

# Project - High Level Design

on

## IAC-PROVISIONING

## AGRICULTURE-SYSTEM

Course Name: DevOps Fundamentals

**Institution Name:** Medicaps University – Datagami Skill Based Course

*Student Name(s) & Enrolment Number(s):*

Sr no	Student Name	Enrolment Number
1	Vidhi Gupta	EN22CS3011075
2	Tejashwi Bharti	EN22CS3011036
3	Tanishka Soni	EN22CS3011020
4	Tanish Patidar	EN22CS3011016
5	Sriyansh Shrivastava	EN22CS301986

Group Name: 11D9

Project Number: D0-11

Industry Mentor Name: Mr. Vaibhav

University Mentor Name: Dr. Ritesh Joshi

Academic Year: 2025-2026

# Table of Contents

## 1. Introduction:

The *IaC Provisioning for Agriculture System* project focuses on automating the infrastructure setup and deployment of an Agriculture Management System using Infrastructure as Code (IaC) principles. The primary objective of this project is to eliminate manual server configuration, reduce deployment errors, and ensure a repeatable, scalable, and reliable environment setup.

The system provisions cloud infrastructure (AWS EC2) using Terraform and configures the server environment using Docker-based containerization. The backend application (Flask with Gunicorn) and MySQL database are deployed using Docker Compose, ensuring portability and consistency across environments.

This project demonstrates DevOps practices including Infrastructure Automation, Containerization, Cloud Deployment, and CI/CD integration.

**1.1. Scope of the document:** This document describes the high-level architecture and deployment process of the system. It covers infrastructure provisioning, Docker-based deployment, application components, and database structure. The focus is on automated environment setup rather than detailed application code.

**1.2. Intended Audience:** This document is intended for developers, DevOps engineers, cloud engineers, system architects, and academic evaluators who need to understand the system's architecture and deployment approach.

**1.3. System overview:** The Agriculture System is a web-based application that manages users and crop data.

It follows a three-tier architecture:

### Layers:

1. Frontend Layer: React/Vite app to display crops, users, and stats
2. Backend Layer: Python Flask REST API for CRUD operations on crops and users
3. Database Layer: MySQL for persistent data storage
4. Containerization Layer: Docker + Docker Compose
5. Orchestration Layer: Docker Compose or Kubernetes (optional for scaling)
6. Cloud Deployment Layer: AWS EC2 Linux instance deployed via Terraform

## 2. System Design:

The system is designed using Infrastructure as Code (IaC) principles and follows a modular, containerized architecture. It uses Terraform for infrastructure provisioning and Docker for application deployment. The system follows a three-tier architecture to ensure scalability and maintainability.

**2.1. Application Design :** The application follows a layered architecture:

**Frontend Layer** – Handles user interaction.

Processes API requests and business logic.

**Database Layer (MySQL) –** Stores user and crop data. The backend communicates with the database using secure internal Docker networking.

### **2.2. Process Flow:**

- Developer pushes code to Git repository.
- Terraform provisions AWS EC2 infrastructure.
- Docker is installed on the server.
- Docker Compose builds and starts backend and database containers.

Users access the application via Public IP and exposed port.

### **2.3. Information Flow:**

- User sends request through browser.
- Request reaches backend API.
- Backend processes logic and interacts with MySQL database.
- Database sends response to backend.
- Backend returns response to user.

#### **Flow:**

User → Frontend → Backend → Database → Backend → User

### **2.4. Components Design:**

The system consists of the following components:

- **Terraform Module** – Provisions AWS EC2 and networking.
- **Docker Engine** – Manages containers.
- **Backend Container** – Runs Flask application using Gunicorn.
- **Database Container** – Runs MySQL server.
- **Docker Network** – Enables secure communication between containers.

Each component is isolated and independently manageable.

### **2.5. Key Design Considerations:**

- Infrastructure automation using IaC
- Containerization for portability
- Separation of concerns (3-tier architecture)
- Scalability through cloud deployment
- Reduced manual configuration errors
- Security via Docker networking and controlled ports

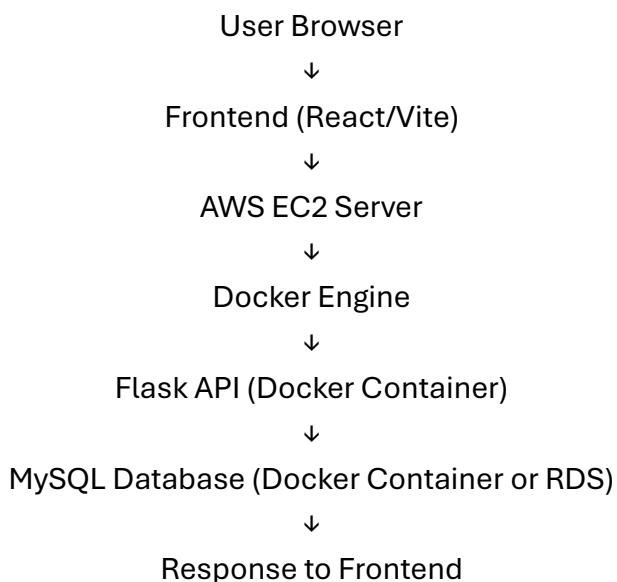
### **2.6. API Catalogue:**

- GET / → Health check / Backend status
- POST /users → Create new user
- GET /users → Fetch user details

- POST /crops → Add crop data
  - GET /crops → Retrieve crop records
- All APIs follow RESTful principles and communicate using JSON format.

## Flow Diagram

- ❑ Developer pushes code to GitHub.
- ❑ CI/CD pipeline triggers.
- ❑ Terraform provisions infrastructure.
- ❑ Ansible installs:
- Docker
  - Docker Compose
    - ❑ Docker builds backend and frontend images.
    - ❑ Containers are started.
    - ❑ Application becomes accessible via Public IP.



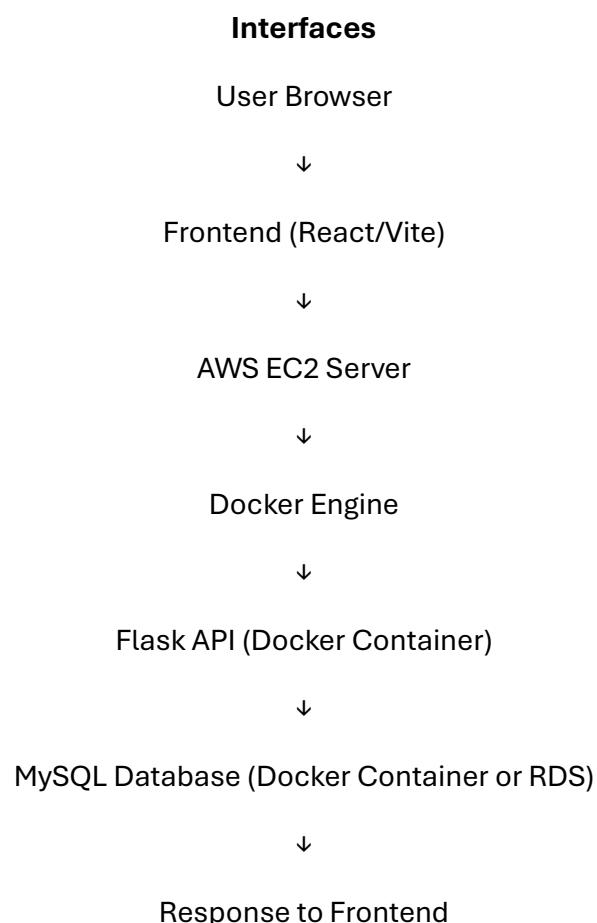
- EC2 Instance
- Security Groups (Ports 22, 5000)
- Docker Engine
- Docker Compose

**Application Components:**

- agri-backend container
- agri-mysql container
- Frontend container

**3 Key Design Considerations**

- Infrastructure reproducibility
- Scalability
- Container isolation
- Automated provisioning
- Secure credential management (.env)
- Minimal manual intervention



## Data Design

### Entities:

- User
- Crop

## 13. Data Model

### User

- id
- name
- email
- password

### Crop

- id
- crop\_name
- season
- price

---

## 14. Data Access Mechanism

- MySQL Connector (Python)
- ORM-like structured service layer
- Docker internal networking

---

## 15. Data Retention Policies

- Data stored in MySQL volume
- Backup via manual export or snapshot
- EC2 EBS snapshot supported

---

## 16. Data Migration

- Terraform supports new environment provisioning
- Database schema auto-created at container startup
- Manual SQL migration supported

---

## 17. Interfaces

### External Interfaces:

- AWS Cloud
- GitHub Repository
- Jenkins / GitHub Actions

### Internal Interfaces:

- REST APIs
- Docker Network

---

## 18. State and Session Management

- Stateless backend

- JWT-based authentication (if implemented)
  - No session persistence on server
- 

**19. Caching**

- No caching implemented (optional: Redis)
  - Can be extended for performance optimization
- 

**20. Non-Functional Requirements**

- Availability: 99%
  - Scalability: Horizontal via Docker/K8s
  - Maintainability: Modular code
  - Portability: Dockerized environment
- 

**21. Security Aspects**

- Security Group firewall rules
  - SSH Key-based access
  - Environment variables for secrets
  - No hardcoded credentials
  - Docker network isolation
- 

**22. Performance Aspects**

- Gunicorn multi-worker support
  - Docker containerization
  - Lightweight Python slim image
  - MySQL optimized connection
- 

**23. References**

- Terraform Documentation
- Docker Documentation
- Flask Documentation
- AWS EC2 Documentation
- GitHub Actions Documentation