FACULTY OF INFORMATION TECHNOLOGY

ITSMA4-14

Project 1

STUDENT NAME:

NAME	SURNAME	STUDENT NUMBER
John	Crouse	EDUV8219015

Project 1 submitted in partial fulfilment of the requirements for ITSMA4-14

Date: 13/06/2025

Table of Contents

Deliverable 1

1. Introduction

The aim of this report is to analyse Droppa's architectural transition from monolithic structure to microservices and explore how key principles and design strategies improve the system's performance, scalability, and agility. This report focuses on Droppa's operational context in the project management software sector and is guided by microservices principles. These principles include single responsibility, decentralised data management, and API gateways.

2. Single Responsibility Principle in Microservices

The Single Responsibility Principle (SRP) is a guiding principle which states that each module or service should only change for one reason. Applying SRP helps to build microservices that implement a small set of closely related business functions (Rudrabhatla, 2020). For Droppa, SRP ensures that microservices like task tracking, notifications, and user authentication can be developed, deployed, and maintained independently. This fastens fixing bugs and updating features as well as enhances system maintainability. This improves overall efficiency of microservices.

3. Decentralised Data Management in Droppa

Decentralised data management allows services to be loosely coupled. It also allows the best database to be chosen. Moreover, it allows each microservice in Droppa to maintain its own database. This enhances performance, reliability, security, and task speed (Ianculescu & Alexandru, 2020). Enhanced performance will lower Droppa's data access latency and improve the responsiveness of their system. Enhanced reliability will mean that data per service will be isolated and failures in one service's data store will not corrupt the whole application (Venčkauskas *et al*., 2023). Enhanced security will allow different services to implement tailored access controls, reducing security risks for Droppa. Faster data-related tasks like inserts, updates, and queries will allow Droppa to deliver to users with faster response times. This will also allow background operations to be automated a lot more efficiently. Droppa can ensure better data resilience and responsiveness in its platform by avoiding bottlenecks and single points pf failure.

4. Advantages of Microservices for Droppa

Compared to monolithic architectures, microservices offer Droppa several benefits. These include flexibility, scalability, operational efficiency, and simpler development and maintenance (El Khalyly *et al*., 2020). With increased flexibility, teams at Droppa can adopt the best tools and frameworks per service. Droppa's adoption of microservices will allow their services to scale independently based on demands. Microservices will make deployments faster, improve fault isolation, and lead to easier team autonomy (Kansal & Balasubramaniam, 2024). All these improvements will make operations more efficient at Droppa. All the advantages mentioned are needed for Droppa to rapidly innovate and serve a larger scale user base.

5. Scalable Growth Strategies

To manage increased demand, Droppa can use three key strategies. Firstly, Droppa can use container orchestration platforms like Kubernetes for automated scaling and load balancing. This will also enhance deployment and management efficiency (Ait Said *et al*., 2024). Secondly, Droppa can implement autoscaling groups in cloud environments. This will help them allocate resources dynamically as well as enable them to adjust their scaling strategies to their specific needs (Suleiman & Murtaza, 2024). Thirdly, Droppa can use tools like Prometheus and Grafana to monitor services. They can provide valuable real-time insights and optimisation.

6. Role of API Gateways

Application Programming Interface (API) gateways act as a single-entry point into the microservices system. For Droppa, API gateways will help to Droppa to manage traffic. API gateways will help achieve this by limiting request rates, closing slow connections, and limiting the number of connections that can be established (Gadge & Kotwani, 2018). API gateways will enforce Droppa's security via authentication and rate-limiting. This will be done by granting identity-based authorisation to users (Zdun *et al*., 2018). Moreover, API gateways enable observability through metrics and logging. Through the three pillars of observability: performance metrics, distributed tracing, and logging strategies, API gateways will help maintain the reliability, stability, and performance of Droppa's systems (Tadi, 2022). Overall, API gateways will improve system management by maintaining a consistent client interface and simplifying service discovery.

7. Inter-Service Communication Optimisation

Effective communication is critical in microservices. Droppa can use asynchronous messaging platforms like Rabbit and Kafka. This way messages will only be sent to the appropriate microservices, and requests will not have to wait for responses, ultimately reducing latency (Weerasinghe & Perera, 2023). Droppa can apply circuit breakers and retries to handle service failures easier. A circuit breaker will help Droppa to minimise the impacts of security breaches by setting a maximum microservice failure tolerance threshold (Haindl *et al*., 2024). Lastly, Droppa can use patterns like Saga for long-running transactions. This would ensure eventual consistency. By using asynchronous messaging, circuit breakers, and eventual consistency, Droppa can ensure scalability, resilience, and responsiveness in their communication flows.

8. Importance of CI/CD in Microservices

Continuous Integration/Continuous Deployment (CI/CD) pipelines are essential for Droppa's microservices. CI/CD pipelines enable rapid, frequent, and efficient deployments. They automate testing and validation. They speed up software delivery to production and ensure code aligignment with best practices (Dakić, 2024). Lastly and most importantly, they reduce human error and downtime. Moreover, with containerisation and service virtualisation, CI/CD pipelines can mitigate pipeline complexity and managing service dependencies for Droppa. This would also allow consumer applications to integrate new versions of common microservices more efficiently (Badampudi *et al*., 2025).

9. Migration Challenges and Mitigation

Droppa may face several migration challenges. The most common and important challenges include service sprawl, data consistency challenges, and cultural shifts. Service sprawl can be minimised through centralised observability and governance. A centralised data catalogue and metadata repository will help Droppa to monitor the origin and changes of data across services (George, 2025). This will improve Droppa's transparency and auditability. Data consistency challenges can be resolved by using the eventual consistency model. This model removes the need of immediate consistency across all parts of the system. It will also guarantee that all data replicas in the system will eventually be merged into a consistent state, given enough time (dos Santos Silva, 2024). Cultural shifts can be managed through team training and DevOps adoption. DevOps adoption will promote collaboration between development and operation teams by breaking down traditional silos (Fritzsch et al., 2023). This will encourage a shared responsibility mindset as well as continuous feedback and communication. These mitigation strategies with proper KPIs and rollback mechanisms will reduce Droppa's migration challenges.

10. Foundational Principles of The Microservices Way

Droppa should adopt several key principles. The most important principle in the microservices way is the Single Responsibility principle. This principle explains that each microservice should be built to perform one specific function. This enhances testability, maintainability, and clarity (Rudrabhatla, 2020).

Domain-Driven Design (DDD) will help align services to Droppa’s business domains. DDD will help design and implement high quality software, as well as build a common language between software developers, architects, and domain experts (Velepucha & Flores, 2023).

The Automy principle includes CI/CD, testing, and deployment pipelines. This would make microservices independently deployable and loosely coupled, allowing teams to build, test, and scale services without impacting others.

The Decentralised Data Management principle allows each service to manage its own database. This prevents data bottlenecks from occurring as well as enable flexibility, improved fault isolation, and accelerated development. Moreover, it increases business performance and task speed (Ianculescu & Alexandru, 2020).

The Resilience and Fault Tolerance principle includes fault tolerance through redundancy and isolation. Models like MapReduce are very good at accomplishing this (Baboi *et al*., 2019). This principle explains that microservices should handle failures with minimal difficulty without affecting the whole system. It uses several techniques that include timeouts, circuit breakers, and retries.

The Scalability principle explains that services should be scaled independently. This would enable Droppa to dynamically and efficiently allocate resources based on demand. Scalability can be implemented as vertical or horizontal scaling. Vertical scaling, also known as “scaling up”, adds more resources to a machine, but suffers the drawback of a tremendous increase in cost. Horizontal scaling, also known as “scaling out, adds more machines and distributes workload, but suffers the drawback of increased complexity (Blinowski *et al*., 2022).

The Observability principle focusses on logging, monitoring, and tracing. These are essential for gaining insight into service health, performance, and system-wide behaviour. By implementing end-to-end and runtime performance, observability supports the management and operation of microservices architectures. Observability also focuses on learning new advancements within the IT infrastructure to prevent extended outages, as well as performing a “root cause analysis” during outages (Usman et al., 2022). These principles ensure a robust, flexible, and scalable architectures.

11. Ensuring High Availability and Fault Tolerance

To ensure high availability and fault tolerance, Droppa could include several key strategies. Firstly, load balancing could be performed across instances. Load balancing helps distributing workloads across multiple computing resources as well as distributing service requests from clients to multiple servers that offer several services (Autili *et al*., 2019). Secondly, redundancy and failover mechanisms can be implemented. Automated failover mechanisms redirect traffic to redundant service instances in events of failure. Through multiple instances of critical services, redundancy achieves smooth transitions and uninterrupted service delivery during outages. All of this ensures high availability and fault tolerance (Ibrahim & Heart, 2020). Lastly, health checks and self-repair systems can be implemented. Self-repair mechanisms focus on detection and recovery. They monitor microservice health and recover them from failures without needing to involve human intervention (Tadi, 2022). Implementing these key strategies during production would ensure continuous service delivery and minimise system failure impacts

12. Conclusion

Droppa can achieve its goals of scalability, flexibility, and global service delivery, by transitioning to a microservices architecture. Droppa can enhance their systems to become future-proof and enhance performance and innovation by implementing key principles like SRP, decentralised data management, and DevOps integration.

Deliverable 2

Question 2.1

When establishing a solid foundation for transitioning to a microservices architecture, FreshMart should consider the following five key factors:

Domain-Driven Design (DDD): FreshMart should clearly define its business domains and align their microservices according to their domains. These can include inventory, delivery, orders, payments, etc. This approach will ensure that each microservice of FreshMart will have a focused responsibility, ultimately enhancing maintainability and reducing complexity. Independent domains will allow specific services to scale based on demand (Hippchen *et al.*, 2017). Domain isolation will help prevent cascading failures, enhancing reliability. Performance will also be enhanced once each service is optimised for their own workload.

Decoupled and Independent Services: Designing loosely coupled services will ensure that updates, deployments, and failures in a service will not affect other services. FreshMart should avoid tight integration between services and integrate services loosely instead. Teams will be able to scale individual services optimally based on usage, like online checkout compared to product catalogue. Implementing decoupled and independent services will help prevent one service to bring down the entire platform of FreshMart, complementing consistent reliability (Alexandre *et al.*, 2020). It would also enhance performance once services are optimised and tuned.

API Gateway and Service Communication Strategy: FreshMart can implement an API gateway to mangage client requests, routing, load balancing, and authentication more efficiently (Gadge, 2018). It would be helpful for FreshMart to use efficient communication methods like gRPC, REST, and messaging queues. Scalability will be complemented by API gateways by handling load balancing and rate limiting for large amounts of online traffic. API gateways would enhance reliability by enabling circuit breakers and fallback logic. Lastly, through optimised routing and caching, API gateways will reduce latency, increasing performance for FreshMart.

Robust DevOps and CI/CD Pipeline: For FreshMart to manage complexity and multiple microservices more effectively, they would need to automate build, test, and deployment pipelines. Containerisation platforms like Kubernetes and Docker would also be required as well as infrastructure to manage complexity and multiple microservices more effectively, as well as reduce operational overhead (Bhatia, 2024). Implementing DevOps and CI/CD pipelines would firstly, enable services to be rapidly rolled out across multiple environments, complimenting high scalability. Secondly, it would enhance reliability by reducing human error through automated testing and deployment. Lastly, through faster iterations and faster resolution of performance issues, performance would be enhanced significantly.

Observability, Monitoring, and Logging: FreshMart should implement distributed tracing, real-time monitoring, and centralised logging to troubleshoot its services more effectively. Monitoring and logging tools can include ELK stack and Prometheus with Grafana (Usman, 2022). Using these tools will also help FreshMart understand their services better. These tools will help identify bottlenecks for FreshMart as their systems grow. With the help of proactive failure detection, these tools will also enhance reliability for FreshMart's systems. Their performance will also be positively impacted by implementing fine-grained visibility into their system behaviour for speed tuning services.

Question 2.2

Decentralised data management improves scalability and performance through independent data stores, optimised access, and reduced contention. Each microservice in FreshMart will manage its independent database. This will help enable services to be scaled independently based on demand. This will also help reduce cross-service dependencies as well as reduce the impact of failures (Akerele et al., 2024). Optimised access can be achieved when services are able to use data models and storage technologies best suited to their appropriate workload. This will improve transaction performance and query speed (Hassan *et al*., 2022). Decentralised data management helps prevent database bottlenecks from occurring. This will allow multiple services to perform read and write operations concurrently without affecting each other. This will ultimately reduce contention within FreshMart's microservices.

Decentralised data management has a major role in fault isolation. It isolates services, enables faster recovery, and enhances resilience of microservices. Because decentralised data management isolates services to access their own data, failures in one service's database will not be able to impact other services (Tadi, 2022). Due to services being isolated, teams can effectively isolate and resolve issues in single data stores without needing to shut down FreshMart's entire system. This will speed up the recovery process significantly (John, 2023). Decentralised data management will also limit the scope of data corruption or loss. This will improve the stability and reliability of the whole FreshMart system, ultimately enhancing its resilience.

Question 2.3

CI/CD practices will play a big role in the smooth deployment of new features and updates for freshMart's microservices. CI/CD pipelines will allow FreshMart build, test, and deploy microservices updates automatically. Minimal manual intervention will be needed with CI/CD pipelines and deployment errors and downtime will be reduced. CI/CD will speed up release cycles. FreshMart's teams will be able to push new features and updates independently and frequently. This will support agile development and faster responses to customer needs. CI/CD will also introduce improved rollback capabilities (Dakić, 2024). When issues arise after deployment, CI/CD tools will revert deployments to its previous stable versions, minimising disruption.

CI/CD practices will increase system performance and scalability through optimised resource usage, consistent configuration, and reduced downtime. CI/CD processes will allow FreshMart's systems to automate performance testing and containerisation. This will ensure services are tuned for efficiency before they reach production. Scalability will be enhanced through consistency (Baladari, 2021). CI/CD processes will ensure consistent environments and configurations are implemented across all deployments, supporting stable horizontal scaling as traffic increases (Rahman, 2023). Lastly, canary deployments from CI/CD pipelines will reduce risk and allow gradual rollouts. This will reduce downtime, maintain system availability, and maintain performance under load.

Question 2.4

There are several reasons why service monitoring and observability is important for maintaining high availability and performance in FreshMart's microservices. Firstly, real-time health monitoring services help to detect anomalies and performance degradation quickly and efficiently (Nobre *et al*., 2023). Secondly, these monitoring services enables end-to-end visibility. They provide insight into service interactions, allowing teams to trace requests across services and detect delays or failures (Faseeha *et al*., 2025). Lastly, these monitoring services include performance metrics. Monitoring tools like Prometheus and Grafna help analyse response times, throughput, and error rates. This ensures that FreshMart's system can scale as necessary and run efficiently.

FreshMart's service monitoring and observability practices contribute to proactive issue resolution through early detection, root cause analysis, and continuous improvement. With early detection, alerts are triggered by unusual patterns, like slow response times and traffic spikes (Naikade, 2020). This allows teams to resolve issues before they escalate. Distributed tracing and centralised logging can help FreshMart to perform root cause analysis, meaning that they can identify the source of problems and reduce solution time (Usman *et al*., 2022). Observability data guide performance tuning, capacity planning, and architectural decisions can help FreshMart to gain insights and continuously improve for long-term reliability.

Question 2.5

FreshMart can adopt five key methods and technologies to ensure reliable and high-performance inter-service communication, especially during peak shopping times. They include the following:

Asynchronous Messaging with Message Brokers: Asynchronous messaging tools like Apache Kafka and RabbitMQ, will provide decoupled services for FreshMart, by allowing them to communicate through queues or streams instead of direct calls (Schumeth, 2024). This will improve reliability and enable buffering during peak traffic spikes.

Service Mesh Implementation: Service Meshes like Istio and Linkerd provide built-in traffic control, retries, and observability for service-to service communication (Sidharth, 2019). This will ensure resilient connections and load balancing for FreshMart without needing to change their application code.

API Gateways: API Gateways like AWS, NGINX, and Kong act as single-entry points for requests. They will handle routing, rate-limiting, and authorisation for FreshMart (Xu *et al*., 2019). They will also prevent backend services from being overwhelmed.

Circuit Breaker Pattern: Circuit breaker libraries like Resilience4j and legacy Hystrix can help FreshMart prevent cascading failures. This can be done by detecting service failures and automatically halting requests to it until full recovery (Montesi & Weber, 2016). It will improve FreshMart's overall system stability.

Load Balancing and Auto-Scaling: Auto-scaling technologies like Kubernetes Horizontal Pod Autoscaler, and load balancing technologies like AWS Elastic Load Balancer, will help Freshmart distribute traffic evenly across service instances (Serracanta *et al*., 2025). They will also add more instances dynamically when demand increases. This will maintain performance during high-traffic periods.

These methods together will help FreshMart maintain smooth service interaction, enhance fault tolerance, and reduce latency, even under pressure.

Question 2.6

There are several strategies that Freshmart can implement to ensure data consistency across their microservices. These strategies include the following:

Event-Driven Architecture with Event Sourcing: Events like OrderPlaced and InventoryUpdated can be used as sources of truth. Using an event-driven architecture with event sourcing can allow services to stay updated asynchronously and maintain a reliable history of every change in FreshMart's system (Charankar & Pandiya, 2024). This will enhance eventual consistency.

SAGA Pattern: Using a SAGA pattern can help FreshMart manage distributed transactions through a series of local transactions by either orchestration or choreography. This will ensure that if one part of the business process fails, compensating actions can be executed (Daraghmi *et al*., 2022). For example, if a payment process of FreshMart fails, the order can be canceled.

Idempotent Operations: APIs and services can be designed for FreshMart to handle repeated requests without having to change the result (Yadav & Mantri, 2024). This will help them prevent data duplication or retry corruption, especially in weak network conditions.

Data Duplication with Sync Mechanisms: Data like product details can be duplicated across services and can be kept in sync through scheduled jobs or real-time updates. This will help services to work independently for FreshMart while accessing updated shared data.

Consistent Data Contracts and API Versioning: FreshMart should define clear data structures and versions for inter-service communication. This will help them ensure backward compatibility during updates as well has prevent services from miscommunicating (Lercher *et al*., 2024).

Implementing these strategies will help FreshMart achieve eventual consistency. These strategies will also minimise conflicts, maintain reliability, and support a scalable microservices environment.

Deliverable 3

Question 3.1

SmartCargo can minimise inter-service dependencies and maintain consistency in core operations like real-time route optimisation and shipment tracking, by carefully defining service boundaries using three key factors. They include the following:

Business Domain Decomposition: Domain-Driven Design (DDD) should be implemented into SmartCargo's system. Their services should be designed around distinct business capabilties or domains like route optimisation, customer notifications, and shipment tracking. Implementing DDD would encourage autonomy for SmartCargo as well as allow each microservice to handle its own data and logic (Myllynen *et al*., 2023). This will enhance maintainability and loose coupling.

Bounded Contexts with Clear Data Ownership: Each microservice in SmartCargo's system should manage its own data. Each microservice should only expose the necessary data to other services through API's or events (Butzin *et al*., 2016). This would promote loose coupling for SmartCargo and reduce the risk of data inconsistency, especially in time-sensitive operations like real-time tracking.

Asynchronous Communication Using Events: Instead of using synchronous calls, SmartCargo should use event-driven communication like PackageScanned and RouteUpdated, instead wherever possible. This would help SmartCargo minimise inter-service blocking and improve the resilience of their system (Tadi, 2022). Moreover, this will allow critical services to operate independently, even other services are delayed or fail.

Implementing these key factors will provide resilient, consistent, and scalable microservices for SmartCargo. This will ensure that their critical logistics operations will remain synchronsised and uninterrupted.

Question 3.2

SmartCargo can implement two advanced operational strategies to ensure fault tolerance and high availability in their microservices during peak global shipping periods. They can use circuit breakers along with retry mechanisms and implement multi-region deployment with load balancing.

Circuit breakers will help SmartCargo detect and isolate service failures and prevent prevent cascading failures for their entire system. SmartCargo can also combine circuit breakers with automated retries and exponential backoff to handle temporary issues easier and more efficiently. This will ensure that service failures will be contained effectively (Chandramouli, 2019). Moreover, it will improve their system's resilience and help them avoid overwhelming downstream services during peak loads.

SmartCargo can deploy their services across different geographic locations and use global load balancers to distribute traffic intelligently. They can enable traffic rerouting during regional outages to enhance their fault tolerance (bin Abdullah, 2023). Their latency and performance will be improved for users globally.

Question 3.3

SmartCargo can adopt five monitoring strategies and metrics to proactively detect and resolve issues across their microservices ecosystem. These strategies and metrics include the following:

Service Latency Monitoring: SmartCargo can track the response times of APIs and microservices. Spikes in latency will likely indicate bottlenecks, overloaded services, or network issues (Khan, 2020). A good service latency monitoring tool for SmartCargo will be Prometheus with Grafana dashboards.

Error Rate Tracking: SmartCargo should use error rate tracking tools like ELK Stack (Elasticsearch, Logstash, Kibana) or Sentry to monitor the rate of HTTP 4xx and 5xxx errors (Lee, 2024). Increasing error rates will signal possible bugs, service crashes, or even bad employments.

Health and Update Checks: SmartCargo can implement liveness and readiness probes for each of their services. This will enable automated orchestration platforms like Kubernetes to restart unhealthy containers or reroute traffic. Kubernetes-native health checks can be used with PagerDuty for alerts (Bellin, 2024).

Distributed Tracing: SmartCargo should trace requests across multiple services using tools like OpenTelemetry with Jaeger or Zipkin. This will help SmartCargo to monitor their journey end-to-end. Moreover, this will help them pinpoint the locations of failures or delays in complex request paths (Gomes *et al*., 2024).

Resource Utilisation Monitoring: SmartCargo should track their CPU, memory, disk Input/Output (I/O), and network usage for each of their services, using monitoring tools like Prometheus with Node Explorer and Grafana alerts (Santos, 2024). This will ensure that SmartCargo can allocate their resources efficiently and highlight their scaling needs or memory leaks.

SmartCargo can combine these five key strategies and metrics, with automated alerting and visual dashboards, to maintain performance proactively, detect anomalies early, and resolve issues before they escalate.

Bibliography

AIT SAID, M., EZZATI, A., MIHI, S. and BELOUADDANE, L., 2024. Microservices adoption: An industrial inquiry into factors influencing decisions and implementation strategies. *International Journal of Computing and Digital Systems*, *15*(1), pp.1417-1432.

Akerele, J.I., Uzoka, A., Ojukwu, P.U. and Olamijuwon, O.J., 2024. Improving healthcare application scalability through microservices architecture in the cloud. *International Journal of Scientific Research Updates*, *8*(02), pp.100-109.

Alexandre, M., da Silva, P., Cesário, V. and Araújo, C., 2020. A decoupled health software architecture using microservices and OpenEHR archetypes. *International Journal of Computer Applications*, *975*, p.8887.

Autili, M., Perucci, A. and De Lauretis, L., 2019. A hybrid approach to microservices load balancing. In *Microservices: Science and Engineering* (pp. 249-269). Cham: Springer International Publishing.

Baboi, M., Iftene, A. and Gîfu, D., 2019. Dynamic microservices to create scalable and fault tolerance architecture. *Procedia Computer Science*, *159*, pp.1035-1044.

Badampudi, D., Usman, M. and Chen, X., 2025. Large scale reuse of microservices using CI/CD and InnerSource practices-a case study. *Empirical Software Engineering*, *30*(2), pp.1-46.

Baladari, V., 2021. Monolith to Microservices: Challenges, Best Practices, and Future Perspectives. *European Journal of Advances in Engineering and Technology*, *8*(8), pp.123-128.

Bellin, L., 2024. Monitoring at high scale for very heterogeneous distributed systems.

Bhatia, S., 2024. The Role of Microservices Architecture in DevOps.

bin Abdullah, A., 2023. Refining Distributed System Efficiency with Microservices: Advanced Strategies for Enhancing Performance, Scalability, and Resilience in Complex Architectural Environments.

Blinowski, G., Ojdowska, A. and Przybyłek, A., 2022. Monolithic vs. microservice architecture: A performance and scalability evaluation. *IEEE access*, *10*, pp.20357-20374.

Butzin, B., Golatowski, F. and Timmermann, D., 2016, September. Microservices approach for the internet of things. In *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)* (pp. 1-6). IEEE.

Chandramouli, R., 2019. Microservices-based application systems. *NIST Special Publication*, *800*(204), pp.800-204.

Charankar, N. and Pandiya, D.K., 2024. Enhancing Efficiency and Scalability in Microservices Via Event Sourcing. *INTERNATIONAL JOURNAL OF ENGINEERING RESEARCH & TECHNOLOGY (IJERT) Volume*, *13*.

Dakić, P., 2024. Software compliance in various industries using ci/cd, dynamic microservices, and containers. *Open Computer Science*, *14*(1), p.20240013.

Daraghmi, E., Zhang, C.P. and Yuan, S.M., 2022. Enhancing saga pattern for distributed transactions within a microservices architecture. *Applied Sciences*, *12*(12), p.6242.

dos Santos Silva, A.F., 2024. *Detection of transaction consistency problems in microservices* (Doctoral dissertation, UNIVERSIDADE DE LISBOA).

dos Santos, L.M.D.S., 2024. Data distribution and access in a microservices architecture.

El Khalyly, B., Belangour, A., Banane, M. and Erraissi, A., 2020. A comparative study of microservices-based IoT platforms. *International Journal of Advanced Computer Science and Applications (IJACSA)*, *11*(7), pp.389-398.

Faseeha, U., Syed, H.J., Samad, F., Zehra, S. and Ahmed, H., 2025. Observability in Microservices: An In-Depth Exploration of Frameworks, Challenges, and Deployment Paradigms. *IEEE Access*.

Fritzsch, J., Bogner, J., Haug, M., Franco da Silva, A.C., Rubner, C., Saft, M., Sauer, H. and Wagner, S., 2023. Adopting microservices and DevOps in the cyber-physical systems domain: A rapid review and case study. *Software: Practice and Experience*, *53*(3), pp.790-810.

Gadge, S. and Kotwani, V., 2018. Microservice architecture: API gateway considerations. *GlobalLogic Organisations, Aug-2017*, *11*.

George, J., 2025. DECODING HR DATA SILOS: A MICROSERVICES APPROACH TO UNIFIED TALENT ANALYTICS.

Gomes, F., Gabriel, V., Rego, P., Trinta, F. and de Souza, J., 2024, February. Impact of OpenTelemetry Configuration on Observability and Telemetry Storage Cost of Microservices-Based Applications. In *International Workshop on ADVANCEs in ICT Infrastructures and Services*.

Haindl, P., Kochberger, P. and Sveggen, M., 2024. A systematic literature review of inter-service security threats and mitigation strategies in microservice architectures. *IEEE Access*.

Hassan, S., Bahsoon, R. and Buyya, R., 2022. Systematic scalability analysis for microservices granularity adaptation design decisions. *Software: Practice and Experience*, *52*(6), pp.1378-1401.

Hippchen, B., Giessler, P., Steinegger, R., Schneider, M. and Abeck, S., 2017. Designing microservice-based applications by using a domain-driven design approach. *International Journal on Advances in Software*, *10*(3&4), pp.432-445.

Ianculescu, M. and Alexandru, A., 2020. Microservices–A catalyzer for better managing healthcare data empowerment. *Studies in Informatics and Control*, *29*(2), pp.231-242.

Ibrahim, A. and Heart, S., 2020. Self-Healing Techniques in API and Microservices Frameworks.

John, T., 2023. Enhancing Microservices Architecture with AI-Based Monitoring and Self-Healing Systems.

Kansal, S. and Balasubramaniam, V.S., 2024. Microservices Architecture in Large-Scale Distributed Systems: Performance and Efficiency Gains. *Journal of Quantum Science and Technology (JQST)*, *1*(4), pp.633-663.

Khan, F., 2020. Microservices metrics visualization.

Lee, I.T.A., Zhang, Z., Parwal, A. and Chabbi, M., 2024. The Tale of Errors in Microservices. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, *8*(3), pp.1-36.

Lercher, A., Glock, J., Macho, C. and Pinzger, M., 2024. Microservice api evolution in practice: a study on strategies and challenges. *Journal of Systems and Software*, *215*, p.112110.

Montesi, F. and Weber, J., 2016. Circuit breakers, discovery, and API gateways in microservices. *arXiv preprint arXiv:1609.05830*.

Myllynen, T., Kamau, E., Mustapha, S.D., Babatunde, G.O. and Adeleye, A., 2023. Developing a Conceptual Model for Cross-Domain Microservices Using Event-Driven and Domain-Driven Design.

Naikade, P.P., 2020. *Automated Anomaly Detection and Localization System for a Microservices Based Cloud System* (Master's thesis, The University of Western Ontario (Canada)).

Nobre, J., Pires, E.S. and Reis, A., 2023. Anomaly detection in microservice-based systems. *Applied Sciences*, *13*(13), p.7891.

Rahman, N.H.B.M., 2023. Exploring The Role Of Continuous Integration And Continuous Deployment (CI/CD) In Enhancing Automation In Modern Software Development: A Study Of Patterns. *Tools, And Outcomes*.

Rudrabhatla, C.K., 2020. Impacts of decomposition techniques on performance and latency of microservices. *International Journal of Advanced Computer Science and Applications*, *11*(8).

Schumeth, J., 2024. *Evaluation of Message Brokers for interprocess communication in a Microservice Architecture* (Doctoral dissertation, University of Applied Sciences).

Serracanta, B., Lukács, A., Rodriguez-Natal, A., Cabellos, A. and Rétvári, G., 2025. On the Stability of the Kubernetes Horizontal Autoscaler Control Loop. *IEEE Access*.

Sidharth, S., 2019. Enhancing Security of Cloud-Native Microservices with Service Mesh Technologies.

Suleiman, N. and Murtaza, Y., 2024. Scaling microservices for enterprise applications: comprehensive strategies for achieving high availability, performance optimization, resilience, and seamless integration in large-scale distributed systems and complex cloud environments. *Applied Research in Artificial Intelligence and Cloud Computing*, *7*(6), pp.46-82.

Tadi, S.R.C.C.T., 2022. Architecting Resilient Cloud-Native APIs: Autonomous Fault Recovery in Event-Driven Microservices Ecosystems. *Journal of Scientific and Engineering Research*, *9*(3), pp.293-305.

Usman, M., Ferlin, S., Brunstrom, A. and Taheri, J., 2022. A survey on observability of distributed edge & container-based microservices. *IEEE Access*, *10*, pp.86904-86919.

Velepucha, V. and Flores, P., 2023. A survey on microservices architecture: Principles, patterns and migration challenges. *IEEE access*, *11*, pp.88339-88358.

Venčkauskas, A., Kukta, D., Grigaliūnas, Š. and Brūzgienė, R., 2023. Enhancing microservices security with token-based access control method. *Sensors*, *23*(6), p.3363.

Weerasinghe, S. and Perera, I., 2023. Optimized strategy for inter-service communication in microservices. *International Journal of Advanced Computer Science and Applications*, *14*(2).

Xu, R., Jin, W. and Kim, D., 2019. Microservice security agent based on API gateway in edge computing. *Sensors*, *19*(22), p.4905.

Yadav, P.S. and Mantri, A., Mitigating Duplicate Message Processing in Microservice Architectures: An Idempotent Consumer Approach. *J Artif Intell Mach Learn & Data Sci 2024*, *1*(1), pp.887-891.

Zdun, U., Stocker, M., Zimmermann, O., Pautasso, C. and Lübke, D., 2018. Guiding architectural decision making on quality aspects in microservice apis. In *Service-Oriented Computing: 16th International Conference, ICSOC 2018, Hangzhou, China, November 12-15, 2018, Proceedings 16* (pp. 73-89). Springer International Publishing.