Archetype AI
AWS Marketplace Deployment Guide

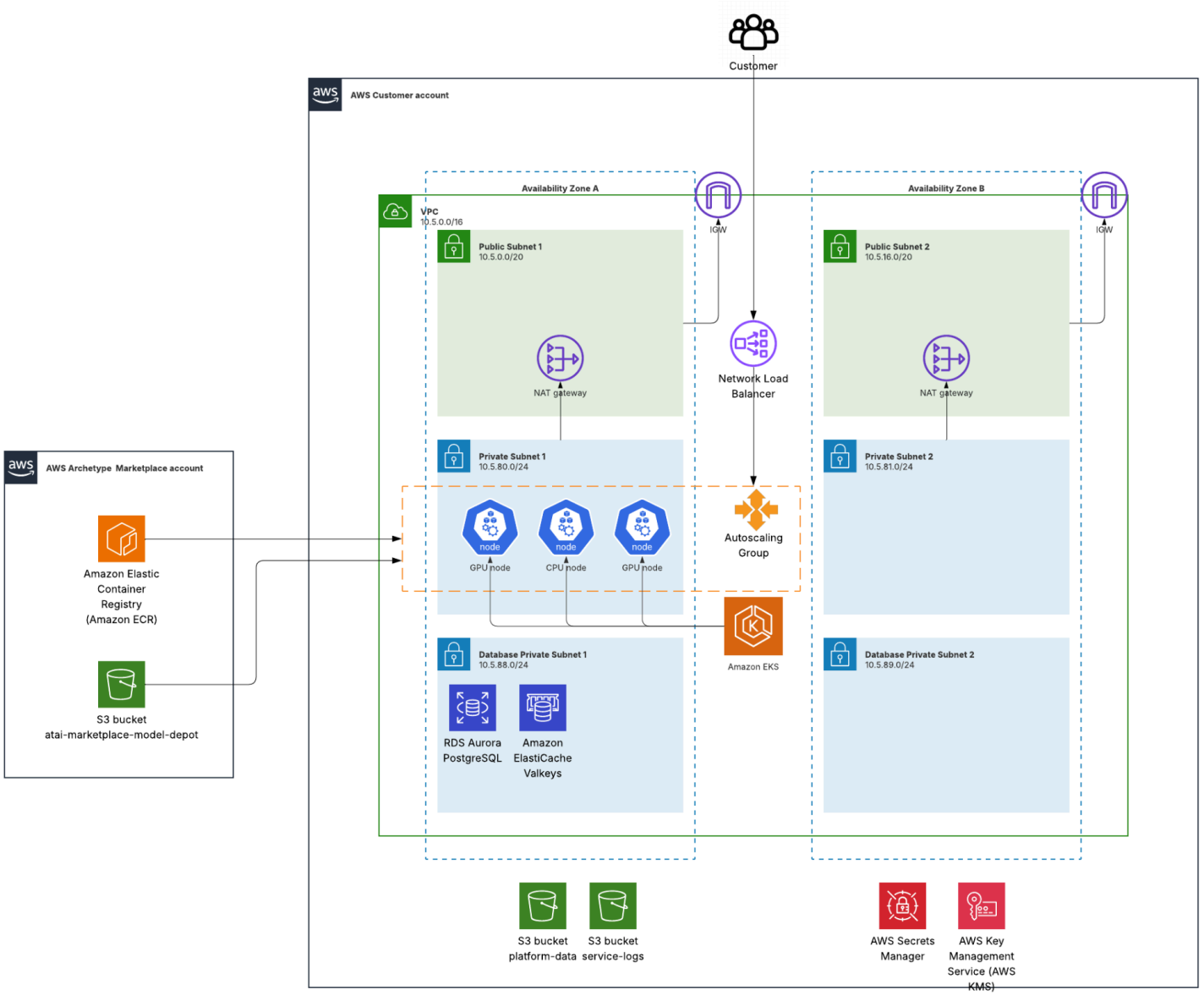
Version: 1.0.0-99

Executive summary

Archetype exists to unlock the potential of physical AI through Newton, a Large Behavior Model (LBM) that understands and reasons about the physical world using multimodal sensor data. Built on advanced machine learning techniques, Newton processes and interprets data from any sensor type while ensuring data privacy and enabling edge deployment. With a focus on the trillion-dollar sensor economy, Newton is positioned to transform industries such as construction and manufacturing by providing new insights into their physical operations.

Table of contents

Infrastructure Overview



atai Infrastructure

AWS License Manager setup

This Marketplace product uses **AWS License Manager** for proper operation. You **must enable and configure AWS License Manager** in your account before deployment.

Failure to do so will prevent the product from functioning correctly. For more information on how to set up License Manager correctly, see **Appendix: License Manager Setup**.

AWS infrastructure prerequisites

This document outlines the AWS infrastructure prerequisites that must be deployed before installing atai-platform helm chart.

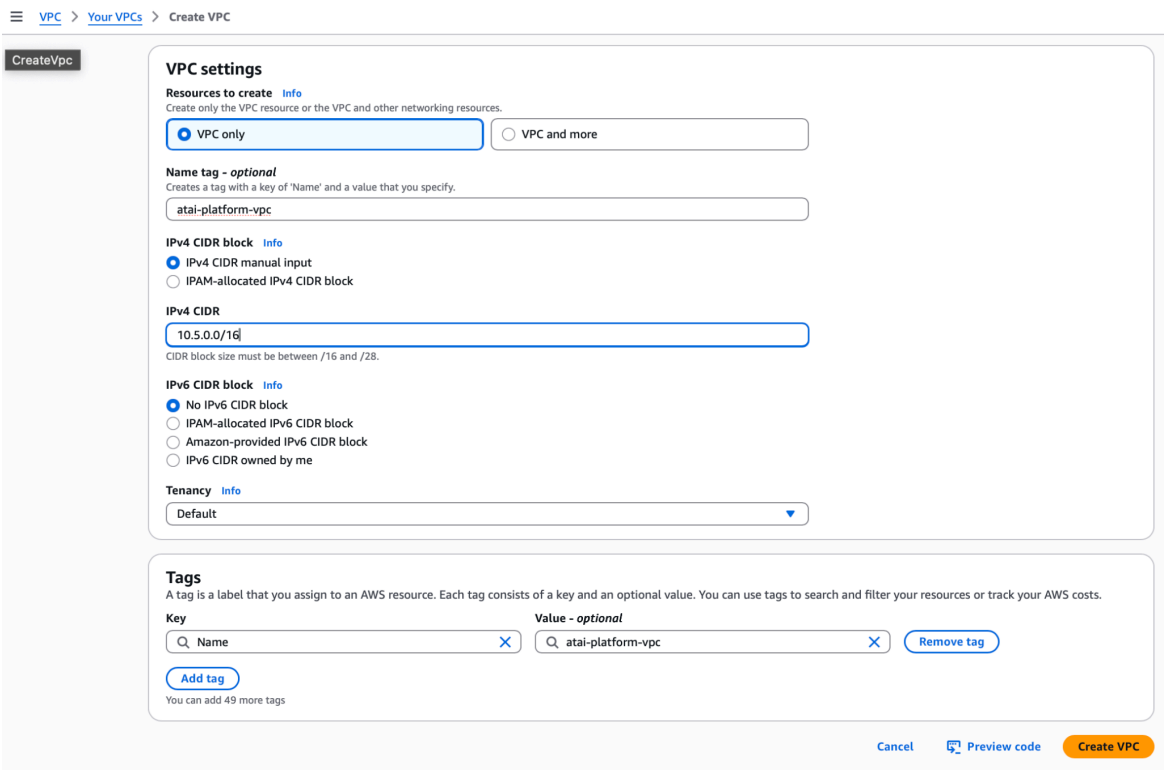
Note: Before starting the creation of the following resources, create a ***secrets.ini*** file based on the ***secrets.ini.template***. This file will be required on Step 3: Kubernetes secrets required for the atai-platform services.

VPC configuration

Create VPC with Public, Private, and Database Subnets

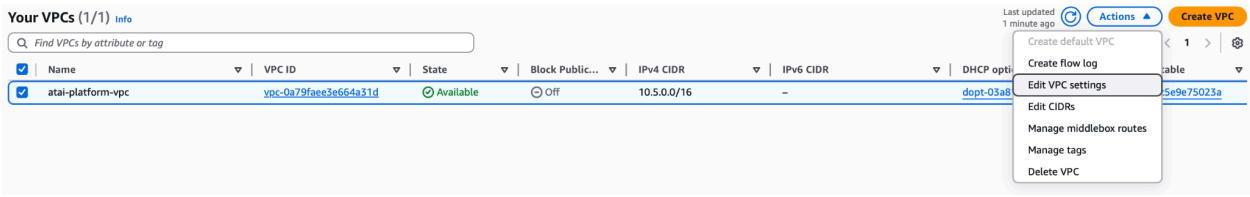
Step 1: Create the VPC (Manual Configuration)

1. Go to VPC Dashboard → Your VPCs → Create VPC
2. Select VPC only (not "VPC and more")
3. Configure:
    a. Name tag: atai-platform-vpc
    b. IPv4 CIDR block: 10.5.0.0/16
    c. IPv6 CIDR block: No IPv6 CIDR block
    d. Tenancy: Default
4. Click Create VPC

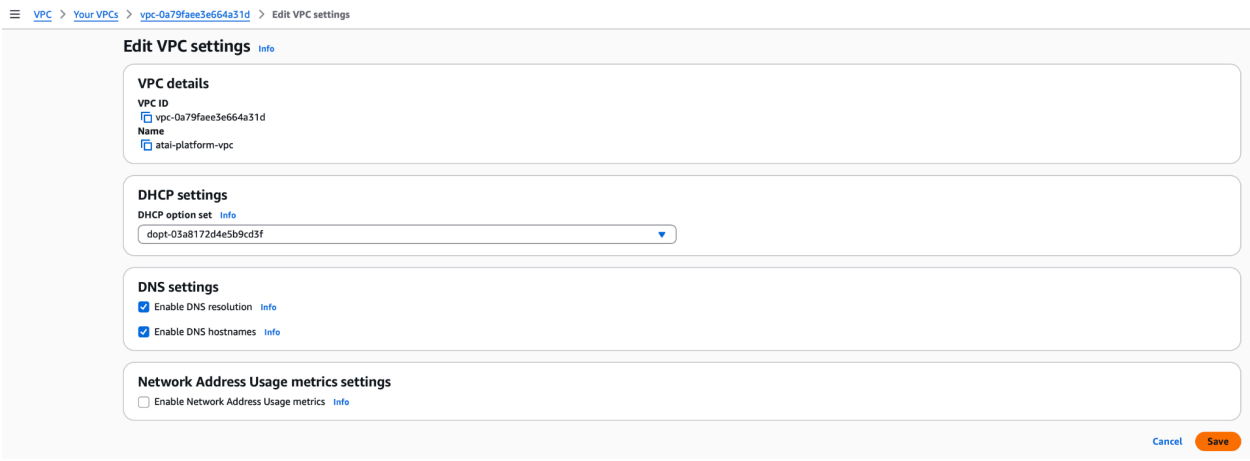


Step 1.1 Enable DNS Hostnames (if not already enabled)

1. Go to Your VPCs → select your VPC
2. Actions → Edit VPC settings

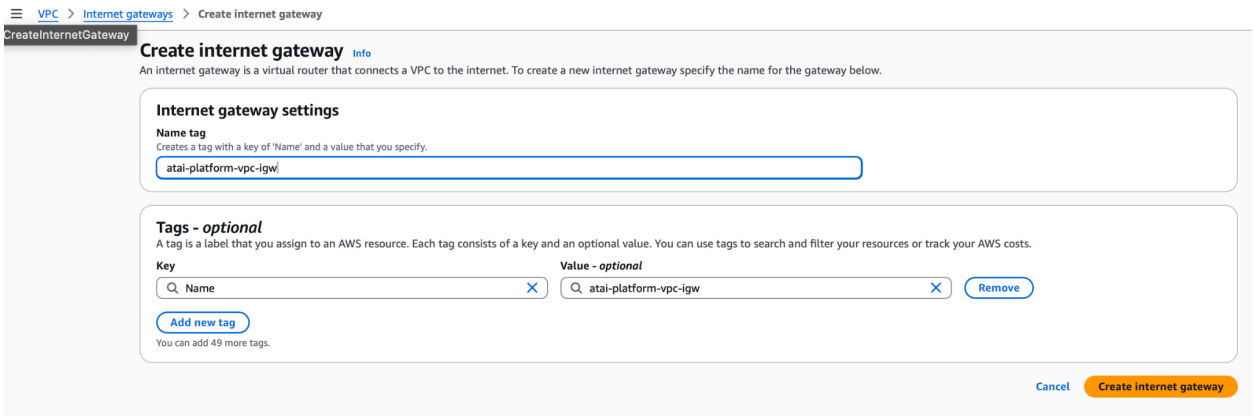


3. Enable DNS hostnames and DNS resolution
4. Save

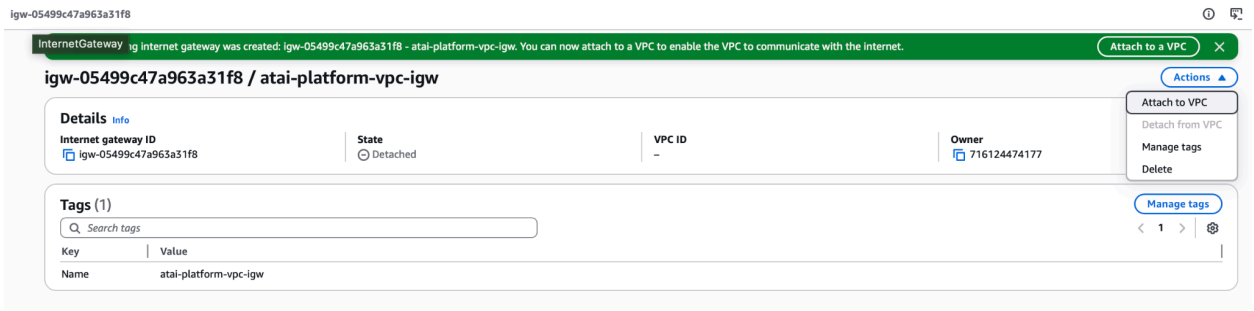


Step 2: Create Internet Gateway

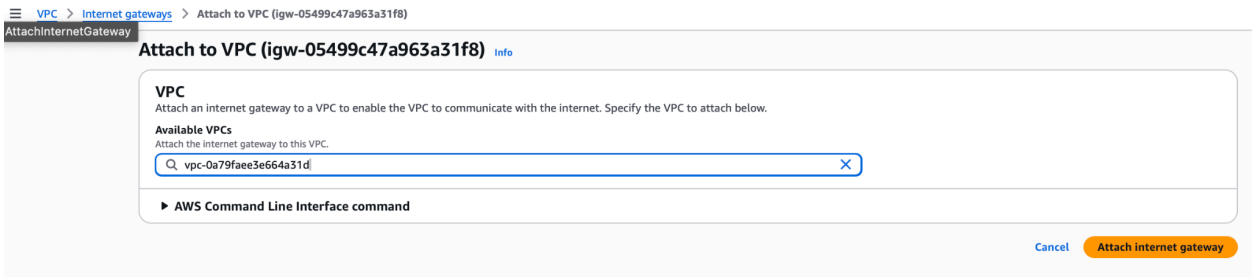
1. Go to VPC Dashboard → Go to Internet Gateways → Create internet gateway
2. Name tag: atai-platform-vpc-igw
3. Click Create internet gateway



4. Select it → Actions → Attach to VPC



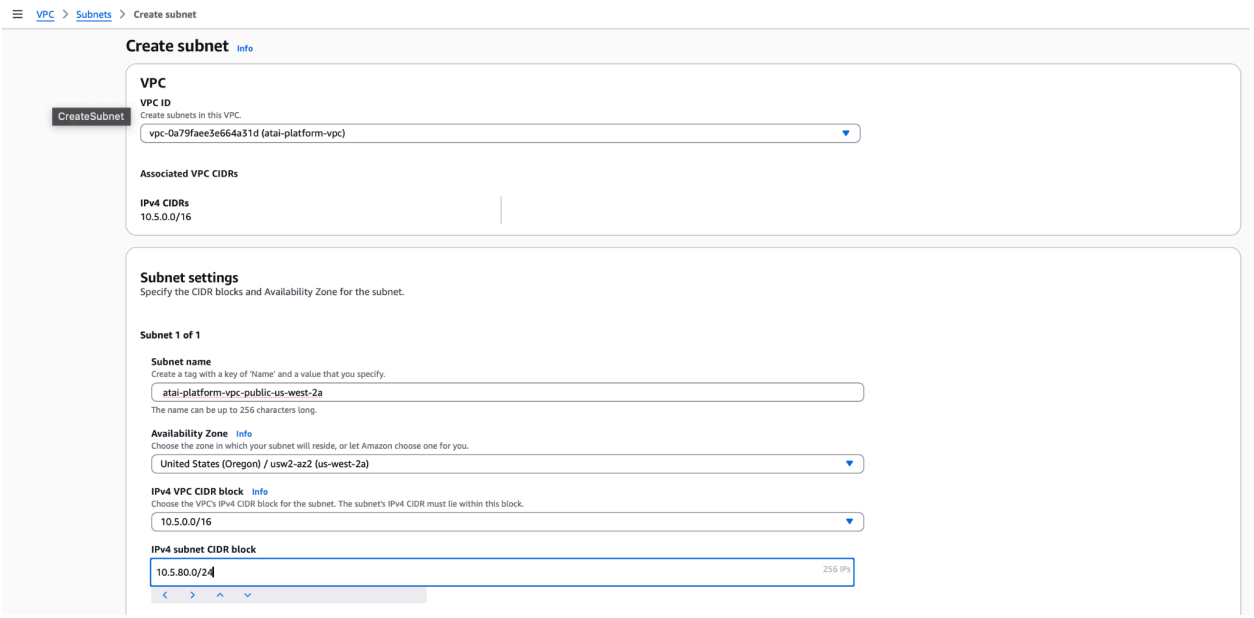
5. Select your VPC → Attach internet gateway

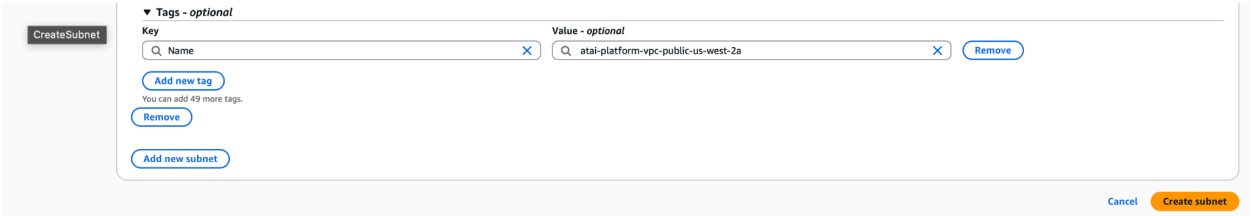


Step 3: Create Public Subnets (with correct CIDRs from the start)

1. Go to VPC Dashboard → Go to Subnets → Create subnet

2. Subnet 1:
    a. Select the VPC created in the Step 1
    b. VPC: Select your VPC
    c. Subnet name: atai-platform-vpc-public-us-west-2a
    d. Availability Zone: us-west-2a (or your AZ1)
    e. IPv4 CIDR block: 10.5.80.0/24
    f. Click Create subnet



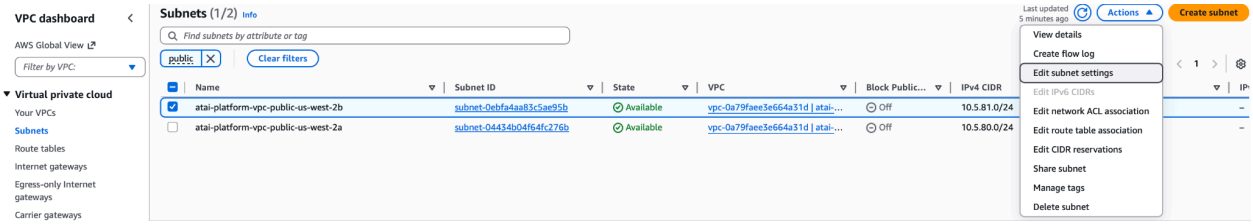


3. Subnet 2:
    a. Select the VPC created in the Step 1
    b. Subnet name: atai-platform-vpc-public-us-west-2b

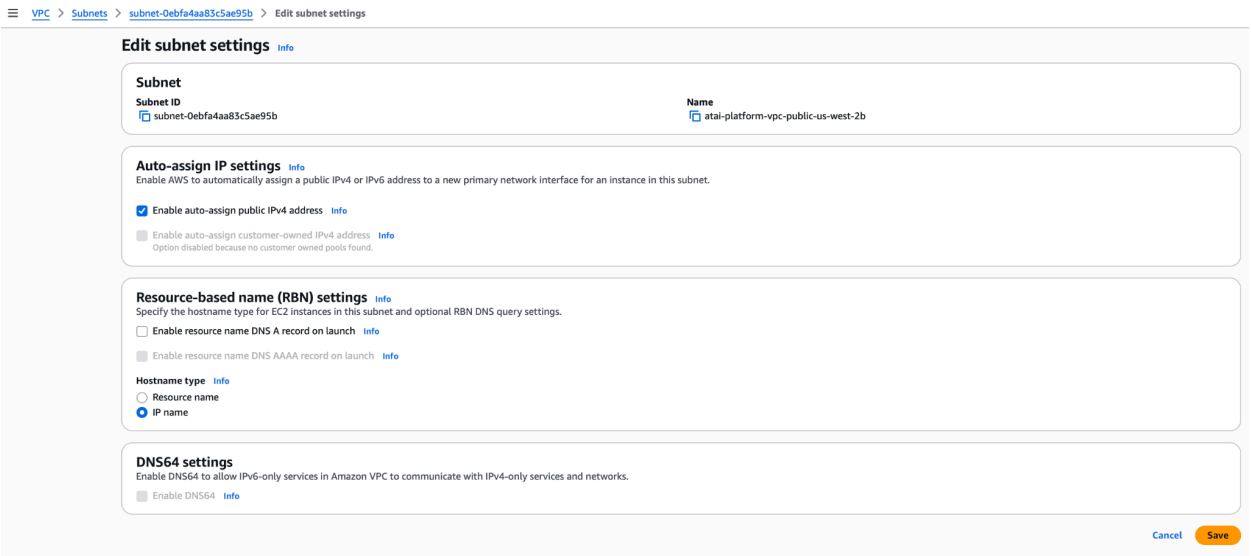
c. Availability Zone: us-west-2b (or your AZ2)
d. IPv4 CIDR block: 10.5.81.0/24
e. Click Create subnet

3.1 Enable Auto-assign Public IPv4 for Public Subnets

1. Go to VPC Dashboard → Go to Subnets → select each public subnet
2. Actions → Edit subnet settings



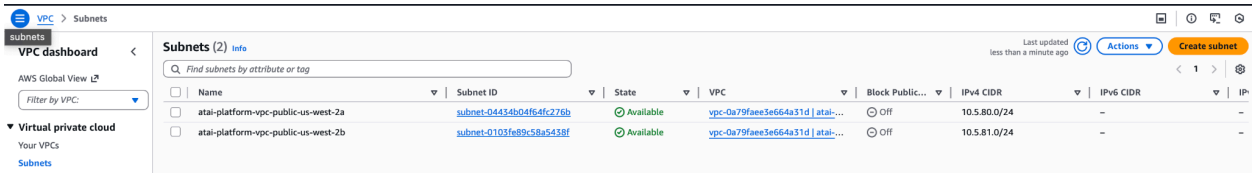
3. Check Enable auto-assign public IPv4 address
4. Save



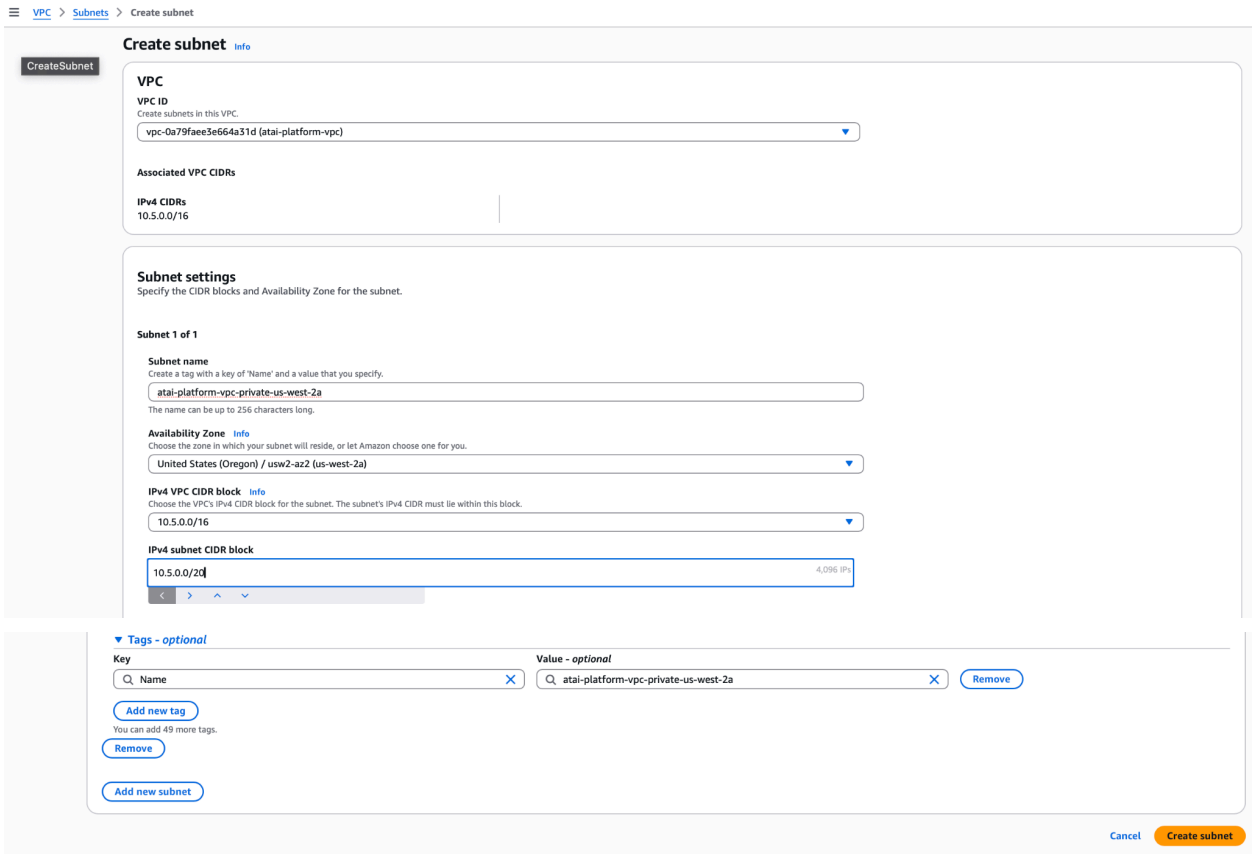
5. Repeat the process for both public subnets.

Step 4: Create Private Subnets

1. Go to VPC Dashboard → Go to Subnets → Create subnet



2. Subnet 1:
    a. Select the VPC created in the Step 1
    b. Name: atai-platform-vpc-private-us-west-2a
    c. Availability Zone: us-west-2a (or your AZ1)
    d. CIDR: 10.5.0.0/20
    e. Click Create subnet

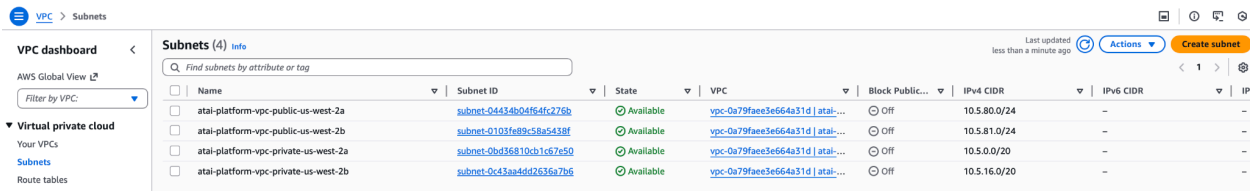


3. Subnet 2:
    a. Select the VPC created in the Step 1
    b. Name: atai-platform-vpc-private-us-west-2b
    c. Availability Zone: us-west-2b (or your AZ2)
    d. CIDR: 10.5.16.0/20

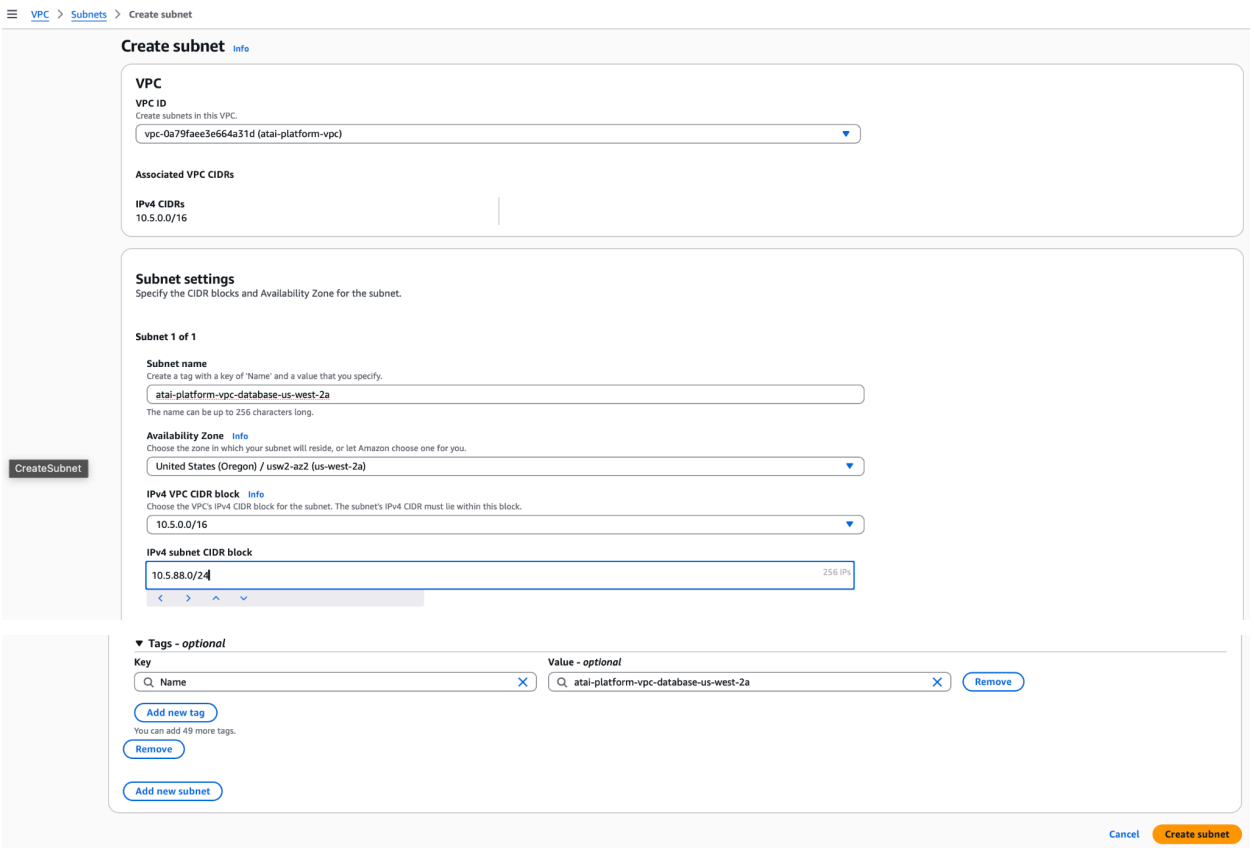
e. Click Create subnet

Step 5: Create Database Subnets

1. Go to VPC Dashboard → Go to Subnets → Create subnet



2. Subnet 1:
    a. Select the VPC created in the Step 1
    b. Name: atai-platform-vpc-database-us-west-2a
    c. Availability Zone: us-west-2a (or your AZ1)
    d. CIDR: 10.5.88.0/24
    e. Click Create subnet

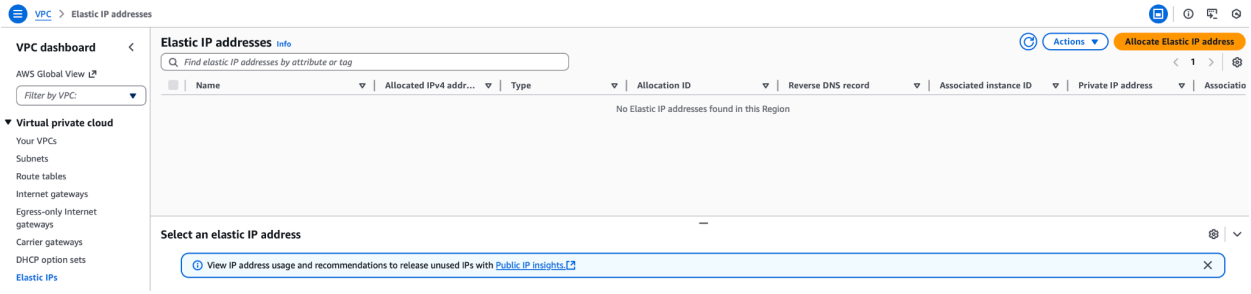


3. Subnet 2:

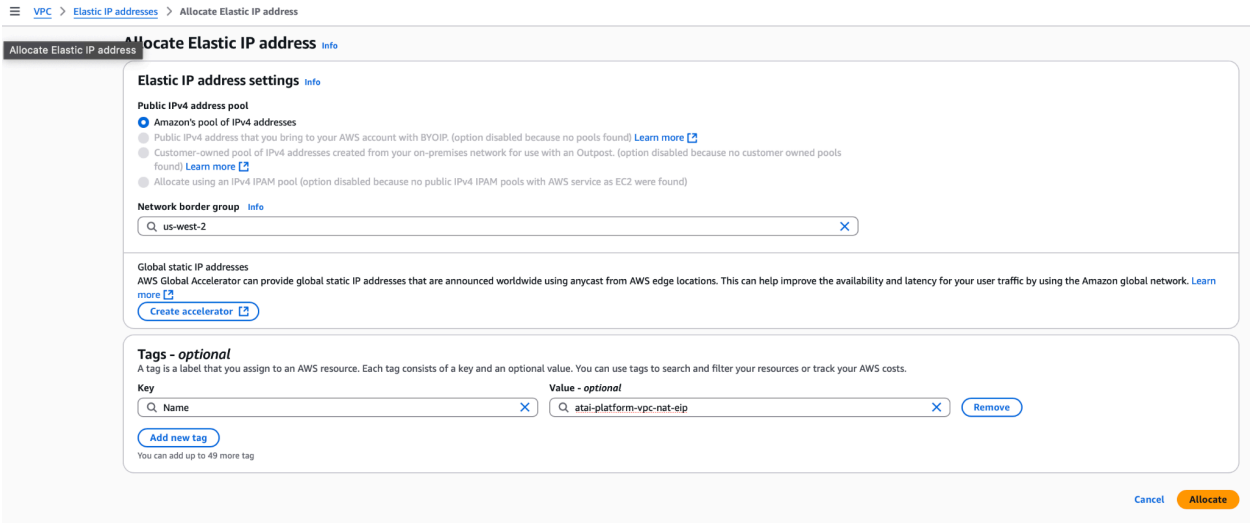
a. Select the VPC created in the Step 1
b. Name: atai-platform-vpc-database-us-west-2b
c. Availability Zone: us-west-2b (or your AZ2)
d. CIDR: 10.5.89.0/24
e. Click Create subnet

Step 6: Allocate Elastic IP for NAT Gateway

1. Go to VPC Dashboard → Go to Elastic IPs → Allocate Elastic IP address

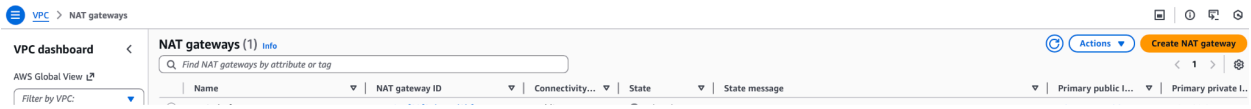


2. Network border group: Select your region
3. Public IPv4 address pool: Amazon's pool
4. Add Name tag: atai-platform-vpc-nat-eip
5. Click Allocate

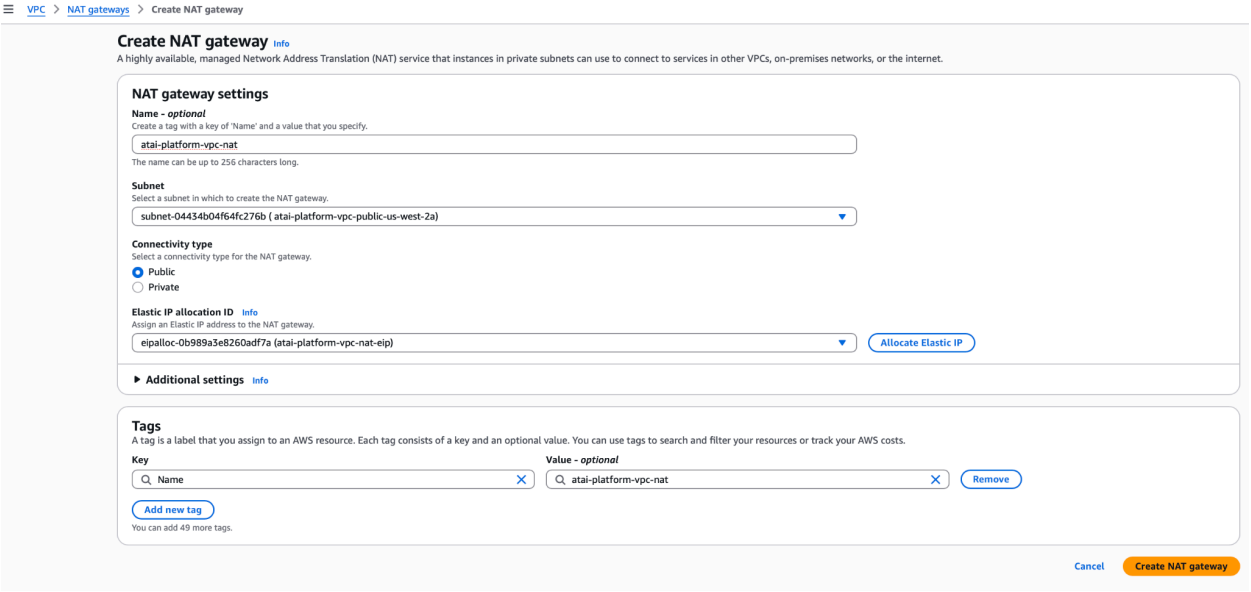


Step 7: Create NAT Gateway

1. Go to VPC Dashboard → Go to NAT Gateways → Create NAT gateway



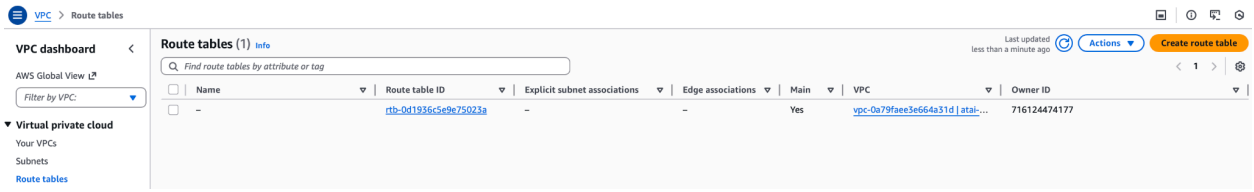
2. Name: atai-platform-vpc-nat
3. Subnet: Select first public subnet (10.5.80.0/24)
4. Elastic IP allocation ID: Select the EIP you just created in Step 6
5. Click Create NAT gateway (wait a few minutes)



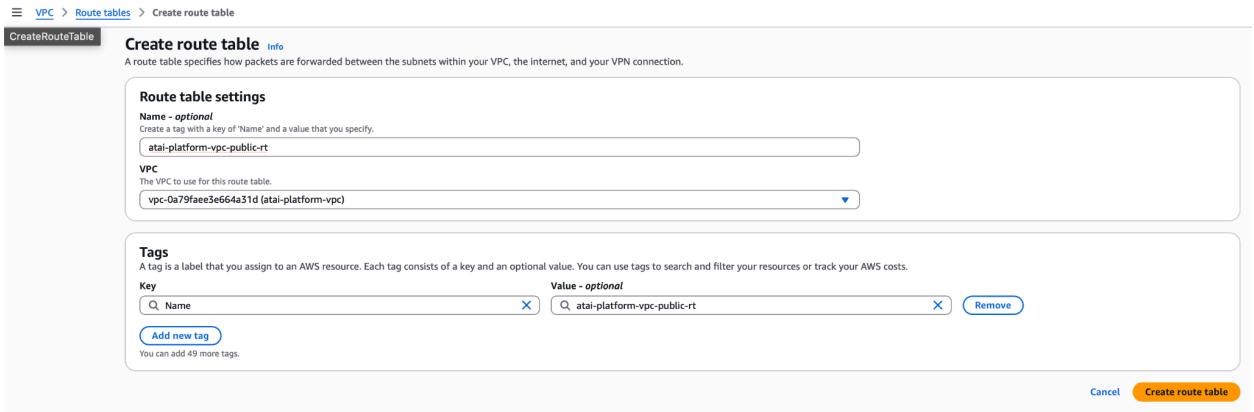
Step 8: Create and Configure Route Tables

Public Route Table:

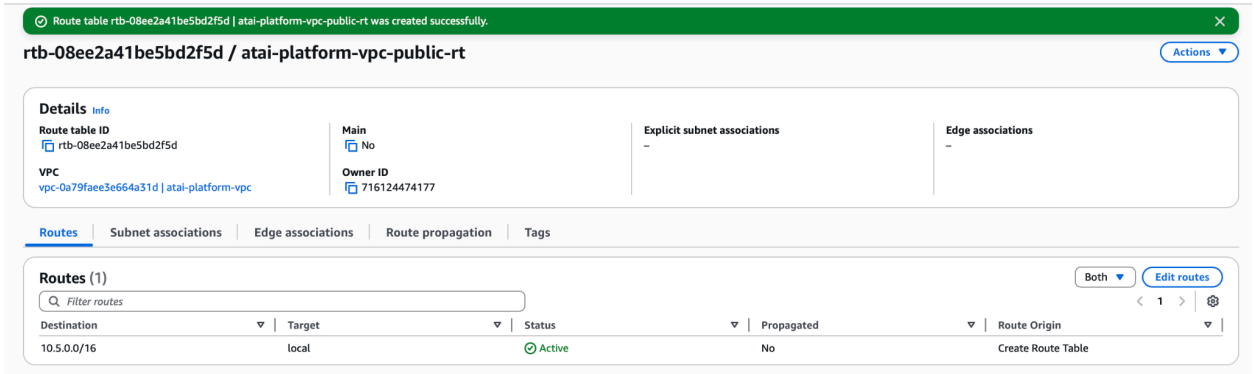
1. Go to VPC Dashboard → Route Tables → Create route table

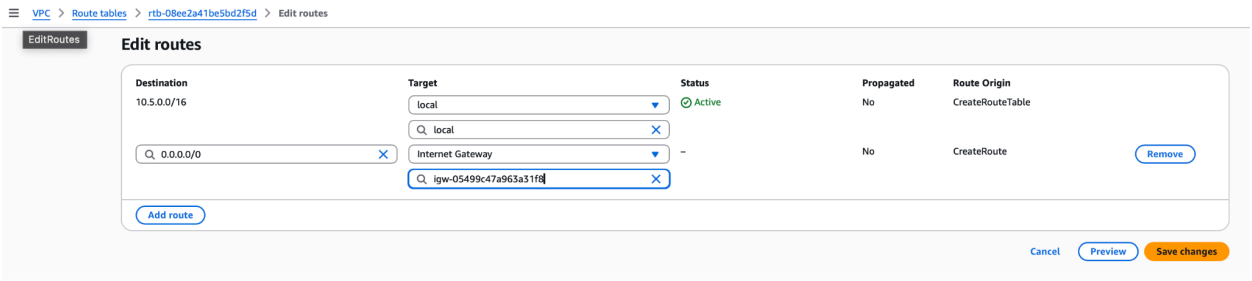


2. Name: atai-platform-vpc-public-rt
3. VPC: Select your VPC from Step 1
4. Click Create route table

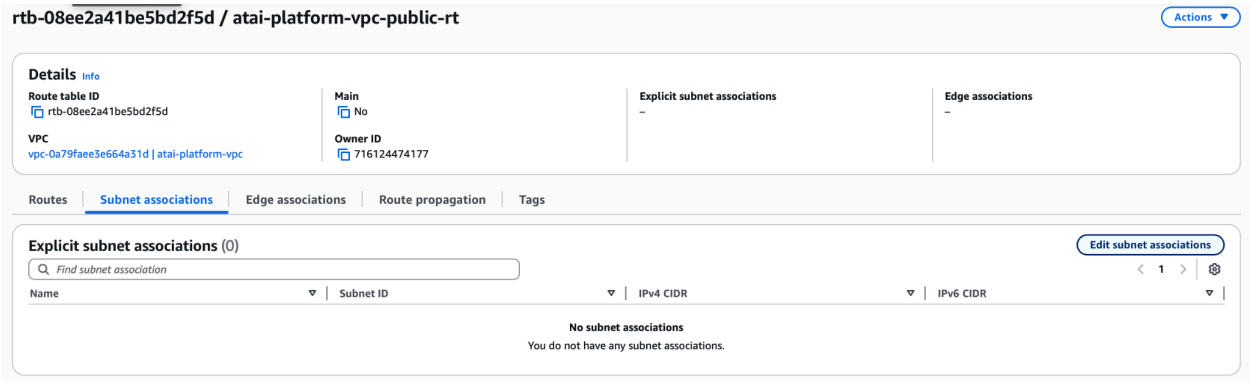


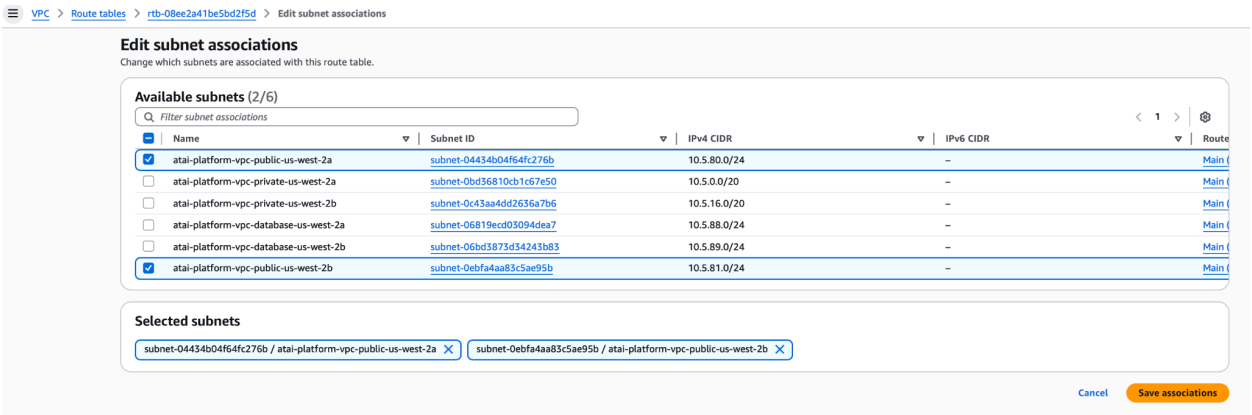
5. Routes → **Edit routes** → Add route:
    a. Destination: 0.0.0.0/0
    b. Target: Internet Gateway → select your IGW from Step 2
    c. Save changes





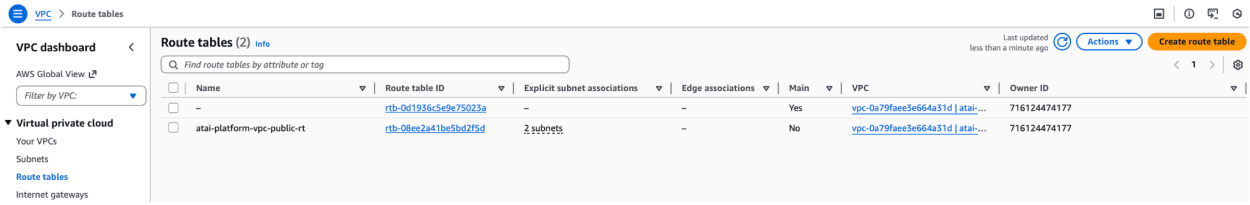
6. Subnet associations → **Edit subnet associations**:
    a. Select both public subnets → Save associations



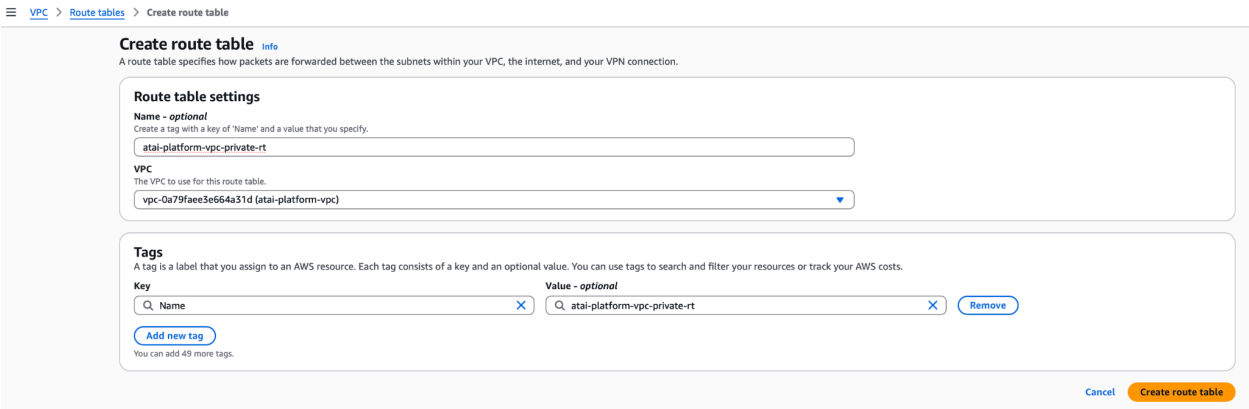


Private Route Table:

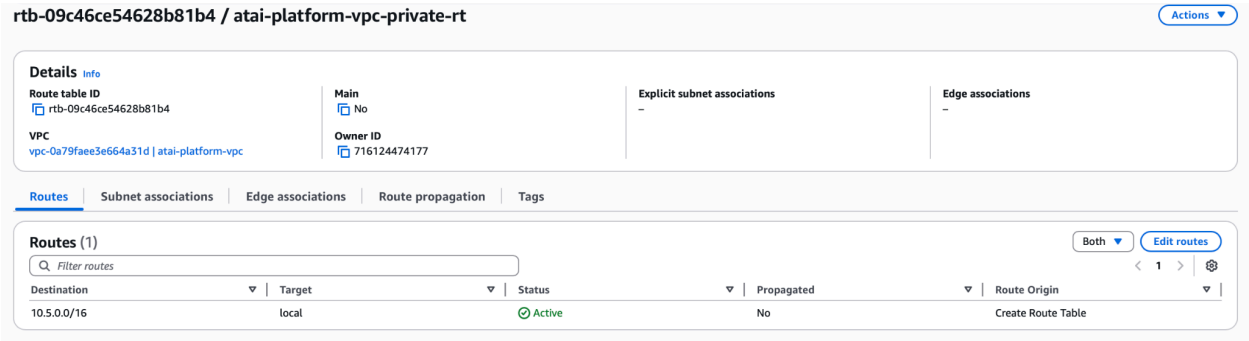
1. Go to VPC Dashboard → Route Tables → Create route table

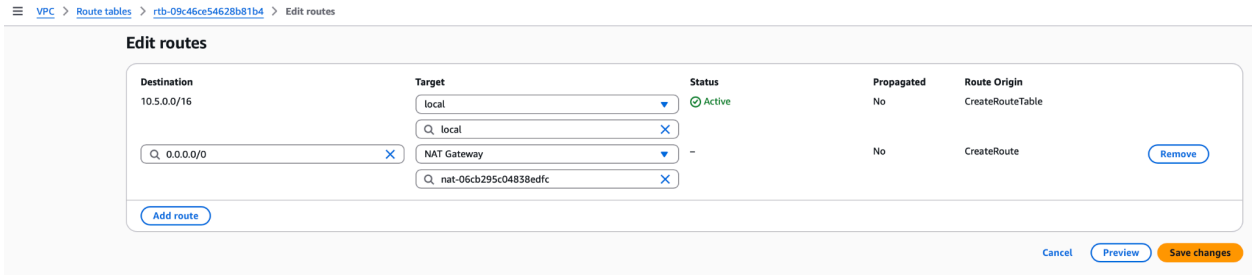


2. Name: atai-platform-vpc-private-rt
3. VPC: Select your VPC from Step 1
4. Click Create route table

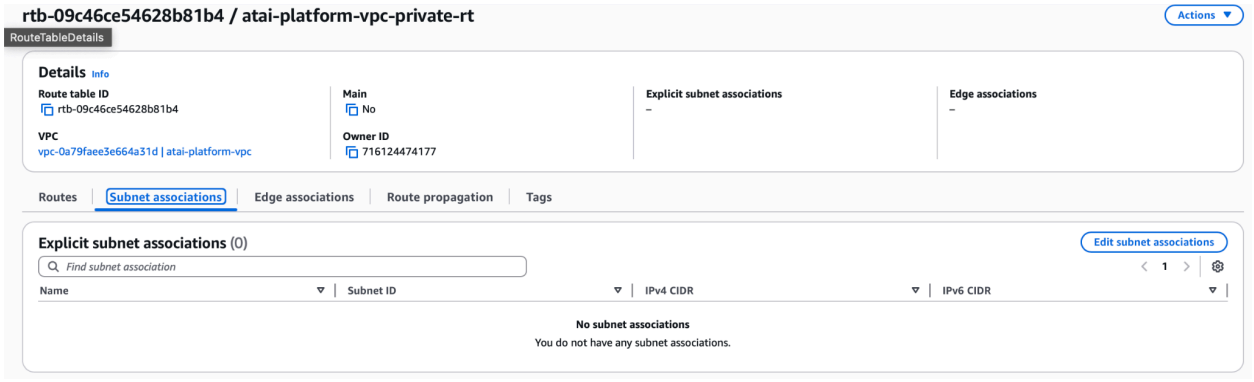


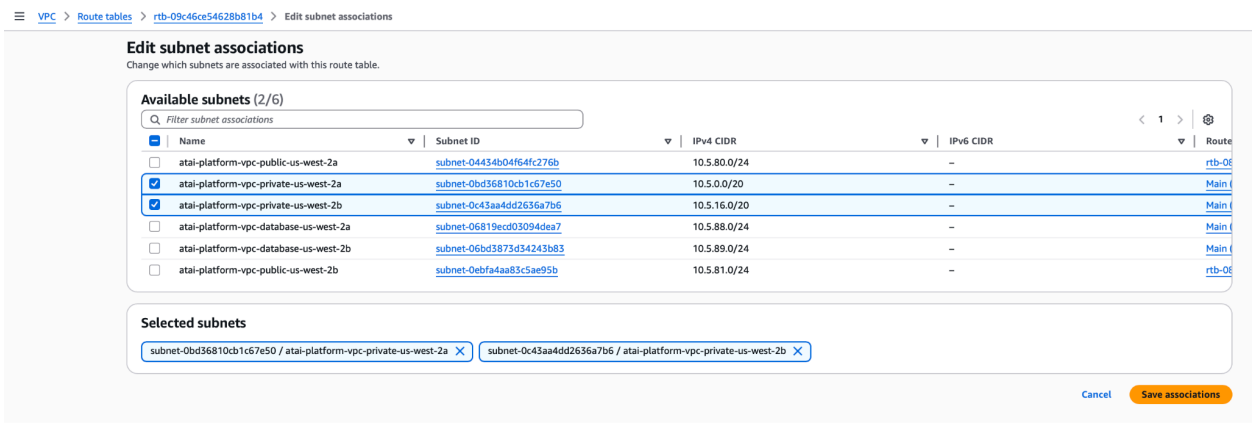
5. Routes → Edit routes → **Add route**:
    a. Destination: 0.0.0.0/0
    b. Target: NAT Gateway → select your NAT Gateway
    c. Save changes





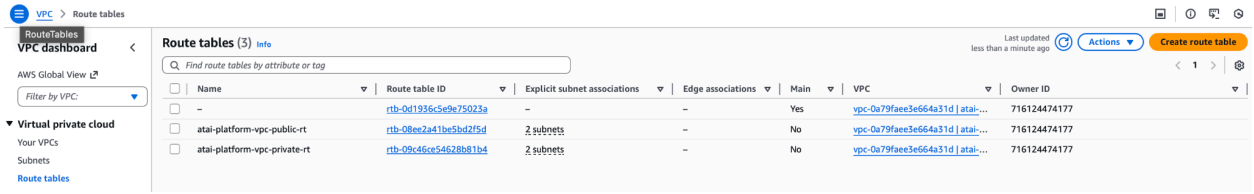
6. Subnet associations → Edit subnet associations:
    a. Select both private subnets → Save associations



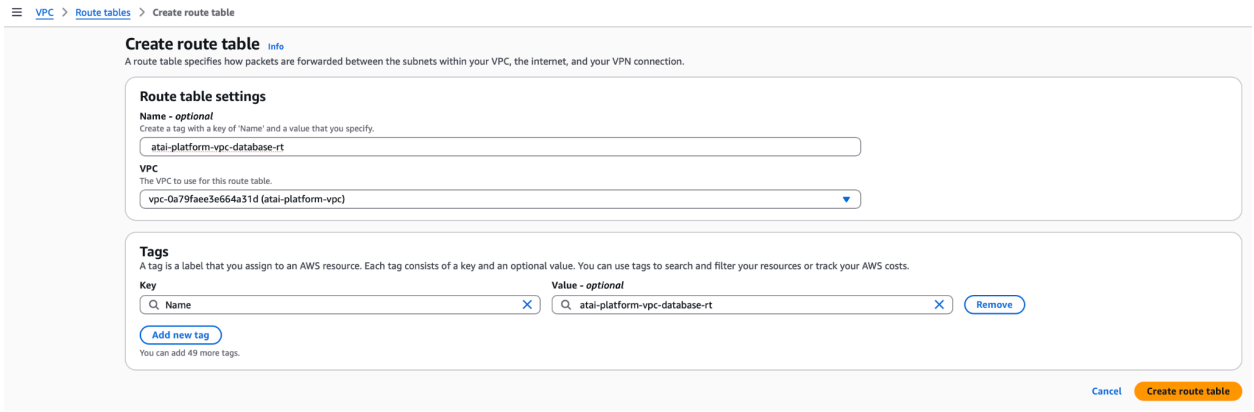


Database Route Table:

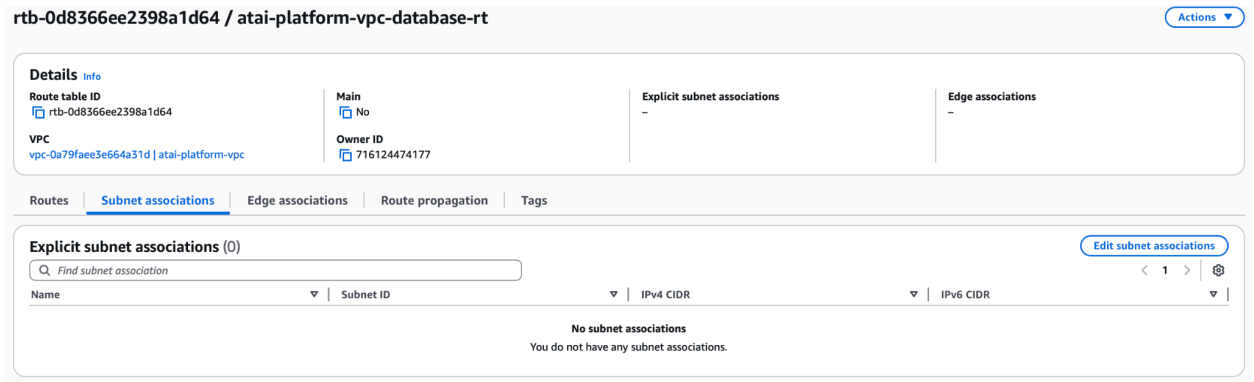
1. Go to VPC Dashboard → Route Tables → Create route table

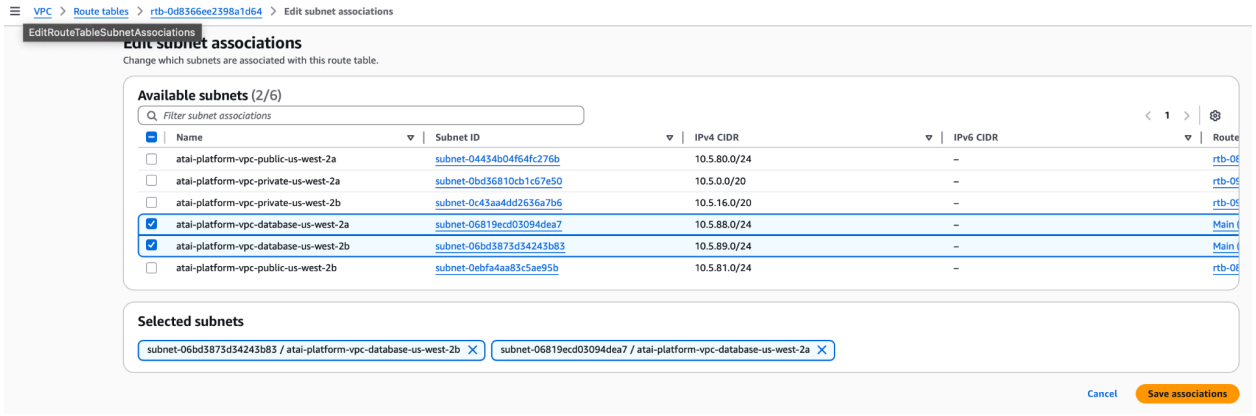


2. Name: atai-platform-vpc-database-rt
3. VPC: Select your VPC from Step 1
4. Click Create route table



5. (No extra routes needed - isolated)
6. Subnet associations → Edit subnet associations:
    a. Select both database subnets → Save associations





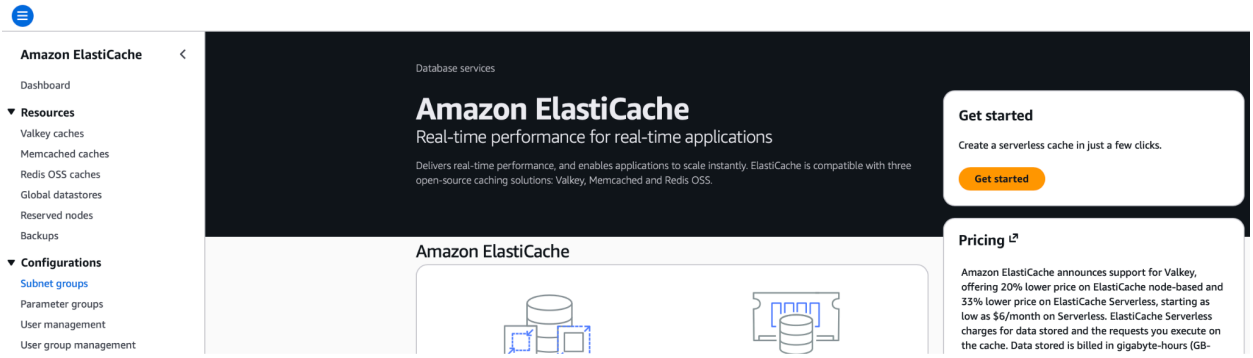
Valkey clusters configuration

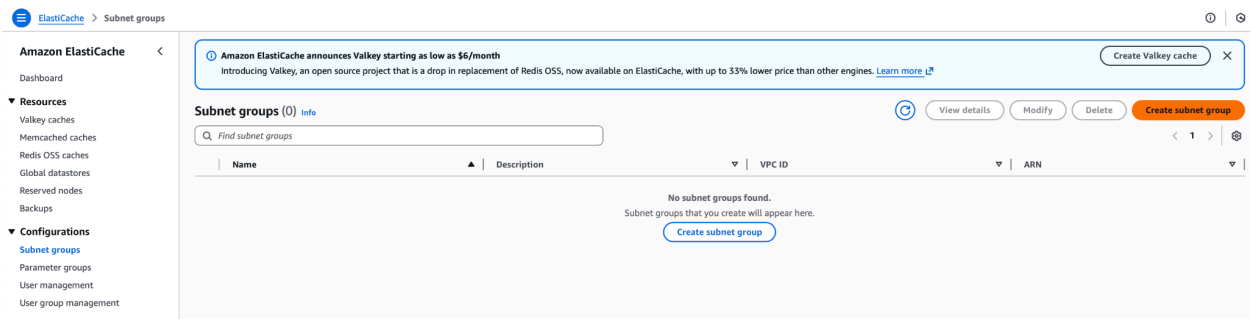
Prerequisites

1. VPC with database subnets
2. (Optional) Security group allowing access from EKS pods
3. Subnet group for database subnets

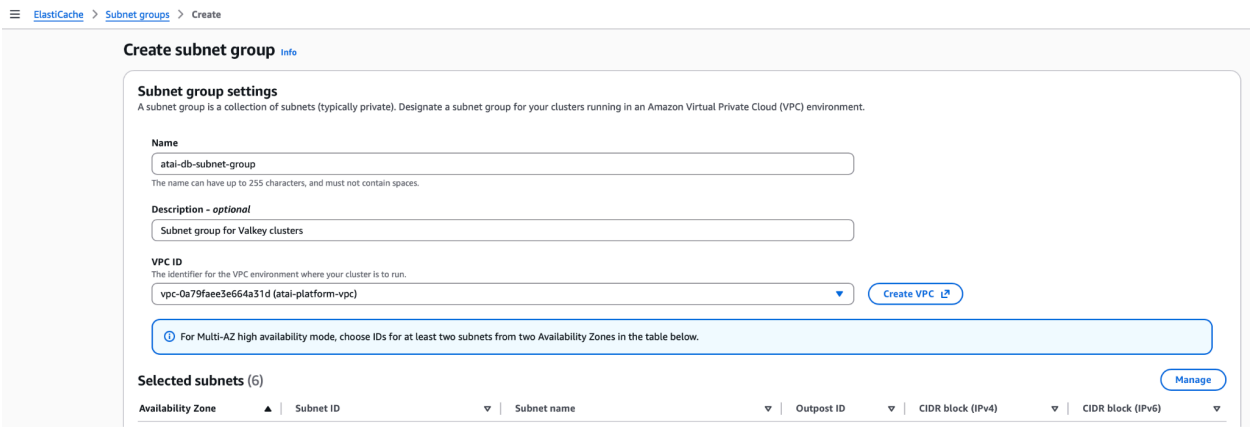
Step 1: Create ElastiCache Subnet Group (if not existing)

1. Go to ElastiCache → Subnet groups → Create subnet group

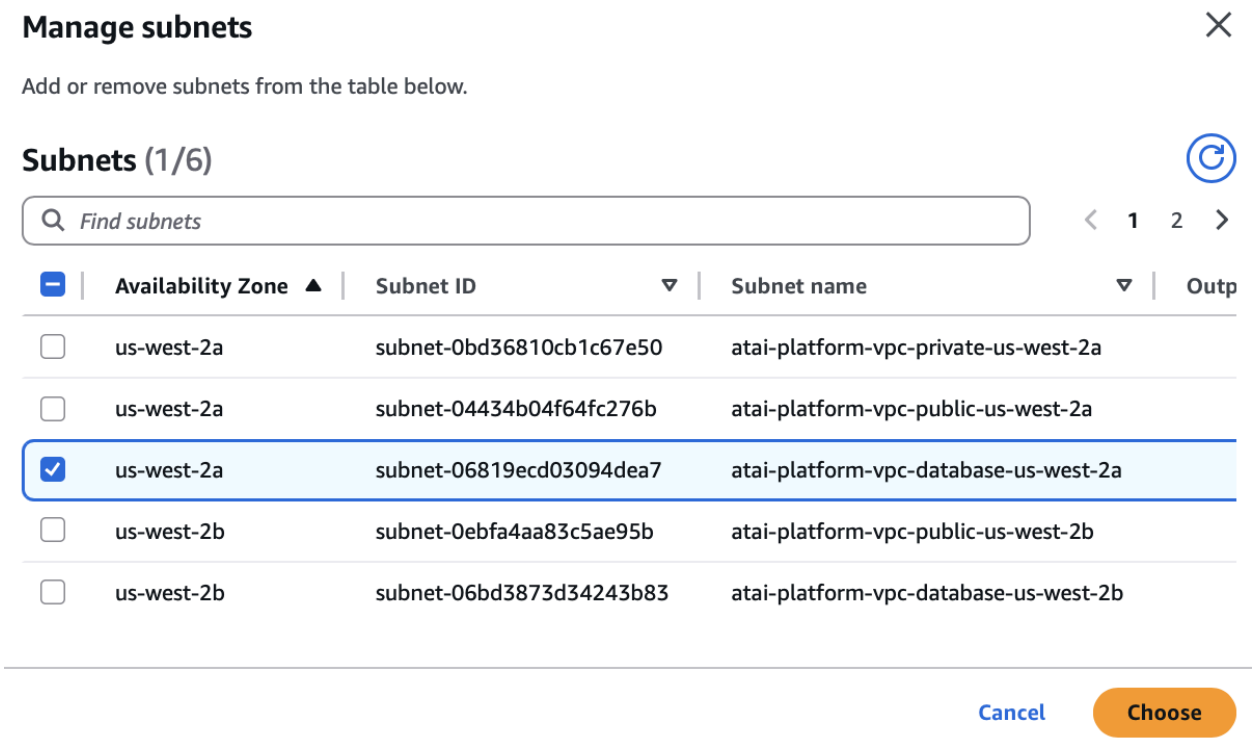




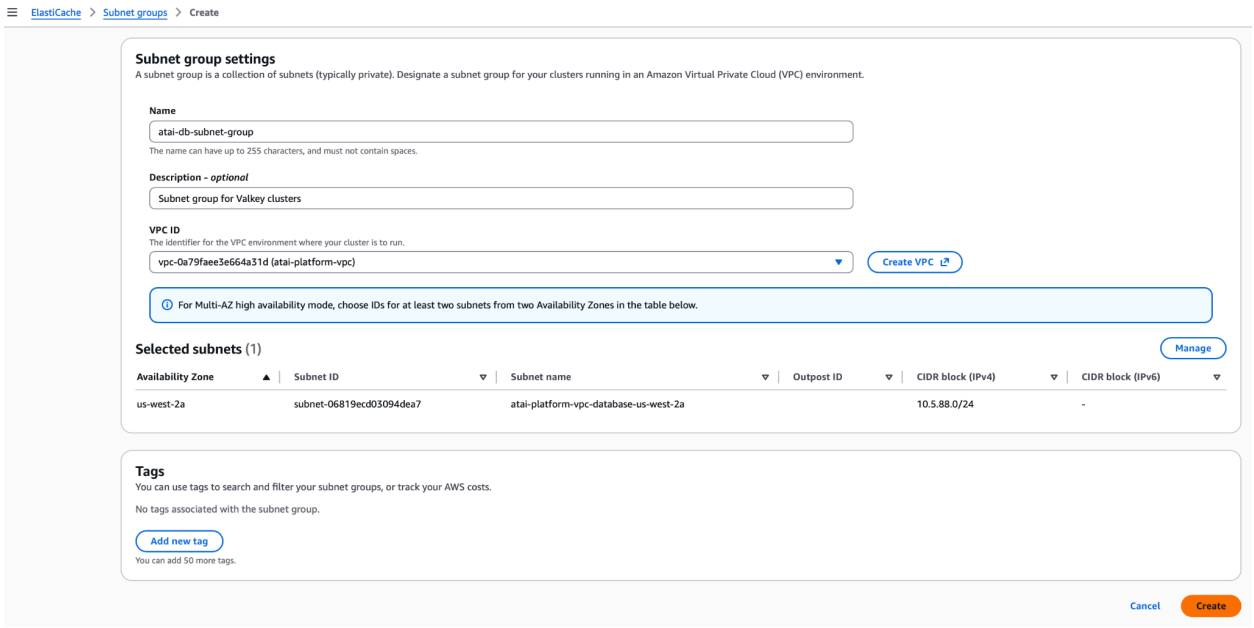
2. Name: atai-db-subnet-group (or your name)
3. Description: Subnet group for Valkey clusters (single AZ for low latency)
4. VPC: Select your VPC



5. In the section **Selected subnets** click on **Manage**
    a. Availability Zones: Select only the first AZ (e.g., us-west-2a) and click on **Choose**
        i. Important: Use only one AZ to reduce latency
        ii. Note: EKS Managed node groups will use a private subnet in the same AZ (different subnet CIDR)



6. Subnets: Select only the first database subnet (e.g., 10.5.88.0/24 in us-west-2a)
    a. Do not select the second database subnet
7. Click Create

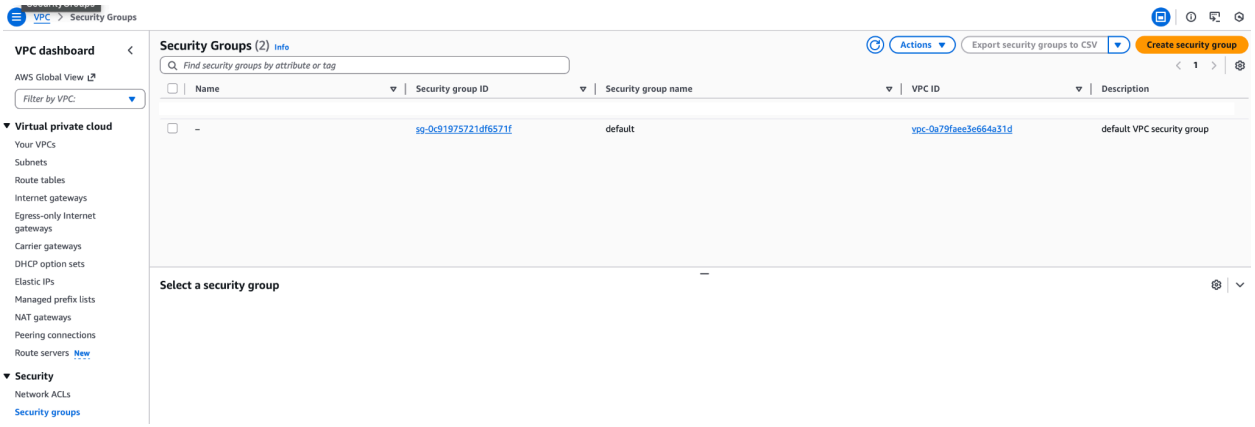


Important Notes:

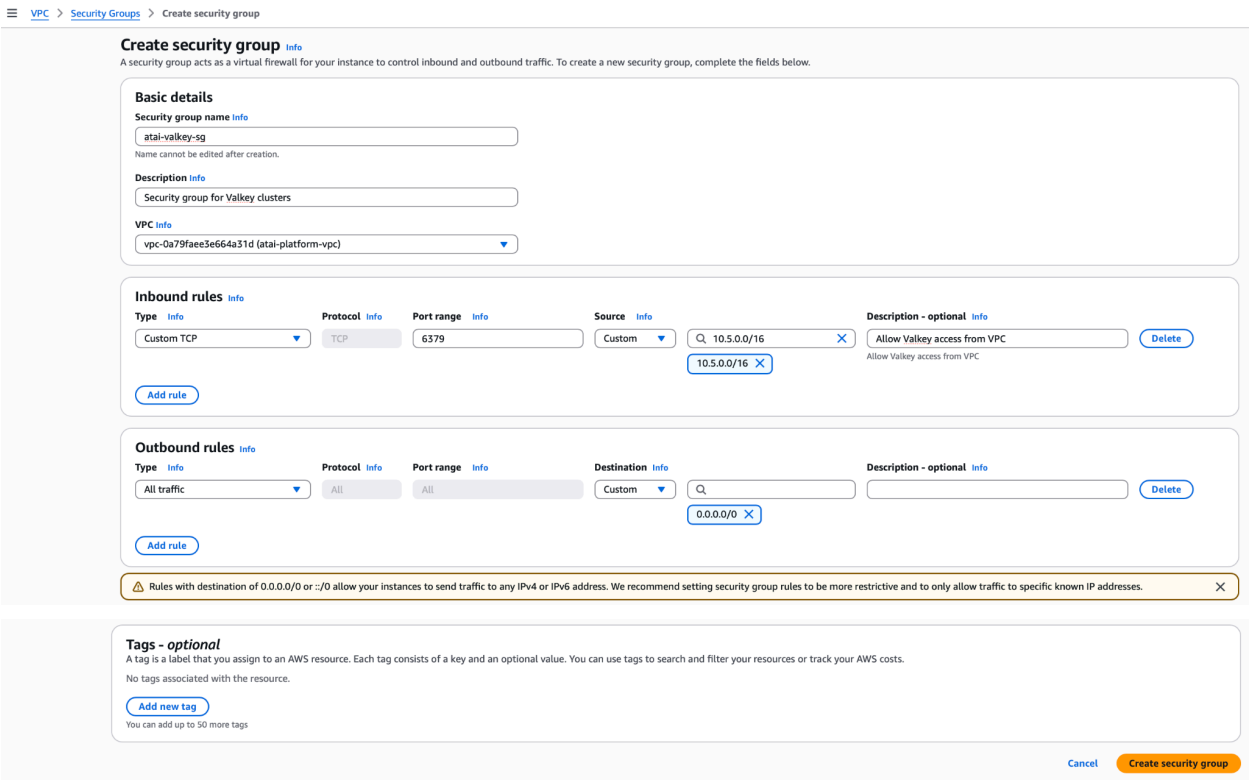
1. Single AZ deployment reduces cross-AZ network latency
2. All Valkey clusters will be created in this single AZ
3. EKS managed node groups should use a private subnet in the same AZ (e.g., 10.5.0.0/20 in us-west-2a) for optimal latency
4. Example: If you use us-west-2a for database subnet 10.5.88.0/24, use us-west-2a for private subnet 10.5.0.0/20 for node groups

Step 2: Create Security Group (if not existing)

1. Go to VPC → Security Groups → Create security group



2. Name: atai-valkey-sg (or your name)
3. Description: Security group for Valkey clusters
4. VPC: Select your VPC from section VPC configuration Step 1
5. Inbound rules: Add rule:
    a. Type: Custom TCP
    b. Port: 6379
    c. Source: Custom → Enter your VPC CIDR (e.g., 10.5.0.0/16)
    d. Description: Allow Valkey access from VPC

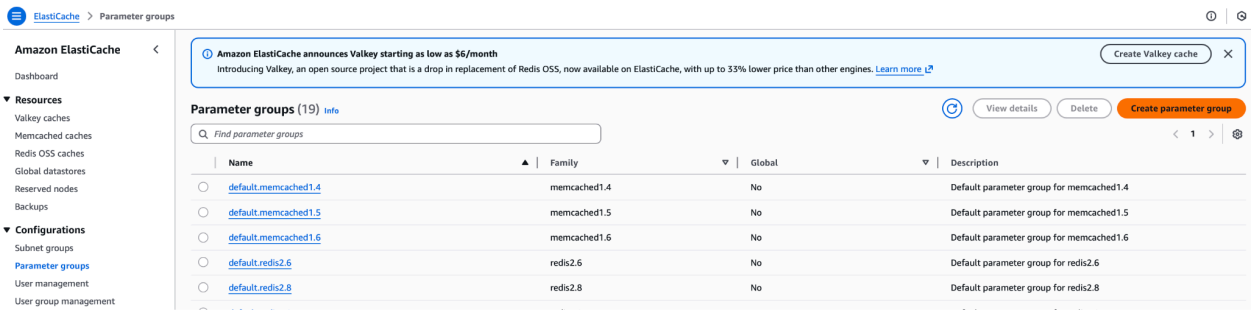


Note: For initial setup, opening to the VPC CIDR simplifies connectivity. Later, restrict to specific security groups (e.g., EKS node group security group) for tighter security.

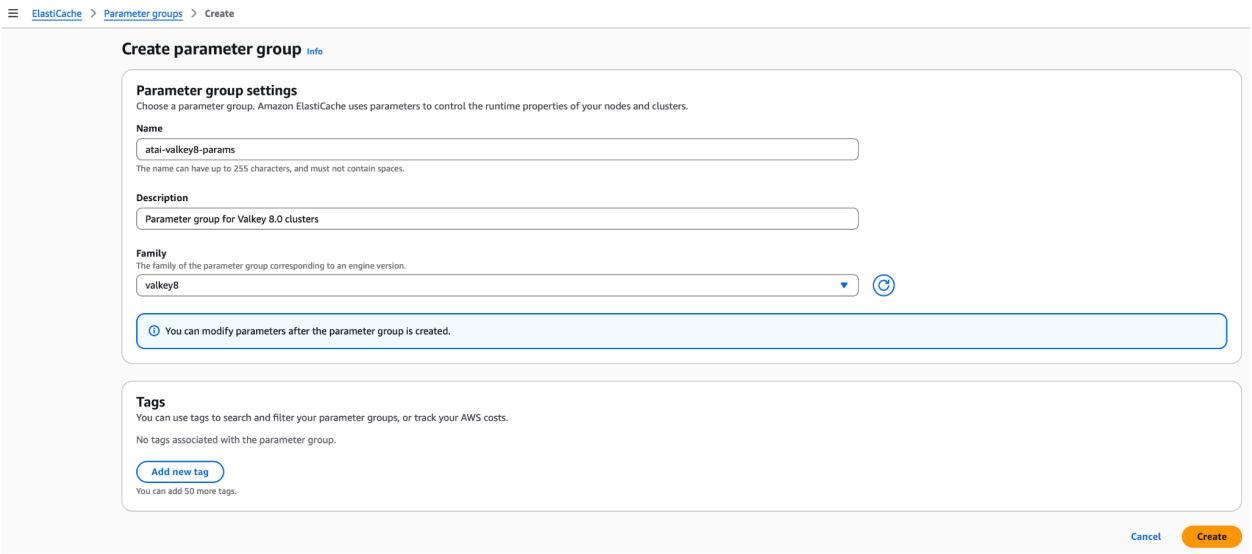
6. Click Create security group

Step 3: Create Parameter Group for Valkey 8.0

7. Go to ElastiCache → Parameter groups → Create parameter group



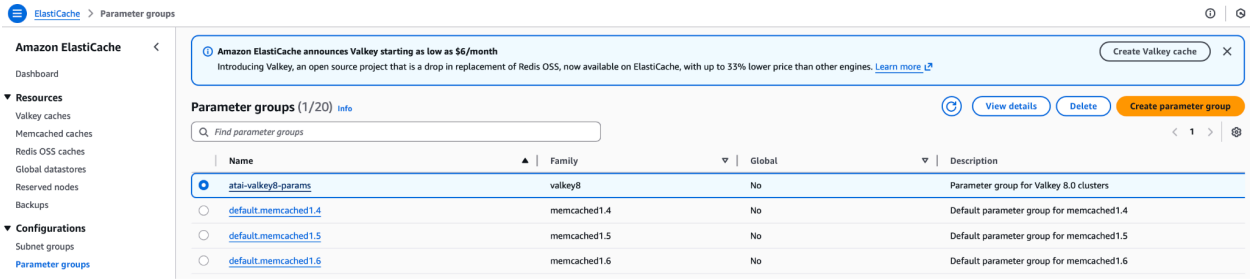
8. Group name: atai-valkey8-params
9. Description: Parameter group for Valkey 8.0 clusters
10. Parameter group family: valkey8



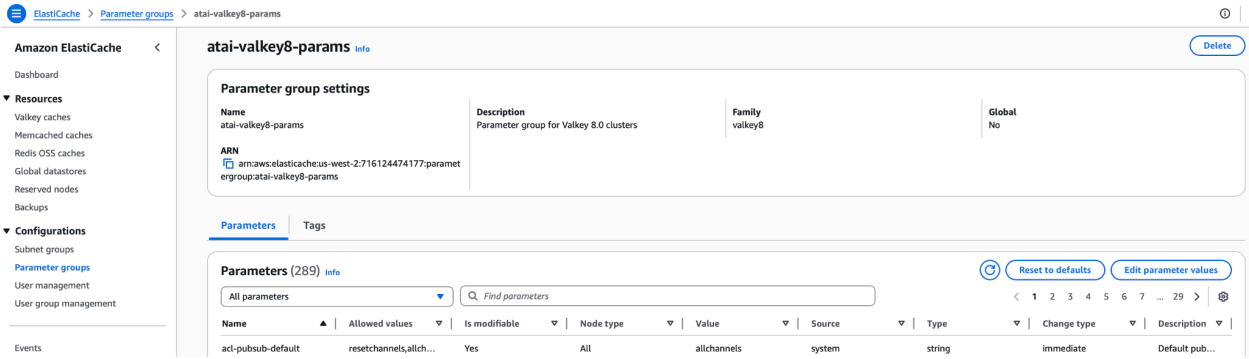
11. Click Create

Step 3.1 After creation, edit the parameter group:

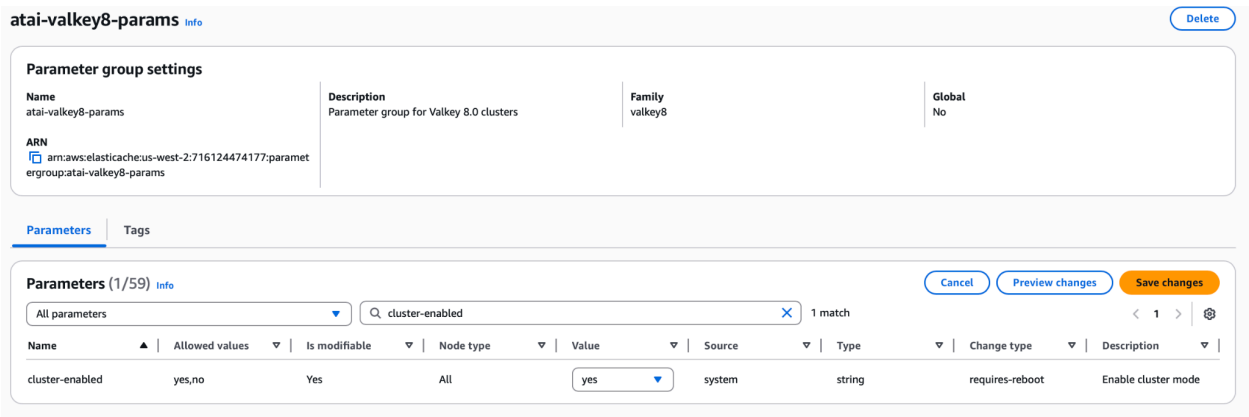
1. Select the parameter group → Edit



2. Click on **Edit parameter values**



3. In search bar type **cluster-enabled**
4. With the dropdown menu set the value to **yes**



5. Click Save changes

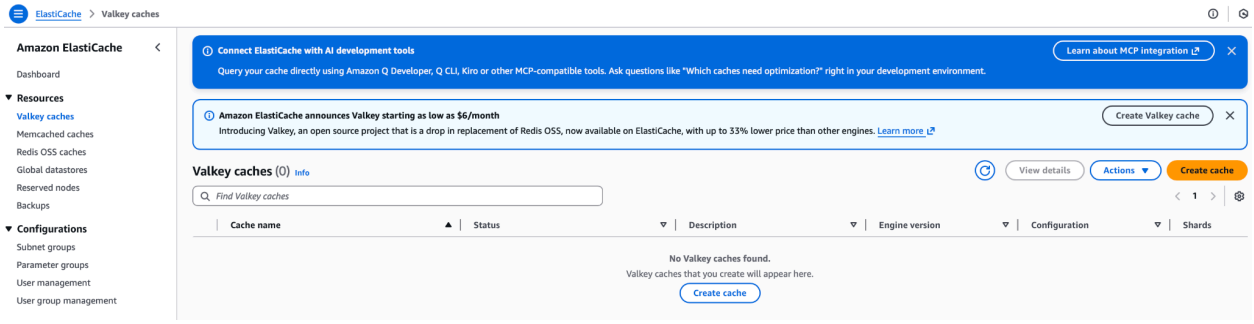
Important: **The cluster-enabled = yes parameter is required for cluster mode to work**. Without it, the clusters won't function properly in cluster mode, even if you enable cluster mode in the cluster settings.

Step 4: Create Valkey Clusters

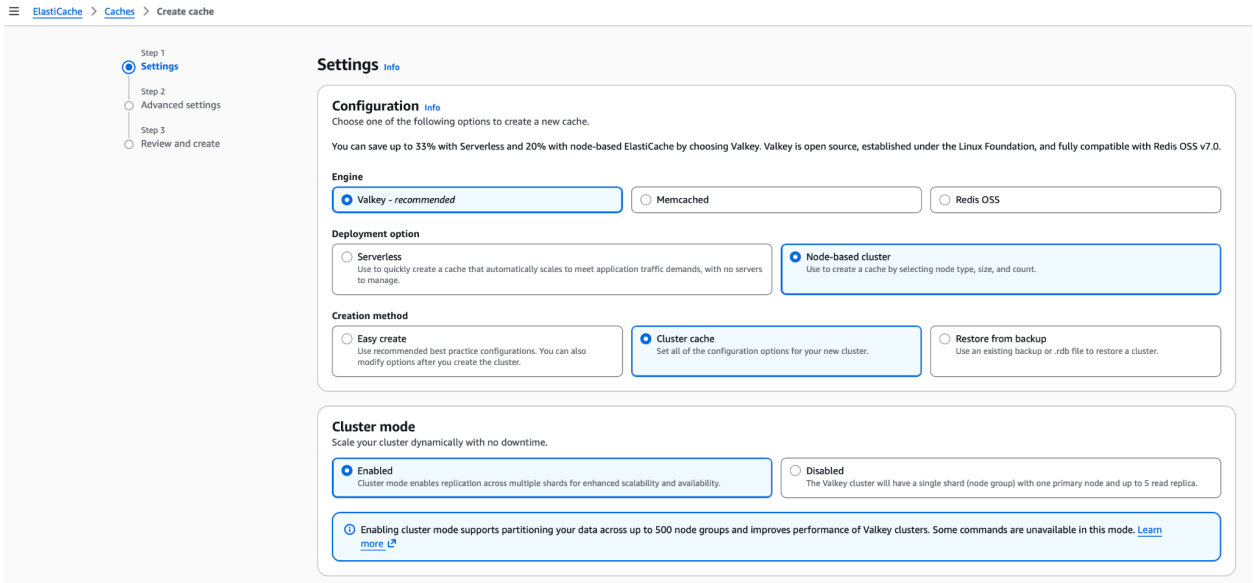
For each cluster, follow these steps.

Cluster 1: registry

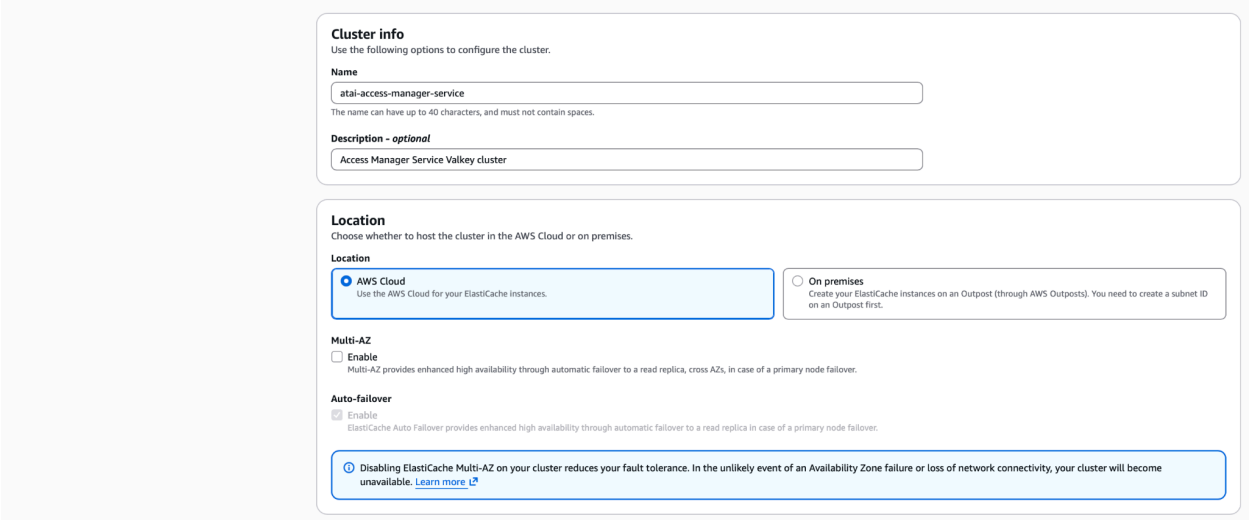
1. ElastiCache → Valkey caches → **Create cache**



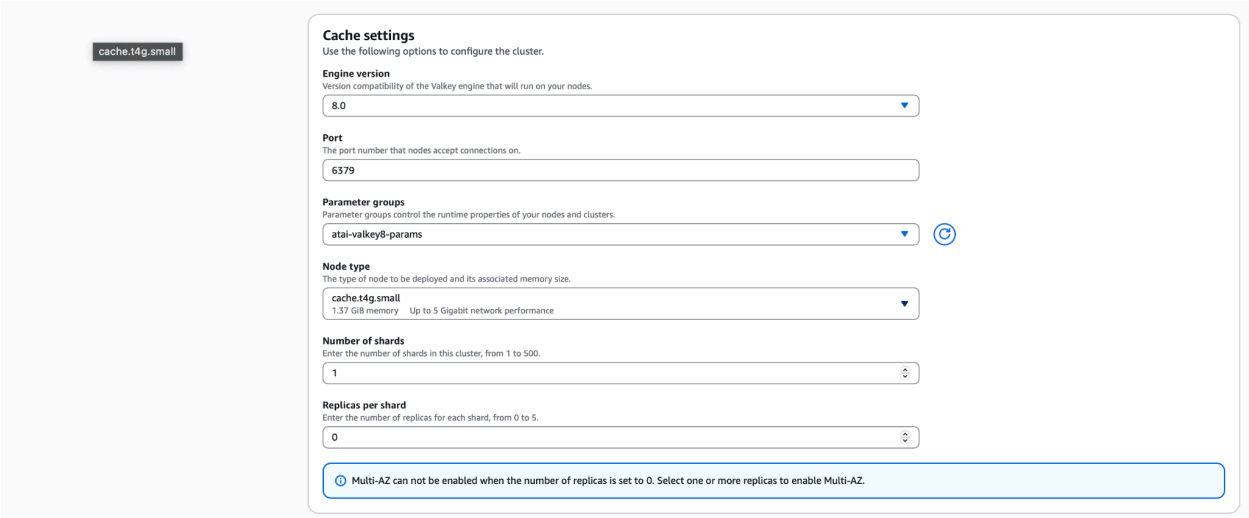
2. Engine: Valkey
3. Deployment option: Node-based cluster
4. Creation method: Cluster cache
5. Cluster mode: Enable



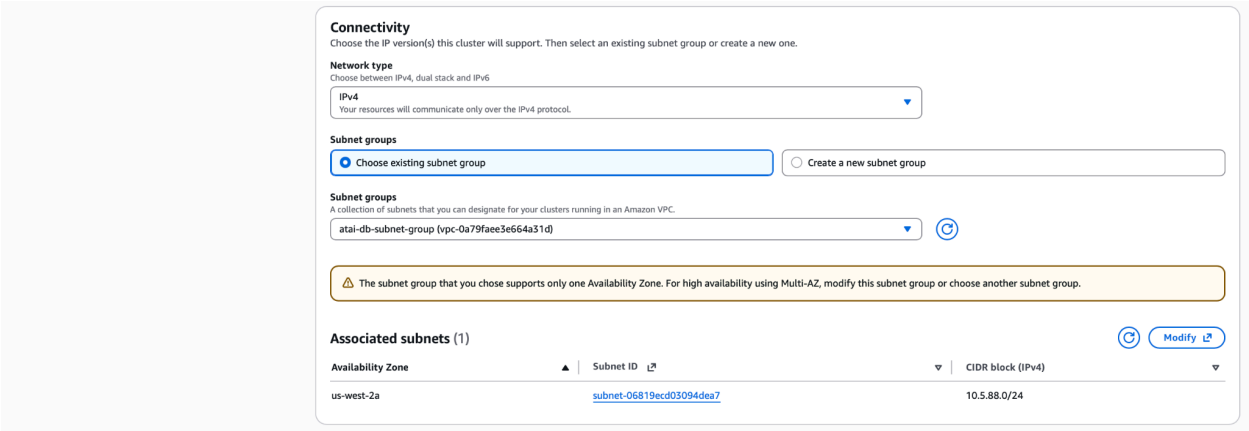
6. Cluster info
    a. Name: atai-registry-service
    b. Description: Registry Service Valkey cluster
7. Location
    a. Location: AWS Cloud
    b. Muti-AZ: Disable (to reduce latency)



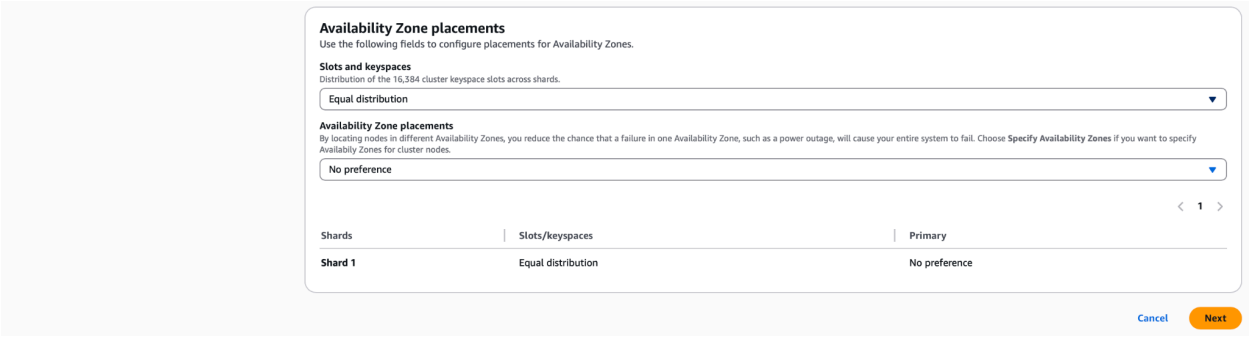
8. Cache setting
    a. Engine version: 8.0
    b. Port: 6379
    c. Parameter group: Select the parameter group created in the Step 3 of this section
    d. Node type: cache.t4g.small
    e. Number of shards: 1
    f. Replicas per shard: 0



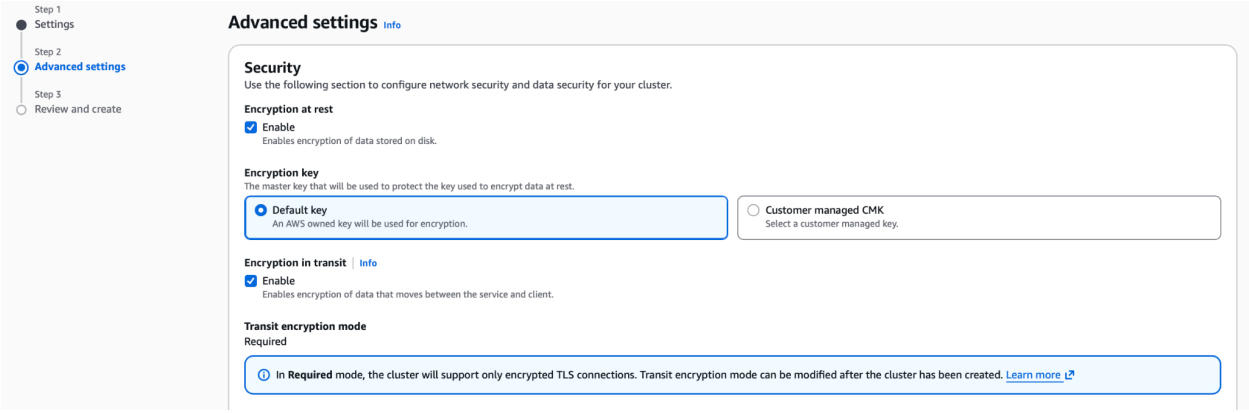
9. Connectivity:
    a. **Subnet groups**: Select your database subnet group (single AZ) from Step 1



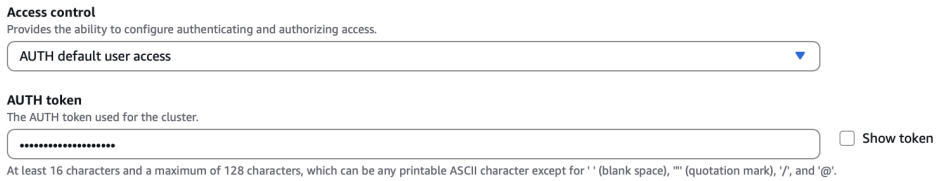
    b. Use the default Availability Zone placements -> Next



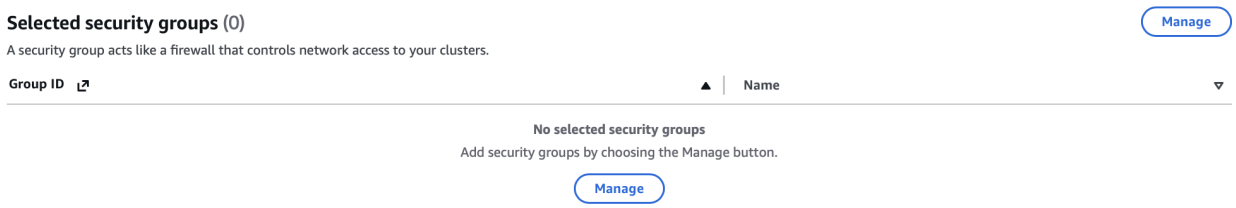
10. Enable encryption at rest and Encryption in transit



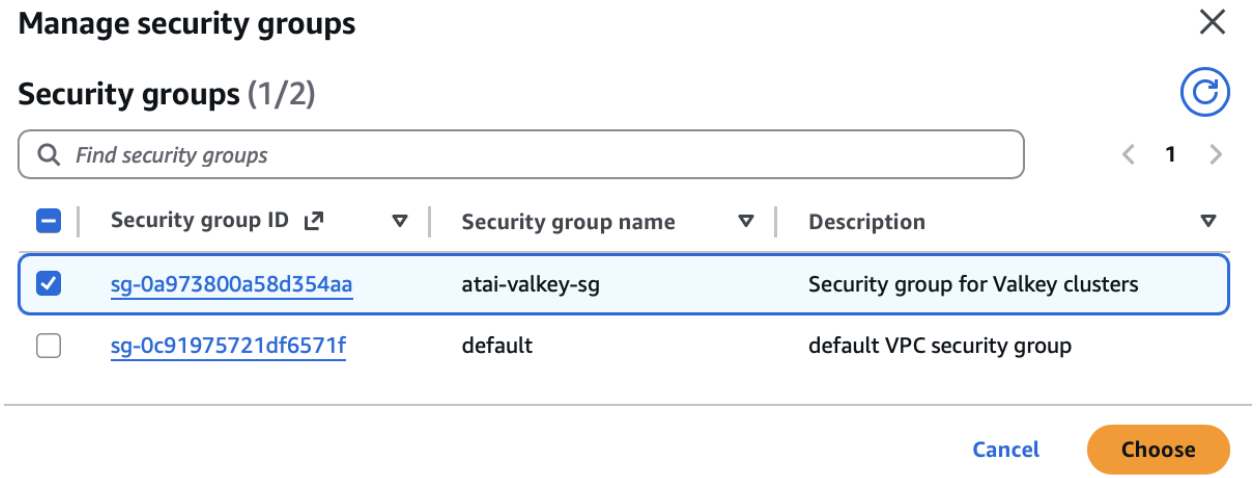
11. For access control choose **AUTH default user access**
    a. Important: Save this auth token securely. It is required later to communicate with your Valkey cluster.
    b. Security recommendation: Create a unique auth token per Valkey cluster (do not reuse the same token across clusters).
    c. Store each token in a secure location such as AWS Secrets Manager.



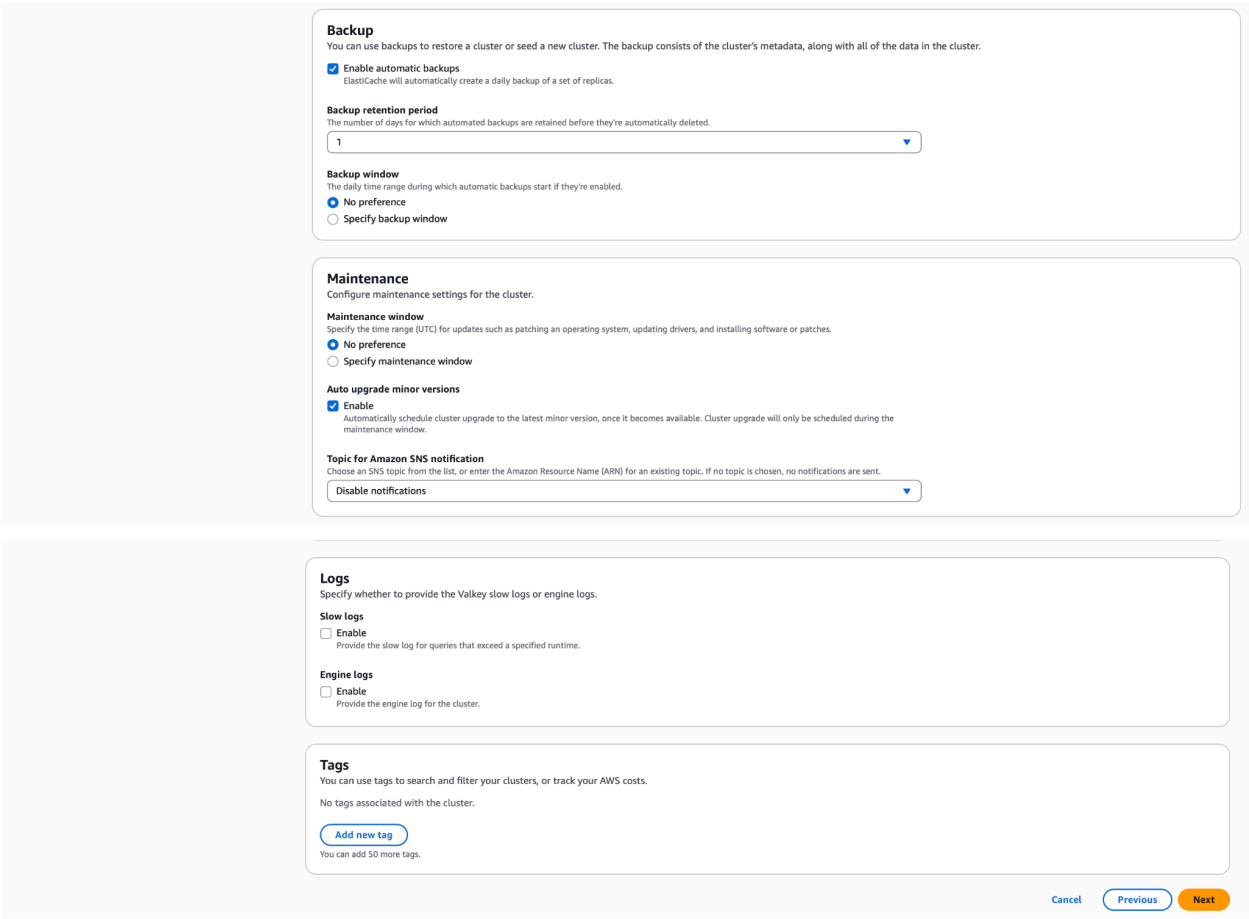
12. Security groups
    a. Click on **Manage**



    b. Select the security security group created in Step 2
    c. Click on **Choose**

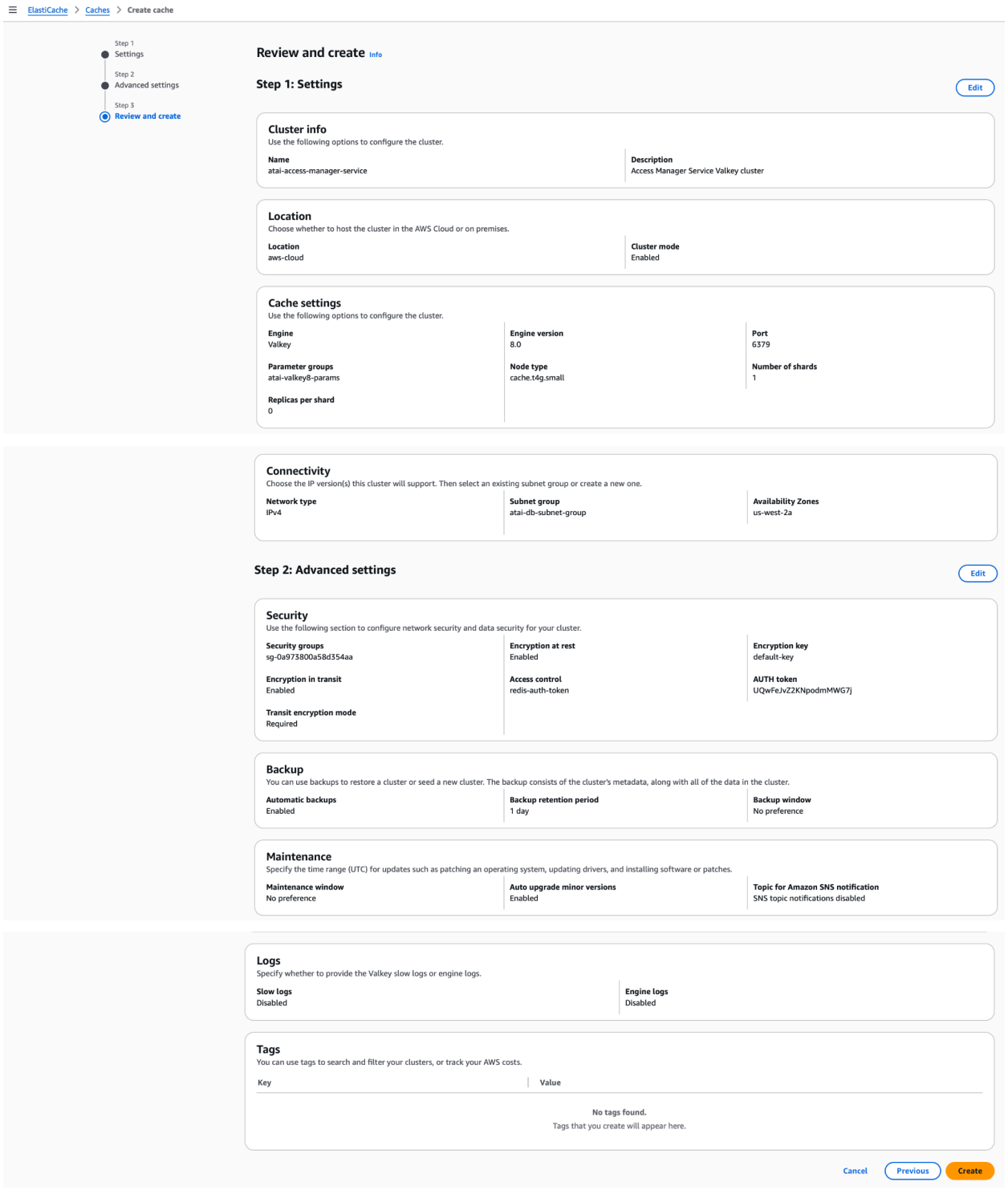


13. Use the default configuration for **Backups**, **Maintenance**, **Logs** and **Tags.**



14. Click Next

15. Review your configuration



16. Click on **Create**

⚠️ Store your Valkey host, port, and authtoken in a secure location. You will need them later in the *atai-platform* prerequisites, Step 3.2: Kubernetes Secrets Generation. These values will be referenced as:

```
None
[atai-platform-registry-service-backend]
REGISTRY_SERVICE_IP_ADDRESS=clustercfg.atai-registry-service.<HASH>.<AWS_REGION
_CODE.cache.amazonaws.com
REGISTRY_SERVICE_PASSWORD=<REGISTRY_SERVICE_PASSWORD>
REGISTRY_SERVICE_PORT=6379
```

Note: The **atai-registry-service** secrets will be required for following services:

- gpq-node-newton-model-c23
- gpq-node-newton-model-omega
- api-service-backend
- api-service-health-node
- lens-node-worker-node
- lens-node-service-backend
- lens-service-db-backend

These values will be referenced as:

```
None
[atai-platform-gpq-node-newton-model-c23]
REGISTRY_SERVICE_IP_ADDRESS=clustercfg.atai-registry-service.<HASH>.<AWS_REGION
_CODE>.cache.amazonaws.com
REGISTRY_SERVICE_PASSWORD=<REGISTRY_SERVICE_PASSWORD>
REGISTRY_SERVICE_PORT=6379

[atai-platform-gpq-node-newton-model-omega]
REGISTRY_SERVICE_IP_ADDRESS=clustercfg.atai-registry-service.<HASH>.<AWS_REGION
_CODE>.cache.amazonaws.com
REGISTRY_SERVICE_PASSWORD=<REGISTRY_SERVICE_PASSWORD>
REGISTRY_SERVICE_PORT=6379

[atai-platform-api-service-backend]
REGISTRY_SERVICE_IP_ADDRESS=clustercfg.atai-registry-service.<HASH>.<AWS_REGION
_CODE>.cache.amazonaws.com
REGISTRY_SERVICE_PASSWORD=<REGISTRY_SERVICE_PASSWORD>
REGISTRY_SERVICE_PORT=6379

[atai-platform-api-service-health-node]
REGISTRY_SERVICE_IP_ADDRESS=clustercfg.atai-registry-service.<HASH>.<AWS_REGION
_CODE>.cache.amazonaws.com
REGISTRY_SERVICE_PASSWORD=<REGISTRY_SERVICE_PASSWORD>
REGISTRY_SERVICE_PORT=6379
```

```
[atai-platform-lens-node-worker-node]
REGISTRY_SERVICE_IP_ADDRESS=clustercfg.atai-registry-service.<HASH>.<AWS_REGION
_CODE>.cache.amazonaws.com
REGISTRY_SERVICE_PASSWORD=<REGISTRY_SERVICE_PASSWORD>
REGISTRY_SERVICE_PORT=6379

[atai-platform-lens-node-service-backend]
REGISTRY_SERVICE_IP_ADDRESS=clustercfg.atai-registry-service.<HASH>.<AWS_REGION
_CODE>.cache.amazonaws.com
REGISTRY_SERVICE_PASSWORD=<REGISTRY_SERVICE_PASSWORD>
REGISTRY_SERVICE_PORT=6379

[atai-platform-lens-service-db-backend]
REGISTRY_SERVICE_IP_ADDRESS=clustercfg.atai-registry-service.<HASH>.<AWS_REGION
_CODE>.cache.amazonaws.com
REGISTRY_SERVICE_PASSWORD=<REGISTRY_SERVICE_PASSWORD>
REGISTRY_SERVICE_PORT=6379
```

Cluster 2: access-manager

Same as Cluster 1, but:

1. Name: atai-access-manager-service
2. Auth token: Generate a new unique token (different from previous clusters)

⚠️ Store your Valkey host, port, and token in a secure location. You will need them later in the atai-platform prerequisites, Step 3.2: Kubernetes Secrets Generation.

Note: The secrets from **atai-registry-service** will still be required for the following secret. These values will be referenced as:

```
None
[atai-platform-access-manager-service-backend]
ACCESS_MANAGER_SERVICE_IP_ADDRESS=clustercfg.atai-access-manager-service.<HASH>
.<AWS_REGION>.cache.amazonaws.com
ACCESS_MANAGER_SERVICE_PASSWORD=<ACCESS_MANAGER_SERVICE_PASSWORD>
ACCESS_MANAGER_SERVICE_PORT=6379
REGISTRY_SERVICE_IP_ADDRESS=clustercfg.atai-registry-service.<HASH>.<AWS_REGION
_CODE>.cache.amazonaws.com
REGISTRY_SERVICE_PASSWORD=<REGISTRY_SERVICE_PASSWORD>
REGISTRY_SERVICE_PORT=6379
```

Cluster 3: api-events

Same as Cluster 1, but:

3. Name: atai-api-events-service
4. Auth token: Generate a new unique token (different from previous clusters)

⚠️ Store your Valkey host, port, and token in a secure location. You will need them later in the atai-platform prerequisites, Step 3.2: Kubernetes Secrets Generation.

Note: The secrets from **atai-registry-service** will still be required for the following secret. These values will be referenced as:

```
None
[atai-platform-api-events-service-backend]
API_EVENTS_SERVICE_IP_ADDRESS=clustercfg.atai-api-events-service.<HASH>.<AWS_RE
GION_CODE>.cache.amazonaws.com
API_EVENTS_SERVICE_PASSWORD=<API_EVENTS_SERVICE_PASSWORD>
API_EVENTS_SERVICE_PORT=6379
REGISTRY_SERVICE_IP_ADDRESS=clustercfg.atai-registry-service.<HASH>.<AWS_REGION
_CODE>.cache.amazonaws.com
REGISTRY_SERVICE_PASSWORD=<REGISTRY_SERVICE_PASSWORD>
REGISTRY_SERVICE_PORT=6379
```

Cluster 4: dfc (10 shards)

Same as Cluster 1, but:

1. Name: atai-dfc-service
2. Number of shards: 10 (instead of 1)
3. Replicas per shard: 0 (same for the other 10 shard caches)
4. Auth token: Generate a new unique token (different from previous clusters)

⚠️ Store your Valkey host, port, and token in a secure location. You will need them later in the atai-platform prerequisites, Step 3.2: Kubernetes Secrets Generation.

Note: The secrets from **atai-registry-service** will still be required for the following secret. These values will be referenced as:

```
None
[atai-platform-dfc-service-backend]
DFC_SERVICE_IP_ADDRESS=clustercfg.atai-dfc-service.<HASH>.<AWS_REGION_CODE>.cac
he.amazonaws.com
DFC_SERVICE_PASSWORD=<DFC_SERVICE_PASSWORD>
DFC_SERVICE_PORT=6379
REGISTRY_SERVICE_IP_ADDRESS=clustercfg.atai-stage-registry-service.<HASH>.<AWS_
REGION_CODE>.cache.amazonaws.com
```

```
REGISTRY_SERVICE_PASSWORD=<REGISTRY_SERVICE_PASSWORD>
REGISTRY_SERVICE_PORT=6379
```

Cluster 5: file (10 shards)

Same as Cluster 3, but:

1. Name: atai-{environment}-file-service
2. Number of shards: 10 (instead of 1)
3. Replicas per shard: 0 (same for the other 10 shard caches)
4. Auth token: Generate a new unique token (different from previous clusters)

⚠️ Store your Valkey host, port, and token in a secure location. You will need them later in the atai-platform prerequisites, Step 3.2: Kubernetes Secrets Generation.

Note: The secrets from **atai-registry-service** will still be required for the following secret. These values will be referenced as:

```
None
[atai-platform-file-service-worker-node]
FILE_SERVICE_IP_ADDRESS=clustercfg.atai-file-service.<HASH>.<AWS_REGION_CODE>.c
ache.amazonaws.com
FILE_SERVICE_PASSWORD=<FILE_SERVICE_PASSWORD>
FILE_SERVICE_PORT=6379
REGISTRY_SERVICE_IP_ADDRESS=clustercfg.atai-registry-service.<HASH>.<AWS_REGION
_CODE>.cache.amazonaws.com
REGISTRY_SERVICE_PASSWORD=<REGISTRY_SERVICE_PASSWORD>
REGISTRY_SERVICE_PORT=6379
```

Cluster 6: gpq (10 shards)

Same as Cluster 3, but:

1. Name: atai-gpq-service
2. Number of shards: 10 (instead of 1)
3. Replicas per shard: 0 (same for the other 10 shard caches)
4. Auth token: Generate a new unique token (different from previous clusters)

⚠️ Store your Valkey host, port, and token in a secure location. You will need them later in the atai-platform prerequisites, Step 3.2: Kubernetes Secrets Generation.

Note 1: The gpq secrets will be required for follow services:

- gpq-service-backend
- gpq-service-event-router

Note 2: The secrets from **atai-registry-service** will still be required for the following secret. These values will be referenced as:

```
None
[atai-platform-gpq-service-event-router]
GPQ_SERVIC_IP_ADDRESS=clustercfg.atai-gpq-service.<HASH>.<AWS_REGION_CODE>.cach
e.amazonaws.com
GPQ_SERVICE_PASSWORD=<GPQ_SERVICE_PASSWORD>
GPQ_SERVIC_PORT=6379
REGISTRY_SERVICE_IP_ADDRESS=clustercfg.atai-registry-service.<HASH>.<AWS_REGION
_CODE>.cache.amazonaws.com
REGISTRY_SERVICE_PASSWORD=<REGISTRY_SERVICE_PASSWORD>
REGISTRY_SERVICE_PORT=6379

[atai-platform-gpq-service-backend]
GPQ_SERVICE_IP_ADDRESS=clustercfg.atai-gpq-service.<HASH>.<AWS_REGION_CODE>.cac
he.amazonaws.com
GPQ_SERVICE_PASSWORD=<GPQ_SERVICE_PASSWORD>
GPQ_SERVICE_PORT=6379
REGISTRY_SERVICE_IP_ADDRESS=clustercfg.atai-registry-service.<HASH>.<AWS_REGION
_CODE>.cache.amazonaws.com
REGISTRY_SERVICE_PASSWORD=<REGISTRY_SERVICE_PASSWORD>
REGISTRY_SERVICE_PORT=6379
```

Cluster 7: health

Same as Cluster 1, but:

1. Name: atai-health-service
2. Auth token: Generate a new unique token (different from previous clusters)

⚠️ Store your Valkey host, port, and token in a secure location. You will need them later in the atai-platform prerequisites, Step 3.2: Kubernetes Secrets Generation.

Note: The secrets from **atai-registry-service** will still be required for the following secret. These values will be referenced as:

```
None
[atai-platform-health-service-backend]
HEALTH_SERVICE_IP_ADDRESS=clustercfg.atai-health-service.<HASH>.<AWS_REGION_COD
E>.cache.amazonaws.com
HEALTH_SERVICE_PASSWORD=<HEALTH_SERVICE_PASSWORD>
HEALTH_SERVICE_PORT=6379
```

```
REGISTRY_SERVICE_IP_ADDRESS=clustercfg.atai-registry-service.<HASH>.<AWS_REGION
_CODE>.cache.amazonaws.com
REGISTRY_SERVICE_PASSWORD=<REGISTRY_SERVICE_PASSWORD>
REGISTRY_SERVICE_PORT=6379
```

Cluster 8: lens (10 shards)

Same as Cluster 3, but:

1. Name: atai-lens-service
2. Number of shards: 10 (instead of 1)
3. Replicas per shard: 0 (same for the other 10 shard caches)
4. Auth token: Generate a new unique token (different from previous clusters)

⚠️ Store your Valkey host, port, and token in a secure location. You will need them later in the atai-platform prerequisites, Step 3.2: Kubernetes Secrets Generation.

Note: The secrets from **atai-registry-service** will still be required for the following secret. These values will be referenced as:

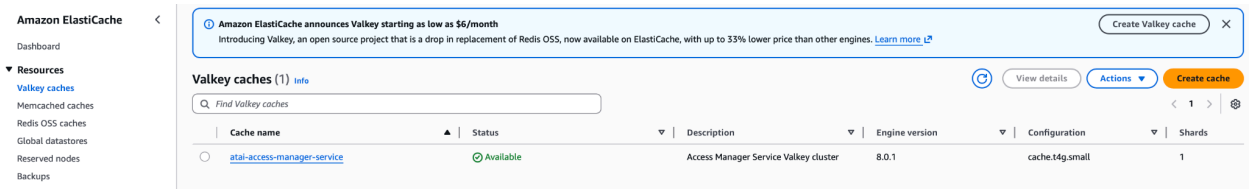
```
None
[atai-platform-lens-service-backend]
LENS_SERVICE_IP_ADDRESS=clustercfg.atai-lens-service.<HASH>.<AWS_REGION_CODE>.c
ache.amazonaws.com
LENS_SERVICE_PASSWORD=<LENS_SERVICE_PASSWORD>
LENS_SERVICE_PORT=6379
REGISTRY_SERVICE_IP_ADDRESS=clustercfg.atai-registry-service.<HASH>.<AWS_REGION
_CODE>.cache.amazonaws.com
REGISTRY_SERVICE_PASSWORD=<REGISTRY_SERVICE_PASSWORD>
REGISTRY_SERVICE_PORT=6379
```

Step 5: Store Auth Tokens in AWS Secrets Manager (Recommended)

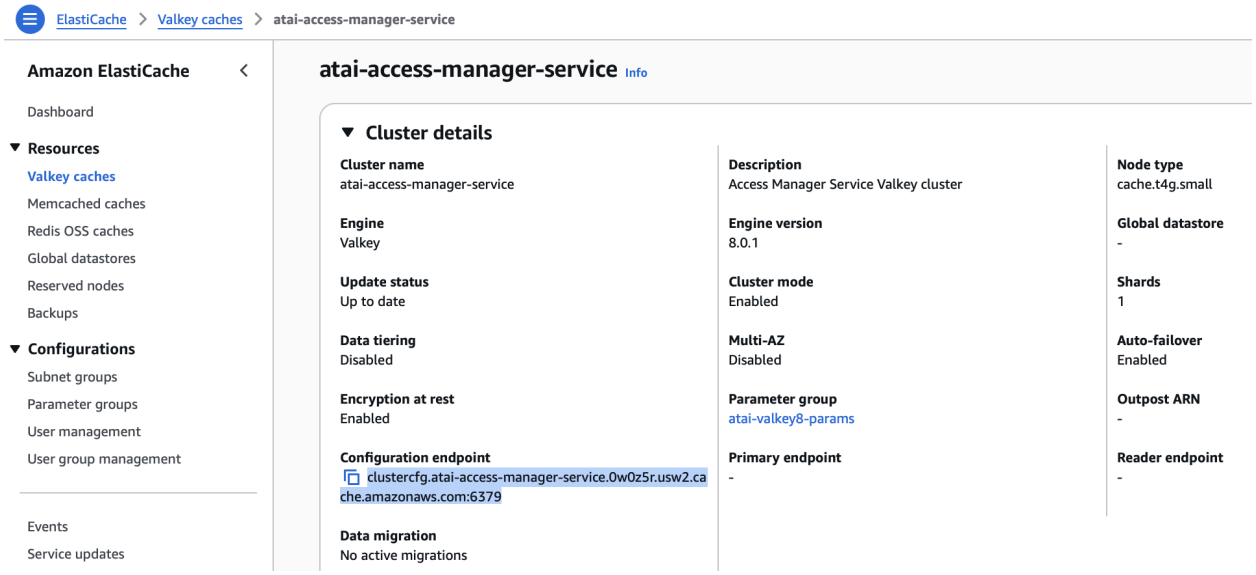
Note: This step is recommended but not mandatory. You can store credentials in Secrets Manager for easier access, better security, and integration with your applications. If you prefer to manage credentials differently, you can skip this step.

For each cluster, store the auth token and connection details:

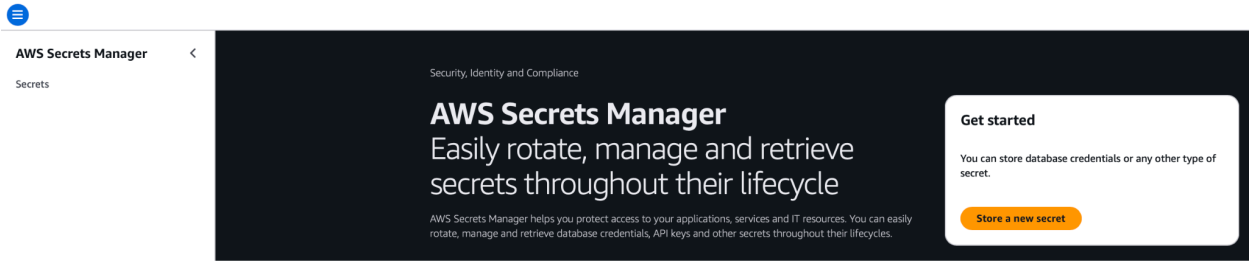
1. Go to ElastiCache → Valkey caches → Click on your Valkey cache cluster



2. Copy the **Configuration endpoint**

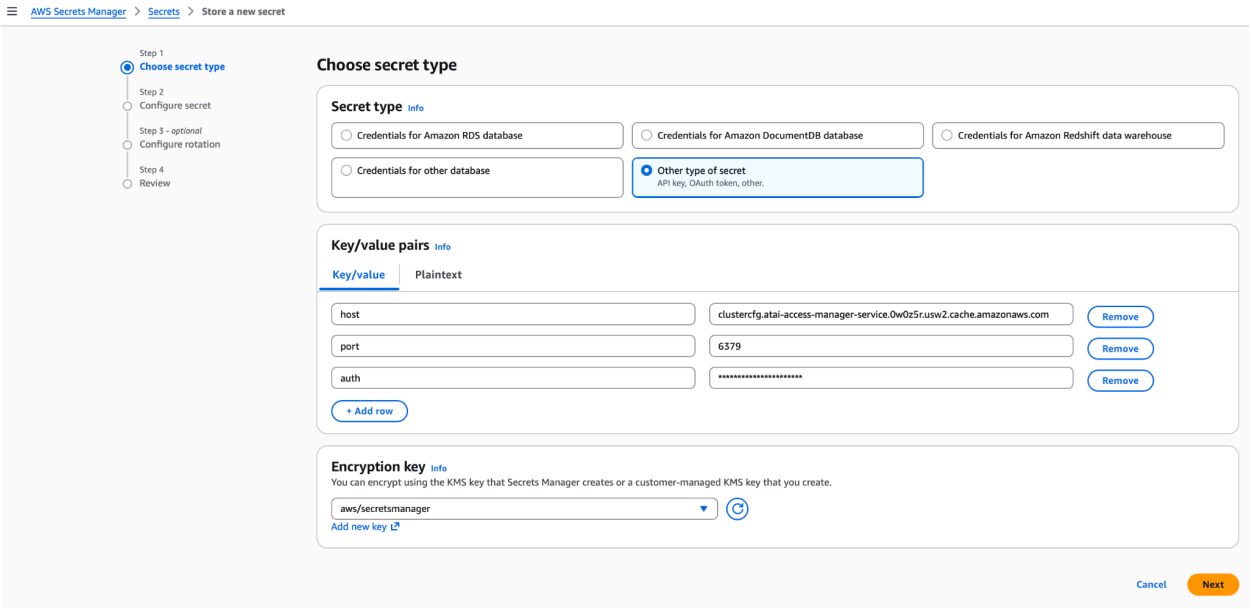


3. Go to Secrets Manager → **Store a new secret**

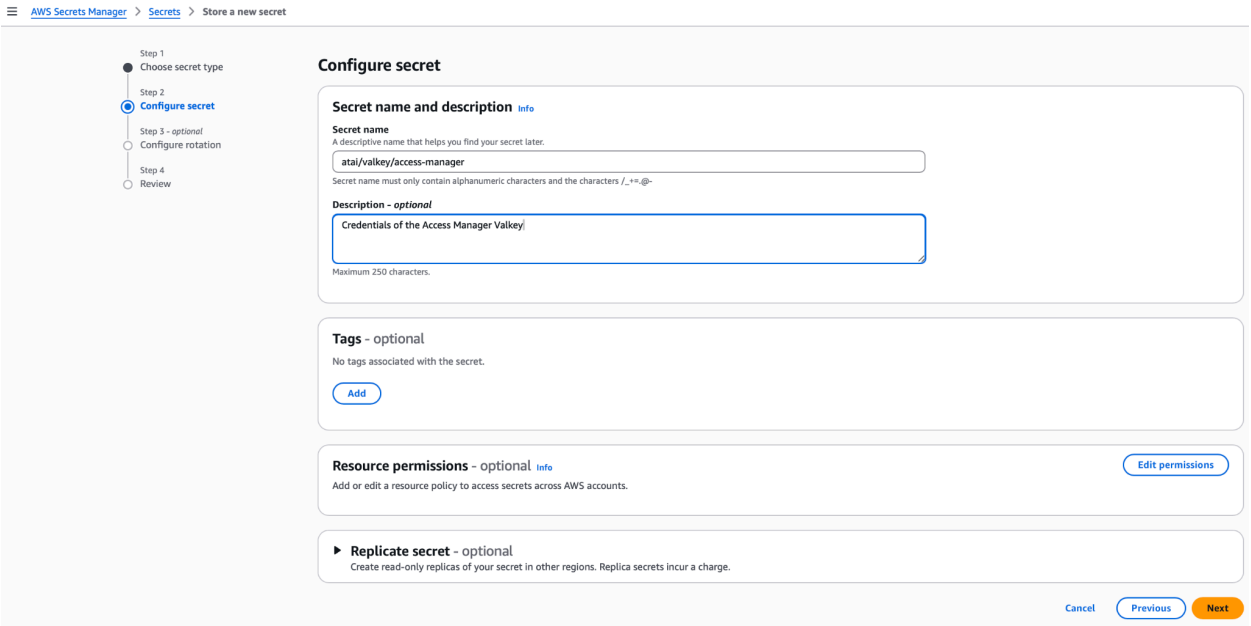


4. Secret type: Other type of secret
5. Key/value pairs: Add:

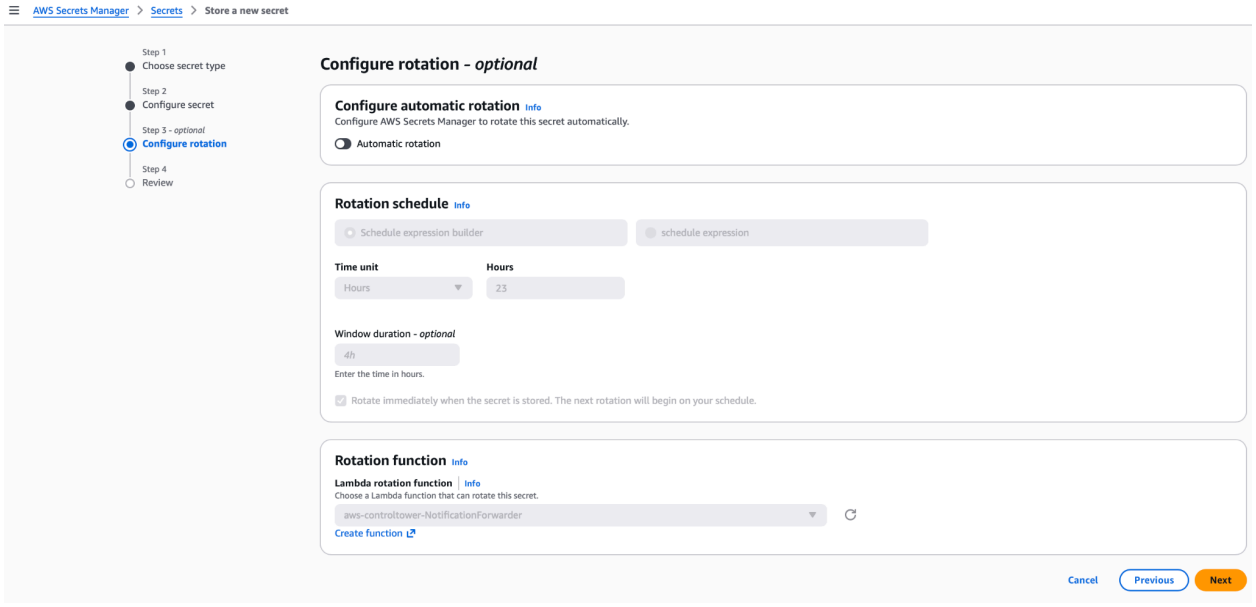
a. host: Configuration endpoint address (from cluster details, e.g., atai-access-manager-service.xxxxx.cache.amazonaws.com)
b. port: 6379
c. auth_token: The auth token you set for this cluster



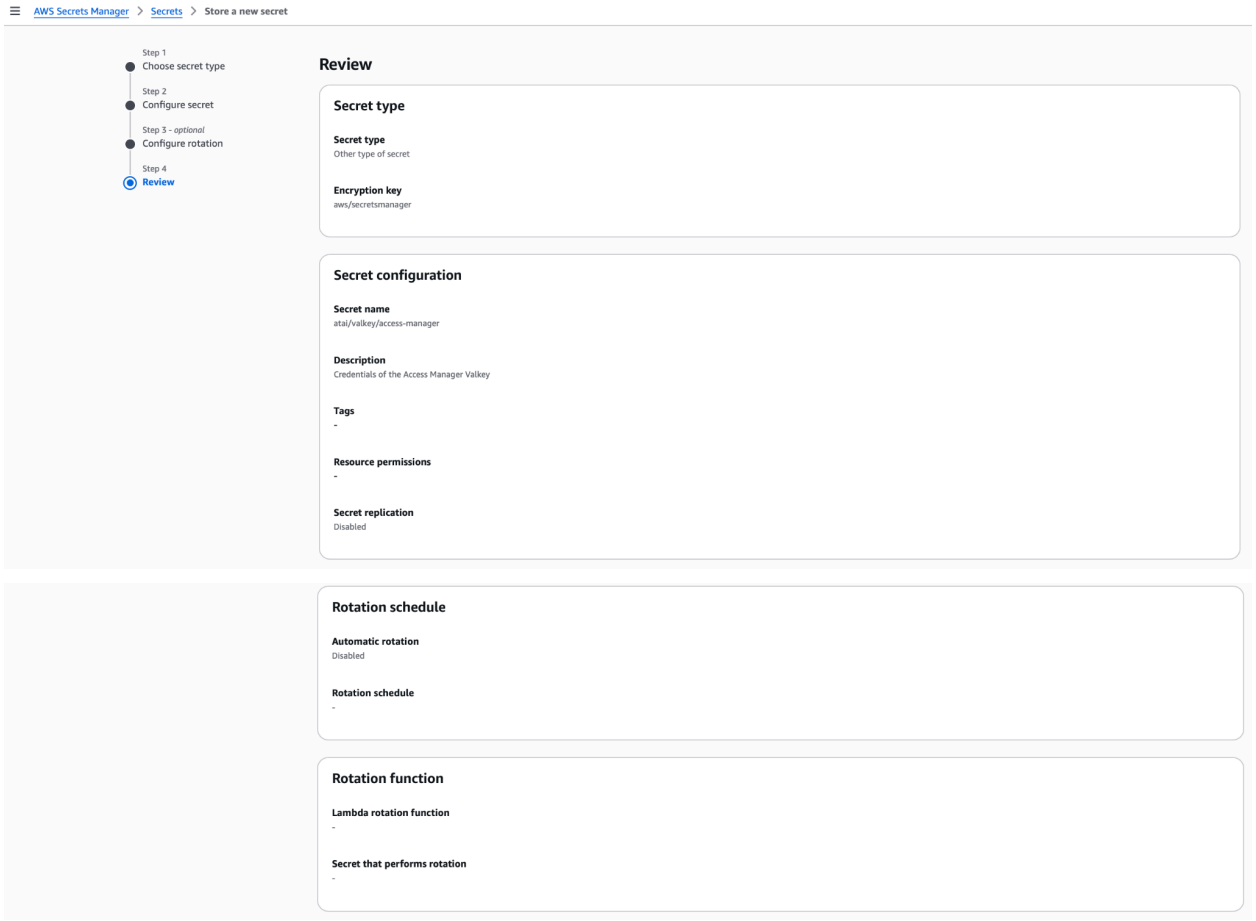
6. Secret name: atai/valkey/{service-name}
   Examples:
   a. atai/valkey/access-manager
   b. atai/valkey/api-events
   c. atai/valkey/dfc
   d. etc.
7. Encryption key: Use AWS managed key (default) or your KMS key
8. Click Next → Next

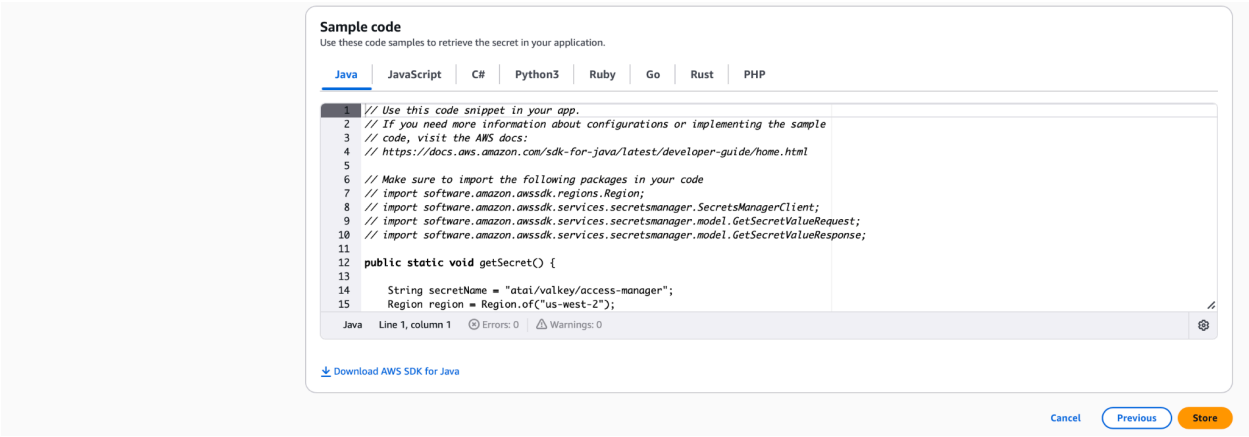


9. Do not configure rotation Click Next



10. Review and click on **Store**





11. Repeat for all 9 clusters with their respective:
    a. Configuration endpoint (host)
    b. Auth token
    c. Service name in the secret path

Benefits of using Secrets Manager:

- Applications can retrieve credentials programmatically
- Credentials are encrypted at rest
- Access is logged for audit purposes
- No need to hardcode credentials in application code

Alternative: If you skip this step, ensure you have the auth tokens and configuration endpoints stored securely elsewhere, as they are required for applications to connect to the Valkey clusters.

⚠️ Store your Valkey host, port, and token in a secure location. You will need them later in the *atai-platform* prerequisites, Step 3.2: Kubernetes Secrets Generation.

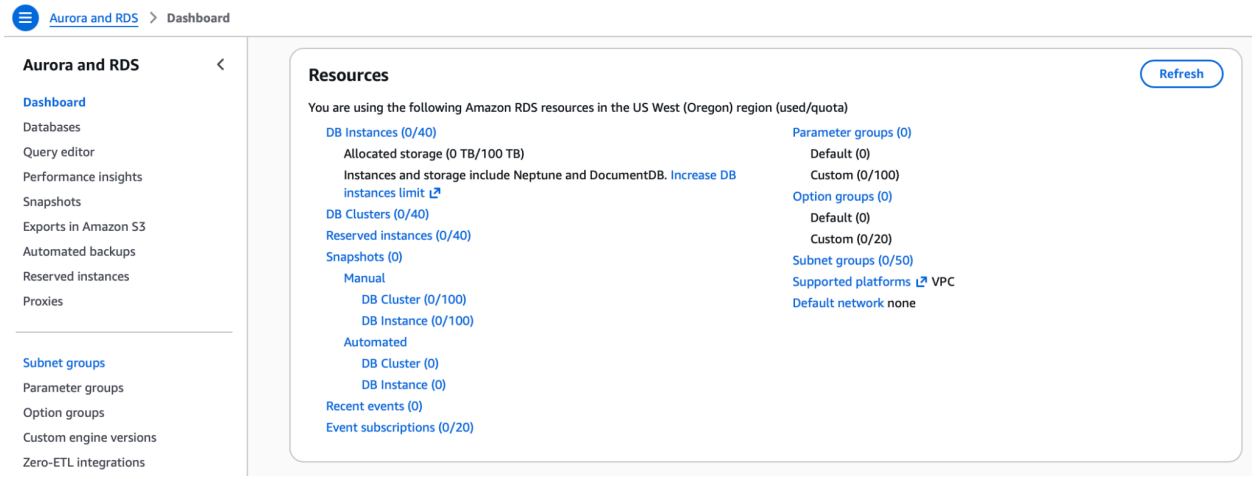
PostgreSQL database configuration

Prerequisites

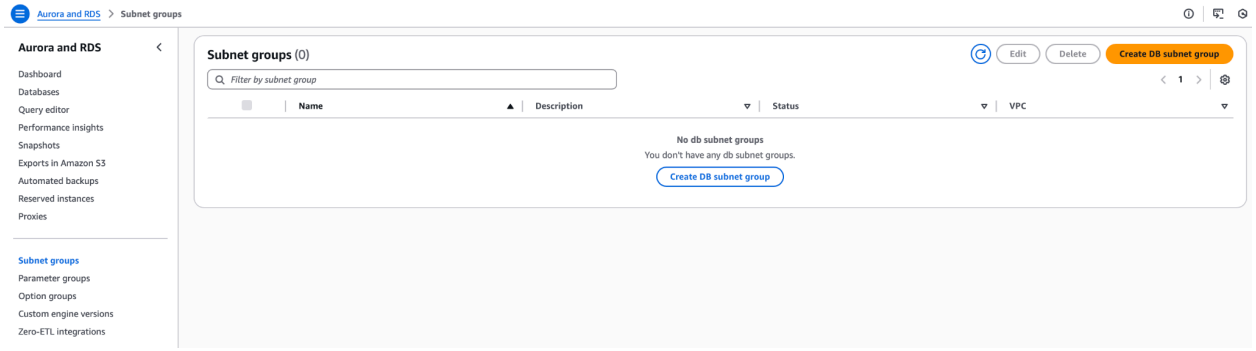
1. VPC with database subnets (at least 2 AZs required for Aurora subnet group)
2. (Optional) Security group allowing access from EKS pods
3. Database subnet group

Step 1: Create Database Subnet Group (if not already created)

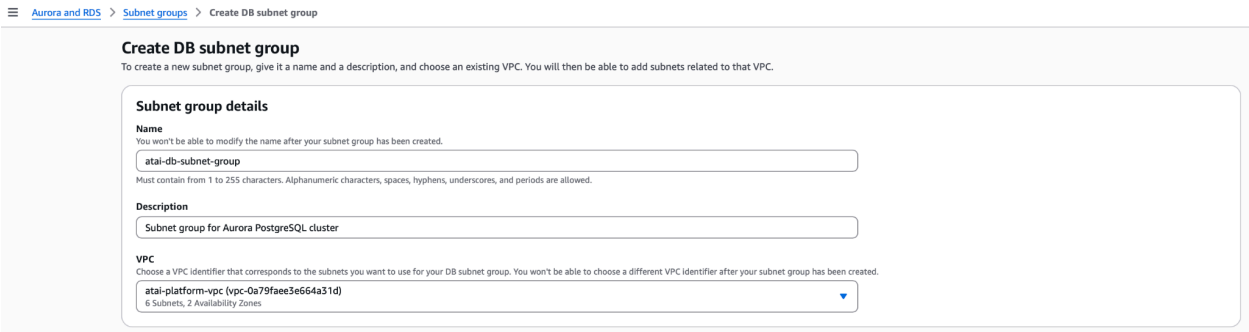
1. Go to RDS dashboard



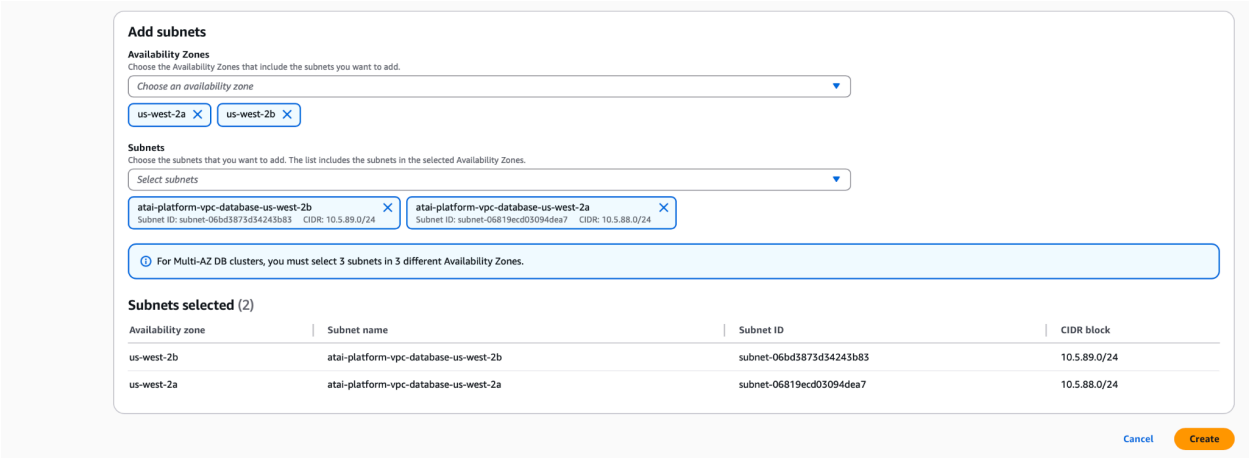
2. Subnet groups → Create DB subnet group



3. Name: atai-db-subnet-group (or your name)
4. Description: Subnet group for Aurora PostgreSQL cluster
5. VPC: Select your VPC



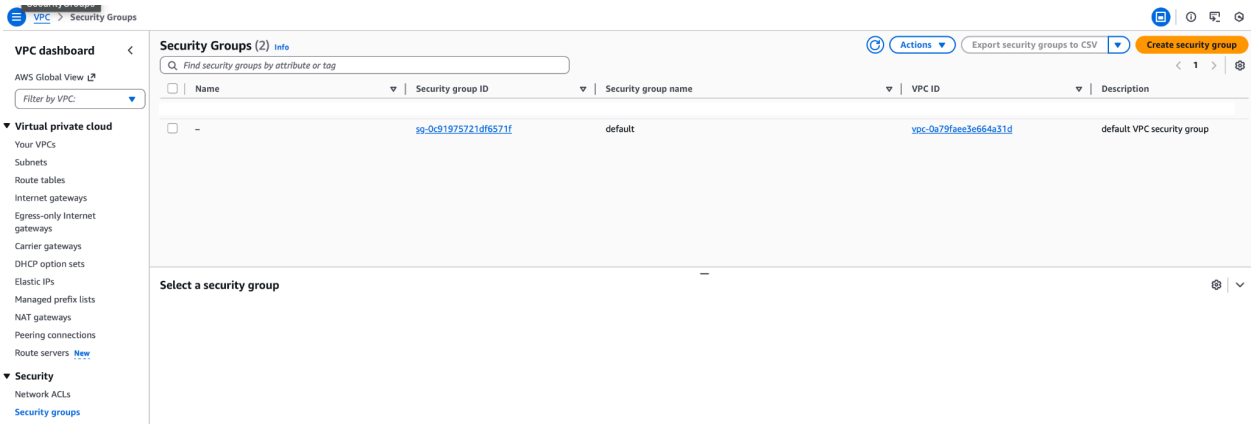
6. Availability Zones: Select at least 2 AZs (AWS requirement for Aurora)
    a. Note: Aurora requires at least 2 AZs for the subnet group, even if you deploy a single instance
    b. Example: Select us-west-2a and us-west-2b
7. Subnets: Select your database subnets atai-platform-vpc-database-us-west-2a and atai-platform-vpc-database-us-west-2b (e.g., 10.5.88.0/24 in us-west-2a and 10.5.89.0/24 in us-west-2b)



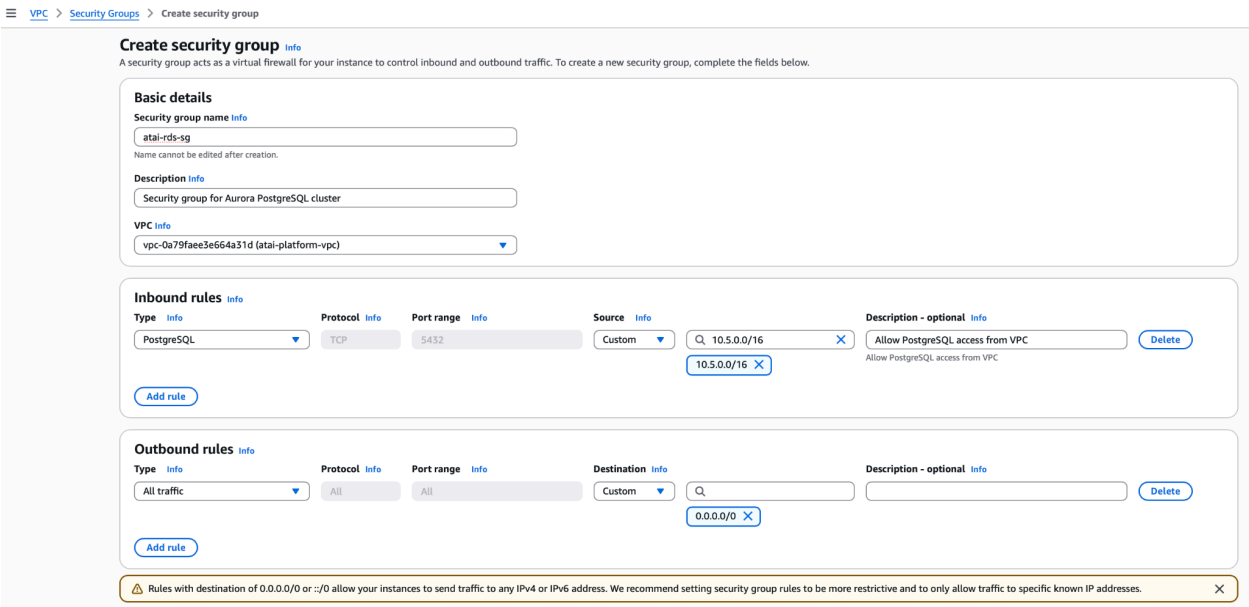
8. Click Create

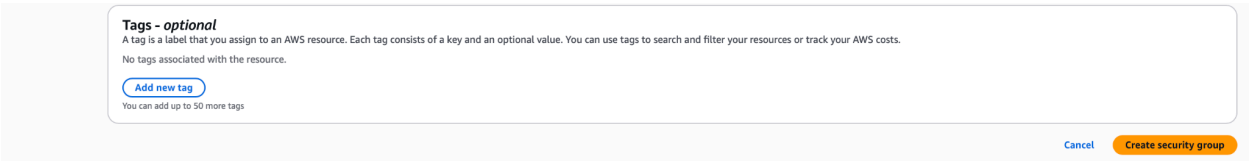
Step 2: Create Security Group (if not existing)

1. Go to VPC → Security Groups → Create security group



2. Name: atai-rds-sg (or your name)
3. Description: Security group for Aurora PostgreSQL cluster
4. VPC: Select your VPC from section VPC configuration Step 1
5. Inbound rules: Add rule:
    a. Type: PostgreSQL
    b. Port: 5432
    c. Source: Custom → Enter your VPC CIDR (e.g., 10.5.0.0/16)
    d. Description: Allow PostgreSQL access from VPC

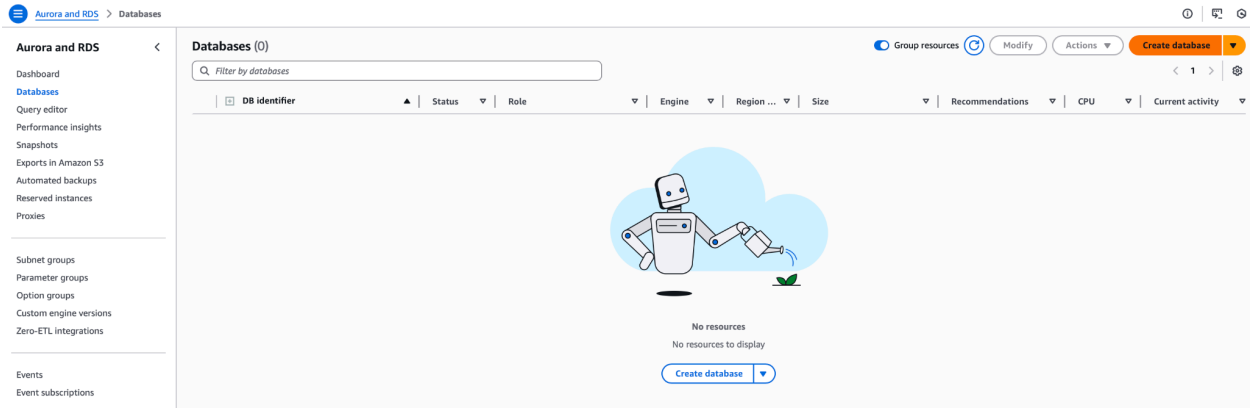




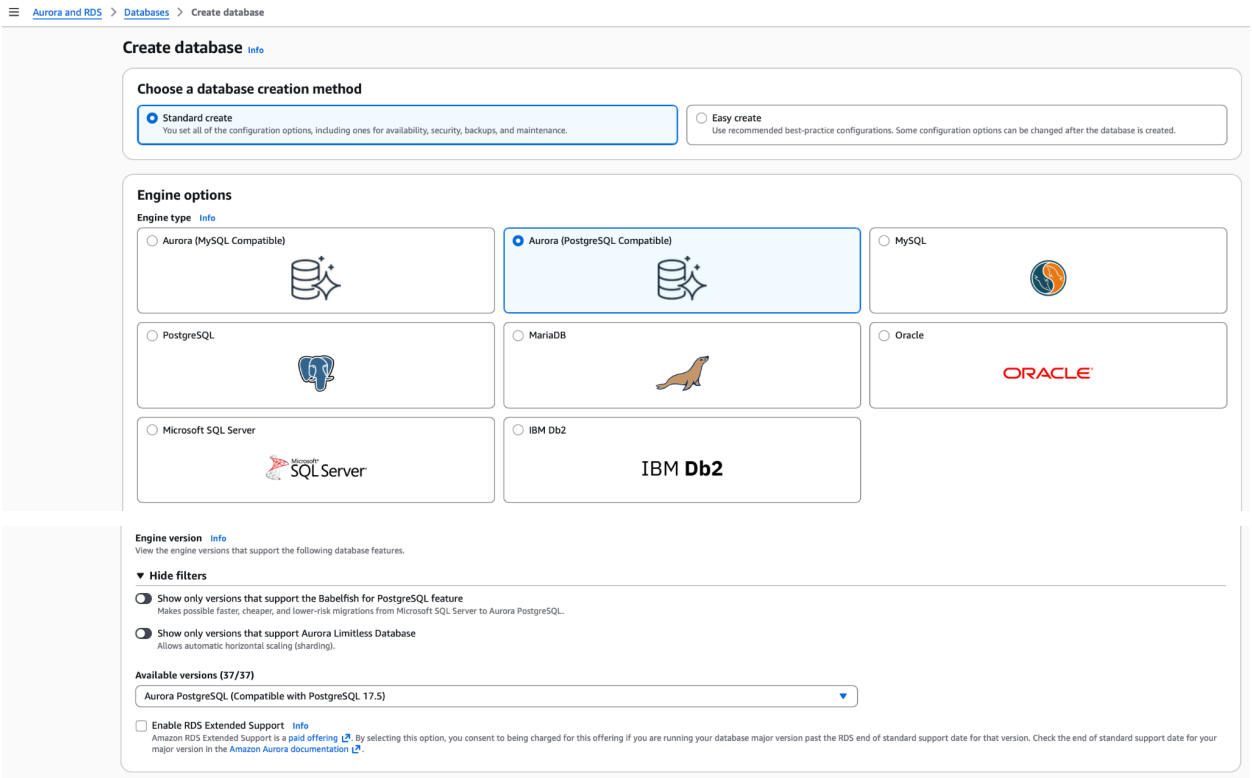
Note: For initial setup, opening to the VPC CIDR simplifies connectivity. Later, restrict to specific security groups (e.g., EKS node group security group) for tighter security.
Click Create security group

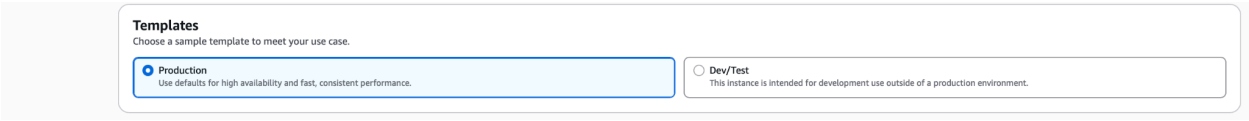
Step 3: Create Aurora PostgreSQL Cluster

1. Go to RDS → Databases → Create database

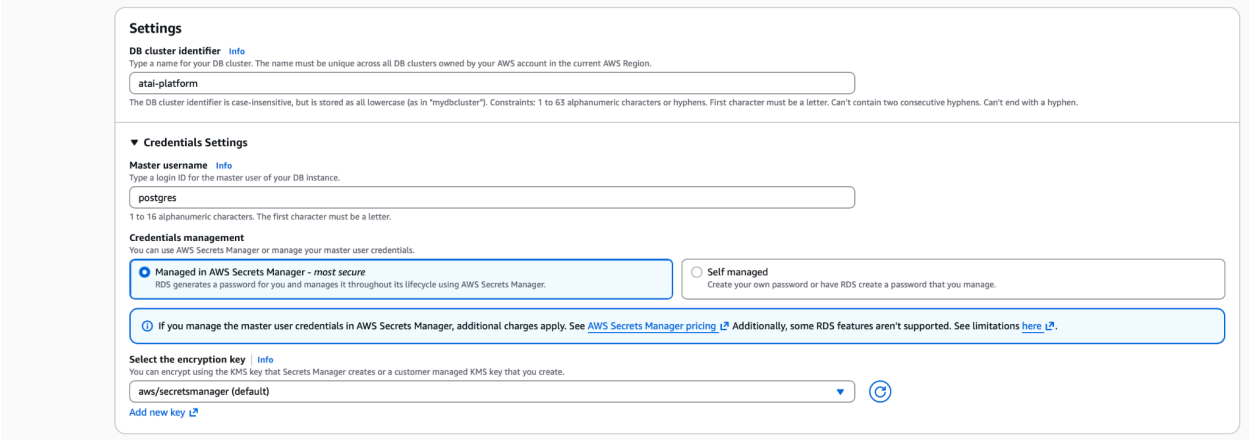


2. Database creation method: Standard create
3. Engine options:
4. Engine type: Amazon Aurora
    a. Edition: Amazon Aurora PostgreSQL-Compatible Edition
    b. Available versions: Select Aurora PostgreSQL 17.5
    c. Templates: Production (or Dev/Test for non-production)

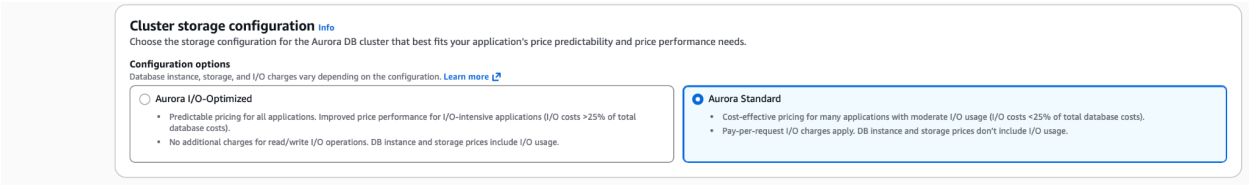




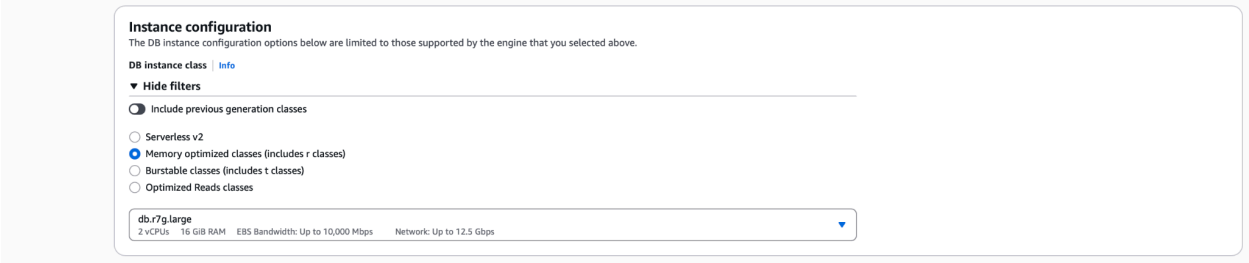
5. Settings:
    a. DB cluster identifier: atai-platform
    b. Master username: postgres (or your preferred username)
    c. Credentials management: Managed in AWS secrets manager



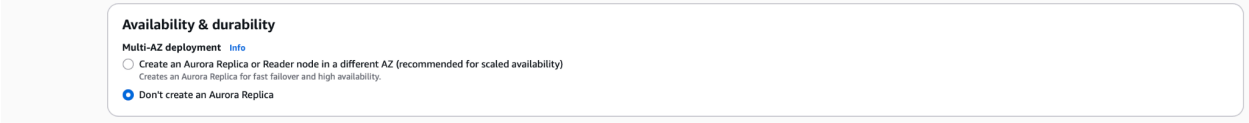
6. Cluster storage configuration
    a. Storage type: Aurora Standard



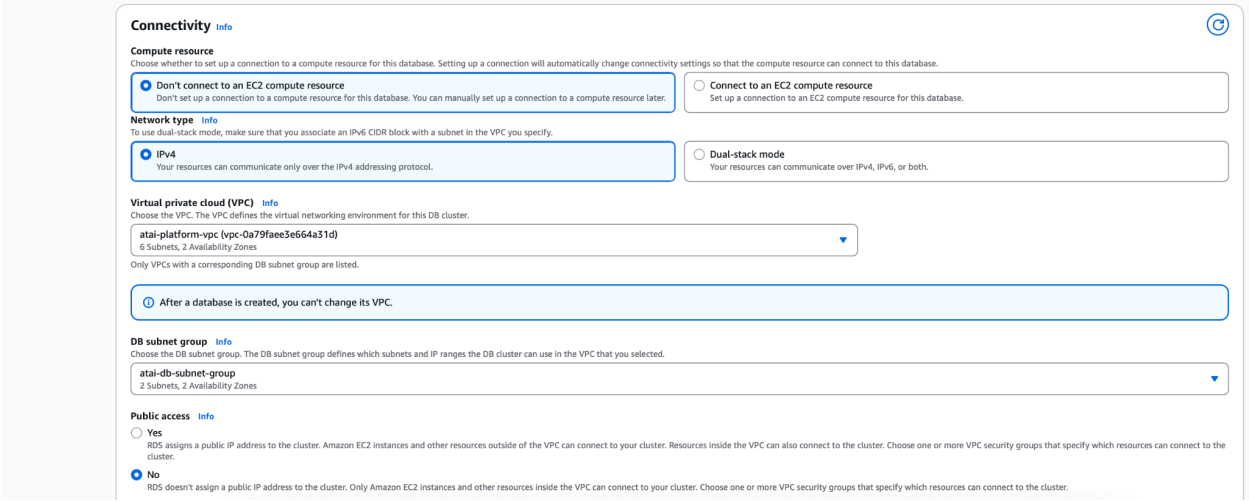
7. Instance configuration
    a. DB instance class: Select your instance class (e.g., db.t4g.medium for dev, db.r7g.large for production)



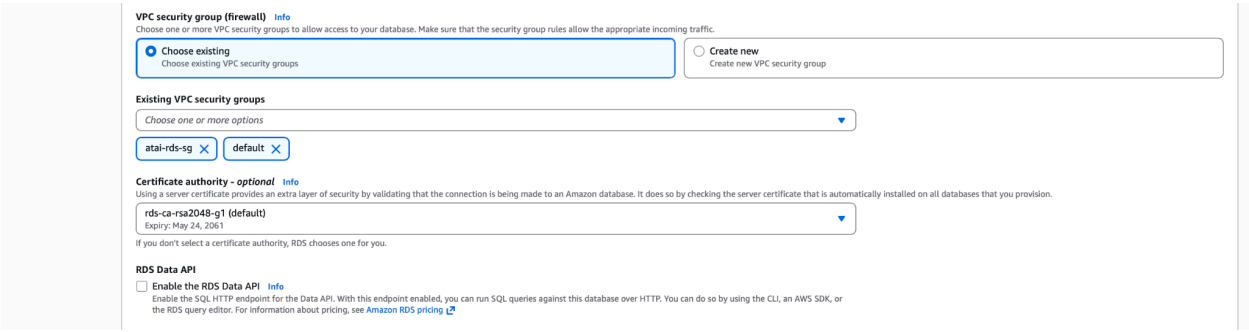
8. For Availability and Durability select Don’t create an Aurora Replica



9. Connectivity:
    a. Virtual private cloud (VPC): Select your VPC
    b. DB subnet group: Select your database subnet group (created in Step 1)
    c. Publicly accessible: No (should be in private subnets)

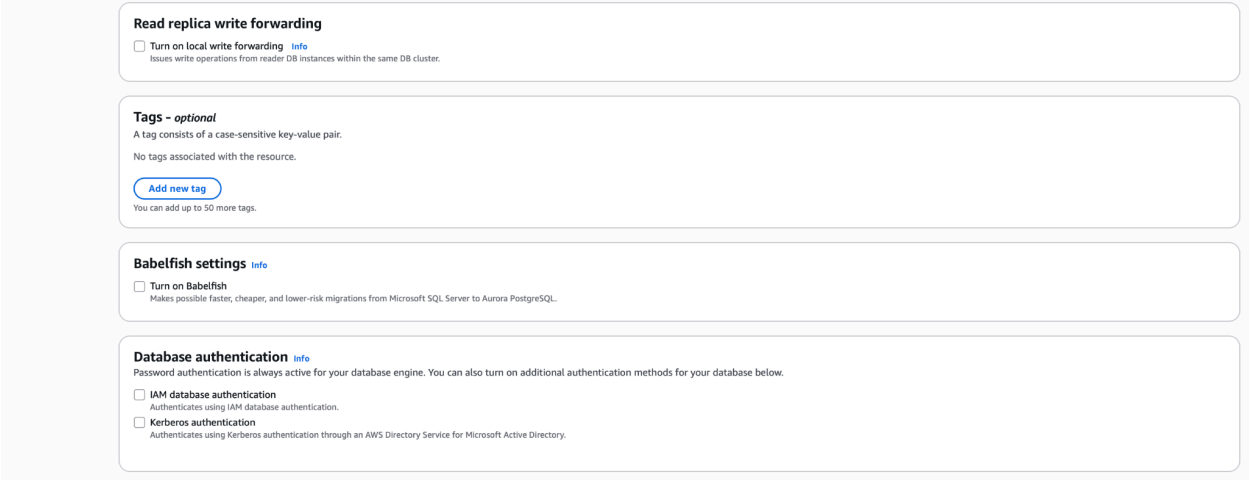


    d. VPC security group: Choose existing → Select atai-rds-sg
    e. Availability Zone: No preference (AWS will choose) or select your preferred AZ for the primary instance

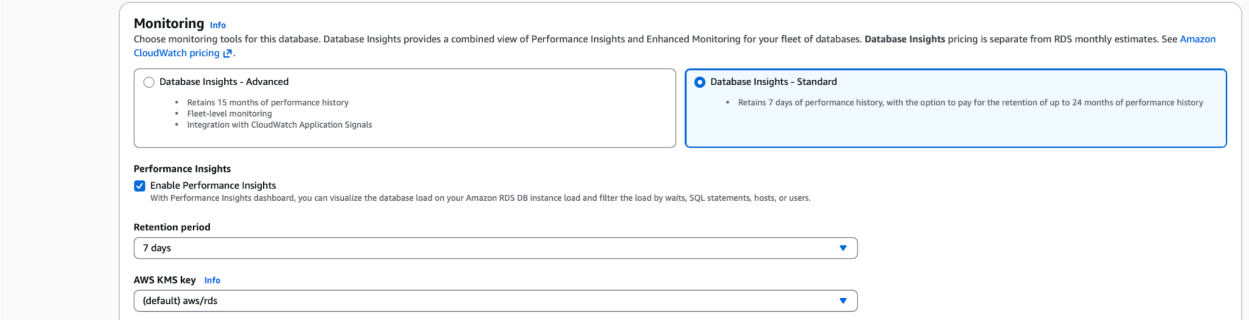


Note: For low latency, you can select the same AZ as your EKS node groups (e.g., us-west-2a)

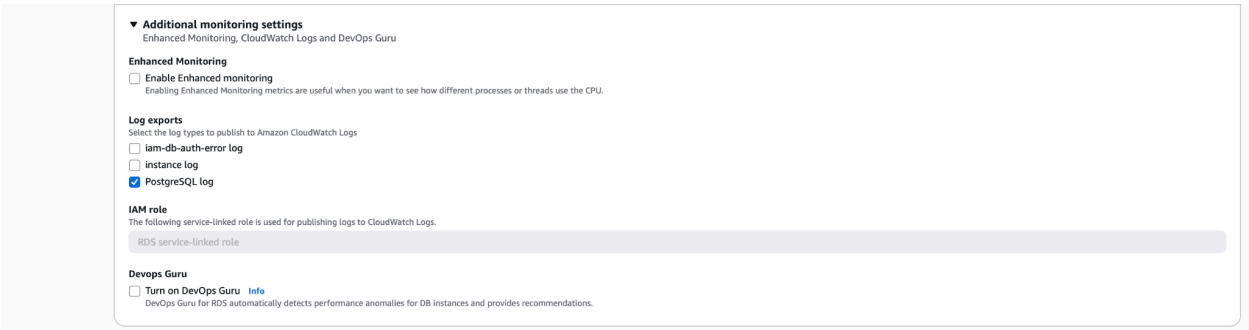
10. Leave the default values for Read replica write forwarding, Tags, Babelfish settings and Database authentication.



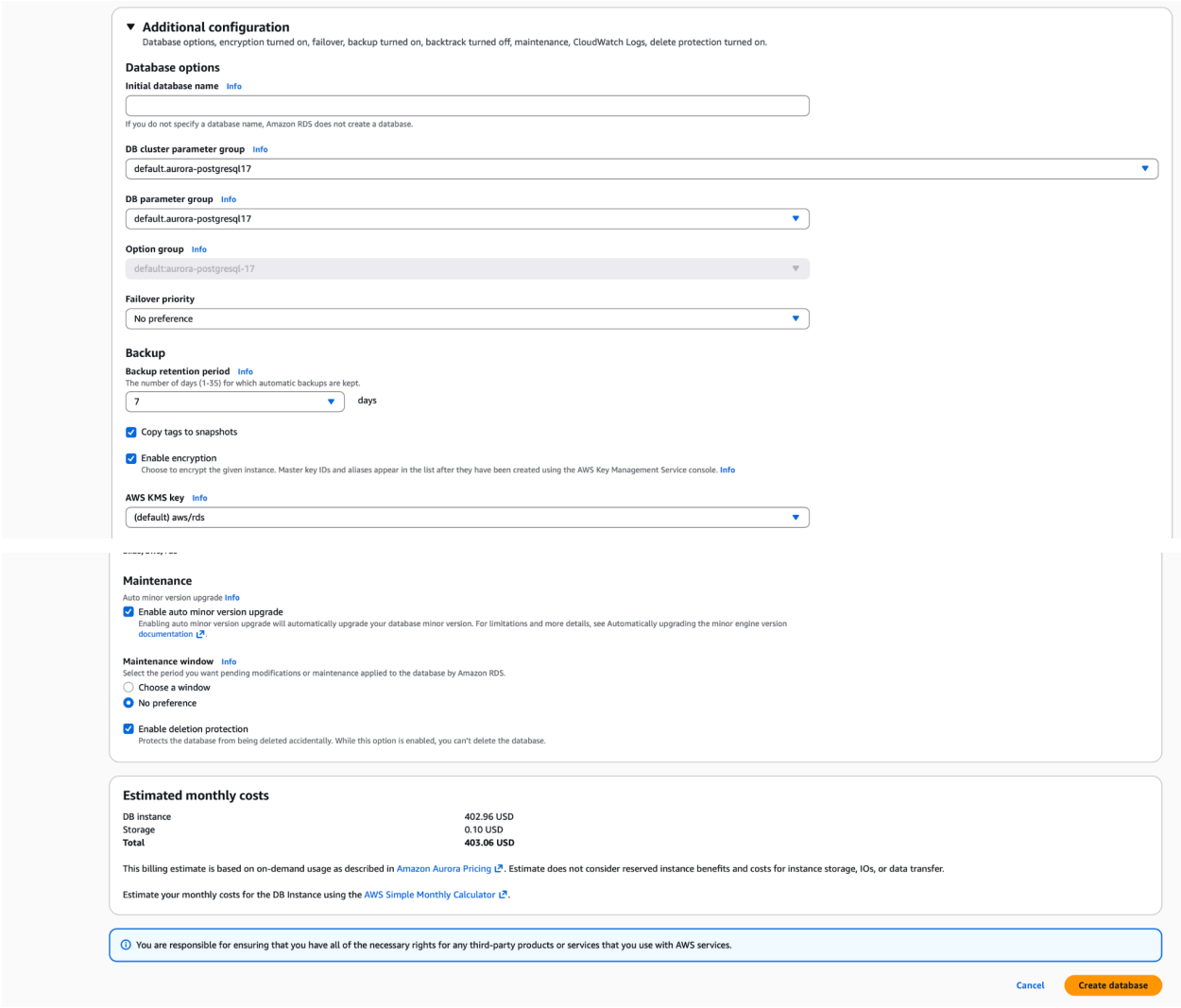
11. **Monitoring**
    a. Select Database insight - Standard



    b. Disable Enhanced monitoring
    c. Select PostgreSQL log under the Logs exports section
    d. You can turn off Devops Guru to avoid extra costs



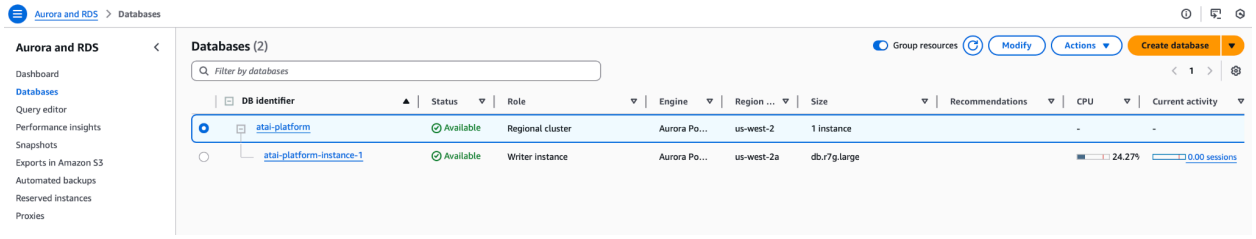
12. Additional configuration (optional — defaults are fine, customize if needed):
    a. Initial database name: Leave blank (or enter a name if you want to create an initial database)
    b. DB cluster parameter group: default.aurora-postgresql17 (or create custom if needed)
    c. DB parameter group: default.aurora-postgresql17 (or create custom if needed)
    d. Backup retention period: 7 days (default, or your preference)
    e. Backup window: No preference (default, or set a specific window)
    f. Enable encryption: Enable encryption (it's enabled by default)
    g. Encryption key: Use AWS managed key (default) or your KMS key



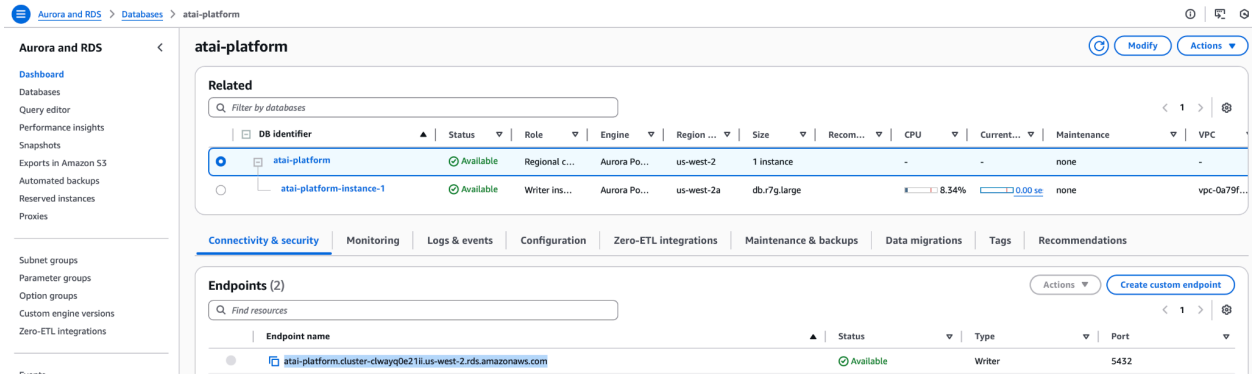
Note: No additional configurations are required — defaults work. You can customize backup retention, monitoring, and other settings if needed.

13. Click on **Create database**

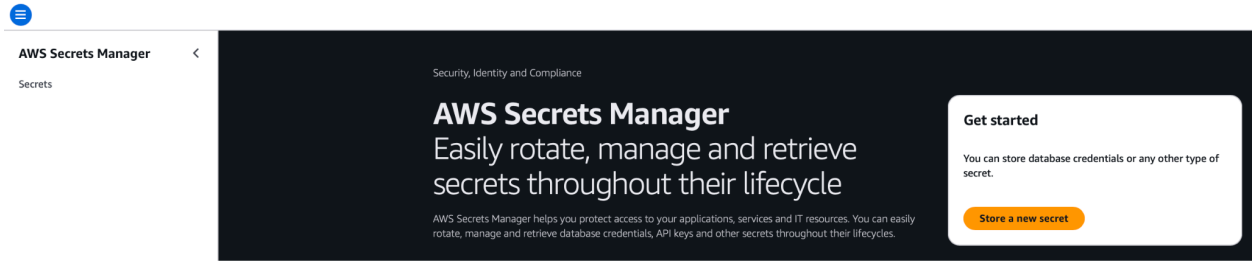
14. Get my database hostname
    a. After you create your database go to the RDS dashboard → **Databases**
    b. Select your database cluster



    c. Copy the **Cluster endpoint**

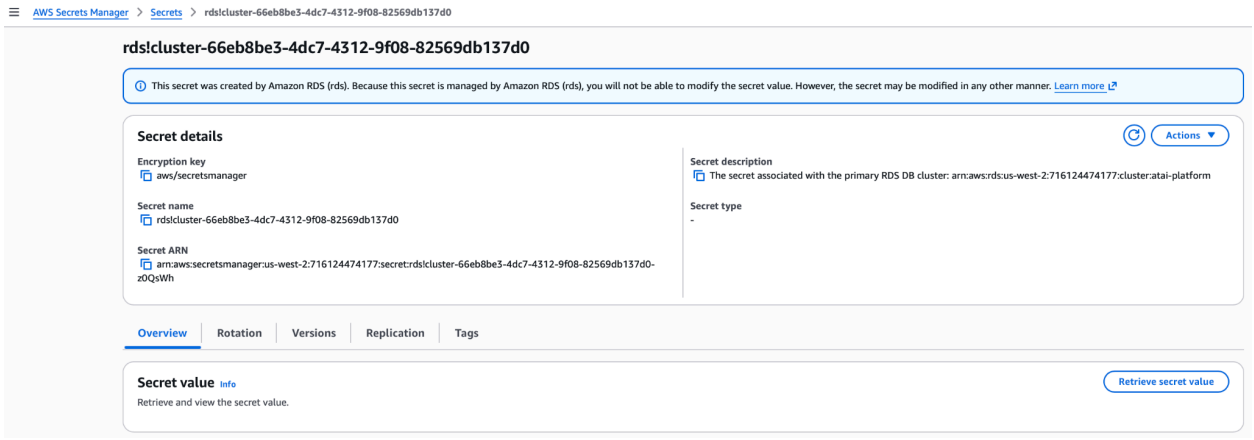


15. Get my database credentials
    a. Go to Secrets Manager → **Secrets**



    b. Click on the new RDS secret

    c. Click on **Get secret value**. You'll be able to see the user and password to connect to your RDS postgreSQL.



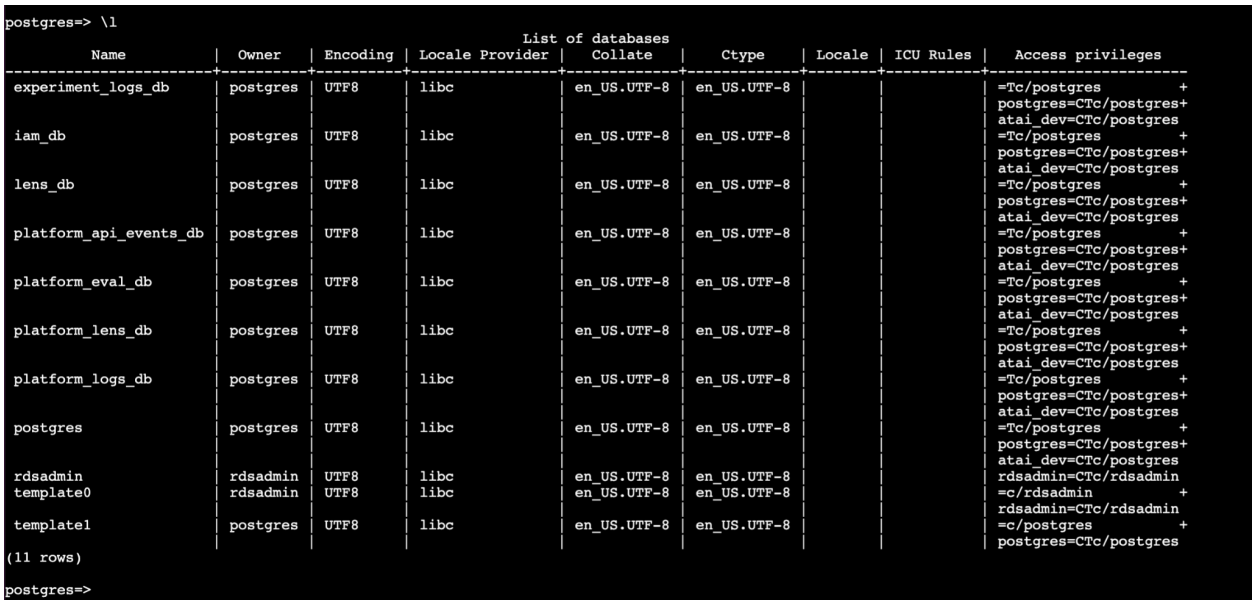
Step 4: Extra Database Configuration steps

Connect to your RDS, in our case we are using bastion host machine:

```
None
psql -h your-rds-endpoint.region.rds.amazonaws.com -U master_username -d
postgres
```

Create databases

```
None
CREATE DATABASE experiment_logs_db;
CREATE DATABASE iam_db;
CREATE DATABASE platform_api_events_db;
CREATE DATABASE platform_eval_db;
CREATE DATABASE platform_lens_db;
CREATE DATABASE platform_logs_db;
CREATE DATABASE lens_db;
```



```
postgres=> \l
                                                    List of databases
        Name         |  Owner   | Encoding | Locale Provider |   Collate   |    Ctype    | Locale | ICU Rules |   Access privileges
---------------------+----------+----------+-----------------+-------------+-------------+--------+-----------+-----------------------
 experiment_logs_db  | postgres | UTF8     | libc            | en_US.UTF-8 | en_US.UTF-8 |        |           | =Tc/postgres         +
                     |          |          |                 |             |             |        |           | postgres=CTc/postgres+
                     |          |          |                 |             |             |        |           | atai_dev=CTc/postgres
 iam_db              | postgres | UTF8     | libc            | en_US.UTF-8 | en_US.UTF-8 |        |           | =Tc/postgres         +
                     |          |          |                 |             |             |        |           | postgres=CTc/postgres+
                     |          |          |                 |             |             |        |           | atai_dev=CTc/postgres
 lens_db             | postgres | UTF8     | libc            | en_US.UTF-8 | en_US.UTF-8 |        |           | =Tc/postgres         +
                     |          |          |                 |             |             |        |           | postgres=CTc/postgres+
                     |          |          |                 |             |             |        |           | atai_dev=CTc/postgres
 platform_api_events_db | postgres | UTF8  | libc            | en_US.UTF-8 | en_US.UTF-8 |        |           | =Tc/postgres         +
                     |          |          |                 |             |             |        |           | postgres=CTc/postgres+
                     |          |          |                 |             |             |        |           | atai_dev=CTc/postgres
 platform_eval_db    | postgres | UTF8     | libc            | en_US.UTF-8 | en_US.UTF-8 |        |           | =Tc/postgres         +
                     |          |          |                 |             |             |        |           | postgres=CTc/postgres+
                     |          |          |                 |             |             |        |           | atai_dev=CTc/postgres
 platform_lens_db    | postgres | UTF8     | libc            | en_US.UTF-8 | en_US.UTF-8 |        |           | =Tc/postgres         +
                     |          |          |                 |             |             |        |           | postgres=CTc/postgres+
                     |          |          |                 |             |             |        |           | atai_dev=CTc/postgres
 platform_logs_db    | postgres | UTF8     | libc            | en_US.UTF-8 | en_US.UTF-8 |        |           | =Tc/postgres         +
                     |          |          |                 |             |             |        |           | postgres=CTc/postgres+
                     |          |          |                 |             |             |        |           | atai_dev=CTc/postgres
 postgres            | postgres | UTF8     | libc            | en_US.UTF-8 | en_US.UTF-8 |        |           | =Tc/postgres         +
                     |          |          |                 |             |             |        |           | postgres=CTc/postgres+
                     |          |          |                 |             |             |        |           | atai_dev=CTc/postgres
 rdsadmin            | rdsadmin | UTF8     | libc            | en_US.UTF-8 | en_US.UTF-8 |        |           | rdsadmin=CTc/rdsadmin
 template0           | rdsadmin | UTF8     | libc            | en_US.UTF-8 | en_US.UTF-8 |        |           | =c/rdsadmin          +
                     |          |          |                 |             |             |        |           | rdsadmin=CTc/rdsadmin
 template1           | postgres | UTF8     | libc            | en_US.UTF-8 | en_US.UTF-8 |        |           | =c/postgres          +
                     |          |          |                 |             |             |        |           | postgres=CTc/postgres
(11 rows)

postgres=>
```

Create atai_dev database user

```
None
CREATE USER atai_dev WITH PASSWORD 'your_secure_password';
```

```
(10 rows)

postgres=> \du
                                   List of roles
      Role name      |                         Attributes
---------------------+------------------------------------------------------------
 atai_dev            |
 postgres            | Create role, Create DB                                    +
                     | Password valid until infinity
 rds_ad              | Cannot login
 rds_extension       | No inheritance, Cannot login
 rds_iam             | Cannot login
 rds_password        | Cannot login
 rds_replication     | Cannot login
 rds_superuser       | Cannot login
 rdsadmin            | Superuser, Create role, Create DB, Replication, Bypass RLS+
                     | Password valid until infinity
 rdswriteforwarduser | No inheritance

postgres=>
```

⚠️ Store your atai_dev user and password in a secure location. You will need them later in the *atai-platform* prerequisites, Step 3.2: Kubernetes Secrets Generation.

Note 1: The **database** secrets will be required for following services:

- lens-service-db-backend
- api-events-service-backend

Note 2: Add the secrets in the same section as the services mentioned in **Note 1**. These values will be referenced as:

```
None
[atai-platform-api-events-service-backend]
...
API_EVENTS_SERVICE_DB_HOST=<DB_NAME>.<CLUSTER_HASH>.<AWS_REGION>.rds.amazonaws.
com
API_EVENTS_SERVICE_DB_PASS=<API_EVENTS_SERVICE_DB_PASS>
API_EVENTS_SERVICE_DB_PORT=5432
API_EVENTS_SERVICE_DB_USER=atai_dev
```

```
[atai-platform-lens-service-db-backend]
...
POSTGRESQL_HOST=<DB_NAME>.<CLUSTER_HASH>.<AWS_REGION>.rds.amazonaws.com

POSTGRESQL_PASSWORD=<POSTGRESQL_PASSWORD>
POSTGRESQL_PORT=5432
POSTGRESQL_USER=atai_dev
```

Then assign proper permissions to the new PostgreSQL user:

```
None
GRANT ALL PRIVILEGES ON DATABASE experiment_logs_db TO atai_dev;
GRANT ALL PRIVILEGES ON DATABASE iam_db TO atai_dev;
GRANT ALL PRIVILEGES ON DATABASE platform_api_events_db TO atai_dev;
GRANT ALL PRIVILEGES ON DATABASE platform_eval_db TO atai_dev;
GRANT ALL PRIVILEGES ON DATABASE platform_lens_db TO atai_dev;
GRANT ALL PRIVILEGES ON DATABASE platform_logs_db TO atai_dev;
GRANT ALL PRIVILEGES ON DATABASE lens_db TO atai_dev;
GRANT ALL PRIVILEGES ON DATABASE postgres TO atai_dev;
```

Additional permissions required:

```
None
\c database_name (this should be done on all DBs)

GRANT USAGE ON SCHEMA public TO atai_dev;
GRANT CREATE ON SCHEMA public TO atai_dev;
GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA public TO atai_dev;
GRANT ALL PRIVILEGES ON ALL SEQUENCES IN SCHEMA public TO atai_dev; ALTER
DEFAULT PRIVILEGES IN SCHEMA public GRANT ALL ON TABLES TO atai_dev;
ALTER DEFAULT PRIVILEGES IN SCHEMA public GRANT ALL ON SEQUENCES TO atai_dev;
GRANT ALL PRIVILEGES ON ALL FUNCTIONS IN SCHEMA public TO atai_dev; ALTER
DEFAULT PRIVILEGES IN SCHEMA public GRANT ALL ON FUNCTIONS TO atai_dev;
```

Solve a common permission issue:

```
None
-- Issue:psycopg2.errors.InsufficientPrivilege:
--   permission denied for schema public
---  LINE 1: CREATE TABLE IF NOT EXISTS platform_lens_db(
--   Fix: on that specific DB

GRANT CREATE ON SCHEMA public TO atai_dev;
GRANT USAGE ON SCHEMA public TO atai_dev;
```

EKS cluster configuration

Prerequisites

1. An existing VPC and subnets that meet Amazon EKS requirements
2. At least two private subnets in your VPC:
    a. Recommended naming:
        i. atai-platform-vpc-private-us-west-2a,
        ii. atai-platform-vpc-private-us-west-2b (or similar)
    b. Recommended CIDR blocks: /20 per subnet
        i. Example: 10.5.0.0/20 (subnet 1)
        ii. Example: 10.5.16.0/20 (subnet 2)
    c. Subnets must be in different Availability Zones
    d. Subnets must have routes to NAT Gateway or Internet Gateway for outbound internet access
3. The kubectl command line tool is required. The version can be the same as or up to one minor version earlier or later than the Kubernetes version of your cluster.
4. Version 2.12.3 or later or version 1.27.160 or later of the AWS Command Line Interface (AWS CLI) installed and configured on your device.
5. An IAM principal with permissions to create and describe an Amazon EKS cluster

Step 1: Create cluster IAM role

1. If you already have a cluster IAM role, or you're going to create your cluster with eksctl, then you can skip this step. By default, eksctl creates a role for you.
2. Run the following command to create an IAM trust policy JSON file.

```
None
cat > eks-cluster-role-trust-policy.json << 'EOF'
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "eks.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
EOF
```

3. Create the Amazon EKS cluster IAM role. If necessary, preface eks-cluster-role-trust-policy.json with the path on your computer that you wrote the file to in the previous step. The command associates the trust policy that you created in the previous step to the role. To create an IAM role, the IAM principal that is creating the role must be assigned the iam:CreateRole action (permission).

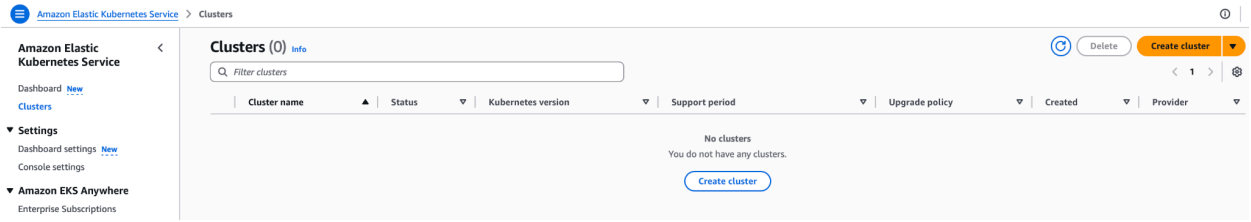
```
None
aws iam create-role --role-name atai-platform-eks-cluster-role
--assume-role-policy-document file://"eks-cluster-role-trust-policy.json"
```

4. You can assign either the Amazon EKS managed policy or create your own custom policy. For the minimum permissions that you must use in your custom policy, see Amazon EKS cluster IAM role.
   Attach the Amazon EKS managed policy named AmazonEKSClusterPolicy to the role. To attach an IAM policy to an IAM principal, the principal that is attaching the policy must be assigned one of the following IAM actions (permissions): iam:AttachUserPolicy or iam:AttachRolePolicy.

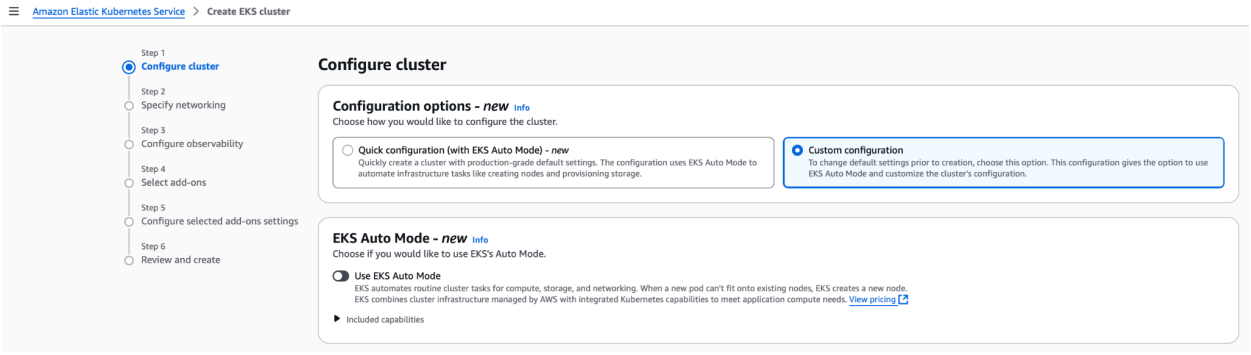
```
None
aws iam attach-role-policy --policy-arn
arn:aws:iam::aws:policy/AmazonEKSClusterPolicy --role-name
atai-platform-eks-cluster-role
```

Step 2: Create cluster

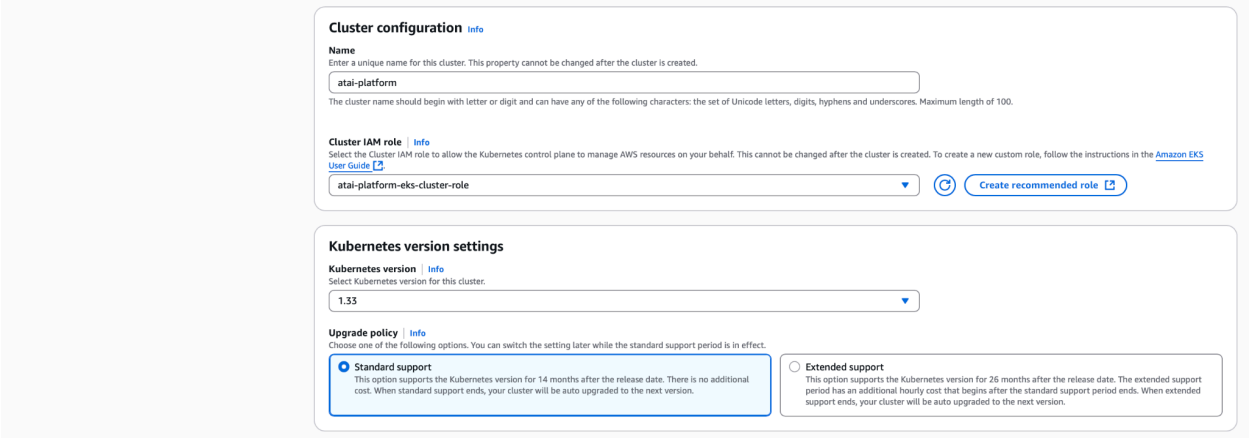
1. Open EKS Console → Click on **Create cluster**



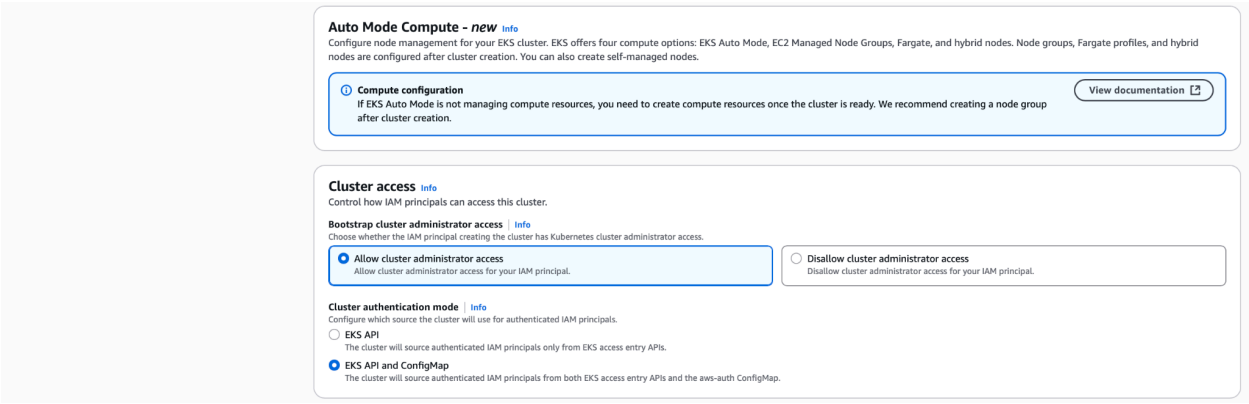
2. Under Configuration options select Custom configuration
3. Under EKS Auto Mode, toggle Use EKS Auto Mode off.



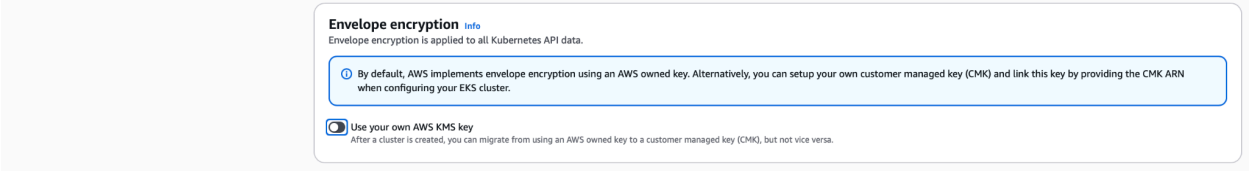
4. On the Configure cluster page, enter the following fields:
    a. Name: atai-platform
    b. Cluster IAM role – Choose the Amazon EKS cluster IAM role that you created in the Step 1 to allow the Kubernetes control plane to manage AWS resources on your behalf.
    c. Kubernetes version: 1.33
    d. Support type: Standard support



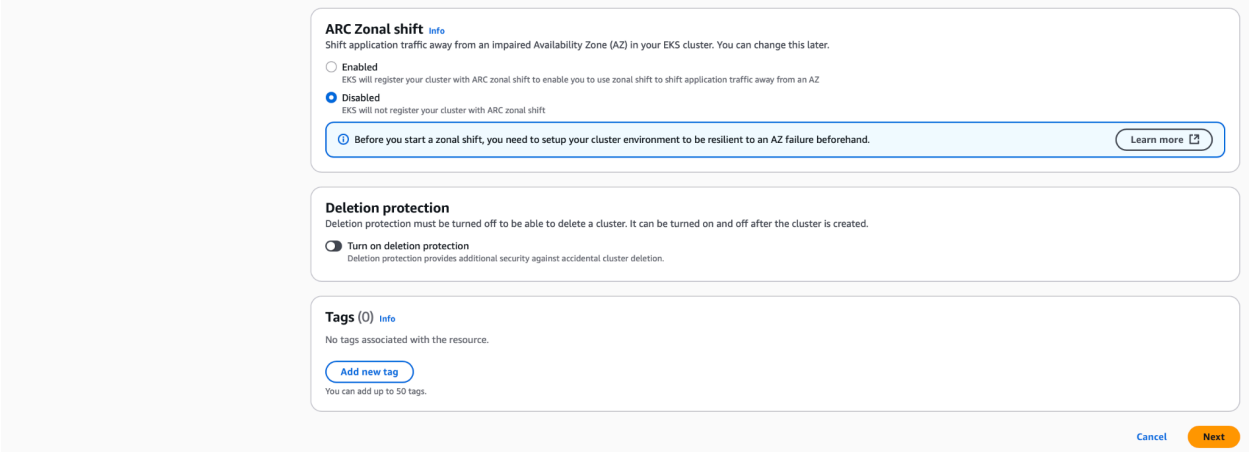
5. Cluster access
    a. Select Allow cluster administrator access
    b. Cluster authentication mode: EKS API and ConfigMap



6. Envelop encryption
    a. By default, AWS implements envelope encryption using an AWS owned key. Alternatively, you can setup your own customer managed key (CMK) and link this key by providing the CMK ARN when configuring your EKS cluster.



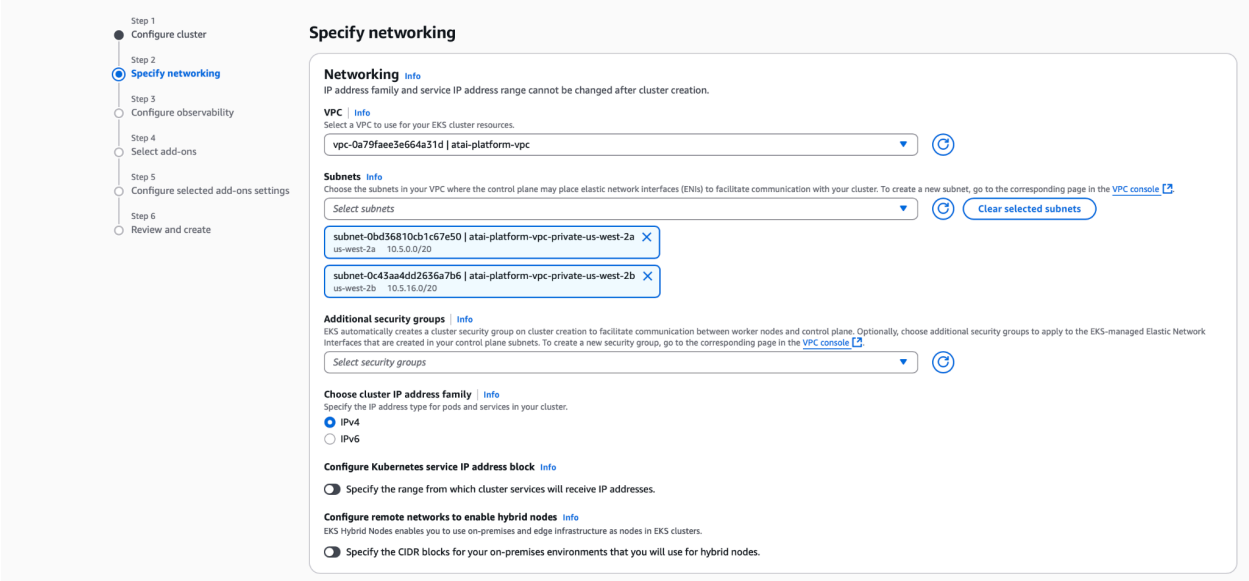
7. Use the default configuration for ARC Zonal Shift (disabled by default).
   (Optional) Enable Deletion protection
   (Optional) Add tags
   Click on **Next**



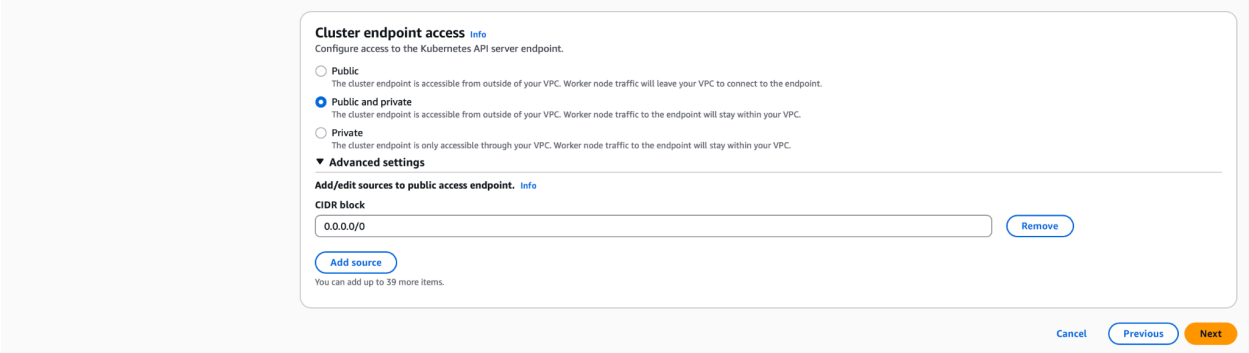
8. Networking
    a. VPC: Select your VPC
    b. Subnets: Select at least two private subnets:
        i. atai-platform-vpc-private-us-west-2a (10.5.0.0/20)
        ii. atai-platform-vpc-private-us-west-2b (10.5.16.0/20)
    c. (Optional) Security groups: EKS automatically creates a cluster security group on cluster creation to facilitate communication between worker nodes and control plane

After the cluster is created, you must **retrieve the security group ID** assigned by AWS.
This ID is required when creating Launch Templates for your Managed Node Groups, to ensure proper networking and access between the control plane and the worker nodes. To learn how to get the security group automatically created by EKS got to Step 4: Get the default cluster security group

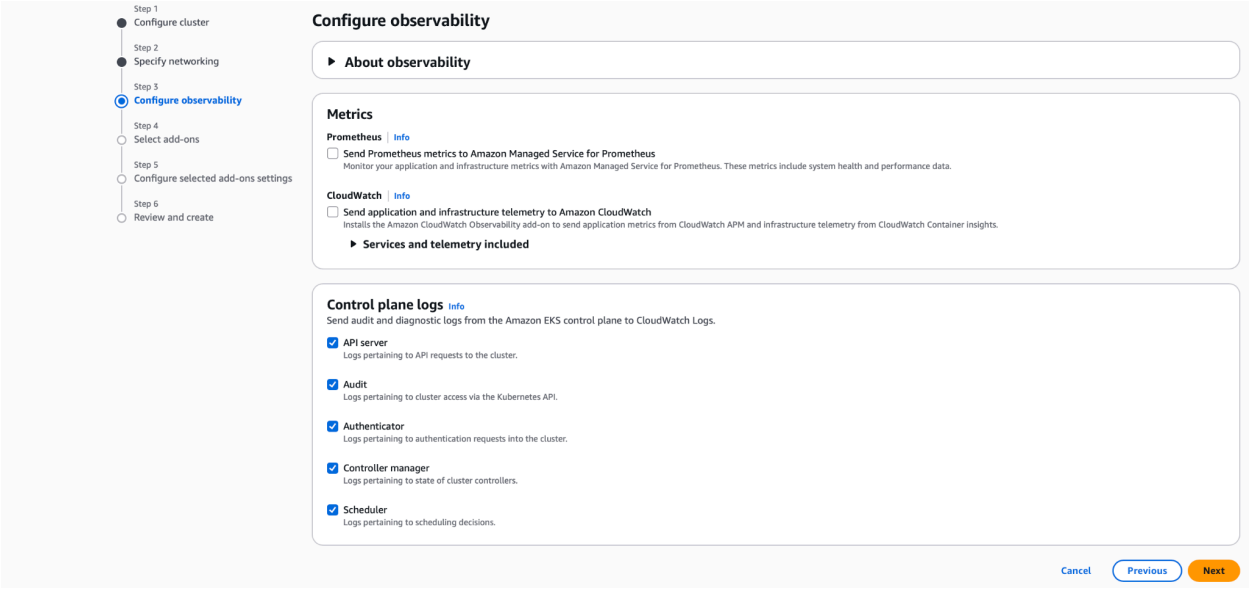
    d. Cluster IP address family: IPv4



9. Cluster endpoint access
    a. Public and private
    b. Click on **Next**

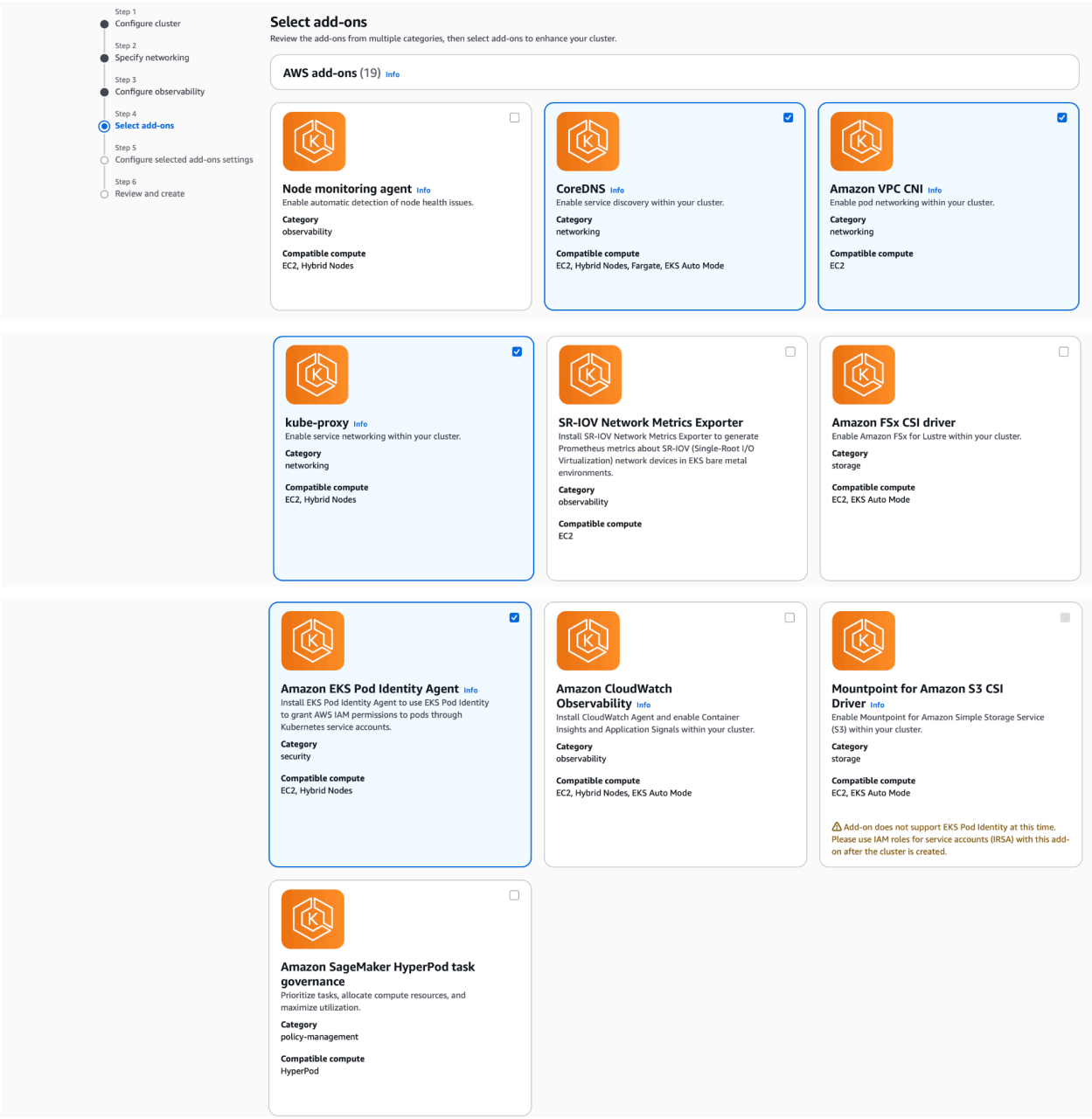


10. Configure observability page:
    a. Control plane logging: Enable all:
        i. API server
        ii. Audit
        iii. Authenticator
        iv. Controller manager
        v. Scheduler
    b. Prometheus metrics: Optional

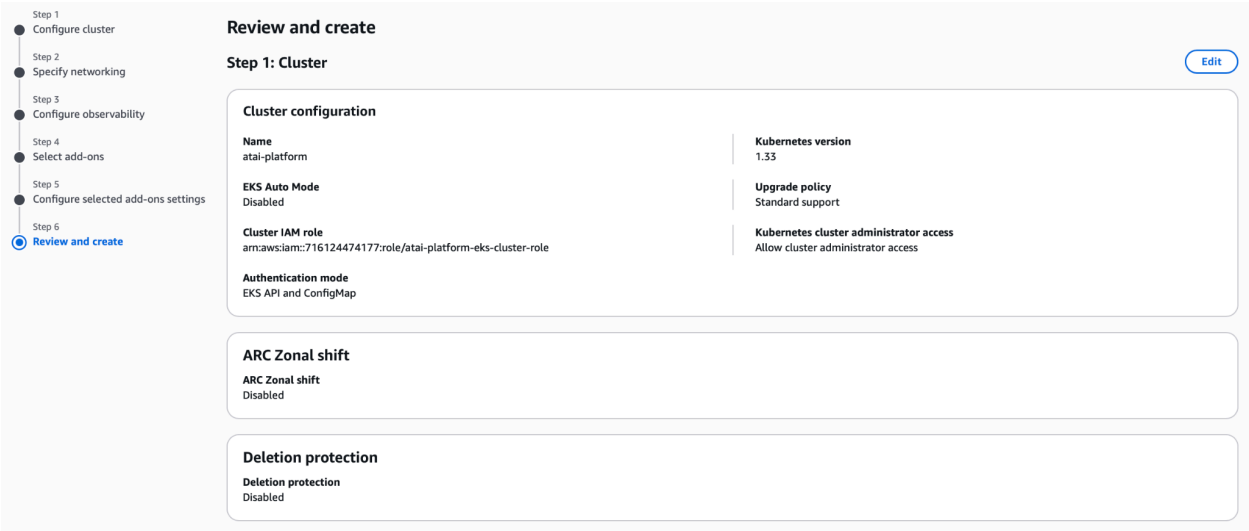


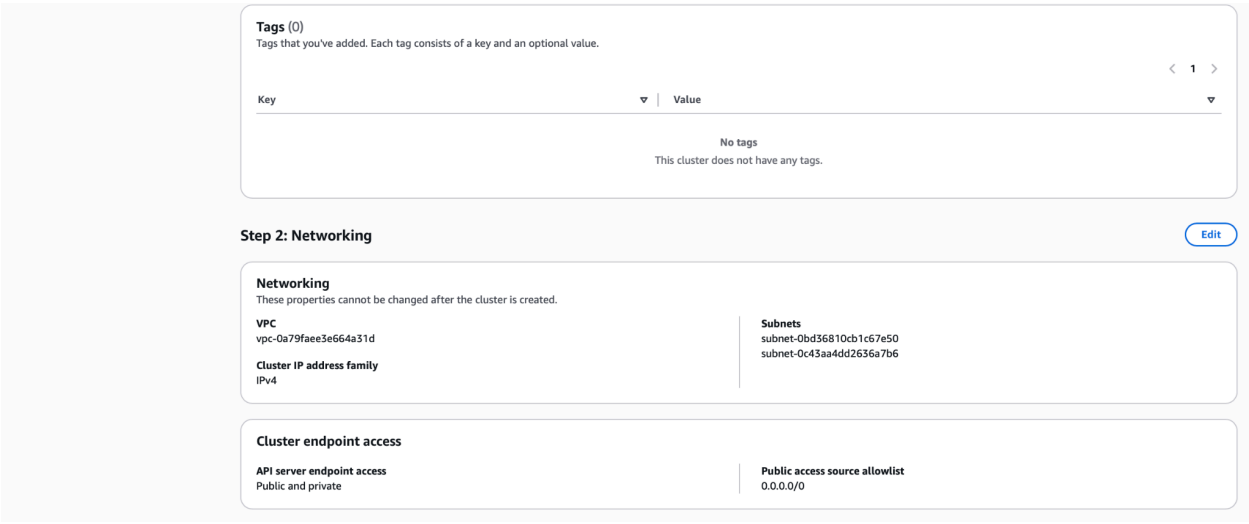
11. Select add-ons: Select these add-ons:
    a. Amazon VPC CNI (required)
    b. CoreDNS (required)
    c. kube-proxy (required)
    d. (Optional) EKS Pod Identity Agent (for Pod Identity)

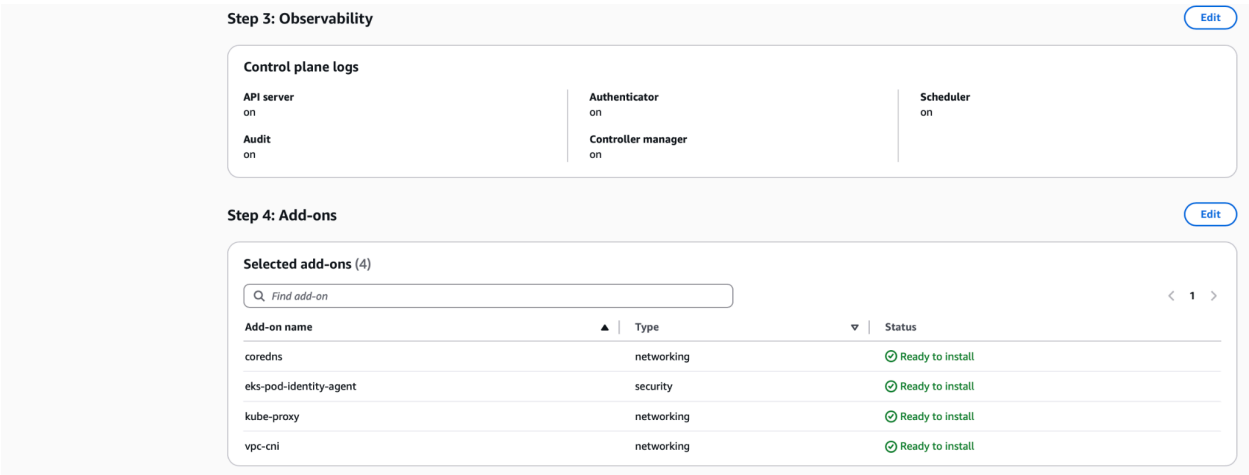
Click Next

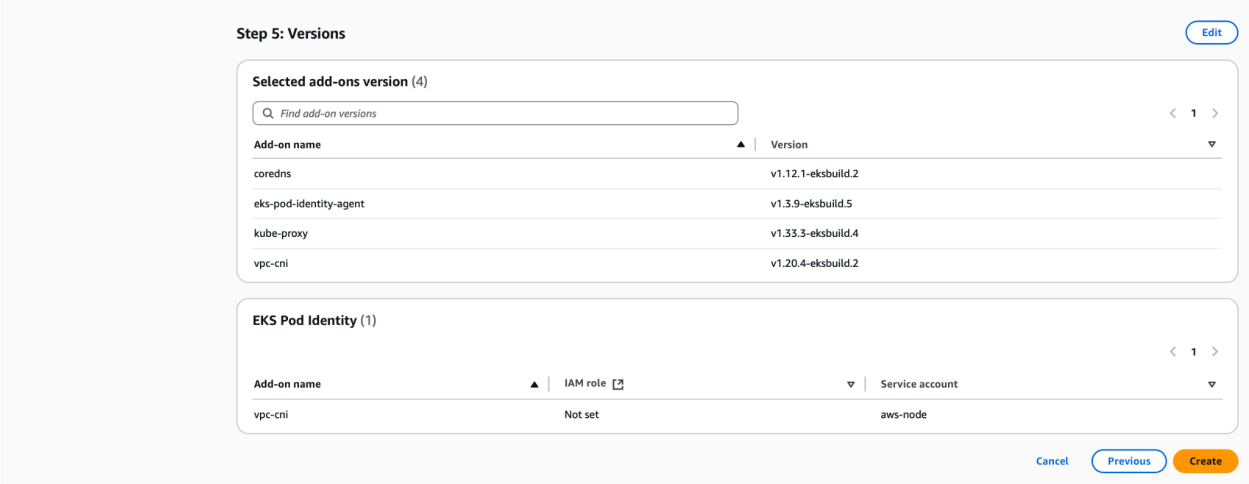


12. Configure selected add-ons settings
    a. VPC CNI: v1.20.4-eksbuild.2 or Default/Current version
    b. CoreDNS: v1.12.1-eksbuild.2 or Default/Current version
    c. kube-proxy: v1.33.3-eksbuild.4 or Default/Current version
    d. EKS Pod Identity Agent: v0.1.35 or Default/Current version
13. Review your cluster configuration and click on **Create**









Step 3: Update kubeconfig

1. Enable kubectl to communicate with your cluster by adding a new context to the kubectl config file.

```
None
aws eks update-kubeconfig --region region-code --name atai-platform
```

An example output is as follows.

```
None
Added new context arn:aws:eks:region-code:111122223333:cluster/atai-platform to
/home/username/.kube/config
```

2. Confirm communication with your cluster by running the following command.

```
None
kubectl get svc
```

An example output is as follows.

```
None
NAME         TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
kubernetes   ClusterIP   10.100.0.1   <none>        443/TCP   28h
```

Step 4: Get the default cluster security group

After the cluster is created, you must **retrieve the security group ID** assigned by AWS. This ID is required when creating Launch Templates for your Managed Node Groups, to ensure proper networking and access between the control plane and the worker nodes.

You can determine the ID of your cluster security group in the AWS Management Console under the cluster's Networking section. Or, you can do so by running the following AWS CLI command.

```
None
aws eks describe-cluster --name  atai-platform --query
cluster.resourcesVpcConfig.clusterSecurityGroupId
```

EKS managed node group configuration

Prerequisites

1. EKS cluster is ACTIVE
2. IAM role for node groups created (see Step 1 below)
3. Security group for nodes (see Step 2 below)
4. An existing VPC and subnets that meet Amazon EKS requirements
5. One private subnets in your VPC:
    a. Recommended naming:
        i. atai-platform-vpc-private-us-west-2a,
    b. Recommended CIDR blocks: /20 per subnet
        i. Example: 10.5.0.0/20 (subnet 1)
    c. Subnets must have routes to NAT Gateway or Internet Gateway for outbound internet access
6. The kubectl command line tool is required. The version can be the same as or up to one minor version earlier or later than the Kubernetes version of your cluster.
7. Version 2.12.3 or later or version 1.27.160 or later of the AWS Command Line Interface (AWS CLI) installed and configured on your device.
8. An IAM principal with permissions to create and describe an Amazon EKS cluster, and create Managed Node Groups.

Step 1:Creating the Amazon EKS node IAM role

1. Run the following command to create an IAM trust policy JSON file.

```
None
cat > node-role-trust-relationship.json << 'EOF'
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "sts:AssumeRole"
            ],
            "Principal": {
                "Service": [
                    "ec2.amazonaws.com"
                ]
            }
        }
    ]
}
EOF
```

2. Create the IAM role.

```
None
aws iam create-role \
  --role-name atai-platform-eks-node-role \
  --assume-role-policy-document file://"node-role-trust-relationship.json"
```

3. Attach two required IAM managed policies to the IAM role.

```
None
aws iam attach-role-policy \
  --policy-arn arn:aws:iam::aws:policy/AmazonEKSWorkerNodePolicy \
  --role-name atai-platform-eks-node-role

aws iam attach-role-policy \
  --policy-arn arn:aws:iam::aws:policy/AmazonEC2ContainerRegistryPullOnly \
  --role-name atai-platform-eks-node-role
```

```
aws iam attach-role-policy \
  --policy-arn arn:aws:iam::aws:policy/AmazonEC2ContainerRegistryReadOnly \
  --role-name atai-platform-eks-node-role
```

4. Attach one of the following IAM policies to the IAM role depending on which IP family you created your cluster with. In this manual, we created the cluster with IPv4, therefore run the following command:

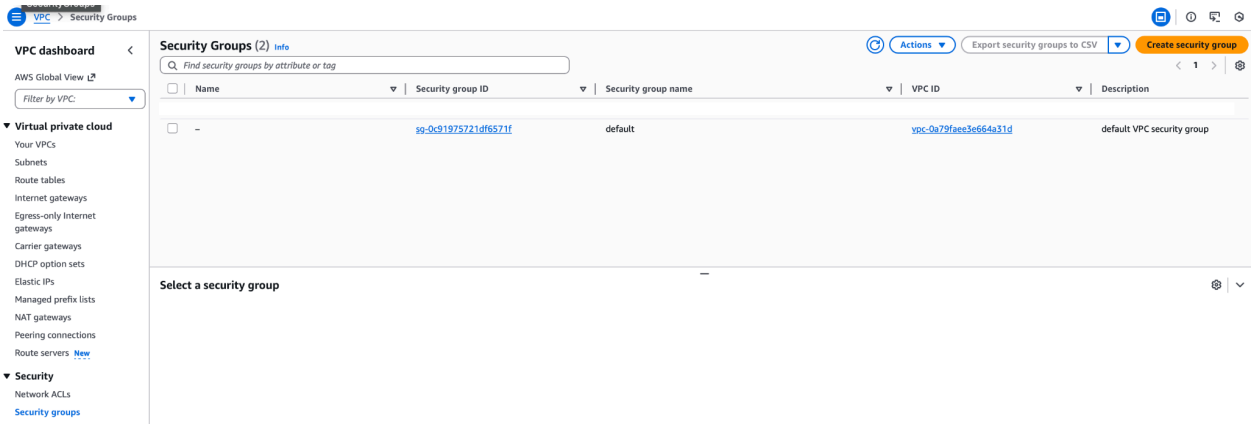
```
None
aws iam attach-role-policy \
  --policy-arn arn:aws:iam::aws:policy/AmazonEKS_CNI_Policy \
  --role-name atai-platform-eks-node-role
```

5. Attach the following IAM policy to the IAM role to allow SSH access to the nodes through AWS Session Manager (SSM) for debugging purposes.

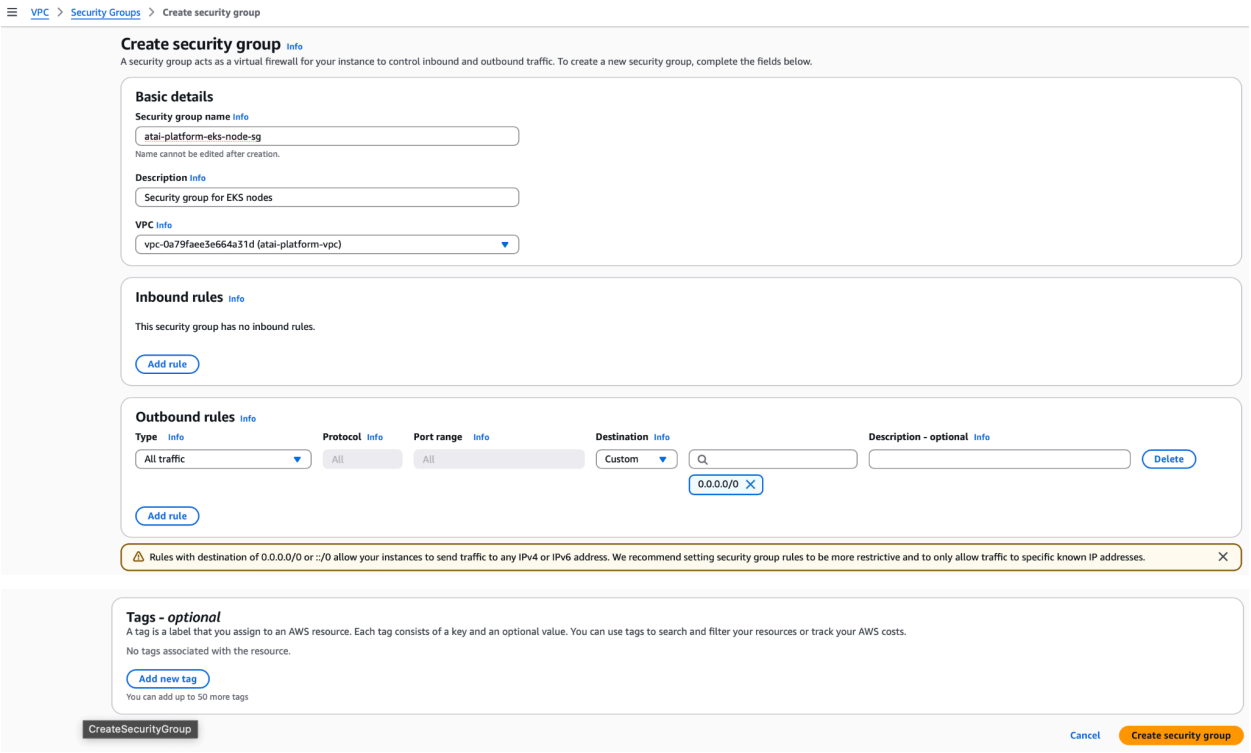
```
None
aws iam attach-role-policy \
  --policy-arn arn:aws:iam::aws:policy/AmazonSSMManagedInstanceCore \
  --role-name atai-platform-eks-node-role
```

Step 2: Creating Node Security Group

6. Go to VPC → Security Groups → Create security group



7. Name: atai-platform-eks-node-sg (or your name)
8. Description: Security group for EKS nodes
9. VPC: Select your VPC from section VPC configuration Step 1

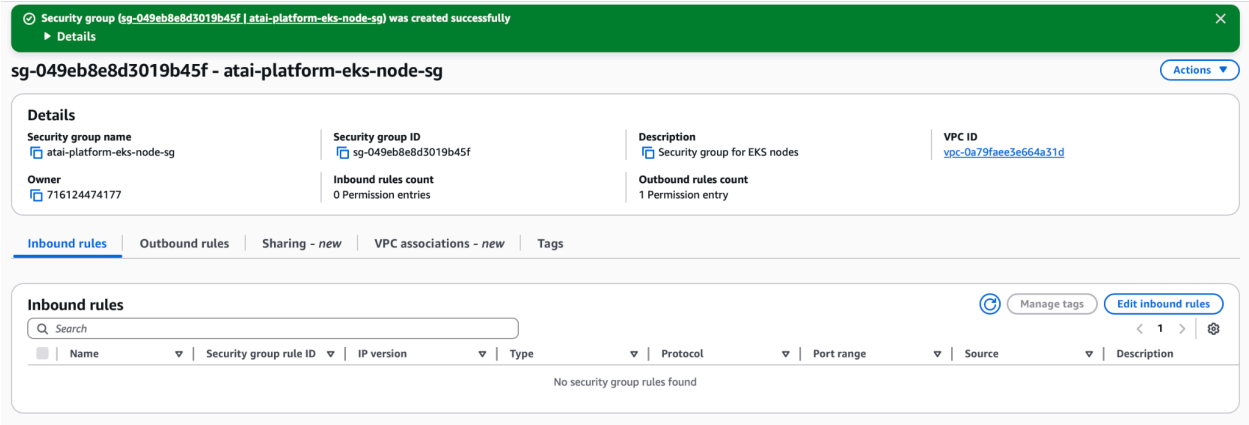


10. Click on **Create security group**

Step 2.1 Add self rules to the node security group

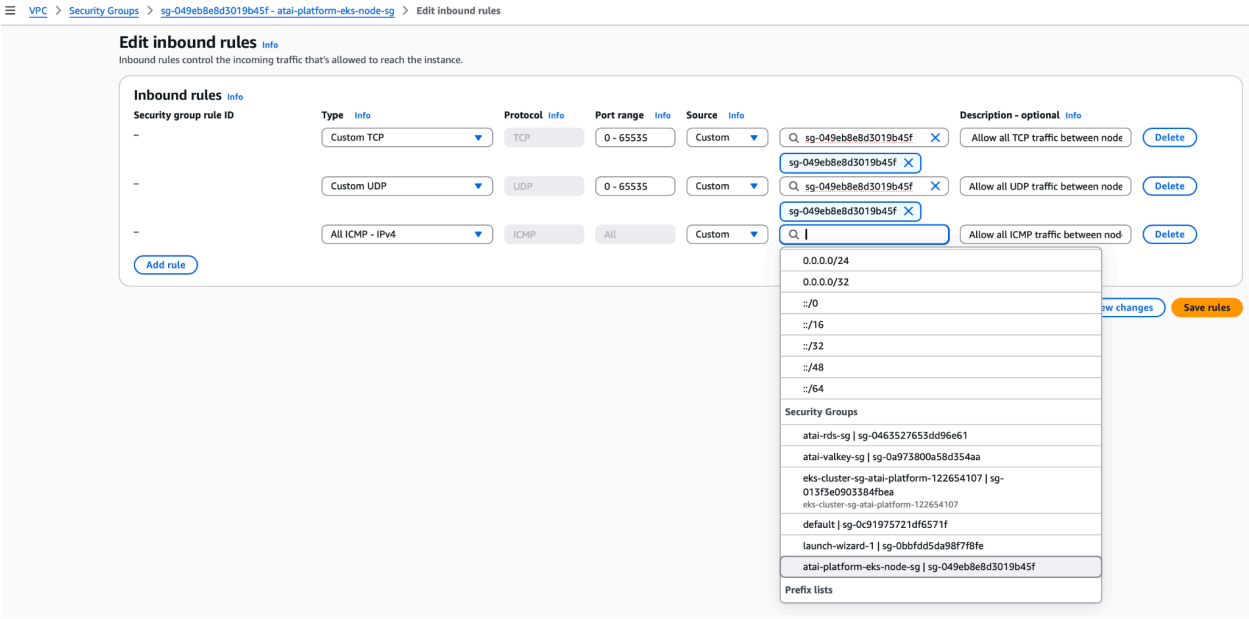
A "self rule" in a security group allows traffic from the same security group.

1. Select your security group configured in Step 2 and click on **Edit inbound rules**



2. Inbound rules: Add rule:
    a. Rule 1: All TCP
        i. Type: All TCP
        ii. Source: Custom
            1. In the Source field, select Custom (not CIDR blocks or IP addresses).
            2. Click on the blank textbox next to Source
            3. Navigate to Security groups.
            4. Select the same security group you're currently configuring.
        iii. Description: Allow all TCP traffic between nodes
    b. Rule 2: All UDP
        i. Type: All UDP
        ii. Source: Custom
            1. In the Source field, select Custom (not CIDR blocks or IP addresses).
            2. Click on the blank textbox next to Source
            3. Navigate to Security groups.
            4. Select the same security group you're currently configuring.
        iii. Description: Allow all UDP traffic between nodes
    c. Rule 3: ICMP
        i. Type: All ICMP - IPv4
        ii. Port: None
        iii. Source: Custom
            1. In the Source field, select Custom (not CIDR blocks or IP addresses).
            2. Click on the blank textbox next to Source
            3. Navigate to Security groups.
            4. Select the same security group you're currently configuring.
        iv. Description: Allow all ICMP traffic between nodes

3. Click on **Save rules**



Note: Creating Self-Referencing Security Group Rules

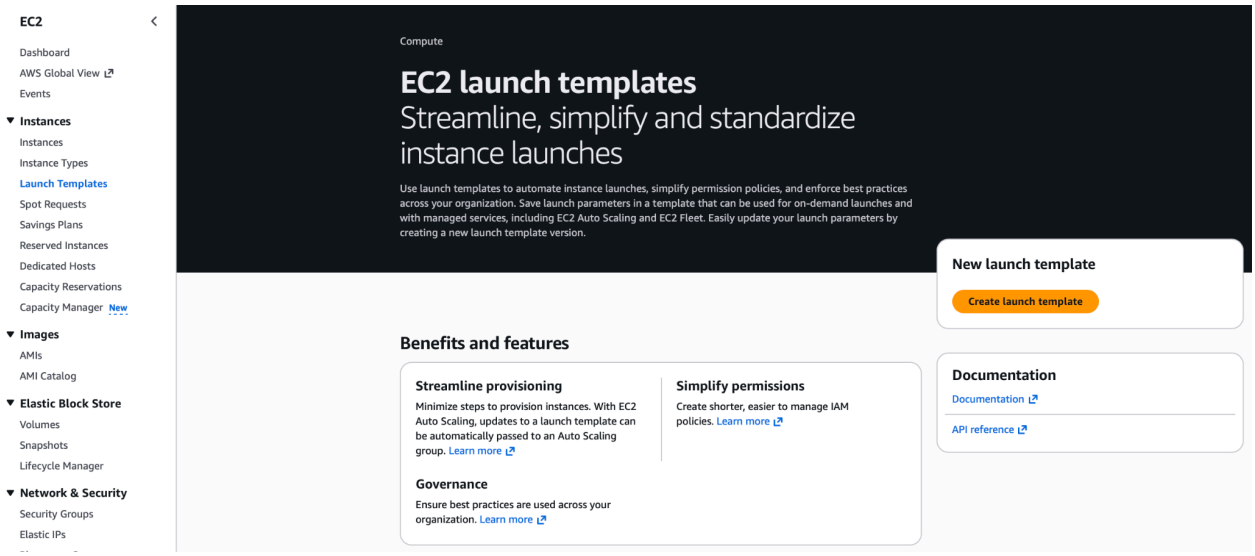
When adding ingress rules that allow traffic from the same security group (self-rule):

1. In the Source field, select Custom (not CIDR blocks or IP addresses).
2. Open the dropdown menu.
3. Navigate to Security groups.
4. Select the same security group you're currently configuring.

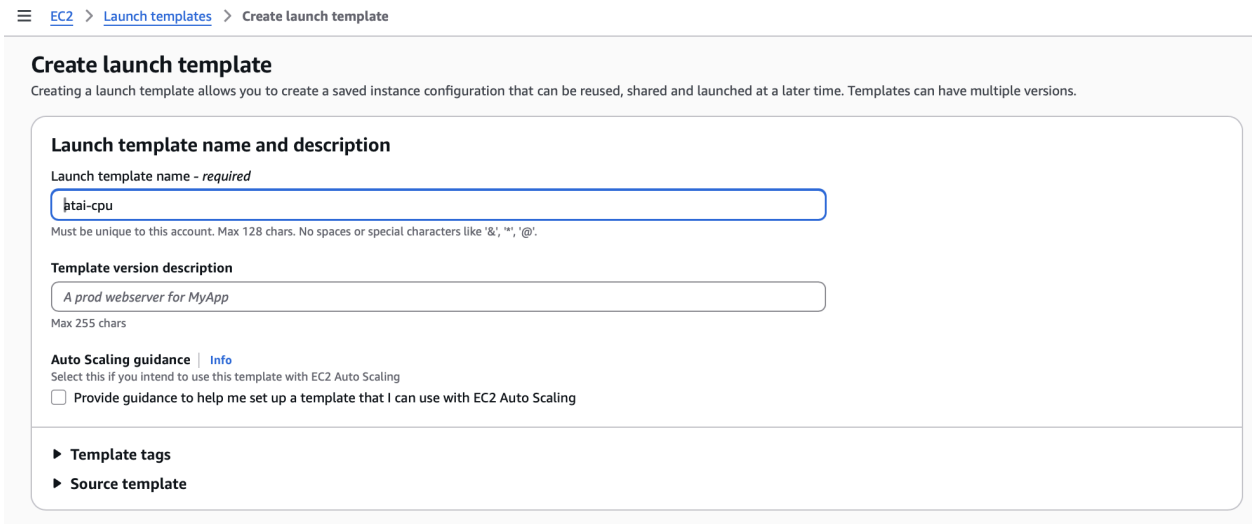
Step 3: Launch templates

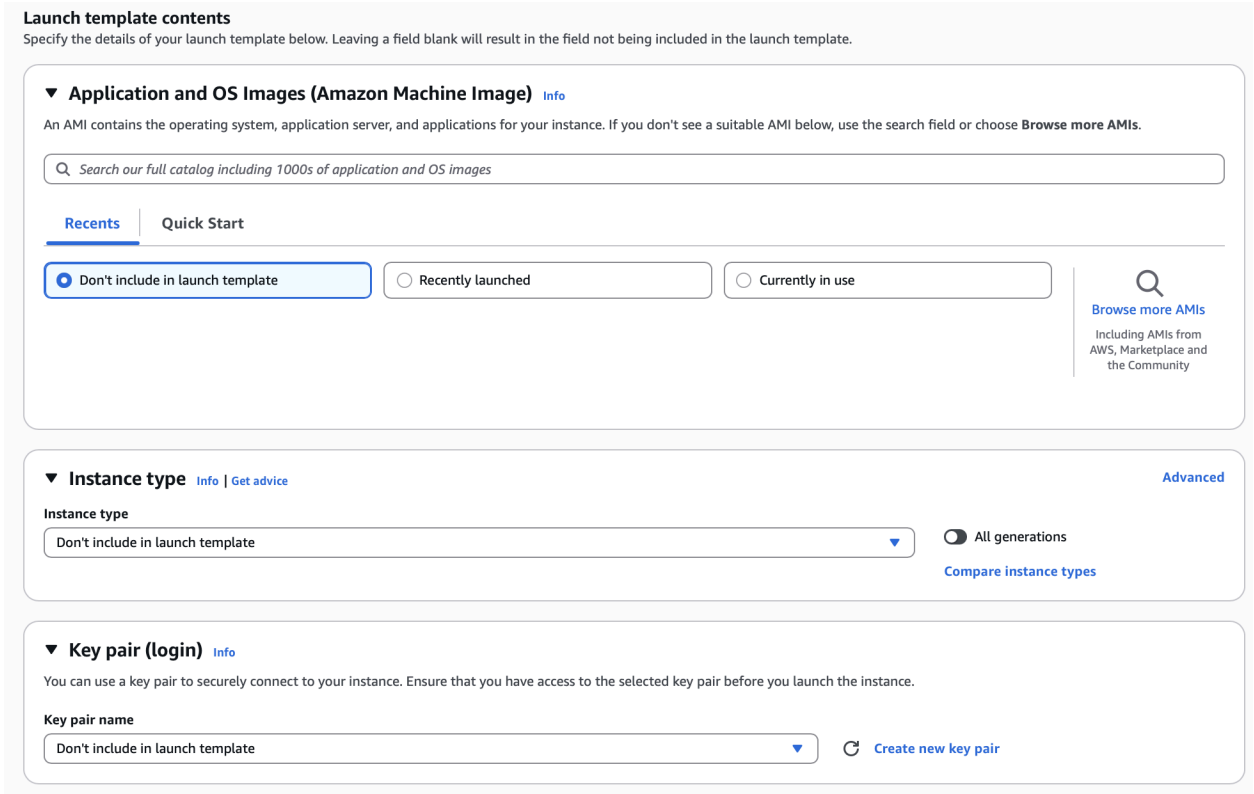
Step 3.1 CPU Node Group

1. Go to EC2 → Launch template → **Create Launch Template**

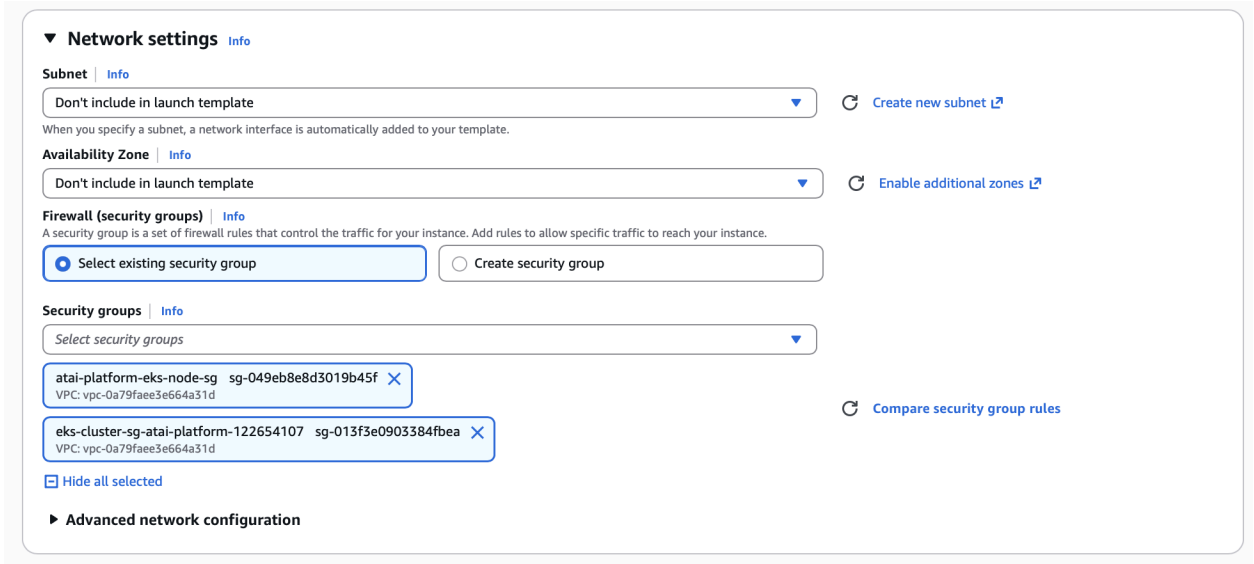


2. Name: atai-cpu

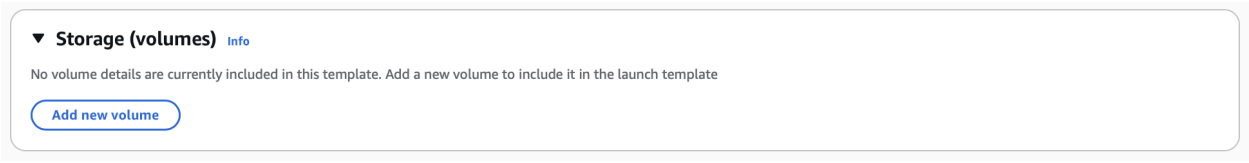




3. Network settings
    a. Select the security group created in the Step 2
    b. Select the default cluster security group that you get from the section **EKS cluster configuration - Step 4: Get the default cluster security group.**

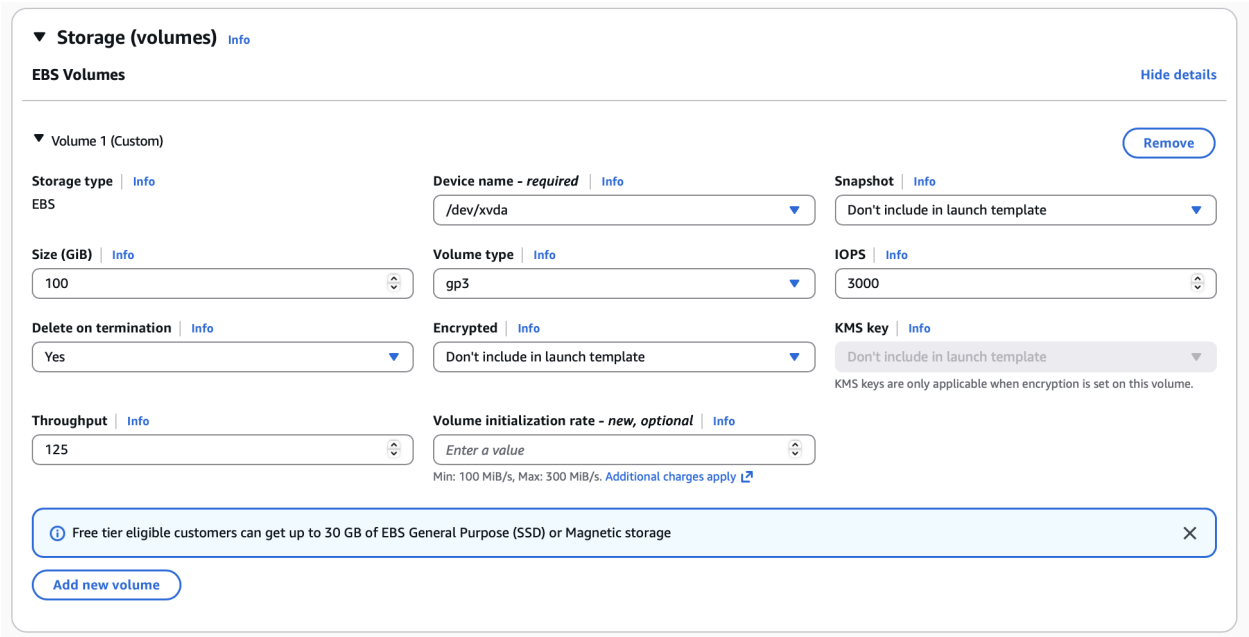


4. Storage (Volumes) - Click on **Add new volume**

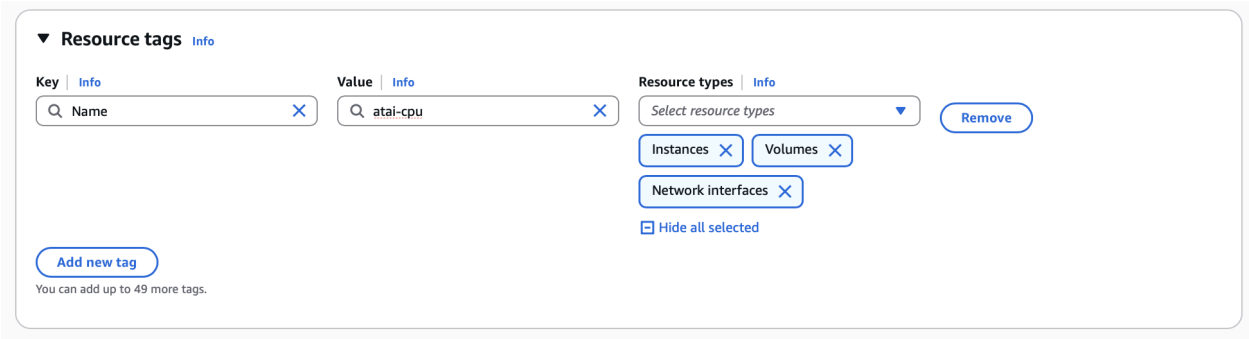


   a. Volume 1 configuration
      i. Size: 100
      ii. Volume type: gp3
      iii. Device name:
         1. Open the dropdown menu and click on **Specify a custom value**
         2. Value: /dev/xvda

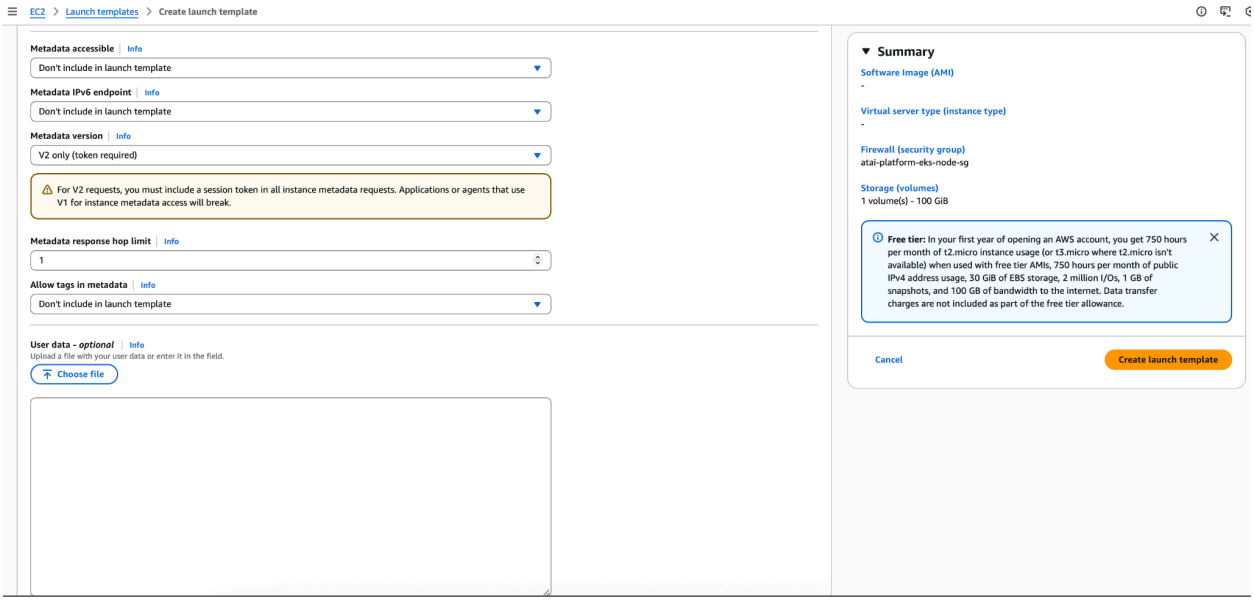
      iv. IOPS: 3000
      v. Throughput: 125



5. Resource tags
    a. Key: Name
    b. Value: atai-cpu
    c. Resource types
        i. Instances
        ii. Volumes
        iii. Network Interfaces



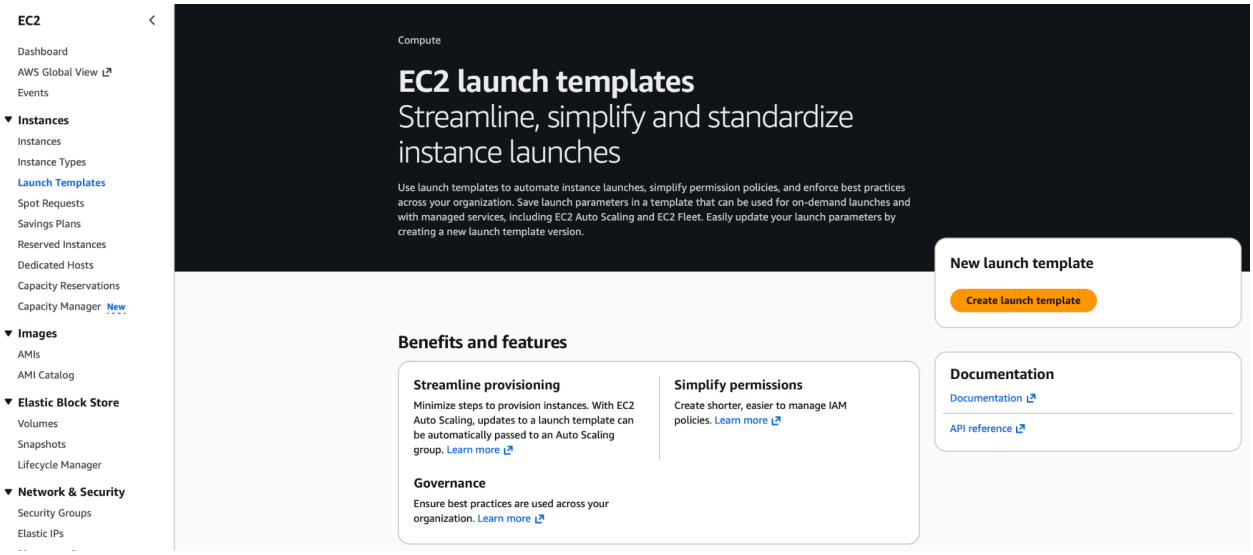
6. Advanced details
    a. Metadata version: V2
    b. Metadata response hop limit: 1



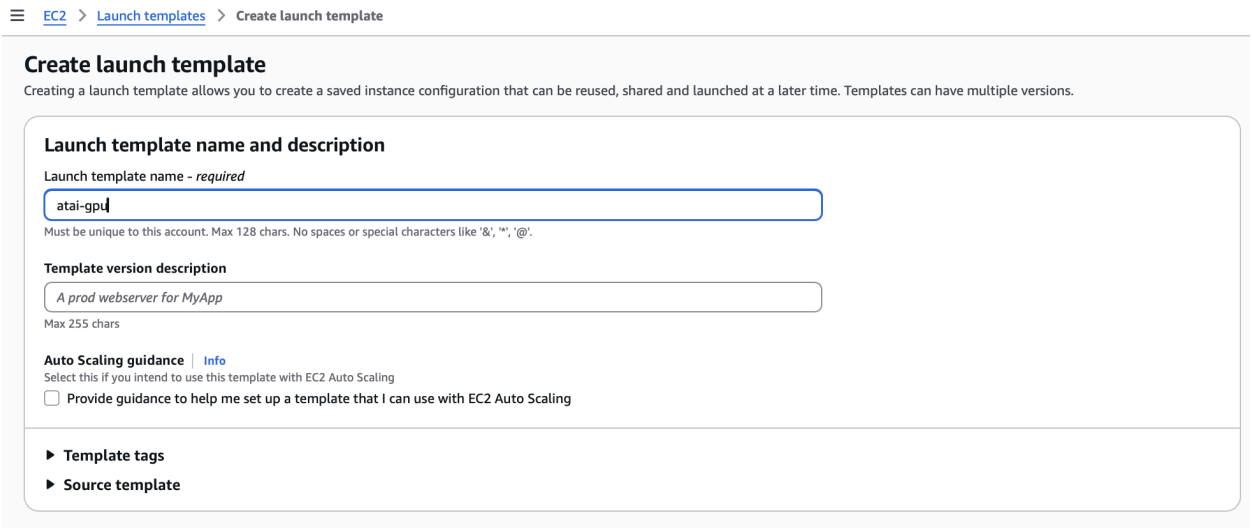
7. Click on **Create Launch Template**

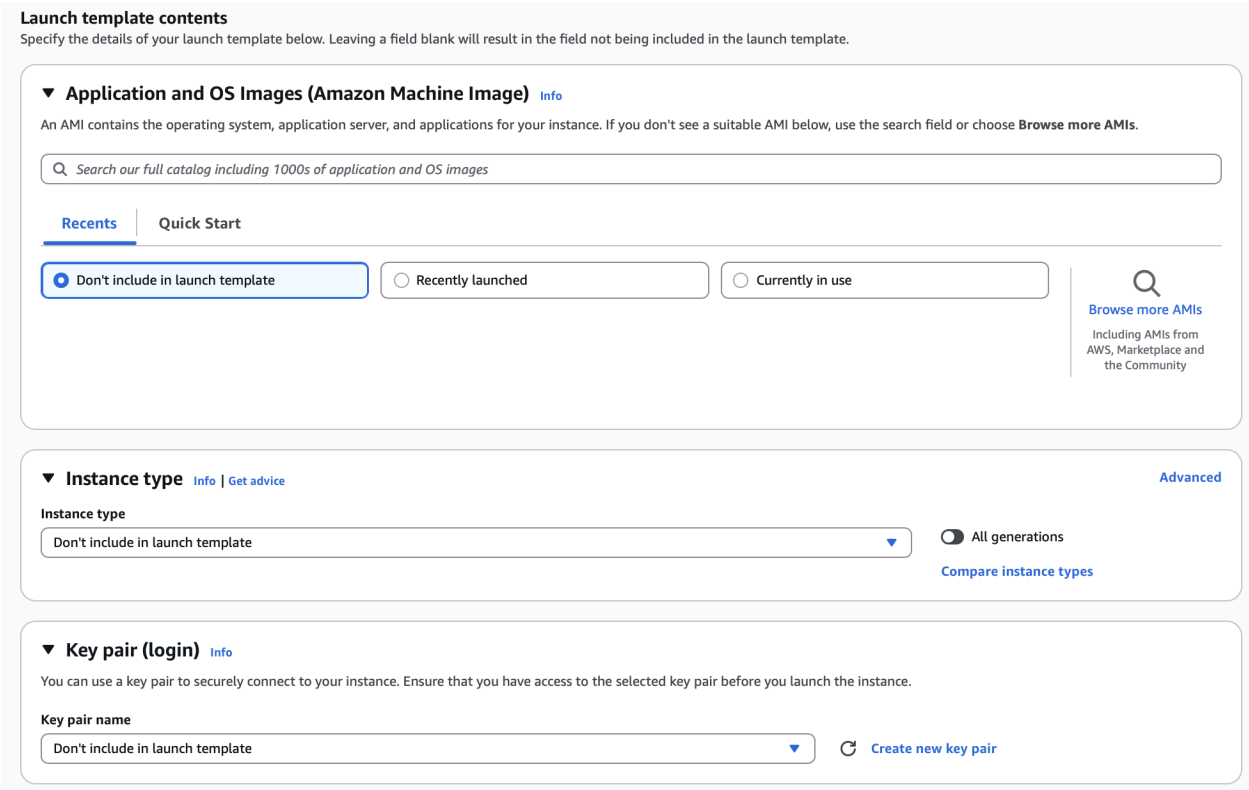
Step 3.2 GPU Node Group

8. Go to EC2 → Launch template → **Create Launch Template**

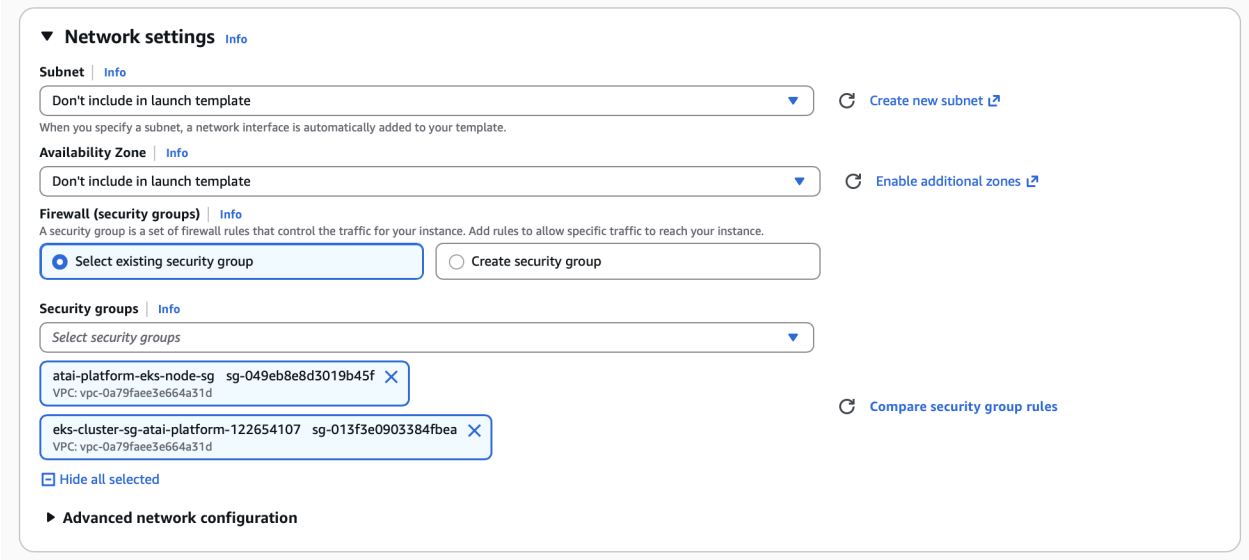


9. Name: atai-gpu

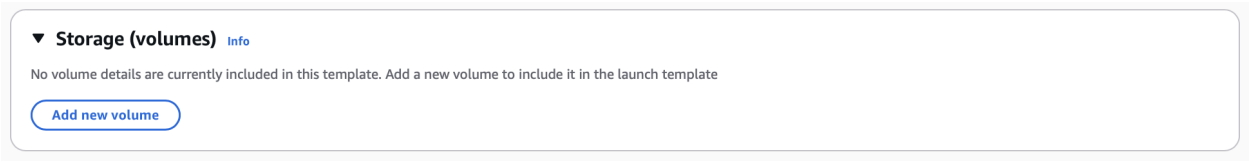




10. Network settings
    a. Select the security group created in the Step 2
    b. Select the default cluster security group that you get from the section EKS cluster configuration Step 4: Get the default cluster security group.

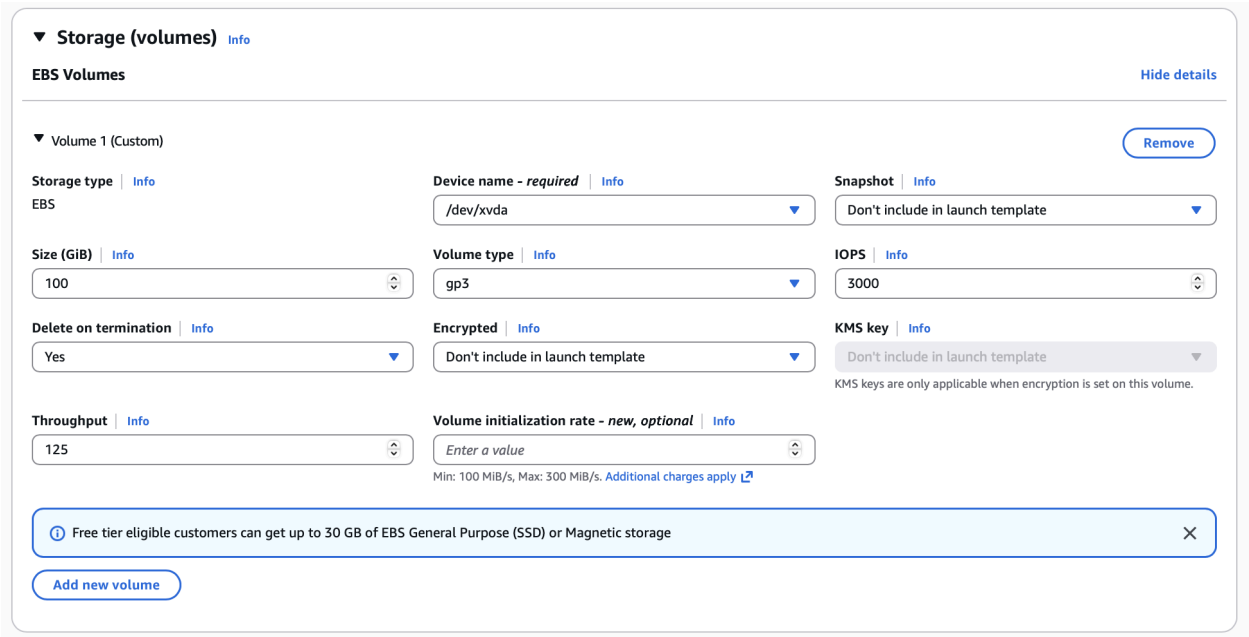


11. Storage (Volumes) - Click on **Add new volume**

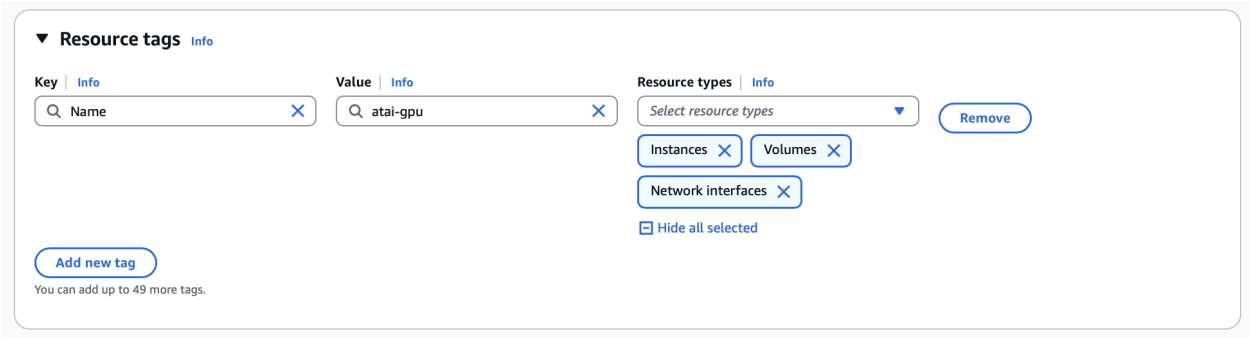


a. Volume 1 configuration
   i. Size: 100
   ii. Volume type: gp3
   iii. Device name:
      1. Open the dropdown menu and click on **Specify a custom value**
      2. Value: /dev/xvda

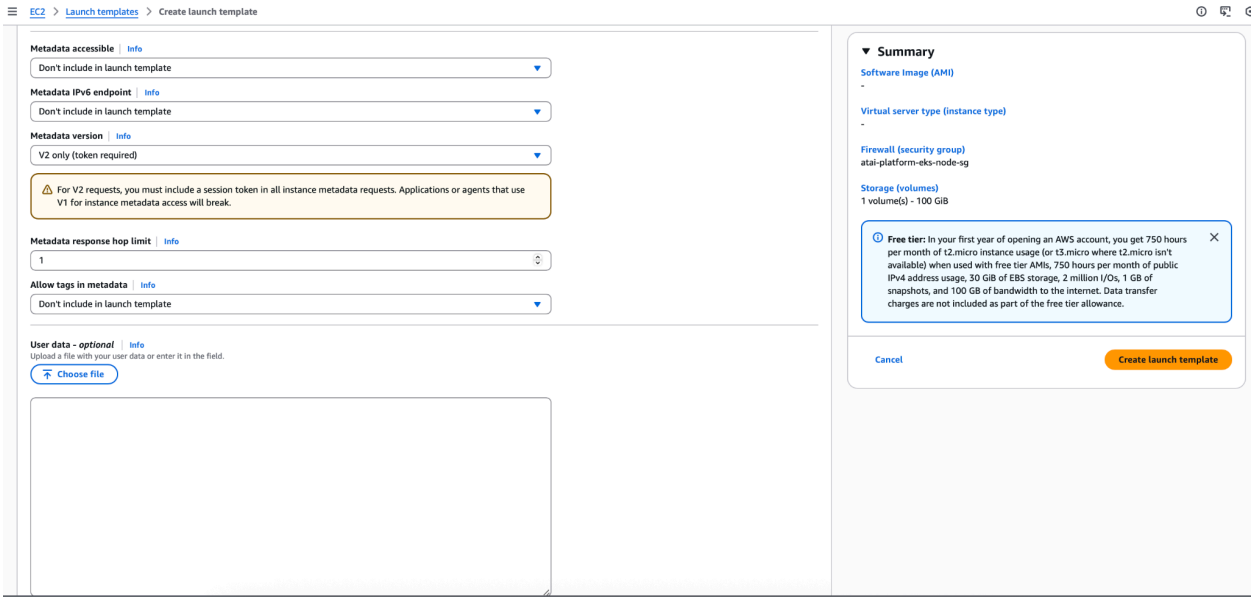
   iv. IOPS: 3000
   v. Throughput: 125



12. Resource tags
    a. Key: Name
    b. Value: atai-gpu
    c. Resource types
        i. Instances
        ii. Volumes
        iii. Network Interfaces



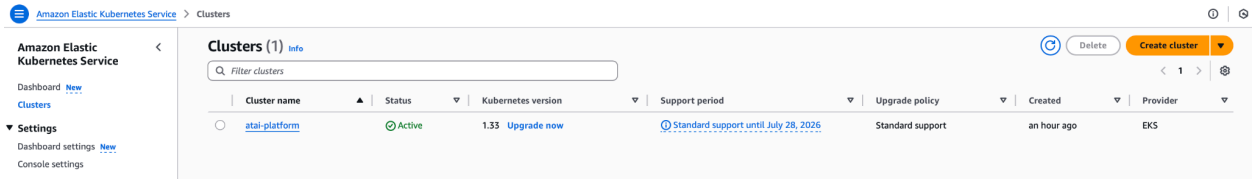
13. Advanced details
    a. Metadata version: V2
    b. Metadata response hop limit: 1



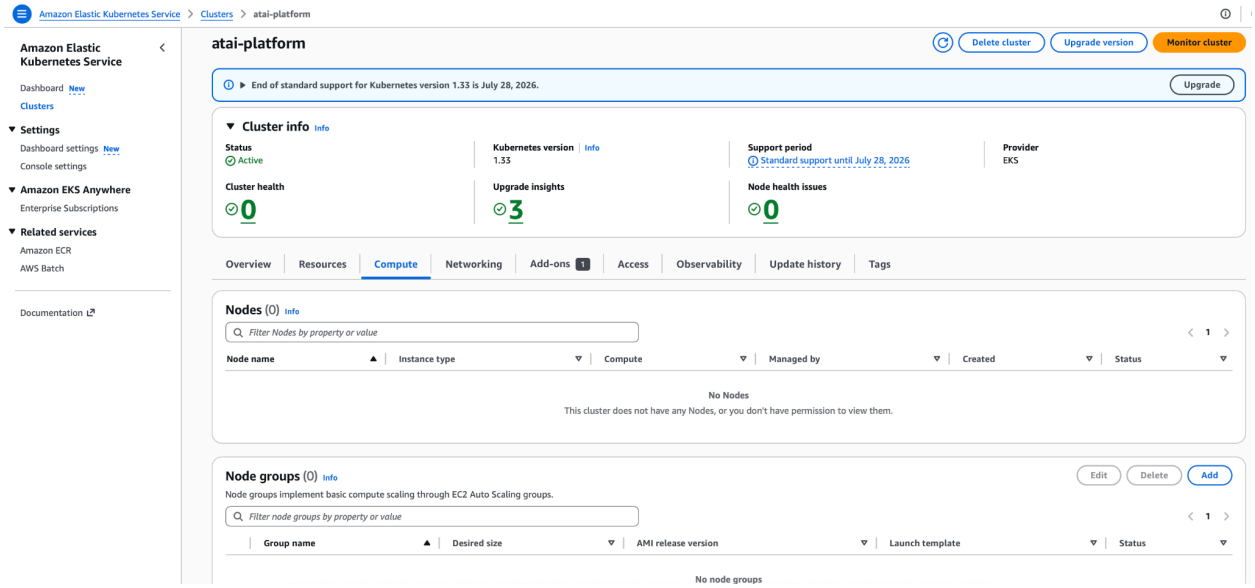
14. Click on **Create Launch Template**

Step 4: Create CPU Node Group

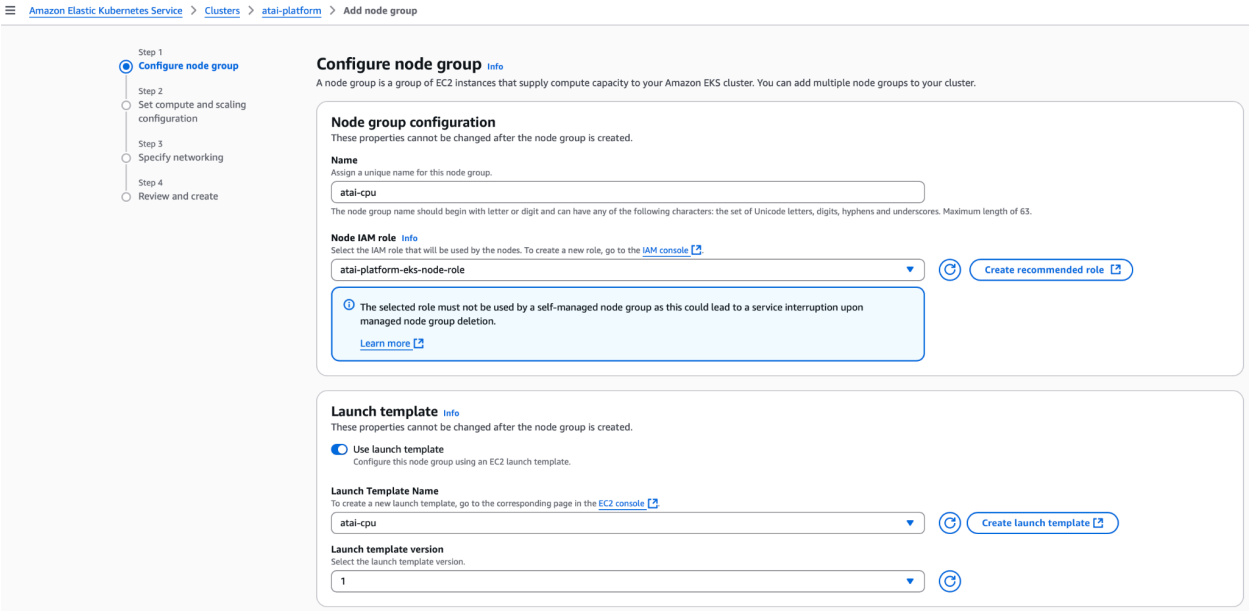
1. EKS Console → Your cluster



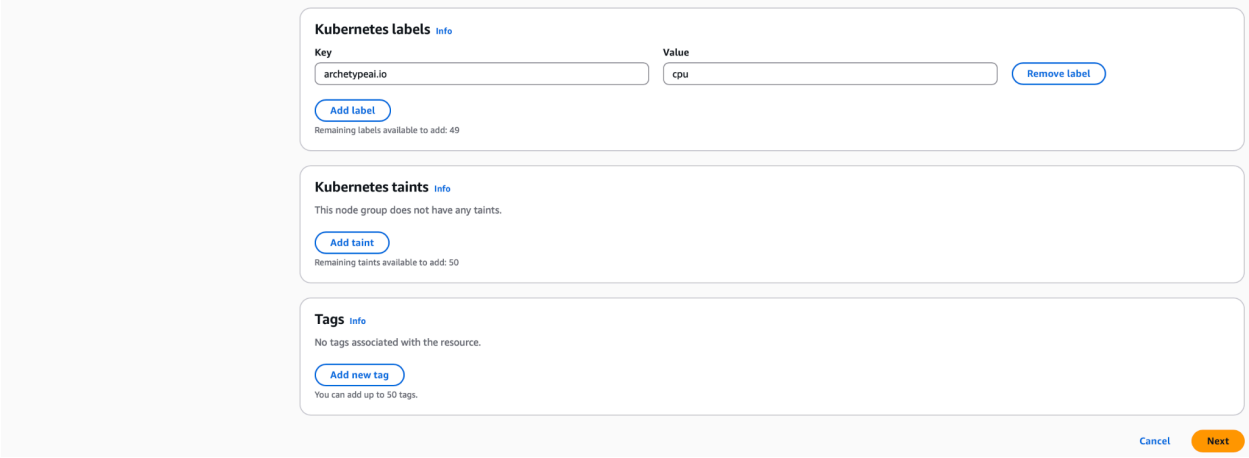
2. Select Compute tab → Node groups section → **Add node group**



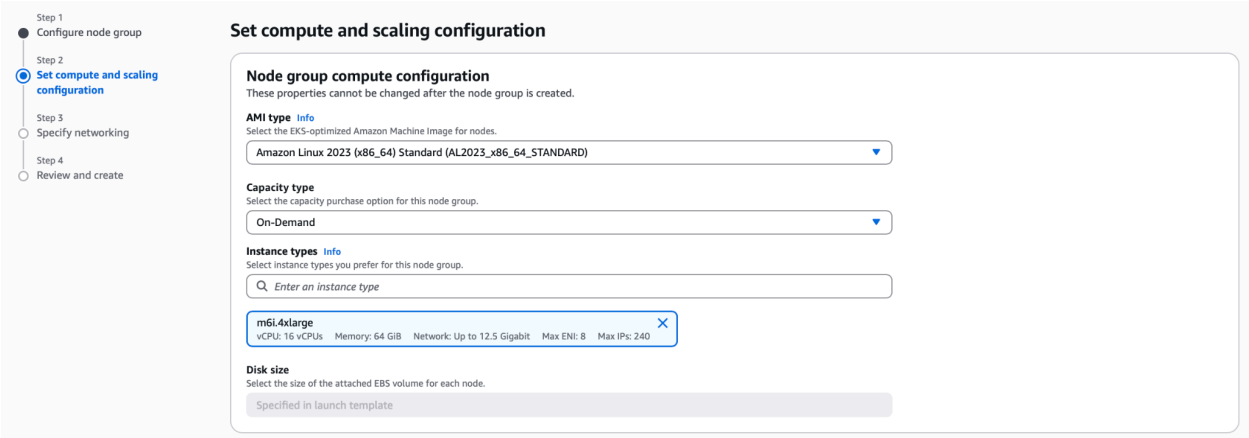
3. Configure node group:
    a. Node group name: atai-cpu
    b. Node IAM role: Select the role from Step 1
    c. Launch template: Select the Launch template created in the Step 3.1



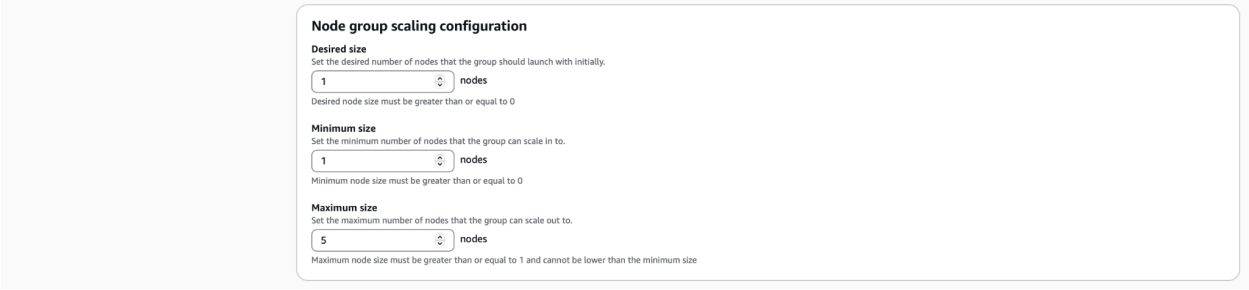
4. Configure Kubernetes labels and taints:
    a. Labels: Add label:
        i. Key: archetypeai.io
        ii. Value: cpu
    b. Taints: None (leave empty)



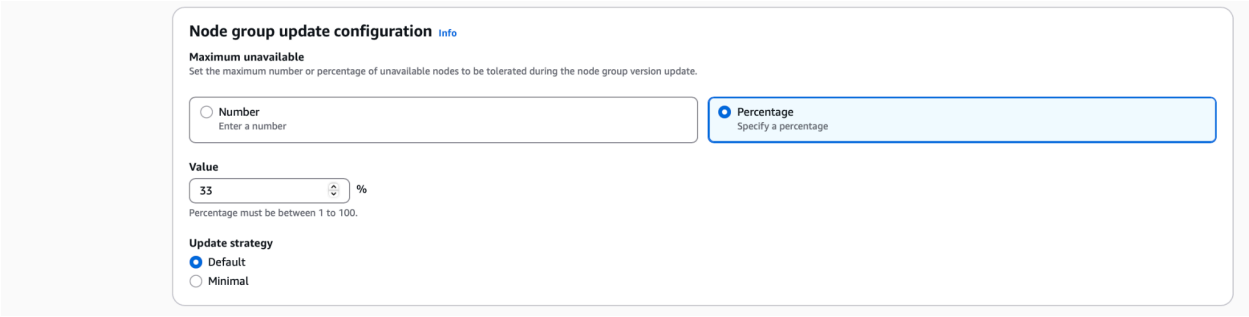
5. Node group compute configuration
    a. AMI type: Amazon Linux 2023 (AL2023_x86_64_STANDARD)
    b. Capacity type: On-Demand
    c. Instance types: m6i.4xlarge
    d. Disk size: Specified in the Launch template



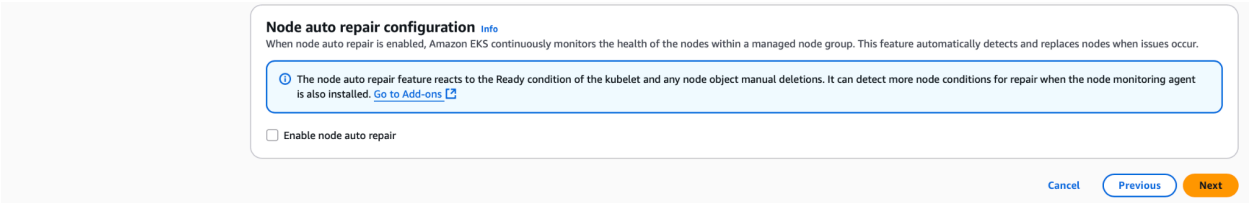
6. Node group scaling configuration:
    a. Desired size: 1 (or 2 for production)
    b. Minimum size: 1
    c. Maximum size: 5



7. Node group update configuration:
    a. Maximum unavailable: Percentage
    b. Value: 33%
    c. Update strategy: Default

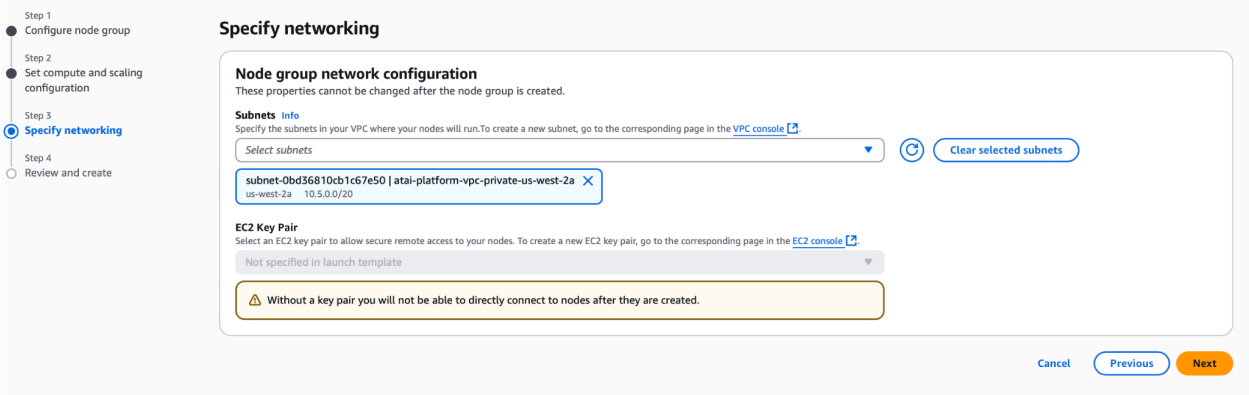


8. Node group auto repair configuration: Disable

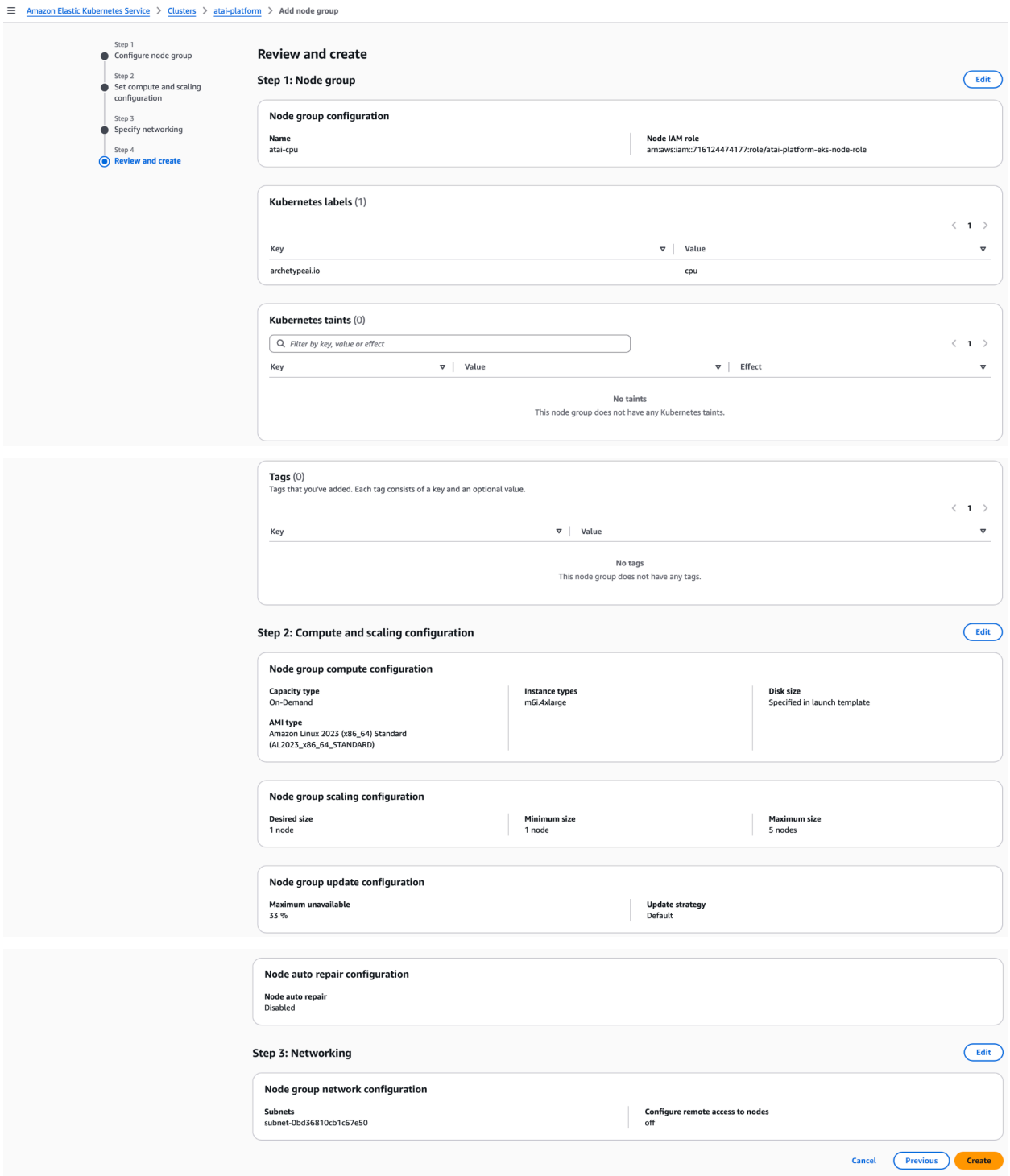


9. Node group network configuration
    a. Subnets: Select your private subnets:
        i. atai-platform-vpc-private-us-west-2a (10.5.0.0/20)

    It's important to select the private subnet in the same AZs as your Valkey clusters and RDS PostgreSQL cluster.

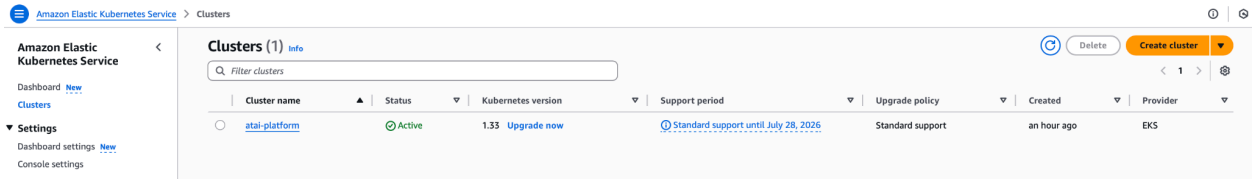


10. Review and click on **Create**
    Wait for node group to be ACTIVE (5-10 minutes)

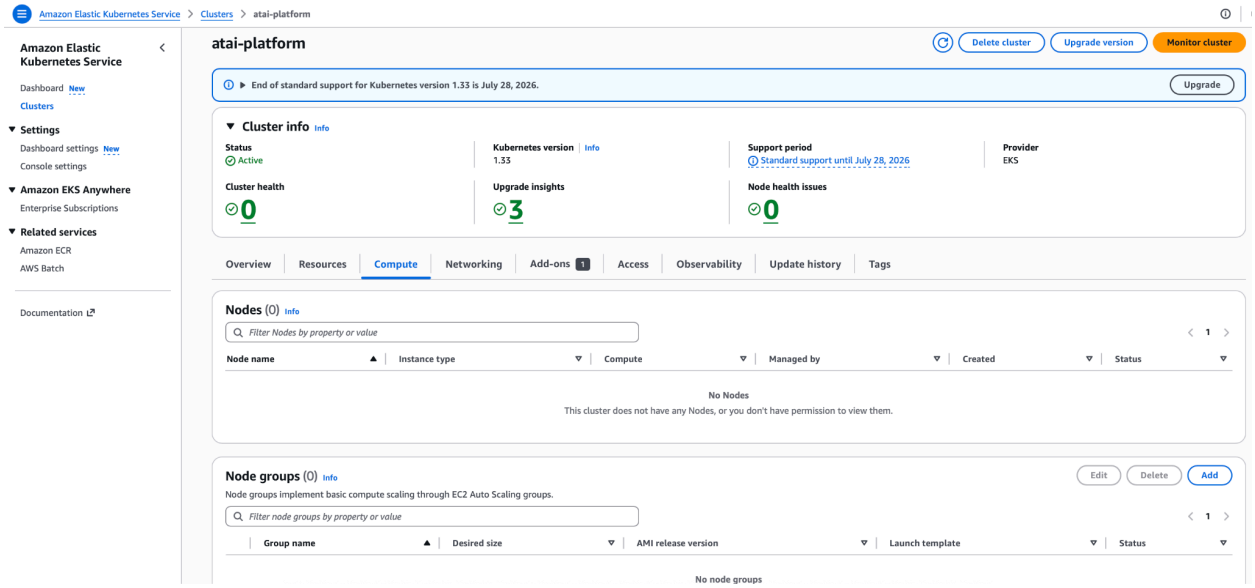


Step 5: Create GPU Node Group

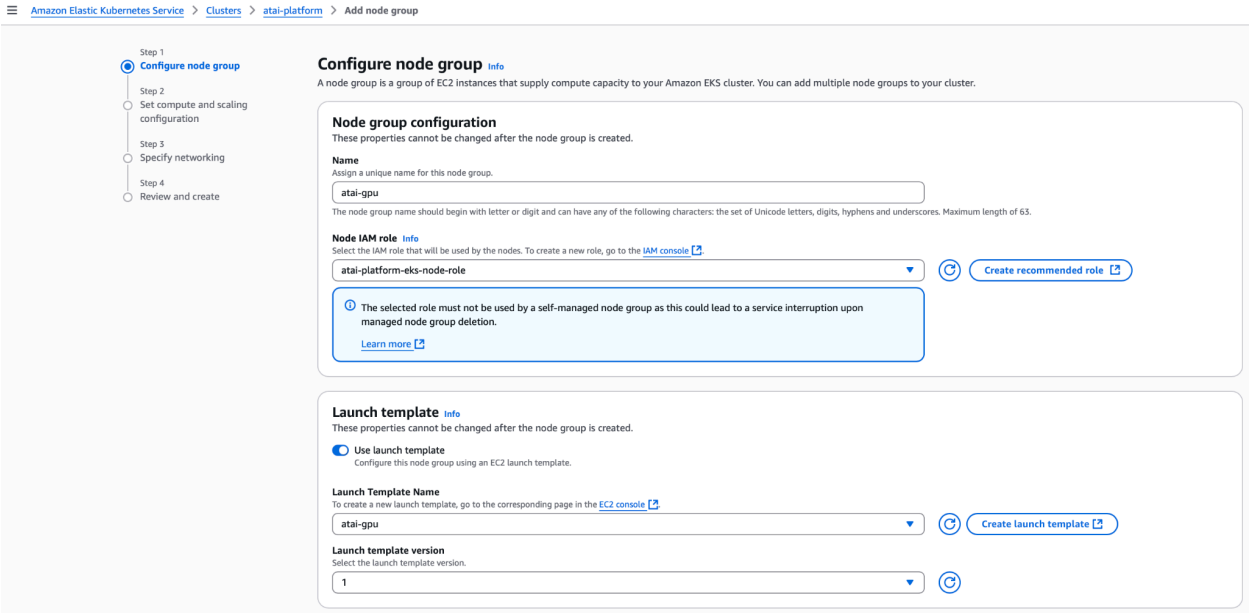
1. EKS Console → Your cluster



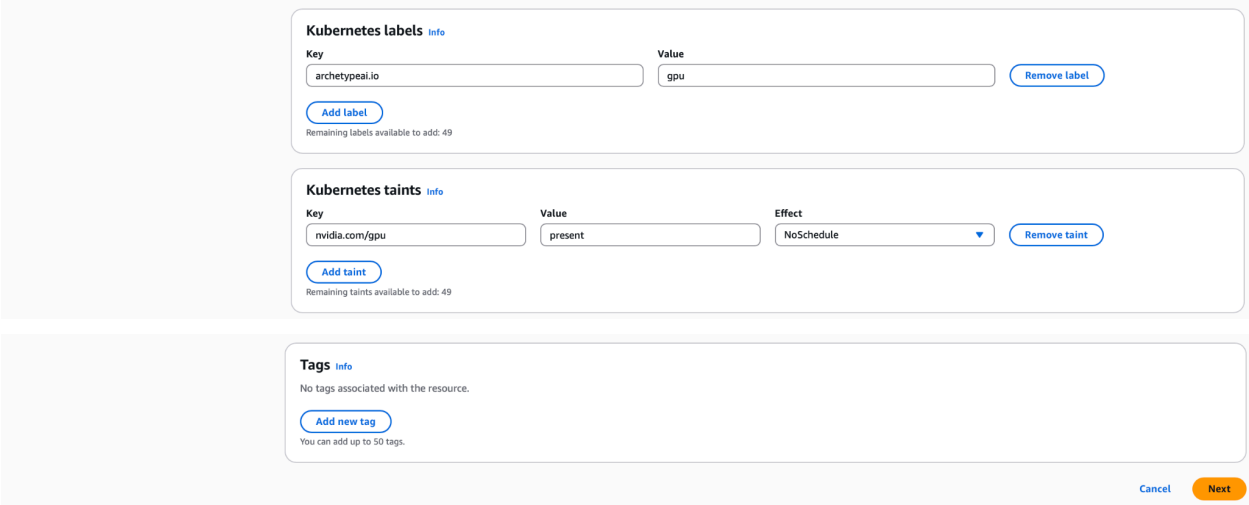
2. Select Compute tab → Node groups section → **Add node group**



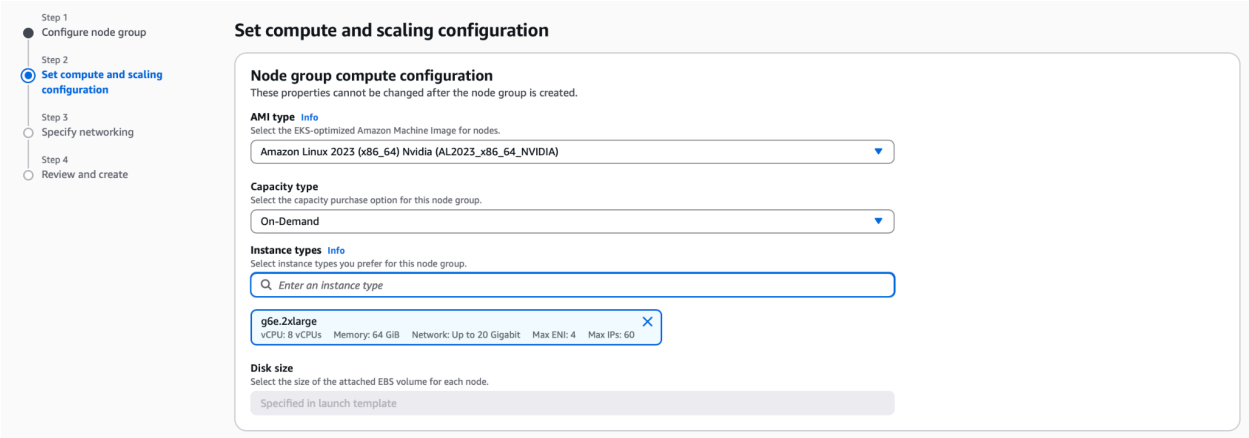
3. Configure node group:
    a. Node group name: atai-gpu
    b. Node IAM role: Select the role from Step 1
    c. Launch template: Select the Launch template created in the Step 3.2



4. Configure Kubernetes labels and taints:
    a. Labels: Add label:
        i. Key: archetypeai.io
        ii. Value: gpu
    b. Taints:
        i. Key: nvidia.com/gpu
        ii. Value: present
        iii. Effect: NoSchedule

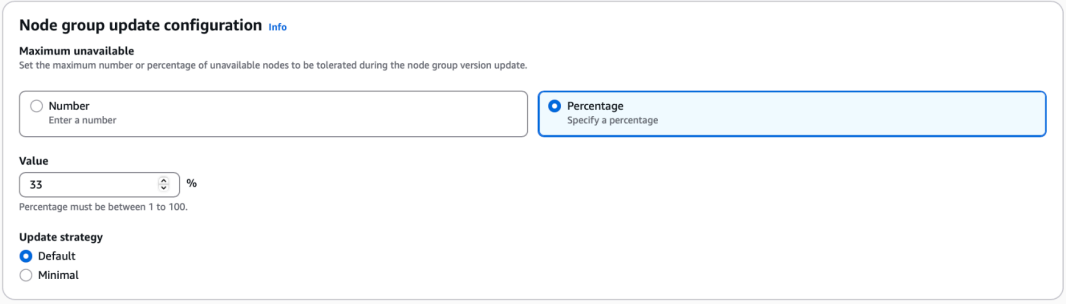


5. Node group compute configuration
    a. AMI type: Amazon Linux 2023 (x86_64) Nvidia (AL2023_x86_64_NVIDIA)
    b. Capacity type: On-Demand
    c. Instance types: g6e.2xlarge
    d. Disk size: Specified in the Launch template

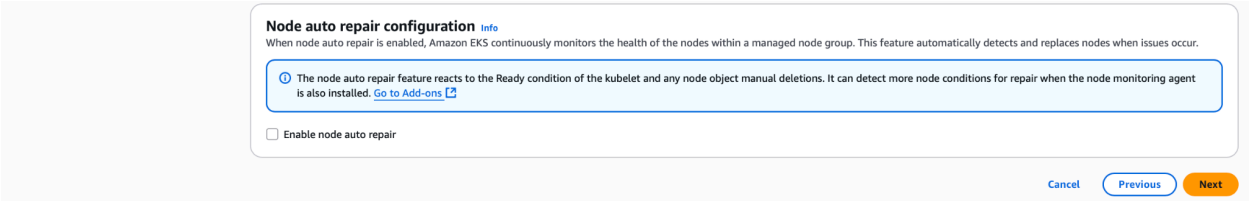


6. Node group scaling configuration:
    a. Desired size: 4
    b. Minimum size: 4
    c. Maximum size: 10

7. Node group update configuration:
    a. Maximum unavailable: Percentage
    b. Value: 33%
    c. Update strategy: Default

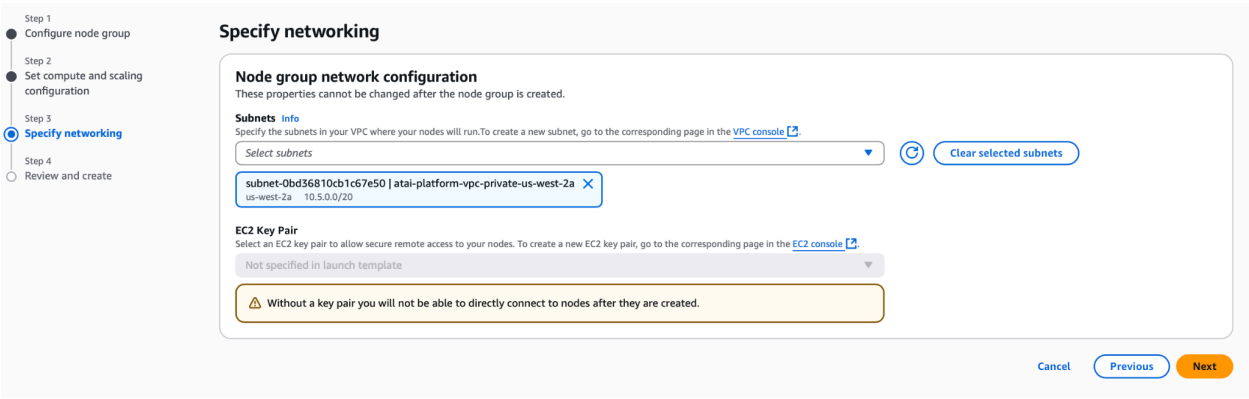


8. Node group auto repair configuration: Disable

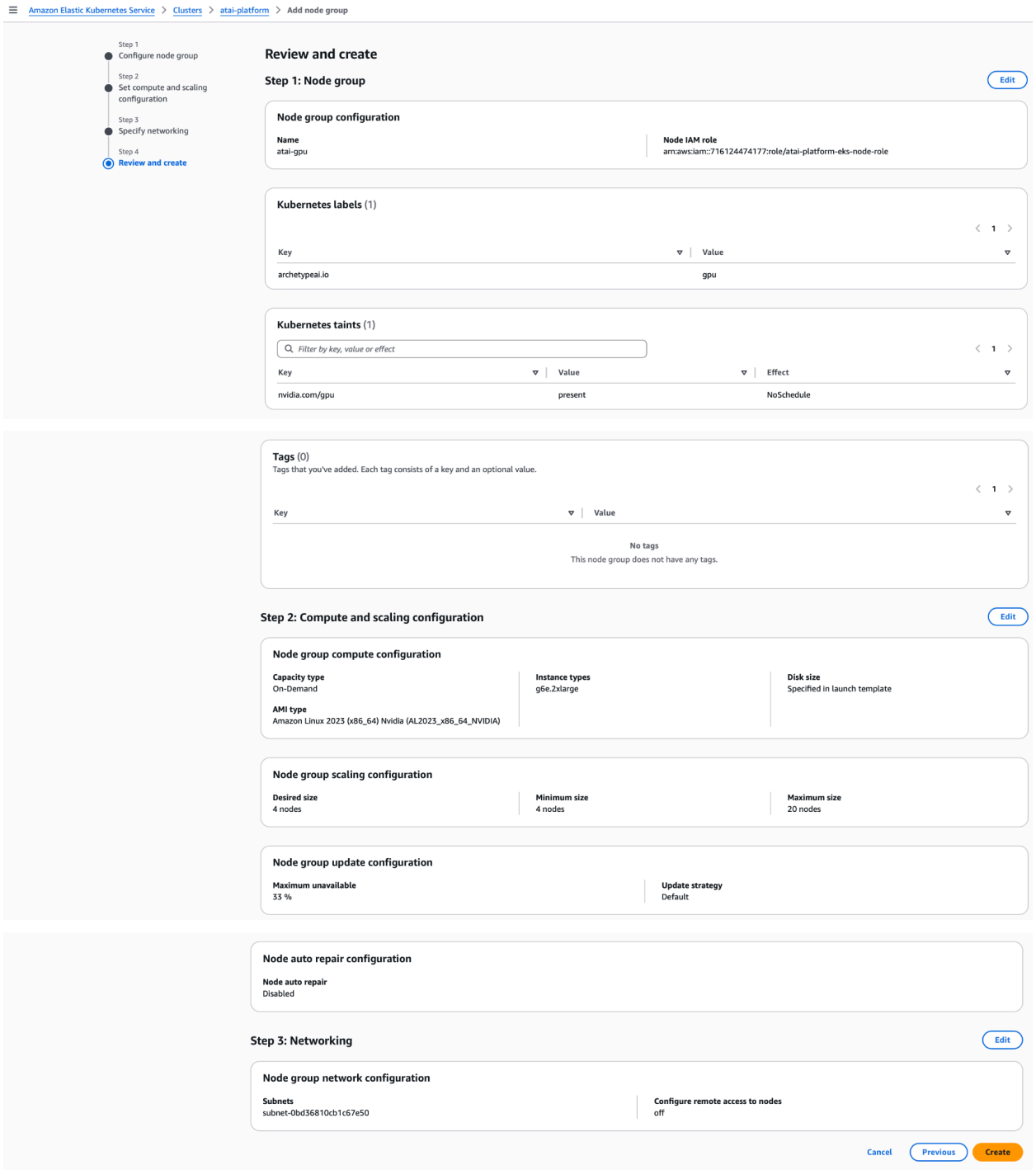


9. Node group network configuration
    a. Subnets: Select your private subnets:
        i. atai-platform-vpc-private-us-west-2a (10.5.0.0/20)

    It's important to select the private subnet in the same AZs as your Valkey clusters and RDS PostgreSQL cluster.



10. Review and click on **Create**
    Wait for node group to be ACTIVE (5-10 minutes)



EKS configuration - Install the NVIDIA Device Plugin

The NVIDIA device plugin DaemonSet to enable GPU resource scheduling in Kubernetes.

Prerequisites

1. EKS cluster is ACTIVE
2. The kubectl command line tool is required. The version can be the same as or up to one minor version earlier or later than the Kubernetes version of your cluster.
3. An IAM principal with permissions to create and describe an Amazon EKS cluster

Step 1: Manual installation

1. Direct download from GitHub:

```
None
curl -L -o nvidia-device-plugin-v0.18.0.yml
https://raw.githubusercontent.com/NVIDIA/k8s-device-plugin/v0.18.0/deployments/
static/nvidia-device-plugin.yml
```

2. Then apply the Kubernetes manifest:

```
None
kubectl apply -f nvidia-device-plugin-v0.18.0.yml
```

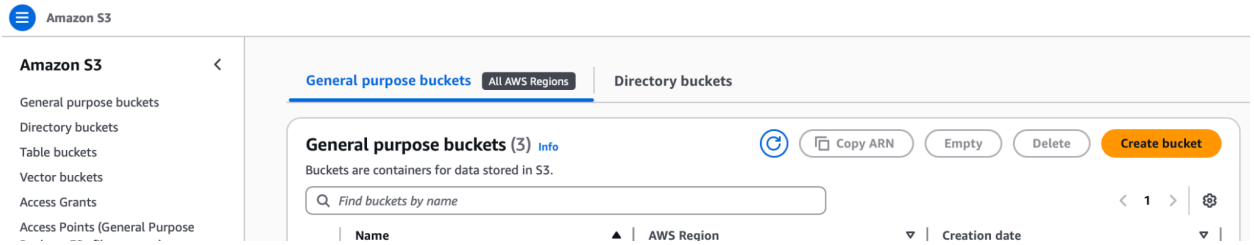
3. Verify nodes are annotated with the nvidia label:

```
None
kubectl get nodes -o
custom-columns="NAME:.metadata.name,GPU:.status.allocatable.nvidia\.com/gpu"
```

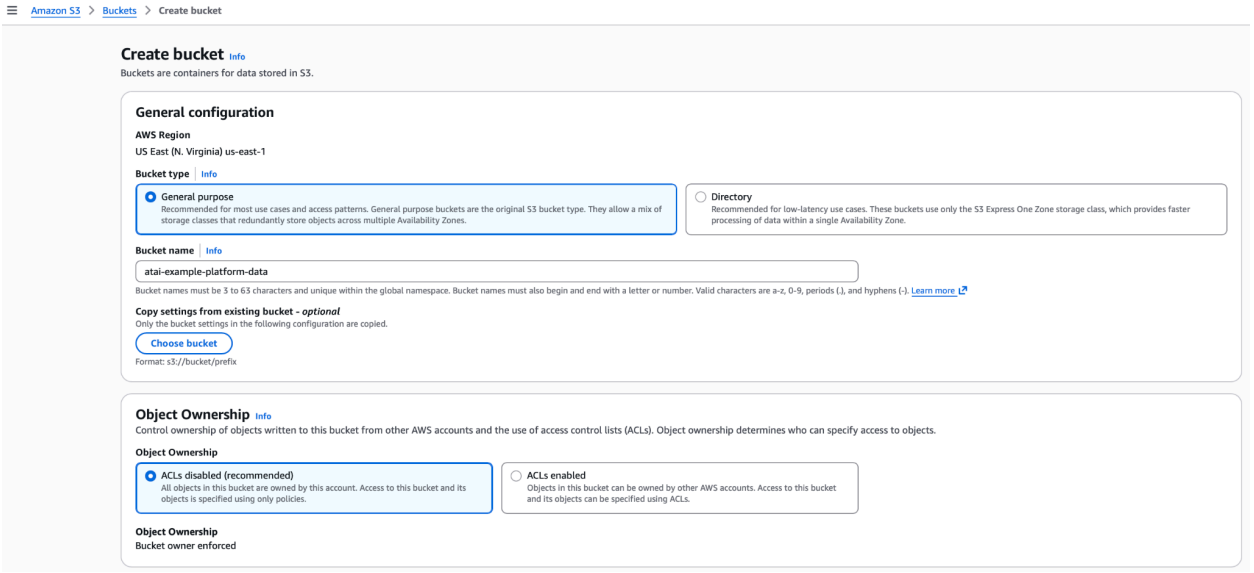
S3 configuration

Step 1: Create the platform-data bucket

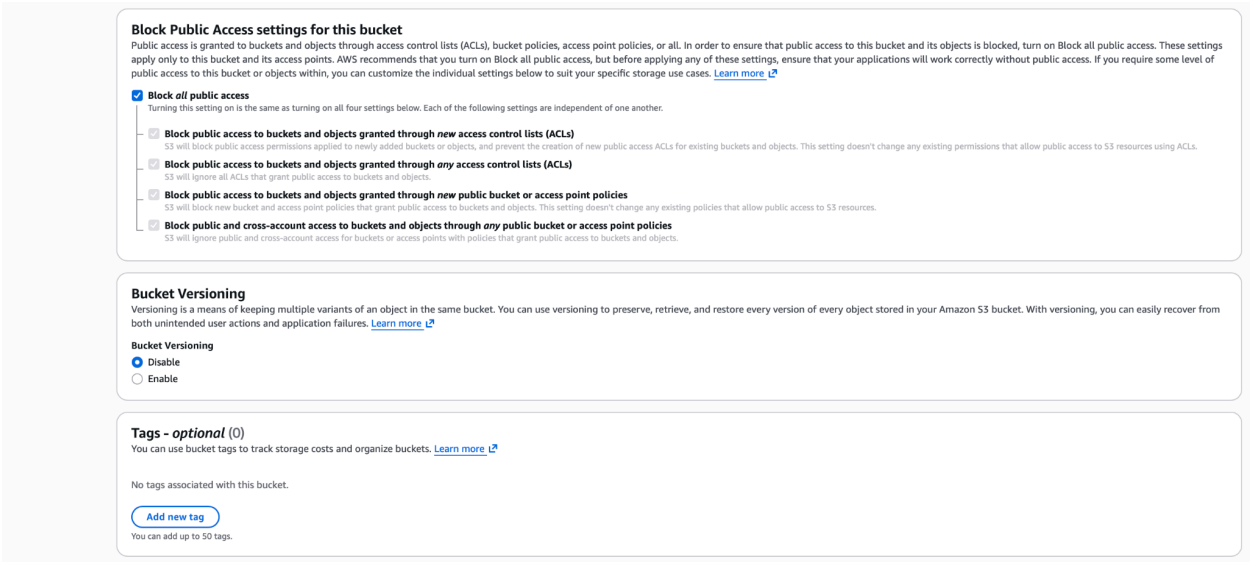
1. Go to the S3 dashboard and click on **Create Bucket**
2. Make sure, you are located in your home AWS region



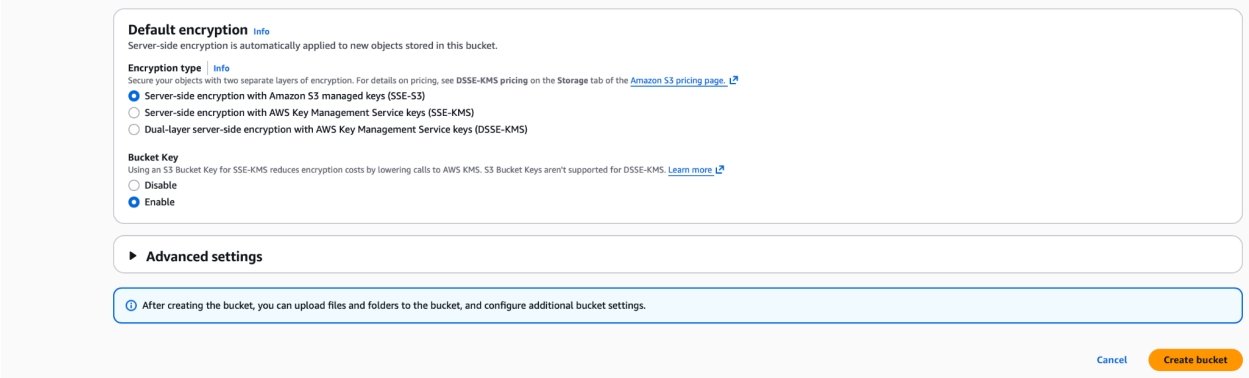
3. Bucket name: atai-<CUSTOMER UNIQUE ID>-platform-data



4. Keep the selection in the **Block Public Access setting for the bucket**



5. Use the default Encryption configuration



6. Click on **Create bucket**

⚠️ Store your bucket endpoint in a secure location. You will need them later in the *atai-platform* prerequisites, Step 3.2: Kubernetes Secrets Generation.

Note: The **platform-data** bucket will be required for following services:

- access-manager-service-backend
- api-service-health-node
- api-service-backend
- dfc-service-backend
- file-service-worker-node
- file-service-backend
- lens-node-worker-node
- lens-node-service-backend
- lens-service-backend

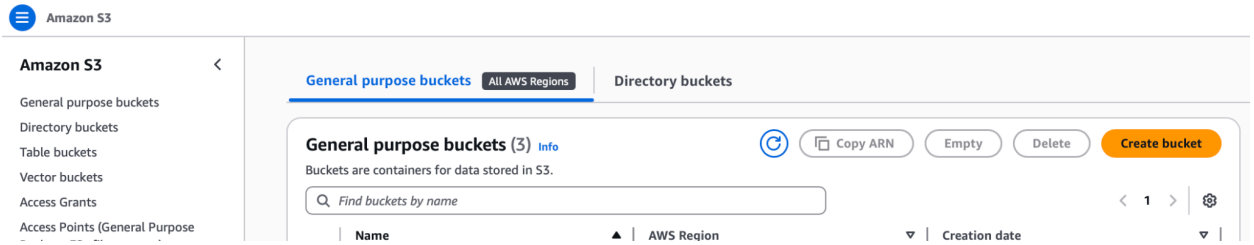
Note 2: The expected format for `<PLATFORM_S3_BUCKET>` is `s3://<BUCKET_NAME>`. These values will be referenced as:

```
None
[atai-platform-access-manager-service-backend]
...
PLATFORM_ASSETS_DIR=s3://<BUCKET_NAME>

[atai-platform-api-service-health-node]
...
PLATFORM_ASSETS_DIR=s3://<BUCKET_NAME>

[atai-platform-api-service-backend]
...
```

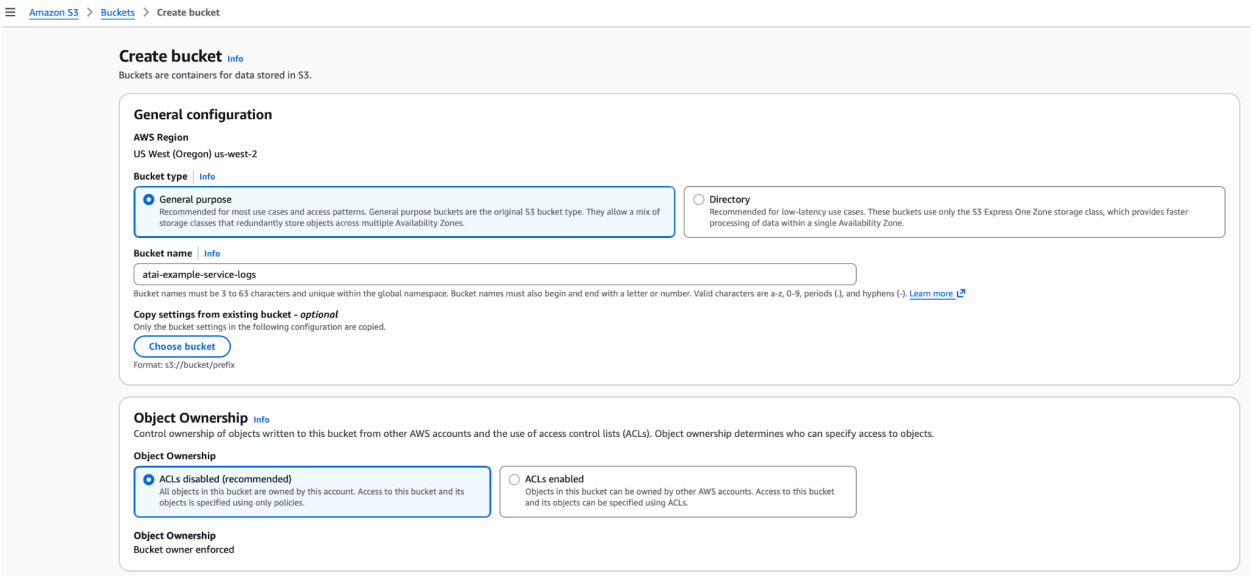
```
PLATFORM_ASSETS_DIR=s3://<BUCKET_NAME>

[atai-platform-dfc-service-backend]
...
COLD_STORAGE_ROOT_PATH=s3://<BUCKET_NAME>/dfc_service


[atai-platform-file-service-worker-node]
...
PLATFORM_ASSETS_DIR=s3://<BUCKET_NAME>

[atai-platform-file-service-backend]
...
PLATFORM_ASSETS_DIR=s3://<BUCKET_NAME>

[atai-platform-lens-node-worker-node]
...
PLATFORM_ASSETS_DIR=s3://<BUCKET_NAME>

[atai-platform-lens-node-service-backend]
...
PLATFORM_ASSETS_DIR=s3://<BUCKET_NAME>

[atai-platform-lens-service-backend]
...
PLATFORM_ASSETS_DIR=s3://<BUCKET_NAME>
```

Step 2: Create the service logs bucket

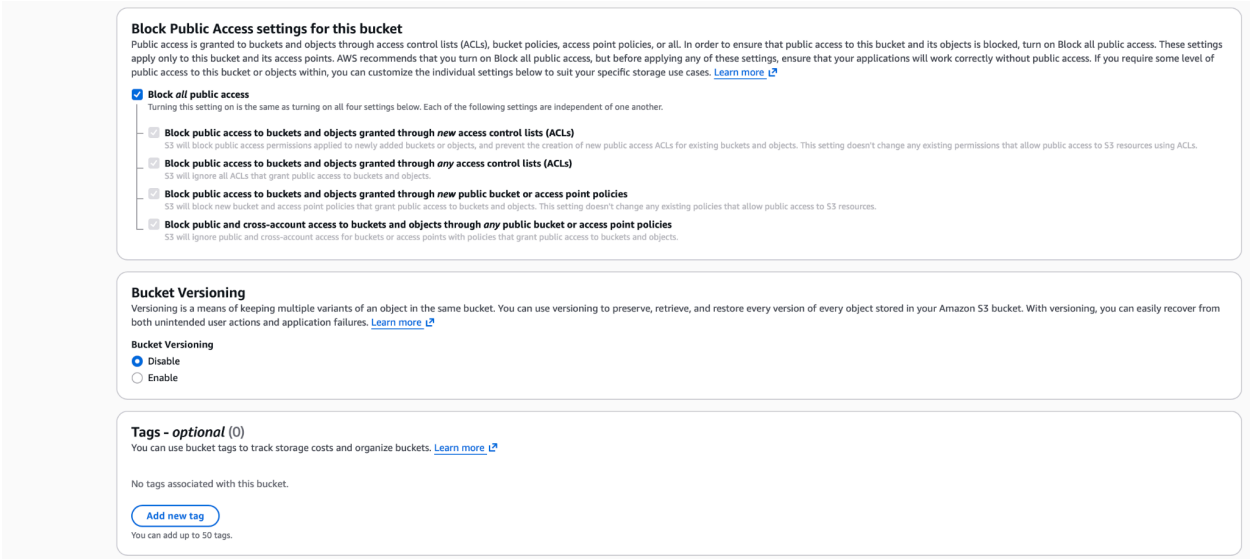
1. Go to the S3 dashboard and click on **Create Bucket**
2. Make sure, you are located in your home AWS region



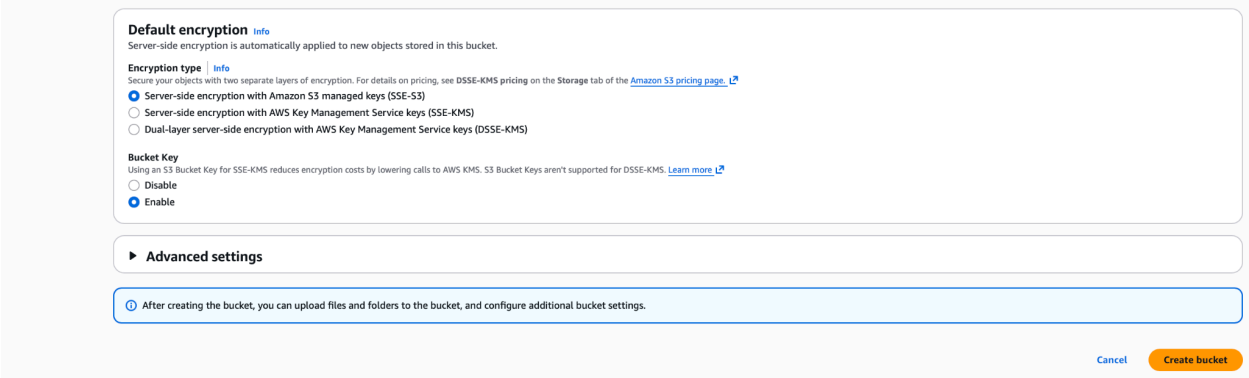
3. Bucket name: atai-<CUSTOMER UNIQUE ID>-service-logs



4. Keep the selection in the **Block Public Access setting for the bucket**



5. Use the default Encryption configuration



6. Click on **Create bucket**

⚠️ Store your bucket endpoint in a secure location. You will need them later in the *atai-platform* prerequisites, Step 3.2: Kubernetes Secrets Generation.

Note: The **service-logs** bucket will be required for following services:

- access-manager-service-backend

Note 2: The expected format for `<SERVICE_LOGS_S3_BUCKET>` is `s3://<BUCKET_NAME>`. These values will be referenced as:

```
None
[atai-platform-access-manager-service-backend]
...
PLATFORM_LOGS_BUCKET=s3://<BUCKET_NAME>
```

Application Endpoints Configuration

Platform Architecture Overview

The Archetype platform consists of two main entry points that must be publicly accessible:

1. **Console**: The web-based user interface where users interact with the platform
2. **API**: The backend service that handles business logic and data processing

For proper operation, the API must be publicly accessible so the Console can communicate with it, and vice versa. This is a standard configuration for web applications where the frontend (Console) and backend (API) need to exchange data.

Required Public URLs

Once configured, the following URLs must be publicly accessible and secured:

1. **console-archetype.<your-domain>** - Console web application
2. **api-archetype.<your-domain>** - API backend service

Both URLs must be accessible over HTTPS with valid SSL certificates.

How Do I Expose My Services Publicly?

To make your Console and API services publicly accessible, you need to use a Kubernetes Ingress.

What is an Ingress?

An Ingress is a Kubernetes resource that defines routing rules for external traffic. It specifies which domain names should route to which services and on which paths. However, an Ingress resource by itself doesn't handle traffic, you need an Ingress Controller to process these rules and route the actual traffic.

An Ingress Controller acts as a reverse proxy that:

- Receives external HTTP/HTTPS traffic
- Reads Ingress resource rules
- Routes traffic to the appropriate backend services based on domain names and paths

Common Ingress Controllers:Popular Ingress Controller options include:

- AWS Load Balancer Controller - Creates AWS Application Load Balancers (ALB) or Network Load Balancers (NLB)
- NGINX Ingress Controller - A widely-used, feature-rich Ingress Controller
- Traefik - Another popular option

The Archetype platform uses a combination of AWS Load Balancer Controller and NGINX Ingress Controller to provide both AWS-native load balancer management and flexible traffic routing capabilities.

If you don't have an Ingress Controller configured: See Appendix EKS configuration - Install the AWS Load Balancer controller and then EKS configuration - Install the NGINX ingress controller.

How Do I Secure the Traffic?

Once your services are exposed via Ingress, it's important to secure the traffic with SSL/TLS certificates. This ensures all communications are encrypted and secure.

Common Certificate Solutions: Ingress Controllers typically integrate with certificate management solutions:

- Let's Encrypt through Cert-Manager - Automated, free SSL certificates that are automatically renewed
- AWS Certificate Manager (ACM) - Native AWS-managed certificates

Cert-Manager is a Kubernetes add-on that automatically provisions, renews, and manages SSL certificates. It can integrate with Let's Encrypt to obtain free certificates automatically, or work with AWS ACM for AWS-managed certificates.

If you need help configuring certificates: See Appendix: EKS configuration - Configure Cert-Manager and Let's Encrypt

How Do I Configure DNS?

After your Ingress Controller is configured, it typically creates an AWS Network Load Balancer (NLB) or Application Load Balancer (ALB) as the entry point for your services.

DNS Configuration Steps:

1. Your Ingress Controller creates an AWS load balancer (NLB or ALB)
    a. The load balancer receives a public DNS name or IP address
2. You need to add DNS records that point your domain names to this load balancer

Required DNS Records:

1. Add the following DNS records (A records or CNAME records) pointing to the load balancer created by your Ingress solution:
    a. **console-archetype.<your-domain>** → Points to the load balancer
    b. **api-archetype.<your-domain>** → Points to the load balancer

The exact DNS configuration depends on your DNS provider and whether your load balancer provides a DNS name (use CNAME) or an IP address (use A record).

Deployment Checklist

Before Helm Chart Installation:

- ☐ **VPC and subnets** deployed with proper CIDR blocks
- ☐ **8 Valkey instances** created with correct names and versions
- ☐ **1 RDS PostgreSQL** instance created with Aurora engine
- ☐ **EKS cluster** deployed with Kubernetes 1.33
- ☐ **1 CPU node group** deployed with m6i.4xlargeinstances
- ☐ **4 GPU node group** deployed with g6e.2xlarge instances and taints
- ☐ **S3 bucket** atai-{customer-prefix}-platform-data and atai-{customer-prefix}-service-logs created
- ☐ **Network connectivity** verified between pods and databases
- ☐ **k8s Ingress** solution installed in the EKS cluster

Support

For questions about infrastructure requirements, contact the Archetype team (support@archetypeai.dev) for validation of instance types and scaling configurations.

atai-platform

Prerequisites

1. The kubectl command line tool is required. The version can be the same as or up to one minor version earlier or later than the Kubernetes version of your cluster.
2. The eksctl command line tool is required. For more information visit Installation options for Eksctl.
3. Version 2.12.3 or later or version 1.27.160 or later of the AWS Command Line Interface (AWS CLI) installed and configured on your device.
4. An IAM principal with permissions to create and describe an Amazon EKS cluster

Download the configuration files

1. Run the commands below to get the configuration files and script required for the atai-platform configuration:

```
None
mkdir -p scripts/policies && \
curl -o scripts/6.create-irsa.sh
https://archetypeai-marketplace-assets.s3.us-west-2.amazonaws.com/scripts/6.cre
ate-irsa.sh && \
curl -o scripts/7.create-k8s-secrets.sh
https://archetypeai-marketplace-assets.s3.us-west-2.amazonaws.com/scripts/7.cre
ate-k8s-secrets.sh && \
curl -o scripts/secrets.ini.template
https://archetypeai-marketplace-assets.s3.us-west-2.amazonaws.com/scripts/secre
ts.ini.template && \
curl -o scripts/policies/platform-data-access.json.tpl
https://archetypeai-marketplace-assets.s3.us-west-2.amazonaws.com/scripts/polic
ies/platform-data-access.json.tpl && \
curl -o scripts/policies/service-logs-access.json.tpl
https://archetypeai-marketplace-assets.s3.us-west-2.amazonaws.com/scripts/polic
ies/service-logs-access.json.tpl && \
curl -o scripts/policies/model-depot-access.json
https://archetypeai-marketplace-assets.s3.us-west-2.amazonaws.com/scripts/polic
ies/model-depot-access.json
```

2. Make the scripts executable:

```
None
chmod +x scripts/6.create-irsa.sh scripts/7.create-k8s-secrets.sh
```

Step 1: Kubernetes namespaces

1. Create the namespace to install all the components of the atai-platform

```
None
$ kubectl create namespace atai-platform
namespace/atai-platform created
```

Step 2: Kubernetes Service account for IAM roles (IRSA)

1. Associated an IAM OIDC provider

```
None
$ eksctl utils associate-iam-oidc-provider \
    --region <AWS_REGION> \
    --cluster atai-platform \
    --approve

2025-11-07 21:47:54 [ℹ]  will create IAM Open ID Connect provider for cluster
"atai-platform" in "us-west-2"
2025-11-07 21:47:55 [✔]  created IAM Open ID Connect provider for cluster
"atai-platform" in "us-west-2"
```

2. Run the script 6.create-irsa.sh

```
None
$ ./6.create-irsa.sh \
  --region us-west-2 \
  --customer-name example \
  --cluster-name atai-platform \
  --platform-data-bucket "atai-example-platform-data" \
  --service-logs-bucket "atai-example-service-logs"

[INFO] Checking required commands...
[INFO] Getting AWS account ID...
[INFO] AWS Account ID: 123456789123
[INFO] Verifying cluster atai-platform exists...
[INFO] Associating IAM OIDC provider...
2025-11-09 09:51:36 [ℹ]  IAM Open ID Connect provider is already associated
with cluster "atai-platform" in "us-west-2"
[INFO] Verifying S3 buckets exist...
```

```
{
    "BucketRegion": "us-west-2",
    "AccessPointAlias": false
}
{
    "BucketRegion": "us-west-2",
    "AccessPointAlias": false
}
[INFO] Verified buckets exist: atai-example-platform-data,
atai-example-service-logs
[INFO] Creating IAM policies...
[INFO] Creating policy: atai-example-platform-data-access
[INFO] Created policy:
arn:aws:iam::123456789123:policy/atai-example-platform-data-access
[INFO] Creating policy: atai-example-service-logs-access
[INFO] Created policy:
arn:aws:iam::123456789123:policy/atai-example-service-logs-access
[INFO] Creating policy: atai-example-model-depot-access
[INFO] Created policy:
arn:aws:iam::123456789123:policy/atai-example-model-depot-access
[INFO] Creating namespace: atai-platform
Warning: resource namespaces/atai-platform is missing the
kubectl.kubernetes.io/last-applied-configuration annotation which is required
by kubectl apply. kubectl apply should only be used on resources created
declaratively by either kubectl create --save-config or kubectl apply. The
missing annotation will be patched automatically.
namespace/atai-platform configured
[INFO] Creating IAM role and service account: atai-platform-sa
2025-11-09 09:51:52 [ℹ]  1 iamserviceaccount (atai-platform/atai-platform-sa)
was included (based on the include/exclude rules)
2025-11-09 09:51:52 [!]  metadata of serviceaccounts that exist in Kubernetes
will be updated, as --override-existing-serviceaccounts was set
2025-11-09 09:51:52 [ℹ]  1 task: {
    2 sequential sub-tasks: {
        create IAM role for serviceaccount "atai-platform/atai-platform-sa",
        create serviceaccount "atai-platform/atai-platform-sa",
    } }2025-11-09 09:51:52 [ℹ]  building iamserviceaccount stack
"eksctl-atai-platform-addon-iamserviceaccount-atai-platform-atai-platform-sa"
2025-11-09 09:51:53 [ℹ]  deploying stack
"eksctl-atai-platform-addon-iamserviceaccount-atai-platform-atai-platform-sa"
2025-11-09 09:51:53 [ℹ]  waiting for CloudFormation stack
"eksctl-atai-platform-addon-iamserviceaccount-atai-platform-atai-platform-sa"
2025-11-09 09:52:23 [ℹ]  waiting for CloudFormation stack
"eksctl-atai-platform-addon-iamserviceaccount-atai-platform-atai-platform-sa"
```

```
2025-11-09 09:52:26 [i]  created serviceaccount
"atai-platform/atai-platform-sa"
[INFO] Created IAM role and service account successfully
[INFO] Setup completed successfully!
[INFO]
[INFO] Summary:
[INFO]   - Created namespace: atai-platform
[INFO]   - Created 3 IAM policies
[INFO]   - Using S3 buckets:
[INFO]     * Platform Data: atai-example-platform-data
[INFO]     * Service Logs:  atai-example-service-logs
[INFO]   - Created IAM role: atai-example-platform-role
[INFO]   - Created service account: atai-platform-sa
[INFO]
[INFO] Policy ARNs:
[INFO]   Platform Data:
arn:aws:iam::123456789123:policy/atai-example-platform-data-access
[INFO]   Service Logs:
arn:aws:iam::123456789123:policy/atai-example-service-logs-access
[INFO]   Model Depot:
arn:aws:iam::123456789123:policy/atai-example-model-depot-access
[INFO]
[INFO] Bucket ARNs:
[INFO]   Platform Data: arn:aws:s3:::atai-example-platform-data
[INFO]   Service Logs:  arn:aws:s3:::atai-example-service-logs
[INFO]   Model Depot:   arn:aws:s3:::atai-marketplace-model-depot
(base) ccastellanos@Cristians-MacBook-Pro scripts %
```

Step 3: Kubernetes secrets required for the atai-platform services

Step 3.1 Generate values for the IAM service secret

Master Key (`IAM_MASTER_KEY`)

Used to authenticate **administrative operations**, such as creating and managing organizations and keys.

Purpose:

- Solves the *bootstrapping problem* — allows you to populate an empty database.
- Can be **changed at any time**, since it's only stored in runtime.
- Gets **hashed with Argon2** before being stored in memory.

To generate:

```
None
openssl rand -base64 32
```

After generation stores the secret in a secure place, you will need it for Step 3.2 Kubernetes secret generation.

Server Salt (`IAM_SERVER_SALT`)

Used as the **salt input for Argon2** when hashing API keys.

Key Points:

- Must remain **persistent** for the lifetime of the service.
  (Changing it invalidates all existing keys.)
- Leaking it doesn't immediately compromise security **if keys are service-generated** (not manually uploaded).

To generate:

```
None
openssl rand -base64 48
```

After generation stores the secret in a secure place, you will need it for Step 3.2 Kubernetes secret generation.

IAM db name (`IAM_DB_NAME`)
Postgres database that was previously created called iam_db in the section PostgreSQL database configuration Step 4: Extra Database Configuration steps. You will need it for Step 3.2 Kubernetes secret generation.

IAM db user (`IAM_DB_USER`)
atai_dev postgres user that was created in the PostgreSQL database configuration Step 4: Extra Database Configuration steps. You will need it for Step 3.2 Kubernetes secret generation.

IAM db password (`IAM_DB_PASSWORD`)
atai_dev postgres user password that was created in the PostgreSQL database configuration Step 4: Extra Database Configuration steps. You will need it for Step 3.2 Kubernetes secret generation.

IAM db port (`IAM_DB_PORT`)
Port of your PostgreSQL instance

IAM db password (`IAM_DB_HOST`)
Host of your PostgreSQL instance

Step 3.2 Kubernetes secret generation

1. Create a ***secrets.ini*** file using the ***secrets.ini.template*** file as a base. Below you will find a guide on how to replace the placeholder values in the ***secrets.ini.template*** file.

Placeholder	**Description**
<PLATFORM_ENVIRONMENT_TAG>	Deployment environment identifier (e.g., atai-sandbox, customer-name, etc,.) Example: **atai-sandbox**
<PLATFORM_API_ENDPOINT>	Public API endpoint URL for the platform. 1. Identify your custom domain (e.g., example.com, customer.com) 2. Replace {your-domain} in the format with your actual domain: **https://api.archetype.{your-domain}/v0.5**
<CONSOLE_ENDPOINT>	Public endpoint URL for console/web interface 1. Identify your custom domain (e.g., example.com, customer.com) 2. Replace {your-domain} in the format with your actual domain: **https://console.archetype.{your-domain}**
<LENS_EXTERNAL_SESSION_ENDPOINT>	External endpoint URL for lens session management 1. Identify your custom domain (e.g., example.com, customer.com) 2. Replace {your-domain} in the format with your actual domain: **wss://api.sandbox.{your-domain}/v0.5**

Note 1: the items above will be required for following services:

- <PLATFORM_ENVIRONMENT_TAG>
  - All services
- <PLATFORM_API_ENDPOINT>
  - api-service-health-node
  - api-service-backend
  - console-2-service-frontend
  - lens-node-service-backend
- <CONSOLE_ENDPOINT>
  - api-service-health-node
  - api-service-backend
  - console-2-service-frontend
- <LENS_EXTERNAL_SESSION_ENDPOINT>
  - api-service-health-node
  - api-service-backend

Note 2: Some values are hardcoded and must be exactly as the template referenced below.

- <PLATFORM_BOT_API_KEY>
  - lens-service-backend
- <MODEL_DEPOT_URI>
  - gpq-node-newton-model-c23
  - gpq-node-newton-model-omega
- <FLAGS_SECRET>
  - console-2-service-frontend

These values will be referenced as:

```
Shell
[atai-platform-access-manager-service-backend]
...
PLATFORM_ENVIRONMENT_TAG=<PLATFORM_ENVIRONMENT_TAG>

[atai-platform-api-events-service-backend]
...
PLATFORM_ENVIRONMENT_TAG=<PLATFORM_ENVIRONMENT_TAG>

[atai-platform-api-service-health-node]
...
PLATFORM_ENVIRONMENT_TAG=<PLATFORM_ENVIRONMENT_TAG>
PLATFORM_API_ENDPOINT=<PLATFORM_API_ENDPOINT>
CONSOLE_ENDPOINT=<CONSOLE_ENDPOINT>
LENS_EXTERNAL_SESSION_ENDPOINT=<LENS_EXTERNAL_SESSION_ENDPOINT>
```

```
[atai-platform-api-service-backend]
...
PLATFORM_ENVIRONMENT_TAG=<PLATFORM_ENVIRONMENT_TAG>
PLATFORM_API_ENDPOINT=<PLATFORM_API_ENDPOINT>
CONSOLE_ENDPOINT=<CONSOLE_ENDPOINT>
LENS_EXTERNAL_SESSION_ENDPOINT=<LENS_EXTERNAL_SESSION_ENDPOINT>

[atai-platform-console-2-service-frontend]
...
PLATFORM_ENVIRONMENT_TAG=<PLATFORM_ENVIRONMENT_TAG>
PUBLIC_CONSOLE_API_URL=<PLATFORM_API_ENDPOINT>
CONSOLE_ENDPOINT=<CONSOLE_ENDPOINT>
CONSOLE_API_URL=<PLATFORM_API_ENDPOINT>
FLAGS_SECRET=abc123ABC132

[atai-platform-dfc-service-backend]
...
PLATFORM_ENVIRONMENT_TAG=<PLATFORM_ENVIRONMENT_TAG>

[atai-platform-file-service-worker-node]
...
PLATFORM_ENVIRONMENT_TAG=<PLATFORM_ENVIRONMENT_TAG>

[atai-platform-file-service-backend]
...
PLATFORM_ENVIRONMENT_TAG=<PLATFORM_ENVIRONMENT_TAG>

[atai-platform-gpq-node-newton-model-c23]
...
PLATFORM_ENVIRONMENT_TAG=<PLATFORM_ENVIRONMENT_TAG>
MODEL_DEPOT_URI=s3://atai-marketplace-model-depot

[atai-platform-gpq-node-newton-model-omega]
...
PLATFORM_ENVIRONMENT_TAG=<PLATFORM_ENVIRONMENT_TAG>
MODEL_DEPOT_URI=s3://atai-marketplace-model-depot

[atai-platform-gpq-service-event-router]
...
PLATFORM_ENVIRONMENT_TAG=<PLATFORM_ENVIRONMENT_TAG>

[atai-platform-gpq-service-backend]
...
PLATFORM_ENVIRONMENT_TAG=<PLATFORM_ENVIRONMENT_TAG>
```

```
[atai-platform-health-service-backend]
...
PLATFORM_ENVIRONMENT_TAG=<PLATFORM_ENVIRONMENT_TAG>

[atai-platform-lens-node-worker-node]
...
PLATFORM_ENVIRONMENT_TAG=<PLATFORM_ENVIRONMENT_TAG>

[atai-platform-lens-node-service-backend]
...
PLATFORM_ENVIRONMENT_TAG=<PLATFORM_ENVIRONMENT_TAG>
PLATFORM_API_ENDPOINT=<PLATFORM_API_ENDPOINT>

[atai-platform-lens-service-db-backend]
...
PLATFORM_ENVIRONMENT_TAG=<PLATFORM_ENVIRONMENT_TAG>

[atai-platform-lens-service-backend]
...
PLATFORM_ENVIRONMENT_TAG=<PLATFORM_ENVIRONMENT_TAG>
PLATFORM_BOT_API_KEY=abc123ABC132

[atai-platform-registry-service-backend]
...
PLATFORM_ENVIRONMENT_TAG=<PLATFORM_ENVIRONMENT_TAG>
```

2. Run the script 7.create-k8s-secrets.sh

Shell

```
$ ./7.create-k8s-secrets.sh -n atai-platform -f secrets.ini

secret/atai-platform-access-manager-service-backend created
secret/atai-platform-api-events-service-backend created
secret/atai-platform-api-service-health-node created
secret/atai-platform-api-service-backend created
secret/atai-platform-console-2-service-frontend created
secret/atai-platform-dfc-service-backend created
secret/atai-platform-file-service-worker-node created
secret/atai-platform-file-service-backend created
secret/atai-platform-gpq-node-newton-model-c23 created
```

```
secret/atai-platform-gpq-node-newton-model-omega created
secret/atai-platform-gpq-service-event-router created
secret/atai-platform-gpq-service-backend created
secret/atai-platform-health-service-backend created
secret/atai-platform-lens-node-worker-node created
secret/atai-platform-lens-node-service-backend created
secret/atai-platform-lens-service-db-backend created
secret/atai-platform-lens-service-backend created
secret/atai-platform-registry-service-backend created
```

Helm chart installation

Prerequisites

1. The kubectl command line tool is required. The version can be the same as or up to one minor version earlier or later than the Kubernetes version of your cluster.
2. Version 2.12.3 or later or version 1.27.160 or later of the AWS Command Line Interface (AWS CLI) installed and configured on your device.
3. Helm 3.19.0. Learn more about Helm installation here.

Step 1: Installation

1. Retrieve an authentication token and authenticate your clients. Enter the AWS CLI:

```
None
aws ecr get-login-password \
    --region us-east-1 | helm registry login \
    --username AWS \
    --password-stdin 709825985650.dkr.ecr.us-east-1.amazonaws.com
```

2. Create a tmp folder:

```
None
mkdir awsmp-chart && cd awsmp-chart
```

3. Pull the atai-platform helm chart:

```
None
$ helm pull
oci://709825985650.dkr.ecr.us-east-1.amazonaws.com/archetype-ai/atai_core/helm/
atai-platform --version 1.0.0-99

Pulled:
709825985650.dkr.ecr.us-east-1.amazonaws.com/archetype-ai/atai_core/helm/atai-p
latform:0.1.0
Digest: <sha256:...>
709825985650.dkr.ecr.us-east-1.amazonaws.com/archetype-ai/atai_core/helm/atai-p
latform:0.1.0 contains an underscore.
OCI artifact references (e.g. tags) do not support the plus sign (+). To
support
```

```
storing semantic versions, Helm adopts the convention of changing plus (+) to
an underscore (_) in chart version tags when pushing to a registry and back to
a plus (+) when pulling from a registry.
```

4. Create a ***values.yaml*** file using the following base structure:

```
Go
api-service-backend:
  ingress:
    enabled: true
    className: nginx
    annotations:
      nginx.ingress.kubernetes.io/ssl-redirect: "true"
      nginx.ingress.kubernetes.io/backend-protocol: "HTTP"
      nginx.ingress.kubernetes.io/force-ssl-redirect: "true"
    host: <PLATFORM_API_ENDPOINT>
    path: /
    pathType: Prefix
    tls:
      enabled: true
      clusterIssuer: letsencrypt-stage
console-2-service-frontend:
  ingress:
    enabled: true
    className: nginx
    annotations:
      nginx.ingress.kubernetes.io/ssl-redirect: "true"
      nginx.ingress.kubernetes.io/backend-protocol: "HTTP"
      nginx.ingress.kubernetes.io/force-ssl-redirect: "true"
    host: <CONSOLE_ENDPOINT>
    path: /
    pathType: Prefix
    tls:
      enabled: true
      clusterIssuer: letsencrypt-stage
```

5. Install the atai-platform helm chart:

```
Shell
$ helm upgrade atai-platform atai-platform-1.0.0-99.tgz \
--namespace atai-platform \
--values values.yaml \
--install

Release "atai-platform" does not exist. Installing it now.
I1111 11:06:28.291728   35840 warnings.go:110] "Warning:
spec.template.spec.containers[0].env[9]: hides previous definition of
\"REDIS_USE_INSECURE_TLS\", which may be dropped when using apply"
NAME: atai-platform
LAST DEPLOYED: Tue Nov 11 11:06:23 2025
NAMESPACE: atai-platform
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

Getting Started with the Archetype Platform

To get started with the Archetype Platform, please visit following documentation page:
https://docs.archetypeai.app/overview/introduction

Appendix

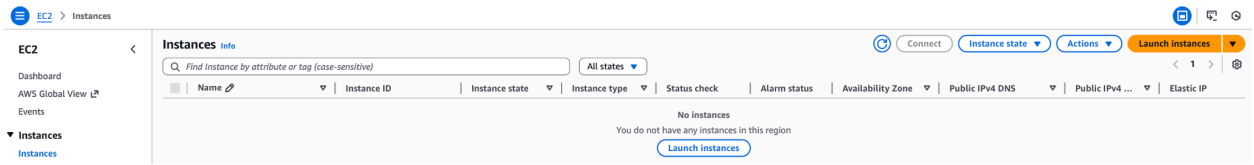
Bastion host configuration

Prerequisites

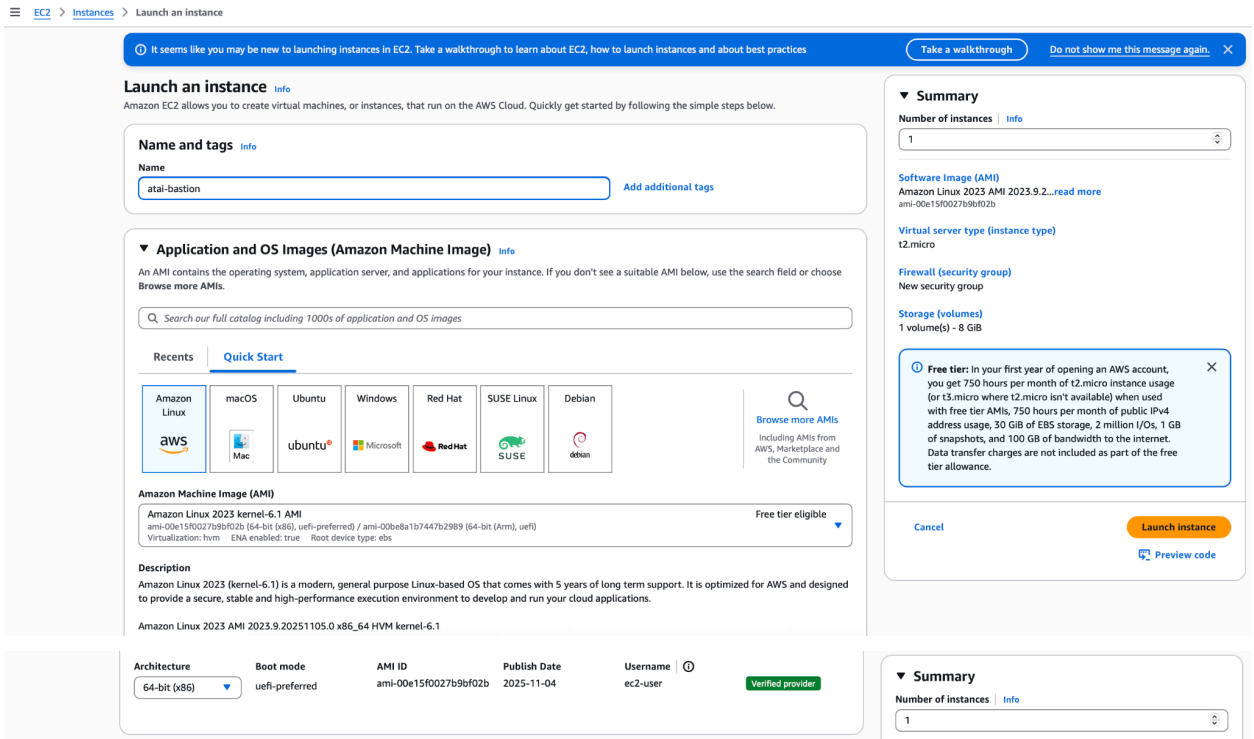
1. VPC with public subnet
2. Internet Gateway attached to VPC
3. Route table configured for public subnet

Step 1: Launch EC2 Instance (Bastion Host)

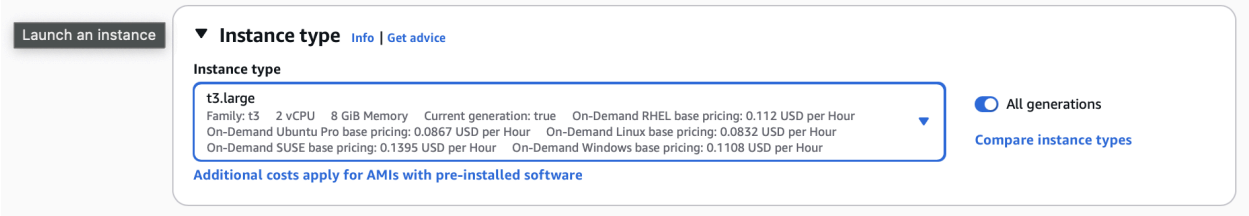
1. Go to EC2 → Instances → Launch instance



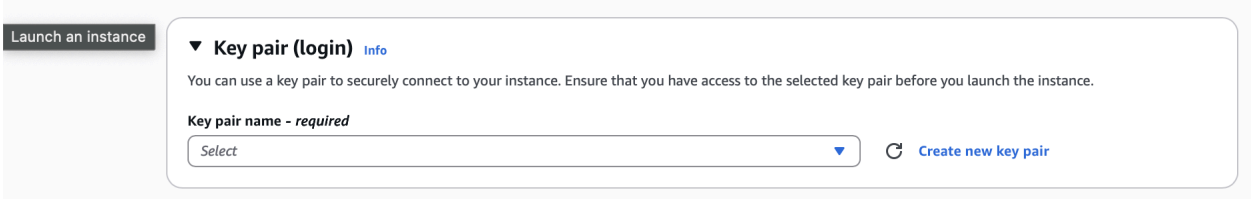
2. Name: atai-bastion
3. Application and OS Images (Amazon Machine Image):
    a. Search for: Amazon Linux 2023 AMI
    b. Select: Amazon Linux 2023 AMI (x86_64, HVM, kernel 6.1)



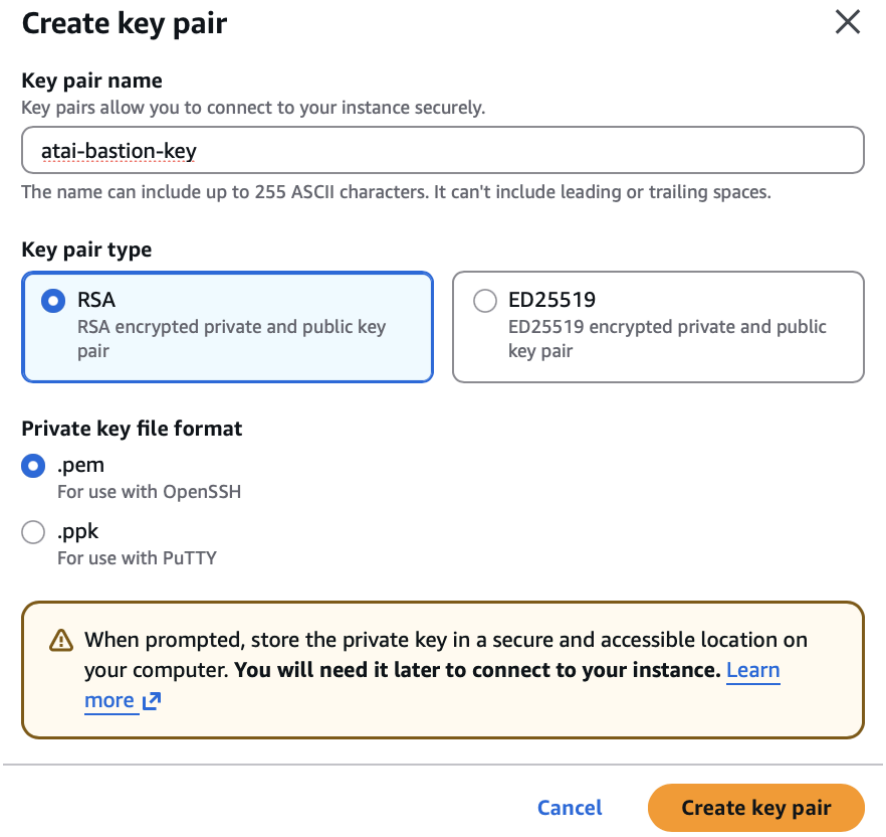
4. Instance type: Select your instance type (e.g., t3.medium for dev, t3.large for production)



5. Select an existing key pair or click on **Create a new key pair**

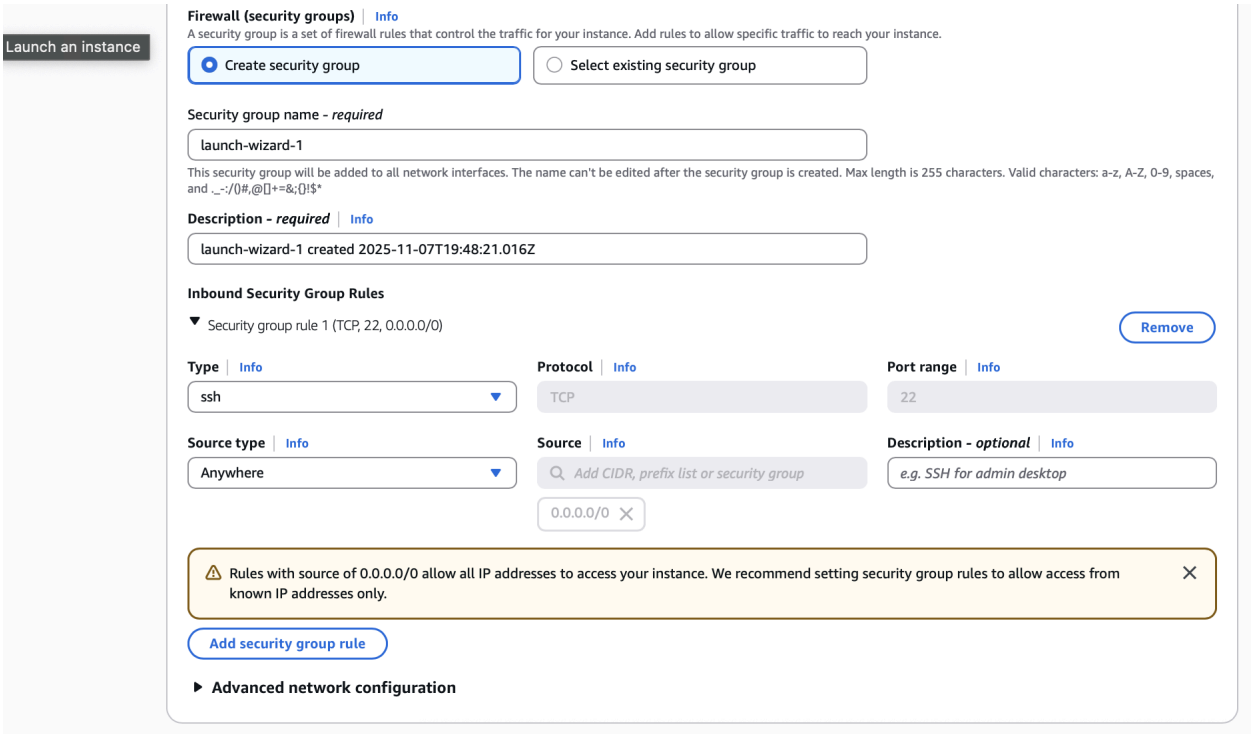


6. Assign a new atai-bastion-key, then click on **Create key pair**

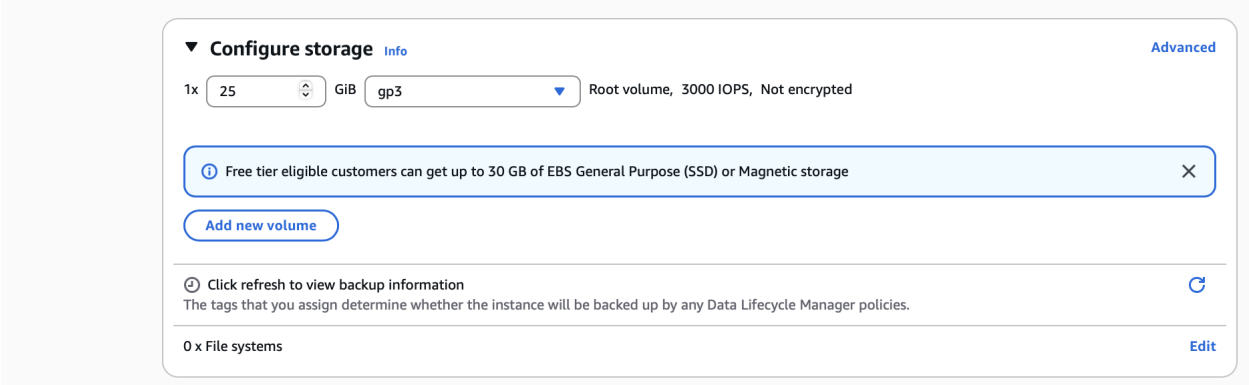


7. Network settings:
    a. VPC: Select your VPC
    b. Subnet: Select a public subnet (e.g., atai-platform-vpc-public-us-west-2a)
    c. Auto-assign public IP: Enable (or use Elastic IP from Step 4)

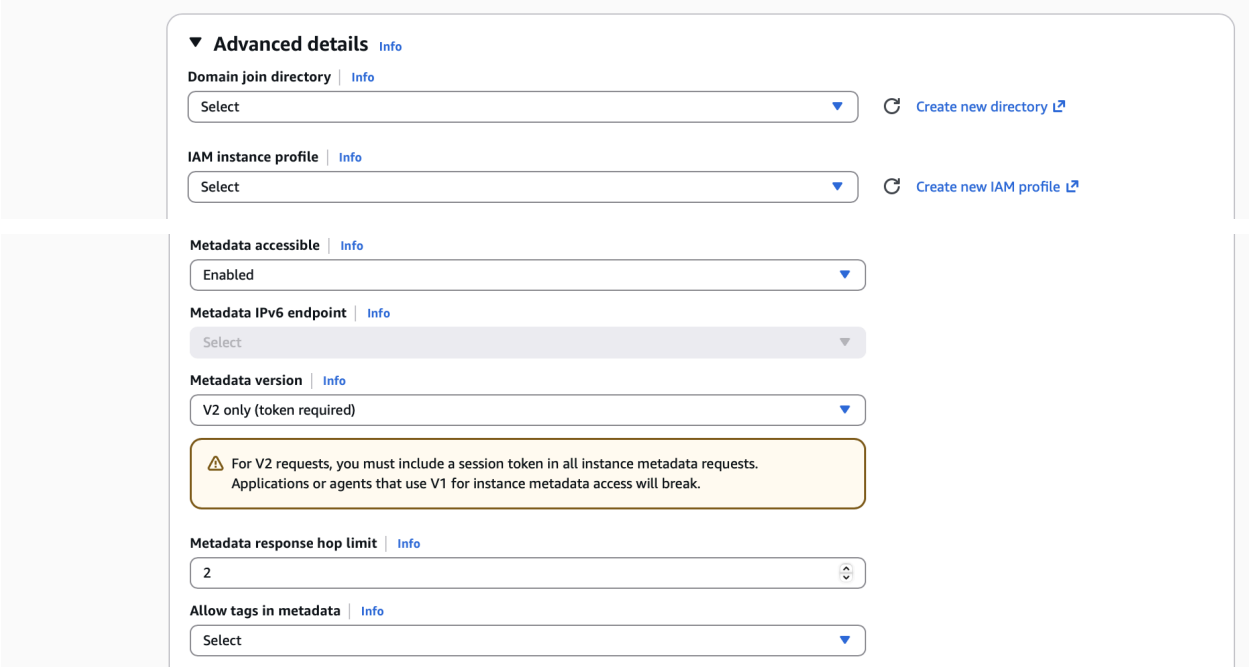
    d. Firewall (security groups): Select **Create security group**
        i. By default AWS will add an SSH rule



8. Configure storage
    a. Default (8 GiB gp3)
    b. Recommended 25 GB, but you can adjust based on your requirements



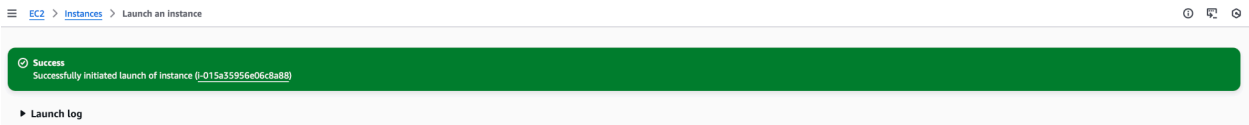
9. Advanced details (expand):
    a. Metadata accessible: Enable
    b. Metadata version: V2 only (token required) (IMDSv2)
    c. Metadata token response hop limit: 2



d. User data: Paste the following script:

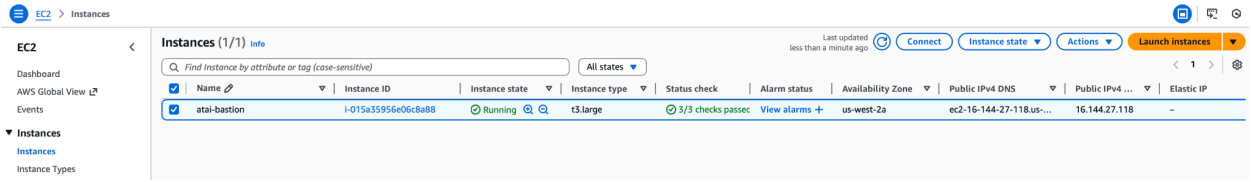
```
None
#!/bin/bash

# Update system packages
sudo dnf update -y

# Install essential packages
sudo dnf install -y \
    htop \
    vim \
    git \
    wget \
    unzip \
    jq \
    postgresql17 \
    redis6

# Install AWS CLI v2
curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o
"awscliv2.zip"
unzip awscliv2.zip
sudo ./aws/install
rm -rf aws awscliv2.zip
```

10. Click on **Launch instance**.

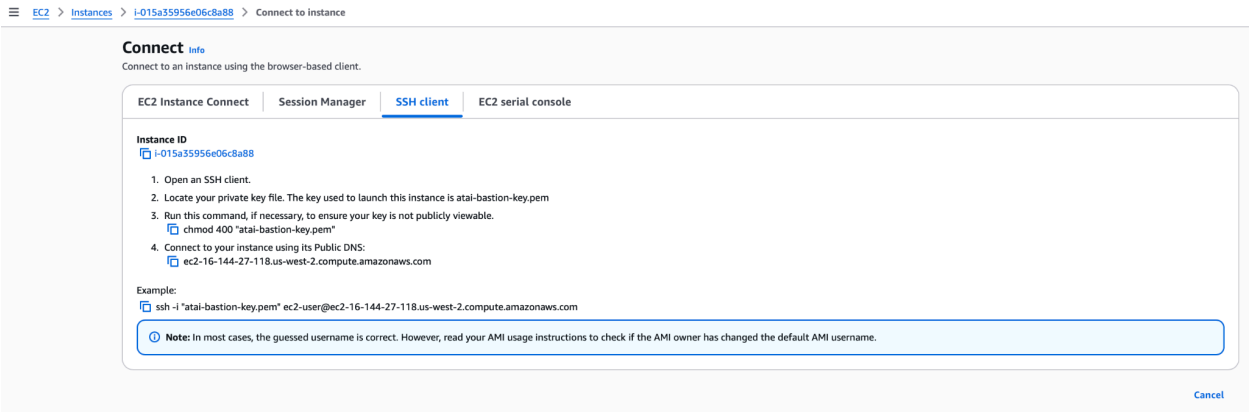


Step 2: Connect to your Bastion host

1. Go to EC2 → Instances → Select your atai-bastion instance and click on **Connect**



2. Click on the tab **SSH client**



   a. Open an SSH client.
   b. Locate your private key file. The key used to launch this instance is atai-bastion-key.pem
   c. Run this command, if necessary, to ensure your key is not publicly viewable.

```
None
chmod 400 "atai-bastion-key.pem"
```

   d. Connect to your instance using its Public DNS e.g,:

```
None
ec2-16-144-27-118.us-west-2.compute.amazonaws.com
```

Example

```
None
ssh -i "atai-bastion-key.pem"
ec2-user@ec2-16-144-27-118.us-west-2.compute.amazonaws.com
```

AWS Service Quotas

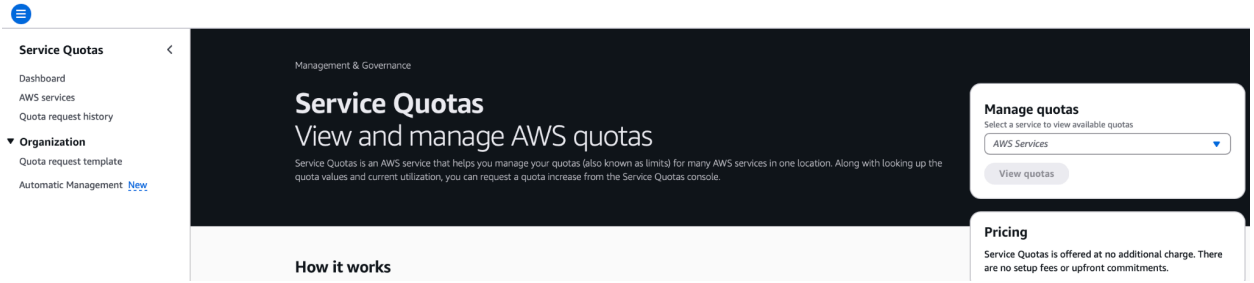
Running On-Demand G and VT instances

Requirements:

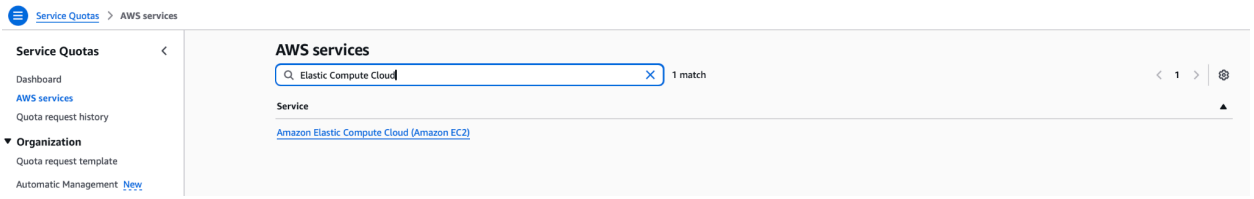
- Instance Type: g6e.2xlarge
- vCPUs per instance: 8 vCPUs
- Minimum instances required: 10
- Total vCPUs needed: 10 × 8 = 80 vCPUs

Service Quota to check:

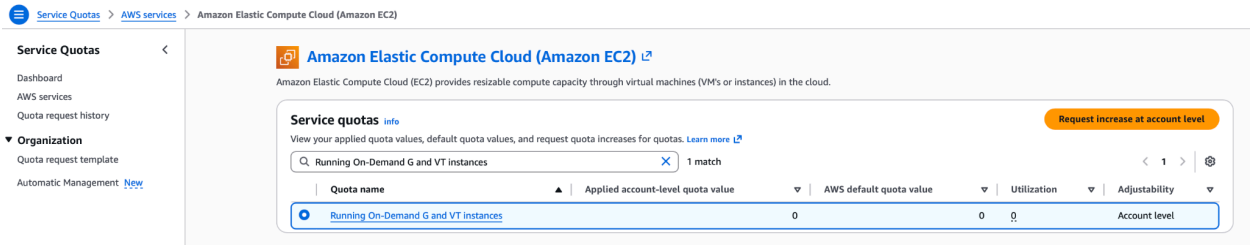
3. Go to AWS Service Quotas → In the left panel click on **AWS Services**



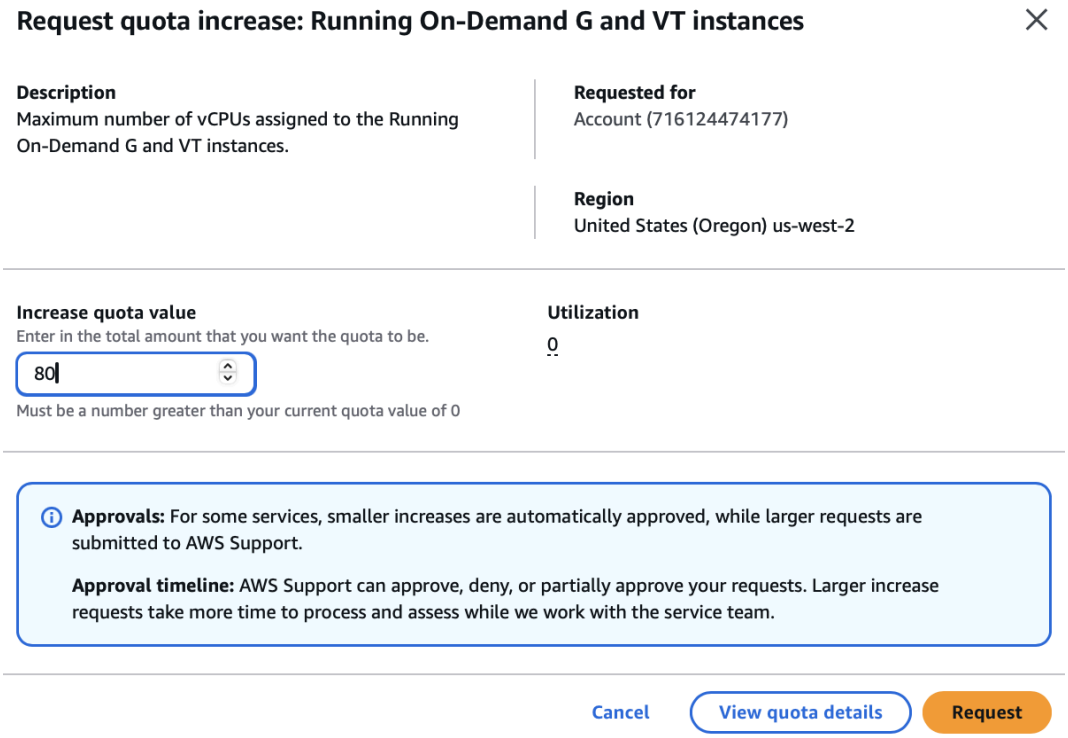
4. In the search bar type AWS Elastic Compute Cloud



5. In the search bar type **Running On-Demand G and VT instances**, select the quota and click on **Request increase at account level.**



6. Set the **Increase account value** to 80 vCPUs and click on **Request**



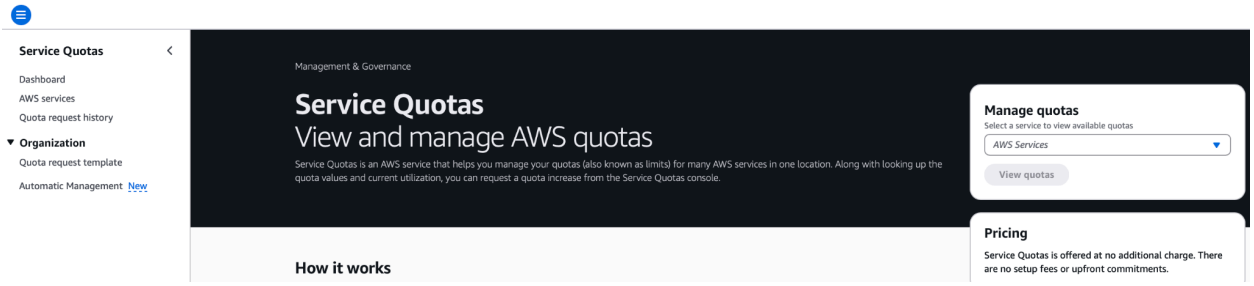
Running On-Demand Standard (A, C, D, H, I, M, R, T, Z) instances

Requirements:

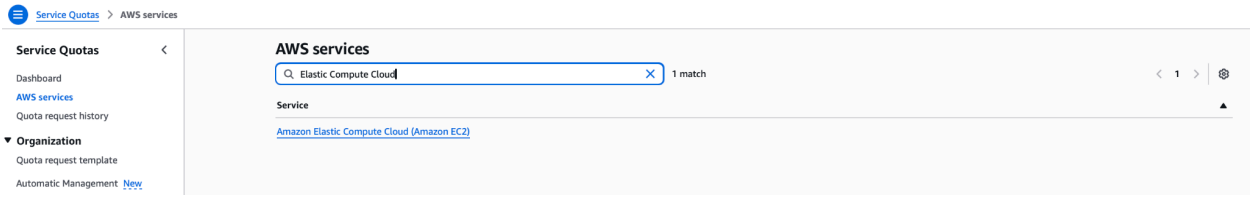
- Instance Type: m6i.4xlarge
- vCPUs per instance: 16 vCPUs
- Minimum instances required: 6
- Total vCPUs needed: 6 × 16 = 96 vCPUs

Service Quota to check:

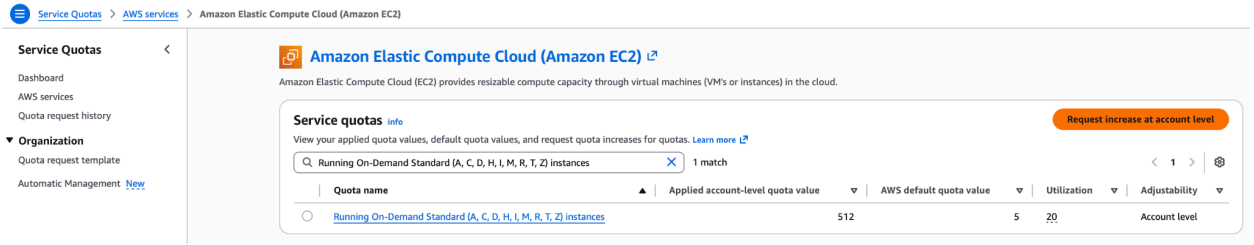
1. Go to AWS Service Quotas → In the left panel click on **AWS Services**



2. In the search bar type AWS Elastic Compute Cloud



3. In the search bar type **Running On-Demand Standard (A, C, D, H, I, M, R, T, Z) instances**, select the quota and click on **Request increase at account level.**



4. Set the **Increase account value** to 96 vCPUs and click on **Request**

Note: If your existing quota for Standard Instances (A, C, D, H, I, M, R, T, Z family) already exceeds 96 vCPUs, no increase is needed for this family.

AWS License Manager Setup

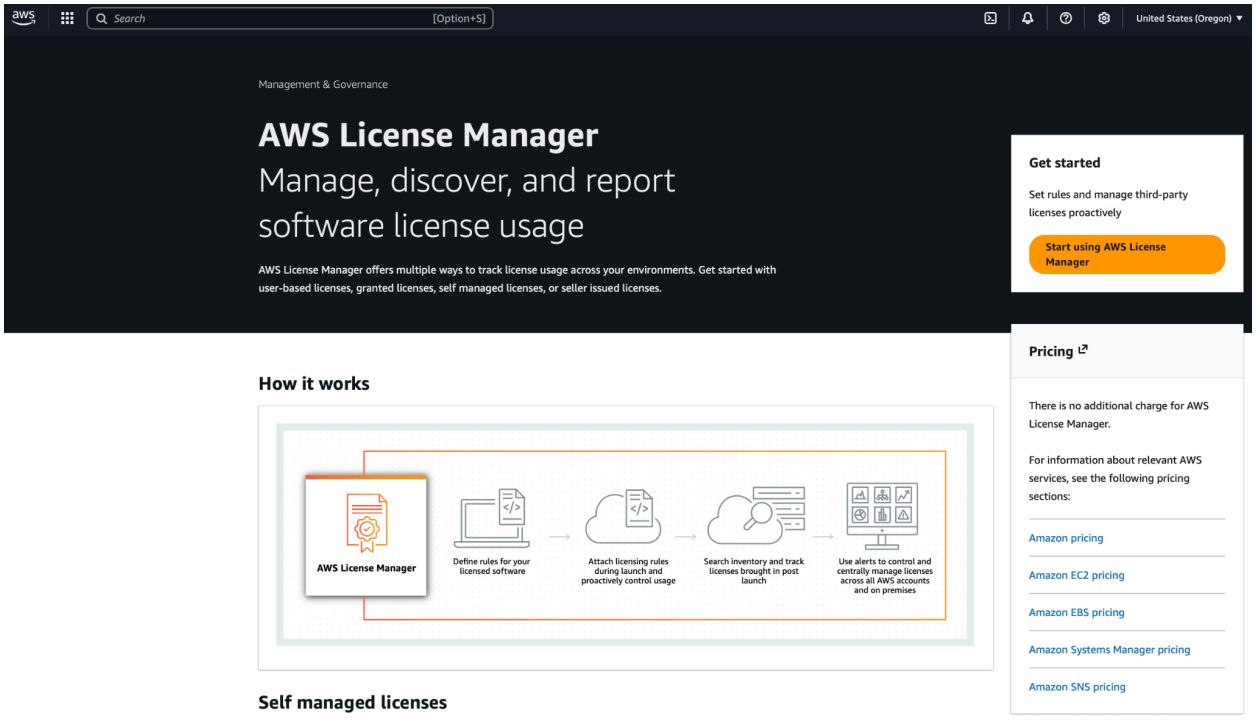
All atai-platform services are integrated with AWS License Manager and require it to be enabled and configured. Without proper License Manager setup, the product will not function. If you see the error bellow in the logs of the atai_core pods:

```
None
Error could not call LM checkout api **An error occurred
(AccessDeniedException) when calling the CheckoutLicense operation: Service
role not found. Consult setup procedures in License Manager Us
```

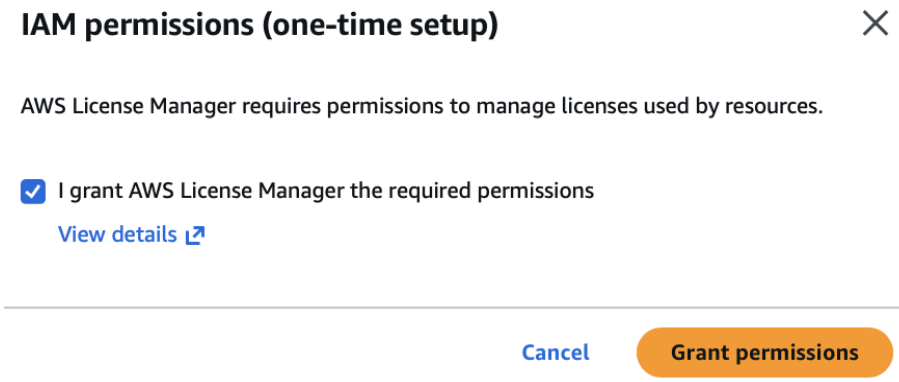
Follow the steps below to configure it and prevent service outages. To use AWS License Manager, you must first complete onboarding steps. The following procedure walks you through the onboarding steps in the AWS Management Console.

Get started with License Manager

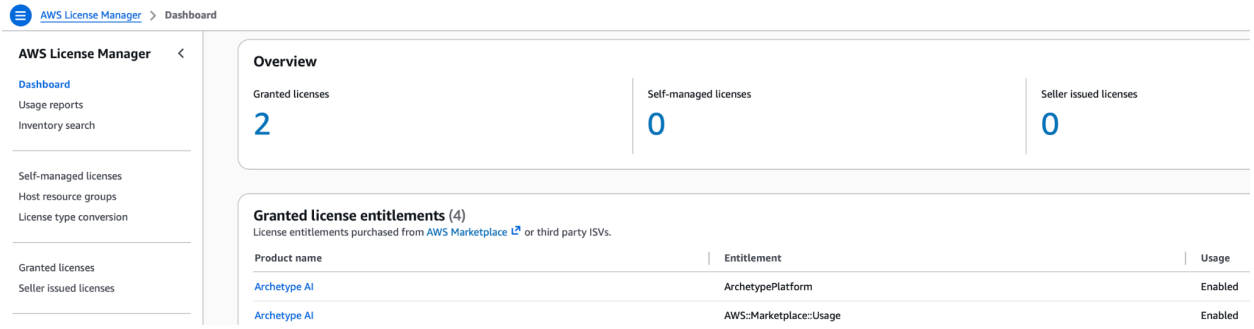
1. Open the License Manager console at https://console.aws.amazon.com/license-manager/.



2. You are prompted to configure permissions for License Manager and its supporting services. Follow the directions to configure the required permissions.



3. With the initial setup complete, you can proceed with your installation.



EKS configuration - Install the AWS Load Balancer controller

Prerequisites

Before starting this tutorial, you must complete the following steps:

1. Create an Amazon EKS cluster. To create one, see Get started with Amazon EKS
2. Install Helm on your local machine. Helm installed - Version 3.x or later
3. Make sure that your Amazon VPC CNI plugin for Kubernetes, kube-proxy, and CoreDNS add-ons are at the minimum versions listed in Service account tokens
4. The eksctl command line tool is required. For more information visit Installation options for Eksctl.
5. Version 2.12.3 or later or version 1.27.160 or later of the AWS Command Line Interface (AWS CLI) installed and configured on your device.
6. Learn about AWS Elastic Load Balancing concepts. For more information, see the Elastic Load Balancing User Guide.
7. Learn about Kubernetes service and ingress resources

Step 1: Create IAM Role using eksctl

1. Create IAM OIDC provider

```
None
eksctl utils associate-iam-oidc-provider \
    --region <region-code> \
    --cluster <your-cluster-name> \
    --approve
```

2. Download an IAM policy for the AWS Load Balancer Controller that allows it to make calls to AWS APIs on your behalf.

```
None
curl -O
https://raw.githubusercontent.com/kubernetes-sigs/aws-load-balancer-controller/
v2.14.1/docs/install/iam_policy.json
```

⚠️ If you are a non-standard AWS partition, such as a Government or China region, review the policies on GitHub and download the appropriate policy for your region.

1. Create an IAM policy using the policy downloaded in the previous step.

```
None
aws iam create-policy \
    --policy-name AWSLoadBalancerControllerIAMPolicy \
    --policy-document file://iam_policy.json
```

2. Replace the values for cluster name, region code, and account ID

```
None
eksctl create iamserviceaccount \
    --cluster=<cluster-name> \
    --namespace=kube-system \
    --name=aws-load-balancer-controller \

--attach-policy-arn=arn:aws:iam::<AWS_ACCOUNT_ID>:policy/AWSLoadBalancerControl
lerIAMPolicy \
    --override-existing-serviceaccounts \
    --region <aws-region-code> \
    --approve
```

Step 2: Install AWS Load Balancer Controller

1. Add the eks-charts Helm chart repository. AWS maintains this repository on GitHub.

```
None
helm repo add eks-charts https://aws.github.io/eks-charts
```

2. Update your local repo to make sure that you have the most recent charts.

```
None
helm repo update eks-charts
```

3. Install the AWS Load Balancer Controller.
   Add the following flags to the helm command that follows:
   a. --set region=region-code
   b. --set vpcId=vpc-xxxxxxxx
   Replace my-cluster with the name of your cluster. In the following command, aws-load-balancer-controller is the Kubernetes service account that you created in a previous step. For more information about configuring the helm chart, see values.yaml on GitHub.

```
None
helm install aws-load-balancer-controller
eks-charts/aws-load-balancer-controller \
    -n kube-system \
    --set clusterName=atai-platform \
    --set region=region-code \
    --set vpcId=vpc-xxxxxxxx \
    --set serviceAccount.create=false \
    --set serviceAccount.name=aws-load-balancer-controller \
    --version 1.14.0
```

4. The helm install command automatically installs the custom resource definitions (CRDs) for the controller. The helm upgrade command does not. If you use helm upgrade, you must manually install the CRDs. Run the following command to install the CRDs:

```
None
wget
https://raw.githubusercontent.com/aws/eks-charts/master/stable/aws-load-balance
r-controller/crds/crds.yaml
kubectl apply -f crds.yaml
```

Step 3: Verify that the controller is installed

1. Verify that the controller is installed.

```
None
kubectl get deployment -n kube-system aws-load-balancer-controller
```

An example output is as follows.

```
None
NAME                           READY   UP-TO-DATE   AVAILABLE   AGE
aws-load-balancer-controller   2/2     2            2           76s
```

EKS configuration - Install the NGINX ingress controller

Why We Use Both AWS Load Balancer Controller and NGINX Ingress Controller

Our platform uses both components because they serve different roles and work together.

1. **AWS Load Balancer Controller** - Handles AWS infrastructure provisioning
    a. Automatically creates AWS Network Load Balancers (NLB) or Application Load Balancers (ALB) when Kubernetes Services of type LoadBalancer are created
    b. Manages AWS-specific configurations (security groups, subnets, tags, target groups)
    c. Provides native AWS integration and control
2. **NGINX Ingress Controller** - Handles advanced traffic routing and request processing
    a. Provides capabilities beyond what AWS Load Balancer Controller offers for routing

In Summary

1. **AWS Load Balancer Controller** = Creates and manages the AWS infrastructure (the "front door")
2. **NGINX Ingress Controller** = Provides the routing intelligence (the "traffic director")
3. Together = Automated AWS infrastructure + powerful routing capabilities

This combination is a common pattern for production Kubernetes workloads on AWS, providing both infrastructure automation and flexible request handling.

Prerequisites

Before installing NGINX Ingress Controller, ensure you have:

1. EKS Cluster - Your Kubernetes cluster must be running and accessible
2. kubectl configured - You must be able to connect to your cluster (kubectl get nodes should work). The version can be the same as or up to one minor version earlier or later than the Kubernetes version of your cluster.
3. Install Helm on your local machine. Helm installed - Version 3.x or later
4. Public Subnets - You need the subnet IDs where the load balancer will be created
5. VPC ID - The VPC ID where your cluster is running

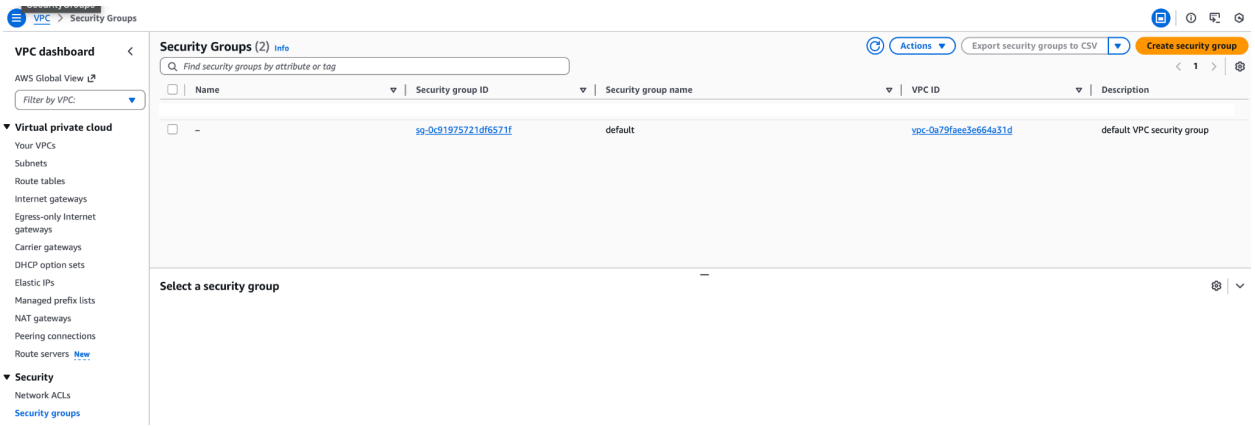
Step 1: Create Security Group for NGINX Ingress Controller

The NGINX Ingress Controller needs a security group that allows:

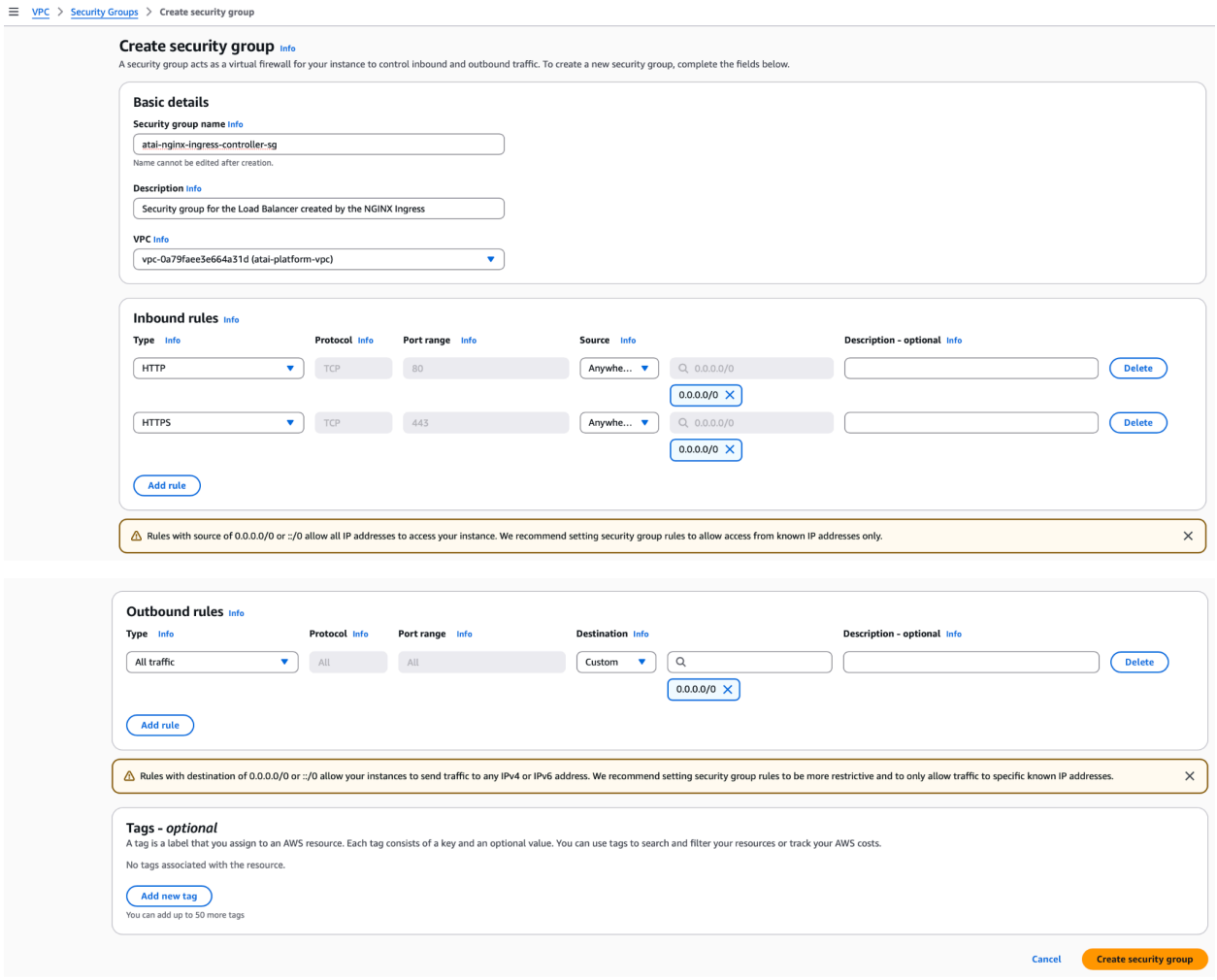
- Inbound traffic on port 80 (HTTP)
- Inbound traffic on port 443 (HTTPS)
- Outbound traffic to anywhere

To create a new security group from the AWS console use the following steps:

1. Go to VPC → Security Groups → Create security group



12. Name: atai-nginx-ingress-controller-sg (or your name)
13. Description: Security group for the Load Balancer created by the NGINX Ingress controller
14. VPC: Select your VPC from section VPC configuration Step 1
15. Inbound rules: Add rule:
    a. HTTP
        i. Type: HTTP
        ii. Port: 80
        iii. Source: Custom → Enter 0.0.0.0/0
    b. HTTPS
        i. Type: HTTPS
        ii. Port: 443
        iii. Source: Custom → Enter 0.0.0.0/0



Step 2: Add helm repository

1. Add the NGINX Ingress Controller Helm repository:

```
None
$ helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx
$ helm repo update

"ingress-nginx" has been added to your repositories
Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "ingress-nginx" chart repository
Update Complete. ⎈Happy Helming!⎈
```

2. Verify the repository was added:

```
None
$ helm repo list
NAME            URL
ingress-nginx   https://kubernetes.github.io/ingress-nginx
```

Step 3: Prepare configuration values

1. Create a file named nginx-ingress-values.yaml with the following content:

```
None
cat << EOF > nginx-ingress-values.yaml
controller:
  # Use Deployment for standard setup, or DaemonSet for high availability
  kind: Deployment

  service:
    # Use LoadBalancer type to create an AWS load balancer
    type: LoadBalancer
    # Use AWS Load Balancer Controller to manage the load balancer
    loadBalancerClass: "service.k8s.aws/nlb"
    # External traffic policy: "Cluster" for standard, "Local" for HA
(preserves source IP)
    externalTrafficPolicy: Cluster
    annotations:
      # Create a Network Load Balancer
      service.beta.kubernetes.io/aws-load-balancer-type: "nlb"
      # Make it internet-facing
      service.beta.kubernetes.io/aws-load-balancer-scheme: "internet-facing"
      # Use the security group you created in Step 1
      service.beta.kubernetes.io/aws-load-balancer-security-groups:
"sg-xxxxxxxxx"
      # Allow AWS Load Balancer Controller to manage security group rules

service.beta.kubernetes.io/aws-load-balancer-manage-backend-security-group-rule
s: "true"
      # Specify the public subnets where the load balancer will be created
      # Replace with your actual public subnet IDs, comma-separated
      service.beta.kubernetes.io/aws-load-balancer-subnets:
"subnet-xxxxxxxxx,subnet-yyyyyyyyy"
      # Optional: Name for the load balancer target group
      service.beta.kubernetes.io/aws-load-balancer-group-name:
"nginx-external-group"
  # Allow large file uploads (up to 500MB, default is 1MB)
  config:
    proxy-body-size: "500m"
EOF
```

⚠️Important: Replace the following values in the file:

- sg-xxxxxxxxx - Your security group ID from Step 1
- subnet-xxxxxxxxx,subnet-yyyyyyyyy - Your public subnet IDs of your public subnets such as atai-platform-vpc-public-us-west-2a and atai-platform-vpc-public-us-west-2b (comma-separated, no spaces)

Step 2.1 High Availability Configuration (Optional)

If you want high availability with source IP preservation, change these values in the YAML file:

```
None
controller:
  kind: DaemonSet  # Changed from Deployment
  service:
    externalTrafficPolicy: Local  # Changed from Cluster
```

Step 4: Install NGINX Ingress Controller

1. Install using helm

```
None
helm install ingress-nginx-controller-external ingress-nginx/ingress-nginx \
  --namespace ingress-nginx-controller \
  --create-namespace \
  --version 4.13.2 \
  --values nginx-ingress-values.yaml
```

Step 5: Verify installation

Step 5.1. Check Pod Status

1. Wait a few moments, then check if the pods are running:

```
None
$ kubectl get pods -n ingress-nginx-controller
```

2. You should see the NGINX Ingress Controller pod(s) in Running status:

```
None
NAME                                                  READY   STATUS
RESTARTS   AGE
ingress-nginx-controller-external-controller-xxxxxxxxx    1/1     Running   0
2m
```

Step 5.2 Check Service Status

1. Check the LoadBalancer service:

```
None
$ kubectl get svc -n ingress-nginx-controller
```

You should see a service of type LoadBalancer:

```
None
NAME                                                 TYPE
CLUSTER-IP      EXTERNAL-IP
PORT(S)                      AGE
ingress-nginx-controller-external-controller         LoadBalancer
172.20.133.73   a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6.elb.<region>.amazonaws.com
80:30495/TCP,443:31274/TCP   57m
```

⚠️Note the EXTERNAL-IP - This is the DNS name of your AWS Network Load Balancer. You'll need this for DNS configuration.

DNS configuration

After your Ingress Controller is configured, it typically creates an AWS Network Load Balancer (NLB) or Application Load Balancer (ALB) as the entry point for your services.

DNS Configuration Steps:

1. Your Ingress Controller creates an AWS load balancer (NLB or ALB)
    a. The load balancer receives a public DNS name or IP address

    If you installed the NGINX ingress controller following the instruction from Appendix [EKS configuration - Install the NGINX ingress controller], you can get the Network Load Balancer associated with your NGINX ingress controller with the command below:

```
None
$ LB_DNS=$(kubectl get svc ingress-nginx-controller-external-controller -n
ingress-nginx-controller -o
jsonpath='{.status.loadBalancer.ingress[0].hostname}')
$ echo "Load Balancer DNS: $LB_DNS"

Load Balancer DNS: a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6.elb.<region>.amazonaws.com
```

2. You need to add DNS records that point your domain names to this load balancer

Required DNS Records:

1. Add the following DNS records (A records or CNAME records) pointing to the load balancer created by your Ingress solution:
    a. **console-archetype.<your-domain>** → Points to the load balancer
    b. **api-archetype.<your-domain>** → Points to the load balancer

EKS configuration - Configure Cert-Manager and Let's Encrypt

Prerequisites

1. EKS Cluster - Your Kubernetes cluster must be running and accessible
2. kubectl configured - You must be able to connect to your cluster (kubectl get nodes should work). The version can be the same as or up to one minor version earlier or later than the Kubernetes version of your cluster.
3. Install Helm on your local machine. Helm installed - Version 3.x or later

Step 1: Add helm repository

1. Add the Cert-manager Helm repository:

```
None
helm repo add jetstack https://charts.jetstack.io
helm repo update
```

2. Verify the repository was added:

```
None
helm repo list
```

Step 2: Install Cert-manager

1. Install Cert-Manager using Helm. The chart will automatically install the required CRDs:

```
None
helm install cert-manager jetstack/cert-manager \
  --namespace cert-manager \
  --create-namespace \
  --version 1.18.2 \
  --set installCRDs=true \
  --set serviceAccount.create=true
```

Step 3: Verify installation

Step 3.1 Check Pod Status

1. Wait a few moments (30-60 seconds), then check if the pods are running:

```
None
kubectl get pods -n cert-manager
```

You'll see three pods in running status

```
None
NAME                                       READY   STATUS    RESTARTS   AGE
cert-manager-xxxxxxxxx-xxxxx               1/1     Running   0          2m
cert-manager-cainjector-xxxxxxxxx-xxxxx    1/1     Running   0          2m
cert-manager-webhook-xxxxxxxxx-xxxxx       1/1     Running   0          2m
```

Step 3.2. Check Deployment Status

1. Verify the deployments are ready:

```
None
kubectl get deployments -n cert-manager
```

All deployments should show READY 1/1:

```
None
NAME                      READY   UP-TO-DATE   AVAILABLE   AGE
cert-manager              1/1     1            1           6m33s
cert-manager-cainjector   1/1     1            1           6m33s
cert-manager-webhook      1/1     1            1           6m33s
```

Step 3.3 Verify Cert-Manager is Working

1. Test that Cert-Manager can create resources:

```
None
# Check if ClusterIssuer CRD is available
kubectl api-resources | grep cert-manager
# You should see resources like:
# certificates, certificaterequests, challenges, clusterissuers, issuers,...
```

Step 4: Create Cluster Issuer

Understanding Issuers and ClusterIssuers

Cert-Manager supports two types of certificate issuers:

- **Issuer** - Scoped to a specific namespace. Can only issue certificates for resources in that namespace.
- **ClusterIssuer** - Cluster-wide. Can issue certificates for any namespace in the cluster.

For most use cases, use a ClusterIssuer so you can issue certificates across all namespaces without creating multiple issuers.

Understanding Certificate Challenges

To obtain SSL certificates from Let's Encrypt, Cert-Manager must prove domain ownership. There are two challenge types:

- **HTTP-01 Challenge**:
    - Let's Encrypt makes an HTTP request to your domain
    - Cert-Manager creates a temporary file that Let's Encrypt can access
    - Requires your domain to be publicly accessible and point to your load balancer
    - Simpler to set up
    - Requires DNS to be configured first (domain must point to your load balancer). You must complete the DNS configuration before creating the ClusterIssuer.
- **DNS-01 Challenge:**
    - Let's Encrypt verifies domain ownership via DNS records
    - Cert-Manager creates a TXT record in your DNS provider
    - Works even if your domain isn't fully accessible yet
    - More complex (requires API access to your DNS provider)
    - Better for wildcard certificates

For simplicity, we'll use HTTP-01 challenge, which requires:

1. Your domain names (console.archetype.<your-domain> and api.archetype.<your-domain>) must have DNS records pointing to your NGINX Ingress Controller's load balancer. For more information about the DNS configuration visit the Appendix DNS configuration before creating the ClusterIssuer.
2. The load balancer must be publicly accessible
3. Port 80 (HTTP) must be open for Let's Encrypt validation

Let's Encrypt Servers

Let's Encrypt provides two ACME servers:

- **Staging Serve**r (https://acme-staging-v02.api.letsencrypt.org/directory)
    - For testing
    - Higher rate limits (good for testing)
    - Certificates are not trusted by browsers (you'll see a warning)
    - Use this first to test your configuration

- **Production Server** (https://acme-v02.api.letsencrypt.org/directory)
    - For production use
    - Lower rate limits (50 certificates per registered domain per week)
    - Certificates are trusted by browsers
    - Use this after testing with staging
    - Recommendation: Start with the staging server to test, then switch to production once everything works.

Step 4.1. DNS configuration

You must add the following DNS records (A records or CNAME records) pointing to the load balancer created by your Ingress solution:

c. **console-archetype.<your-domain>** → Points to the load balancer
d. **api-archetype.<your-domain>** → Points to the load balance

For more information visit the Appendix DNS configuration before creating the ClusterIssuer.

Step 4.2 Create the Cluster Issuer

1. Create a ClusterIssuer YAML file. For HTTP-01 challenge, create cluster-issuer-http.yaml:

```
None
cat << EOF > cluster-issuer-http.yaml
apiVersion: cert-manager.io/v1
kind: ClusterIssuer
metadata:
  name: letsencrypt-stage
  namespace: cert-manager
spec:
  acme:
    # Let's Encrypt staging server (use for testing)
    server: https://acme-staging-v02.api.letsencrypt.org/directory
    # Your email address (Let's Encrypt will send expiration notices here)
    email: your-email@example.com
    # Private key secret name (Cert-Manager will create this)
    privateKeySecretRef:
      name: letsencrypt-stage
    # HTTP-01 challenge solver
    solvers:
    - http01:
        ingress:
          class: nginx
EOF
```

⚠️Important: Replace the following values in the file:

- your-email@example.com - Your DNS administrator email address

2. Apply the cluster issuer to the cluster

```
None
kubectl apply -f cluster-issuer-http.yaml
```

Step 4.3 Verify the cluster issuer

1. Check that the ClusterIssuer was created:

```
None
kubectl get clusterissuer -n cert-manager
```

You should see:

```
None
NAME                READY   AGE
letsencrypt-stage   True    4m47s
```

2. Check details status

```
None
kubectl describe clusterissuer letsencrypt-stage -n cert-manager
```