**ATS CloudinityInc.**

**Frontend (React.js):**

* The React application will serve as the UI for the job application form.
* Users fill in their details and upload their resume here.
* The application makes **HTTP API calls** to interact with the backend.

**Backend (AWS Lambda & API Gateway):**

* AWS Lambda functions will handle the business logic of your application.
* Use AWS API Gateway to create **HTTP APIs** that the React app will call.
* The APIs will be responsible for processing requests from the frontend, such as saving data to DynamoDB and uploading files to S3.

**Storage:**

* **AWS S3:** Store the resumes uploaded by the users. Lambda functions can upload files to S3 buckets.
* **AWS DynamoDB:** Store the details of the applicants. Lambda functions will interact with DynamoDB to store and retrieve data.

**CI/CD Pipeline (Jenkins, Docker, GitHub):**

* **GitHub:** Host your code repository for the React app, Lambda functions, and any other services you create.
* **Docker:** Use Docker to containerize your application, especially useful if you have services other than Lambda.
* **Jenkins:** Automate your deployment pipeline. Jenkins can pull code from GitHub, run tests, build Docker images (if needed), and deploy to AWS.
* **AWS SAM (Serverless Application Model):** Utilize SAM templates to define your serverless resources (Lambda, API Gateway, DynamoDB) and deploy them as a stack.

**Testing and Quality Assurance (SonarQube):**

* Integrate **SonarQube** into your Jenkins pipeline for continuous code quality checks and security scanning.

**Security and Best Practices**:

* Ensure that your AWS resources (like S3 buckets and DynamoDB tables) have proper access controls and policies.
* Implement authentication and authorization mechanisms if necessary.
* Use **environment variables** and **AWS Secrets Manager** for managing sensitive data like API keys.

**Monitoring and Logging:**

* Implement logging in your Lambda functions.
* Use **AWS CloudWatch** for monitoring and logging the performance of your Lambda functions, API Gateway, and other AWS resources.

**Architecture 1: Serverless Architecture**

**Key Components:**

* AWS Lambda: To run backend code without provisioning or managing servers.
* Amazon API Gateway: To create, publish, maintain, monitor, and secure APIs.
* AWS DynamoDB: As a NoSQL database service for applications.
* AWS S3: To store static assets, including the React frontend and any uploaded files.
* AWS Cognito: For user authentication and authorization.

**Flow:**

* Frontend (React App): Hosted on AWS S3, accessible to users via a web browser.
* User Authentication (Optional): AWS Cognito is used if user authentication is required.
* API Calls: The React app makes API calls to AWS API Gateway.
* Serverless Backend: API Gateway routes requests to the appropriate AWS Lambda functions.
* Data Storage: Lambda functions interact with AWS DynamoDB for data persistence.
* File Storage: Any file uploads are handled by Lambda functions and stored in AWS S3.

**Architecture 2: Spring Boot on AWS Fargate**

**Key Components:**

* AWS Fargate: For running containers without managing servers or clusters.
* Amazon RDS or DynamoDB: As the database service.
* Elastic Load Balancer (ELB): To distribute incoming application traffic.
* Amazon ECR (Elastic Container Registry): To store Docker images.
* CI/CD.

**Flow:**

* Spring Boot Application: Packaged as a Docker container.
* Docker Image Storage: Images stored in Amazon ECR.
* Deployment: Docker containers are deployed on AWS Fargate.
* Database Connection: The Spring Boot app connects to either Amazon RDS (for SQL) or DynamoDB (for NoSQL).
* Load Balancing: ELB distributes incoming traffic to ensure scalability and reliability.
* CI/CD Pipeline

Estimated sprint time for the project micro services:

* Creating Git and AWS account and managing IAM roles – **1 day**
* Building React-app -> **2 weeks**
* Lambda functions for DynamoDB and S3 -> **2 weeks**
* Creating REST API for Lambda and react app -> **2weeks**