User Journey

1. Mission Creation

- The user launches the **Mission Control Dashboard** (React frontend) at http://localhost:3000.
- From the dashboard, they submit a mission request with relevant details.
- The frontend sends a POST /missions request to the **Commander's Camp (Flask API)**.
- The API assigns a unique mission_id, stores it in Redis, and queues the mission in RabbitMQ's orders_queue.
- The user immediately sees the mission status as **QUEUED**.

2. Mission Dispatch & Execution

- The **Soldier Workers** continuously listen to orders_queue.
- When a new mission arrives:
    - The worker marks it **IN_PROGRESS** and simulates execution for 5–15 seconds.
    - Using a secure, short-lived token, it publishes status updates back to RabbitMQ (status_queue).

3. Mission Tracking

- The **Commander's Camp API** listens to status_queue in the background.
- Each update from the worker is validated (token checked, timestamp recorded).
- The mission's latest state is persisted to Redis.
- The dashboard refreshes to show **IN_PROGRESS**, then **COMPLETED** or **FAILED**.

4. Mission Completion

- Once execution ends:
    - Successful runs (≈90%) are marked **COMPLETED**.
    - Failures are marked **FAILED**.
- The user can fetch all mission statuses via:
    - GET /missions → list view
    - GET /missions/<mission_id> → individual mission report

5. Health & Monitoring

- The user can verify system readiness anytime via:
- GET /health

  This endpoint validates connections to **Redis** and **RabbitMQ**.

Developer Experience & Challenges

1. Service Dependency Timing (Docker Compose)

- **Challenge:** RabbitMQ and Redis were not always ready when Flask or workers started, causing connection errors.
- **Solution:** Added retry mechanisms in `get_rabbitmq_conn()` and Redis connection handling with exponential backoff.

2. Token Rotation Synchronization

- **Challenge:** Worker tokens expired mid-mission, leading to rejected status updates.
- **Solution:** Implemented a **25-second proactive refresh interval** with a 5-second grace window to ensure continuity.

3. Message Loss & Acknowledgments

- **Challenge:** RabbitMQ messages occasionally vanished during rapid testing.
- **Root Cause:** Lack of message acknowledgment caused queue re-delivery or loss.
- **Fix:** Added `basic_ack` and `prefetch_count` for proper message acknowledgment and load balancing.

4. Persistence Handling (Redis Failures)

- **Challenge:** Redis downtime caused temporary status loss.
- **Solution:** Implemented a **dual in-memory + Redis store**, ensuring the API can still serve statuses even when Redis is unavailable.

5. Asynchronous Worker Scaling

- **Challenge:** With multiple workers running concurrently, maintaining consistent mission states was tricky.
- **Solution:** Used **durable queues** with **acknowledgments** to guarantee each mission is handled by exactly one worker.

6. Testing Under Load

- **Challenge:** Needed to validate concurrency, token handling, and message durability with 20+ missions simultaneously.
- **Solution:** Built a **test runner** that:
    - Launches multiple missions concurrently
    - Logs each step with timestamps
    - Summarizes total success/failure rates at the end