Api Avengers

MICROSERVICE HACKATHON PROBLEM STATEMENT

21st November, 2025

Location: Multi purpose building, IT Business Incubator (ITBI), CUET

Organizer: Department of Electronics and Telecommunication Engineer, CUET

Abir, the lead backend engineer of the ***CareForAll*** donation platform, had been preparing for the winter charity season for months. This was the time when thousands of people rushed in to support medical campaigns, and the team expected record-breaking traffic. Everything seemed ready—servers scaled, UI polished, and the mood was hopeful.Then the chaos began.

The first warning came as a support ticket from a donor claiming she had been charged twice. Abir brushed it off—maybe a misunderstanding. But within minutes, more complaints flooded in. The payment provider system had retried its webhooks, and because the system had **no idempotency**, every duplicate webhook triggered another charge. Abir felt a knot in his stomach. There weren't even proper logs to confirm what happened.

Soon after, another crisis hit. Donors were being charged, yet campaign totals stayed the same. Abir traced the problem to a mid-request crash: the Pledge Service wrote the pledge to the database but failed to publish the event. Without an Outbox or retry system, the donation vanished from the rest of the platform. Campaign owners began calling support in panic.

As the night went on, the gateway system behaved even more strangely. Some pledges received "captured" webhooks before "authorized." With no state machine enforcing order, the system overwrote states backward—from CAPTURED to AUTHORIZED—breaking totals entirely. At one point, a campaign total even showed a negative number. Screenshots spread online before Abir could react.

The worst part? **No monitoring. No alerts. No tracing.**Abir had no way to track which pledges were duplicated, lost, or stuck. He was blindly digging through scattered logs while thousands of donors continued refreshing pages.

And at the peak of traffic, the **final blow** hit.The Totals endpoint recalculated sums from scratch for every request. With thousands of simultaneous donors, the database hit 100% CPU and practically shut down. Campaigns showing "0 raised" sent donors and organizers into chaos. The platform's reputation collapsed within hours.

By dawn, Abir sat exhausted in front of his screen. It was clear that the system hadn't failed because of high traffic—it failed because the architecture lacked the foundations needed for real-world conditions: **idempotency, reliable events, state control, read models, retries, and observability**.

He knew they needed to rebuild the entire system—this time correctly. So he hired **your team** to complete the challenge and accomplish the mission.You are tasked with designing and implementing the **next-generation fundraising backend**—one that can withstand the chaos that destroyed the old system. Your solution must be more robust, more predictable, and more transparent. It must survive retries, failures, races, and real-world disorder ***without breaking correctness***.

CHECKPOINT 1 : *ARCHITECTURE & DESIGN*

Your team needs to map out all the core services in the ***CareForAll*** ecosystem. Since different components of the fundraising flow can fail in different ways, you must define clear boundaries, responsibilities, and interactions between services. The architecture must be fault-tolerant, and capable of handling sudden traffic bursts that can exceed a 1000 requests per seconds. Create an **architectural diagram** to explain your design to the judges. Additionally, design **data models** for each service. Keep both your data models and API design simple and clear.You may write down your

*In terms of scalability, you are **not required to use any orchestrator such as Kubernetes or Docker Swarm**. Instead, you may demonstrate your scalability strategy using Docker Compose features (e.g., scaling services with replicas). This is sufficient for showcasing how your system would handle increased load in a real-world environment.*

CHECKPOINT 2 : *CORE IMPLEMENTATION*

With the system design in place, your team should implement all the services together. There is no need to build a full-featured frontend—focus primarily on the core backend functionality. A minimal frontend is **sufficient** to demonstrate the system to the judges. For each service, implement only the essential API endpoints and handle **inter-service communication** where required. The frontend must access all services through **a single base URL**, which may require adding a load balancer or API gateway. Each service should include unit tests, with integration tests encouraged where possible.

The system must maintain **transparent** donation histories for both registered and unregistered users.**Fix the payment gateway service with proper unit testing and ensure idempotency issues are resolved**. Additionally, a simple admin panel is required to manage and monitor campaigns.

As bonus features, real-time chat support and a notification service for users and administrators may be implemented.

Once the implementation is complete, dockerize all services, ensuring the docker-compose file is fully **self-contained** and does not rely on any external dependencies, as the judges will run the entire system directly.

CHECKPOINT 3 : *OBSERVABILITY & MONITORING*

Set up **monitoring**, **logging**, and **tracing** for your services using Docker Compose. You must demonstrate **end-to-end** tracing of a full donation workflow and include a test scenario showing system behavior under stress or partial failure. For metrics, you may use tools like NodeExporter, cAdvisor, Prometheus, and Grafana. For logging, Logstash/Beats with Elasticsearch and Kibana are suitable options. For tracing, use OpenTelemetry with Jaeger or Dynatrace. All events and logs should be stored in Elasticsearch. ***A minimal observability setup is acceptable, and tool selection is up to the participants.***

CHECKPOINT 4 : *CI/CD PIPELINE*

Finally, you must create a CI/CD pipeline. You may use **GitHub Actions, Jenkins, or any similar CI/CD tool of your choice**—the names mentioned are examples, not requirements. Set up workflows that automatically run tests on every pull request or push to the default branch. No code should be merged into the default branch unless the CI stage succeeds. Your pipeline must also be intelligent enough to detect which microservice has changed and run tests or build Docker images **only for that specific service**, ensuring both speed and efficiency—just like real-world microservice development pipelines. You must follow **proper versioning practices** for all services. Each service should maintain clear semantic versions (e.g.,***v1.0.2***), and your CI/CD pipeline should tag Docker images and releases using these versions.

*Additionally, as a **bonus**, you may demonstrate a lightweight deployment step by automatically spinning up the entire system using **docker compose up** during the CD phase, showcasing how the platform can be launched end-to-end with a single automated workflow.*

***DELIVERABLE* :**

1. Teams must submit a presentation slide deck through a form that will be provided at the end of the hackathon. ***This presentation must be completed within the 8-hour hackathon window***
2. Each team must submit **a single GitHub repository** containing all services. ***No commits may be pushed to the default branch after the hackathon ends***.

***GENERAL INSTRUCTIONS* :**

- These are some general instructions for participants of any segment:
- Participants must start a brand-new GitHub repository onsite and submit it before the competition ends.
- All work must be developed during the hackathon; projects that are pre-built or partially prepared beforehand will ***not be accepted***.
- If teams use publicly available code, proper attribution is required.
- Unless stated otherwise, teams are free to choose their preferred technology stack.

This challenge is intentionally extensive, and it is not expected that participants will build every feature within the time limit. Focus on the core components first.

Any unfair practices, misconduct, or violation of rules will result in disqualification. The organizers reserve the right to make all final decisions, and participants will be informed if any rule changes occur during the event.

Api Avengers

MICROSERVICE HACKATHON PROBLEM STATEMENT

21st November, 2025

Location: Multi purpose building, IT Business Incubator (ITBI), CUET

Organizer: Department of Electronics and Telecommunication Engineer, CUET

Abir, the lead backend engineer of the ***CareForAll*** donation platform, had been preparing for the winter charity season for months. This was the time when thousands of people rushed in to support medical campaigns, and the team expected record-breaking traffic. Everything seemed ready—servers scaled, UI polished, and the mood was hopeful.Then the chaos began.

The first warning came as a support ticket from a donor claiming she had been charged twice. Abir brushed it off—maybe a misunderstanding. But within minutes, more complaints flooded in. The payment provider system had retried its webhooks, and because the system had **no idempotency**, every duplicate webhook triggered another charge. Abir felt a knot in his stomach. There weren't even proper logs to confirm what happened.

Soon after, another crisis hit. Donors were being charged, yet campaign totals stayed the same. Abir traced the problem to a mid-request crash: the Pledge Service wrote the pledge to the database but failed to publish the event. Without an Outbox or retry system, the donation vanished from the rest of the platform. Campaign owners began calling support in panic.

As the night went on, the gateway system behaved even more strangely. Some pledges received "captured" webhooks before "authorized." With no state machine enforcing order, the system overwrote states backward—from CAPTURED to AUTHORIZED—breaking totals entirely. At one point, a campaign total even showed a negative number. Screenshots spread online before Abir could react.

The worst part? **No monitoring. No alerts. No tracing.**Abir had no way to track which pledges were duplicated, lost, or stuck. He was blindly digging through scattered logs while thousands of donors continued refreshing pages.

And at the peak of traffic, the **final blow** hit.The Totals endpoint recalculated sums from scratch for every request. With thousands of simultaneous donors, the database hit 100% CPU and practically shut down. Campaigns showing "0 raised" sent donors and organizers into chaos. The platform's reputation collapsed within hours.

By dawn, Abir sat exhausted in front of his screen. It was clear that the system hadn't failed because of high traffic—it failed because the architecture lacked the foundations needed for real-world conditions: **idempotency, reliable events, state control, read models, retries, and observability**.