

Course 4: Designing a Cloud-Native Architecture with Kubernetes, Helm & Identity Federation

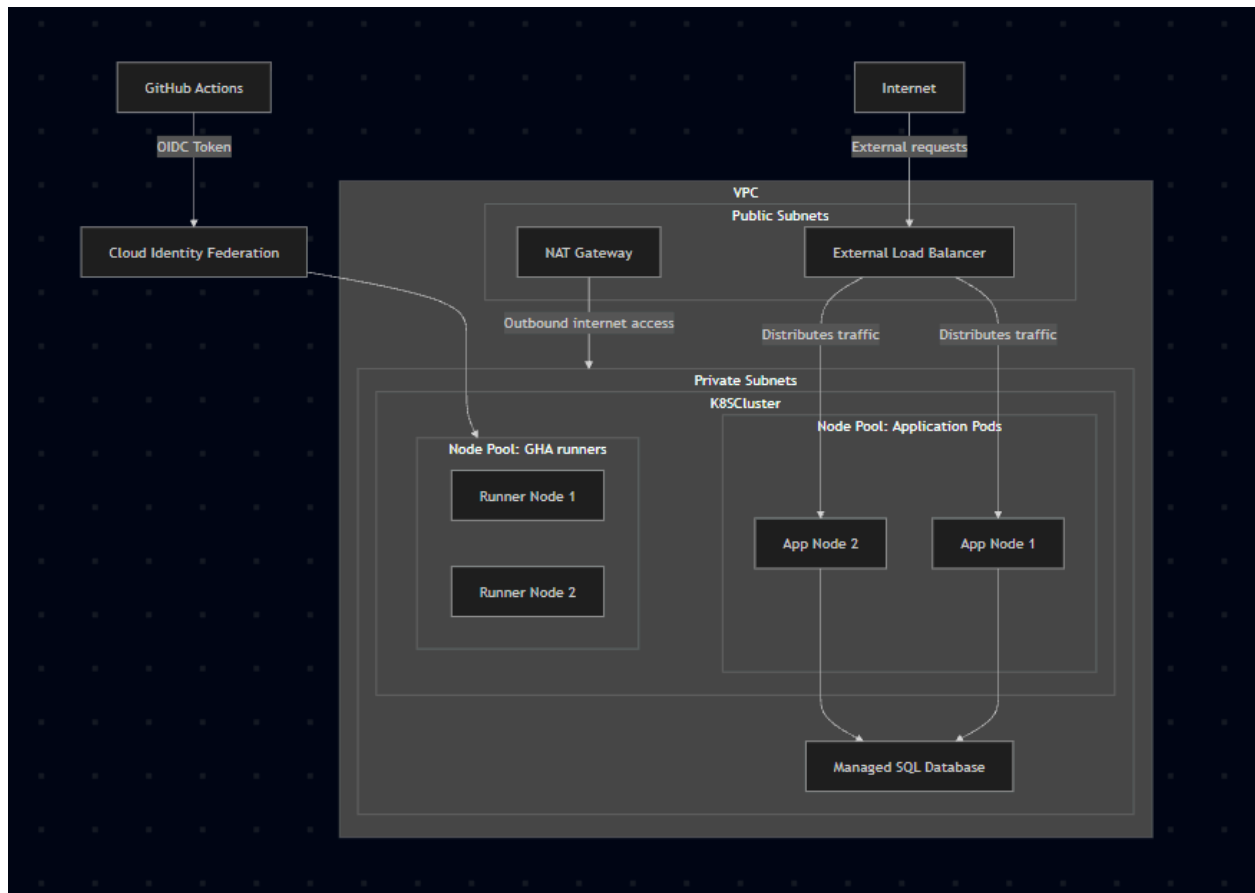
1. Infrastructure Overview

Before starting a project, you should define the high-level architecture. Your diagram may evolve during the development of the project and that's normal but you need to maintain it until the end: you'll use your diagram to present your infrastructure and to justify your choices during your defense.

- Your diagram should contains (none exhaustive list and may depend on your technical and provider choices) :
 - VPC
 - Subnets
 - Load balancers
 - NAT
 - K8S cluster
 - Node Pool(s) for GitHub Actions self-hosted runners
 - Node Pool(s) for application pods
 - Managed SQL database
 - Identity Federation flow for authentication
 - Traffic flows + protocols
 - Github/Developer/pipeline

1.1 Infrastructure Diagram

Here a simple and none-exhaustive example of an Cloud agnostic diagram for the infra we want to build:



⚠ Your final diagram must not be Cloud agnostic but specific to your Cloud Provider and the more exhaustive as possible.

⚠ This diagram is purposely incomplete, naive and simple. You have to do your own diagram !

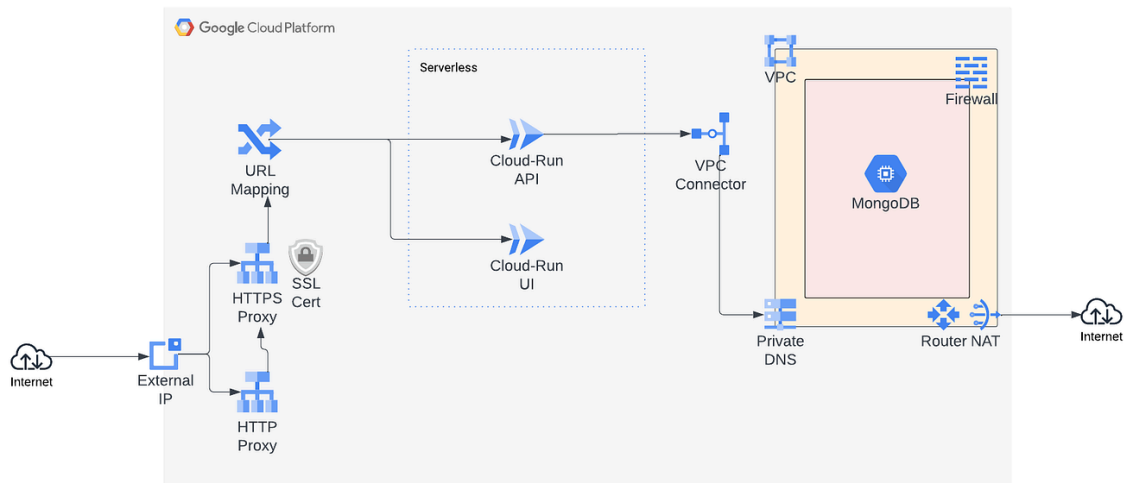
👉 Once you've acknowledged this document and have a big picture of your infrastructure: Create your own diagram with your team.

👉 You should look for GCP/AWS infrastructure diagram templates on the internet instead of creating it from scratch.

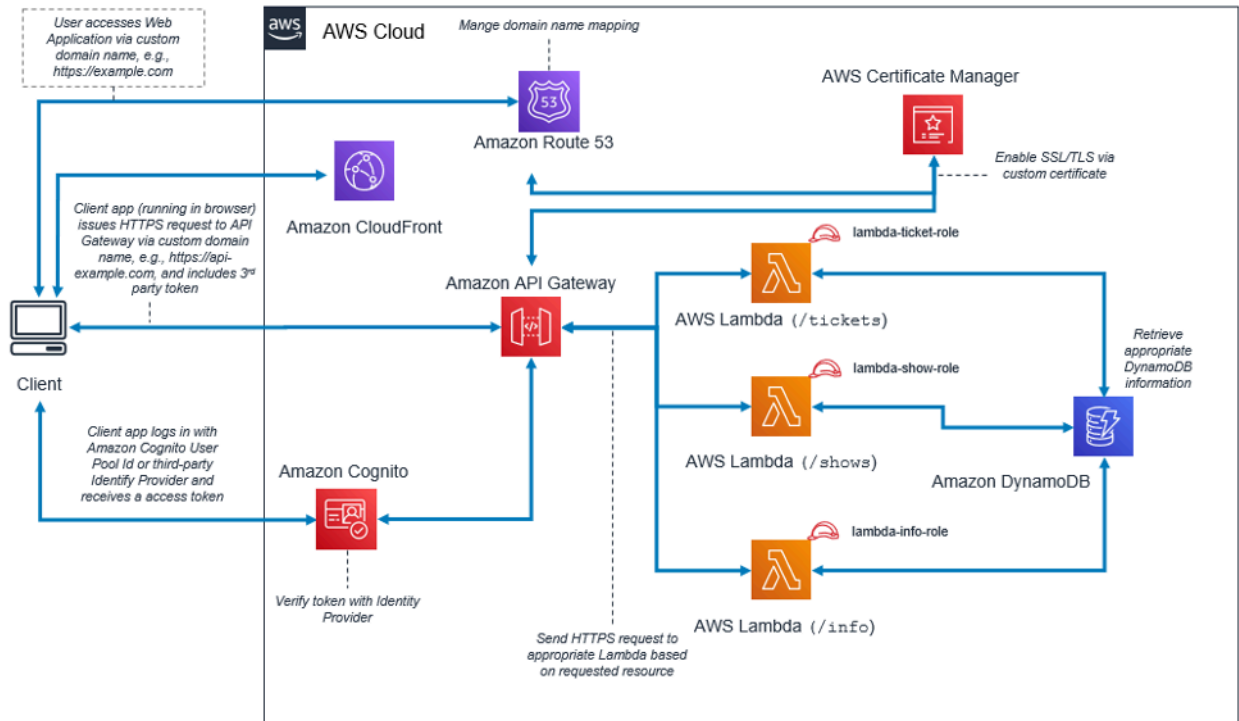
1.1 Advanced Infrastructure Diagram examples

Few examples of architecture diagrams that are not related to this project, so you can see what a cloud specific diagram could look like:

GCP:



AWS:



2. Helm

Helm is the **required tool** for deploying both **self-hosted runners** and your **custom application**. Understanding Helm properly is crucial for reproducible and maintainable Kubernetes deployments.

2.1 Helm Overview

- Helm is a **package manager for Kubernetes**, similar to apt or yum.
- Helm packages are called **charts**, which bundle Kubernetes manifests and templates.
- Charts can be **official**, **community-provided**, or **custom-built**.

2.2 Finding and Using Charts

- Official charts: Artifact Hub or cloud vendor repositories (Bitnami, Helm stable charts).
- Community charts: GitHub repositories or Helm chart hubs.
- Custom charts: You will create one for your application.

Example: Using a PostgreSQL chart from Bitnami:

```
helm repo add bitnami https://charts.bitnami.com/bitnami
helm repo update
helm install my-postgres bitnami/postgresql --values values.yaml
```

2.3 The `values.yaml` File

- `values.yaml` defines configuration parameters for your chart.
- Override default settings without modifying templates.
- Example `values.yaml` for an app:

```
replicaCount: 3
image:
  repository: myapp
  tag: "1.0.0"
service:
  type: ClusterIP
  port: 8080
resources:
  limits:
    cpu: "500m"
    memory: "512Mi"
```

- Use `values.yaml` to:
 - Configure **replica count**
 - Set **container image and tag**
 - Define **service type and ports**
 - Specify **resource limits/requests**

2.4 Terraform Integration

- Helm releases can be managed with Terraform using the `helm_release` resource.
- Example:

```
resource "helm_release" "myapp" {  
  name      = "myapp"  
  repository = "https://charts.myorg.com/"  
  chart     = "myapp"  
  version   = "1.0.0"  
  values    = [file("values.yaml")]  
}
```

- Benefits of Terraform + Helm:
 - Keep **infrastructure as code** consistent
 - Track releases and upgrades
 - Integrate deployment into CI/CD pipelines

2.5 Best Practices

- Always **version your charts** to prevent accidental upgrades.
- Use `values.yaml` and terraform variables to adapt the same chart to **different environments**.
- Prefer official/community charts when possible; create custom charts only for unique requirements.

3. GitHub Actions Self-Hosted Runners

3.1 Why Use Self-Hosted Runners

- GitHub free minutes are limited.
- Self-hosted runners allow **offloading CI/CD jobs** to your cluster.

3.2 Deploying Runners

- Deploy **Github Actions runners** using **Helm charts**.
- Runners should **register dynamically** and **unregister** after jobs complete.
- Your runners must scale automatically. ⚠ Be careful when doing auto-scaling, an incorrect configuration could lead you to consume all your Cloud providers free credits quickly.

3.3 Managing Free Minutes and Load

- Track your GitHub free minutes.
- Decide which jobs run on **GitHub-hosted** vs. **self-hosted runners**.
- Use **labels** in workflows:

```
jobs:
  terraform:
    runs-on: [self-hosted, linux, terraform]
```

3.4 Smart Runner Switching

- Avoid committing workflow changes to switch runners.
- Use **dynamic labels**, environment variables, or workflow dispatch inputs:

```
jobs:
  deploy:
    runs-on: [self-hosted, ${ inputs.runner_label }]
```

- There are multiple ways to automate/manage the runners switching. Try to find a proper and convenient one.

3.5 Best Practices

- Use an official or community maintained helm chart.
 - Scale your runners automatically and set up safeguards to avoid uncontrolled costs.
 - Log runner activity for auditing and debugging.
 - Clean up old runners to avoid stale registrations.
-

4. Identity Federation (GitHub Actions → Cloud Provider)

4.1 Why Identity Federation?

- Avoid long-lived credentials.
- Use short-lived credentials issued by the cloud provider.

4.2 How It Works

- GitHub Actions authenticates using an **OIDC token**.
- The cloud provider validates the token and issues **temporary credentials** for job duration.
- This ensures that **only authorized jobs** can request temporary credentials.

4.3 Security

You should ensure that you will generate temporary credentials only if you can guarantee that the token is coming from your Github repository and is using the correct environment.

4.4 Setup

Since WIF is specific to your Cloud provider, you should refer to Github and cloud providers documentation. You will find tons of official documentation and unofficial labs to set up your Identity Federation.

5. The Application

You can choose any language/framework for this application, however, some languages may be more appropriate for this use case 🤔.

Requirements:

- All traffic must be using safe protocols.
- Stateless: no local file persistence.
- REST API exposing CRUD endpoints.
- Interact with a managed SQL database.
- Scale horizontally with multiple replicas.
- Handle **unordered requests** using `request_timestamp`.
- Each task must include `title`, `content`, `due_date` and `done(bool)` fields.
- **Deploy the application within your cluster using Helm/Terraform**, with a **custom chart**.
- Your application must be available on the internet and enforce an authentication.
- It must follow the security best practices.

HTTP Codes to Use:

- **200 OK** → Successful GET/PUT/DELETE
- **201 Created** → Successful POST
- **400 Bad Request** → Invalid request body
- **401 Unauthorized** → Missing or invalid authentication
- **404 Not Found** → Resource not found
- **409 Conflict** → Timestamp/concurrency/duplicate conflict
- **429 Too Many Requests** → Cluster temporarily overloaded
- **500 Internal Server Error** → Unexpected server failures

Task Manager API

Method	Endpoint	Description	Body Example	Header Example
POST	/tasks	Create a new task	<pre>{ "title": "Write", "content": "Prepare lesson", "due_date": "2025-09-30", "request_timestamp": "2025-09-25T20:00:00Z" }</pre>	<pre>correlation_id: abc123 Authorization: Bearer <token></pre>
GET	/tasks	List all tasks	—	<pre>correlation_id: abc124 Authorization: Bearer <token></pre>

GET	/tasks/{id}	Get a specific task	—	correlation_id: abc125 Authorization: Bearer <token>
PUT	/tasks/{id}	Update a task	{ "title": "Review", "content": "Check slides", "done": true, "request_timestamp": "2025-09-25T20:01:00Z" }	correlation_id: abc126 Authorization: Bearer <token>
DELETE	/tasks/{id}	Delete a task	{ "request_timestamp": "2025-09-25T20:02:00Z" }	correlation_id: abc127 Authorization: Bearer <token>

Processing Logic Notes:

- Requests may arrive **out-of-order**.
 - Process updates/deletes **according to request_timestamp**.
 - Requests will always include **correlation_id** in the header to trace requests for debugging.
-

6. Monitoring & Observability

6.1 Goals

- Monitor cluster health, runner usage, and app performance.
- Debug your app.
- Detects bottlenecks in runners or app pods.

6.2 Suggested Tools

You are free to use any monitoring tools that you will consider relevant, however, here the most common tools to achieve that:

- **Prometheus** → Metrics collection
- **Grafana** → Dashboards/Alerting
- **OpenTelemetry** → Distributed tracing

6.3 Monitoring Requirements & Best Practices

- You may choose any monitoring tools but must justify your choice.
 - Track self-hosted runner pod resource usage.
 - Monitor application pods: CPU, memory, request latency, error rates.
 - Trace failed requests.
 - Set up alerts for scaling, failures, or high latency.
 - Include dashboards and alerts for **quick debugging**.
 - Maintain historical metrics for **trend analysis** and scaling planning.
-

7. Key Takeaways

- Deploy and manage **self-hosted GitHub runners** via Helm.
 - Build a **scalable CRUD app** within your cluster
 - Use **Helm** for deploying **both runners and app**.
 - Implement **monitoring and observability**.
 - Implement **Identity Federation**
-

8. Project Defense

8.1 Repository

- All students must work on the **same repository**.
- This repository will be the **reference** for your project defense.

8.2 GCP / AWS

- Your **production environment** will also serve as the **reference** for your project defense.

8.3 Community / Open Source




In a professional environment, you will often rely on **open-source components**.

In this project, you will have many opportunities to use some of them.

You are allowed to use open-source tools for:

- GitHub Actions
- Helm charts
- Terraform providers/modules
- Libraries for your application

Rules:

-  You must be able to **justify why** you are using it.
-  You must be able to **explain what it does**.
-  It must **not introduce security issues**.

8.4 Evaluation Criteria

During your project defense, you will be evaluated on:

Infrastructure

- Your **understanding** of the overall architecture.
- The **clarity and accuracy** of your infrastructure diagram.

Repository

- **Branching model** and quality of commits.
- Use of **GitHub Actions** and **self-hosted runners**.
- Adoption of **GitFlow** and management of pull requests.
- Handling of **issues** and **secrets management**.

Infrastructure, Application & IaC

- **Security:** IAM, least privilege, secrets, access.
- **Quality & maintainability:** clean, documented code.
- **Scalability:** ability of the system to handle growth.
- **Operability:** monitoring, debugging, and resilience.
- **Cost awareness:** smart use of resources (GitHub minutes, cloud costs, etc.).

Teamwork

- Your ability to **collaborate effectively** with group members.

Workflow

- How well you can explain and demonstrate your **workflow** (from your favorite IDE to production).