1. Architecture Overview

The solution will be divided into three main components:

1. **Java Spring Backend Service**: A Spring Boot application that provides APIs.
2. **React Frontend Service**: A frontend application built with React.js.
3. **Infrastructure Deployment**: Using Azure to deploy and manage resources required for the application.

Azure Resources Involved

1. **Azure Kubernetes Service (AKS)**: To host the Java Backend and React Frontend containers.
2. **Azure Container Registry (ACR)**: To store the built Docker images for the backend and frontend.
3. **Azure App Gateway or Load Balancer**: To route traffic to the appropriate services.
4. **Azure PostgreSQL Database**: Replacing the in-memory database with a managed database.
5. **Azure Monitor**: To enable logging, monitoring, and alerts.
6. **Azure Key Vault**: To securely store and access secrets, connection strings, and sensitive information.

2. CI/CD Pipelines

Pipeline 1: Backend Service CI/CD

- **CI**:
    1. Trigger on code commit or pull request.
    2. Checkout the code and run tests (`mvn test`).
    3. Build the Docker image (`docker build -t <backend_image> .`).
    4. Push the Docker image to **Azure Container Registry** (`docker push <backend_image>`).
- **CD**:
    1. Deploy to AKS using `kubectl` or `helm`.
    2. Manage different environments (Dev, QA, Prod) by applying different configurations.

Pipeline 2: Frontend Service CI/CD

- **CI**:
    1. Trigger on code commit or pull request.
    2. Build the React project using `npm install` and `npm run build`.
    3. Build the Docker image for the frontend (`docker build -t <frontend_image> .`).
    4. Push the Docker image to **Azure Container Registry** (`docker push <frontend_image>`).
- **CD**:
    1. Deploy the built frontend image to AKS.
    2. Ensure proper routing via Ingress or Load Balancer.

Pipeline 3: Infrastructure Deployment

- **IaC (Infrastructure as Code)**:
    1. Use **Azure Resource Manager (ARM) templates** or **Terraform** scripts to define the infrastructure.

2. Deploy the following components:
    - AKS Cluster
    - ACR
    - PostgreSQL Database
    - App Gateway or Load Balancer
    - Azure Monitor setup
    - Azure Key Vault

- **CD**:
    1. Use a pipeline to apply the ARM/Terraform configurations.
    2. Implement `terraform destroy` or `az group delete` commands for cleanup.

3. Step-by-Step Implementation

Step 1: Infrastructure as Code (IaC)

Azure Resource Manager (ARM) templates provide a declarative way to define infrastructure, making it easier to manage and deploy Azure resources consistently. For this assignment, we'll create ARM templates for the following components:

1. **Resource Group**: A container to hold all the Azure resources.
2. **Azure Container Registry (ACR)**: To store Docker images for the frontend and backend applications.
3. **Azure Kubernetes Service (AKS)**: To host the containers and deploy the application.
4. **Azure PostgreSQL Database**: As a managed database service to replace the in-memory database.
5. **Azure Key Vault**: To store secrets and sensitive configuration.
6. **Azure Monitor**: To enable logging, monitoring, and alerts.

1. Create the ARM Templates

The solution will include multiple ARM templates to set up the infrastructure, and a master `azuredeploy.json` template to link them together.

2. Master Template: `azuredeploy.json`

The `azuredeploy.json` template will link the individual ARM templates using the **`linkedTemplates`** approach, making it easier to manage and reference dependencies between different resources.

3. ARM Template for Each Resource

3.1 Azure Container Registry Template: `acr-template.json`

This template creates an Azure Container Registry to store Docker images.

3.2 Azure Kubernetes Service Template: `aks-template.json`

This template creates an AKS cluster and connects it to ACR for image pulling.

3.3 Azure PostgreSQL Template: `postgresql-template.json`

Deploys a managed PostgreSQL database instance.

4. Key Vault Template

This Key Vault will be used to securely store and access sensitive information such as database credentials, client secrets for the AKS service principal, and other application secrets.

Key Vault Template Details

1. **`keyVaultName`**: Name of the Key Vault instance.
2. **`location`**: Azure region where the Key Vault will be created.
3. **`objectId`**: Azure Active Directory object ID (service principal or managed identity) that will be granted access to the Key Vault. For AKS, this can be the service principal's Object ID.
4. **`dbAdminUsername`** and **`dbAdminPassword`**: Credentials for the PostgreSQL database that will be stored in the Key Vault.
5. **`clientSecret`**: The client secret for the AKS service principal.

5. Monitor Template

`monitor-template.json` file that sets up **Azure Monitor** resources for logging, monitoring, and alerting. It includes a **Log Analytics workspace** and **Application Insights** to monitor the deployed infrastructure and application components, providing detailed insights into performance, availability, and overall health.

Monitor Template: `monitor-template.json`

This template creates the following resources:

1. **Log Analytics Workspace**: A workspace for storing logs and metrics for various Azure services.
2. **Application Insights**: An application monitoring resource to track the performance and health of your Java backend and React frontend.
3. **Diagnostic Settings**: Configured on the AKS cluster to send logs and metrics to the Log Analytics workspace.

Monitor Template Details

1. **`location`**: Specifies the Azure region where the monitoring resources will be deployed.
2. **`logAnalyticsWorkspaceName`**: Name of the **Log Analytics Workspace**.
3. **`applicationInsightsName`**: Name of the **Application Insights** instance.
4. **`aksResourceId`**: Resource ID of the Azure Kubernetes Service (AKS) cluster. This is required to set up diagnostic settings to send AKS logs and metrics to the Log Analytics workspace.

Step 2: **CI/CD Pipelines with Azure DevOps**

1. Create Pipelines using Azure DevOps YAML templates.

Step 3: Configure AKS with Azure DevOps

- Create a **Service Connection** in Azure DevOps to connect to AKS.
- Use `kubectl` commands or **Helm Charts** for deploying the application.

Step 4: Implement Logging and Monitoring

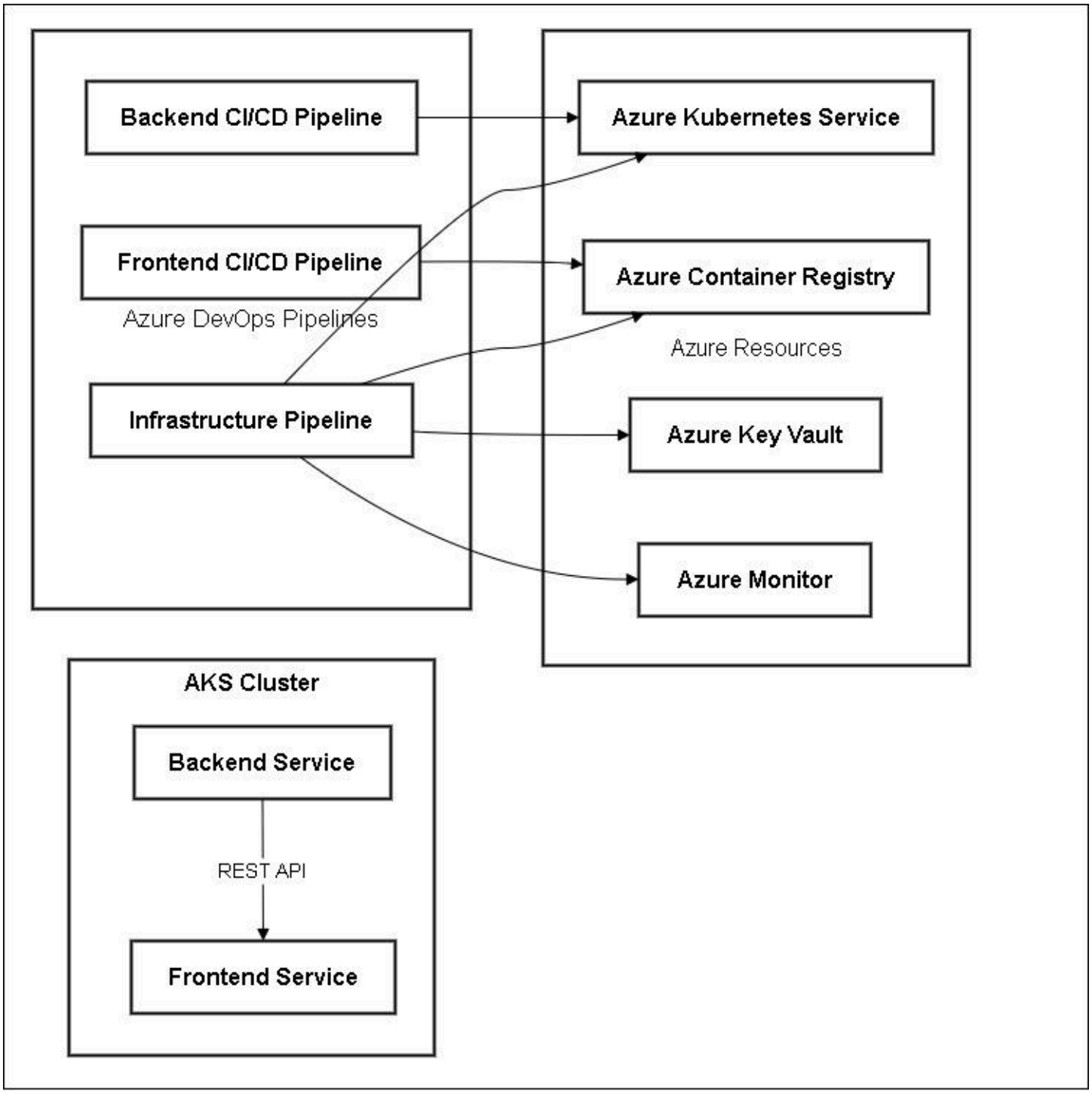
- Set up **Azure Monitor** with Log Analytics for container insights.
- Use `kubectl logs` and `kubectl describe` for troubleshooting.

Step 5: Database Integration

- Replace the in-memory database with **Azure PostgreSQL**.
- Update the `application.properties` in the Java backend to use the Azure PostgreSQL connection string from **Azure Key Vault**.

Step 6: Automatic Scaling and Clean Up

- Enable **Horizontal Pod Autoscaler** in AKS for auto-scaling.
- Create a `terraform destroy` or `az group delete` pipeline for resource cleanup.


Backend CI/CD Pipeline
Frontend CI/CD Pipeline
Azure DevOps Pipelines
Infrastructure Pipeline
Azure Kubernetes Service
Azure Container Registry
Azure Resources
Azure Key Vault
Azure Monitor
AKS Cluster
Backend Service
REST API
Frontend Service

Solution Directory Structure

react-and-spring-data-rest/

react-and-spring-data-rest/

```
├── .mvn
│   └── wrapper/
├── src/
│   ├── main/
│   │   ├── java/
│   │   │   └── com/
│   │   │       └── contoso/
│   │   │           └── payroll/
│   │   │               ├── Application.java
│   │   │               └── [Other Java files]
│   │   ├── js/
│   │   │   ├── api/
│   │   │   │   ├── uriListConverter.js
│   │   │   │   └── uriTemplateInterceptor.js
│   │   │   ├── app.js
│   │   │   ├── client.js
│   │   │   ├── follow.js
│   │   │   └── websocket-listener.js
│   │   ├── resources/
│   │   │   ├── static/
│   │   │   │   └── built/
│   │   │   │       └── main.css
│   │   │   ├── templates/
│   │   │   │   └── index.html
│   │   │   └── application.properties
│   ├── test/
│   │   └── java/
│   │       └── com/
│   │           └── contoso/
│   │               └── payroll/
│   │                   └── [Test files]
├── .dockerignore
├── .gitignore
├── azure-pipelines/
│   ├── backend-pipeline.yml
│   ├── frontend-pipeline.yml
│   └── infrastructure-pipeline.yml
├── arm-templates/
```

```
│   ├── azuredeploy.json
│   ├── parameters.json
│   ├── keyvault-template.json
│   └── monitor-template.json
├── docker/
│   ├── backend/
│   │   ├── Dockerfile
│   │   └── [Other Docker files]
│   └── frontend/
│       ├── Dockerfile
│       └── [Other Docker files]
├── scripts/
│   └── deployment-scripts/
│       ├── deploy.sh
│       └── cleanup.sh
├── README.md
└── LICENSE
```

Explanation of Each Directory

1. **`.mvn/wrapper`**: Maven wrapper files for building the Java application.
2. **`src/main/java/com/contoso/payroll`**: Java source code for the backend application.
3. **`src/main/js`**: JavaScript files for the React frontend application.
4. **`src/main/resources`**: Resource files, including static assets and configuration files for the backend.
5. **`src/test`**: Contains unit and integration tests for the Java backend.
6. **`.dockerignore`**: Specifies files to ignore when building Docker images.
7. **`.gitignore`**: Specifies files and directories to ignore in Git.
8. **`azure-pipelines`**: Contains YAML files for Azure DevOps pipelines:
    - **`backend-pipeline.yml`**: CI/CD pipeline for the Java backend.
    - **`frontend-pipeline.yml`**: CI/CD pipeline for the React frontend.
    - **`infrastructure-pipeline.yml`**: Pipeline for deploying the infrastructure using ARM templates.
9. **`arm-templates`**: Contains ARM templates for deploying Azure resources:
    - **`azuredeploy.json`**: Main deployment template.
    - **`parameters.json`**: Parameters file for the ARM templates.
    - **`keyvault-template.json`**: Template for deploying Azure Key Vault.
    - **`monitor-template.json`**: Template for setting up monitoring resources.
10. **`docker`**: Contains Docker-related files:

    - **`backend`**: Contains Dockerfile and related files for the Java backend.
    - **`frontend`**: Contains Dockerfile and related files for the React frontend.
11. **`scripts`**: Contains scripts for deployment and cleanup tasks:
    - **`deployment-scripts`**: Shell scripts for automating deployments and cleanups.
12. **`k8s/:`** Defines the deployment configuration for the frontend/backend service. It specifies the Docker image, number of replicas, and resource limits for the frontend pods.
13. **`k8s/:`** Defines a service to expose the frontend application. This is usually a `LoadBalancer` or `NodePort` service, depending on how the frontend is accessed.
14. **`README.md`**: Documentation file explaining how to set up and use the project.
15. **`LICENSE`**: License information for the project.

4. Instructions to Fork, Configure, and Deploy the Solution

This guide explains how to fork the repository, configure the Azure cloud environment, and deploy the solution on your own Azure environment using Azure DevOps CI/CD pipelines.

Prerequisites

1. **Azure Subscription**: Ensure that you have an active Azure account.
2. **Azure CLI**: Install the Azure CLI on your local machine.
3. **Azure DevOps Account**: Create an Azure DevOps account if you don't already have one.
4. **Docker**: Ensure Docker is installed and running.
5. **Kubernetes Cluster**: The solution will be deployed on an Azure Kubernetes Service (AKS) cluster. Ensure you have the AKS service created.
6. **Terraform (Optional)**: If using Terraform for provisioning infrastructure.

Step 1: Fork the Repository

1. Go to the Git repository that contains the project files.
2. Click the "Fork" button at the top right corner to create a copy of the repository under your own GitHub account.
3. Clone the forked repository to your local machine:

   ```
   git clone https://github.com/prashantgr/IaC_Azure_Cloud.git
   ```
4. Navigate to the project folder:

   ```
   cd prash-DevOps-Project
   ```

Step 2: Configure the Azure Resources

1. **Create an Azure Resource Group**: Run the following command to create a resource group for your deployment:

```
az group create --name prash-devops-rg --location eastus
```

2. **Deploy the Infrastructure using ARM Templates**: Use the provided ARM templates (or Terraform, if configured) to deploy the required infrastructure (AKS, ACR, Key Vault, and Monitoring).
   Run the ARM template deployment for AKS:

   ```
   az deployment group create --resource-group prash-devops-rg --template-file arm-templates/aks-template.json
   ```

   Similarly, deploy the Key Vault and Monitoring resources:

   ```
   az deployment group create --resource-group prash-devops-rg --template-file arm-templates/keyvault-template.json
   az deployment group create --resource-group prash-devops-rg --template-file arm-templates/monitor-template.json
   ```

3. **Configure Azure Container Registry (ACR)**: Create an Azure Container Registry (ACR) for storing Docker images:

   ```
   az acr create --resource-group prash-devops-rg --name prashContainerRegistry --sku Basic
   ```

   After creating the ACR, enable the AKS cluster to use ACR:
   ```
   az aks update -n XYZ-aks-cluster -g prash-devops-rg --attach-acr prashContainerRegistry
   ```

Step 3: Setup Azure DevOps Pipelines

1. **Create a New Azure DevOps Project**:
   - Go to your Azure DevOps organization.
   - Create a new project named `prash-DevOps-Project`.
2. **Import the Repository**:
   - Go to "Repos" in your new project.
   - Import the GitHub repository that you forked earlier (https://github.com/prashantgr/IaC_Azure_Cloud.git).
3. **Create and Configure Service Connections**:
   - Navigate to **Project Settings** > **Service connections**.
   - Add a new Azure Resource Manager service connection:
     - Select the appropriate Azure subscription.
     - Grant permissions to all pipelines.
     - Set it as the default connection.
   - Add a new **Docker Registry Service Connection**:
     - Enter the ACR details (`prashContainerRegistry.azurecr.io`).
     - Use your Azure credentials.
4. **Setup Pipelines for CI/CD**:
   - Navigate to **Pipelines** in your Azure DevOps project.

    - Create a new pipeline for each of the following files:
5. **Backend Pipeline** (`prash-backend-pipeline.yml`):
    - YAML file path: `prash-DevOps-Project/backend-pipeline.yml`
    - This pipeline will build, test, and push the backend Docker image to ACR.
6. **Frontend Pipeline** (`prash-frontend-pipeline.yml`):
    - YAML file path: `prash-DevOps-Project/frontend-pipeline.yml`
    - This pipeline will build, test, and push the frontend Docker image to ACR.
7. **Infrastructure Pipeline** (`infrastructure-pipeline.yml`):
    - YAML file path: `prash-DevOps-Project/infrastructure-pipeline.yml`
    - This pipeline will deploy the AKS, Key Vault, and other required resources using ARM templates.
8. **Configure Pipeline Variables**: Edit each pipeline's variables based on your setup:
    - **Azure Subscription ID**
    - **Resource Group Name**: `[rash-devops-rg`
    - **AKS Cluster Name**: `prash-aks-cluster`
    - **ACR Name**: `prashContainerRegistry`
    - **Docker Repository**: `backend` or `frontend` depending on the pipeline.

Step 4: Deploy the Solution Using the Pipelines

1. **Run the Infrastructure Pipeline** (`infrastructure-pipeline.yml`):
    - This will provision all required Azure resources.
    - Ensure that the deployment completes successfully.
2. **Run the Backend Pipeline** (`backend-pipeline.yml`):
    - This pipeline builds the backend Java application, runs unit tests, creates a Docker image, and pushes it to ACR.
3. **Run the Frontend Pipeline** (`frontend-pipeline.yml`):
    - This pipeline builds the React frontend application, creates a Docker image, and pushes it to ACR.

Step 5: Deploy the Application to AKS

1. **Get AKS Credentials**: Run the following command to connect your local `kubectl` to the AKS cluster:

   ```
   az aks get-credentials --resource-group prash-devops-rg --name
   prash-aks-cluster
   ```
2. **. Run the Deployment Script** (`prash-deploy.sh`): This script automates the deployment of both the backend and frontend services:

   ```
   ./deploy-scripts/deploy.sh
   ```
3. **Verify the Deployment**:
   Run the following command to check the status of your deployments:

   ```
   kubectl get deployments
   kubectl get services
   ```

Look for external IPs assigned to the frontend service (`frontend-service`) for accessing the application.

Step 6: Cleanup Resources (Optional)

To remove all deployed resources, use the `cleanup.sh` script provided in the `XYZ-deploy-scripts` directory:

```
./deploy-scripts/cleanup.sh
```

This script will delete all the Kubernetes deployments and Azure resources created for this solution.

Step 7: Access the Deployed Application

- **Frontend**: Navigate to the external IP address for the `frontend-service`.
- **Backend**: If exposed, the backend can be accessed via the external IP address for the `backend-service`.

Final Project Folder Structure

```
XYZ-DevOps-Project/
├── XYZ-backend/
│   ├── Dockerfile
│   ├── src/
│   └── pom.xml
├── XYZ-frontend/
│   ├── Dockerfile
│   ├── src/
│   └── package.json
├── XYZ-arm-templates/
│   ├── XYZ-aks-template.json
│   ├── XYZ-keyvault-template.json
│   ├── XYZ-monitor-template.json
│   └── XYZ-network-template.json
├── XYZ-deploy-scripts/
│   ├── XYZ-deploy.sh
│   ├── XYZ-cleanup.sh
├── k8s/
│   ├── backend-deployment.yaml
│   ├── backend-service.yaml
│   ├── frontend-deployment.yaml
│   ├── frontend-service.yaml
├── XYZ-backend-pipeline.yml
├── XYZ-frontend-pipeline.yml
├── XYZ-infrastructure-pipeline.yml
└── README.md
```

This structure provides a well-organized setup for deploying the entire solution. Let me know if you'd like to refine or adjust any part of it!