Project – High Level Design

On

Dockerized Marketing PHP Laravel Service

Course Name: Devops

Institution Name: Medicaps University – Datagami Skill Based Course

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

Sr no	Student Name	Enrolment Number
1	GAURAV RAUNDHALE	EN22IT301036
2	SANIYA KHAN	EN22IT301091
3	SOHAM SINGH KUSHWAHA	EN22IT301104
4	VEDANT MEENA	EN22IT301120

Group Name:08D12

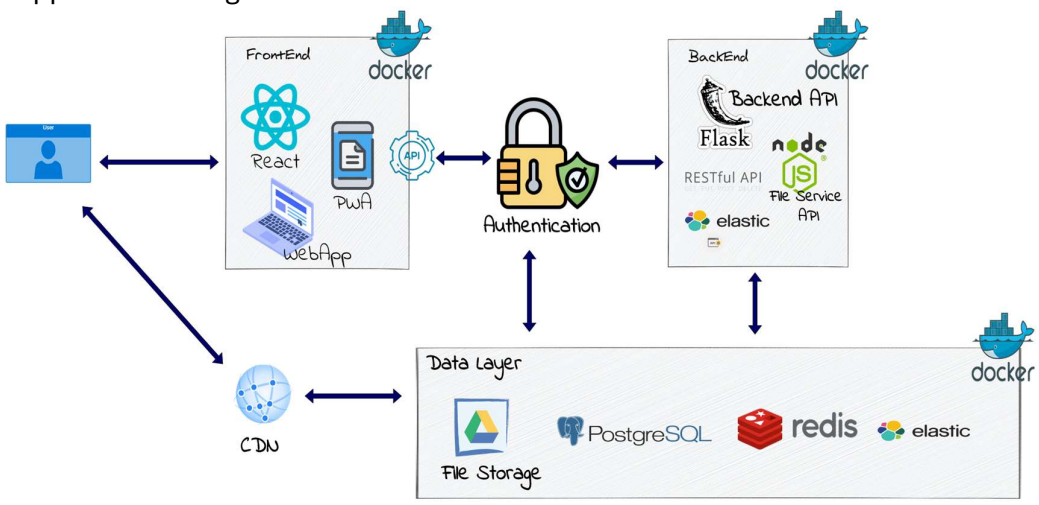
Project Number: DO-46

Industry Mentor Name: Mr.Vaibhav

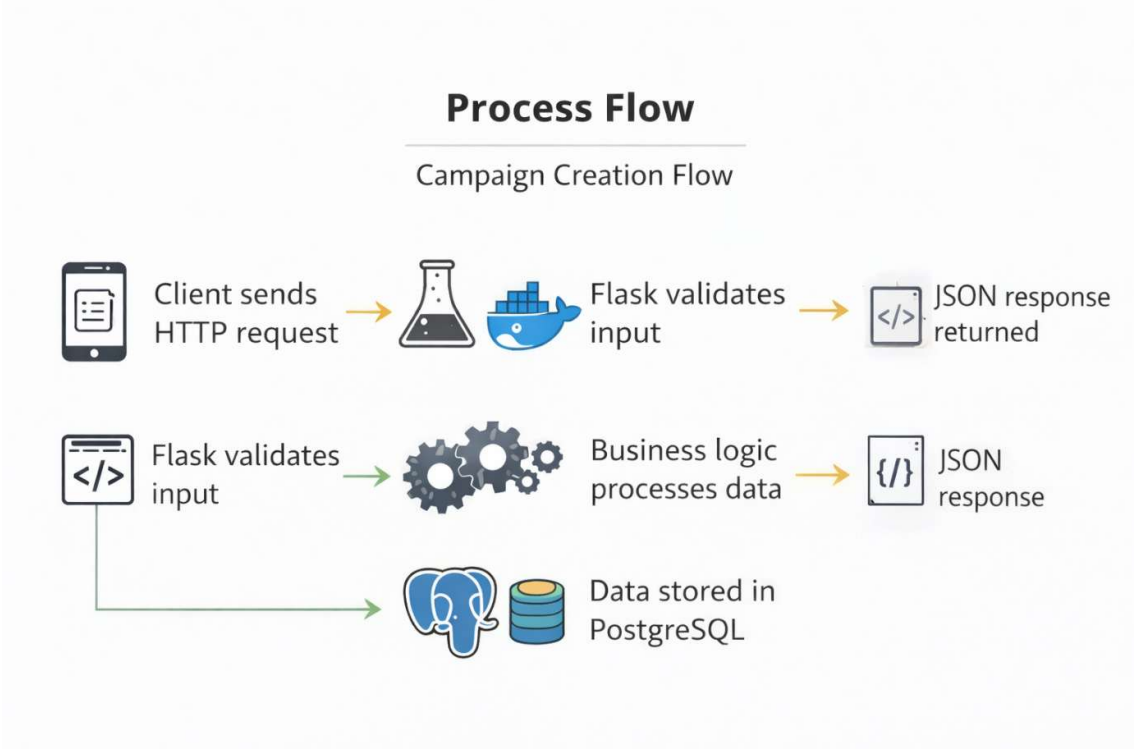
University Mentor Name:Prof.Akshay Saxena

Academic Year: 2026

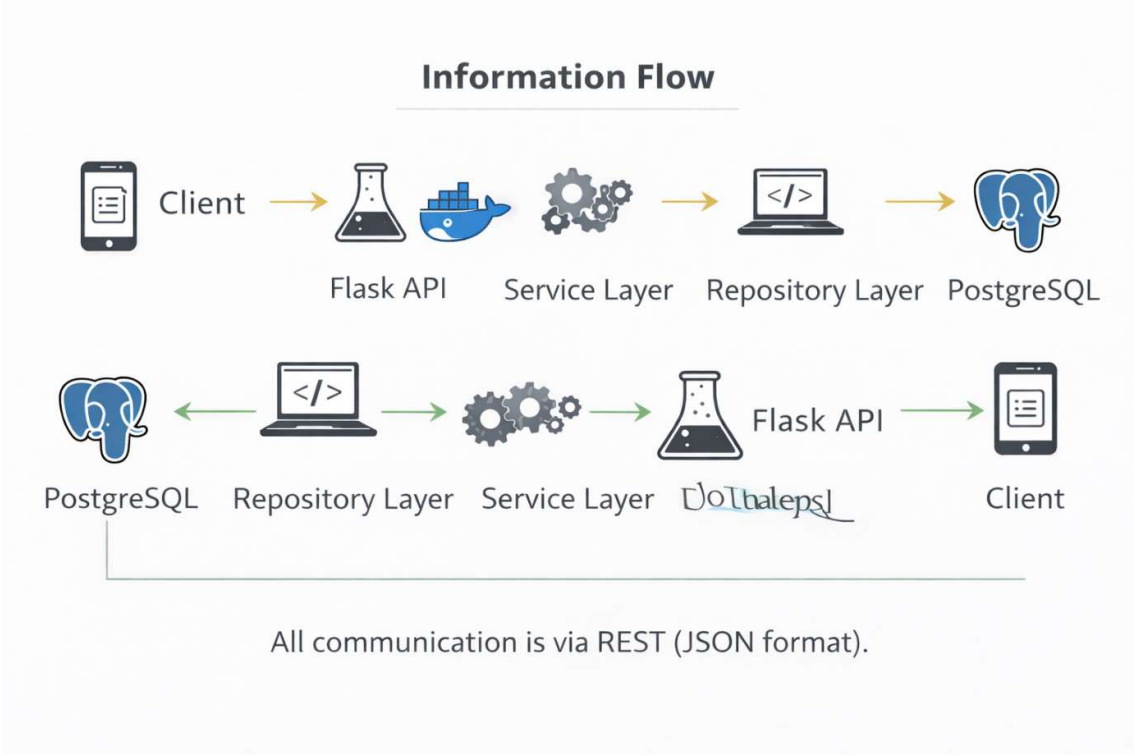
Table of Contents



2.2. Process Flow-

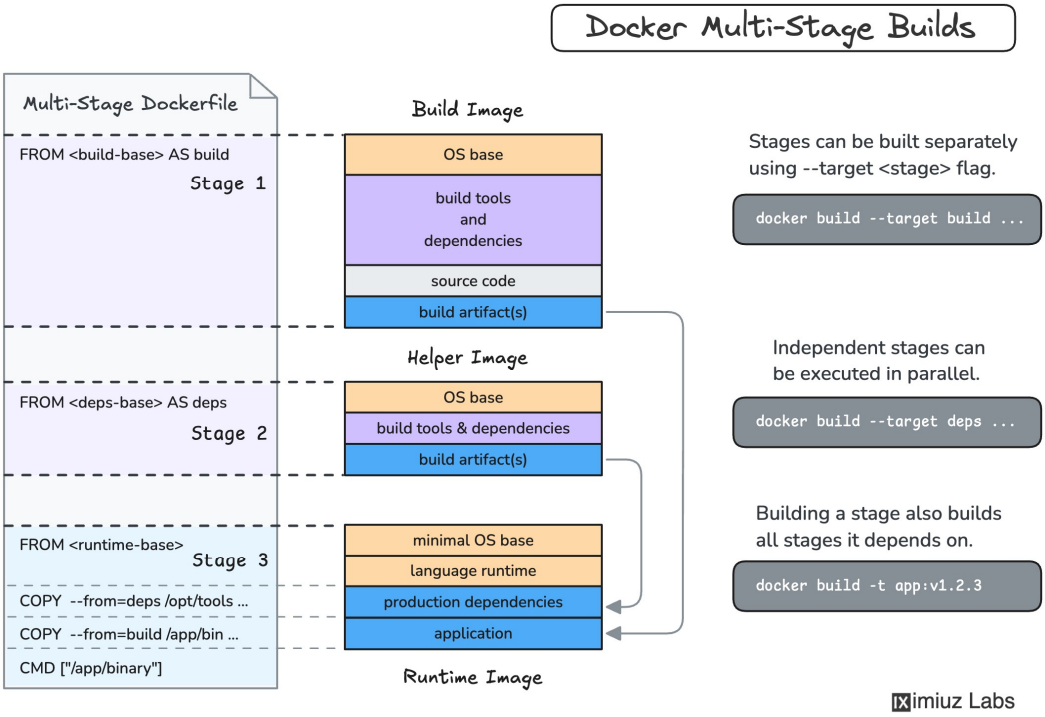


2.3. Information Flow.



1.

2.4. Components Design



2.5. Key Design Considerations

- **Portability:** Using `docker-compose.yml` ensures developers can spin up the entire stack with one command (`docker-compose up`).
- **Image Optimization:** Utilizing Python slim base images and multi-stage builds to reduce image size (target <200MB).
- **Statelessness:** The application containers (Flask) are stateless; ensuring horizontal scalability.

2.6. API Catalogue.

The Marketing Service exposes RESTful endpoints:

- POST /api/v1/campaigns - Create a new marketing campaign.
- GET /api/v1/leads - Retrieve generated leads.
- POST /api/v1/analytics - Ingest click/view data.

3. Data Design.

3.1. Data Model

The PostgreSQL database follows a relational model including:

- **Campaigns Table:** Stores campaign details (ID, Name, Budget, Status).
- **Leads Table:** Stores customer info captured via campaigns.
- **Logs/Events Table:** Stores audit trails for marketing actions.

3.2. Data Access Mechanism

- **Flask-SQLAlchemy** (or a similar Python ORM) is used for all PostgreSQL database interactions to prevent SQL injection and ensure abstraction.
- PostgreSQL Database credentials are injected via Environment Variables (`.env` file in Dev, Kubernetes Secrets in Prod).

3.3. Data Retention Policies

- **Application Logs:** Streamed to stdout/stderr (standard Docker logging) via Python's `logging` module and collected by a logging driver (e.g., ELK stack or CloudWatch).
- **Database Backups:** Scheduled cron jobs run inside a separate utility container to dump PostgreSQL data to external storage (S3/Volume).

3.4. Data Migration

- Schema changes are managed via **Flask-Migrate (Alembic)**.
- A "Migration Job" runs automatically as an initContainer (`flask db upgrade`) or a pre-deployment step in the CD pipeline to ensure the DB schema is up-to-date before the app starts.

4. Interfaces

• User Interface:

A Web Dashboard built using **HTML, CSS, and Flask (Jinja2 templates)** served through a Dockerized Flask application.
The application can be deployed behind **Nginx** as a reverse proxy in production environments.

• System Interface:

• Connects to **SMTP Servers** (e.g., SendGrid, Mailgun, or Gmail SMTP) for sending marketing emails.
• Connects to external **Payment Gateway APIs** (if billing or subscription features are included).
• Connects to **PostgreSQL Database** for persistent data storage.

5. State and Session Management

To ensure the Flask containers remain **stateless** (allowing Kubernetes to scale them up/down freely):

• **Sessions:** Stored in Redis (instead of local files) to maintain session persistence across multiple containers.

• **CSRF Protection:** Managed using Flask security mechanisms (such as Flask-WTF or custom token validation), ensuring secure form submission and request validation.

This ensures scalability and proper session handling in a distributed container environment.

6. Caching

• Application Cache:

Redis is used to cache frequently accessed database queries (e.g., marketing campaign data) to reduce load on PostgreSQL and improve response time.

• Configuration Optimization:

During the Docker build process:

• Python dependencies are pre-installed using requirements.txt
• Production configurations are optimized
• Debug mode is disabled for better runtime performance

This ensures faster startup and improved application efficiency.

7. Non-Functional Requirements

7.1 Security Aspects

• Least Privilege Principle:
Containers run as a non-root user inside the Docker image to enhance security.

• Secret Management:
Sensitive data such as database passwords, API keys, and SMTP credentials are not hardcoded.

They are injected using:
– Docker environment variables
– Docker Secrets
– Kubernetes Secrets

• Image Scanning:
The CI/CD pipeline includes vulnerability scanning (e.g., Trivy) to detect security vulnerabilities in Docker base images.

7.2 Performance Aspects

• Horizontal Scaling:
Kubernetes HPA (Horizontal Pod Autoscaler) automatically increases or decreases Flask pods based on traffic load.

• **Load Balancing:**
Kubernetes services distribute traffic across multiple Flask containers to maintain high availability.

• **Efficient Database Connections:**
Connection pooling is used to optimize database performance and reduce overhead.

8. References

Flask Documentation: https://flask.palletsprojects.com/

Docker Official Documentation: https://docs.docker.com/

Kubernetes Concepts: https://kubernetes.io/docs/concepts/

12-Factor App Methodology: https://12factor.net/