**Name**: Tural Mehtiyev

**Project Title**: Scalability analysis of microservice architecture on a particular software application

1. **What are you going to do?**

I'm going to conduct a detailed analysis of a microservice architecture system to assess its scalability on ~~a particular software application~~ to do list application (with 3 microsevices). Scalability in this context means the ability of the system to handle increased load without a detrimental impact on performance, or the capacity to expand to accommodate growth. This includes an examination of how well the system can handle increasing load, the strategies it uses to scale, and areas where scalability can be improved.

Below is the initial list of the scalability attributes to be used in the project:

* Performance: This refers to the ability of the architecture to maintain or improve its functioning as the load increases.
* Load Scalability: This measures how well the architecture can handle an increase in concurrent tasks or processes, including the number of user requests or transactions it can manage effectively.
* Resource Utilization: It's essential to understand how efficiently the architecture uses available resources (CPU, memory, storage, network bandwidth, etc.) when scaling up or out.
* Capacity: This refers to the maximum load that the system can handle under specific conditions. This could involve the number of microservices it can manage, the number of concurrent users it can support, the volume of data it can process, etc.
* ~~Fault Tolerance and Resilience: Microservices architectures are designed to continue functioning even when individual services fail. This attribute refers to how well the architecture handles such failures and whether it can still scale under these conditions.~~

The following are key metrics to consider when assessing the scalability of a software architecture.

* Response Time: This measures how quickly the system responds to a user's request. In a scalability context, we want to observe how this metric changes as load increases. A scalable system should maintain relatively stable response times even under increased load.
* Throughput: This measures the number of requests the system can handle per unit of time, typically expressed as Requests Per Second (RPS) or Transactions Per Second (TPS). In terms of scalability, we want to see this number increase proportionately to increases in load or resources.
* Resource Usage: This metric monitors the consumption of resources like CPU, memory, disk I/O, and network I/O at different load levels. Efficient resource usage is a key element of scalability, as it means that adding resources will result in proportional increases in capacity.
* Error Rates: This represents the percentage of requests that result in errors. High or increasing error rates under load can indicate scalability problems, as it suggests that the system is struggling to handle the increased demand.
* ~~Latency: This refers to the delay before a transfer of data begins following an instruction for its transfer. In distributed systems like microservices, minimizing latency is crucial for maintaining performance as the system scales.~~
* ~~System Utilization: This measures how well the system's capacity is being used, often broken down into CPU utilization, memory utilization, etc. A system that can maintain high utilization rates as it scales is using its resources efficiently.~~
* ~~Service Time: This is the time taken by the system to complete a request once it starts processing it. It's a measure of the system's internal efficiency and can provide insights into potential bottlenecks.~~
* ~~Scalability Ratio: This is the ratio of the throughput achieved by adding more resources to the system to the throughput of a single resource. It can provide a direct measure of the system's scalability.~~
* ~~Queue Length: This measures the number of incoming requests waiting to be processed. A consistently long queue could mean that the system is unable to keep up with the incoming load, indicating scalability issues.~~
* ~~Availability: This refers to the ability of the system to remain available and operational as it scales. High availability is crucial for ensuring a consistent user experience, especially in a microservices architecture where the failure of a single service can impact the entire system.~~

1. **How is it done today? Current Limitations?**

Today, scalability analysis is usually done through a mix of load testing, code reviews, and architectural analysis. Current limitations include the time and expertise required to set up and interpret load tests, difficulty predicting future loads, and the challenge of identifying bottlenecks in complex systems. In addition, microservices, due to their distributed nature, introduce additional complexity in managing and coordinating different services, data consistency, network latency, and fault tolerance, which can also affect scalability.

1. **What is your idea to do something better?**

My plan is to develop a comprehensive methodology that combines empirical testing with theoretical analysis. This includes setting up automated load tests that simulate different types of load increase, performing a detailed review of the system's code and architecture to identify potential bottlenecks. The results of this analysis will be compiled into a detailed report that includes both specific, actionable recommendations for improving scalability and general insights that can be applied to other microservice architectures.

1. **Who will benefit from your work? Why?**

Both developers and organizations can benefit from this work. Developers can use the insights gained from this analysis to design more scalable microservice architectures, while organizations can benefit from the improved capacity planning and potentially lower infrastructure costs that result from better scalability. In addition, better scalability can improve user experience by ensuring that the system remains responsive even under high load.

1. **What risks do you anticipate?**

* Representativeness of the chosen system: There's a risk that the chosen system might not be representative of other microservice architectures, limiting the applicability or generalizability of the findings. Different systems can have unique characteristics that impact scalability in ways not observed in the chosen system.
* Incomplete identification of scalability issues: Load tests and analysis might not reveal all potential scalability issues, particularly those that only manifest under very specific or extreme conditions. This might lead to an overly optimistic assessment of the system's scalability.
* ~~Quality and availability of documentation: Finding a system with comprehensive, accurate, and up-to-date documentation can be challenging. If the system's documentation is incomplete, outdated, or not well-structured, this could complicate the analysis process and potentially lead to misunderstandings about the system's design and behavior.~~
* Technical complexities and unknowns: Given the complex nature of microservices-based architectures, there might be technical challenges and unforeseen issues that arise during the analysis. This could impact the project timeline or the depth of analysis possible within the project scope.

1. **Out of pocket costs? Complete within 11 weeks?**

There may be minimal out-of-pocket costs for cloud computing resources if load tests are performed in a cloud environment. With good project planning and a clear focus, the project can likely be completed within 11 weeks.

1. **Midterm results?**

~~By the midterm point, I expect to have completed the setup of the load tests, run initial tests, and started the code and architectural analysis. I should have some preliminary findings and be able to identify potential areas for improvement.~~

1. **Final Demonstration?**

~~The final demonstration will include a presentation of the findings and recommendations from the scalability analysis. I will also demonstrate the load testing setup and discuss how the tests were used to assess scalability. If possible, I may also show before-and-after performance results to illustrate the impact of any recommended changes that were implemented.~~

---------------------------------------------------------------------------------------

1. **Potential Research Questions:**
2. How does the choice of microservice architecture impact the performance scalability of an application?
   1. **Communication Patterns**: How do different communication patterns (e.g., synchronous request-response, asynchronous messaging) employed in microservice architectures impact performance scalability?
   2. **Service Discovery and Registration**: What are the performance implications of different service discovery and registration mechanisms (e.g., service registries, service meshes) in microservice architectures?
   3. **Data Management**: How does the choice of data management strategies (e.g., shared databases, database per service, event sourcing) influence performance scalability in microservice architectures?
   4. **Fault Isolation**: How does the design of fault isolation mechanisms (e.g., circuit breakers, bulkheads) in microservice architectures affect performance scalability under failure conditions?
   5. **Deployment Flexibility**: To what extent does the flexibility to independently deploy and scale individual microservices impact the overall performance scalability of the application?
   6. **API Gateway**: How does the implementation of an API gateway affect the performance scalability of a microservice architecture by handling requests, load balancing, or caching?
3. What are the most effective load balancing strategies for achieving performance scalability in microservice architectures?
   1. **Round Robin Load Balancing**: This strategy evenly distributes incoming requests across available service instances in a cyclic manner. Each instance receives an equal number of requests, ensuring a balanced distribution of the workload. Round robin load balancing is simple to implement and works well when services have similar processing capabilities.
   2. **Weighted Round Robin Load Balancing**: Weighted round robin extends the basic round robin strategy by assigning different weights to each service instance. The weights represent the relative processing capacity of the instances. Higher-weighted instances receive more requests, allowing you to allocate resources based on their capabilities.
   3. **Least Connection Load Balancing**: This strategy directs new requests to the service instance with the fewest active connections at the time. It helps distribute the workload proportionally to the current load on each instance, ensuring that instances with fewer connections receive more requests. This approach can be useful when the processing times of requests vary across instances.
   4. **IP Hash Load Balancing**: With IP hash load balancing, the load balancer assigns requests to specific instances based on the client's IP address. This strategy ensures that requests from the same client are consistently directed to the same service instance. It can be beneficial in scenarios where maintaining session affinity or preserving state is important.
   5. **Random Load Balancing**: This approach randomly selects a service instance for each incoming request. While it may not provide as even a distribution as other strategies, it can still achieve a reasonable workload distribution. Random load balancing is simple to implement and can be effective when service instances have similar capabilities.
   6. **Dynamic Load Balancing**: Dynamic load balancing strategies involve continuously monitoring the performance and health of service instances and adjusting the load distribution accordingly. This can include evaluating factors such as response times, resource utilization, error rates, or service availability. Dynamic load balancing allows for intelligent decision-making based on real-time conditions, ensuring efficient resource allocation.
   7. **Content-Based Load Balancing**: Content-based load balancing, also known as request routing or intelligent routing, involves examining the characteristics of incoming requests (such as URL, headers, or payload) to determine the most suitable service instance to handle the request. This strategy can be effective when certain instances are better suited to handle specific types of requests.
4. How does the granularity of microservices affect performance scalability? (Difficult to test)
5. What are the trade-offs between vertical and horizontal scaling in achieving performance scalability in microservice architectures?
6. What techniques can be employed to optimize resource utilization and improve performance scalability in microservice applications?
7. How do different caching mechanisms impact the performance scalability of microservices?
8. What are the challenges and best practices for achieving performance scalability in containerized microservice environments? (Theoretical)
9. How does the use of service mesh technologies influence the performance scalability of microservice architectures?
10. What are the performance scalability limitations and bottlenecks in microservice architectures, and how can they be mitigated? (Theoretical)
11. How does the selection of programming languages and frameworks impact the performance scalability of microservices? (General)

**Roadmap:**

1. **Definition the Project Concept:**

* Identify the purpose and goals of the microservice architecture.
* Determine the key functionalities and services that the architecture will provide.
* Understand the expected performance requirements, such as response time, throughput, and scalability.

1. **System Design:**

* Define the overall architecture of the microservices, including their relationships and interactions.
* Determine the flow of the process and the sequence of microservice invocations.
* Identify the inputs and outputs of each microservice.
* Decide on the communication protocols, such as REST, messaging queues, or event-driven architectures.
* Determine the data storage requirements and select appropriate databases or data stores.

1. **Identification of Performance Metrics:**

* Determine the performance metrics that you want to measure during the performance test.
* Common performance metrics include response time, throughput, latency, error rates, and resource utilization.
* Align the chosen performance metrics with the project goals and requirements.

1. **Test Environment Setup:**

* Set up the necessary infrastructure, including servers, networking, and databases.
* Install and configure the microservices and any supporting components.
* Ensure that the test environment closely resembles the production environment to ensure accurate results.

1. **Test Scenario Design:**

* Define realistic test scenarios that mimic the expected usage patterns of the microservices.
* Determine the workload, including the number of concurrent users, request rates, and data volumes.
* Design test cases to cover different types of interactions and inputs for the microservices.
* Consider both normal and peak load scenarios to assess performance under various conditions.

1. **Load Generation:**

* Select a suitable load testing tool, such as Apache JMeter, Gatling, or Locust.
* Configure the load testing tool to generate the desired workload based on the defined test scenarios.
* Generate realistic user behavior by incorporating think times, session management, and user profiles.

1. **Performance Test Execution:**

* Execute the performance tests using the defined test scenarios and load profiles.
* Monitor and collect performance metrics during the test execution.
* Gather relevant data on response times, throughput, error rates, and resource utilization.
* Ensure that the tests run for a sufficient duration to capture stable performance measurements.

1. **Analyze and Interpret Results:**

* Analyze the collected performance data and metrics.
* Identify any performance bottlenecks, such as slow microservices, high response times, or resource limitations.
* Interpret the results in the context of the project goals and performance requirements.
* Compare the observed performance against the defined benchmarks or thresholds.

1. **Performance Optimization:**

* Based on the analysis of the results, identify areas for improvement.
* Optimize the microservices or underlying infrastructure to address any performance issues.
* Apply techniques such as caching, database indexing, load balancing, or scaling.
* Repeat the performance test after implementing optimizations to validate the improvements.

1. **Documentation and Reporting:**

* Document the performance test procedures, configurations, and results.
* Create a detailed report summarizing the performance findings, optimizations, and recommendations.
* Share the report with relevant stakeholders and project teams.

**Stacks to be used.**

* Programming Languages and Frameworks
* Python (FastAPI)
* Database
* PostgreSQL
* Container Technologies
* Docker
* Orchestration Engines
* Kubernetes
* Service Discovery
* To be decided (Consul, Etcd, Eureka)
* API Gateways
* To be decided (Kong, Ocelet, KrakenD)
* Event Bus Tools and technologies
* To be decided (Apache Kafka, RabbitMQ)
* Logging Tools
* To be decided (Kibana, Logstash)
* Monitoring Tools
* To be decided (Grafana, Prometheus, cAdvisor)
* Documentation Tools
* Swagger UI
* API Testing Tools
* Postman
* Performance testing tools
* To be decided (Apache JMeter, Gatling, Locust)

**Application Requirements**

* The application must accept a range of HTTP requests (from a fixed set) and deliver a response.
* The application must consist of several microservices, each with a distinct function which stresses the underlying resources to a greater or lesser degree.
* In response to an external HTTP request the application should make internal HTTP requests to the required microservices.
* The application should be scalable on a the microservice level, for example we should be able to scale up one microservice but not the others.
* The application must record all external HTTP requests and log the request response time.
* The application should be simple to configure and deploy in an agile manner.

**Questions raised by the professors before the proposal**

* What attributes or dimensions of a software architecture would you use to assess scalability of an architecture? e.g., for example - functionality, capacity, performance, etc.

Clarification questions: Are not these scalability attributes somehow related? For example, can we talk separately about performance scalability without capacity scalability?

Answer: Performance Scalability

* How would you analyze a software architecture to assess the different attributes of scalability? Which software architecture would you pick to start your analysis and assessment?

Answer: Microservice architecture is picked to start my analysis and assessment. Different metrics (Response Time, throughput, Resource Usage, Error Rates)

* Are there interactions among the measurement of the different attributes according to the key metrics? Typically, we make a linearity assumption, but often find this is not true.

Answer: To be clarified

* You will need to define what you mean by:

a. Software Architecture

b. Scalability

c. Effect of Design for Dynamics

I suggest you think about what attributes you want to consider in assessing the scalability of software architectures.

* I suggest you decide what outcomes/results you want to produce from your project:

1. a set of attributes to assess scalability of a software architecture.

2. a model that proposes some relationships between the attributes

3. an analysis of a particular sw architecture for some application that already exists for which you can obtain good documentation, say the Android OS or some app.

Clarification Question: Can you please provide one example of relationship between attributes?

Answer: What if I choose one attribute (performance) to assess scalability (via Response Time, throughput, Resource Usage) on a simple application (with around 3 services). Instead of finding the relationship between the attributes, I can find the effects of design (for example, effects of communication patterns between microservices on the performance scalability)

**Current Problems of the project**

* 1. Direction of the research
* The topic and the research question are not clear yet.
* The end deliverables are not clear.
* The methodology of the experiment is not clear.
  1. The number of new tech stacks for the project