EdgeDelta EKS & Kiro Workshop

External link - https://catalog.workshops.aws/eks-web-application/en-US

This guide demonstrates how to use Kiro AI to build, containerize, and deploy a full-stack microservices application on Amazon EKS with an Application Load Balancer - all through natural language prompts.

What You'll Build

- **Frontend**: Nginx-based web UI
- **Backend**: Python Flask API that calls external APIs
- **Infrastructure**: EKS cluster with ALB Ingress
- **Container Registry**: Amazon ECR
- **Load Balancer**: Application Load Balancer (ALB)

Architecture: User → ALB → Frontend Pod → Backend Pod → External API

Prerequisites

- AWS Account with appropriate permissions (or AWS Workshop Studio account)
- `AWS CLI` installed on your local machine
- `eksctl` installed (Kiro can help install this)
- `kubectl` installed (Kiro can help install this)
- `finch` or `Docker` installed (for building container images)
- Kiro AI assistant
- Github Account

Kiro Download and Set Up:

Link to download: here
https://kiro.dev/downloads/

Getting Started: Two Approaches

Approach 1: Using AWS Workshop Studio (Recommended)

If you're using AWS Workshop Studio, you can skip the VSCode Server setup and work directly from your local machine with Kiro:

1. **Get AWS credentials from Workshop Studio** - Copy your temporary credentials (Access Key ID, Secret Access Key, and Session Token)
2. **Configure AWS CLI profile on your local machine:**

```
aws configure set aws_access_key_id "[AWS_ACCESS_KEY_ID]"
--profile kiro-eks-workshop
aws configure set aws_secret_access_key
"[AWS_SECRET_ACCESS_KEY]" --profile kiro-eks-workshop
aws configure set aws_session_token "[AWS_SESSION_TOKEN]"
--profile kiro-eks-workshop
aws configure set region "us-west-1" --profile kiro-eks-workshop
```

1. **Create a Kiro steering rule** to automatically use this profile:

Prompts to Kiro:

```
Create a steering rule that tells you to always use the AWS CLI
profile
"kiro-eks-workshop" for all AWS commands in this workspace.
When you are checking for specific versions of a package, make
sure you are trying
to use the "--version" flag is "version" is not returning
anything.
```

This creates a `.kiro/steering/aws-profile.md` file that ensures Kiro uses the correct AWS profile throughout the workshop.

Another prompt to create a steering rule

```
I want you to create another steering file. Whenever we are
developing something that
is going to be pushed to the cloud. I want you to create local
containers and copies
that will allow us to test what the application is going to look
like in production.
```

1. **Start building!** Now you can simply ask Kiro to perform AWS operations without repeatedly specifying the profile. For example:
    - "Help me create an EKS cluster using eksctl with these config details: [paste config]"
    - "Build and push my container images to ECR"

    - "Install the AWS Load Balancer Controller"

Benefits of this approach:

- Work from your own machine with your preferred Kiro setup
- No need to configure VSCode Server
- Kiro remembers to use the workshop credentials automatically
- More flexible and faster iteration

Approach 2: Using Your Own AWS Account

If you're using your own AWS account, simply configure your AWS CLI as usual and either:

- Use your default profile, or
- Create a named profile and tell Kiro to use it (or create a steering rule as shown above)

Step 1: Create an EKS Cluster

Prompt (if you set up the steering rule):

```
Help me create an EKS cluster using eksctl with this
configuration:
- Cluster name: eks-kiro-demo
- Region: us-west-1
- Kubernetes version: 1.33
- 3 m5.large nodes in a managed node group
- Enable CloudWatch logging
```

Prompt (if you didn't set up the steering rule):

```
Help me create an EKS cluster using eksctl with this
configuration:
- Cluster name: eks-kiro-demo
- Region: us-west-1
- Kubernetes version: 1.33
- 3 m5.large nodes in a managed node group
- Enable CloudWatch logging
- Use the AWS CLI profile: kiro-eks-workshop
```

What Kiro will do:

- Create an `eks-kiro-demo-cluster.yaml` configuration file
- Run `eksctl create cluster` command
- Wait for cluster creation (15-20 minutes)
- Configure kubectl to connect to your cluster

Pro Tip: With the steering rule in place, you can simply paste the cluster configuration from the workshop and say "Create an EKS cluster with this config" - Kiro will handle the rest!

Once Complete:

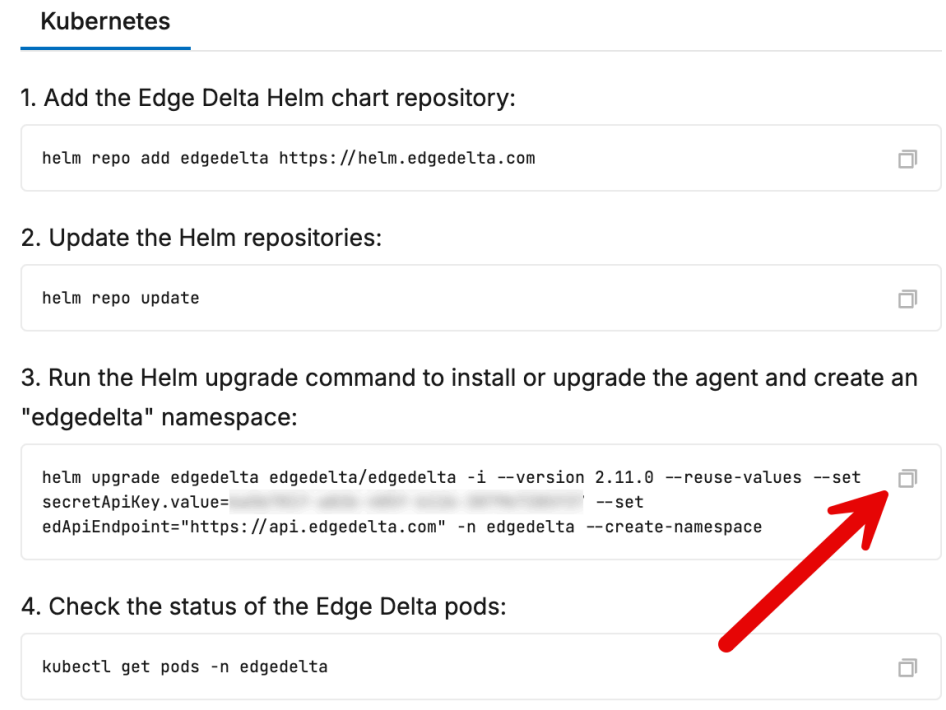
- Open up AWS console and verify EKS assets available

Step 2: Create EdgeDelta Account

- Navigate to https://edgedelta.com/start-free-trial
- Once initial sign-up is complete, navigate to Admin -> Plan -> Upgrade to Pro -> Promo Code: EDGEFRIENDS20

Step 3: Add Edge Delta K8s Connector

- In the Edge Delta App, navigate to the AI Team Tab
  - Click “Connectors” -> Add connector
  - Choose Kubernetes Logs
    - Name: Kiro Workshop
    - Select “Kubernetes” from the Target Environments list
    - Click “Create Connector”
    - Copy installation command



  - From the kiro app run:

- Now, deploy the edge delta agents to this cluster using edge delta's helm chart and the following information: [PASTE COMMAND]

Step 4: Create Edge Delta API Token

- Navigate to Admin -> API Tokens
- Click “Create Token”
- Name: Kiro
- Create a new token with resource: Monitors and Access Type Read/Write
- Resources: All current and future monitors
- Copy and save your api token
- Copy and save your orgID

Step 5: Vibe Code to Create Simple Front End Flappy Kiro

Use this prompt:

```
I want to build a Flappy Bird clone called Flappy Kiro.
Flappy Kiro is an arcade-style game in which the player controls
a ghost called
Ghosty, which moves persistently to the right. They are tasked
with navigating
Ghosty through a series of walls that have equally sized gaps
placed at random
heights. Ghosty automatically descends and only ascends when the
player taps the
spacebar. Each successful pass through a pair of walls awards
the player one point.
Colliding with a wall or the ground ends the gameplay.
I want you to have 3 different difficulty levels for the user to
choose from, "Easy",
"Medium" and "Hard".
This application should be connected to a simple
HTML/Javascript front end that is
communicating to a simple Python/Flask backend/API that will be
used to store a global
leaderboard that allows users to submit their high score with a
username of their
choosing (with some username validation insuring that it
is appropriate).
These high scores should be stored in a simple .json file that
is stored on the
```

```
backend server.
Make sure that there is good OTEL logging for both the front end
and backend
applications?
```

What Kiro will do:

- Update the front end index.html file to create a simple application that can be played on your browser

---

Step 6: Build the Microservices Application

Prompt:

```
Using the existing code that we just created for the
Flappy Kiro front and backend,
create a simple microservices application for Kubernetes with
the following architecture:

**Frontend Service:**
- Simple web UI (HTML/JavaScript) served by Nginx
- Makes API calls to the backend service that will be storing
the flappy kiro leaderboard
- Exposed via an AWS Application Load Balancer (ALB) for
external access

**Backend Service:**
- REST API (Python Flask) storing the leaderboard data in a
.JSON file on the server
- Provides an endpoint that the frontend calls
- Fetches data stored as a leaderboard for users and returns it
- Internal service (ClusterIP) - not directly accessible from
outside

**Kubernetes Resources Needed:**
- Deployments for both frontend and backend (2 replicas each)
- Services: frontend (NodePort) and backend (ClusterIP)
- Ingress resource configured for AWS ALB to route traffic to
frontend
- Container images should be built and pushed to Amazon ECR

**Traffic Flow:**
User → ALB (Ingress) → Frontend Service → Frontend Pod → Backend
Service → Backend Pod → Stored data for leaderboard
```

```
Please include:
- Dockerfiles for both services
- Open Telemetry Logging throughout the entire infrastructure
- Kubernetes manifests (deployments, services, ingress)
- Script to build and push images to ECR
```

What Kiro will create:

- `backend/Dockerfile` - Python Flask application
- `backend/app.py` - API with random data endpoint
- `backend/requirements.txt` - Python dependencies
- `frontend/Dockerfile` - Nginx web server
- `frontend/index.html` - Simple web UI
- `frontend/nginx.conf` - Nginx configuration with proxy to backend
- `k8s/backend-deployment.yaml` - Backend deployment and service
- `k8s/frontend-deployment.yaml` - Frontend deployment and service
- `k8s/ingress.yaml` - ALB Ingress configuration
- `build-and-push.sh` - Script to build and push images

Step 7: Build and Push Container Images

Prompt:

```
Build the frontend and backend Docker images for AMD64
architecture and push them to ECR using finch.
```

What Kiro will do:

- Login to Amazon ECR (using the kiro-eks-workshop profile if you set up the steering rule)
- Create ECR repositories for frontend and backend
- Build images with `--platform=linux/amd64` flag
- Tag and push images to ECR
- Update Kubernetes manifests with correct image URLs

Once Complete:

- Open up AWS console and verify ECR assets available

Step 8: Install AWS Load Balancer Controller

This is the most complex step. Use this prompt:

Prompt:

```
Install the AWS Load Balancer Controller on my EKS cluster
following these steps:

1. Associate IAM OIDC provider with the cluster
2. Download the IAM policy from:
https://raw.githubusercontent.com/kubernetes-sigs/aws-load-balan
cer-controller/v2.13.3/docs/install/iam_policy.json
3. Create IAM policy named AWSLoadBalancerControllerIAMPolicy
4. Create IAM service account for the controller
5. Install cert-manager for webhook certificates
6. Download and deploy the controller v2.13.3
7. Remove the ServiceAccount section from the manifest (lines
730-738)
8. Replace cluster name with eks-kiro-demo
9. Apply the IngressClass configuration
```

Note: If you set up the steering rule, you don't need to specify the AWS profile - Kiro will use it automatically!

What Kiro will do:

- Set up IAM OIDC provider
- Create necessary IAM policies and roles
- Install cert-manager
- Deploy AWS Load Balancer Controller
- Configure IngressClass for ALB
- Troubleshoot any permission issues

Common Issue: If you see webhook errors, tell Kiro:

```
I see webhook errors. A GitHub issue suggests deleting these to
fix:
kubectl delete mutatingwebhookconfigurations
cert-manager-webhook
kubectl delete ValidatingWebhookConfiguration
cert-manager-webhook
```

Step 9: Deploy the Application

Prompt:

```
Deploy the frontend and backend applications to the EKS cluster
and create the ALB ingress.
```

What Kiro will do:

- Apply backend deployment and service
- Apply frontend deployment and service

- Apply ingress resource
- Wait for ALB to be provisioned
- Verify pods are running

Step 10: Create Edge Delta Monitor

Prompt:

```
Using what you know about the application, create edge delta monitors using api-token:[paste your api token] and orgid:[paste your orgid]. Use Edge Delta's documentation and use their api to create the monitors.
```

Step 11: Test the Application

Prompt:

```
Test the application by:
1. Getting the ALB URL from the ingress
2. Curl the frontend homepage
3. Curl the backend API endpoint multiple times to verify random responses
```

What Kiro will do:

- Get the ALB DNS name
- Test the frontend UI
- Test the backend API
- Show you different random responses from the external API

Verification Commands

Once deployed, you can verify everything with these prompts:

```
Show me the status of all pods
Get the ALB URL for the frontend ingress
Check the AWS Load Balancer Controller logs
Describe the ingress resource to see events
```

Expected Results

After completion, you should have:
✅ EKS cluster with 3 worker nodes
✅ Frontend and backend pods running (2 replicas each)
✅ Container images in Amazon ECR
✅ Application Load Balancer provisioned
✅ Working application accessible via ALB URL
✅ Backend returning random data from external API
Test URLs:

- Frontend: `http://<alb-dns-name>/`
- Backend API: `http://<alb-dns-name>/api/data`

Troubleshooting with Kiro

If something goes wrong, simply describe the issue to Kiro:

```
The pods are in CrashLoopBackOff status. Check the logs and fix
the issue.
The ALB ingress is not getting an address. Check the controller
logs and troubleshoot.
I'm getting permission errors. What IAM permissions are missing?
```

Kiro will:

- Check logs and diagnostics
- Identify the root cause
- Suggest and implement fixes
- Verify the solution works

Key Learnings

This guide demonstrates how Kiro can:

1. **Understand complex requirements** - Translate high-level architecture into working code
2. **Generate multi-file projects** - Create Dockerfiles, Kubernetes manifests, and scripts
3. **Execute infrastructure commands** - Run AWS CLI, kubectl, and eksctl commands
4. **Troubleshoot issues** - Debug permission errors, pod failures, and configuration problems
5. **Iterate and fix** - Adapt when things don't work the first time

---

Cleanup

When you're done, clean up resources:
Prompt:

```
Delete the EKS cluster and all associated resources:
1. Delete the ingress
2. Delete the deployments and services
3. Delete the EKS cluster
4. Delete ECR repositories
5. Delete IAM policies and roles
```

Tips for Using Kiro with EKS

1. **Use steering rules** - Set up a steering rule for your AWS profile once, then forget about it
2. **Be specific about AWS profiles** - If not using steering rules, always mention which AWS CLI profile to use
3. **Specify architecture** - For EKS, build images with `--platform=linux/amd64`
4. **Ask for verification** - Request Kiro to check status after each major step
5. **Provide error messages** - Copy/paste error messages for faster troubleshooting
6. **Request explanations** - Ask "Why did this fail?" to understand issues
7. **Iterate naturally** - Treat Kiro like a DevOps teammate, not a script
8. **Paste configurations directly** - When working from workshop materials, just paste the config and ask Kiro to use it

---

Advanced Prompts

Once comfortable, try these advanced scenarios:

```
Add horizontal pod autoscaling to the backend based on CPU usage
Configure HTTPS on the ALB with an ACM certificate
Add a Redis cache between frontend and backend
Set up a CI/CD pipeline to automatically deploy on code changes
Implement health checks and readiness probes for both services
```

Conclusion

This guide showed how Kiro can accelerate EKS development by:

- Generating production-ready code from natural language

- Executing complex infrastructure commands
- Troubleshooting and fixing issues autonomously
- Documenting the entire process

The same conversational approach works for any cloud infrastructure task - just describe what you want to build, and let Kiro handle the implementation details.

Created with Kiro AI - Your AI-powered DevOps teammate