**Auto Mail Trigger System – Detailed Project Description**

**1. Introduction**

The **Auto Mail Trigger System** is a Python-based automation solution designed to improve the efficiency and reliability of quality control operations in organizations. By leveraging scheduled scripts, database querying, and email automation, the system ensures timely notifications to responsible personnel whenever a quality check is due. This eliminates the need for manual monitoring and helps organizations maintain compliance with internal and external quality standards.

This document details the functionalities, architecture, implementation, and deployment of the Auto Mail Trigger System.

**2. Objectives**

* **Automate Notification Process**: Eliminate manual follow-ups by triggering emails based on predefined due conditions.
* **Improve Data Accuracy**: Regular checks on the database ensure that due records are promptly addressed.
* **Enhance Operational Efficiency**: Reduce response time to pending quality checks through automated alerts.
* **Reliable Scheduling**: Ensure daily execution using Linux cron jobs.

**3. System Overview**

The system comprises four key modules:

1. **Database Connectivity**
2. **Due Check Logic**
3. **Email Notification System**
4. **Scheduling via Cron**

Each of these components works together to perform daily evaluations of quality check records and alert stakeholders in case of overdue items.

**4. Functional Components**

**4.1 Database Integration**

**Technology Used**: PostgreSQL (psycopg2 in Python)

* The system establishes a connection to a PostgreSQL database using Python’s psycopg2 module.
* The relevant table, quality\_check, contains records that include:
  + id
  + check\_name
  + due\_date
  + status (e.g., pending, complete)
  + assigned\_to (email or employee ID)

**Sample Code:**

python

CopyEdit

import psycopg2

conn = psycopg2.connect(

dbname="your\_db",

user="your\_user",

password="your\_password",

host="localhost",

port="5432"

)

cursor = conn.cursor()

**4.2 Due Check Logic**

**Logic Description**:

* The script fetches all records from the quality\_check table.
* It compares the due\_date with the current date.
* If the due\_date is **today or earlier** and the status is pending, the record is marked as "due."

**Python Implementation:**

python

CopyEdit

from datetime import datetime

today = datetime.today().date()

cursor.execute("SELECT \* FROM quality\_check WHERE due\_date <= %s AND status = 'pending'", (today,))

due\_records = cursor.fetchall()

* The system stores these due records for email alert generation.

**4.3 Email Notification System**

**Technology Used**: Python smtplib, email module

* When due records are found, the system constructs a detailed email and sends it to the corresponding authority (assigned\_to field or mapped email).
* SMTP settings support Gmail, Outlook, or organizational mail servers.

**Email Content Includes**:

* Check Name
* Due Date
* Assigned Personnel
* Message to take action

**Sample Code:**

python

CopyEdit

import smtplib

from email.mime.text import MIMEText

from email.mime.multipart import MIMEMultipart

sender\_email = "your\_email@example.com"

password = "your\_email\_password"

for record in due\_records:

recipient = record[4]

msg = MIMEMultipart()

msg['From'] = sender\_email

msg['To'] = recipient

msg['Subject'] = f"Quality Check Due: {record[1]}"

body = f"""

Dear Team,

The following quality check is due:

- Task: {record[1]}

- Due Date: {record[2]}

- Status: Pending

Please take the necessary action.

Regards,

Auto Mail Trigger System

"""

msg.attach(MIMEText(body, 'plain'))

with smtplib.SMTP('smtp.gmail.com', 587) as server:

server.starttls()

server.login(sender\_email, password)

server.sendmail(sender\_email, recipient, msg.as\_string())

**5. Deployment & Scheduling**

**5.1 Deployment**

* The script is deployed on a **Linux server** (can be local or cloud-hosted, such as AWS EC2).
* Required Python packages are installed in a virtual environment or system-wide:
  + psycopg2
  + smtplib
  + email
  + datetime

**Example Directory Structure**:

arduino

CopyEdit

auto\_mail\_trigger/

├── main.py

├── config.py

├── logs/

└── requirements.txt

**5.2 Cron Job Scheduling**

**Scheduler**: Linux cron

* The script is scheduled to run **daily at 9:00 AM**.
* Cron syntax:

bash

CopyEdit

0 9 \* \* \* /usr/bin/python3 /home/user/auto\_mail\_trigger/main.py >> /home/user/auto\_mail\_trigger/logs/output.log 2>&1

* The output is logged for audit purposes.

**Benefits of Using Cron**:

* Low overhead
* Reliable and built-in for Unix systems
* Easy to modify schedule (e.g., hourly, weekly)

**6. Error Handling & Logging**

* Basic error handling is implemented for:
  + Database connection failures
  + Email send failures
  + Empty record sets
* Logs are generated to track:
  + Start and end of script execution
  + Number of emails sent
  + Errors with timestamps

**Sample Logging Setup:**

python

CopyEdit

import logging

logging.basicConfig(filename='logs/output.log', level=logging.INFO)

logging.info("Script started at " + str(datetime.now()))

**7. Security Considerations**

* **Email Credentials**: Use environment variables or .env files to store sensitive credentials.
* **Database Security**: Limit database user permissions; use SSL if available.
* **Spam Control**: Prevent duplicate notifications by tracking previous sends (optional enhancement).
* **Error Alerts**: Optional email or Slack alert if the script fails to execute.

**8. Future Enhancements**

* Web dashboard for monitoring due checks.
* Integration with other tools like Slack or Microsoft Teams.
* Retry logic for failed emails.
* Record logging in the database after email sent.

**9. Conclusion**

The **Auto Mail Trigger System** is a simple yet powerful tool for automating quality control alerts. By combining PostgreSQL data access, date-based logic, email alerts, and cron scheduling, the system helps organizations stay proactive in managing their quality check tasks. This reduces human error, ensures accountability, and improves overall operational compliance.

# ****Python Backend Service with CI/CD Pipeline – Detailed Project Description****

## ****1. Introduction****

In modern software development, the integration of backend services with continuous integration and continuous deployment (CI/CD) has become essential for ensuring agility, scalability, and rapid delivery of new features. This project focuses on a **Python-based backend service**, which is fully **containerized using Docker** and **automated through Jenkins CI/CD pipelines**.

The system is designed to deliver structured, JSON-formatted data via an API and supports seamless code integration, automated builds, multi-environment deployments, and real-time notifications through email.

## ****2. Objectives****

* **Build a robust and scalable backend service** for data retrieval and delivery.
* **Ensure consistent deployment** using containerization with Docker.
* **Automate the development-to-deployment workflow** using Jenkins CI/CD pipelines.
* **Minimize manual intervention** through automation and continuous monitoring.
* **Provide real-time feedback** via email alerts during every stage of the CI/CD process.

## ****3. System Overview****

This solution consists of the following major components:

1. **Python Backend Service** – Built using Flask or FastAPI, exposes RESTful APIs.
2. **Docker Containerization** – Ensures consistent environments for development, testing, and production.
3. **Jenkins CI/CD Pipeline** – Automates code integration, testing, image building, and deployment.
4. **Deployment Targets** – Code is deployed sequentially to the **Quality** and **Production** servers.
5. **Email Notifications** – Keeps the development/DevOps teams updated at each pipeline stage.

## ****4. Functional Components****

### ****4.1 Backend Service (API Layer)****

**Language**: Python  
**Framework**: Flask or FastAPI

The core backend service handles requests and returns data in JSON format through a RESTful API endpoint.

**Functional Features**:

* Establishes a connection to a relational database (e.g., PostgreSQL or MySQL).
* Queries and returns data in structured JSON format.
* Provides health check endpoints for monitoring and deployment validation.

**Sample FastAPI Code**:

python

CopyEdit

from fastapi import FastAPI

import psycopg2

import json

app = FastAPI()

@app.get("/data")

def get\_data():

conn = psycopg2.connect(database="demo", user="user", password="pass", host="db", port="5432")

cursor = conn.cursor()

cursor.execute("SELECT \* FROM sample\_table")

rows = cursor.fetchall()

return {"data": rows}

### ****4.2 Docker Containerization****

**Tool**: Docker

Containerization ensures the application runs uniformly across different environments (development, testing, staging, production).

**Docker Features**:

* Lightweight and portable images.
* Encapsulation of dependencies.
* Simplified deployment.

**Dockerfile Example**:

dockerfile

CopyEdit

FROM python:3.9-slim

WORKDIR /app

COPY requirements.txt .

RUN pip install -r requirements.txt

COPY . .

CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]

**Docker Compose (Optional)** can be used to run the backend along with services like PostgreSQL.

### ****4.3 Jenkins CI/CD Pipeline****

**Tool**: Jenkins  
**Source Control**: Git (e.g., GitHub or GitLab)

The CI/CD pipeline is a fully automated process that connects the backend service codebase with deployment environments through Jenkins.

#### ****Pipeline Workflow****:

1. **Checkout Code**:  
   Jenkins pulls the latest code from the Git repository.
2. **Build Docker Image**:  
   Jenkins uses the Dockerfile to build the backend application image.
3. **Unit Testing (Optional)**:  
   If configured, tests are executed inside the container.
4. **Deploy to Quality Server**:  
   Upon a successful build, the image is deployed to the **Quality (QA) server** for testing.
5. **Deploy to Production Server**:  
   If QA deployment and testing pass, Jenkins continues to deploy the image to the **Production server**.
6. **Email Notifications**:  
   At each stage (build success/failure, deployment status), Jenkins sends an email to predefined recipients.

**Jenkinsfile Sample (Declarative Pipeline)**:

groovy

CopyEdit

pipeline {

agent any

environment {

DOCKER\_IMAGE = 'backend-service:latest'

}

stages {

stage('Checkout') {

steps {

git url: 'https://github.com/org/backend-service.git', branch: 'main'

}

}

stage('Build Docker Image') {

steps {

sh 'docker build -t $DOCKER\_IMAGE .'

}

}

stage('Deploy to QA') {

steps {

sh 'docker run -d -p 8000:8000 $DOCKER\_IMAGE'

}

}

stage('Deploy to Production') {

when {

expression { currentBuild.result == null || currentBuild.result == 'SUCCESS' }

}

steps {

sh 'docker tag $DOCKER\_IMAGE prod/backend-service'

sh 'docker push prod/backend-service'

sh 'ssh prod-user@prod-server "docker pull prod/backend-service && docker run -d -p 8000:8000 prod/backend-service"'

}

}

}

post {

success {

mail to: 'team@example.com',

subject: "Pipeline Success: ${env.JOB\_NAME}",

body: "Deployment completed successfully on both QA and Production."

}

failure {

mail to: 'team@example.com',

subject: "Pipeline Failed: ${env.JOB\_NAME}",

body: "Pipeline failed. Please check the Jenkins logs."

}

}

}

## ****5. Deployment Architecture****

### ****Environments****:

* **Quality Server**: For testing and QA validation.
* **Production Server**: Final deployment environment after successful QA testing.

### ****Network Flow****:

text

CopyEdit

[ Git Repo ] → [ Jenkins ] → [ Docker Build ] → [ QA Server ] → [ Prod Server ]

↓ ↓

[ Email Notifications ] ←───┘

**Deployment Targets**:

* SSH or Docker Swarm/Kubernetes-based deployment (scalable).
* Port and resource monitoring tools like Prometheus, Grafana (optional for enterprise use).

## ****6. Email Notification System****

* Jenkins sends email notifications for:
  + Build success/failure
  + QA deployment result
  + Production deployment result

**Jenkins Email Plugin** is used for configuration.

**Email Template Sample**:

css

CopyEdit

Subject: Deployment Status: ${BUILD\_STATUS}

Body:

Job: ${JOB\_NAME}

Build Number: ${BUILD\_NUMBER}

Status: ${BUILD\_STATUS}

Time: ${BUILD\_TIMESTAMP}

Check the console output: ${BUILD\_URL}

## ****7. Security Considerations****

* Use **secrets management** (e.g., Jenkins credentials, .env files) for passwords and tokens.
* **Role-based access** for Jenkins and Git repositories.
* Secure **Docker images** with vulnerability scanning (e.g., Trivy, Snyk).
* Use **SSH keys** and firewalls for deployment servers.

## ****8. Future Enhancements****

* Add automated unit and integration tests before builds.
* Integrate with Kubernetes for production-grade orchestration.
* Use GitHub Actions or GitLab CI for fully cloud-native CI/CD.
* Introduce canary deployments or blue-green deployments.
* API versioning and Swagger documentation for better maintainability.

## ****9. Conclusion****

The **Python Backend Service with CI/CD Pipeline** is a complete, scalable solution for building and deploying RESTful APIs with minimal manual overhead. Leveraging modern tools like Docker and Jenkins, the system provides a reliable, repeatable, and secure way to manage application lifecycle—from development to deployment.

With proper containerization and automation, the project ensures high availability, ease of scaling, and rapid response to changes—an essential feature for modern DevOps-driven environments.