NebulaStack Deployment Architecture & CI/CD System

A Complete Containerized DevOps Workflow

Author: Muhammad Shahwaiz Ali

Table of Contents

1) Executive Summary

NebulaStack is a fully containerized micro-application built to demonstrate a real-world DevOps workflow. The system integrates a modern technology stack including Nginx, Flask, PostgreSQL, Docker, Docker Compose, GitHub Actions, and AWS EC2 to form an end-to-end automated deployment pipeline.

The platform replicates a production-ready architecture where code changes made by developers are automatically tested, packaged, and deployed to a cloud server without manual intervention.

The report outlines the architectural design, components, networking logic, data flow, CI/CD workflow, security considerations, operational practices, and future scalability paths. The goal is to provide a complete, transparent, and technically authentic explanation of how a modern DevOps system functions in practice.

---

2) System Architecture

a) High-Level Overview

NebulaStack is designed using a clear and modular three-tier architecture that separates the application into independent functional layers. This separation ensures maintainability, security, scalability, and predictable behavior across different environments. Each layer operates in its own container and communicates through an internal Docker network, while the entire stack is deployed on an AWS EC2 instance using Docker Compose for orchestration. This structure mirrors the deployment patterns used in modern cloud-native applications.

1. Frontend Layer – Nginx

The frontend layer is powered by Nginx, which acts as the entry point to the application.

Nginx is responsible for:

- Serving static web assets such as HTML, CSS, and images.
- Forwarding all `/api/` requests to the backend service using reverse-proxy rules.
- Applying essential HTTP security headers to mitigate browser-based attacks.

This layer provides fast content delivery and isolates the backend API from direct public exposure.

2. Backend Layer – Flask API

The backend component is implemented using a lightweight Flask application.

Its responsibilities include:

- Exposing RESTful endpoints to handle incoming API requests.
- Managing core business logic such as creating, retrieving, and validating message data.
- Establishing secure connections with the PostgreSQL database to perform CRUD operations.

Because the API runs in an isolated container, its configuration remains consistent across environments.

3. Data Layer – PostgreSQL

PostgreSQL functions as the system’s durable and ACID-compliant data store.

Its role includes:

- Persisting all application data in a transactional, reliable manner.
- Utilizing a Docker-managed named volume to preserve data across container restarts or updates.

This approach ensures database state is never lost during deployments or CI/CD operations.

b) Component Responsibilities

NebulaStack is composed of several core components, each responsible for a specific part of the system’s functionality. Together, these components create a reliable, secure, and fully containerized environment capable of automated builds and deployments. Below is an expanded and professionally written breakdown of each component.

- Nginx

Nginx serves as the public-facing entry point of the application. It manages both static content delivery and traffic routing to backend services. Its lightweight design and high-performance architecture make it ideal for handling concurrent requests efficiently.

Key Responsibilities:

- Static file hosting: Delivers HTML, CSS, JavaScript, and assets directly to users.
- Reverse proxy routing: Forwards `/api/` requests to the internal Flask backend.
- Request buffering: Handles incoming traffic smoothly even under load.
- Security header injection: Adds protective headers to mitigate XSS, clickjacking, and MIME-sniffing attacks.
- Container isolation: Runs in an isolated Docker environment to prevent configuration drift.

- API (Flask)

The Flask-based backend executes all application logic. Since the API is containerized and internal-only, it remains secure while offering a consistent runtime across environments.

Key Responsibilities:

- Input validation: Ensures that incoming data meets expected formats and constraints.
- CRUD operations: Manages creation and retrieval of messages stored in the database.
- JSON responses with structured logging: Produces clean, machine-readable logs useful for monitoring.
- DB connection pooling: Maintains efficient PostgreSQL connections using psycopg2.

- Internal-only exposure: The API is not exposed directly to the public; it is accessible only via Nginx or internal containers.

- PostgreSQL

PostgreSQL is used as the system’s durable and secure datastore. Running inside a private Docker network ensures that only authorized services can access it.

Key Responsibilities:

- Secure message storage: Keeps all user messages in an ACID-compliant relational database.
- Auto table creation: Initializes the `messages` table automatically during the first API operation.
- Private network protection: Completely isolated from any external access.
- Persistent volume: A named volume guarantees that data persists across container restarts, rebuilds, or deployments.

- CI/CD

Continuous Integration and Continuous Deployment automate the entire development workflow. Every code change pushed to the repository triggers a pipeline that verifies and deploys updates.

Key Responsibilities:

- Automated builds: Containers are built with every change to maintain integrity.
- Automated tests: Optional pytest or unit tests can validate API logic before deployment.
- Automated deployment: The pipeline connects securely to EC2 and redeploys the updated stack without manual intervention.

- AWS EC2

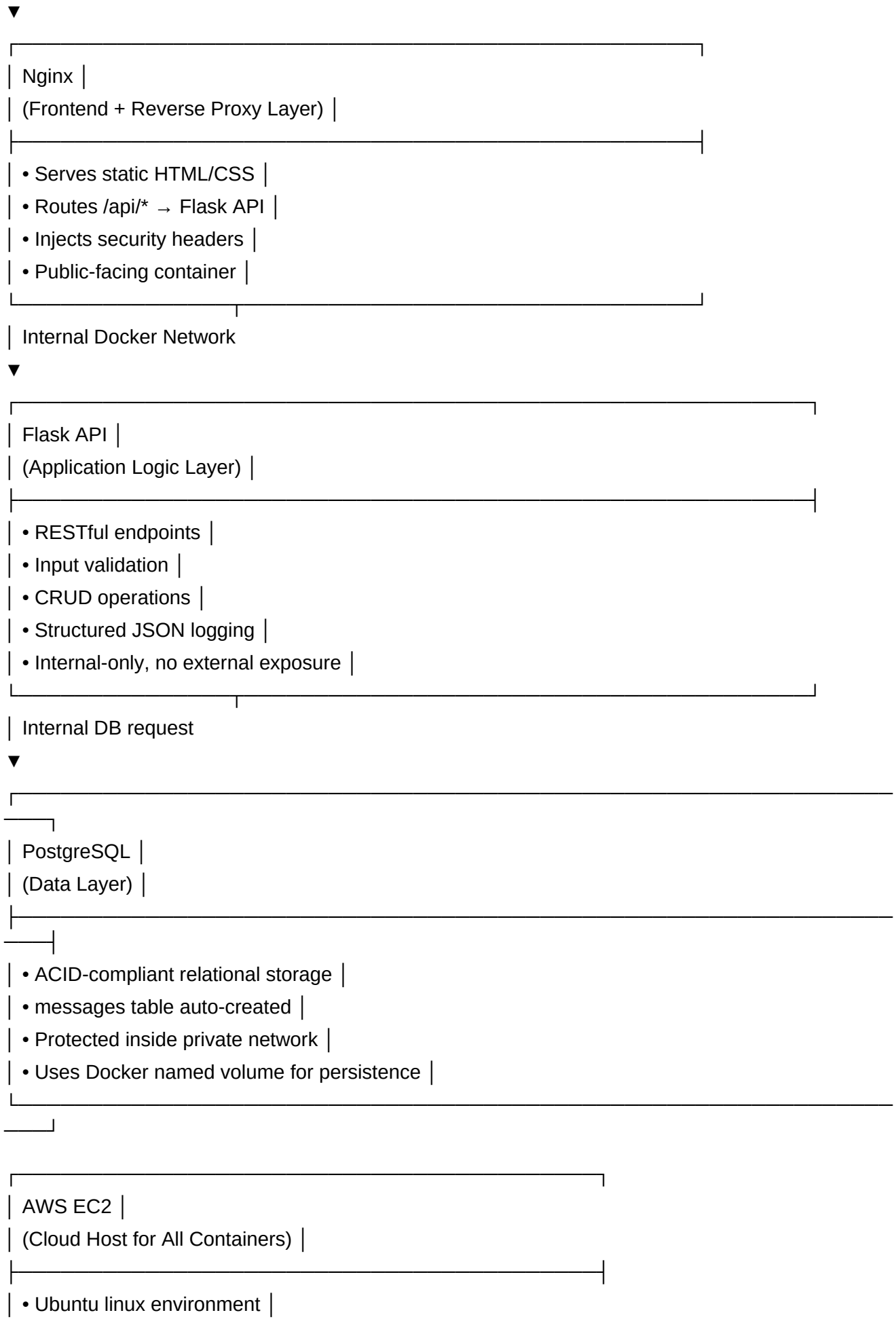
The AWS EC2 instance hosts the production environment for NebulaStack. Docker and Docker Compose run directly on this server, making it easy to manage and deploy multi-container applications.

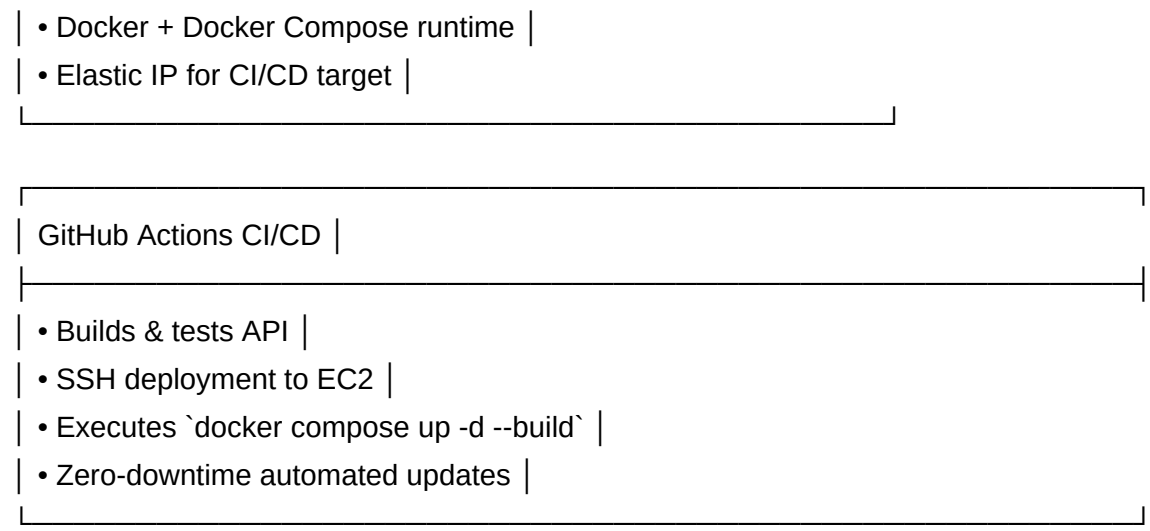
Key Responsibilities:

- Linux host environment: Provides a stable and secure runtime for containers.
- Docker runtime: Executes and manages containers composing the NebulaStack application.
- Elastic IP assignment: Ensures the GitHub Actions workflow always deploys to the correct server.

NebulaStack Architecture Diagram

```
┌──────────────────────────────────────────────┐
│ Users/Clients │
│ (Browser / HTTP Client) │
└──────────────────────┬───────────────────────┘
                       │
                       │ HTTP:8080
```


Nginx
(Frontend + Reverse Proxy Layer)
• Serves static HTML/CSS
• Routes /api/* → Flask API
• Injects security headers
• Public-facing container
Internal Docker Network
Flask API
(Application Logic Layer)
• RESTful endpoints
• Input validation
• CRUD operations
• Structured JSON logging
• Internal-only, no external exposure
Internal DB request
PostgreSQL
(Data Layer)
• ACID-compliant relational storage
• messages table auto-created
• Protected inside private network
• Uses Docker named volume for persistence
AWS EC2
(Cloud Host for All Containers)
• Ubuntu linux environment




```

---

# **3) Development Environment & Tools**

NebulaStack is developed in a consistent and standardized environment designed to mirror real-world DevOps practices. The development setup closely reflects the production environment to ensure reliability, reduce configuration drift, and maintain predictable behavior across all stages of the software lifecycle. By aligning local and cloud environments using Docker Compose, the application behaves identically in development, testing, and deployment, significantly minimizing integration issues.

## **a) Core Development Components**

### **1. Operating System – Ubuntu Linux:**

A stable, widely adopted Linux distribution favored for containerized and cloud-native development due to its predictable package management and strong community support.

### **2. Editor – VS Code:**

A highly extensible development environment offering Git integration, Docker plugins, Python support, and debugging tools that streamline the entire workflow.

### **3. Runtime – Python 3.12:**

The latest stable version used to run the Flask API, offering improved performance, updated libraries, and long-term support.

### **4. Container Engine – Docker:**
```

Ensures that each component of NebulaStack runs inside an isolated, reproducible container, eliminating dependency conflicts.

5. Orchestrator – Docker Compose:

Used to define and manage the multi-container architecture, ensuring consistent networking, volume management, and environment variables across every environment.

6. Version Control – Git:

Tracks code changes, enables collaborative workflows, manages branches, and integrates directly with CI/CD pipelines.

7. Automation – GitHub Actions:

Powers the continuous integration and continuous deployment process. Every push triggers automated testing, building, and deployment to EC2.

8. Cloud Hosting – AWS EC2 (t3.micro or similar):

The production environment where the complete Docker-based stack is deployed. EC2 provides flexibility, scalability, and full control over the runtime environment.

b) Environment Parity Advantage

NebulaStack maintains strict environment parity between local development and production deployments by using the same Docker Compose configuration in both environments. This approach eliminates configuration drift—one of the most common causes of deployment failures—and ensures that the system behaves consistently regardless of where it runs. Developers build, run, and test the application in a local environment that is structurally identical to the EC2 production environment, resulting in predictable behavior, fewer runtime surprises, and simplified debugging.

- Benefits of Using an Identical Compose Setup

- No surprises during deployment:

The application behaves exactly the same on EC2 as it does locally, reducing unexpected errors in production.

- Identical runtime behavior:

Containers start, interact, and run with the same configuration, dependencies, and environment variables.

- Consistent network structure:

Service names, DNS resolution, and internal communication work the same in both environments.

- Unified environment variables:

Database credentials, API settings, and modes (e.g., production/local) use the same structure, avoiding misconfigurations.

- Matching container lifecycle:

Startup order, volume behavior, restarts, and rebuilds follow an identical lifecycle across environments.

This alignment reflects the core DevOps philosophy of environment parity, ensuring that software behaves reliably at every stage of the development and deployment pipeline.

---

4) Containerisation Strategy

a) API Dockerfile

The API service is packaged inside a Docker image based on the `python:3.12-slim` base image. This minimal Python distribution significantly reduces image size, improves pull times during CI/CD, and lowers the overall attack surface by excluding unnecessary system libraries. The Dockerfile is structured to optimize caching and ensure predictable builds, which is crucial for fast deployments in containerized environments.

Key Advantages of Using `python:3.12-slim`:

- Small footprint: Keeps the final image lightweight and fast.
- Faster pull times: Improves deployment speed on EC2 and CI runners.
- Reduced attack surface: Fewer preinstalled components mean fewer vulnerabilities.

- Optimizations Implemented

- Layered dependency installation: `requirements.txt` is copied and installed separately, allowing Docker to cache dependencies unless the file changes.
- Clean COPY structure: Only the required application files are copied, preventing unnecessary bloat.
- Simple entrypoint: The API runs directly via Python, making debugging and testing straightforward.

- Dockerfile Snippet Used in NebulaStack

```
FROM python:3.12-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
CMD ["python", "app.py"]
```

- Layer Breakdown

- Base layer: Provides the Python runtime and minimal OS utilities.
- Dependency layer: Installs Flask, psycopg2, and other Python packages.
- Application layer: Copies the API source code, including `app.py`.
- Execution layer: Specifies the command executed when the container starts (`python app.py`).

This structure ensures efficient builds and predictable behavior across environments.

b) Nginx Dockerfile / Configuration

The web service uses `nginx:alpine`, chosen for its exceptional performance and extremely small size. Nginx functions both as a static file server and as a reverse proxy that routes API requests to the Flask backend. Its configuration ensures fast delivery, security, and isolation from the rest of the system.

- Static File Hosting

All frontend assets HTML, CSS, images are copied into the container’s public directory:

```
/usr/share/nginx/html
```

This enables Nginx to serve the NebulaStack landing page directly without backend involvement.

- Reverse Proxy Configuration

To expose the API under the same domain, Nginx forwards all requests beginning with `/api/` to the internal Flask container:

```
location /api/ {
proxy_pass http://api:5000/;
}
```

Because the backend lives inside the private Docker network, this proxy configuration allows secure, seamless communication while preventing external access to the API container.

- Security Headers

NebulaStack includes standard HTTP headers recommended by Nginx best practices:

```
add_header X-Frame-Options "DENY";
add_header X-XSS-Protection "1; mode=block";
add_header X-Content-Type-Options "nosniff";
```

These headers prevent clickjacking, cross-site scripting, and MIME-type sniffing attacks, ensuring stronger client-side security.

c) PostgreSQL Container — Rewritten & Expanded

The PostgreSQL database runs inside a dedicated container using the official `postgres:16-alpine` image. This version is optimized for security, performance, and minimal size while maintaining full compatibility with the PostgreSQL ecosystem.

- Why `postgres:16-alpine`?

- Lightweight: Faster to pull, start, and rebuild.
- Secure: Includes only essential components.
- Officially maintained: Ensures consistent updates and vulnerability patches.

- Environment Variables

PostgreSQL is configured using standard environment variables:

- `POSTGRES_DB` – initial database name
- `POSTGRES_USER` – database username
- `POSTGRES_PASSWORD` – secure password

These values are injected via Docker Compose, allowing easy management and environment-specific overrides.

- Persistent Volume

To prevent data loss across container restarts or deployments, PostgreSQL uses a Docker-managed volume:

```
db-data:/var/lib/postgresql/data
```

This guarantees persistent storage even when the container is rebuilt during CI/CD operations.

---

5. Docker Compose Orchestration

Docker Compose acts as the central orchestration layer for NebulaStack, defining how each container is built, connected, configured, and executed. Using a single YAML configuration, Compose ensures that all services behave consistently across local development and production environments.

a) Key Features

Docker Compose provides several critical capabilities that make the multi-container architecture predictable, secure, and easy to manage. These features ensure smooth communication between services, reliable startup behavior, and consistent configuration across environments.

- Private bridge network – All services communicate internally through a Docker-managed private network, ensuring isolation from external traffic.
- Named volume for database – PostgreSQL uses a persistent Docker volume to preserve data during container rebuilds or deployments.
- Dependency ordering – `depends_on` instructs Compose to start essential services (like the database) before dependent ones (like the API or Nginx).
- Proxy routing – The web service forwards `/api/` traffic to the Flask API inside the private network.
- Runtime environment variables – Database credentials, hostnames, and configuration values are injected directly through Compose for clean and environment-agnostic setups.

b) Why Compose?

Docker Compose is chosen because it dramatically simplifies multi-container management while ensuring consistent and reproducible behavior across teams and environments. It eliminates complex manual setup and provides a unified way to run the entire stack with a single command.

- Reproducible environments – The same configuration works locally, in CI, and in production, reducing environment drift.
- Minimal setup overhead – Developers need only `docker compose up` to bring the entire application online.
- Multi-container orchestration – Networking, volumes, environment variables, and service boundaries are handled automatically.
- Simple scaling possibilities – Services such as the API can be scaled horizontally using Compose's built-in scaling feature.

---

6) Back-end API Design

a) Endpoint Contracts

The back-end API exposes a minimal and well-defined set of RESTful endpoints that provide message management functionality and basic system diagnostics. Each endpoint follows predictable behavior and returns structured JSON responses to ensure compatibility with clients and monitoring tools.

Endpoint	Method	Description	Returns
`/`	GET	Confirms the API is reachable	Plain text
`/health`	GET	Reports API runtime health and environment	JSON
`/db-test`	GET	Verifies connectivity with PostgreSQL	JSON
`/messages`	GET	Retrieves a list of all stored messages	JSON array
`/messages`	POST	Creates a new message entry	JSON response + `201 Created`

These endpoints form the core interface between the front-end and the database layer and follow standard REST conventions for clarity and consistency.

---

b) Request Flow

Every incoming API request follows a structured path through the NebulaStack architecture to ensure predictable, secure, and efficient handling. The flow below describes how a typical client request is processed:

1. The client sends a request to the application via the public endpoint hosted by Nginx.
2. Nginx routes all `/api/*` paths to the internal Flask container through its reverse proxy rules.
3. The Flask API validates incoming data, particularly for POST requests where message content must be checked.
4. After validation, the API interacts with PostgreSQL using `psycopg2` to execute queries or retrieve data.
5. Each operation is logged in structured JSON format, allowing for clean observability and easier debugging.
6. The API returns the response back through Nginx, which forwards it to the client.

This request cycle ensures separation of responsibilities: Nginx handles routing and security, Flask processes logic, and PostgreSQL manages persistent data.

---

7) Database Schema

The database layer in NebulaStack is intentionally kept simple, efficient, and optimized for the application's core purpose. All data is stored in a single relational table named `messages`, which is created automatically by the backend on the first API interaction. This minimal structure supports fast reads and writes while ensuring reliability and consistency through PostgreSQL's ACID compliance.

Table Structure

The `messages` table consists of the following fields:

- `id` – Auto-incrementing primary key used to uniquely identify each message.
- `content` – The actual message text submitted by the client.
- `created_at` – Timestamp automatically assigned when a record is inserted, allowing accurate chronological sorting.

Benefits of This Schema

- Natural ordering:

Messages can be easily sorted by creation time without additional logic.

- Minimal schema design:

Lightweight structure ensures fast queries and simplifies the backend logic.

- Ideal for CRUD workflows:

Perfectly supports create, read, and basic retrieval patterns used in the application.

- Easy extensibility:

New fields such as `author`, `status`, or `tags` can be added without refactoring the core architecture.

---

8) Nginx Reverse Proxy

) Why Reverse Proxy?

- Security: API directly expose nahi hoti; Nginx middle layer ban jata hai.
- Performance: Static files cache hoti hain, load fast hota hai.
- Smart Routing: URLs clean rehte hain, `/api/` traffic backend ko smoothly forward hoti hai.
- Centralized Protection: Security headers ek jagah maintain hote hain.

b) Error Handling

Nginx gracefully handle karta hai:

- API downtime: Agar backend dead bhi ho, user ko clean error milta hai.
- Slow responses: Timeout rules se freeze nahi hota.
- Connection failures: Retry logic aur proper fallback behavior deta hai.

---

9) Networking & Ports

Docker networking uses a built-in DNS resolver.

Each service becomes reachable by its Compose service name:

Service	Hostname	Purpose
API	`api`	DB access, proxy routing
DB	`postgres`	API database host
Web	`web`	Public exposure

No container exposes DB to the public network, reducing risk.

Outgoing rules allow only:

- API → DB
- Web → API

---

10) Environment Variables

Environment variables provide decoupling between code and configuration.

Good practices shown:

- Separation of secrets
- Runtime configurability
- Different values for dev/prod

---

11) HTTP Status Codes

Each endpoint returns precise industry-standard status codes:

- 200 – Successful query
- 201 – New record created
- 400 – Invalid request body
- 404 – Record missing
- 500 – Unexpected internal failure

These align with REST best practices.

---

12) CI/CD Workflow

a) Pipeline Structure

The pipeline contains:

The CI/CD pipeline consists of four stages: building the application, optional linting, testing, and automated deployment to EC2.

- Build
- Lint (optional)
- Test
- Deployment to EC2

b) Job: Build/Test

This job checks out the repository, sets up Python, installs dependencies, executes tests with `pytest`, and builds the required Docker images.

- Checks out repo
- Installs Python
- Installs dependencies
- Runs tests using `pytest`
- Builds Docker images

c) Job: Deployment

Deployment connects to the EC2 instance using GitHub Secrets, pulls the latest code, and executes `docker compose up -d --build` to rebuild and restart all containers.

- Establishes SSH connection with EC2 using GitHub Secrets
- Pulls latest code
- Runs `docker compose up -d --build`
- Automatically restarts containers

d) Why GitHub Actions?

GitHub Actions provides cloud-hosted runners, secure secrets management, version-controlled workflows, and cost-free execution for public repositories.

- Cloud-hosted runners
- Secrets management
- Workflow as code
- Zero-cost for public repos

---

13) AWS EC2 Deployment

The server is prepared with all required Docker tooling, proper access control, and a stable network configuration to support automated deployments. An Elastic IP ensures the pipeline always targets the same machine, preventing failures caused by changing public IPs.

a) Server Setup

- Install Docker
- Install Docker Compose
- Clone the repository
- Apply correct permissions
- Assign an Elastic IP
- Open required ports (22, 8080)

b) Role of Elastic IP

- Provides a permanent static IP for CI/CD
- Prevents deployment failures caused by IP rotation

---

14) Versioning & Tagging

a) Current Strategy

Manual image tags such as `v1`, `v2`, and similar incremental versions.

b) Future Strategy

- Semantic versioning
- Tags linked to specific commits
- Automatic version tags generated by the CI pipeline

---

15) Security & Secrets

- Security Controls

The system implements multiple layers of security to minimize exposure and protect all critical components. The database stays isolated within the private network, sensitive access is handled through SSH keys and encrypted secrets, and only essential ports remain open. Additional security headers strengthen the web layer against common browser-based attacks.

- Private internal database network
- No public database exposure
- SSH private key authentication instead of passwords
- Strictly limited security group ports
- GitHub Secrets for all sensitive values
- Security headers protecting against XSS and clickjacking

---

16) Testing Strategy

a) Manual Testing

Manual verification is done through direct `curl` requests, browser-level endpoint checks, and container-level testing with Docker Compose. These quick tests ensure that core routes, proxy behavior, and container communication work as expected before automated checks run.

- `curl` endpoint checks
- Browser-based testing
- Compose-level validation

b) Automated Testing

Automated testing is handled through `pytest`, covering health checks and CRUD operations. The CI pipeline blocks deployment until all tests pass, ensuring consistent code quality and preventing faulty builds from reaching production.

- `pytest` for API health + CRUD
- CI requires passing tests before deploy

---

17) Troubleshooting & Challenges

Issue	Cause	Fix
GitHub authentication failure	HTTPS origin	Switched to SSH remote
SSH permissions	Key too open	Applied `chmod 600`
API failing to reach DB	Wrong host	Set `DB_HOST=postgres`
CI asking host verification	First-time SSH prompt	`StrictHostKeyChecking=no`

---

18) Future Improvements

The project can be extended with stronger security, better scalability, and improved observability as it evolves. These enhancements will prepare the stack for production-grade deployment and long-term reliability.

- JWT-based authentication
- Multi-stage Docker builds for optimized production images
- Container health checks
- Migration to Kubernetes
- Horizontal scaling support
- Prometheus/Grafana monitoring
- Centralized external logging

---

19) References

Official documentation references:

- Docker Documentation
- Docker Compose Documentation
- Flask Documentation
- psycopg2 Documentation
- PostgreSQL Documentation
- GitHub Actions Documentation
- Nginx Documentation
- AWS EC2 Documentation

---

20) Source Mapping

Component / Feature	Documentation / Source Reference
Dockerfile syntax	Docker Docs (docker.com)
Compose networks & volumes	Docker Compose Docs
Flask routing & JSON responses	Flask Docs
psycopg2 DB connection	Psycopg2 Docs
PostgreSQL SQL schema	Postgres Docs
Nginx reverse proxy rules	Nginx Docs
Security headers	OWASP + Nginx Docs
CI workflow syntax	GitHub Actions Docs
SSH-based deployments	GitHub Actions Examples
EC2 setup & networking	AWS EC2 Docs

---

21) Final Summary

This project demonstrates how a small, modular stack can achieve production-grade reliability through clean architecture and automated operations. Each component plays a focused role, and together they create a workflow that removes the usual friction of deployment, scaling, and maintenance.

The platform solves the long-standing issues of manual setup, shifting environments, and unrepeatable builds by enforcing consistency from development to production. Every update moves through the same pipeline, every service runs in an isolated container, and every deployment is predictable.

Key benefits include:

- Consistent builds through Dockerized environments
- Zero-touch deployments with GitHub Actions
- Stronger security through isolated networking and controlled exposure
- Stable application routing via Nginx
- Reliable data handling with PostgreSQL
- Clear structure that scales easily into Kubernetes or multi-service setups

Ultimately, this application is a foundation: simple enough to understand end-to-end, yet powerful enough to grow into a fully distributed system. It brings discipline, automation, and clarity to the development process qualities that define modern DevOps engineering.