CE3.8

Cohort 4 Grp 3

Raymond Ang, Lim WJ, Rudyn, Ong Joseph, Sagar

There are lot of complexity when we are using or moving to microservices architecture.

Resolve these challenges when we are moving to microservices.

Challenges	Problems	Focus	Strategies
1. Need for Testing and Monitoring	· More services to monitor. · Developed using different programming languages. · Interdependencies between services can all have cascading downstream effects (any issue in one area can lead to a problem in another microservice elsewhere).	Reliability and performance of the system through comprehensive testing, monitoring, and debugging practices.	1. Implement Comprehensive Testing Strategies 2. Continuous Monitoring and Observability 3. Automated Testing 4. Chaos Engineering 5. Performance Testing

2. Debugging Issues	· Each service has its own set of logs resulting in large distributed unstructured data to debug. · Work backward through status codes and vague error messages generated across the network.		1. Centralized Logging and Distributed Tracing 2. Debugging Tools and Techniques 3. Code Reviews and Pair Programming 4. Error Handling and Logging
3. Compromised Security	· Data is distributed in a microservices-based framework. · Maintaining the confidentiality and integrity of user data is difficult. · Setting up access controls and administering secured authentication to individual services. · Increased attack surface vulnerability. · Loss of control and visibility of application components. · Extremely difficult to test for vulnerabilities since each microservice communicates with others through different infrastructure layers.	Security and operational aspects of the system	1. Implement Security Best Practices 2. Authentication and Authorisation Mechanisms - (local device) secret manager for microservice 3. Encryption and Data Protection 4. Regular Security Audits and Penetration Testing 5. Secure protocol - HTTPS, TLS, SSH

4. Increased Operational Complexity	· Require serious effort to ensure that the whole application is resilient, and failovers can be avoided. · Requires sophisticated tooling for automated provisioning in a highly secure and resilient manner. · Microservice-based application, when one component fails, it can cascade the effect to the entire system.		1. Automation of Deployment and Operations 2. Infrastructure as Code (IaC) 3. Monitoring and Alerting Systems 4. Incident Management Procedures
5. Inter-Service Communication Breakdown	· Microservices that rely on each other will need to communicate, which is done using well-defined APIs without sharing the same technology stacks, libraries, or frameworks. · Poor configuration can easily lead to increased latency. · Managing communication between microservices can be hard without using automation and advanced methodologies such as Agile. · Production would require some form of service mesh capabilities to scale.	Managing communication and network-related challenges within a distributed system	1. Implement Robust Communication Protocols and Standards 2. Circuit Breaker Patterns 3. Retry Strategies 4. Message Queues and Pub/Sub Systems 5. For API call, Enforcing rate limits to improve latency for the rest.
6. Network Management	· Less fault tolerance and needs more load balancing. · Distributed monoliths with a slew of drawbacks of a monolith application		1. Implement Network Segmentation and Security Policies - Setup Security Groups

	○ "Chatty" applications that depend upon various services to satisfy a request or process. Need all dependent microservices to be available and operational at the same time		2. Traffic Management and Load Balancing 3. Service Mesh Implementation - Service to service communication (security) 4. Network Monitoring and Analysis
7. Maintenance of Microservices	· Due to different languages and technological bases used for each of the service, during maintenance, more resources will be required to maintain all the tools and technologies.	Ensuring the ongoing maintenance and support of the microservices architecture	1. Versioning and Dependency Management 2. Automated Testing and Continuous Integration 3. Regular tidying the code (refactoring) and Code Reviews 4. Health Checks and Self-Healing Mechanisms 5. Domain-Driven Design (DDD)
8. Requires Team Expertise	· Ill-prepared design and development teams. · Requires experience working with distributed architectures.		1. Continuous Learning and Skill Development 2. Training and Workshops

	· Collaborate with other microservice teams to cover the whole cycle.		3. Collaboration and Knowledge Sharing 4. Hiring and Onboarding
9. Achieving Data Consistency	· Data redundancy or duplication due to microservice having own database. · Difficult to achieve data consistency due to o network latency, o distributed transactions, and o eventual consistency models.	Data across different microservices remains accurate and coherent despite the distributed nature of the system	1. Use the Right Database 2. Transactional Updates 3. Event Sourcing 4. CQRS (Command Query Responsibility Segregation) 5. Saga Pattern - Data consistency across microservices in distributed transaction scenarios. A saga is a sequence of transactions that updates each service and publishes a message or event to trigger the next transaction step. 6. Idempotent Operations 7. Eventual Consistency 8. Consistency Boundaries

			9. Data Validation 10. Monitoring and Auditing 11. Documentation and Communication

10. Overcoming Design Complexity	· Service boundaries, · inter-service communication, · data management, and · system orchestration.	Simplify the design and structure of the system while ensuring that it remains scalable, maintainable, and adaptable to changes.	1. Clear Business Goals and Domain Boundaries 2. Decompose Monolithic Applications 3. Use Domain-Driven Design (DDD) 4. Single Responsibility Principle (SRP) 5. API Contracts and Documentation