Google Summer of Code

Proposal Gsoc 2025

Migrate IHR API for Internet Health Report

by

Shashwat Darshan

Gsoc Proposal Internet Health Report

I. About Me

Name: Shashwat Darshan\ **Email:** shashwatdarshan153@gmail.com\ **GitHub:** Shashwat-Darshan\
Location: New Delhi, India\ **Timezone:** GMT+5:30\ **Resume:** Link

I am an enthusiastic software developer with a deep passion for backend technologies, system architecture, and high-performance computing. I have extensive hands-on experience with Python, Django, FastAPI, SQL, and containerization tools such as Docker. My technical foundation is built upon rigorous project-based learning and real-world applications, where I have successfully developed and optimized backend systems. Although my previous projects have been on a smaller scale, I am eager to undertake this significant migration project and work alongside experienced mentors at IHR to develop an efficient, high-performing, and maintainable backend architecture. I see this as an opportunity to further refine my technical capabilities, expand my problem-solving skills, and contribute meaningfully to an open-source initiative that impacts a global audience.

II. Why I Decided to Work for IHR

I am thrilled at the prospect of contributing to the Internet Health Report (IHR) project because it presents a combination of complex technical challenges and a mission-driven approach that strongly aligns with my skills and values. The chance to migrate the IHR API from an outdated Django framework to the modern FastAPI, streamline database management using Bash scripts, and deploy the application efficiently through Docker aligns perfectly with my technical expertise in Django, FastAPI, SQL, and containerization technologies.

Beyond the technical aspects, I am deeply passionate about the project's overarching goal of enhancing internet resiliency and accessibility—two critical components in today's connected world. By participating in this initiative, I will have the opportunity to collaborate with a diverse and global developer community, gain invaluable hands-on experience in large-scale system migration, and contribute to an essential tool that informs and empowers millions. The combination of technical challenge, real-world impact, and professional growth makes this a compelling and deeply rewarding endeavor for me.

III. Project Title

Migrating IHR Backend from Django 2.2 to FastAPI for Improved Performance, Scalability, and Maintainability

IV. Abstract

The current IHR backend is built on Django 2.2.27, an outdated version of the Django web framework that lacks support for modern asynchronous programming. This results in performance bottlenecks under high-concurrency conditions, limits the system's ability to efficiently scale, and increases the complexity of maintaining and extending the codebase. As the number of users and data interactions continues to grow, the system's synchronous nature poses challenges in handling concurrent requests effectively, often causing slowdowns and reduced responsiveness during peak traffic periods.

This proposal outlines a migration plan to FastAPI, a high-performance, asynchronous Python web framework specifically designed for building APIs with modern development practices. FastAPI's built-in support for asynchronous I/O, automatic OpenAPI documentation generation, and dependency injection will not only improve the backend's responsiveness but also simplify the overall architecture and enhance maintainability. Furthermore, replacing Django's ORM with optimized raw SQL scripts executed via Bash, as per the project's requirements, will enable more efficient and transparent database interactions. These changes are expected to significantly boost system performance, enhance developer productivity, and lay the foundation for a more scalable, secure, and future-ready backend infrastructure for the IHR project.

V. Key Objectives and Motivation

Key Objectives

- **API Migration:** Migrate all Django API endpoints to FastAPI using asynchronous processing to reduce latency, increase throughput, and improve scalability.
- **Database Management:** Implement Bash scripts for database initialization, schema migrations, and management using raw SQL for greater flexibility and performance, following the project requirements.
- **Containerization & Documentation:** Dockerize the FastAPI application for seamless deployment across different environments and auto-generate comprehensive API documentation using FastAPI's built-in OpenAPI support.
- **Security:** Maintain existing Django token authentication via JWT in FastAPI.

Motivation for Migration

- **Performance Bottlenecks:** The synchronous nature of Django's views and operations causes slow response times under heavy loads. Asynchronous processing in FastAPI will allow the system to handle a larger number of concurrent requests more efficiently.
- **Maintainability Issues:** The current monolithic codebase is complex and difficult to maintain. Adopting a modular design with type-safe models will simplify the code and improve long-term maintainability. The use of Bash scripts for DB management aligns with project goals but will require careful implementation for robust state tracking and error handling.
- **Scalability Constraints:** As data volume grows and user demand increases, the existing Django-based architecture struggles to scale. FastAPI's support for asynchronous requests, combined with potentially optimized database interactions, will enhance the system's scalability.
- **Security:** The modernization process will maintain a secure authentication mechanism using JWT, ensuring continuity and familiarity. Migration Challenges and Solutions

VI. Expected Outcomes

- **Substantial improvements in API performance** through asynchronous request handling and optimized query execution.
- **A more modular, maintainable, and scalable backend architecture** that is easier to extend and optimize in the future.
- **Enhanced developer experience** through automatic API documentation, type safety, and better error handling.
- **Improved system resilience and fault tolerance**, ensuring the IHR API can handle increasing workloads efficiently.
- **Performance Targets:**

- 95% of API endpoints respond under **200ms** at 1000 RPM
- Database initialization completes in < 30 seconds

Future Work (Subject to Mentor Approval)

- [ ] CI/CD pipeline using GitHub Actions
- [ ] Enhanced monitoring with Prometheus/Grafana
- [ ] Integrate Celery for handling background tasks asynchronously
- [ ] WebSocket support for real-time features
- [ ] Kubernetes deployment configuration

Database Management with Bash Scripts

1. Directory Structure

```
ihr_fastapi/
    ├── db_scripts/
    │ ├── init_db.sh
    │ ├── migrate_db.sh
    │ ├── backup_db.sh
    │ ├── test_db.sh
    │ └── seed_db.sh
    └── sql/
     ├── schema/
     │ ├── 01_tables.sql
     │ └── 02_indexes.sql
     └── migrations/
     ├── 20240601_add_hegemony_timebin_column.sql
     └── 20240615_alter_asn_table.sql
```

Note: Careful implementation will be required to ensure these scripts are robust, handle errors gracefully, and manage migration state effectively.

Docker Integration

Dockerfile:

```
FROM python:3.11-slim-buster

    WORKDIR /app

    COPY requirements.txt .
    RUN pip install --no-cache-dir -r requirements.txt

    COPY ./app /app/app
    COPY ./db_scripts /app/db_scripts
    COPY ./sql /app/sql

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

docker-compose.yml:

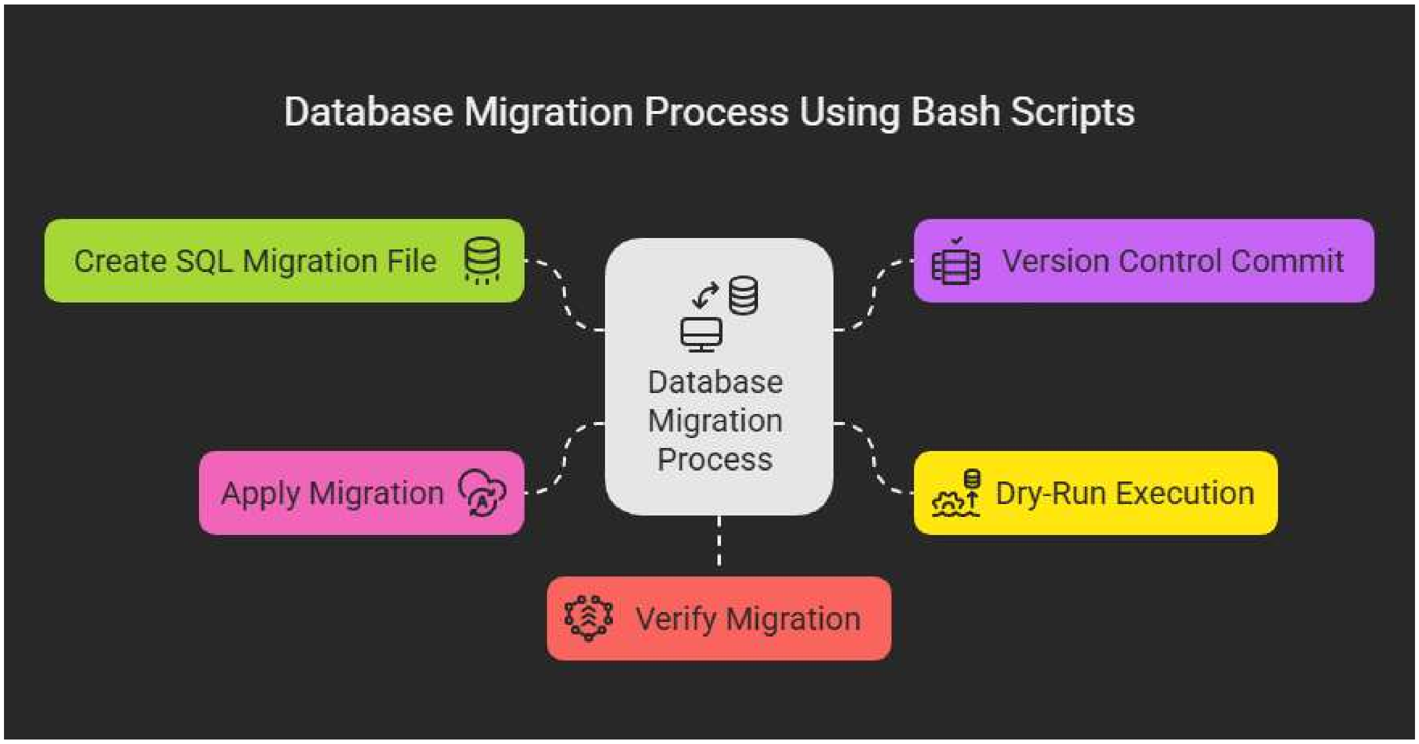
```
version: \'3.8\'
    services:
     app:
     build: .
     ports: [\"8000:8000\"]
     environment:
     - POSTGRES_USER=${POSTGRES_USER}
     - POSTGRES_DB=${POSTGRES_DB}
     # Add other necessary environment variables (e.g., JWT secret, POSTGRES_PASSWORD)
     depends_on:
     - postgres
     postgres:
     image: postgres:14-alpine
     environment:
     POSTGRES_USER: ${POSTGRES_USER}
     POSTGRES_DB: ${POSTGRES_DB}
     POSTGRES_PASSWORD: ${POSTGRES_PASSWORD} # Ensure password is set
     volumes:
     - pgdata:/var/lib/postgresql/data

    volumes:
     pgdata:
```

Migration Strategy

A complete switch to FastAPI is the preferred migration strategy. The migration will be executed in a phased manner, focusing on migrating the backend component by component to ensure stability and maintainability.

The migration will proceed component by component, starting with the database layer integration using Bash scripts, followed by the API endpoints, and then the authentication mechanisms.

Database Layer (Bash-based approach)


Database Migration Process Using Bash Scripts
Create SQL Migration File
Version Control Commit
Database Migration Process
Apply Migration
Dry-Run Execution
Verify Migration

1. Component-wise Migration

The database management will be handled primarily through Bash scripts for schema initialization, migrations, seeding, and general management, interacting with the PostgreSQL database using raw SQL, as specified in the project idea. **Emphasis will be placed on making these scripts robust, including error handling, logging, and clear state management (e.g., tracking applied migrations).** The FastAPI application will use `asyncpg` to interact with the database asynchronously, potentially executing SQL directly or invoking helper functions that utilize the defined SQL files/logic.

API Endpoints

Existing Django API endpoints will be migrated to FastAPI, leveraging asynchronous programming capabilities via `async` / `await` and `asyncpg`. The goal is to maintain 100% API compatibility during this transition.

```
# Example using asyncpg directly (preferred for async)
    import asyncpg
    from fastapi import APIRouter, Depends, HTTPException
    from typing import List
    # Assuming a dependency function get_db_pool() provides an asyncpg pool
    # from app.core.database import get_db_pool

    router = APIRouter()

    @router.get(\"/hegemony/\", response_model=List[dict])
    async def get_hegemony(timebin: str = None, pool: asyncpg.Pool = Depends(get_db_pool)):
     # Migrate existing Django view logic to async
     # Utilize asyncpg for database interactions
     try:
     # Example query (replace with actual logic based on SQL files)
     query = \"SELECT * FROM hegemony_data WHERE timebin = $1;\" # Parameterized query
     results = await pool.fetch(query, timebin)
     # Convert asyncpg Records to dictionaries if needed for the response model
     return [dict(record) for record in results]
     except asyncpg.PostgresError as e:
     # Proper logging should be added here
     raise HTTPException(status_code=500, detail=f\"Database error: {e}\")
     except Exception as e:
     # Proper logging should be added here
     raise HTTPException(status_code=500, detail=f\"An unexpected error occurred: {e}\")
```

2. Migration Phases

(This subsection outlines the phases described in detail in Section VII below)

1. **Phase 1: Initial Setup and Core Database Scripting (2 weeks)**
2. **Phase 2: Database Scripting & Schema Conversion (3 weeks)**
3. **Phase 3: API Endpoint Migration (4 weeks)**
4. **Phase 4: Authentication and Security Implementation (2 weeks)**
5. **Phase 5: Optimization, Testing, and Deployment Prep (2 weeks)**

3. Key Components to Migrate

1. **Database Management:**

- Schema initialization (`init_db.sh`)
- Migrations (`migrate_db.sh` with state tracking)

- Backup and restoration (`backup_db.sh`, `restore_db.sh`)
- Database seeding (`seed_db.sh`)

2. **API Endpoints:** *(Examples)*

- `/hegemony/`
- `/hegemony/countries/`
- `/hegemony/asns/`
- User management endpoints (if applicable)

3. **Authentication:**

- Token-based auth → JWT (generation, validation middleware)
- Password hashing (`passlib`)
- Permissions logic

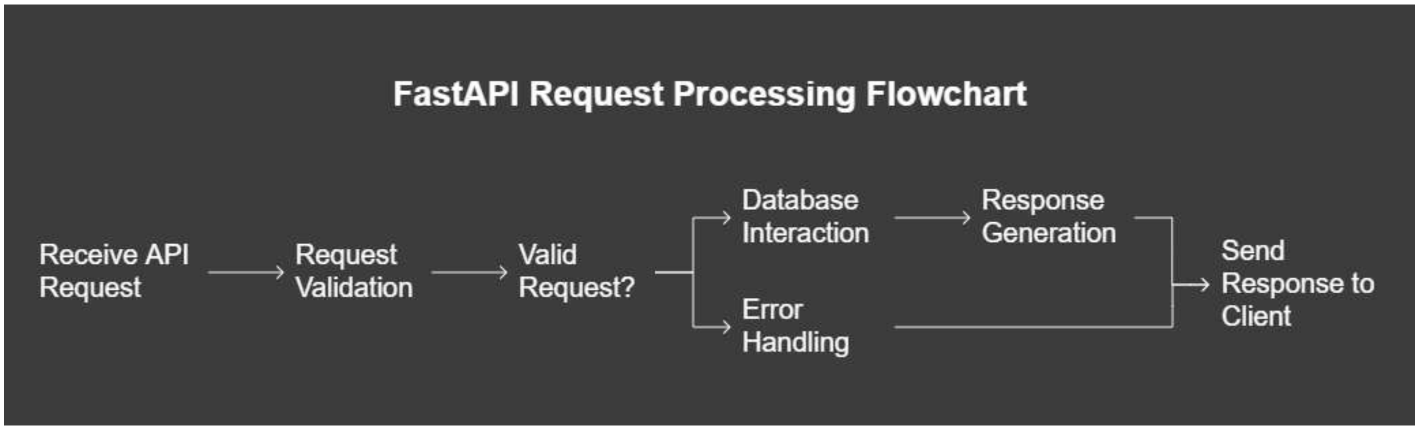
4. Technical Considerations

```
# Requirements
    fastapi>=0.100.0
    uvicorn[standard]>=0.20.0
    pydantic>=2.0.0
    python-jose[cryptography]>=3.3.0
    passlib[bcrypt]>=1.7.4
    redis>=4.5.0 # For caching implementation in Phase 5
    asyncpg>=0.29.0 # For asynchronous PostgreSQL interactions
    # psycopg2-binary # May be needed by other tools or for sync fallback/testing
```

VII. Deliverables and Timeline

Project Deliverables

Code Migration



- Migrate all Django views and API endpoints to FastAPI with async support (`async` / `await` , `asyncpg`).
- Implement structured request handling and response validation using Pydantic models.
- Manage database schema changes and versioning with robust Bash scripts executing raw SQL (including state tracking and error handling).

Security:

- Implement secure JWT-based authentication to maintain parity with the existing Django system.
- Strengthen input validation and enforce security best practices (e.g., security headers).
- Implement basic rate limiting (if time permits within Phase 4).

Performance Optimizations:

- Introduce caching mechanisms using **Redis** to optimize frequently accessed database queries/API responses.
- Conduct load testing to identify potential bottlenecks and benchmark against targets.
- Leverage `asyncpg` connection pooling and asynchronous query execution.

Deployment and Containerization:

- Containerize the FastAPI application using Docker and Docker Compose for efficient development and deployment workflows.

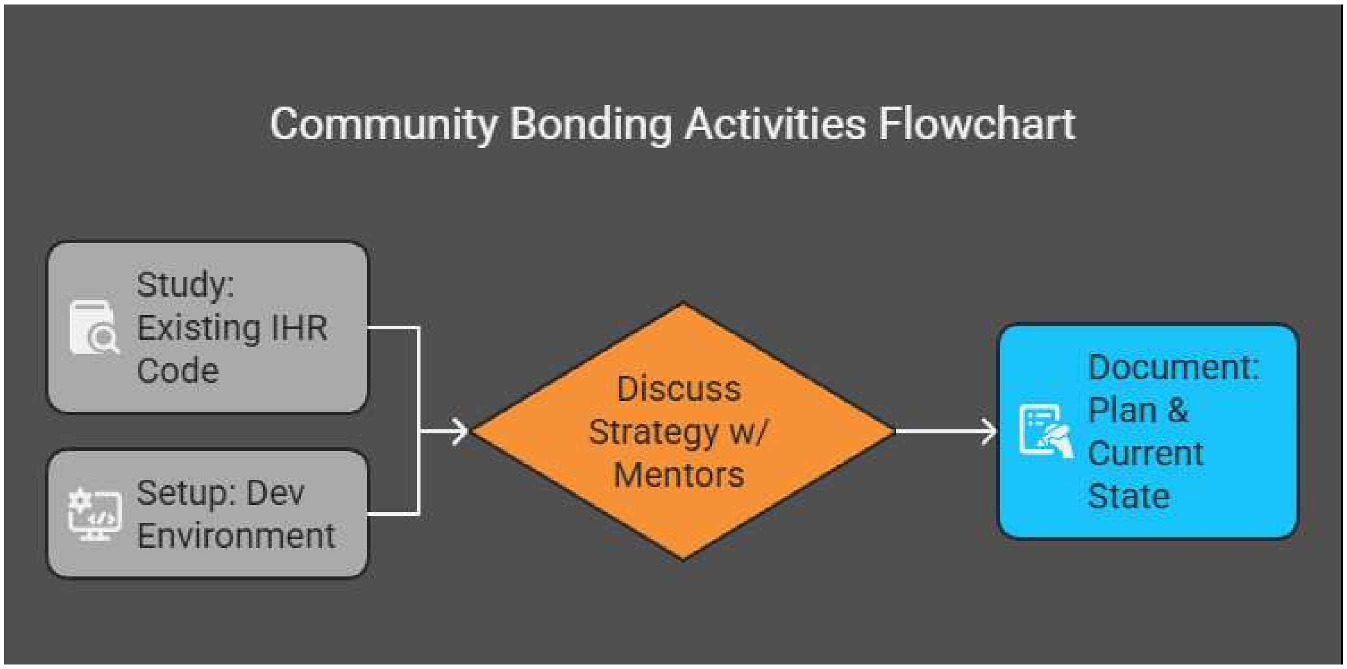
Documentation and Testing:

- Maintain comprehensive project documentation: README, migration guides, API documentation (auto-generated via OpenAPI), Bash script usage (`DATABASE.md`), and system architecture details.
- Implement extensive test coverage using `pytest` and `pytest-asyncio` for robustness.
- Conduct security checks and performance audits to validate the migration's success against objectives.

Timeline and Implementation Plan

Community Bonding (May 8 - June 1, 2025)

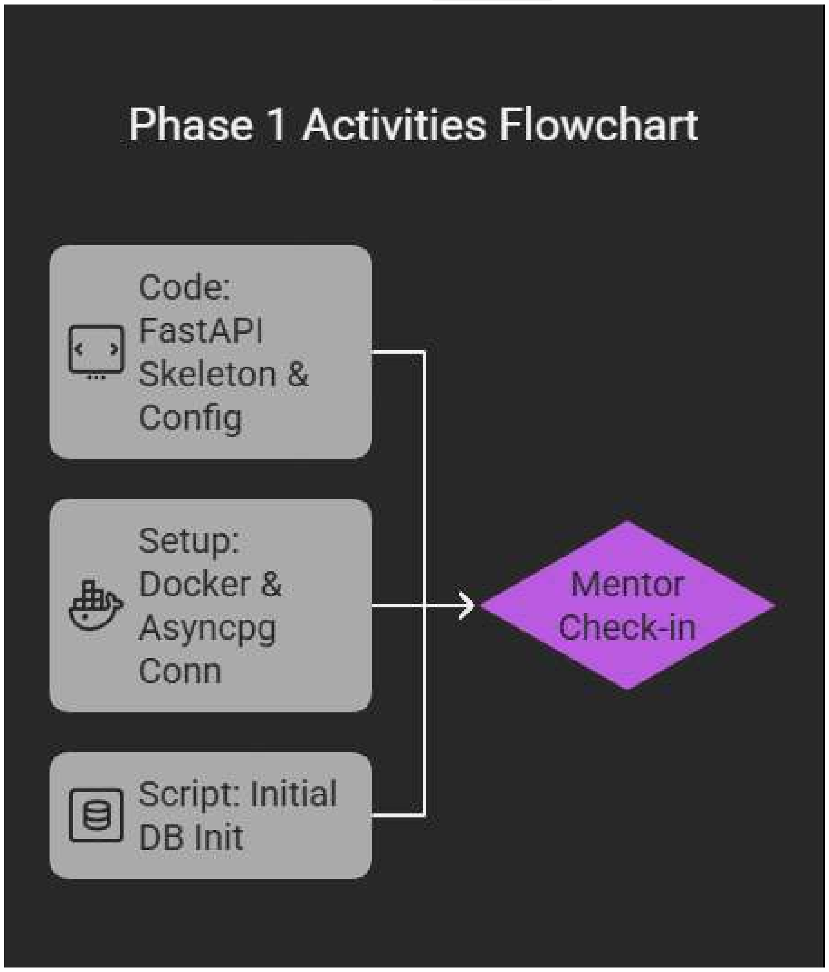
- **Goals:** Understand the existing Django 2.2 codebase in detail, engage with mentors and discuss migration strategies, document current API behavior/dependencies/challenges, define key milestones and deliverables.


Community Bonding Activities Flowchart
Study: Existing IHR Code
Setup: Dev Environment
Discuss Strategy w/ Mentors
Document: Plan & Current State

- **Tasks Overview:**
- Study and analyze existing models (`models.py`) including `Hegemony` , `ASN` , `Country` , and database caching strategies.
- Review API endpoints in `views.py` such as network monitoring and hegemony analysis.
- Document authentication methods, database relationships, and performance bottlenecks.
- Prepare a high-level migration roadmap with mentors.
- Set up a development environment (Python, Docker, PostgreSQL, FastAPI basics).
- Contribute to any existing simple issues in the IHR codebase (if applicable) to familiarize with workflow.
- Have introductory calls with the mentors to align expectations.

Phase 1: Core Setup (June 2 - June 15, 2025)

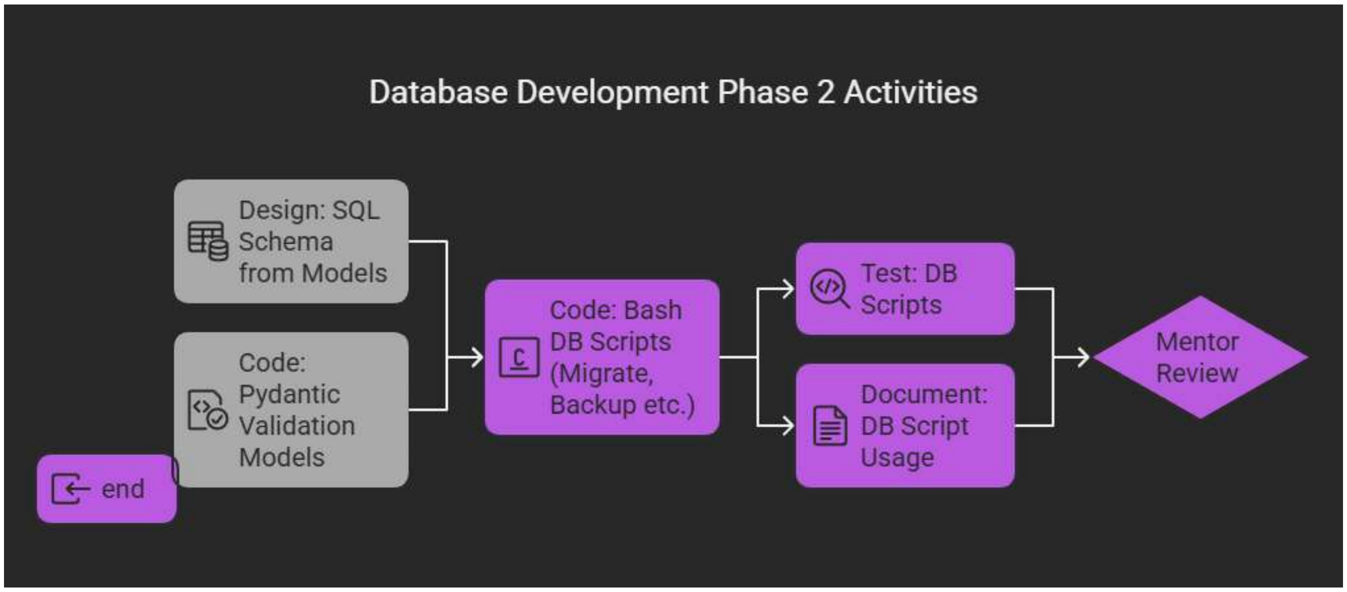
- **Goals:** Set up the FastAPI project structure, establish core middleware (logging, error handling), request handling, database connections (`asyncpg`), and implement initial basic DB scripts.


Phase 1 Activities Flowchart
Code: FastAPI Skeleton & Config
Setup: Docker & Asyncpg Conn
Script: Initial DB Init
Mentor Check-in

- **Tasks:**
- Create FastAPI project skeleton following best practices (e.g., `/app` , `/core` , `/api` , `/schemas` , `/db_scripts` , `/sql`).
- Implement environment configuration management (e.g., using Pydantic settings).
- Set up Docker containers (FastAPI app, PostgreSQL) via `docker-compose.yml` .
- Implement `asyncpg` connection pool setup and dependency injection for routes.
- Develop initial `db_scripts/init_db.sh` to create the database and roles.
- Implement basic middleware (e.g., logging requests).
- Create a basic health check endpoint.
- **Key Challenges:** Ensuring seamless configuration transition, handling initial async requests correctly, setting up robust DB connection management.

Phase 2: Database Scripting & Schema Conversion (June 16 - July 5, 2025)

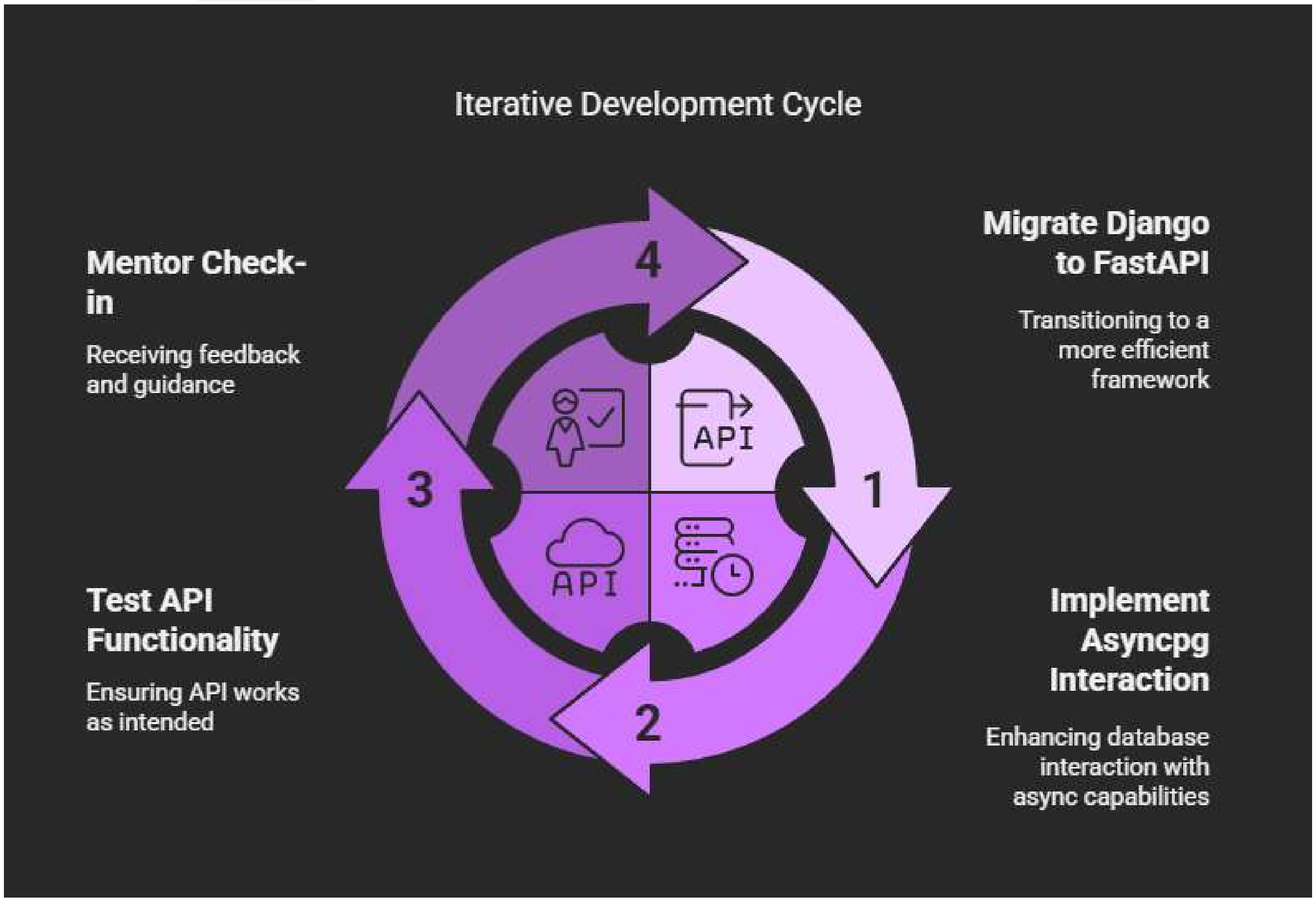
- **Goals:** Convert Django models to SQL schema files, develop and test robust Bash scripts for DB management (init, migrate, backup, seed) with error handling and state tracking, implement dry-run for migrations.


Database Development Phase 2 Activities
Design: SQL Schema from Models
Code: Pydantic Validation Models
end
Code: Bash DB Scripts (Migrate, Backup etc.)
Test: DB Scripts
Document: DB Script Usage
Mentor Review

- **Tasks:**
- Translate Django models into `sql/schema/*.sql` files (tables, indexes, constraints).
- Develop `db_scripts/migrate_db.sh`:
- Implement logic to detect and apply new migrations from `sql/migrations/`.
- Include robust error handling (`set -e`, explicit checks).
- Implement state tracking (e.g., `schema_migrations` table).
- Implement `--dry-run` functionality as shown previously.
- Consider basic rollback mechanisms or instructions for manual rollback.
- Develop `db_scripts/backup_db.sh` (using `pg_dump`) and potentially `restore_db.sh` (`psql`).
- Develop `db_scripts/seed_db.sh` for populating initial/test data.
- Define Pydantic models in `/schemas` mirroring the SQL structure for validation.
- Thoroughly test all Bash scripts.
- **Challenges:** Ensuring Bash script robustness, correct state management for migrations, handling potential large datasets during seeding/migrations efficiently, translating complex Django model relationships/logic to SQL/Bash.

Phase 3: API Endpoint Migration (July 6 - July 25, 2025)

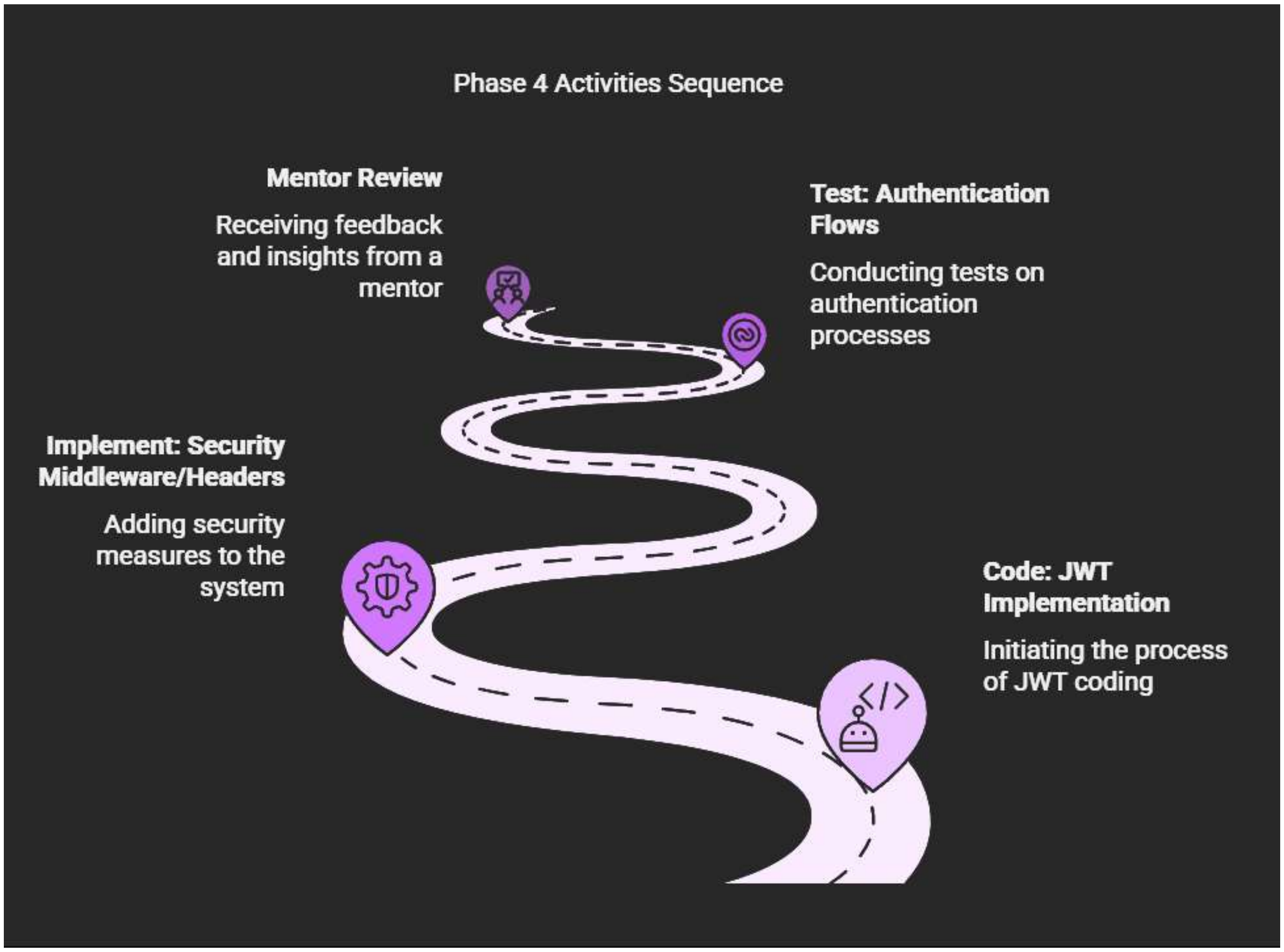
- **Goals:** Convert key Django API views into FastAPI endpoints, implement asynchronous request handling using `asyncpg`, ensure feature parity and API compatibility.


Iterative Development Cycle
Migrate Django to FastAPI
Transitioning to a more efficient framework
1
Implement Asyncpg Interaction
Enhancing database interaction with async capabilities
2
Test API Functionality
Ensuring API works as intended
3
Mentor Check-in
Receiving feedback and guidance
4
API
API

- **Tasks:**
- Migrate Django API views (e.g., from `views.py`) to FastAPI routers/endpoints (`/api/v1/`).
- Replace Django ORM calls with asynchronous `asyncpg` queries.
- Implement request validation using Pydantic models defined in Phase 2.
- Implement response serialization using `response_model` with Pydantic.
- Ensure error responses are handled gracefully and consistently.
- Validate migrated API behavior against the existing Django system (functional parity).
- **Challenges:** Maintaining 100% API consistency, handling potential subtle differences in behavior between sync Django ORM and async raw SQL (`asyncpg`), debugging async code effectively.

Phase 4: Authentication & Security (July 26 - August 5, 2025)

- **Goals:** Implement JWT authentication matching existing functionality, migrate permissions logic, enforce security best practices.

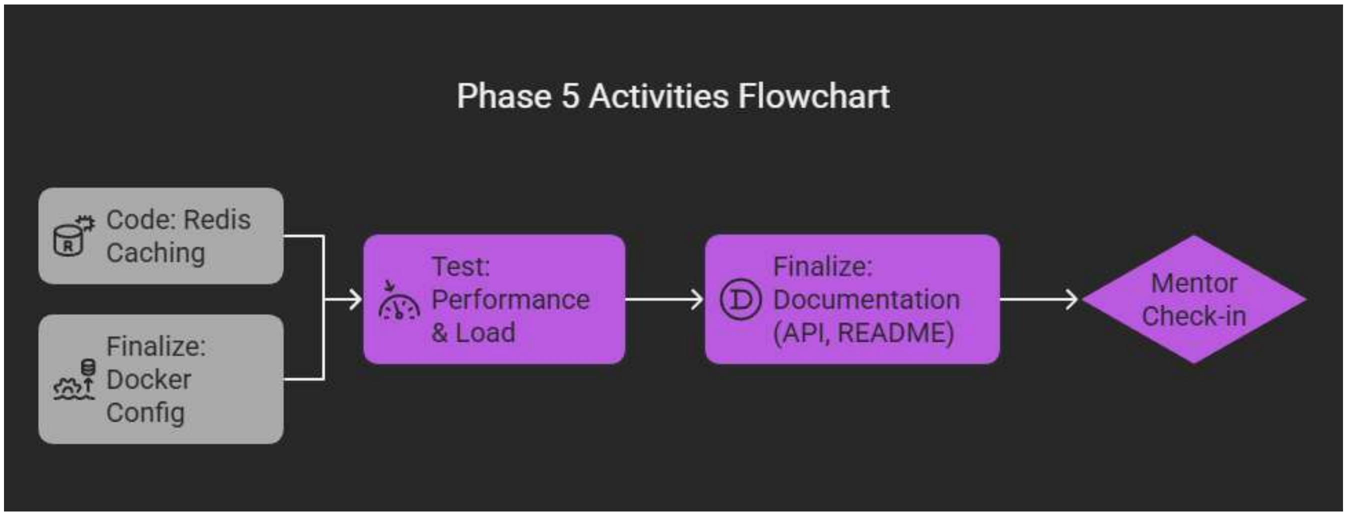


Phase 4 Activities Sequence
Mentor Review
Receiving feedback and insights from a mentor
Test: Authentication Flows
Conducting tests on authentication processes
Implement: Security Middleware/Headers
Adding security measures to the system
Code: JWT Implementation
Initiating the process of JWT coding

- **Tasks:**
- Implement JWT token generation (on login) and validation (middleware/dependency) using `python-jose`.
- Implement password hashing/verification using `passlib`.
- Create authentication routes (`/login`, potentially `/register` if needed).
- Implement dependency functions to get current user from JWT and enforce permissions.
- Migrate existing user roles/permissions logic (this might involve DB checks via `asyncpg`).
- Implement security enhancements: input validation (via Pydantic), security headers (via middleware), basic rate limiting (e.g., using `slowapi` if time permits).
- **Challenges:** Ensuring seamless security transition without breaking existing client workflows, mapping Django\'s auth system concepts to FastAPI/JWT, securely handling secrets.

Phase 5: Optimization & Deployment Prep (August 6 - August 15, 2025)

- **Goals:** Implement Redis caching for key endpoints, conduct performance benchmarking, finalize Docker setup, finalize API documentation.



- **Tasks:**
- Integrate Redis connection using a library like `redis-py` (async support).
- Identify high-traffic/slow endpoints suitable for caching.
- Implement caching logic (e.g., decorator or middleware) for selected endpoints, including cache invalidation strategies where necessary.
- Conduct performance benchmarking and load testing (e.g., using `locust`) against targets (<200ms).
- Analyze results and perform basic query optimization if needed.
- Finalize `Dockerfile` and `docker-compose.yml` for production-like setup (consider multi-stage builds, non-root user).
- Ensure auto-generated OpenAPI documentation is accurate and user-friendly.
- Write/finalize user documentation (README, `DATABASE.md` on script usage).
- **Challenges:** Implementing effective caching strategies with proper invalidation, setting up realistic load tests, ensuring Docker setup is efficient and secure.

Final Evaluation (August 16 - August 25, 2025)

- **Goals:** Validate migration success against objectives, perform final testing and documentation updates, address mentor feedback, submit final deliverables.
- **Tasks:**
- Conduct thorough end-to-end integration tests.
- Perform final load testing to confirm performance targets.
- Complete all documentation (code comments, README, API docs, migration guide).
- Ensure FastAPI's OpenAPI documentation is accurate and reflects all endpoints/schemas.
- Address any remaining bugs or issues identified during testing/mentor reviews.
- Prepare final code submission according to GSoC guidelines.
- Gather final mentor feedback and incorporate necessary refinements.
- **Challenges:** Ensuring all edge cases are tested, completing final documentation thoroughly, managing time for final refinements.

Project Timeline Overview Table

Phase	Tasks	Timeline	Estimated Hours
Phase 1: Core Setup	\- Develop FastAPI skeleton- Configure Docker/Docker Compose- Implement basic healthcheck/logging/error handling- Setup `asyncpg` connection pool- Initial DB init script	June 2 – June 15, 2025	25h
Phase 2: Database Scripting & Schema Conversion	\- Convert Django models to an SQL schema- Develop, test, and version-control robust Bash scripts for database management (init, migrate w/ state & dry-run, backup, seed)- Implement data validation using Pydantic models	June 16 – July 5, 2025	40h
Phase 3: API Migration	\- Refactor all API endpoints from Django to FastAPI- Implement asynchronous request handling (`asyncpg`)- Implement comprehensive request/response validation (Pydantic)- Ensure API compatibility	July 6 – July 25, 2025	45h
Phase 4: Authentication & Security	\- Implement JWT-based authentication- Integrate permissions logic- Implement security best practices (headers, input validation)- Basic rate limiting (if feasible)	July 26 – August 5, 2025	30h
Phase 5: Optimization & Deployment Prep	\- Implement Redis caching for key API endpoints- Conduct basic load testing and performance benchmarking- Finalize Docker Compose setup- Finalize and update documentation (README, API docs, `DATABASE.md`)	August 6 – August 15, 2025	35h
Final Evaluation & Wrap-up	\- Final Testing, Documentation Polish, Address Feedback, Code Submission	August 16 – August 25, 2025	*(Buffer/Testing)*
Total Estimated Hours			**175h**

VIII. Unit Testing Strategy

A robust testing framework is crucial for the success of this migration. The testing strategy will include:

- **Framework:** Utilize `pytest` along with `pytest-asyncio` to support asynchronous testing.
- **API Testing:** Employ tools like `httpx` within `pytest` to make requests to the FastAPI test client, validating routes, request/response formats, status codes, and adherence to the OpenAPI schema.
- **Database Testing:** Ensure data integrity post-migration by comparing outcomes with baseline results. Test database interaction logic, potentially using test databases or transaction rollbacks. Test the functionality of the Bash DB management scripts (e.g., does init create the schema? does migrate apply changes correctly?).
- **Performance Testing:** Conduct load tests using tools like `locust` or `k6` to confirm that the new endpoints meet performance targets (e.g., 95% response times under **200ms** under expected load).
- **Security Testing:** Validate the effectiveness of JWT authentication (token validation, expiry, protected routes) and other security measures like input validation.

Each test case will be thoroughly documented. Fixtures (`pytest` fixtures) will be used extensively to set up necessary environments (like database connections, test data) and ensure test reproducibility.

IX. My Progress and Research

To ensure I could create a detailed and viable proposal, I have dedicated significant time over the past few weeks to researching the project requirements and the existing IHR codebase.

My preparation involved:

1. **Exploring the Existing Codebase:** I have forked the `ihr-django` repository to my personal GitHub account: My fork. Within this fork, I have been actively exploring the current Django 2.2 application, focusing on understanding the structure of the models (`models.py`), the logic within the views (`views.py`), the existing database interaction patterns, authentication mechanisms, and identifying key dependencies. This hands-on exploration has helped me grasp the complexities involved in the migration.
2. **Researching Target Technologies:** I have refreshed my knowledge and researched best practices for FastAPI, particularly concerning asynchronous operations with `asyncpg` for PostgreSQL interaction, structuring larger FastAPI applications, and implementing JWT authentication securely using `python-jose` and `passlib`.
3. **Investigating the Database Scripting Approach:** Given the requirement to use Bash scripts for database management, I researched techniques for writing robust and maintainable shell scripts for tasks like schema initialization, migration application (including state tracking and error handling, similar

to concepts in tools like Alembic or Flyway, but implemented in Bash), and backups using tools like `psql` and `pg_dump`.
4. **Dockerization:** I reviewed best practices for containerizing FastAPI applications using Docker and Docker Compose, including multi-stage builds and handling environment variables.

This preliminary research and hands-on exploration of the codebase have directly informed the detailed migration strategy, timeline, and deliverables outlined in this proposal, giving me confidence in the feasibility of the plan.

I am committed to delivering a high-quality and well-documented proposal that aligns with the project requirements and my understanding of the existing codebase.

X. Expectations from Mentor

I am always looking to learn and improve upon my mistakes. I would like my mentors to provide constructive feedback, so we can work together to make my project better. This gives me the opportunity to collaborate with them and learn from their experiences as professionals in this field. I am particularly interested in discussing the practicalities and potential trade-offs of the Bash-based database management approach. I would also love to understand their motivations for working with the IHR and learn about their career journeys.

>**XI. Other Commitments**

During the GSoC period, my primary focus will be on this project. I do not have any full-time commitments that would interfere with my ability to complete the expected deliverables. However, I may occasionally work on personal development projects or engage in my college studies, especially during exam periods. I will ensure transparent communication regarding my availability.

XII. Future Plans After GSoC

After the GSoC period is over, I plan to continue contributing to the IHR project by maintaining and improving the backend infrastructure, potentially working on items identified in the \"Future Work\" section like implementing CI/CD pipelines or enhanced monitoring. I am also interested in exploring opportunities in backend system architecture, distributed computing, and cloud-native development. Additionally, I would love to mentor future GSoC participants and help them navigate the program, sharing my firsthand experience of the challenges and learning opportunities it offers.