Term: Fall 2025 **Subject:** Computer Science & Engineering (CSE) **Number:** 512

Course Title: **D**istributed **D**atabase **S**ystems (CSE 512)

GROUP PROJECT MILESTONE 2 TEMPLATE

Project Title:

Disaster Resilient Social Platform - MeshNetwork

Team Name and Members:
Abhave Abhilash, 1228161950
Yashwardhan Ramchaware, 1237156078
Aditya Gupta, 1228242108

1. Introduction and Recap

During natural disasters like hurricanes, tornadoes, and flash floods, communication infrastructure is frequently disrupted, isolating users in affected areas. Existing disaster communication platforms such as social media platforms (Facebook, X, etc.) rely on centralized infrastructure and business logic that fails in cases of network isolation.

Project Objectives:

- Design and implement a geo-distributed, fault-tolerant distributed database system specifically for disaster scenarios where availability is highest priority over consistency.
- Ensure geographic data partitioning so that users can interact with local updates to the platform even during network failures.
- Ensure fault tolerance and “island mode” operation to allow regional autonomy during extended network partitions

2. System Architecture and Design

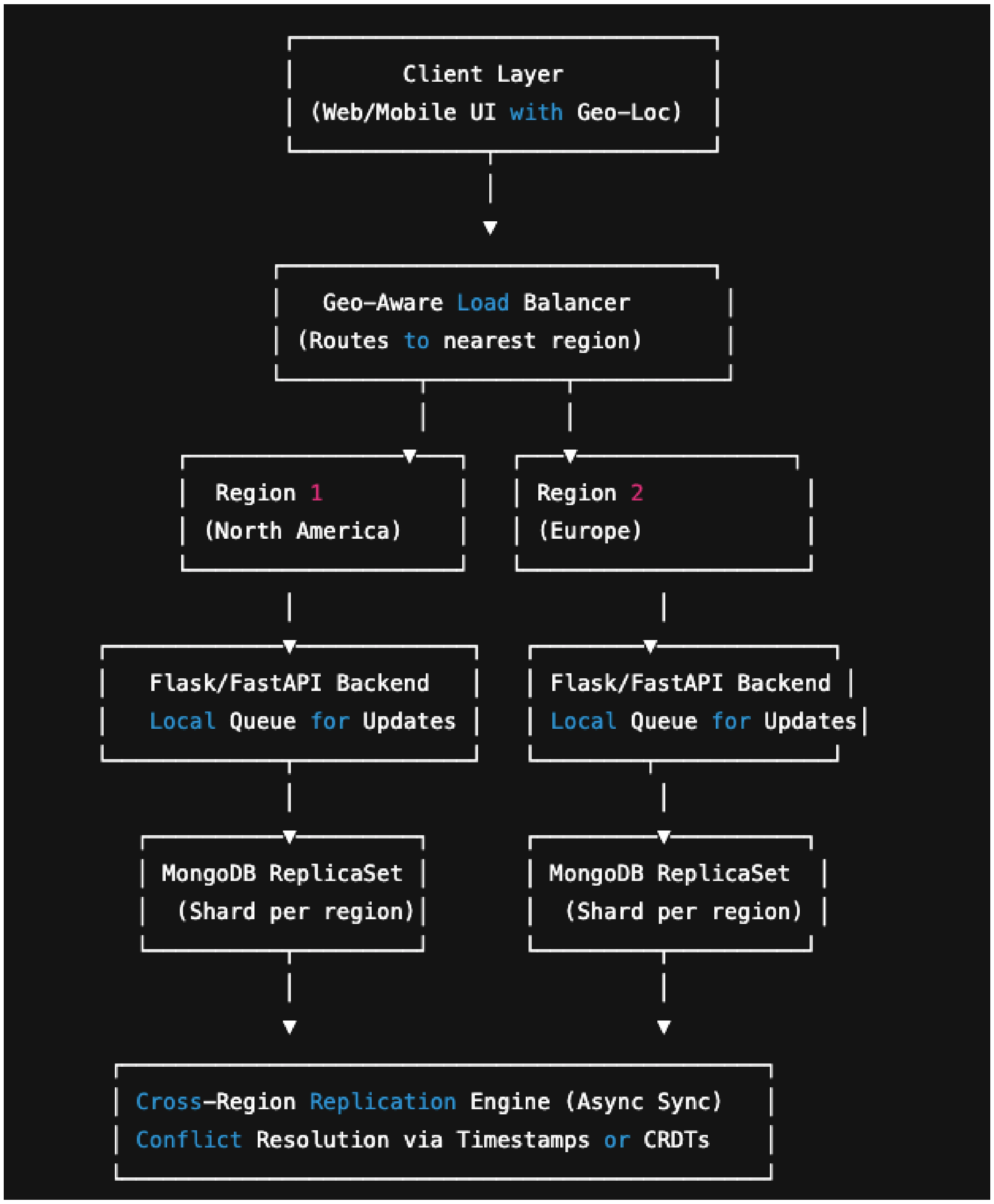
2.1 High-Level Architecture

The system uses a geographically distributed fault tolerant architecture, which makes sure that operations continue even if one or more regions lose network connection during the time of a disaster. The geographically aware load balancer routes all client requests to the nearest healthy regional cluster based on network latency and node health checks. Client requests are processed, basic authentication is performed, and updates are written to a local queue to maintain performance during temporary failures. Every regional cluster (e.g., North America and Europe) operates independently and consists of three main components:

- **Flask/FastAPI Backend**
    - Client requests are processed, basic verification is performed, and updates are written to a local queue to maintain operation during temporary failures.
- **Local MongoDB Replica Set**
    - Stores all regional data. Within each region, synchronous replication between the primary and secondary replicas guarantees strong local consistency and data durability.
- **Cross-Region Replication Engine**
- Periodically exchanges updates between regions using **asynchronous synchronization**. When conflicts occur, they are resolved deterministically with a **Last-Write-Wins (timestamp-based)** rule or, conceptually, via **CRDT-style merges**. This ensures **eventual global consistency** once all regions reconnect.

Under normal circumstances requests are processed locally, with global synchronization in the background. During a partition each zone continues to operate in "island mode" performing write operations in a queue until the network recovers. This architecture implements the trade-off principle of the CAP theorem, prioritizing availability (A), distribution tolerance (P) over direct global consistency (C), which is essential for a disaster resilient communication system.

Diagram :



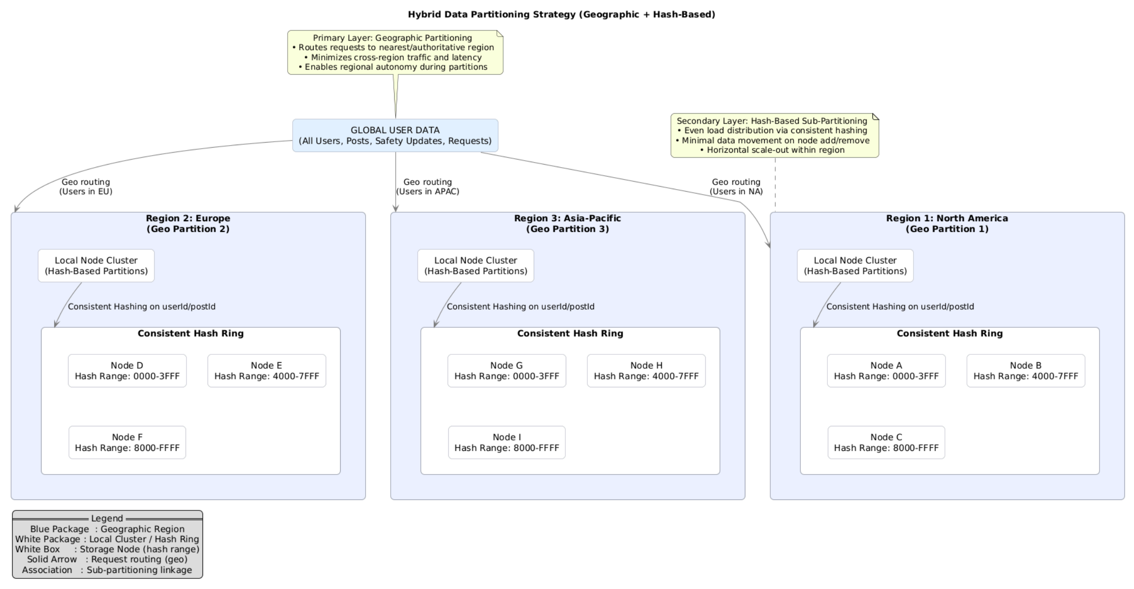
```
                ┌─────────────────────────────┐
                │         Client Layer        │
                │ (Web/Mobile UI with Geo-Loc)│
                └──────────────┬──────────────┘
                               │
                               ▼
               ┌─────────────────────────────┐
               │   Geo-Aware Load Balancer    │
               │ (Routes to nearest region)   │
               └──────────┬──────────┬────────┘
                          │          │
         ┌────────────────▼──┐    ┌──▼────────────────┐
         │  Region 1         │    │ Region 2           │
         │ (North America)   │    │ (Europe)           │
         └───────────────────┘    └────────────────────┘
                  │                          │
     ┌────────────▼───────────┐  ┌───────────▼───────────┐
     │   Flask/FastAPI Backend │  │ Flask/FastAPI Backend │
     │   Local Queue for Updates│ │ Local Queue for Updates│
     └────────────┬───────────┘  └───────────┬───────────┘
                  │                          │
        ┌─────────▼─────────┐    ┌───────────▼──────────┐
        │ MongoDB ReplicaSet │    │ MongoDB ReplicaSet    │
        │  (Shard per region)│    │   (Shard per region)  │
        └─────────┬─────────┘    └───────────┬──────────┘
                  │                          │
                  ▼                          ▼
      ┌──────────────────────────────────────────────────┐
      │ Cross-Region Replication Engine (Async Sync)     │
      │ Conflict Resolution via Timestamps or CRDTs      │
      └──────────────────────────────────────────────────┘
```

2.2 Feature 01 (For Example, Data Partitioning Strategy)

In this project we are using a **hybrid partitioning strategy** so that we can balance scalability and disaster tolerance, for that the two partitioning methods are:

1. **Geographic Partitioning (Primary Layer):**
    - Every region in the project (e.g. North America, Europe, Asia-Pacific) manages data for users within its own region.
    - Requests are routed to the local region to ensure low latency and local fault isolation so that regions can stay up even when the network connection to the other regions is down.
2. **Hash-Based Sub-Partitioning (Secondary Layer):**
    - Within a region, data is distributed across available nodes using **consistent hashing** on userId.
    - Ensures load is balanced and allows nodes to be added or removed with minimal data movement and cost.

Illustration of Partitioning Scheme -


Hybrid Data Partitioning Strategy (Geographic + Hash-Based)
Primary Layer: Geographic Partitioning
• Routes requests to nearest/authoritative region
• Minimizes cross-region traffic and latency
• Enables regional autonomy during partitions
GLOBAL USER DATA (All Users, Posts, Safety Updates, Requests)
Secondary Layer: Hash-Based Sub-Partitioning
• Even load distribution via consistent hashing
• Minimal data movement on node add/remove
• Horizontal scale-out within region
Geo routing (Users in EU)
Geo routing (Users in APAC)
Geo routing (Users in NA)
Region 2: Europe (Geo Partition 2)
Region 3: Asia-Pacific (Geo Partition 3)
Region 1: North America (Geo Partition 1)
Local Node Cluster (Hash-Based Partitions)
Consistent Hashing on userId/postId
Consistent Hash Ring
Node D Hash Range: 0000-3FFF
Node E Hash Range: 4000-7FFF
Node F Hash Range: 8000-FFFF
Node G Hash Range: 0000-3FFF
Node H Hash Range: 4000-7FFF
Node I Hash Range: 8000-FFFF
Node A Hash Range: 0000-3FFF
Node B Hash Range: 4000-7FFF
Node C Hash Range: 8000-FFFF
Legend
Blue Package : Geographic Region
White Package : Local Cluster / Hash Ring
White Box : Storage Node (hash range)
Solid Arrow : Request routing (geo)
Association : Sub-partitioning linkage

Why this type of Partitioning –

We are implementing this partitioning strategy as it improves the following -

· **Performance:** Most operations are local to a region (e.g., nearby help requests), which minimizes the inter-region traffic, thus improving the overall performance.

· **Autonomy:** Each region can function independently during network isolation. So, even if a node goes down, the regional node will function and that also autonomously.

· **Scalability:** Adding new nodes to the system doesn't require a full data redistribution, and thus supports easy horizontal scaling.

· **Resilience:** The overall system stays connected even if some nodes fail and Local partitions remain functional even if other regions fail.

2.3 Feature 02 (Replication and Consistency Model)

Replication Type:

1. **Intra-Region Replication (Synchronous):**
    - For the intra-region based replication, Each region has its own MongoDB database and for robustness, we are implementing replicas of the original data, for our project we are using the **replica sets** with one primary node, one or two secondary nodes, where all 3 are available for reads.
    - For Write operations, we are ensuring that writes must be acknowledged only after replication and there are atleast two writes of the same data within the distributed database to ensure durability and robustness.
2. **Inter-Region Replication (Asynchronous):**
    - For the Inter-Region based replication, Each region acts as a **master** capable of accepting writes from other regions and accordingly updating the database.
    - Changes happening in the database are periodically pushed to other regions(For example. Region NA changes some data locally and is offline, once it comes back online the changes are updated to other Regions aswell like AP and EU) through a **replication script or API** which will integrate the updates with other regions and sync them.
    - Updates carry timestamps to resolve conflicts automatically (Last-Write-Wins).

Aspect	**Chosen Approach**	**Reason / Theoretical Basis**
Replication Model	Hybrid (sync local + async global)	Ensures fast local durability while allowing regions to operate independently during network failures.
Consistency Guarantee	**Strong (intra-region)** for local writes, **Eventual (inter-region)** for global state	Locally, updates are immediately visible after commit; globally, replicas converge once asynchronous synchronization completes—demonstrating CAP's A + P trade-off.
Conflict Resolution	Timestamp-based LWW	Provides simple, deterministic convergence while maintaining eventual consistency semantics.
Failover Mechanism	Automatic replica promotion (MongoDB Replica Sets)	Maintains service continuity and consistency within a region even when primary nodes fail.
Resilience Focus	Regional autonomy + self-healing sync	Allows regions to remain operational during network partitions and automatically reconcile upon recovery.

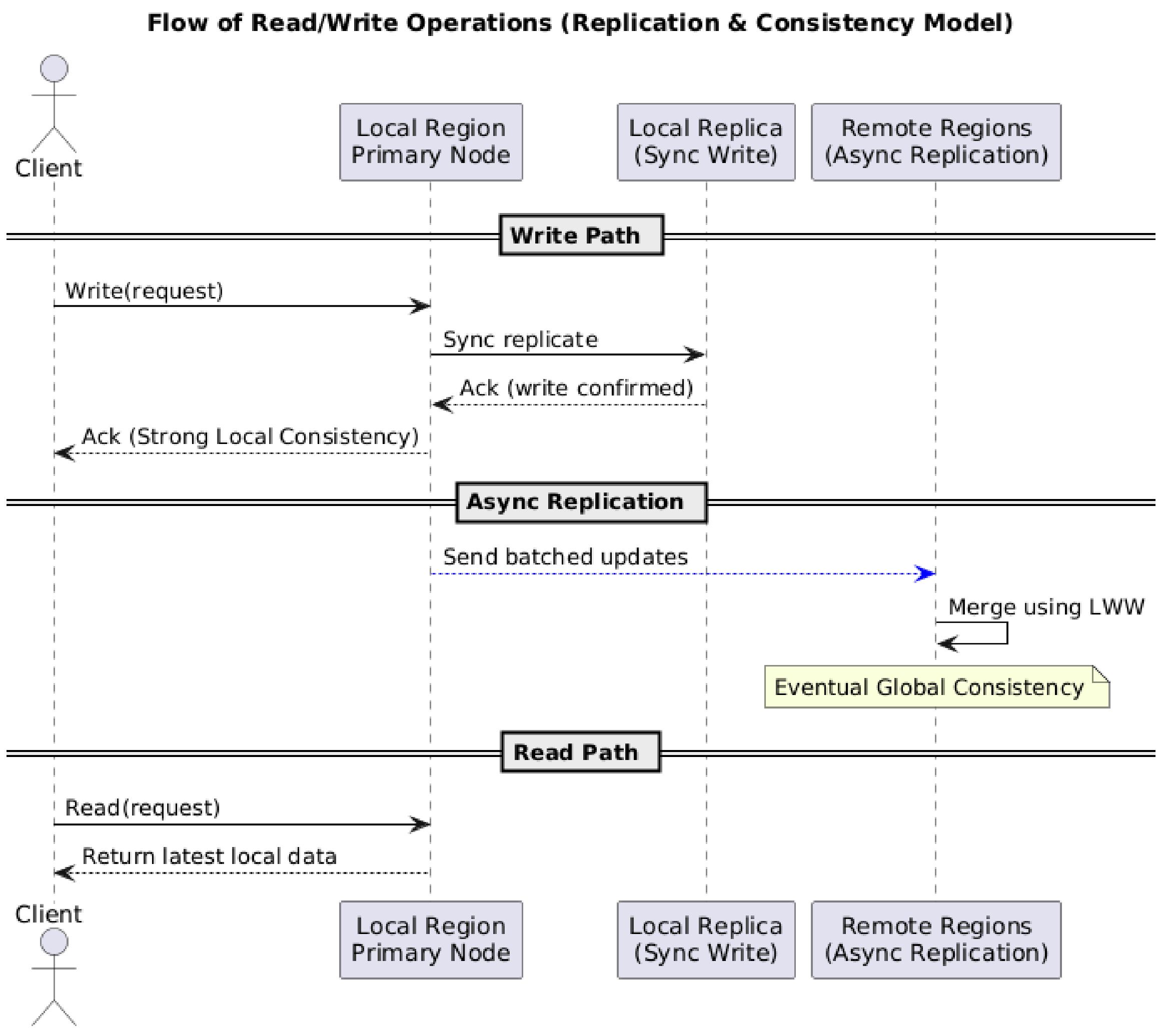

Flow of Read/Write Operations (Replication & Consistency Model)
Client
Local Region Primary Node
Local Replica (Sync Write)
Remote Regions (Async Replication)
Write Path
Write(request)
Sync replicate
Ack (write confirmed)
Ack (Strong Local Consistency)
Async Replication
Send batched updates
Merge using LWW
Eventual Global Consistency
Read Path
Read(request)
Return latest local data

Figure: Flow of read/write operations

Trade-Off Analysis :

- **Advantages:**
  - Data is always available for writes (even during partitions) as every regional cluster can accept local writes independently thus even if a region is isolated it continues to work locally.
  - No single point of failure as every region will maintain its own replica set (ie. primary and secondary nodes) thus the system avoids dependence on any other Regional coordinator.
  - Latency is low for local operations as user requests are processed by the nearest geographic region, minimizing round-trip time and network latency.

- **Disadvantages:**
    - Because of the replication between regions is asynchronous, different regions may temporarily store old or conflicting versions of the same data until updates are synchronized after network connection is restored.
    - The "last-write-wins" mechanism uses timestamps to resolve conflicts, but clock drift or simultaneous updates can overwrite or lose new information and in some cases can even reduce accuracy.

2.4 Feature 03 (For Example, Fault Tolerance and Recovery)

Fault Tolerance:

1. Nodes run continuous health checks to monitors, database connectivity, network connectivity to other regions, server status/resource utilization, and cross-region replication lag
2. Nodes within regional clusters exchange heartbeat messages, and nodes that fail to respond to three consecutive messages are marked as potentially failed/suspect.
3. If a region is isolated for over 60 seconds, the region operates under island mode operation to allow continued function.

Recovery Mechanisms:

1. **Automatic Failover**
    a. MongoDB replica sets implement automatic primary election when primary node fails
    b. Clients experience minimal service interruption
    c. Zero data loss for acknowledged writes (with write concern: majority)
2. **Logging**
    a. Write operations are logged with timestamps and region identifiers
    b. In network isolation, each region maintains a local operation log that queues updates which is used for syncing purposes.
3. **Replica Re-synchronization**
    a. Copy the current dataset from healthy replicas
    b. Apply operations that occurred during downtime
    c. Continuously add new operations until fully synchronized
4. **Cross-Region Reconciliation**
    a. Identify all queued operations
    b. Push and pull updates to and from other regions
    c. Finally, mark as “fully synchronized"

Failure Scenarios and Responses

Failure Type	**Detection Time**	**Recovery Action**	**User Impact**
Single node within	~15 seconds	Automatic primary	Minimal

region		node election	
Multiple nodes in region	~15 seconds	Continue with remaining majority	None
Majority nodes in region	Immediate	Region is read-only until restoration	No more local writes
Region isolation	~1 minute	“Island Mode” operation, queue updates	Cannot sync updates cross-region
Region reconnection	Immediate	Reconciliation via oplog replay	Sync then normal operation

Testing Failure

1. **Single Node Failure Test:**
   a. Kill secondary node in a regional cluster
   b. Verify automatic detectioN/failover
   c. Measure recovery time and data loss (should be none)
2. **Primary Node Failure Test:**
   a. Kill primary node in a region
   b. Verify automatic primary election and measure time
   c. Confirm acknowledged writes are preserved
3. **Network Partition Test:**
   a. Isolate a regional cluster using Docker network controls
   b. Verify island mode operation
   c. Restore network connectivity and verify automatic reconciliation
4. **Cascading Failure Test:**
   a. Kill multiple nodes
   b. Verify operation continues while majority remains
   c. Test recovery when nodes are restored

Visualization/Monitoring:
We will use our React frontend to visualize node status, failure events, and recovery processes in real-time.

2.5 Feature 04 (For Example, Query Processing and Optimization)

Query Planning and Execution Over Multiple Nodes

- When performing query planning and execution over multiple nodes, we will be using a multi-step process that figures out localization of data (eg. where is the data stored) data querying (retrieving the data from that specific location), and then query execution to actually run the data retrieval process on those special nodes, and then data combination to combine and present the relevant data, from the separate nodes, to the user. Below is our detailed process, that explains the details of how this process works.
    - Step 1:
        - User makes request
    - Step 2:
        - Query Planning - Our Query Router, which is in our flask backend, decides which nodes to use within our mesh network and here our router has two options, either we are going to do Local query, which is just a single region or a single node in our MongoDB, or a Global query, where we would query over multiple nodes in the MongoDB.
    - Step 3:
        - Query Execution - our flask backend will then send our query to the MongoDB replica set
        - MongoDB will then route to the primary node for writes (eg. post about this disaster), and the secondary nodes for reads (what are the disasters in x region)
        - Our mongo db will then execute the query .find() according to the type of post we need then return the results back to the user.
    - Step 4:
        - And then we have our Result Aggregation, step, which will aggregate the data from multiple nodes, if necessary (eg. a global query) and present that to the user.

Techniques

- So we are going to be using the following techniques: query routing, scatter-gather pattern, local-first query execution, index-based query optimization, map-reduce-style aggregation, and distributed joins
    - Query routing will help us properly direct our query to the correct region or node.
    - Scatter-gather will allow us to send concurrent queries to multiple nodes for data fetching
    - Local-first query execution will allows to us to reduce the network latency and the amount of data being transferred, if being transferred from multiple local nodes
    - Index-based query optimization will speed up database queries
    - Map-reduce style aggregation will break aggregation into maps, which will help process the data locally and then reduce, or combine the final results

        - Distributed joins to combine data from multiple regions or collections

Example

- Sarah is in Phoenix, Arizona and wants to see emergency shelters near her that have opened in the last 24 hours
    - Phase 1: System obtains request
        - The Flask Backend in North America receives Sarah's request and does initial processing. The system will verify her authentication token and will parse her query to extract the key components: emergency shelters, geographic proximity, and time constraint.
    - Phase 2: Query Planning & Strategy Decision
        - The Query Router will then determine the optimal execution strategy by analyzing network health, relevance and freshness of data and an execution plan.
    - Phase 3: Local Execution - Querying North America
        - The system sends the query to the MongoDB database cluster in North America and the system will decide to route the query to one of the secondary nodes to balance the load. This is where our optimization process starts:
            - The database examines its available indexes to find the most efficient way to locate the relevant data and since the database has a compound index built on region, post type, and timestamp in that specific order, which perfectly matches the query pattern.
            - The query optimizer decides to use this compound index rather than performing a full collection scan (as mentioned in the techniques above) and the database can jump directly to the section containing North America shelter posts from the last 24 hours.
        - Then we get to our query execution phase
            - The database will go through the index starting at the North America region segment, and navigate to the shelter post type section, scan through the timestamp-ordered entries starting from the 24-hour cutoff, and then apply the geographic filter for each post.
    - Phase 4: Result Aggregation and Merging
        - The Result Aggregator will collect the responses from all queried regions. And merge them into a single result set and sort the final result to give Sarah the most relevant to least relevant results.
    - Phase 5: User Data Enrichment - Distributed Join (if necessary)
        - In our case, since Sarah wanted the profiles of users who made the specific post, our system will also fetch user profile information via a distributed join that extracts user Ids, home regions, and then combining this with the fetched posts.

3. Implementation Plan

3.1 Technology Stack

Backend: Python (3.10+) with Flask

Python and Flask are optimal for prototyping due to Flask's lightweight structure. PyMongo library will be helpful for this project. All teammates have prior experience building REST APIs with Python and Flask.

Frontend: React with TypeScript

TypeScript will be more beneficial over JavaScript due to the extensive typechecking to prevent bugs. All teammates have prior experience building with React for the frontend. TypeScript is mostly similar to JavaScript, so there shouldn't be too much of a learning curve.

Database: MongoDB (6.0+) with Sharding and Replica Sets

MongoDB has native sharding support. Replica sets provide automatic primary election. There is also a builtin oplog which we can use for our synchronization step. There is also native support for location-based queries which is extremely critical for disaster scenarios.

Containerization: Docker

We can use Docker to simulate multiple regions by running multiple instances. We can simulate network partitions by controlling connectivity. It will hopefully be simple to simulate node failures and recoveries.

3.2 Development Phases

- Outline phases (e.g., Module Development, Integration, Testing) with expected deliverables.

Phase 1: Database Setup & Regional Node Implementation

Oct 15 - Oct 21, 2025 – Aditya

Objectives:

1. Set up MongoDB instances with replica set configuration for each geographic region
2. Set up Docker containers/networks to simulate multiple regions
3. Implement Flask endpoints for user operations
4. Establish database schemas for profiles, safety status, and help requests

Deliverables:

1. Docker Compose configuration for multi-region deployment
2. 3 regional clusters (North America, Europe, Asia-Pacific), each with 1 primary and 2 secondary
3. API with endpoints: /api/mark-safe, /api/post-help, /api/get-status
4. Database schemas with geospatial indexes

Phase 2: Geographic Data Partitioning

Oct 15 - Oct 21, 2025 – Abhave

Objectives:

1. Implement hash-based sharding within regions on userId
2. Configure sharding for horizontal data distribution
3. Create geo-location service to route requests to appropriate regional clusters

Deliverables:

1. Sharding configuration with hash-based partitioning on userId
2. Show query latency for local vs. cross-region requests

Phase 3: Inter-Region Synchronization Engine & Conflict Resolution

Oct 22 - Oct 31, 2025 – Yashwardhan

Objectives:

1. Develop asynchronous replication mechanism between regions
2. Implement oplog for updates

3. Build conflict resolution logic using LWW with timestamps

Deliverables:

1. Synchronization service running in each region
2. Oplog with persistent storage
3. Timestamp-based conflict resolution algorithm
4. Automated reconciliation process running at network recovery

Phase 4: Fault Tolerance & Failure Simulation

Nov 1 - Nov 10, 2025 – Abhave

Objectives:

1. Implement health monitoring and heartbeat protocol
2. Configure automatic failover for replica sets
3. Create failure simulation Docker scripts
4. Build dashboard showing node health and status

Deliverables:

1. Health monitoring service
2. Automatic failover configuration with <15 second recovery time
3. Island mode logic
4. Shell scripts to simulate failures listed in table above
5. Monitoring dashboard

Phase 5: Frontend Development & Integration

Nov 1 - Nov 10, 2025 – Abhave

Objectives:

1. Build React interface for platform
2. Create visualization components for system architecture and node status
3. Implement real-time updates using WebSockets or polling
4. Demo showing disaster scenarios and recovery

Deliverables:

1. User interface with features: mark safe, post help requests, view nearby updates
2. Real-time node status visualization with color-coded health indicators
3. Demo walk through a complete disaster cycle (isolation, island mode, recovery,

reconciliation)

Phase 6: Testing & Evaluation

Nov 11 - Nov 18, 2025 – Aditya

Objectives:

1. Test all system components
2. Measure metrics: throughput, latency, consistency delay, recovery time
3. Execute all planned failure scenarios
4. Perform load testing

Deliverables:

1. Complete test suite
2. Load testing results with varying numbers of concurrent users
3. Comparison of local vs. global query performance

Phase 7: Final Report & Documentation

Nov 19 - Nov 25, 2025

Objectives:

1. Write comprehensive final report
2. Prepare project presentation
3. Create guide and documentation

Deliverables:

1. Final project report
2. System architecture documentation
3. API documentation
4. Deployment and testing guide
5. Presentation slides for final demo

3.3 Testing and Verification Plan

For Replication, we will be doing one major test:

- Cross-Region Verification
    - Type: Unit Test
    - Tool: MongoDB Change Streams + Pytest
    - Functionality: Used to verify custom cross-region replication engine

    propagates updates between regions
    - Success Criteria: The updates propagate between regions within 5 seconds

For Consistency Testing, we will be doing 2 tests:

- Local Consistency Verification
    - Type: Unit Test
    - Tool: Pytest with MongoDB read/write operations
    - Functionality: Verify read-after-write consistency within single region
    - Success Criteria: Reads immediately reflect writes in same region
- Global Consistency Test
    - Type: Unit Test
    - Tool: Pytest with multi-region client simulation
    - Functionality: Verify all regions converge to same state after partition heals
    - Success Criteria: All regions show identical data within 10 seconds

For Fault Tolerance Testing, we will be doing 4 tests:

- Single Node Failure Recovery
    - Type: Unit Test
    - Tool: Docker commands (docker stop/kill) + Pytest
    - Functionality: Kill single MongoDB node and verify automatic failover
    - Success Criteria: New primary elected within 10 seconds; zero query failures
- Multi-Node Failure Test
    - Type: Unit Test
    - Tool: Docker commands (docker stop/kill multiple containers) + Pytest
    - Functionality: Simultaneously kill multiple nodes across regions
    - Success Criteria: System remains operational with at least one healthy node per region
- Disaster Scenario Simulation
    - Type: Synthetic Workload
    - Tool: Docker commands (docker stop, docker network disconnect) + Locust (user load) + MongoDB built-in failover
    - Functionality: Simulate realistic disaster with network partitions, node failures, and high user load
    - Success Criteria: Local queries succeed with <100ms latency; system remains responsive
- Partition Recovery Test
    - Type: Unit Test
    - Tool: Docker network reconnect + MongoDB Change Streams + Pytest
    - Functionality: Restore connectivity and verify automatic data synchronization
    - Success Criteria: Regions converge to consistent state within 30 seconds

4. Preliminary Evaluation Metrics

- Define the metrics you plan to measure later (e.g., throughput, latency, consistency delay, fault-recovery time). Explain how the design choices support these targets.

- Our performance metrics will focus on throughput and latency with our target throughput being 10,000 reads per second and 5,000 writes per second per region. These numbers are achievable through MongoDB's replica sets that distribute read load across secondary nodes. Additionally, our targets for query latency are <50ms for local queries and <300ms for cross-region queries. Due to our architecture's geographic partitioning, most queries execute locally within the user's region, while our lightweight Flask app and Docker containerization minimize overhead processing.
- We will be measuring consistency via delay, and cross-region propagation metrics. Our target for consistency delay is <5 seconds for cross-region propagation and <50ms for intra-region replication lag. Our asynchronous cross-region replication design allows local writes to complete without waiting for global synchronization, while MongoDB's Change Streams efficiently propagates updates to other nodes. We are hoping for conflict rates below 1% during normal operation and below 5% during network partitions.
- We are then going to be measuring availability metrics via Docker health checks, and MongoDB. Our target is 99.9% availability per docker container (region) with fault recovery times <10 seconds for node failures and <30 seconds for partition recovery. MongoDB replica sets will provide automatic failover and Docker health checks will enable rapid failure detection. We will also be measuring the region's ability to operate indefinitely in island mode during network partitions without data loss, and this is enabled by geographic partitioning ensuring each region maintains complete local data.
- For Scalability metrics, we will be measuring the system's ability to handle load balancing and auto-deployments. Our design seeks to achieve linear scaling up to 10 nodes per region with less than 10% latency degradation. We are also using Docker containerization to enable easy node addition and Flask's stateless architecture which supports unlimited horizontal scaling.

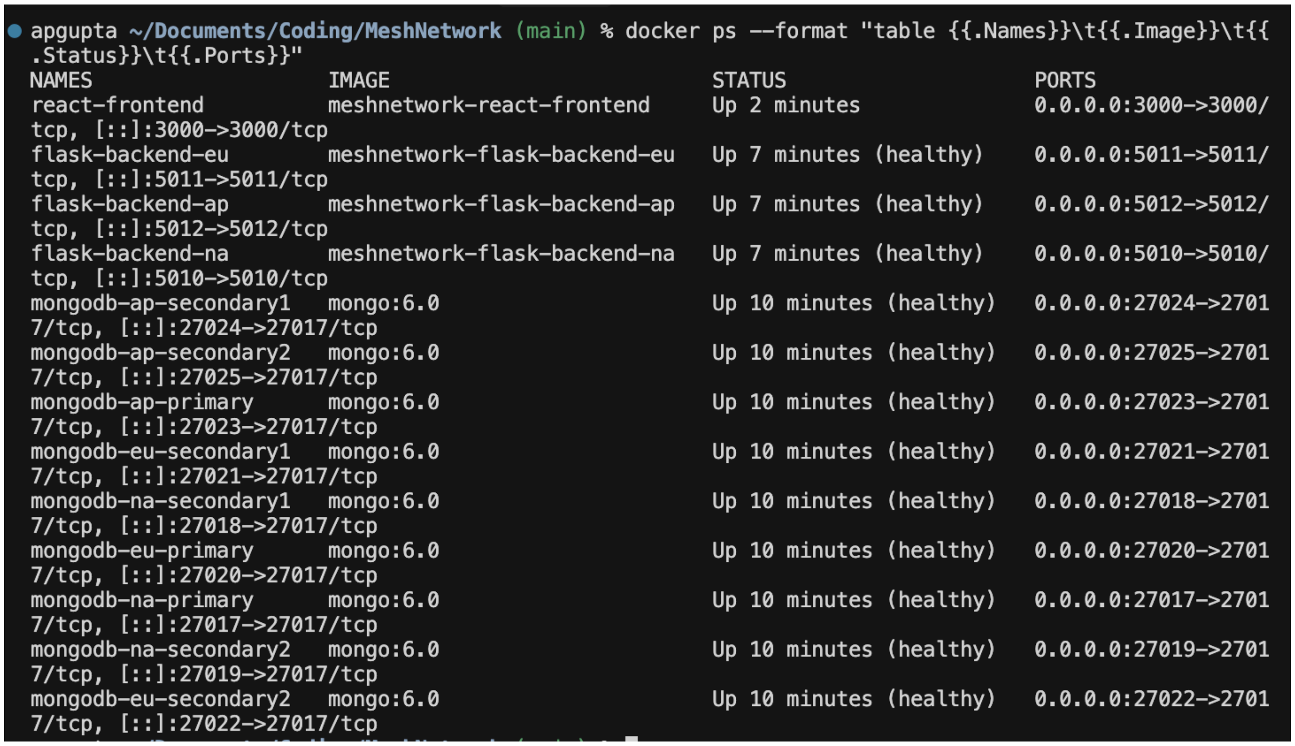
5. Implementation Progress

(This section demonstrates your current progress toward a functional / Working prototype.)

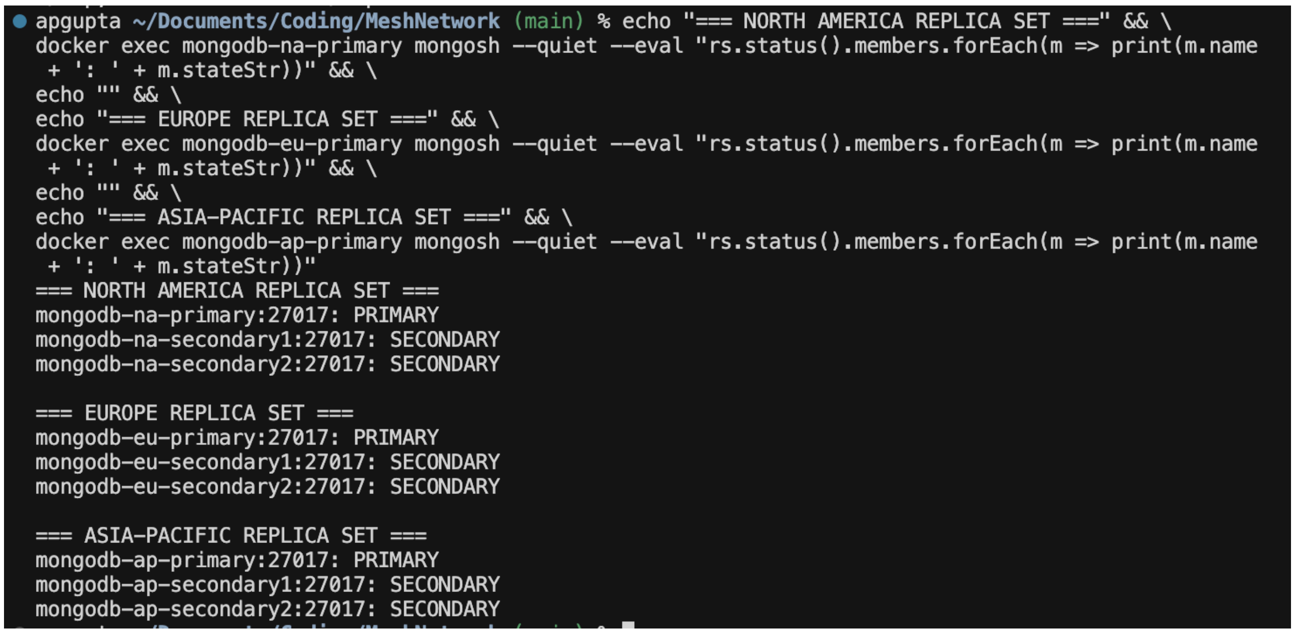
5.1 Completed Tasks

- Summarize modules, functions, or components that are already implemented. Include screenshots/outputs as evidence.

1. This shows all 3 regional clusters with MongoDB replicas (primaries and secondaries) + Flask backends + React frontend.

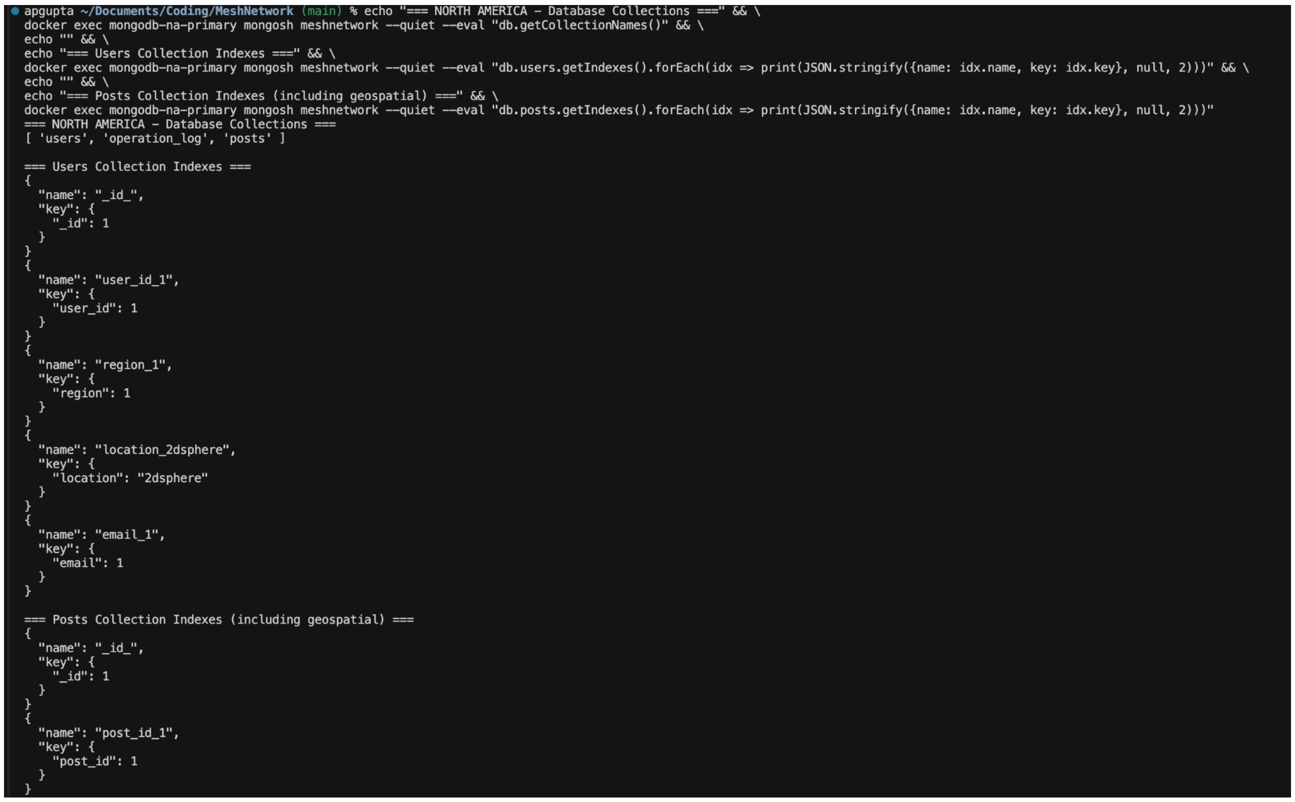


```
● apgupta ~/Documents/Coding/MeshNetwork (main) % docker ps --format "table {{.Names}}\t{{.Image}}\t{{
.Status}}\t{{.Ports}}"
NAMES                    IMAGE                          STATUS                     PORTS
react-frontend           meshnetwork-react-frontend     Up 2 minutes               0.0.0.0:3000->3000/
tcp, [::]:3000->3000/tcp
flask-backend-eu         meshnetwork-flask-backend-eu   Up 7 minutes (healthy)     0.0.0.0:5011->5011/
tcp, [::]:5011->5011/tcp
flask-backend-ap         meshnetwork-flask-backend-ap   Up 7 minutes (healthy)     0.0.0.0:5012->5012/
tcp, [::]:5012->5012/tcp
flask-backend-na         meshnetwork-flask-backend-na   Up 7 minutes (healthy)     0.0.0.0:5010->5010/
tcp, [::]:5010->5010/tcp
mongodb-ap-secondary1    mongo:6.0                      Up 10 minutes (healthy)    0.0.0.0:27024->2701
7/tcp, [::]:27024->27017/tcp
mongodb-ap-secondary2    mongo:6.0                      Up 10 minutes (healthy)    0.0.0.0:27025->2701
7/tcp, [::]:27025->27017/tcp
mongodb-ap-primary       mongo:6.0                      Up 10 minutes (healthy)    0.0.0.0:27023->2701
7/tcp, [::]:27023->27017/tcp
mongodb-eu-secondary1    mongo:6.0                      Up 10 minutes (healthy)    0.0.0.0:27021->2701
7/tcp, [::]:27021->27017/tcp
mongodb-na-secondary1    mongo:6.0                      Up 10 minutes (healthy)    0.0.0.0:27018->2701
7/tcp, [::]:27018->27017/tcp
mongodb-eu-primary       mongo:6.0                      Up 10 minutes (healthy)    0.0.0.0:27020->2701
7/tcp, [::]:27020->27017/tcp
mongodb-na-primary       mongo:6.0                      Up 10 minutes (healthy)    0.0.0.0:27017->2701
7/tcp, [::]:27017->27017/tcp
mongodb-na-secondary2    mongo:6.0                      Up 10 minutes (healthy)    0.0.0.0:27019->2701
7/tcp, [::]:27019->27017/tcp
mongodb-eu-secondary2    mongo:6.0                      Up 10 minutes (healthy)    0.0.0.0:27022->2701
7/tcp, [::]:27022->27017/tcp
```

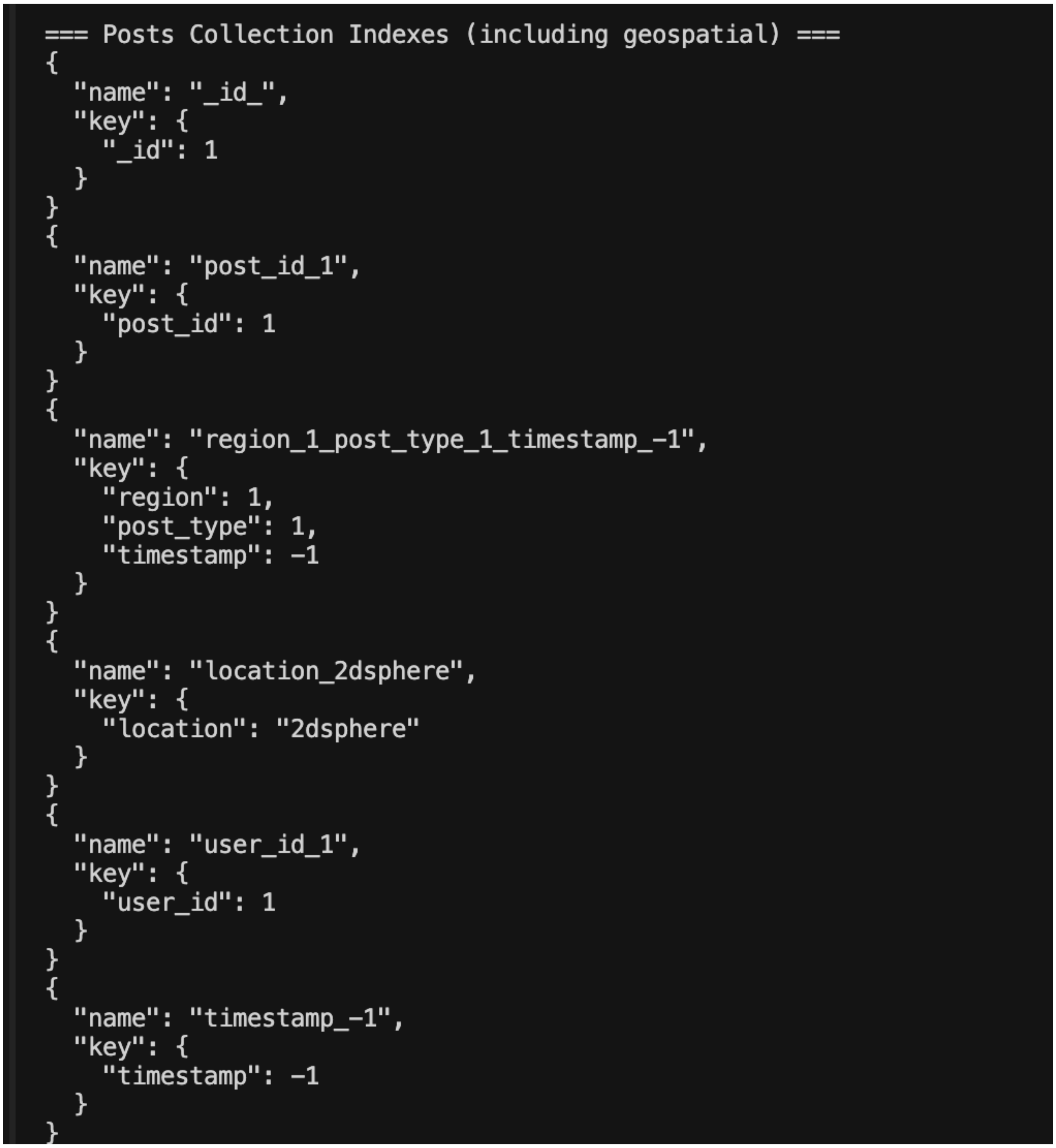


```
● apgupta ~/Documents/Coding/MeshNetwork (main) % echo "=== NORTH AMERICA REPLICA SET ===" && \
  docker exec mongodb-na-primary mongosh --quiet --eval "rs.status().members.forEach(m => print(m.name
   + ': ' + m.stateStr))" && \
  echo "" && \
  echo "=== EUROPE REPLICA SET ===" && \
  docker exec mongodb-eu-primary mongosh --quiet --eval "rs.status().members.forEach(m => print(m.name
   + ': ' + m.stateStr))" && \
  echo "" && \
  echo "=== ASIA-PACIFIC REPLICA SET ===" && \
  docker exec mongodb-ap-primary mongosh --quiet --eval "rs.status().members.forEach(m => print(m.name
   + ': ' + m.stateStr))"
  === NORTH AMERICA REPLICA SET ===
  mongodb-na-primary:27017: PRIMARY
  mongodb-na-secondary1:27017: SECONDARY
  mongodb-na-secondary2:27017: SECONDARY

  === EUROPE REPLICA SET ===
  mongodb-eu-primary:27017: PRIMARY
  mongodb-eu-secondary1:27017: SECONDARY
  mongodb-eu-secondary2:27017: SECONDARY

  === ASIA-PACIFIC REPLICA SET ===
  mongodb-ap-primary:27017: PRIMARY
  mongodb-ap-secondary1:27017: SECONDARY
  mongodb-ap-secondary2:27017: SECONDARY
```

2. These screenshots show that our database schema is complete with geospatial indices.

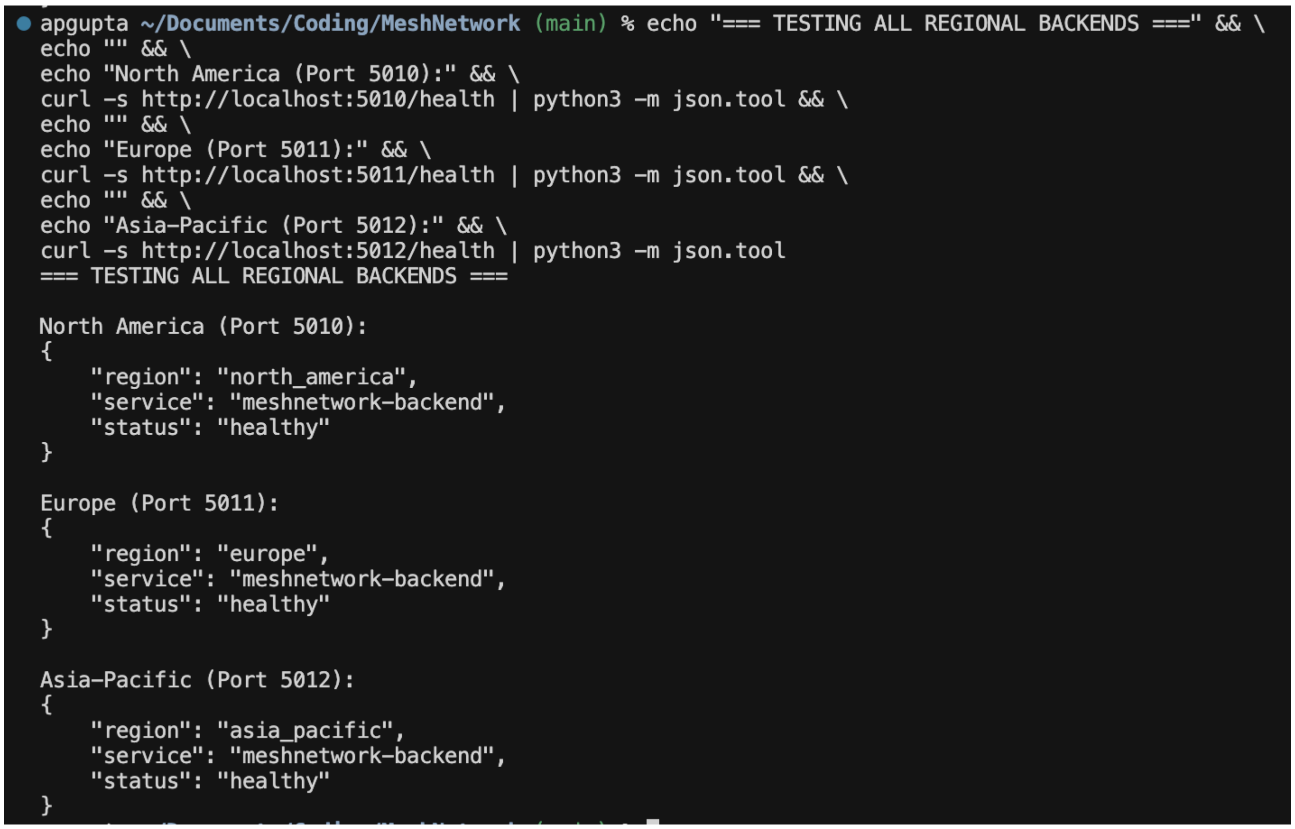


```
● apgupta ~/Documents/Coding/MeshNetwork (main) % echo "=== NORTH AMERICA - Database Collections ===" && \
  docker exec mongodb-na-primary mongosh meshnetwork --quiet --eval "db.getCollectionNames()" && \
  echo "" && \
  echo "=== Users Collection Indexes ===" && \
  docker exec mongodb-na-primary mongosh meshnetwork --quiet --eval "db.users.getIndexes().forEach(idx => print(JSON.stringify({name: idx.name, key: idx.key}, null, 2)))" && \
  echo "" && \
  echo "=== Posts Collection Indexes (including geospatial) ===" && \
  docker exec mongodb-na-primary mongosh meshnetwork --quiet --eval "db.posts.getIndexes().forEach(idx => print(JSON.stringify({name: idx.name, key: idx.key}, null, 2)))"
  === NORTH AMERICA - Database Collections ===
  [ 'users', 'operation_log', 'posts' ]

  === Users Collection Indexes ===
  {
    "name": "_id_",
    "key": {
      "_id": 1
    }
  }
  {
    "name": "user_id_1",
    "key": {
      "user_id": 1
    }
  }
  {
    "name": "region_1",
    "key": {
      "region": 1
    }
  }
  {
    "name": "location_2dsphere",
    "key": {
      "location": "2dsphere"
    }
  }
  {
    "name": "email_1",
    "key": {
      "email": 1
    }
  }

  === Posts Collection Indexes (including geospatial) ===
  {
    "name": "_id_",
    "key": {
      "_id": 1
    }
  }
  {
    "name": "post_id_1",
    "key": {
      "post_id": 1
    }
  }
```

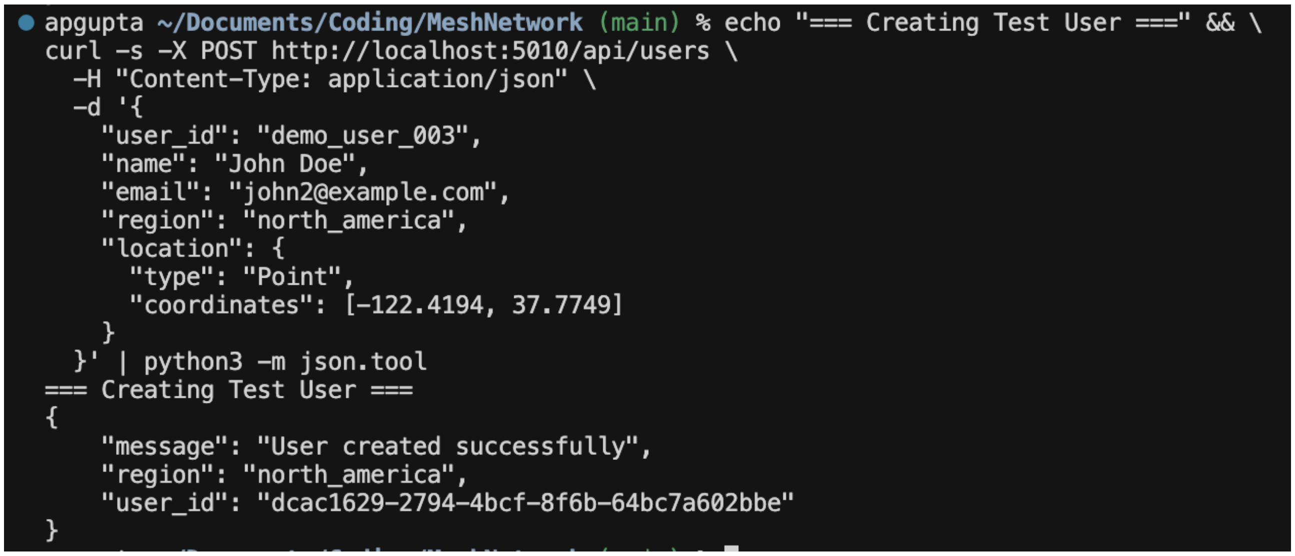


```
=== Posts Collection Indexes (including geospatial) ===
{
  "name": "_id_",
  "key": {
    "_id": 1
  }
}
{
  "name": "post_id_1",
  "key": {
    "post_id": 1
  }
}
{
  "name": "region_1_post_type_1_timestamp_-1",
  "key": {
    "region": 1,
    "post_type": 1,
    "timestamp": -1
  }
}
{
  "name": "location_2dsphere",
  "key": {
    "location": "2dsphere"
  }
}
{
  "name": "user_id_1",
  "key": {
    "user_id": 1
  }
}
{
  "name": "timestamp_-1",
  "key": {
    "timestamp": -1
  }
}
```

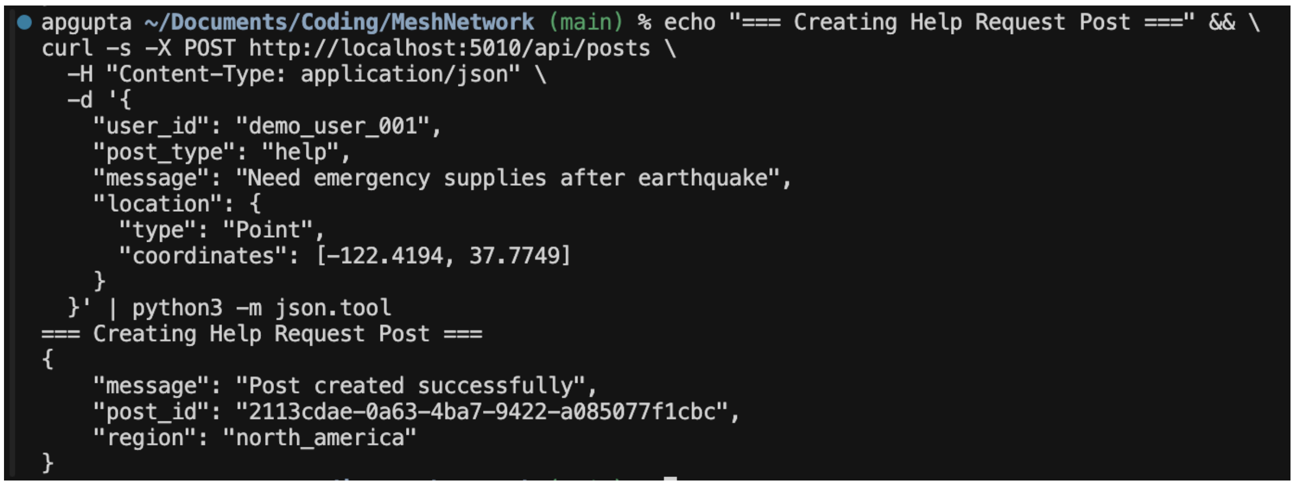
3. These screenshots show that our API endpoints are functional.



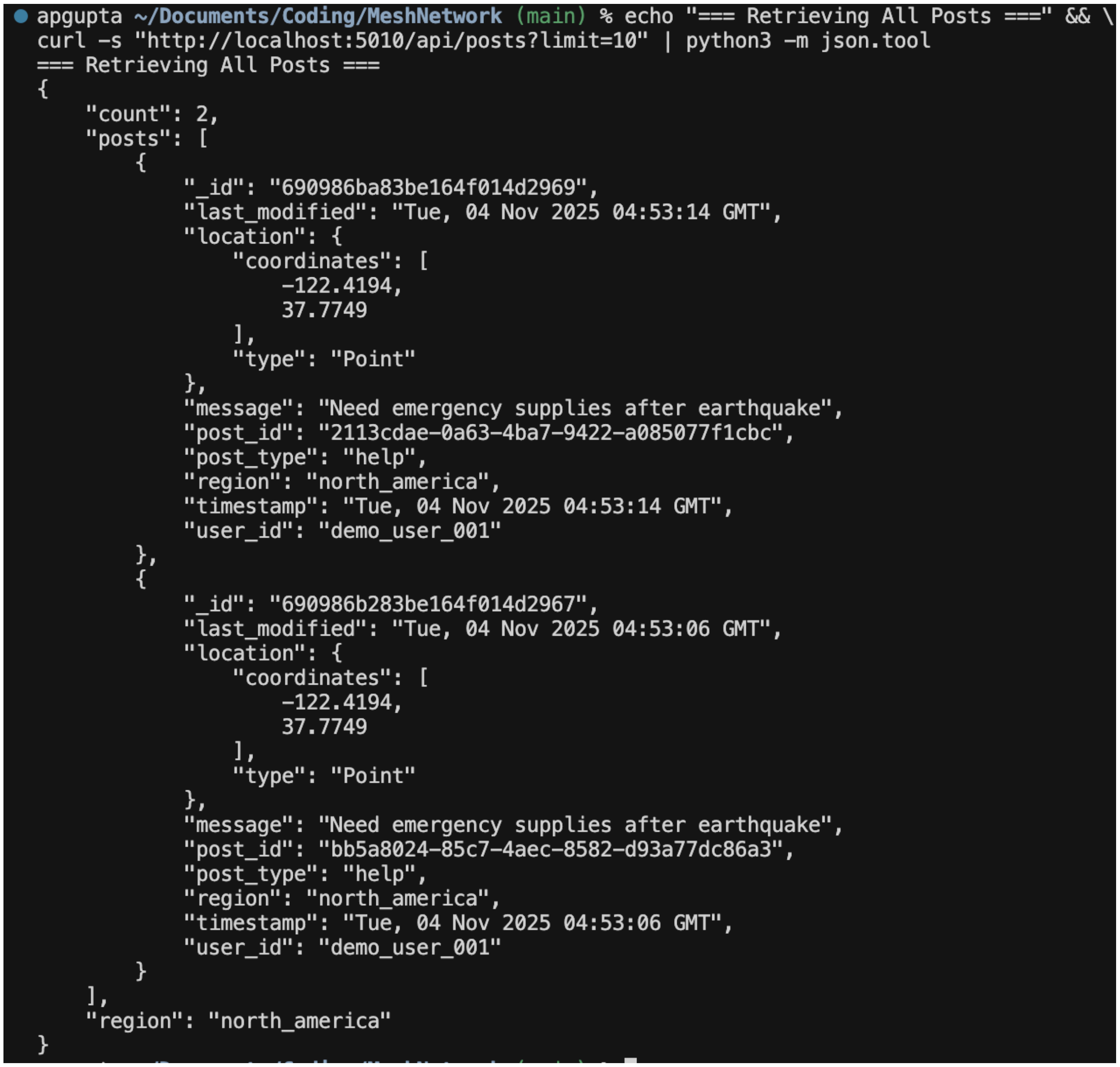
```
● apgupta ~/Documents/Coding/MeshNetwork (main) % echo "=== TESTING ALL REGIONAL BACKENDS ===" && \
  echo "" && \
  echo "North America (Port 5010):" && \
  curl -s http://localhost:5010/health | python3 -m json.tool && \
  echo "" && \
  echo "Europe (Port 5011):" && \
  curl -s http://localhost:5011/health | python3 -m json.tool && \
  echo "" && \
  echo "Asia-Pacific (Port 5012):" && \
  curl -s http://localhost:5012/health | python3 -m json.tool
  === TESTING ALL REGIONAL BACKENDS ===

  North America (Port 5010):
  {
      "region": "north_america",
      "service": "meshnetwork-backend",
      "status": "healthy"
  }

  Europe (Port 5011):
  {
      "region": "europe",
      "service": "meshnetwork-backend",
      "status": "healthy"
  }

  Asia-Pacific (Port 5012):
  {
      "region": "asia_pacific",
      "service": "meshnetwork-backend",
      "status": "healthy"
  }
```



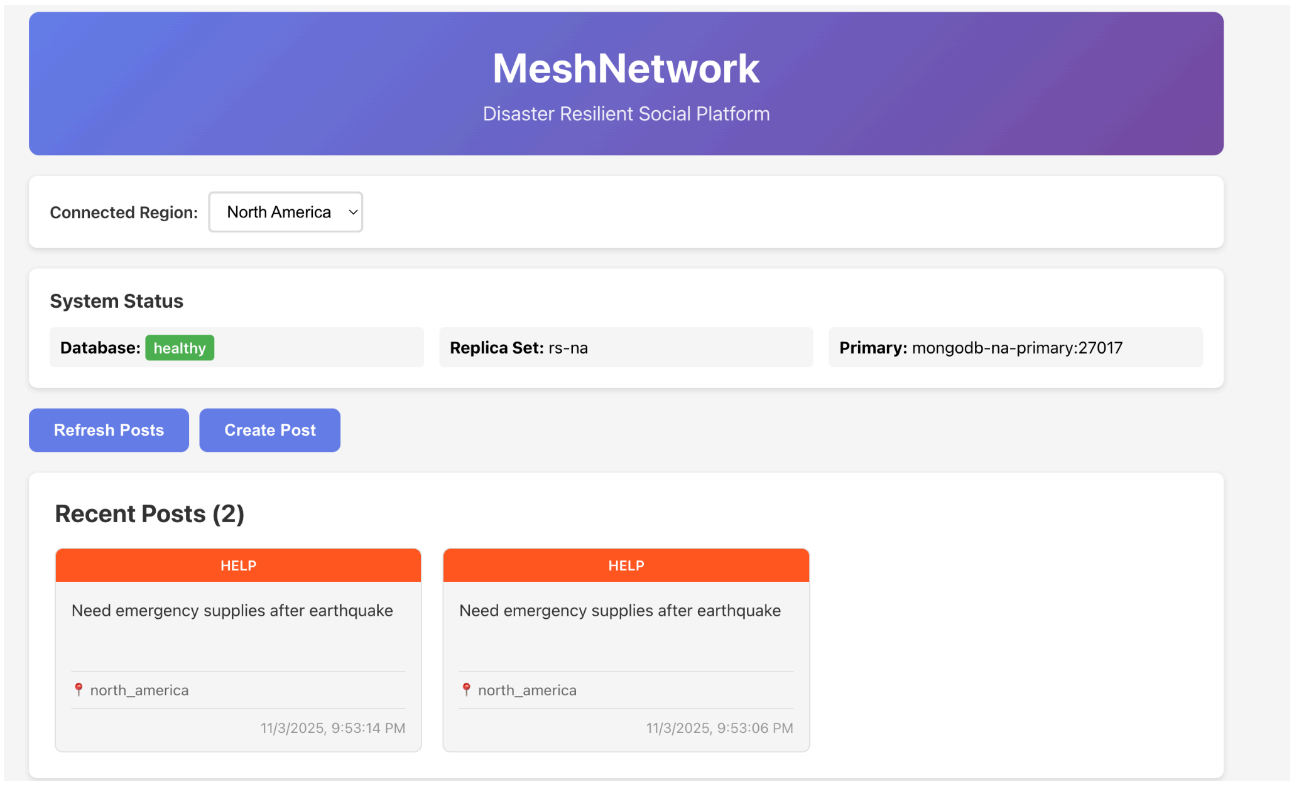
```
● apgupta ~/Documents/Coding/MeshNetwork (main) % echo "=== Creating Test User ===" && \
  curl -s -X POST http://localhost:5010/api/users \
    -H "Content-Type: application/json" \
    -d '{
      "user_id": "demo_user_003",
      "name": "John Doe",
      "email": "john2@example.com",
      "region": "north_america",
      "location": {
        "type": "Point",
        "coordinates": [-122.4194, 37.7749]
      }
    }' | python3 -m json.tool
  === Creating Test User ===
  {
      "message": "User created successfully",
      "region": "north_america",
      "user_id": "dcac1629-2794-4bcf-8f6b-64bc7a602bbe"
  }
```



```
● apgupta ~/Documents/Coding/MeshNetwork (main) % echo "=== Creating Help Request Post ===" && \
curl -s -X POST http://localhost:5010/api/posts \
  -H "Content-Type: application/json" \
  -d '{
    "user_id": "demo_user_001",
    "post_type": "help",
    "message": "Need emergency supplies after earthquake",
    "location": {
      "type": "Point",
      "coordinates": [-122.4194, 37.7749]
    }
  }' | python3 -m json.tool
=== Creating Help Request Post ===
{
    "message": "Post created successfully",
    "post_id": "2113cdae-0a63-4ba7-9422-a085077f1cbc",
    "region": "north_america"
}
```



```
● apgupta ~/Documents/Coding/MeshNetwork (main) % echo "=== Retrieving All Posts ===" && \
curl -s "http://localhost:5010/api/posts?limit=10" | python3 -m json.tool
=== Retrieving All Posts ===
{
    "count": 2,
    "posts": [
        {
            "_id": "690986ba83be164f014d2969",
            "last_modified": "Tue, 04 Nov 2025 04:53:14 GMT",
            "location": {
                "coordinates": [
                    -122.4194,
                    37.7749
                ],
                "type": "Point"
            },
            "message": "Need emergency supplies after earthquake",
            "post_id": "2113cdae-0a63-4ba7-9422-a085077f1cbc",
            "post_type": "help",
            "region": "north_america",
            "timestamp": "Tue, 04 Nov 2025 04:53:14 GMT",
            "user_id": "demo_user_001"
        },
        {
            "_id": "690986b283be164f014d2967",
            "last_modified": "Tue, 04 Nov 2025 04:53:06 GMT",
            "location": {
                "coordinates": [
                    -122.4194,
                    37.7749
                ],
                "type": "Point"
            },
            "message": "Need emergency supplies after earthquake",
            "post_id": "bb5a8024-85c7-4aec-8582-d93a77dc86a3",
            "post_type": "help",
            "region": "north_america",
            "timestamp": "Tue, 04 Nov 2025 04:53:06 GMT",
            "user_id": "demo_user_001"
        }
    ],
    "region": "north_america"
}
```


MeshNetwork

Disaster Resilient Social Platform

Connected Region: North America

System Status

Database: healthy | Replica Set: rs-na | Primary: mongodb-na-primary:27017

Recent Posts (2)

HELP

Need emergency supplies after earthquake

north_america

11/3/2025, 9:53:14 PM

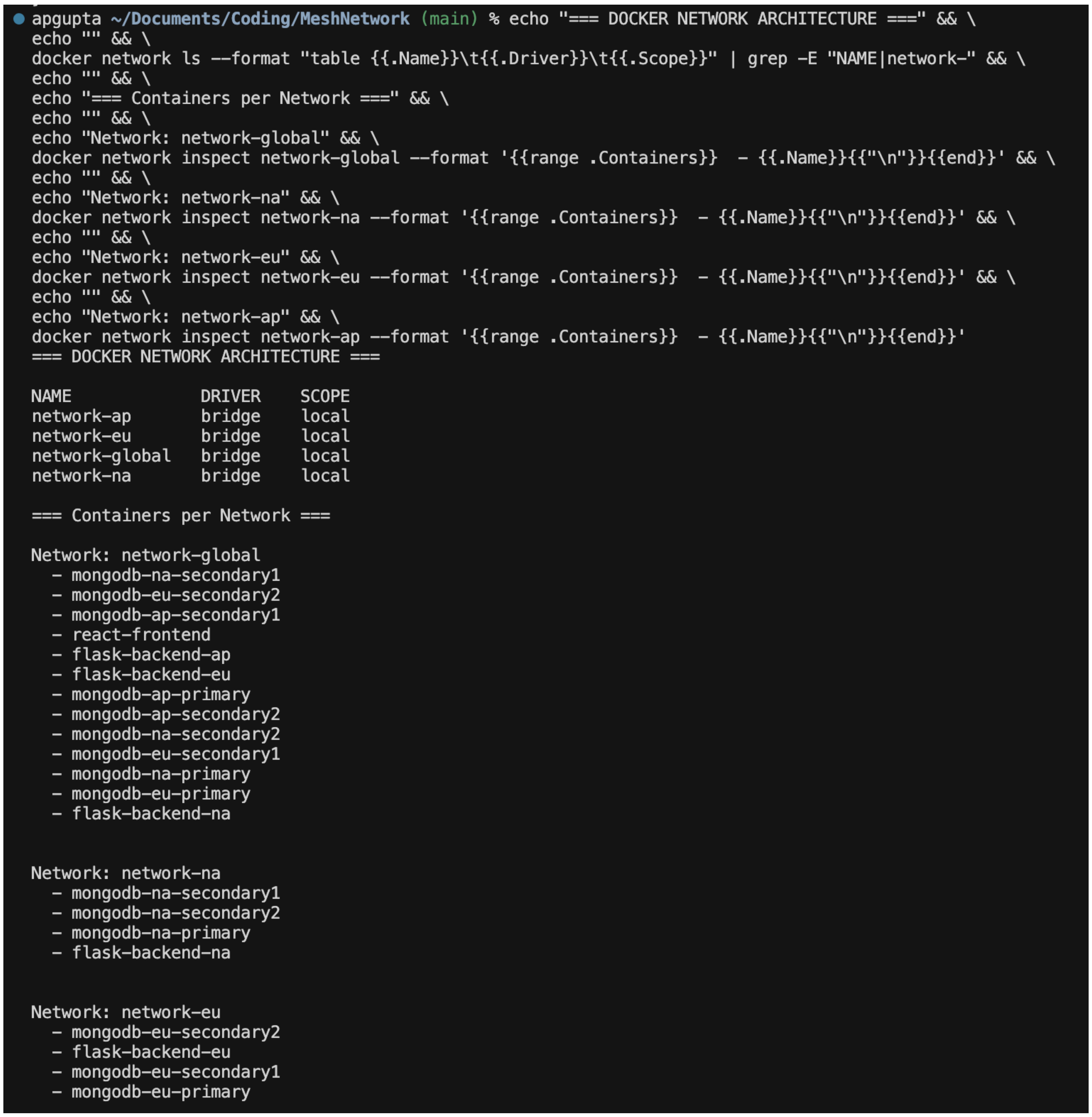
HELP

Need emergency supplies after earthquake

north_america

11/3/2025, 9:53:06 PM

4. Docker Network Visualization:



```
● apgupta ~/Documents/Coding/MeshNetwork (main) % echo "=== DOCKER NETWORK ARCHITECTURE ===" && \
  echo "" && \
  docker network ls --format "table {{.Name}}\t{{.Driver}}\t{{.Scope}}" | grep -E "NAME|network-" && \
  echo "" && \
  echo "=== Containers per Network ===" && \
  echo "" && \
  echo "Network: network-global" && \
  docker network inspect network-global --format '{{range .Containers}}  - {{.Name}}{{"\n"}}{{end}}' && \
  echo "" && \
  echo "Network: network-na" && \
  docker network inspect network-na --format '{{range .Containers}}  - {{.Name}}{{"\n"}}{{end}}' && \
  echo "" && \
  echo "Network: network-eu" && \
  docker network inspect network-eu --format '{{range .Containers}}  - {{.Name}}{{"\n"}}{{end}}' && \
  echo "" && \
  echo "Network: network-ap" && \
  docker network inspect network-ap --format '{{range .Containers}}  - {{.Name}}{{"\n"}}{{end}}'
  === DOCKER NETWORK ARCHITECTURE ===

  NAME             DRIVER    SCOPE
  network-ap       bridge    local
  network-eu       bridge    local
  network-global   bridge    local
  network-na       bridge    local

  === Containers per Network ===

  Network: network-global
    - mongodb-na-secondary1
    - mongodb-eu-secondary2
    - mongodb-ap-secondary1
    - react-frontend
    - flask-backend-ap
    - flask-backend-eu
    - mongodb-ap-primary
    - mongodb-ap-secondary2
    - mongodb-na-secondary2
    - mongodb-eu-secondary1
    - mongodb-na-primary
    - mongodb-eu-primary
    - flask-backend-na


  Network: network-na
    - mongodb-na-secondary1
    - mongodb-na-secondary2
    - mongodb-na-primary
    - flask-backend-na


  Network: network-eu
    - mongodb-eu-secondary2
    - flask-backend-eu
    - mongodb-eu-secondary1
    - mongodb-eu-primary
```

5.2 In-Progress Tasks

- List ongoing work and expected completion dates.
- Ongoing work
    - Nov 4-6: Implement MongoDB Change Streams listener and operation queue (3 days)
    - Nov 7-8: Build push/pull sync logic and Last-Write-Wins conflict resolution (2 days)

5.3 Challenges Encountered

- Initially, we used ports 5000-5002 for our Flask backend endpoints. However, Macbook reserves port 5000 for Airplay use. This issue made debugging a little difficult, since we only caught it much later. We learned that we should check OS-reserved ports before building next time.

- After we changed the ports from 5000-5002 to 5010-5012, our React frontend was still sending requests to port 5000 because the Docker container was being built from the cache. We learned that we should build with no-cache when we have confusing issues like this. We spent quite a bit of time tracing our steps before realizing the caching issue.
- Replica sets require time for primary election after rs.initiate(), but we attempted to create collections immediately, resulting in "not primary" errors. We can fix this by adding in sleep()'s so the primary election is completed first.

5.4 Next Steps

- Specify remaining work required to complete Milestone 3 (implementation and evaluation).

- The following is the remaining work required to complete Milestone 3 with their timelines and descriptions as well.
- Nov 7-8: Build push/pull sync logic and Last-Write-Wins conflict resolution (2 days)
- Nov 9-10: Test island mode operation and partition recovery (2 days)
- Nov 9-10: Integrate scatter-gather into main query flow, add timeout handling (2 days, parallel with island mode testing)
- Nov 11-13: Build React UI components (map view, post feed, post form) (3 days)
- Nov 14: System status dashboard with region health monitoring (1 day)
- Nov 15-16: Run throughput benchmarks, latency tests, consistency delay measurements (2 days)
- Nov 17: Disaster scenario simulations with Locust (1 day)