana via Rancher for real-time metrics. Train ML models on historical data for overload prediction.

**Ansible Integration:** Use Ansible playbooks for declarative configuration, integrated with Django via subprocess calls or Ansible AWX.

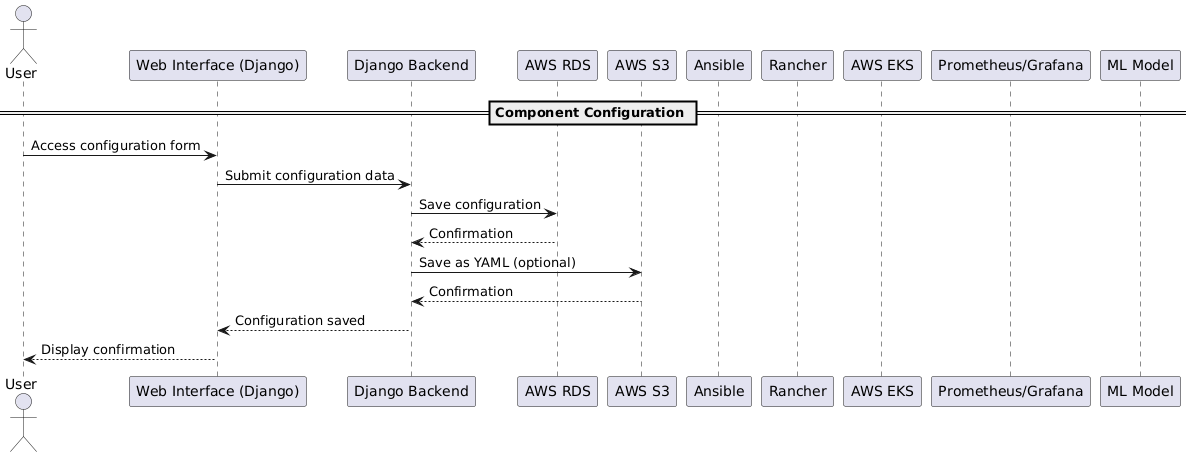
[](<https://www.ansiblepilot.com/articles/install-ansible-awx-operator-for-kubernetes-k8s-and-openshift-ocp-ansible-awx/>)

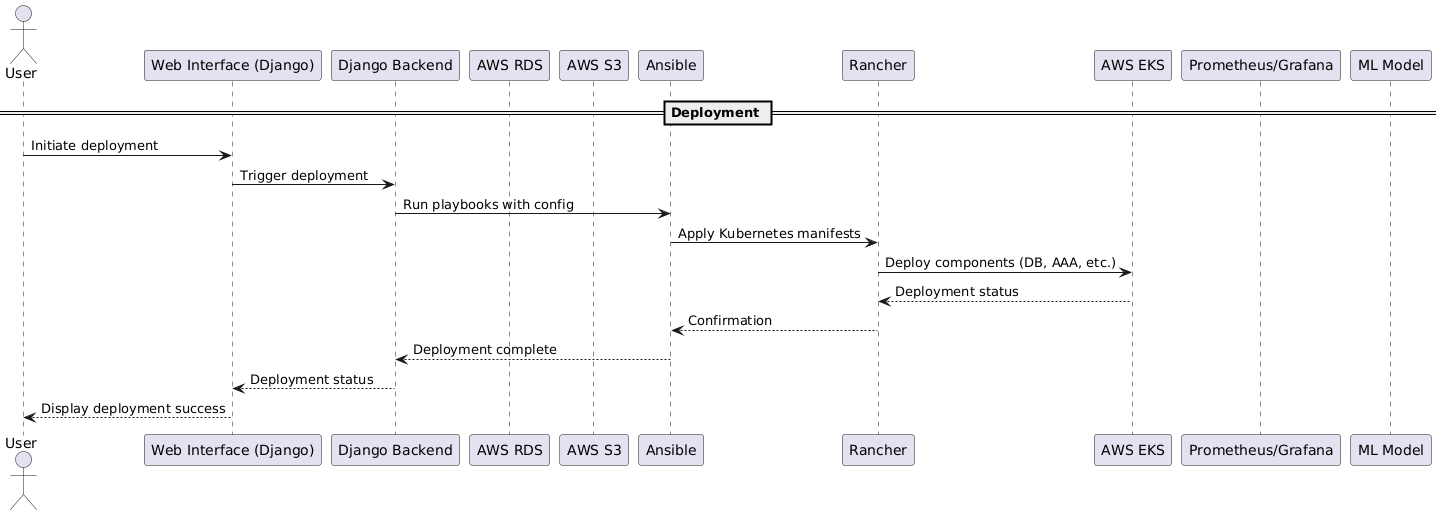
**GitOps**: Store YAML configurations in a Git repository for version control and CI/CD integration.

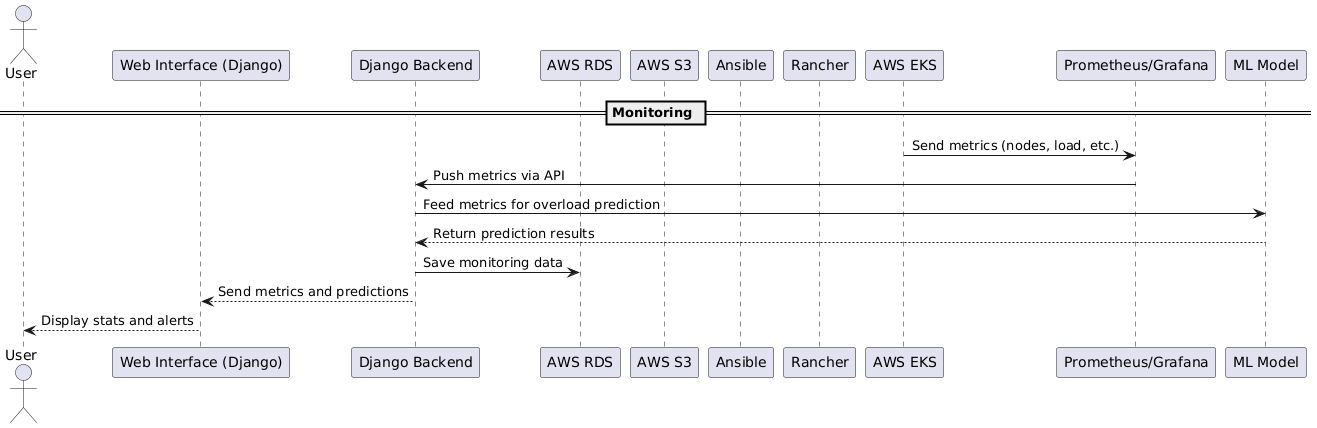
[](<https://octopus.com/devops/kubernetes-deployments/kubernetes-management-tools/>)

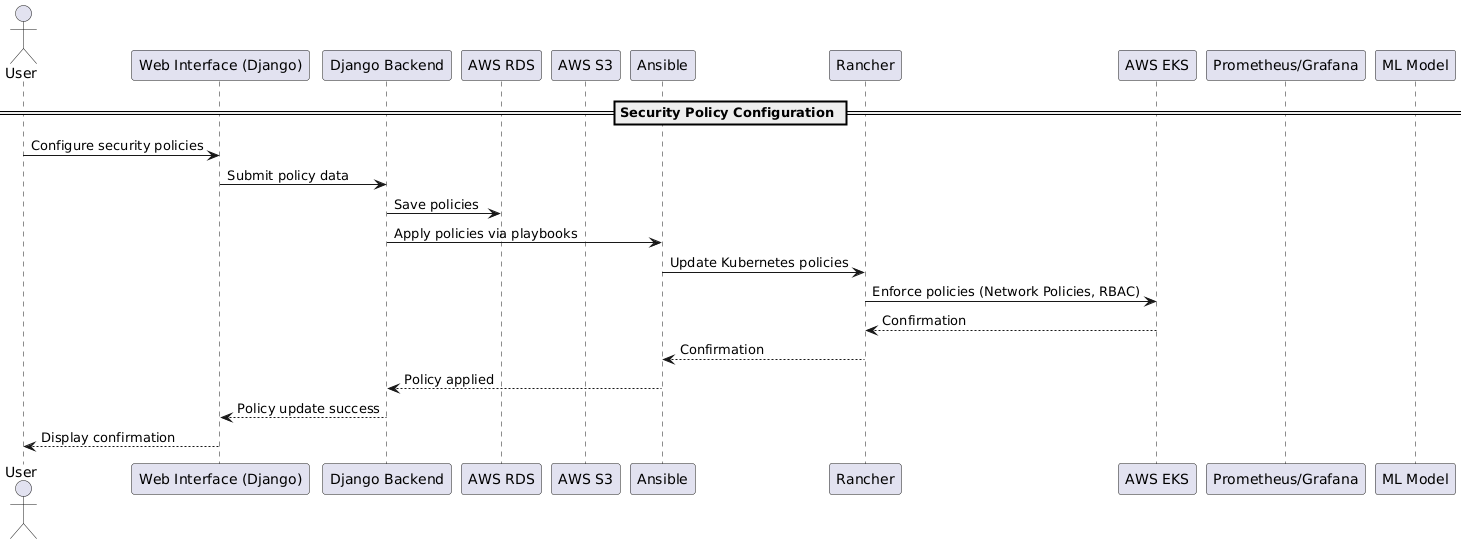
## 7. Deployment Workflow

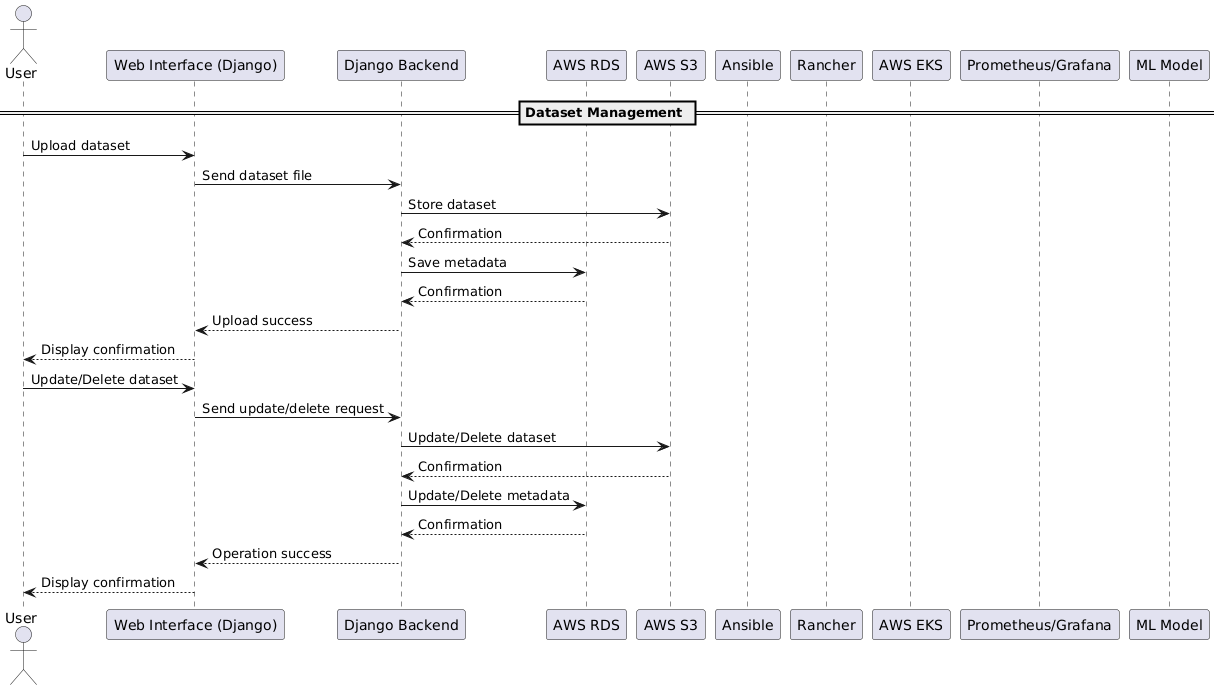
* User logs into the Django web interface and configures components via forms.
* Configurations are validated and saved to AWS RDS or exported as YAML to S3.
* Ansible playbooks are triggered via Django to deploy resources to AWS EKS using Rancher.
* Prometheus monitors the cluster, feeding data to Grafana and ML models for analytics.
* Users configure security policies and manage datasets via the web interface, with changes applied via Ansible.
* Monitoring dashboard displays real-time stats and ML-based overload predictions.











## 8. Future Enhancements

Multi-Cloud Support: Extend to GCP and Azure using Kubespray. (<https://computingforgeeks.com/deploy-production-kubernetes-cluster-with-ansible/>)

Advanced ML: Enhance overload prediction with real-time learning.

GitOps Integration: Fully automate deployments with ArgoCD.

API Access: Expose REST APIs for programmatic configuration.

User Management: Add role-based access control in the web interface.

## 9. References

Deploying Django Apps in Kubernetes, The PyCharm Blog

[](https://blog.jetbrains.com/pycharm/2024/03/deploying-django-apps-in-kubernetes/)

How to Manage Kubernetes with Ansible, Spacelift

[](https://spacelift.io/blog/ansible-kubernetes)

Kubernetes Setup Using Ansible and Kubespray, ComputingForGeeks

[](https://computingforgeeks.com/deploy-production-kubernetes-cluster-with-ansible/)

Kubernetes Management Tools, Octopus

[](https://octopus.com/devops/kubernetes-deployments/kubernetes-management-tools/)

Red Hat Ansible Automation Platform, Red Hat Documentation

[](https://docs.redhat.com/en/documentation/red\_hat\_ansible\_automation\_platform/2.3/html/red\_hat\_ansible\_automation\_platform\_planning\_guide/platform-system-requirements)

Install Ansible AWX Operator for Kubernetes, Ansible Pilot

[](https://www.ansiblepilot.com/articles/install-ansible-awx-operator-for-kubernetes-k8s-and-openshift-ocp-ansible-awx/)

Deliverables:

1. A binary / script (Python/Ansible)

* Check readiness part
* Deploy PaaS (Bring up Rancher + Kubernetes Cluster)
* Deploy dependant components (Ex: immunDB, Postgres)
* Deploy IUDX Components (AAA server, Catlog server, etc…)
* Start a service: Run Monitoring module (runs in parallel) IP : port
  + Progress and state of all PaaS, third party comps, IUDX components

Step1:

**Prerequisites**

* **Nodes**: 6 Linux servers (3 for master nodes, 3 for worker nodes) with static IPs, SSH access, and root/sudo privileges.
  + Example IPs: 192.168.1.10–12 (masters), 192.168.1.13–15 (workers).
  + Minimum requirements: Masters (2 vCPUs, 8GB RAM), Workers (1 vCPU, 2GB RAM).
* **Load Balancer**: A Layer 4 load balancer (e.g., HAProxy, MetalLB, or cloud-based like AWS ELB) to distribute traffic to master nodes on ports 6443 (Kubernetes API) and 9345 (RKE2 node registration).
* **DNS**: A DNS record (e.g., rke2.example.com) pointing to the load balancer’s IP.
* **Ansible**: Installed on your control machine with SSH access to all nodes.

Ansible playbook to set up a high-availability (HA) RKE2 cluster with 3 master nodes and 3 worker nodes (as shown in Figure 01), install Helm, and deploy Rancher on the cluster.

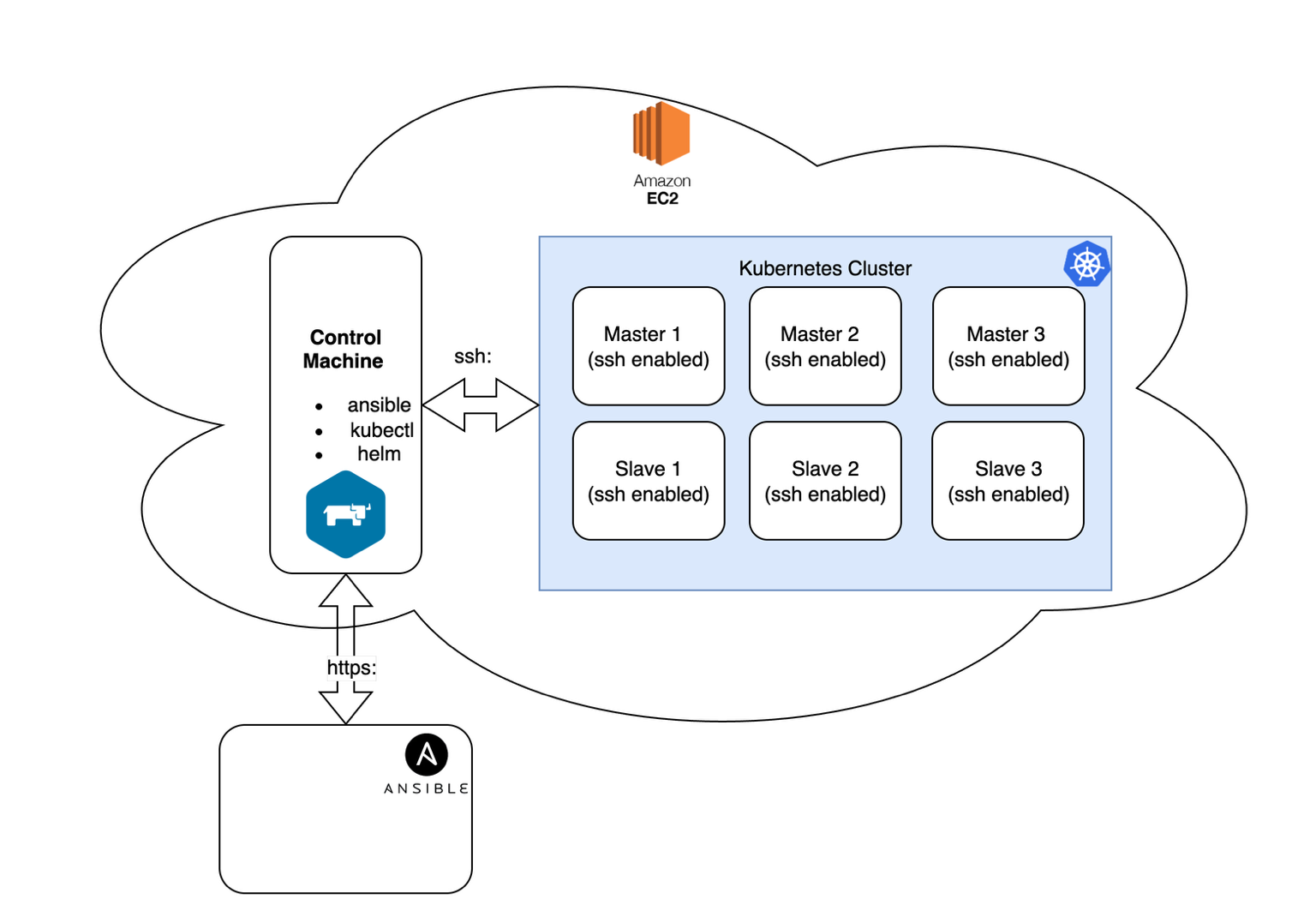


Figure 01: Kubernetes Cluster with 3+3

This assumes you have 6 nodes (Linux-based, e.g., Ubuntu 22.04 or CentOS Stream 8) with SSH access, root privileges, and a load balancer for HA.

The setup will use Ansible to streamline the process, leveraging best practices from Rancher’s documentation and community resources.

2 Write Ansible to install :

* Make ready ansible inventory (inventory.yml)
* Make playbook ready (deploy\_rke2\_rancher.yml)

**Configuration:**



Flow of Execution:

A diagram of a software process

AI-generated content may be incorrect.

**Recommendations:**

**Use Terraform for Cloud Resource Provisioning:** While Ansible is great for configuring software *on* a machine, **Terraform** is the industry standard for provisioning the underlying infrastructure itself (e.g., the VMs for the K8s nodes, VPCs, subnets, firewall rules).

**Flow:** The Django/Rust backend commits the configuration to Git. This commit triggers a CI/CD pipeline which then runs the helm install/upgrade command. This gives you better auditing, manual approval gates if needed, and a clear history of all deployment actions.

**Use Kubernetes Operators:** This is one of the most impactful suggestions. An **Operator** is a custom controller that encodes human operational knowledge for a specific application. Instead of managing a MongoDB StatefulSet yourself, you would use a **MongoDB Operator**.

**Set up the "Three Pillars of Observability":**

1. **Logging:** Deploy a centralized logging stack like **Fluentd/FluentBit + Elasticsearch + Kibana (EFK)** to collect logs from all your pods. This makes debugging across services possible.
2. **Metrics:** Use **Prometheus** to scrape metrics from your applications and the cluster itself. Use **Grafana** to create dashboards for visualizing this data (CPU/RAM usage, database query rates, etc.).
3. **Tracing:** For understanding request flows in a microservices environment, consider implementing distributed tracing with **Jaeger** or **OpenTelemetry**.

* **Cost Management and Governance:**
  + **Resource Quotas:** Use Kubernetes **ResourceQuotas** and **LimitRanges** at the namespace level to prevent any single service provider from consuming all cluster resources.
  + **Cost Monitoring:** Install a tool like **Kubecost** or **OpenCost**. This will give you a detailed breakdown of how much each namespace, deployment, or service is costing in real-time, which is invaluable for billing or showback.

 **Trigger provisioning with Ansible/K8s jobs**  
After submission, run Ansible or spawn a K8s Job with this config.

1. **Using “rancher” and “kubernetes” to deploy multiple clusters**

**✅ Prerequisites**

**On All VMs (Master + Worker nodes):**

* OS: Ubuntu 20.04+ / CentOS 8 / RHEL 8
* Docker installed (20.10+)
* Open ports:
  + **TCP 22 (SSH)**
  + **TCP 6443** (K8s API)
  + **TCP 2379-2380** (etcd)
  + **TCP 10250, 10251, 10252** (kubelet & controller)
  + **TCP/UDP 8472** (flannel/VXLAN)
  + **TCP 30000-32767** (NodePort services)

⚠️ Ensure all nodes can communicate with each other (internal IPs).

**🧑‍💻 Step-by-Step Guide**

**✅ Step 1: Install Rancher Server (on a separate VM or Docker)**

You can run Rancher using Docker:

bash

CopyEdit

docker run -d --restart=unless-stopped \

-p 80:80 -p 443:443 \

--name rancher \

rancher/rancher:latest

📍 Access it in browser: https://<rancher-server-ip>

**✅ Step 2: Log in and Create a Cluster**

1. Visit Rancher UI
2. Go to Cluster Management → Create → **Create a Custom Cluster**
3. Give cluster name: e.g., custom-k8s
4. Under **Cluster Configuration**:
   * Choose **RKE2** or **RKE1**
   * Choose **etcd + control plane + worker** (or separate roles)
5. Click **Next**, and Rancher will give you **node registration commands** like:

bash

CopyEdit

sudo docker run -d --privileged --restart=unless-stopped \

-v /etc/rancher:/etc/rancher \

-v /var/run:/var/run \

rancher/rancher-agent:latest \

--server https://<rancher-server-ip> \

--token <registration-token> \

--etcd --controlplane --worker

**✅ Step 3: Run Agent on Each VM**

On each VM:

* Run the registration command
* Adjust roles per node:
  + On master nodes: --etcd --controlplane --worker
  + On worker nodes: --worker

**✅ Step 4: Rancher Builds the Cluster**

* Rancher connects to agents
* Installs Kubernetes components
* In ~5–10 minutes your cluster is ready

You’ll see status as Active in Rancher UI.

**✅ Step 5: Download Kubeconfig**

In Rancher UI:

* Click on your cluster → Kubeconfig File
* Save it as ~/.kube/config
* Now you can run:

bash

CopyEdit

kubectl get nodes

**🧩 Bonus: Sample Topology**

lua

CopyEdit

+-----------------+ +-----------------+ +-----------------+

| master-1 | <---> | master-2 | <--->| master-3 |

| etcd + ctl + wrk| | etcd + ctl + wrk| | etcd + ctl + wrk|

+-----------------+ +-----------------+ +-----------------+

\ | /

\ / \

+---------------------+--------------------------+

| | |

+---------------+ +---------------+ +---------------+

| worker-1 | | worker-2 | | worker-3 |

+---------------+ +---------------+ +---------------+

**🛠️ Tips**

* Install Docker: curl -fsSL https://get.docker.com | sh
* For firewalls, open required ports (esp. 2379, 6443, 10250)
* Use internal IPs for node registration
* Make sure Rancher has access to all VMs

**2. Configure and Monitoring tool:**

Directory structure:

**# monitoring\_app/**

# ├── manage.py

# ├── monitoring/

# │ ├── \_\_init\_\_.py

# │ ├── settings.py

# │ ├── urls.py

# │ ├── wsgi.py

# │ └── asgi.py

# ├── monitor/

# │ ├── \_\_init\_\_.py

# │ ├── admin.py

# │ ├── apps.py

# │ ├── migrations/

# │ ├── models.py

# │ ├── templates/

# │ │ ├── base.html

# │ │ ├── index.html

# │ │ └── config.html

# │ ├── urls.py

# │ └── views.py

# └── requirements.txt

# monitoring\_app/monitoring/settings.py

INSTALLED\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

'monitor',

]

MIDDLEWARE = [

'django.middleware.security.SecurityMiddleware',

'django.contrib.sessions.middleware.SessionMiddleware',

'django.middleware.common.CommonMiddleware',

'django.middleware.csrf.CsrfViewMiddleware',

'django.contrib.auth.middleware.AuthenticationMiddleware',

'django.contrib.messages.middleware.MessageMiddleware',

'django.middleware.clickjacking.XFrameOptionsMiddleware',

]

ROOT\_URLCONF = 'monitoring.urls'

TEMPLATES = [

{

'BACKEND': 'django.template.backends.django.DjangoTemplates',

'DIRS': [],

'APP\_DIRS': True,

'OPTIONS': {

'context\_processors': [

'django.template.context\_processors.debug',

'django.template.context\_processors.request',

'django.contrib.auth.context\_processors.auth',

'django.contrib.messages.context\_processors.messages',

],

},

},

]

DATABASES = {

'default': {

'ENGINE': 'django.db.backends.sqlite3',

'NAME': BASE\_DIR / 'db.sqlite3',

}

}

STATIC\_URL = '/static/'

STATIC\_ROOT = BASE\_DIR / 'static'

# monitoring\_app/monitoring/urls.py

from django.contrib import admin

from django.urls import path, include

urlpatterns = [

path('admin/', admin.site.urls),

path('', include('monitor.urls')),

]

# monitoring\_app/monitor/models.py

from django.db import models

class MonitoringConfig(models.Model):

prometheus\_url = models.URLField(default='http://prometheus:9090')

refresh\_interval = models.IntegerField(default=30)

cpu\_threshold = models.IntegerField(default=80)

alert\_webhook = models.URLField(blank=True)

def \_\_str\_\_(self):

return f"Monitoring Config (Prometheus: {self.prometheus\_url})"

# monitoring\_app/monitor/views.py

from django.shortcuts import render, redirect

from django.contrib import messages

from .models import MonitoringConfig

import requests

from kubernetes import client, config

import json

def load\_k8s\_config():

try:

config.load\_incluster\_config() # For running inside Kubernetes

except:

config.load\_kube\_config() # For local development

def index(request):

load\_k8s\_config()

v1 = client.CoreV1Api()

# Fetch Kubernetes nodes

nodes = v1.list\_node().items

clusters = [

{

'name': node.metadata.name,

'status': next(c for c in node.status.conditions if c.type == 'Ready').status,

'nodes': 1

} for node in nodes

]

# Fetch Prometheus metrics

config\_obj = MonitoringConfig.objects.first() or MonitoringConfig()

try:

response = requests.get(

f"{config\_obj.prometheus\_url}/api/v1/query",

params={'query': 'up{job="iudx-services"}'}

)

services = response.json().get('data', {}).get('result', [])

except:

services = []

messages.error(request, "Failed to fetch Prometheus metrics")

return render(request, 'monitor/index.html', {

'clusters': clusters,

'services': services,

'config': config\_obj

})

def config\_view(request):

config\_obj = MonitoringConfig.objects.first() or MonitoringConfig()

if request.method == 'POST':

config\_obj.prometheus\_url = request.POST.get('prometheus\_url', config\_obj.prometheus\_url)

config\_obj.refresh\_interval = request.POST.get('refresh\_interval', config\_obj.refresh\_interval)

config\_obj.cpu\_threshold = request.POST.get('cpu\_threshold', config\_obj.cpu\_threshold)

config\_obj.alert\_webhook = request.POST.get('alert\_webhook', config\_obj.alert\_webhook)

config\_obj.save()

messages.success(request, "Configuration saved successfully")

return redirect('index')

return render(request, 'monitor/config.html', {'config': config\_obj})

# monitoring\_app/monitor/urls.py

from django.urls import path

from . import views

urlpatterns = [

path('', views.index, name='index'),

path('config/', views.config\_view, name='config'),

]

# monitoring\_app/monitor/templates/base.html

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>{% block title %}IUDX Cluster Monitoring{% endblock %}</title>

<script src="https://cdn.tailwindcss.com"></script>

</head>

<body class="bg-gray-100 font-sans">

<div class="container mx-auto p-6">

<h1 class="text-3xl font-bold text-blue-800 mb-6">IUDX Cluster Monitoring</h1>

{% if messages %}

<div class="mb-4">

{% for message in messages %}

<div class="p-4 {% if message.tags == 'error' %}bg-red-100 text-red-800{% else %}bg-green-100 text-green-800{% endif %} rounded">

{{ message }}

</div>

{% endfor %}

</div>

{% endif %}

{% block content %}{% endblock %}

</div>

</body>

</html>

# monitoring\_app/monitor/templates/index.html

{% extends 'monitor/base.html' %}

{% block title %}Cluster Dashboard{% endblock %}

{% block content %}

<div class="bg-white shadow-lg rounded-lg p-6">

<div class="grid grid-cols-1 md:grid-cols-2 gap-6">

<div>

<h2 class="text-xl font-semibold mb-4">Cluster Status</h2>

<ul class="list-disc list-inside">

{% for cluster in clusters %}

<li class="mb-2">{{ cluster.name }}: {{ cluster.status }} (Nodes: {{ cluster.nodes }})</li>

{% empty %}

<li>No clusters found</li>

{% endfor %}

</ul>

<h2 class="text-xl font-semibold mt-6 mb-4">Service Metrics</h2>

<ul class="list-disc list-inside">

{% for service in services %}

<li class="mb-2">{{ service.metric.job }}: {{ service.value.1|floatformat:0|yesno:"Up,Down" }}</li>

{% empty %}

<li>No service metrics available</li>

{% endfor %}

</ul>

</div>

<div>

<h2 class="text-xl font-semibold mb-4">Current Configuration</h2>

<p><strong>Prometheus URL:</strong> {{ config.prometheus\_url }}</p>

<p><strong>Refresh Interval:</strong> {{ config.refresh\_interval }} seconds</p>

<p><strong>CPU Threshold:</strong> {{ config.cpu\_threshold }}%</p>

<p><strong>Alert Webhook:</strong> {{ config.alert\_webhook|default:"None" }}</p>

<a href="{% url 'config' %}" class="mt-4 inline-block bg-blue-600 text-white px-4 py-2 rounded hover:bg-blue-700">Edit Configuration</a>

</div>

</div>

</div>

{% endblock %}

# monitoring\_app/monitor/templates/config.html

{% extends 'monitor/base.html' %}

{% block title %}Configure Monitoring{% endblock %}

{% block content %}

<div class="bg-white shadow-lg rounded-lg p-6 max-w-md">

<h2 class="text-xl font-semibold mb-4">Configure Monitoring</h2>

<form method="post">

{% csrf\_token %}

<div class="mb-4">

<label class="block text-sm font-medium">Prometheus URL</label>

<input type="url" name="prometheus\_url" value="{{ config.prometheus\_url }}" class="w-full p-2 border rounded">

</div>

<div class="mb-4">

<label class="block text-sm font-medium">Refresh Interval (seconds)</label>

<input type="number" name="refresh\_interval" value="{{ config.refresh\_interval }}" class="w-full p-2 border rounded">

</div>

<div class="mb-4">

<label class="block text-sm font-medium">CPU Threshold (%)</label>

<input type="number" name="cpu\_threshold" value="{{ config.cpu\_threshold }}" class="w-full p-2 border rounded">

</div>

<div class="mb-4">

<label class="block text-sm font-medium">Alert Webhook URL</label>

<input type="url" name="alert\_webhook" value="{{ config.alert\_webhook }}" class="w-full p-2 border rounded">

</div>

<button type="submit" class="bg-blue-600 text-white px-4 py-2 rounded hover:bg-blue-700">Save Configuration</button>

</form>

</div>

{% endblock %}

# monitoring\_app/requirements.txt

django==4.2.11

requests==2.31.0

kubernetes==26.1.0

gunicorn==20.1.0