STRV.social

Chapter 1: Introduction & Overview

1. Introduction

This document serves as a **comprehensive technical guide** to STRV.social. It covers the **architecture, technologies, workflows, and security measures**, including diagrams to illustrate system components and interactions.

2. Project Overview

This social media platform is designed for **content sharing, user interaction, and multimedia support**. It uses a **Django backend** with **HTMX for interactivity**, a **PostgreSQL database**, and an **embedding system** for media analysis.

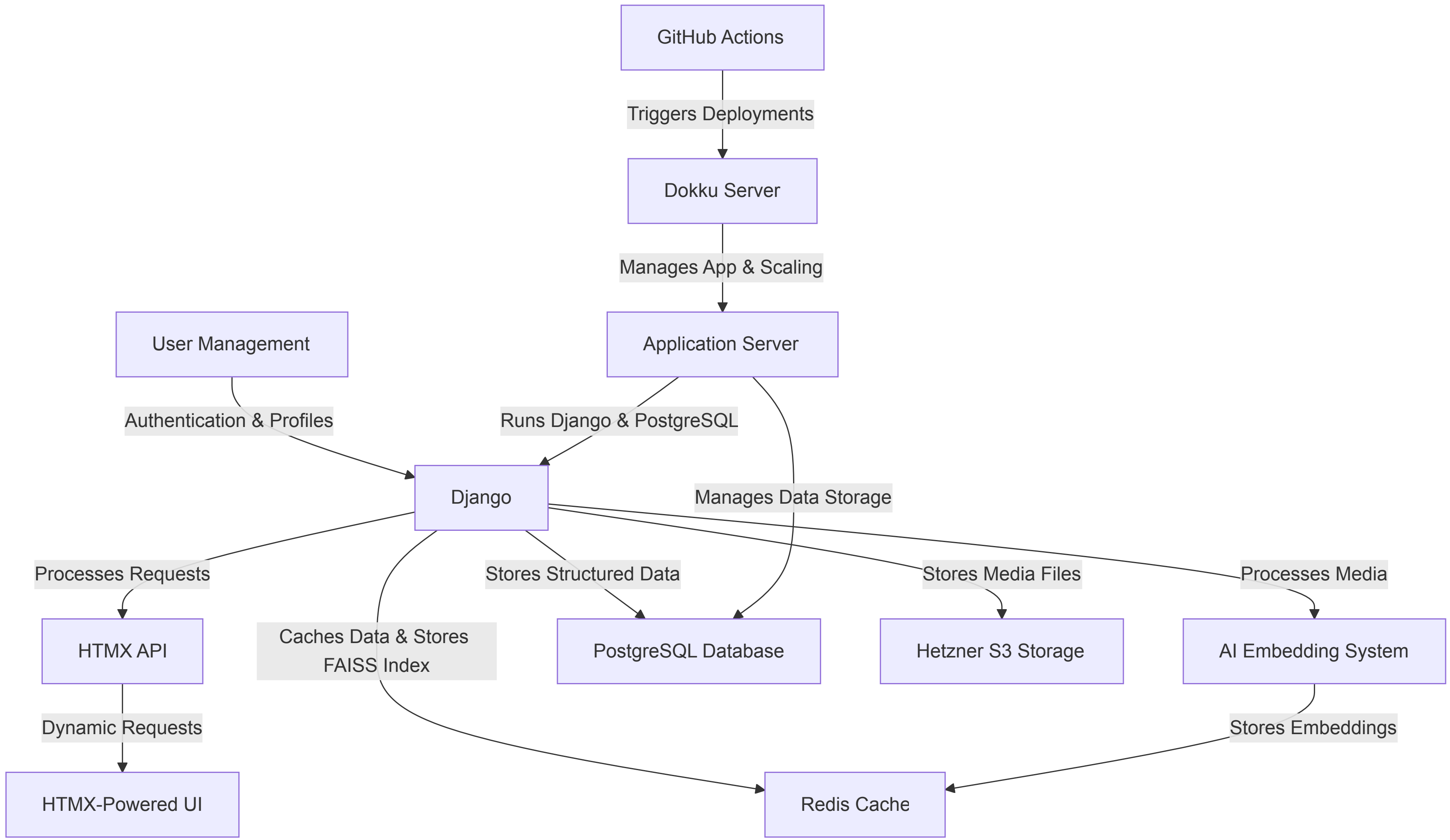
2.1 Key Features

- **User Authentication & Profiles** (Signup, Login, SocialUser model)
- **Content Posting** (Text, Images, GIFs, Audio, Video)
- **Embedding System** (AI-powered content processing)
- **HTMX-Powered Interactivity** (Dynamic updates without page reloads)

- **Docker & Deployment with Dokku**

3. High-Level Architecture

The system is built using a **modular architecture** with separate concerns for users, content, themes, and embeddings.



3.1 Backend Technologies

Component	Technology
Web Framework	Django 5.1
Frontend Interactivity	HTMX
Database	PostgreSQL
Cache + FAISS DB	Redis
Embedding & AI	PyTorch, Faster-Whisper, Librosa
Deployment	Docker, Dokku, GitHub Actions

Chapter 2: System Components & Data Flow

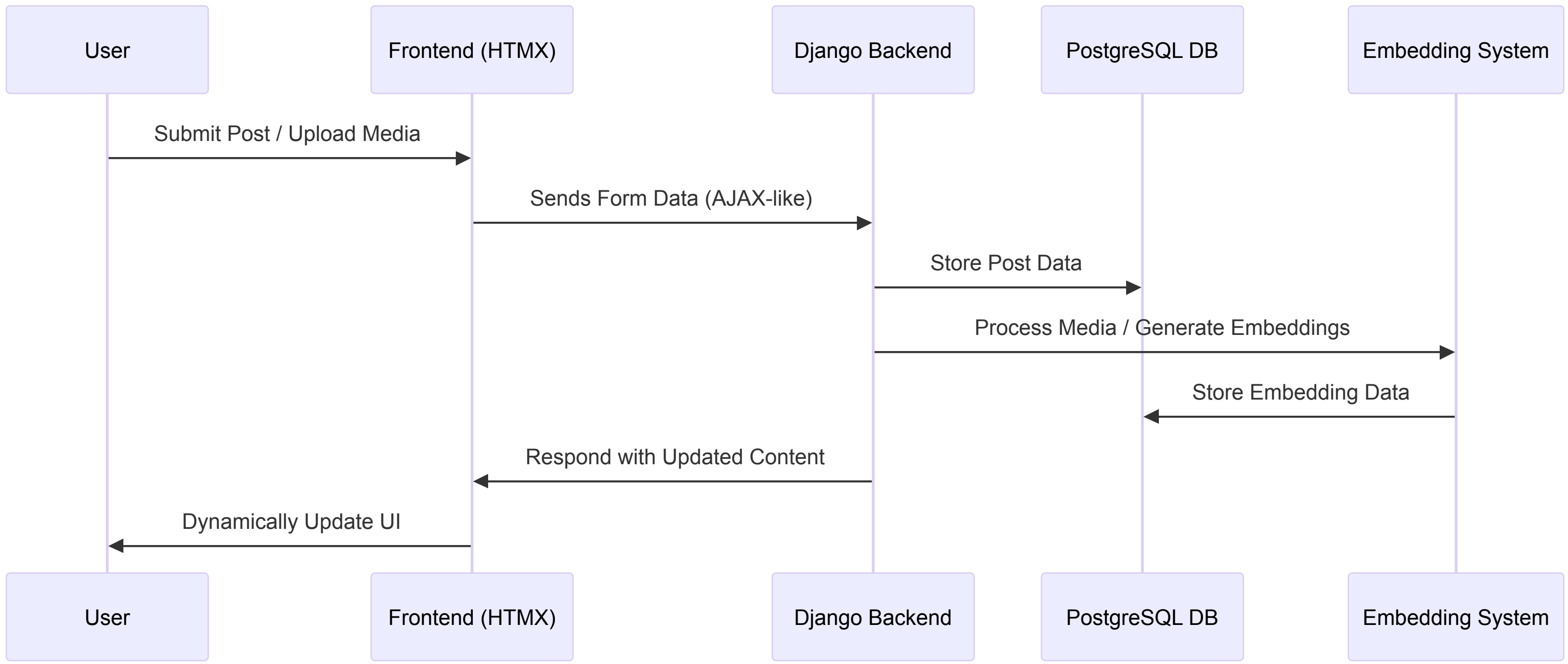
1. System Components Overview

This chapter covers the **core components** of the social media project, explaining their roles and **how they interact**. It also includes diagrams to illustrate workflows.

1.1 Major System Components

Component	Description
User Management	Handles authentication, user profiles, and permissions
Content System	Manages content creation, media uploads, and storage
Embedding System	Uses AI to analyze text, images, audio, and video
HTMX-powered UI	Enables dynamic, partial page updates
Database (PostgreSQL)	Stores users, content, and embeddings
Redis	Stores cache and FAISS indexes
Deployment (Dokku)	Automates deployment and hosting

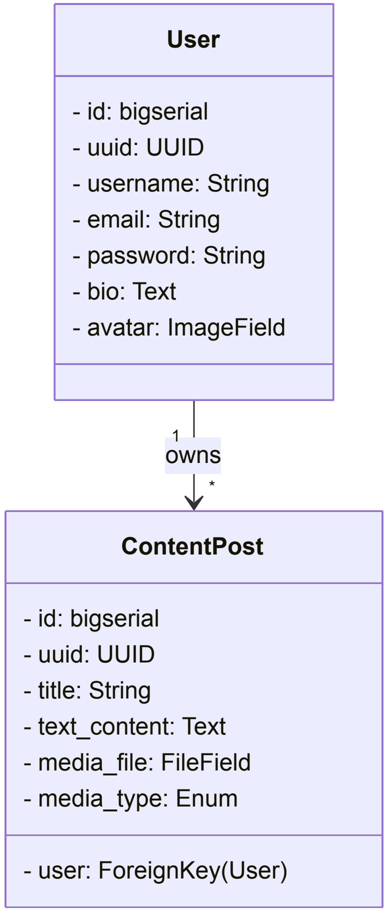
2. Data Flow Overview



3. Detailed Component Interactions

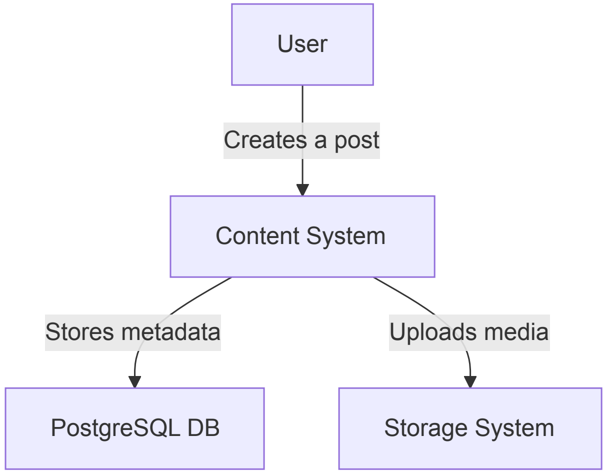
3.1 User Management

- **Signup/Login** handled by Django's authentication system
- Uses **CurrentUserField** to track ownership of posts



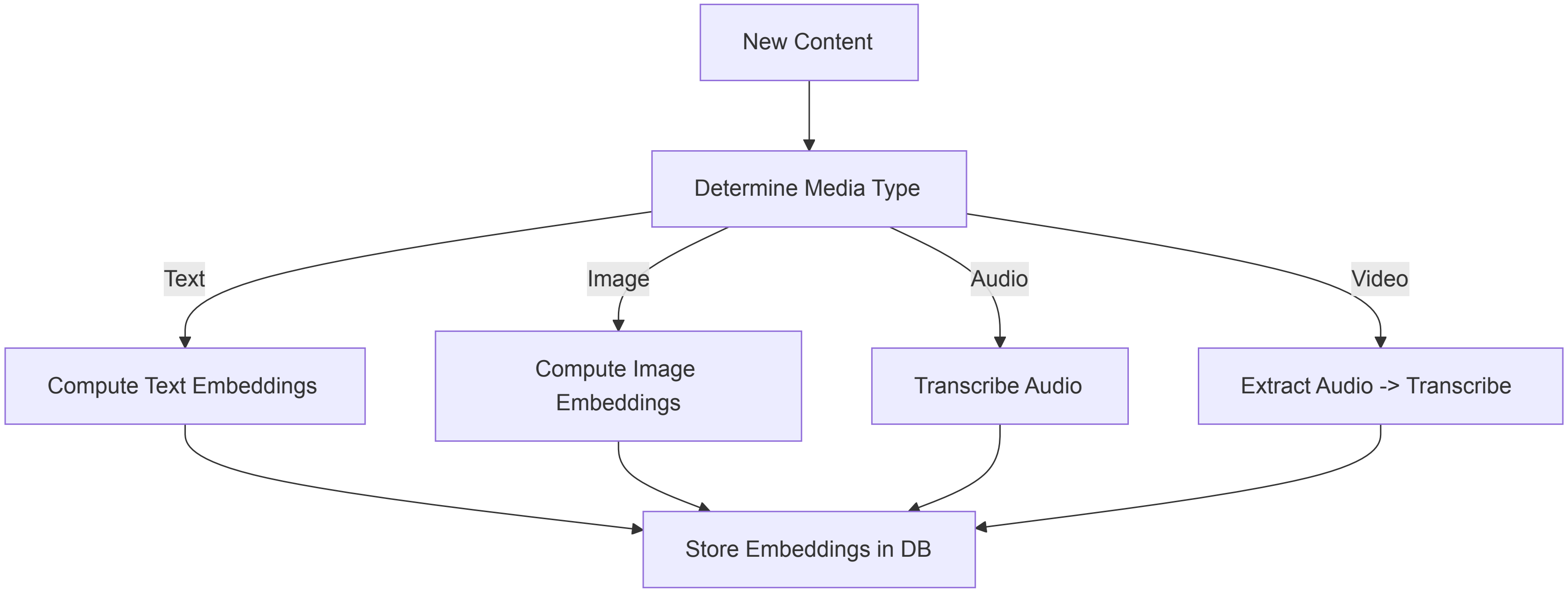
3.2 Content System

- Users can create posts containing **text, images, GIFs, audio, or video**
- Uses **Django ORM** for storing content metadata
- Handles **media uploads** via Django's `FileField`



3.3 Embedding System

- Uses **AI models** to analyze media
- Supports **text, images, audio, and video**
- Stores embeddings in a **JSON field in PostgreSQL**



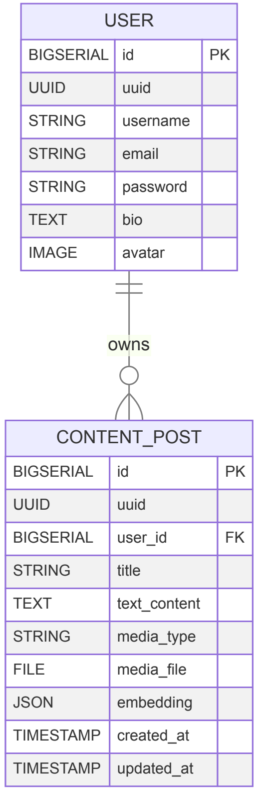
Chapter 3: Database Models & Schema

1. Database Design Overview

We use **PostgreSQL** as the primary database, managed through **Django's ORM**. The schema supports **user accounts, content posts, media embeddings, and media uploads**.

2. Database Schema

Below is an **ER (Entity-Relationship) Diagram** representing the major tables and their relationships.



3. Detailed Model Definitions

3.1 User Model

Represents **registered users**.

```python
class SocialUser(AbstractUser):
    email = models.EmailField(unique=True)
    bio = models.TextField(blank=True, max_length=255)
    avatar = models.ImageField(
        upload_to=partial(generate_random_filename, subdir="avatars"),
        blank=True,
        null=True,
    )
    objects = SocialUserManager()

    USERNAME_FIELD = "email"
    REQUIRED_FIELDS = ["username"]

    def __str__(self):
        return self.email

    @property
    def user(self):
        """Return the user instance. This is a convenience method for consistency."""
        return self

    def get_last_update(self):
        content = self.content.latest()
        return content.updated_at

    def get_avatar_url(self):
        if self.avatar:
            return self.avatar.url

        return static("core/img/user.webp")

    def get_absolute_url(self):
        return reverse("profile-detail", kwargs={"username": self.username})
```

- Extends Django’s built-in `AbstractUser`
- Stores **user bio** and **profile picture**

3.2 ContentPost Model

Represents **posts** with text and/or media.

```python
class ContentPost(models.Model):
    MEDIA_TYPES = [
```

```
        ("text", "Text"),
        ("image", "Image"),
        ("gif", "GIF"),
        ("audio", "Audio"),
        ("video", "Video"),
    ]

    user = models.ForeignKey(SocialUser, on_delete=models.CASCADE, related_name="content")
    title = models.CharField(max_length=255)
    text_content = models.TextField(blank=True, null=True)
    media_type = models.CharField(max_length=10, choices=MEDIA_TYPES)
    media_file = models.FileField(upload_to="uploads/", blank=True, null=True)
    embedding = models.JSONField(blank=True, null=True)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)
```

- Supports **multiple media types**
- Stores **embeddings** for AI-powered recommendations

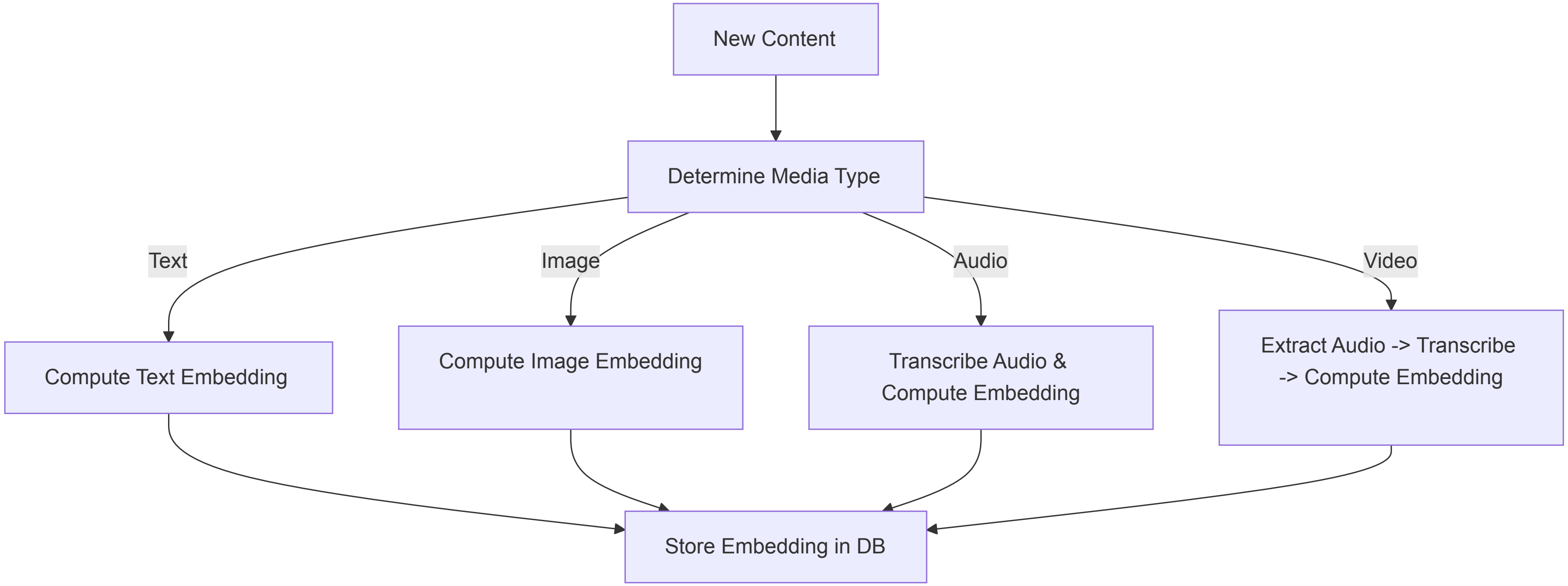
Chapter 4: Content Embedding & AI Processing

1. Overview

This chapter explains how the **embedding system** processes various media types (**text, images, audio, and video**) using **AI models**. Embeddings are used to enable **content analysis and recommendations**.

2. Embedding System Workflow

The embedding system determines **media type** and applies an appropriate AI model to generate embeddings.



3. Text Embeddings

- Uses **DistilBERT** (Transformer model)
- Converts **text into a numerical vector**

```
from transformers import AutoTokenizer, AutoModel
import torch

class EmbeddingProcessor:
    def __init__(self):
        self.text_tokenizer = AutoTokenizer.from_pretrained("distilbert-base-uncased")
        self.text_model = AutoModel.from_pretrained("distilbert-base-uncased")
        self.text_model.eval()

    def compute_text_embedding(self, text: str):
        inputs = self.text_tokenizer(text, return_tensors="pt", truncation=True, padding=True)
        with torch.no_grad():
            outputs = self.text_model(**inputs)
        return outputs.last_hidden_state.mean(dim=1).squeeze().tolist()
```

4. Image Embeddings

- Uses **ResNet18** for feature extraction
- Converts images into **vector representations**

```
import torchvision.models as models
import torchvision.transforms as transforms
from PIL import Image

class EmbeddingProcessor:
    def __init__(self):
        self.image_model = models.resnet18(pretrained=True)
        self.image_transform = transforms.Compose([
            transforms.Resize((224, 224)),
            transforms.ToTensor(),
            transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
        ])

    def compute_image_embedding(self, file_obj):
        image = Image.open(file_obj).convert("RGB")
        image_tensor = self.image_transform(image).unsqueeze(0)
        with torch.no_grad():
            embedding = self.image_model(image_tensor).squeeze().tolist()
        return embedding
```

5. Audio & Video Processing

For **audio**, the system:

1. **Transcribes speech** (using Whisper)
2. **Computes text embedding**
3. If no speech is detected, extracts **audio features**

For **video**, the system:

1. **Extracts audio** (using FFmpeg)
2. **Processes it as an audio file**

```
import librosa
import numpy as np
from faster_whisper import WhisperModel

class EmbeddingProcessor:
    def transcribe_audio(self, audio_path):
        model = WhisperModel("base", device="cuda" if torch.cuda.is_available() else "cpu")
```

```
        segments, _ = model.transcribe(audio_path, beam_size=5)
        return " ".join(segment.text for segment in segments)

    def compute_audio_embedding(self, audio_path):
        y, sr = librosa.load(audio_path, sr=22050)
        mel_spec = librosa.feature.melspectrogram(y=y, sr=sr)
        return np.mean(mel_spec, axis=1).tolist()
```

6. Storing Embeddings

- **Embeddings are stored as JSON** in the database
- **Each post has an embedding field**

```
class ContentPost(models.Model):
    embedding = models.JSONField(blank=True, null=True)
```

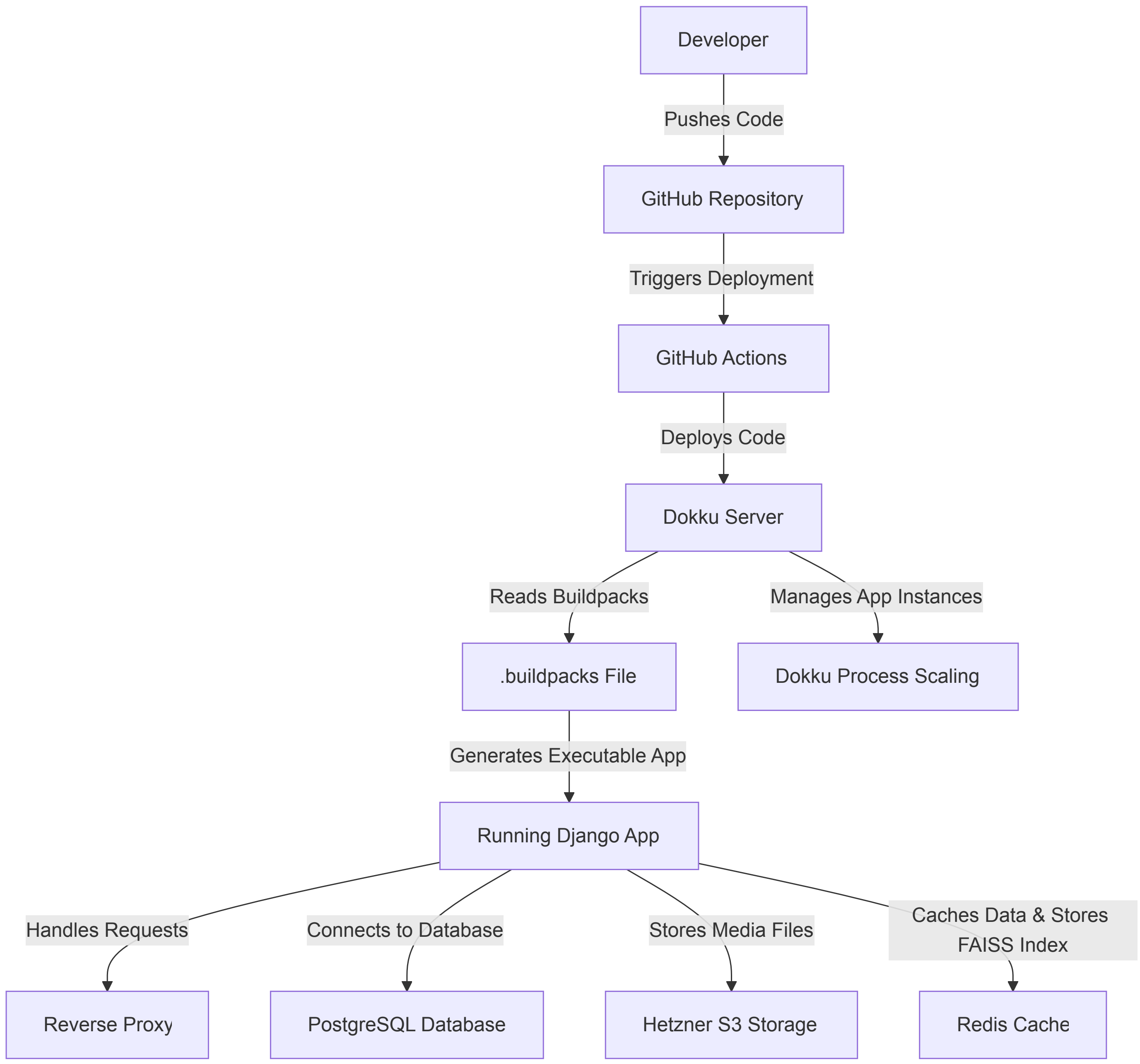
Chapter 5: Deployment & Infrastructure

1. Overview

This chapter details the **deployment process**, focusing on **Dokku with buildpacks**, **GitHub Actions for CI/CD**, and **Hetzner S3 for media storage**. The goal is to ensure a **smooth and automated deployment** while keeping development flexible with Docker for local use.

2. Deployment Architecture

The system is **developed locally with Docker** but deployed using **Dokku and Heroku buildpacks**.



Deployment Highlights

- Local development uses Docker for easy setup

- Production deployment is handled by Dokku using buildpacks
- GitHub Actions automates deployment
- Nginx acts as the reverse proxy
- Hetzner S3 stores media files
- PostgreSQL is used for database storage

3. Local Development with Docker

3.1 Docker Configuration

For local development, **Docker ensures an isolated environment**.

To build the project:

```
docker compose build
```

To launch the project:

```
docker compose up
```

4. Production Deployment with Dokku

4.2 Using Buildpacks in Dokku

Instead of Docker, Dokku **automatically detects and uses buildpacks** from the `.buildpacks` file.

Defining Buildpacks (`.buildpacks`)

```
https://github.com/heroku/heroku-buildpack-nodejs.git
https://github.com/heroku/heroku-buildpack-python.git
https://github.com/heroku/heroku-buildpack-activestorage-preview # FFMPEG
```

Dokku reads this file and **applies the necessary buildpacks** during deployment.

5. Automating Deployment with GitHub Actions

5.1 Setting Up GitHub Actions for Deployment

GitHub Actions ensures **automatic deployments** when code is pushed to the `main` branch.

GitHub Actions Workflow (`.github/workflows/deploy.yml`)

```yaml
name: 'Deploy to Dokku'

on:
  push:
    branches:
      - main

jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Cloning repo
        uses: actions/checkout@v4

      - name: Push to Dokku
        uses: dokku/github-action@master
        with:
          git_remote_url: ${{ secrets.DOKKU_REMOTE_URL }}
          ssh_private_key: ${{ secrets.DOKKU_PRIVATE_KEY }}
          branch: 'main'
```

5.2 Setting Up Secrets in GitHub

Store **Dokku credentials** as **GitHub Secrets** for security.

Secret Name	Value
`DOKKU_REMOTE_URL`	`ssh://dokku@nest.unarlabs.co:22/strvsocial`
`DOKKU_PRIVATE_KEY`	**SSH private key** matching public key on Dokku server

This setup ensures **fully automated deployments** whenever code is pushed to `main`.

6. Environment Configuration

6.1 Managing Configuration Variables

Dokku uses `config:set` to store **sensitive credentials** securely.

```
dokku config:set strvsocial DJANGO_SECRET_KEY=your-secret-key
```

```
dokku config:set strvsocial ALLOWED_HOSTS=strv.social
```

To check stored variables:

```
dokku config strvsocial
```

7. Scaling & Process Management

7.1 Scaling the Application

Dokku allows process scaling like Heroku.

```
dokku ps:scale myapp web=2
```

This **runs 2 instances** of the Django app.

7.2 Managing Application Logs

To monitor logs:

```
dokku logs myapp --tail
```

7.3 Restarting the App

```
dokku ps:restart myapp
```

Chapter 6: Future Features & Scaling Plan

1. Overview

To scale from a **small deployment** to a system supporting **hundreds of thousands of users**, the architecture must evolve. This chapter outlines:

- **Future feature enhancements**
- **Scalability challenges**
- **A step-by-step scaling plan**

2. Future Features

2.1 Improved Search & Recommendations

- **Full-text search using PostgreSQL's `pg_trgm` or Elasticsearch**
- **Content recommendations using AI-generated embeddings with larger context**
    - Include tags, better video and audio handling, engagement and interactions metrics
- **Indexing system to improve retrieval speed**

2.2 Background Processing & Async Tasks

- **Move embedding generation to Celery tasks**
- **Implement a task queue with Redis for handling async jobs**
- **Process media uploads asynchronously (e.g., transcoding videos, extracting audio, generating indexes)**

2.3 Enhanced Media Handling

- **Support for adaptive streaming (HLS for video, waveform generation for audio)**
- **Automatic thumbnail and preview generation**
- **Move from Hetzner S3 to a globally distributed object storage (e.g., AWS S3, Cloudflare R2)**

2.4 Microservices for AI Processing

- **Move embedding generation and indexing to separate microservices**
- **Deploy AI services on GPU-accelerated cloud instances**
- **Use FastAPI for high-performance AI microservices**

3. Scaling Plan: From Small Deployment to Cloud Infrastructure

3.1 Current Bottlenecks in the Dokku Deployment

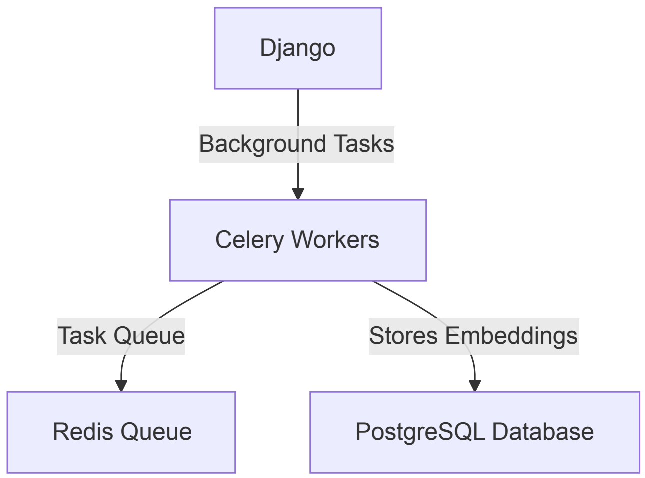
- **Limited Vertical Scaling** – Single PostgreSQL database will struggle with large queries
- **Single App Server** – A single Dokku instance cannot handle high concurrent requests

To scale, we need to **introduce cloud-based infrastructure, async processing, and distributed computing**.

4. Step-by-Step Scaling Plan

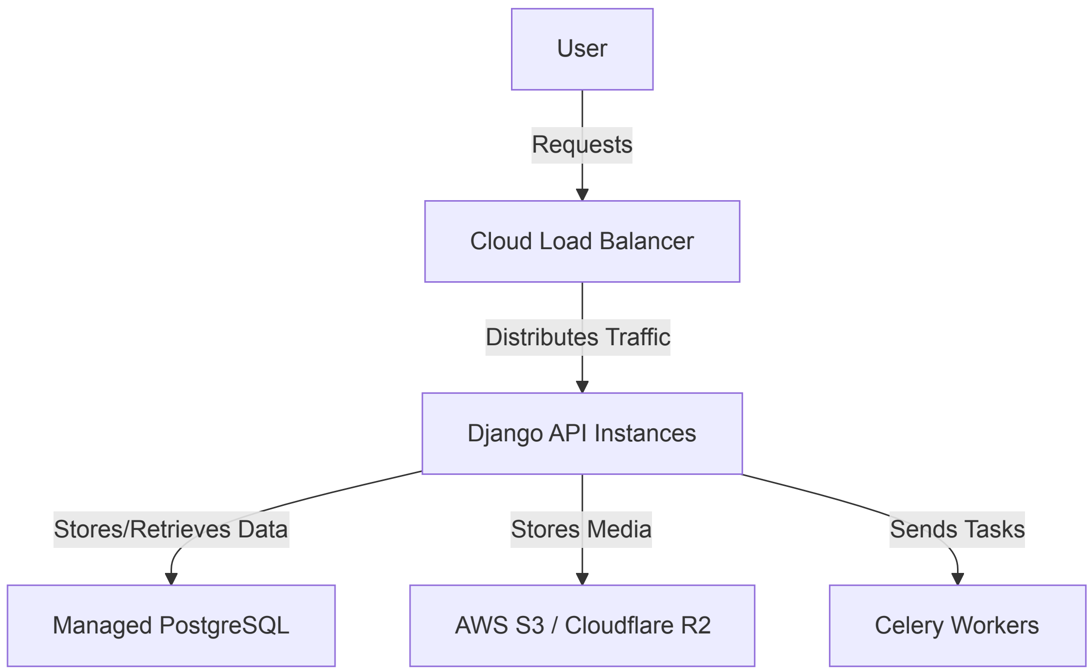
Phase 1: Optimize Current Architecture

🔹 **Enable PostgreSQL connection pooling** (`pgbouncer`)
🔹 **Optimize queries and add caching (Redis) for API responses**
🔹 **Move AI-related tasks to Celery workers instead of synchronous execution**



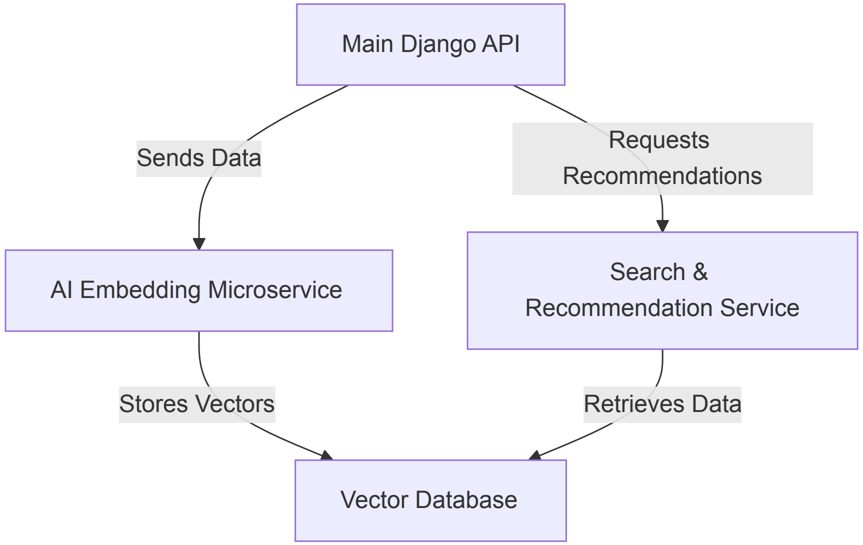
Phase 2: Move to Scalable Cloud Infrastructure

🔷 **Migrate from Dokku to Kubernetes or Nomad on a cloud provider (AWS/GCP/Azure)**
🔷 **Move PostgreSQL to a managed cloud database (AWS RDS, Google Cloud SQL)**
🔷 **Use Cloudflare R2 or AWS S3 for media storage**



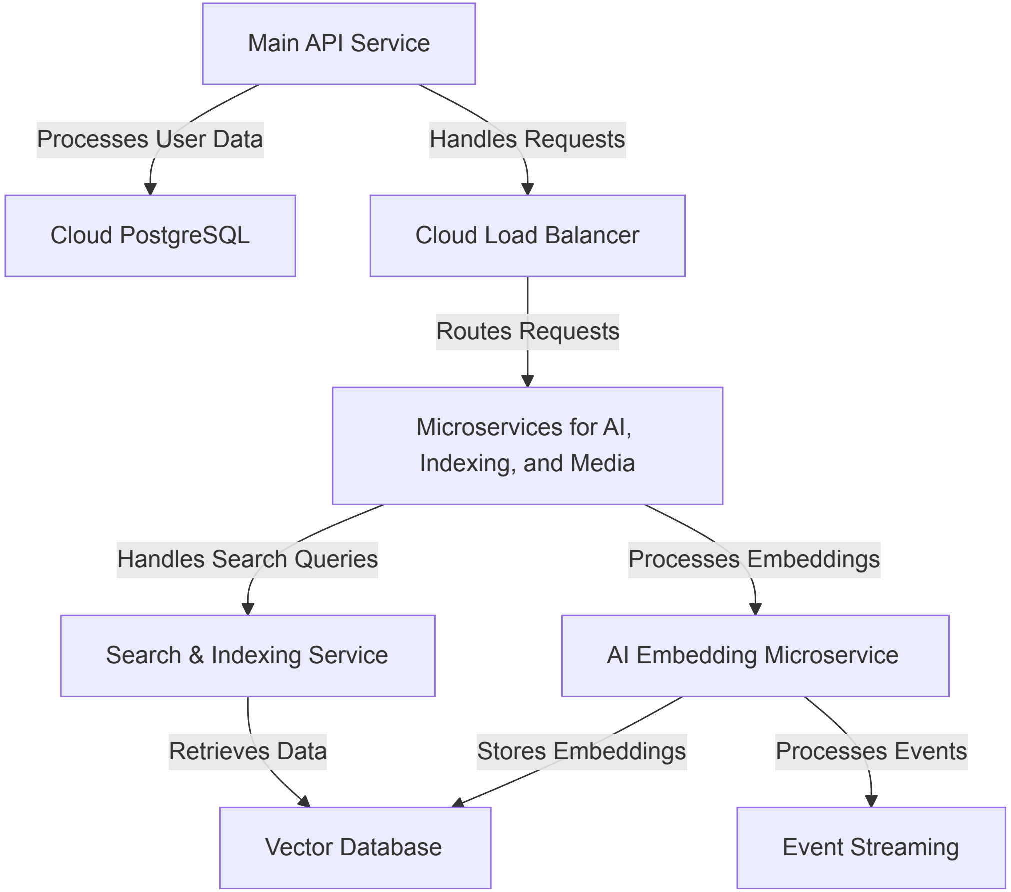
Phase 3: Introduce Microservices for AI Processing

- 🔷 Move embedding generation to a dedicated AI microservice
- 🔷 Deploy AI services with FastAPI and TensorFlow/PyTorch on GPU instances
- 🔷 Create a separate microservice for indexing and recommendation systems



Phase 4: Full Distributed Architecture with Microservices

- 🔷 Deploy all services as independent microservices
- 🔷 Use Kubernetes (K8s) or Nomad for container orchestration
- 🔷 Implement event-driven processing with Kafka for real-time indexing
- 🔷 Run AI services separately on cloud-based GPU clusters



5. Roadmap to Large-Scale Deployment

Phase	Changes	Estimated Capacity
1: Optimize Current Setup	Caching, query optimization, async tasks	10,000 users
2: Move to Cloud	Kubernetes, managed PostgreSQL, object storage	100,000 users
3: AI Microservices	Dedicated GPU instances, AI embeddings	500,000 users
4: Full Microservices Architecture	Kafka, event-driven indexing, scalable API services	Millions of users