Project Closure Report

1. Project Progress Assessment

a. What Worked Well, What Didn't? Lessons Learned and Justification

One of the biggest challenges we faced initially was interpreting the assignment's requirements. While the overarching objective was clear, several smaller specifications were ambiguous. This ambiguity necessitated extensive planning and clarification sessions within the team. Although this upfront planning phase was time-consuming, it ultimately served as a strong foundation for the successful implementation of our solution. The lesson learned here was clear: investing time in thorough planning at the beginning significantly reduces ambiguity and rework during development.

Another major challenge was our collective unfamiliarity with Kubernetes. As most team members had never worked with it before, there was a steep learning curve. Setting up the Kubernetes cluster, configuring services, understanding concepts like pods, services, deployments, and ingress controllers took considerable effort. Debugging deployments was particularly tricky, especially in scenarios where inter-service communication failed due to networking issues or improper configuration. This again reinforced the importance of diving deep into documentation, experimenting, and developing a solid foundational understanding.

Communication within the team worked fairly well overall. Most members were engaged and contributed effectively. However, there were occasional issues related to meeting internal deadlines. These were often caused by misunderstandings or dependencies between services, which further emphasizes the challenges of working in a distributed system. We realized that in systems where microservices depend on each other's availability and interfaces, any single point of failure can compromise the reliability of the entire setup. This experience highlighted a crucial design principle: microservices should be resilient and loosely coupled wherever possible.

One unexpected lesson was the significant communication overhead inherent in team projects. Even when everyone is motivated and technically proficient, aligning efforts, synchronizing work progress, and ensuring consistent design decisions adds a layer of complexity that is often underestimated. This experience underlined that team projects not only test technical ability but also communication, documentation, and planning skills.

To justify our assessment, we conducted retrospectives at key milestones. These helped us realize that while the initial stages were slow, our productivity ramped up significantly once we had set up our Kubernetes environments and CI/CD pipelines. We also maintained a shared documentation platform, which served as a single source of truth for API contracts and service dependencies. These practices ultimately contributed to the relatively smooth execution of the later stages.

b. Exceeding the Estimated Effort

While we had a general estimate of how long the implementation would take, we ended up exceeding our initial timeline slightly. The main reasons were:

- **Unfamiliarity with Kubernetes**: Significant time was spent learning how to configure and troubleshoot services.

- **Inter-service Dependencies**: Due to the complex dependencies between services, debugging often required tracing the flow of data across multiple components.
- **Integration Effort**: While implementing each microservice was relatively straightforward, integrating them into a cohesive system required more effort than anticipated.

These overruns were not due to poor planning but rather the unpredictable nature of working with new technologies in a distributed architecture.

c. What Would We Do Differently?

While overall we are satisfied with how the project progressed, there are some aspects we would approach differently in the future. Most notably:

- **Stricter Internal Deadlines**: Although we eventually delivered everything on time, internal deadlines were occasionally missed. Establishing more rigorous internal milestones would help distribute the workload more evenly.
- **Early Integration**: We could have started integrating services earlier in the process to detect interface mismatches and failures sooner.
- **Automated Testing Pipelines**: Adding automated testing, particularly integration tests, would have saved time in the long run and increased reliability.

Aside from these points, we believe our approach to planning and task division was effective and well-structured. Every team member had a clear role, and we maintained strong communication throughout.

2. Final Software Assessment

a. What Went Well?

Our final software implementation included all the required components outlined in the assignment. Each microservice runs reliably and independently, and when composed together in the Kubernetes cluster, they form a functional simulation of a smart-vehicle system.

Services were designed to be modular, each encapsulating a single responsibility such as simulating a GPS sensor, collecting telemetry data, detecting events, visualizing data, or executing automatic interventions (e.g., braking). Communication between services was done using REST APIs, which worked well after some initial debugging.

Under simulated low-load scenarios (e.g., simulating a few vehicles), the system behaved as expected without any observable crashes or memory issues. Each component was containerized and deployed successfully on our Kubernetes cluster, and we could monitor and scale services as needed.

Additionally, our use of logs and observability tools allowed us to track down bugs relatively quickly. We also documented all APIs clearly, which facilitated integration efforts across the team.

b. What Would We Solve Differently Next Time?

Although the overall software works well, there are a few areas we would refine in a real-world deployment scenario:

- **Programming Language**: Most of our services were implemented in Python, which is great for rapid prototyping but not ideal for high-performance or production-level services. In future implementations, we would consider using a statically typed and compiled language like Go or Rust for performance and reliability benefits.
- **Type Safety**: Python's dynamic typing can lead to runtime errors that are difficult to catch in advance. This became a small but recurring issue, especially when passing complex JSON payloads between services.
- **Architectural Constraints**: The system architecture was largely predetermined by the assignment. In a professional setting, we would have explored different design patterns—such as event-driven architectures using message brokers like Kafka or NATS for better decoupling between services.
- **Frontend Design**: Our frontend directly queried the microservices, which simplified the design but exposed the system to tight coupling. In the future, we would introduce a backend layer that mediates between the frontend and microservices, adding an additional layer of abstraction and security.

Lastly, scalability remains an open question. We haven't tested our system under high load (e.g., thousands of vehicles). In such scenarios, performance bottlenecks may emerge. To support such load, we would need to invest more in performance optimization, caching strategies, rate limiting, and load balancing.

3. Personal Conclusion of the Course

From a personal and academic perspective, this course was one of the most valuable experiences we had during our studies. It not only required us to apply theoretical knowledge from various domains—such as networking, system architecture, and DevOps—but also encouraged us to explore new technologies like Kubernetes, container orchestration, and service discovery.

The open-ended nature of the assignment fostered creativity. We were free to make many of the architectural and implementation decisions, which made each team's submission unique. This degree of autonomy is rare in university assignments and made the project particularly engaging and enjoyable.

Moreover, the hands-on nature of the course was its greatest strength. Instead of simply learning about distributed systems from a textbook, we built one ourselves. This experiential learning approach helped solidify concepts in a way that lectures alone never could.

We are confident that we will encounter distributed systems in our future careers, whether in cloud infrastructure, backend development, or data engineering. Having built one from the ground up during university gives us a significant advantage in terms of understanding complexity, planning large-scale systems, and integrating multiple technologies.

In conclusion, we greatly appreciated the learning opportunity this course provided. It was a demanding but highly rewarding experience, and one that significantly improved our technical, organizational, and collaborative skills.

We would also be very interested in seeing how other teams approached the same problem, as it would provide further insight into different architectural choices, implementation techniques, and design philosophies. Sharing and comparing outcomes would enhance the learning experience even further and expose us to alternative approaches we might not have considered.