# AWS-base High-level project
# Host a containerized Application:

Utilizes radis cache, and an RDS Postgres Aurora DB

Version 1.1.1


**BMV SYSTEM INTEGRATION PRIVATE LIMITED**

**Idea... Implementation... Innovation...**

**:: CORPORATE HEAD OFFICE::**

**A503, The First,**
**Behind The ITC Narmada Hotel & Keshavbaug Party Plot,**
**Off 132 ft Road, Vastrapur, Ahmedabad.**
**Gujarat- 380015**
**Phone: +91 (79) 40 30 53 02**
**Website: www.systemintegration.in**
**Mail: info@systemintegration.in**

System Integration™

# Table of Content

# Introduction to the project

## 1.1 Overview

In the realm of modern software development, containerization has emerged as a pivotal technology, offering scalability, portability, and resource efficiency. As organizations transition towards a microservices architecture, the need for robust hosting environments becomes increasingly vital. This project aims to design a high-level architecture on Amazon Web Services (AWS) to host a containerized application composed of four microservices, accompanied by a Redis cache and an RDS Postgres Aurora database.

## 1.2 Objectives, Goals, and Benefits

### Objectives and Goals:

The primary objective of this project is to establish a resilient and scalable infrastructure on AWS that can effectively host containerized microservices while ensuring optimal performance and reliability. Specific goals include:

1. Designing a modular architecture capable of accommodating four containerized microservices, each fulfilling distinct functionalities within the application.
2. Implementing a Redis cache to enhance application performance by providing fast and efficient data retrieval.
3. Integrating an RDS Postgres Aurora database to manage persistent data storage with high availability and fault tolerance.
4. Optimizing resource allocation and scalability to support fluctuating workloads and ensure consistent performance under varying conditions.
5. Establishing robust monitoring, logging, and deployment practices to facilitate efficient management and troubleshooting of the environment

**Benefits:**

1. **Scalability:** With AWS Fargate, the containers automatically scale based on demand, allowing the application to handle sudden spikes in traffic without manual intervention. This ensures consistent performance and responsiveness for users.
2. **Cost-Efficiency:** Leveraging AWS services enables cost optimization by paying only for the resources utilized. Fargate's pay-as-you-go model eliminates the need to provision and manage infrastructure, reducing idle resource costs and overall expenses.
3. **High Availability:** By deploying services across multiple Availability Zones (AZs), the architecture ensures high availability and fault tolerance. In the event of a failure in one AZ, traffic is automatically routed to healthy instances in other AZs, minimizing downtime and ensuring uninterrupted service delivery.
4. **Managed Services:** AWS manages the underlying infrastructure, including server provisioning, networking, and security updates. This offloads operational overhead from the development team, allowing them to focus on building and improving the application rather than managing infrastructure.
5. **Security:** AWS provides robust security features such as Virtual Private Cloud (VPC) and security groups to enforce network isolation and control access to resources. This ensures that sensitive data remains protected from unauthorized access and potential security threats, maintaining the integrity and confidentiality of the application.

# Components Overview

## 2.1 Amazon Radis Cache

Amazon ElastiCache Redis is a fully managed, in-memory data store service provided by AWS. It is used as a caching layer to improve the performance and scalability of the application by storing frequently accessed data in memory. Redis offers sub-millisecond response times, making it ideal for use cases where low-latency access to data is crucial.

## 2.2 Amazon RDS Postgres Aurora

Amazon RDS (Relational Database Service) Postgres Aurora is a fully managed relational database engine compatible with PostgreSQL. It is designed for high performance, scalability, and availability. Aurora provides features such as automatic backups, automated failover, and read replicas, making it suitable for critical production workloads requiring a highly available and durable database solution.

## 2.3 Docker Image

Docker images are lightweight, standalone, executable packages that contain everything needed to run a piece of software, including the code, runtime, libraries, and dependencies. In this project, Docker images are used to package the microservices composing the application. These images can be easily deployed to Amazon ECS or EKS for execution within containers.

## 2.4 Amazon ECR(Elastic Container Registry)

Amazon ECR is a fully managed Docker container registry provided by AWS. It allows you to store, manage, and deploy Docker images securely. In this project, ECR is used to store the Docker images created for the microservices. ECR integrates seamlessly with Amazon ECS, enabling you to easily deploy containerized applications.

## 2.5 Amazon ECS

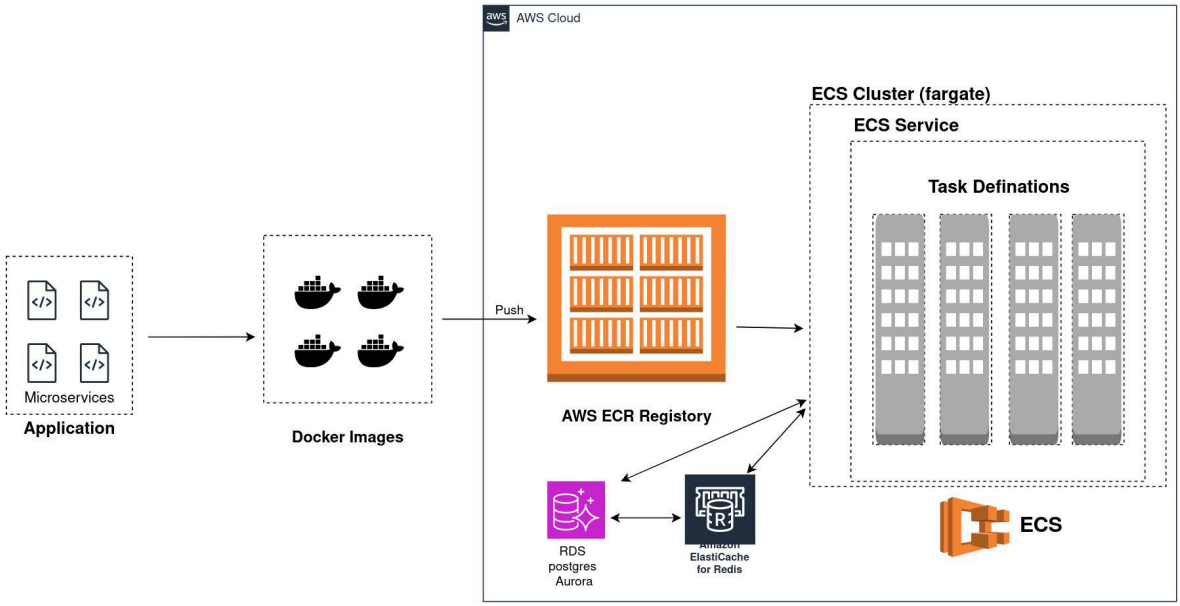
Amazon ECS is a fully managed container orchestration service provided by AWS. It allows you to run, stop, and manage Docker containers on a cluster of EC2 instances or AWS Fargate. ECS simplifies the deployment and management of containerized applications by providing features such as task definitions, service definitions, and load balancing.

### 2.6 Security Groups

Security groups act as virtual firewalls for your AWS resources. They control inbound and outbound traffic to and from instances, allowing you to specify the protocols, ports, and IP ranges that are allowed to access your resources. In this project, security groups are used to enforce network isolation and control access to the containerized applications, Redis cache, and RDS Postgres Aurora database instances.

## Project Workflow

### 3.1 Diagram



### 3.2 Deployment Workflow

1. **Develop Microservices:**
    - Develop the four microservices that constitute the application. Each microservice should encapsulate a specific functionality and expose APIs for communication with other microservices.
2. **Configure RDS PostgreSQL Aurora:**
    - Set up an Amazon RDS PostgreSQL Aurora database instance. Configure the database parameters, such as instance size, storage, and security settings, according to the requirements of the application.
3. **Configure Redis Cache:**
    - Set up an Amazon ElastiCache Redis cluster to serve as a caching layer for the application. Configure the Redis cluster with appropriate settings, such as node type, replication, and security groups.
4. **Containerize Microservices:**
    - Dockerize each microservice by creating Dockerfiles that define the environment and dependencies required to run the microservice. Include the necessary configurations and dependencies, such as libraries, frameworks, and runtime environments.
5. **Build Docker Images:**
    - Build Docker images for each microservice using the Dockerfiles created in the previous step. This process packages the microservices along with their dependencies into lightweight, portable containers.
6. **Push Docker Images to Amazon ECR:**
    - Create an Amazon Elastic Container Registry (ECR) repository for storing Docker images. Push the Docker images for each microservice to the respective repositories in ECR. Ensure that the appropriate permissions are configured to allow ECS to pull images from the ECR repositories.
7. **Define Task Definitions:**
    - Define task definitions for each microservice in Amazon ECS. Task definitions specify the configuration for running containers, including Docker image, CPU/memory resources,

networking, and environment variables. Configure the task definitions to use the Docker images stored in the ECR repositories.

8. **Create ECS Cluster:**
    - Set up an Amazon ECS cluster to host the containerized microservices. Configure the cluster with appropriate instance types, networking settings, and security groups. Launch ECS container instances within the cluster to run the microservice containers.
9. **Create ECS Services:**
    - Create ECS services for each microservice, specifying the task definitions to use and the desired number of tasks to run. ECS services manage the lifecycle of tasks, ensuring that the specified number of tasks are running and handling load balancing across container instances.
10. **Configure Load Balancer:**
    - Set up an Elastic Load Balancer (ELB) or Application Load Balancer (ALB) to distribute incoming traffic among the ECS services. Configure the load balancer to route traffic to the appropriate microservice based on the request path or host header.

## Start a Project: Host an Application by AWS ECS

### Prerequisites:

- **AWS CLI Configuration:** Before proceeding, ensure that AWS CLI is installed and configured on your local machine with appropriate IAM user credentials having the necessary permissions to access AWS services like Amazon RDS, ElastiCache, Amazon ECR, and Amazon ECS.
- **Install Docker:** Make sure Docker is installed on your local machine. Docker will be used to containerize the microservices and build Docker images.
- **Sign in to AWS Account:** Sign in to your AWS Management Console using your AWS account credentials. This is required

for setting up and configuring AWS services mentioned in the deployment workflow.

- **Microservices Utilizing Redis and PostgreSQL Aurora:** Ensure that the microservices you're deploying are already developed and configured to utilize Redis cache and PostgreSQL Aurora database for their functionalities. This includes having the necessary connection parameters and code logic to interact with these services.

### 4.2 Create Docker Images of each microservices

To create a Docker image using this Dockerfile, follow these steps:

1. Save the Dockerfile to a directory along with your application code.
    i. `FROM node:14`
    ii. `WORKDIR /app`
    iii. `COPY package*.json ./`
    iv. `RUN npm install --production`
    v. `COPY . .`
    vi. `EXPOSE 3001`
    vii. `CMD ["node", "index.js"]`
2. Open a terminal or command prompt.
3. Navigate to the directory containing the Dockerfile and your application code.
4. Run the following command to build the Docker images

```
docker build -t <your-image-name> .
```

Replace your-image-name with the desired name for your Docker image.

5. Once the image is built successfully, you can run it using:

```
docker run -p 3001:3001 <your-image-name>
```

This command will run your Docker container, mapping port 3001 of the container to port 3001 on your host machine.

Remember to replace your-image-name with the name you specified when building the Docker image.

Now you have a Docker image of your microservice ready to be deployed to AWS ECS. Repeat the above steps for each microservice in your application, adjusting the Dockerfile and application code accordingly for each microservice.

### 4.3 Create an ECR registry and push Docker images into it

#### 1. Create AWS ECR Repository:

Use the AWS CLI to create an ECR repository:

```
aws ecr create-repository --repository-name your-repository-name --region your-region-name
```

#### 2. Obtain Repository URL:

The repository URL is required for tagging and pushing Docker images. You can obtain it from the output of the previous command or by using the AWS CLI:

```
aws ecr describe-repositories --repository-name your-repository-name --region your-region-name --query 'repositories[0].repositoryUri' --output text
```

#### 3. Push Docker Image into ECR Repository:

Follow these commands to push your Docker image into the ECR repository:

```
# Obtain authentication token
token=$(aws ecr get-login-password --region your-region-name)

# Add the current user to the docker group (if not already added)
sudo usermod -aG docker $USER

# Authenticate Docker with ECR
aws ecr --region your-region-name | sudo docker login -u AWS -p $token your-repository-url

# Tag your Docker image with the ECR repository URL
sudo docker tag your-image-name:latest your-repository-url:latest

# Push the tagged Docker image to the ECR repository
sudo docker push your-repository-url
```

**Repeat for Each Image:**

Repeat the above steps for each Docker image you want to push into the ECR repository, ensuring you use the correct repository URL and image names each time.

### 4.4 Create ECS Cluster and tasks for each Docker image

1. **Create ECS Cluster**:
   - Navigate to the ECS console.
   - Click "Create cluster" and follow the wizard to create a new ECS cluster.
2. **Create Task Definitions**:
   - Navigate to "Task Definitions" in the ECS console.
   - Click "Create new Task Definition" and configure your task definition settings.

    - Add container definitions for your Docker images, specifying the ECR repository URI and tag.
3. **Run Tasks in ECS Cluster**:
    - Navigate to the "Clusters" section in the ECS console.
    - Select your cluster and click "Run new task".
    - Choose the Fargate launch type and select the task definition you created.
    - Configure task size and networking settings, then run the task.
4. **Repeat for Each Docker Image**:
    - Follow the above steps for each Docker image you want to deploy in the ECS cluster, creating separate task definitions for each image.

With this comprehensive documentation, you are equipped to seamlessly deploy your containerized application, which comprises multiple microservices, in an AWS ECS environment. By following the outlined steps and configurations, you can efficiently set up your AWS-based infrastructure, including containerization, database, cache, deployment workflow, and monitoring. This documentation serves as a guide to ensure a smooth deployment process, facilitating scalability, high availability, and efficient management of your application on AWS. With the provided instructions and guidelines, you can confidently navigate through the complexities of deploying and managing your containerized application, empowering you to leverage the full potential of AWS ECS for your project's success.