STUDENT NUMBER : C3WGVD453

STUDENT NAME : Elle Reece Chetty

MODULE CODE: ITEPA3-33

Assignment

1 September 2023

SECTION A

Question 1

1a.

**Concurrency and Parallelism**

Modern corporate programs frequently need to manage numerous activities concurrently and make use of multi-core processors. This is known as concurrency and parallelism. *'Concurrent. Futures'* and '*multiprocessing'* are two Python packages (libraries) that make it simple for programmers to construct parallel and concurrent programming. Additionally, the previously stated *'asyncio'* library offers asynchronous programming capabilities, enabling effective handling of I/O-bound processes. With the help of these advancements, Python can now execute difficult jobs in corporate applications without compromising speed.

**Data Science and Machine Learning Integration**

The ecosystem for Python has expanded to include a wide range of libraries for data science, machine learning, and artificial intelligence. Effective instruments for data manipulation, analysis, and the creation of machine learning models are available thanks to libraries like NumPy, Pandas, Scikit-Learn, and TensorFlow. These skills are essential for contemporary workplace applications that need the integration of machine learning, predictive analytics, and data-driven decision-making.

**Performance Optimization**

Through a number of ways, Python's performance has been substantially enhanced. The advent of the PyPy runtime, which uses Just-In-Time (JIT) compilation to greatly increase execution performance, is one of the most prominent advances. Additionally, improvements like the addition of the asyncio package for asynchronous programming have improved the language's capacity to handle concurrent operations effectively. Additionally, Python's standard interpreter, CPython, has been streamlined over time. Because of these speed enhancements, Python is more suited for business applications that need to handle heavy workloads and real-time processing.

b.

Enterprise IT (Information Technology) is the term used to describe the whole administration of a company's technological infrastructure, systems, and services in order to support a wide variety of business activities and goals. Due to their size, complexity, and necessity for reliable, adaptable, and secure solutions, these businesses have particular needs. Python is a powerful programming language that comes with a number of features that make it ideal for addressing the needs of business IT.

Some key features for Enterprise IT and how Pythons features address these needs:

**Scalability and performance:**

Large-scale applications that require a lot of customers or data are frequently handled by businesses. Python provides a variety of utilities and libraries for performance enhancement, including:

Python's multiprocessing and threading modules let programmers build parallel processes and threads that efficiently make use of multi-core computers for better performance.

Asyncio: Asynchronous programming is made possible by the Python *asyncio* framework, which enables programs to effectively handle a lot of concurrent I/O-bound activities.

**The versatility and Integration:**

Integrating enterprise systems with already-existing programs or services is common. Support for Python's integration capability comes from:

Interoperability: Using tools like the CPython API, Python has strong support for interfacing with other languages, enabling programmers to easily repurpose preexisting libraries and applications.

RESTful APIs: Python's frameworks, such Flask and Django, make it simple to build and use RESTful APIs, simplifying communication between various components of an enterprise system.

**Ease of Development:**

Enterprises need quick development and deployment cycles, therefore ease of development is important. Features of Python that facilitate development include:

Readability: Python's readable syntax speeds up developer communication and cuts down on development time.

Rich Standard Library: The wide standard library for Python offers pre-built modules for many tasks, reducing the requirement to create functions from scratch.

Package management: Third-party libraries can be easily installed and managed with the use of tools like pip, which increases development productivity.

**Security:**

Enterprise IT's top priority is security as information is being dealt with in a crucial field within a wide scale and can be detrimental to a business.

Python takes care of this by:

Python has a strong ecosystem that is developed and often updated with security patches, lowering the chance of vulnerabilities.

Security for web applications: By default, web frameworks like Django stress security best practices, assisting developers in avoiding typical security mistakes.

**Cross-Platform Compatibility:**

Business systems must function on a variety of OSs. Python's cross-platform support makes it possible to create and deploy applications across a variety of settings with little effort.

**Community and aid:**

The Python programming language has a sizable and vibrant community that offers developers working on large-scale projects a wealth of tools, documentation, and assistance. This guarantees that solutions to problems are easily accessible.

c.

Functional Vs Non-functional: Performance, scalability, and security are examples of non-functional needs, while features and capabilities are classified as functional requirements.

Criticality: Determining needs that are essential to the core functioning and business goals is part of prioritizing based on criticality.

Time Sensitivity: Certain tasks could be time-sensitive, such as fulfilling urgent operational demands or adhering to regulatory deadlines.

User Needs: By prioritizing features according to user needs, it is ensured that those that are most closely aligned with user wants are handled first.

Question 2

a.

The term "concurrency" describes a computer system's capacity to control several activities or processes simultaneously. These tasks can start, run, and finish separately in a concurrent system, perhaps overlapping in time.

**Concurrency in GUI Applications:**

Concurrency tends to concentrate on ensuring a responsive user interface while carrying out routine tasks within GUI (Graphical User Interface) applications. GUI programs must maintain the user interface's response to user inputs and clicks.

Example: Take a music player app (YouTube Music, Spotify etc.). The program may simultaneously update the playlist, download album art from the internet, and search for new song recommendations while the user is listening to a song. These background processes guarantee that the user interface responds even when several tasks are being done at once.

**Concurrency in Enterprise Applications:**

Contrarily, concurrency in corporate applications often entails handling several tasks or requests concurrently. These programs frequently handle intense processing, data manipulation, and interaction with a variety of services.

Example: Multiple concurrent processes may start when a client puts an order in a large e-commerce system. These procedures might involve producing invoices, handling payments, sending emails of confirmation, and updating inventory. By ensuring that several order processing operations are carried out concurrently, concurrency shortens processing times and boosts customer satisfaction.

The emphasis on and applications for concurrency in the two differ:

GUI: By offloading background work, preserving UI responsiveness, and avoiding the program from becoming unusable owing to time-consuming procedures, concurrency in GUI applications provides a seamless user experience.

Enterprise: Concurrency improves resource consumption in corporate systems by managing numerous processes at once. It improves overall system performance and enables the program to effectively serve a big number of users.

While concurrency is advantageous for all types of applications, the focus and underlying motives vary depending on the tasks that each type of program handles and the user experience that it seeks to deliver.

b.

**Threading:**

Within a single process, threading entails creating and controlling threads. Since threads share a single memory region, interthread communication may be effective. Threads are useful for I/O-bound processes, in which the application must wait for outside data like document input/output, internet inquiries, or user input over a prolonged period of time.

Example:

Threading is advantageous for web scrapers that need to collect data from numerous websites at once, for instance. different threads can continue obtaining data from different websites while one thread waits for data from one domain.

**Multiprocessing:**

With multiprocessing, distinct processes with their own memory spaces are possible. The multi-core CPUs seen in contemporary systems are used by this method. For CPU-bound tasks requiring intense computing and where each process may make use of a different core for parallel processing, multiprocessing is appropriate.

Examples include filtering and picture transformation, which can be CPU-bound operations. Through the use of multiprocessing, many cores may be successfully used by each process while it works on a separate picture at the same time.

**Asyncio:**

Coroutines and event loops are used in asynchronous programming with the Asyncio toolkit. It is intended for jobs that need waiting for external resources and are I/O-bound. Asyncio doesn't rely on many threads or processes as multiprocessing or threading do. Instead, a single thread is effectively used to manage asynchronous operations.

Example: an asyncio-built web server may manage several client requests at once. The event loop can switch to processing another request while one is waiting for data from a client.

**Concurrent.Futures:**

A high-level interface for coordinating concurrent function execution is provided by the *concurrent.futures* module. The difficulty of directly controlling threads or processes is abstracted. There are two executor classes in it: *ProcessPoolExecutor* for processes and *ThreadPoolExecutor f*or threads. It makes it easier to manage results from asynchronously running functions.

Example: The *ThreadPoolExecutor* makes it simple to submit numerous tasks for execution at the same time and manages the threading complexities. This is particularly useful when you need to run several functions simultaneously and gather their results.

c.

The multiprocessing module in Python may be used to share states among many processes. Multiprocessing is one of the methods offered by the module. Multiprocessing and value. Use an array to generate shared variables that several processes may use and modify. To avoid data loss, these shared objects are synced.

You can also establish a server process using the multiprocessing module that controls a shared object and gives other processes access to it. When you need to communicate more intricate data structures, such as dictionaries or lists, this method is helpful.

d.

A scenario called a "deadlock" might result from improperly freeing up a newly gained lock. When multiple procedures are unable to go forward while each is awaiting the release of a resource, a deadlock has occurred. This leads to a scenario where the processes are effectively at a stop and are unable to go further.

Way to prevent this:

**Lock Ordering**: Guarantee that locks are always obtained throughout all threads or procedures in the same order. By doing this, deadlock-causing circular dependencies are avoided.

**Timeouts:** Use them after gaining locks. The thread or processes can release any current locks and attempt again later if a lock cannot be gained within a predetermined amount of time.

**Resource Allocation**: For obtaining and releasing locks, use context managers (with statements). In spite of exceptions, locks are constantly released thanks to this.

**Avoiding Excessive Locking:** Limit your use of locks by just getting them when they are required. Overuse of locks might make deadlocks more likely.

**Testing and Analysis**: Use tools and techniques for studying concurrency concerns and thoroughly test your concurrent code to spot probable deadlock scenarios.

SECTION B

Question 3

Eduvos

BSc in Software Engineering

Microservices and their Transformation of Enterprise Application Integration

ITEPA3-33

1 September

Elle Reece Chetty

C3WGVD453

C3wgvd453@vossie.net

# Contents

## Introduction

EAI is an abbreviation for "Enterprise Application Integration." It is the process of linking and integrating multiple software programs and systems inside an organization so that they may operate seamlessly together. EAI's purpose is to streamline company processes, communicate data and information, and increase overall organizational efficiency.

EAI is a vital component of current corporate operations. Traditional monolithic designs have dominated EAI over the years. However, the emergence of microservices architecture has fundamentally altered how businesses approach application integration. This paper presents an overview of microservices, analyses their architecture, and benefits, addresses implementation obstacles, investigates how microservices have transformed EAI, and assesses whether major companies should consider using a microservices strategy for their applications.

(2017)

This research will examine microservices in depth and their significant influence on Enterprise Application Integration (EAI). In these pages, we will look at the fundamentals of microservices, their architectural principles, and the various benefits they provide. Furthermore, we will investigate the issues that occur during the deployment of microservices, as well as how this architectural approach has impacted the EAI environment. Furthermore, we will have a critical debate on the viability of major companies shifting to a microservices-based architecture for their applications.

## Overview of Microservices

Microservices is an architectural approach that frames an application as a collection of tiny, loosely connected services that may be built, deployed, and scaled separately. Microservices, as opposed to monolithic programs in which all components are closely interwoven, divide the application down into independent available services, each accountable for a distinct business feature. These services interact via APIs, allowing for simple integration. APIs, or program Programming Interfaces, are sets of rules and protocols that enable one software program to interface with or access the functionality and data of another application, service, or platform.

### Key Principles of Microservices Architecture

Modularity: Microservices encourage the breakdown of large applications into smaller and independent services. Each service should represent a distinct business skill or function. Individual services are easier to comprehend, build, and manage with this modular approach.

Independence: Microservices are designed, tested, and deployed separately. This independence enables teams of developers to work on many services concurrently while remaining loosely linked to the rest of the application.

Polyglot Architecture: Microservices empower teams to select the best technological stack for each service based on its needs. This encourages a polyglot design in which numerous programming languages, frameworks, and systems may be utilized as needed.

Resilience: Microservices are built to be resilient. Failures in a single service should not affect the entire application. Redundancy, failover techniques, and gradual degrade are frequently used to guarantee that the system remains functioning despite failures with individual services.

Scalability: Depending on demand, microservices may be scaled individually. This enables optimal resource use as well as fast performance. Services with high traffic or resource-intensive operations can be horizontally scaled to manage rising demands.

These principles explain the underlying idea of microservices architecture and serve as a guide for designing, developing, and maintaining microservices-based applications.

(Nunes et al., 2021)

## Microservice Architecture

Key Characteristics of microservice architecture :

Service Modularity: A microservices architecture divides an application into a collection of distinct services, each of which is responsible for a particular task or business capability. A typical e-commerce application, for example, may include services for user administration, goods database, processing payments, and order fulfilment.

Independence: Microservices are designed, tested, and deployed separately. Each service can have a separate team of developers, technology stack, and deployment cycle. This independence enables faster growth and greater flexibility.

API-Based Communication: Microservices use well-defined APIs to communicate with one another and with other systems. These APIs allow services to communicate and share data by utilizing common protocols such as HTTP/REST or message queues.

Decentralized Data Management: Microservices are services can have their own information storage options, such as databases or information stores, thanks to decentralized data management. Because of this decentralization, each service may manage its data independently. Maintaining data consistency between services, on the other hand, might be difficult.

Scalability: Based on their individual resource requirements, microservices may be scaled separately. This implies that services with heavy traffic or tasks requiring a lot of resources may be adjusted up or down as necessary.

Technology Diversity: Depending on their specific requirements, different services within a microservices application might employ a variety of technology stacks, programming languages, and frameworks. This encourages creativity and enables teams to select the most effective tools for the job.

Resilience: Microservices are built to be resilient. Mistakes in one service shouldn't end up in application failure.

Organizational Impact: Microservices can have an impact on the hierarchical arrangement of development teams. They frequently result in smaller, cross-functional groups, each in charge of one or more services. This encourages responsibility and accountability.

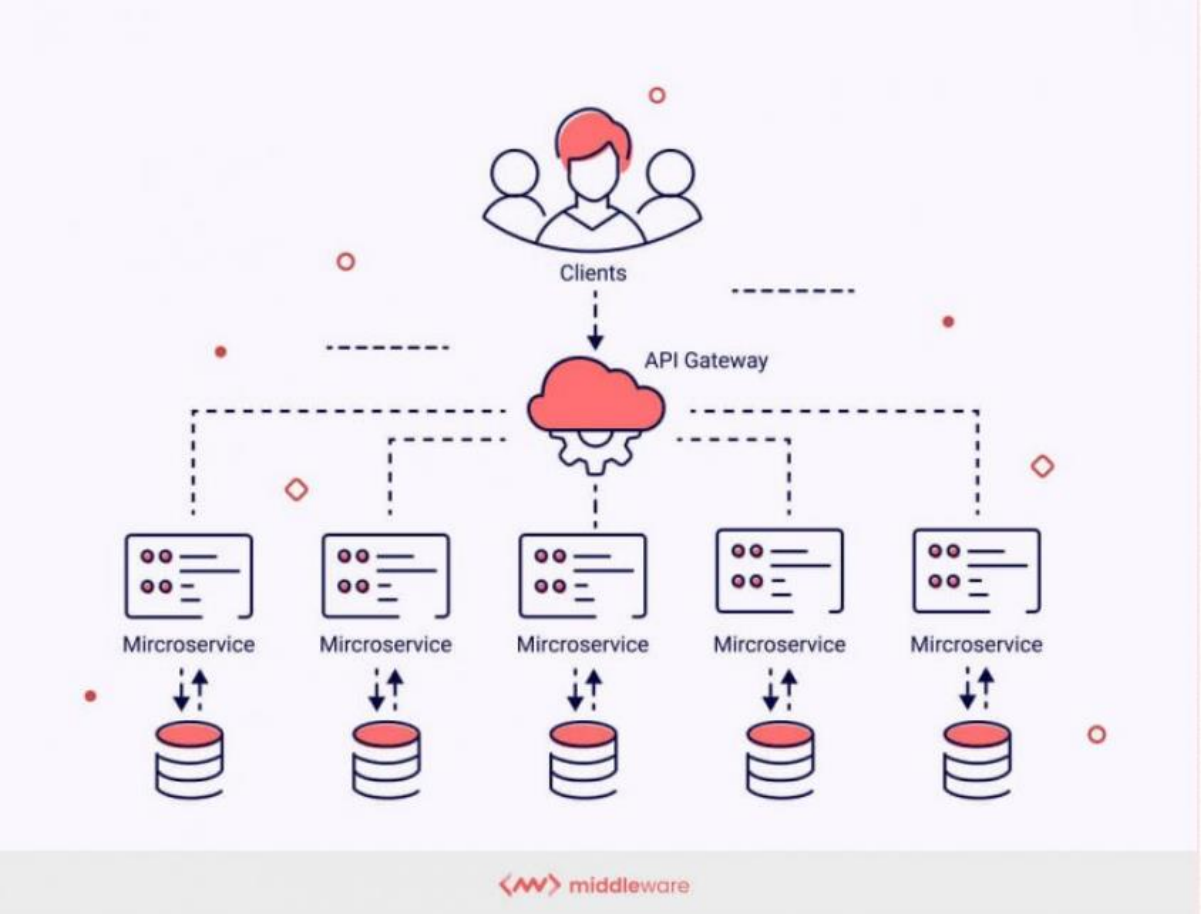


*Figure 1 :diagram illustrating microservice architecture.*

*Found at : https://middleware.io/blog/microservices-architecture/*

The user interface can be used by the client to produce requests. At the same time, the API gateway commissions one or more microservices to accomplish the specified operation. Therefore, even bigger complicated problems requiring the use of several microservices may be handled quite easily.

## Benefits of Microservices

### Scalability and Flexibility

Microservices design is extremely scalable and adaptable. This advantage can be achieved through :

Independent Scaling: Based on its individual resource requirements and workload, each microservice may be scaled separately. During peak traffic periods, for example, you can scale up services that are facing powerful loads while leaving others alone. This improves resource allocation and assures responsiveness.

Resource Efficiency: Because you may distribute resources just where they are needed, microservices provide efficient resource allocation. In contrast, growing monolithic systems frequently entails assigning resources to the entire program, which can be inefficient.

Adaptability :Microservices facilitate adaptation to changes in user demand or business requirements. To handle new features, new services may be introduced, and current services can be scaled up or down as needed.

### Faster Development and Deployment

Microservices design speeds up development and deployment in various ways such as:

Parallel Development: Teams of developers may collaborate on many microservices at the same time. Developers may focus on their individual service without being hampered by the development of different sections of the program since every feature is self-contained and has defined limits.

Reduced Time-to-Market: Because microservices are separated, they can be iterated and released more quickly. New features or upgrades may be created and delivered independently, decreasing the time it takes to put changes into production.

Testing Efficiency: It is simpler to properly test smaller, well-defined services. This supports shorter and more accurate testing cycles, reducing the likelihood of problems or regressions being introduced by modifications.

### Fault Isolation

Fault isolation is a key benefit of microservices design. Because services are isolated, a failure or defect in one service does not affect the whole application. As a result,

Enhanced Reliability: Even when a service fails, the entire system remains stable. Users encounter fewer interruptions, and important company operations can continue to operate even when isolated difficulties develop.

Simplified Debugging and Recovery: Because the area of research is confined to the impacted service, debugging, and diagnosing problems are simplified. It is easier to identify and resolve issues, resulting in less downtime and service disruptions.

### Technology Diversity

Microservices embrace technological variety, which offers various benefits such as:

Innovation: Development teams are allowed to select the best technological stack for each service. This encourages creativity since teams may employ the most up-to-date tools and technology for their specialized work.

Flexibility: Microservices are adaptable to numerous programming languages, structures, and databases. This adaptability is especially useful when dealing with fluctuating service requirements and restrictions.

Vendor Neutrality: Organizations are not locked into a single software vendor or stack. They may choose the finest technology for their needs, which reduces vendor lock-in and increases agility.

### Enhanced Maintenance

The microservices design simplifies maintenance and updates:

Service Boundaries: Because each microservice has a well-defined boundary, it is easier to understand, manage, and update. Developers can concentrate on a single service without affecting the overall program.

Versioning: Service can be individually versioned, providing for retrogression and incremental upgrade deployment. This reduces end-user disturbance during maintenance.

Regression Risk is Reduced: Smaller, separated services are far less probable to generate regressions when updated. Testing may be highly focused and rigorous, reducing the possibility of unforeseen consequences.

## Challenges of Microservices Implementation

### Complexity

While microservices provide significant benefits, they also bring complexity in some areas:

Service Management: While the variety of services increases, it becomes more difficult to manage, deploy, and monitor them. To successfully address this complexity, proper orchestration, and automation solutions, such as Kubernetes and Docker Swarm, are essential.

Service Discovery: Discovering and linking to the proper service among several might be difficult. Service discovery techniques are required for dynamically managing service locations.

Dependency Management: The services may rely on one another. It can be difficult to track and manage these dependencies, therefore circular dependence should be avoided.

### Data consistency

Managing data consistency between microservices may be difficult:

Data Storage Decentralized: every microservice may have a separate information store. It might be difficult to ensure data consistency when numerous services want accessibility to the same data. It is important to implement dispersed transactions, eventually uniformity, or other solutions.

Data Synchronization: Maintaining data synchronization across services might result in challenges such as data conflicts and obsolete information. Mechanisms for data synchronization must be properly devised and implemented.

### Communication Overheard

Inter-service communication can present several difficulties:

Delay: Inter-service communication, particularly when conducted via a network, can cause delay. It is critical to create effective communication protocols and reduce needless phone calls.

Message Formats: Service must agree upon message formats and communication methods. Communication problems can occur when formats are inconsistent or incompatible.

Handling Errors: Handling errors in communication, timeouts, and retries may prove difficult. Error management and fault tolerance techniques must be robust.

### Testing and Monitoring

Testing and monitoring provide distinct issues in a microservices environment:

Distributed Testing: Microservices are dispersed, which makes end-to-end testing more difficult. Interactions between services must be well tested.

Isolating services : Service isolation monitoring might be difficult because they are built to function together. Mocking and stubbing treatments are typical methods for dealing with this.

Troubleshooting: It might be difficult to identify and diagnose problems across various services. Implementing good logging, tracing, and debugging techniques is critical for resolving issues quickly.

# Transformation of EAI with Microservices

## Agility and Stability

Microservices improve EAI's agility and scalability in the following ways:

Rapid Adaptation: Microservices enable firms to adapt swiftly to changing marketplace circumstances and growing business demands. New services can be separately developed and launched, allowing for rapid response to market needs or regulatory changes.

Scalability: EAI systems frequently experience fluctuating demands. The autonomous scaling capacity of microservices allows enterprises to distribute resources where they are most required, guaranteeing optimal utilization during periods of high demand and cost savings throughout lulls.

Resource Efficiency: Scaling just the essential microservices optimizes resource use, lowering infrastructure costs.

## Beter Integration

Microservices increase EAI system integration by exploiting APIs and contemporary communication standards:

API-Centric Approach: Because microservices communicate via APIs, it is easier to incorporate new services into current EAI systems. This standardized communication approach makes it easier to link diverse systems and ensures data flows seamlessly across them.

Ecosystem Expansion: Organizations may simply expand their EAI systems with microservices by adding or changing specific services without interrupting the overall system. This adaptability is essential for integrating fresh collaborators, data sources, or technology.

Loose Coupling: The loose coupling of microservices eliminates dependencies across integrated systems, which makes it easier to replace or update specific services without disrupting others.

## Simplified Maintence

Microservices make EAI system maintenance easier by allowing for granular updates:

Service-Level Maintenance: Updating and maintenance may be handled individually for each service. This level of granularity reduces the possibility of system-wide failures resulting from modifications or mistakes in a single service.

Rollback Capability: In the unlikely circumstance that an upgrade causes problems, microservices allow for simple rollbacks, restoring system stability rapidly.

Continuous Improvement: Teams may concentrate on improving specific services without interfering with the broader EAI infrastructure. This promotes an environment of constant development and innovation.

### Improved Fault Tolerance

Fault separation in microservices improves tolerance for errors in EAI systems:

Isolated Failures: Each microservice is self-contained. If a service fails, the failure does not spread to the whole EAI infrastructure. This separation guarantees that key data flows and company operations may continue as usual.

Redundancy and Resilience: For important services, EAI solutions can include redundancy and resilience techniques. This reduces the effect of service failure and improves overall system dependability.

Graceful Degradation: Microservices provide graceful degradation, which allows non-essential services to scale down momentarily during failures to prioritize key functions. This method keeps critical operations running while fixing difficulties.

## Should Large Enterprises Switch to Microservices?

Several variables influence the choice to move to a microservices architecture. Microservices provide substantial benefits in terms of agility and scalability, but they also present complications and problems. Large businesses should think about the following:

### Legacy Systems

Gradual Approach: Transitioning between monolithic to microservices may be especially difficult when working with large legacy systems. Rewriting or replacing all old components at once may not be practicable or practical. A progressive strategy, in which microservices are slowly integrated alongside existing systems, can limit disruption and hazards.

Legacy Integration: Critical business reasoning and information are frequently found in legacy systems. Integration of microservices with old systems should be planned carefully, as it may necessitate the use of integration tools and adapters to close the divide between current and ancient designs.

Decommissioning old Systems Strategically: Identify old systems that can be phased out or decommissioned when fresh microservices take over their functions. This needs a defined migration plan that ensures data integrity & business continuity.

### Risk Tolerance

Risk Assessment: The shift to microservices entails risks such as possible interruptions, higher difficulty, and a steep learning curve related with new technologies. Assess the organization's risk tolerance and capacity to manage these risks effectively.

Mitigation Strategies: Create mitigation plans for probable problems and interruptions. This comprises service failure contingency plans, data migration procedures, and rollback plans in the event of problems during the transition.

### Business Needs

Alignment with Strategy: Determine whether the agility and scalability advantages provided by microservices connect with the strategic goals of the firm. Microservices are an effective tool for firms who need to adapt swiftly to market developments or develop quickly. They may not, however, be the ideal match for all businesses or circumstances.

Cost-Benefit: Consider the cost-benefit implications of migrating to microservices. While microservices have advantages, there are expenses connected with their development, maintenance, and architecture. A comprehensive cost-benefit analysis can assist in making an educated decision.

The choice to migrate to microservices is influenced by considerations such as the complexity of your present system, scalability requirements, development pace, resource availability, dependency on legacy systems, and integration needs. It is not a one-size-fits-all solution and should be tailored to your individual business objectives and circumstances. While switching to microservices might not be an immediate requirement, it may be a realistic choice in the future. Implementing microservices may become a strategic decision to improve agility and scalability as technology changes and your organization expands. Therefore, even if it may not be the best option right now, it's something to think about in the future.

## Conclusion

The choice to transition to microservices is driven by several criteria, including the complexity of your present system, scalability requirements, development pace, resource availability, dependency on legacy systems, and integration requirements. It is not a solution that fits all and should be tailored to your specific business goals and circumstances.

While moving to microservices might not be an urgent need, it is critical to realize the possibilities as a long-term strategic option. As technology evolves and your firm grows, utilizing microservices can provide increased agility and scalability. As a result, while it might not be the best option right now, it is something to think about for future development and innovation. A well-informed selection based on your individual requirements and goals can prepare you for a smooth transfer when the time comes.

*LIST OF TABLES*



## Bibliography

'Yesterday, Today and Tomorrow' (2017) Gangland Oz, pp. 228–241. doi:10.2307/jj.5993309.17.

Nunes, J.P. et al. (2021) 'State of the art on Microservices Autoscaling: An overview', Anais do XLVIII Seminário Integrado de Software e Hardware (SEMISH 2021) [Preprint]. doi:10.5753/semish.2021.15804.

Taibi, D. et al. (2017) 'Microservices in Agile Software Development', Proceedings of the XP2017 Scientific Workshops [Preprint]. doi:10.1145/3120459.3120483.

Jamshidi, P. et al. (2018) 'Microservices: The journey so far and challenges ahead', IEEE Software, 35(3), pp. 24–35. doi:10.1109/ms.2018.2141039.

Jun Gui and Hebiao Yang (2010) 'Realization of EAI based on service-oriented architecture', 2010 International Conference on Educational and Information Technology [Preprint]. doi:10.1109/iceit.2010.5607607.

Bibliography (entire doc)

LinkedIn. (n.d.). How Do You Leverage Cloud, Microservices, EAI, SOA Skills? Retrieved August 24, 2023, from https://www.linkedin.com/advice/3/how-do-you-leverage-cloud-microservices-eai-soa-skills-eai

ITRex Group. (n.d.). What Is Enterprise Application Integration (EAI)? Retrieved August 24, 2023, from https://itrexgroup.com/blog/what-is-enterprise-application-integration-eai/

FogPlug. (n.d.). The Evolution of Enterprise Application. Retrieved August 24, 2023, from https://www.fogplug.com/p/the-evolution-of-enterprise-application

Netguru. (n.d.). Enterprise Software Development with Python: Challenges and Solutions. Retrieved August 24, 2023, from https://www.netguru.com/blog/enterprise-software-python

Full Stack Python. (n.d.). Enterprise Python: A Comprehensive Guide. Retrieved August 24, 2023, from https://www.fullstackpython.com/enterprise-python.html

Diva Portal. (2015). Requirements Prioritization in Agile Software Development [PDF]. Retrieved August 24, 2023, from https://www.diva-portal.org/smash/get/diva2:836447/FULLTEXT01.pdf

Requirements.com. (n.d.). What Is Requirements Prioritization? Retrieved August 24, 2023, from https://requirements.com/Content/What-is/what-is-requirements-prioritization

TutorialsPoint. (n.d.). Concurrency in Python - Quick Guide. Retrieved August 24, 2023, from https://www.tutorialspoint.com/concurrency_in_python/concurrency_in_python_quick_guide.htm

Towards Data Science. (n.d.). Concurrency and Parallelism in Python. Retrieved August 24, 2023, from https://towardsdatascience.com/concurrency-and-parallelism-in-python-bbd7af8c6625

GeeksforGeeks. (n.d.). Deadlock in DBMS. Retrieved August 24, 2023, from https://www.geeksforgeeks.org/deadlock-in-dbms/

University of Illinois at Chicago. (n.d.). Deadlocks. Retrieved August 24, 2023, from https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/7_Deadlocks.html

'Yesterday, Today and Tomorrow' (2017) Gangland Oz, pp. 228–241. doi:10.2307/jj.5993309.17.

Nunes, J.P. et al. (2021) 'State of the art on Microservices Autoscaling: An overview', Anais do XLVIII Seminário Integrado de Software e Hardware (SEMISH 2021) [Preprint]. doi:10.5753/semish.2021.15804.

Taibi, D. et al. (2017) 'Microservices in Agile Software Development', Proceedings of the XP2017 Scientific Workshops [Preprint]. doi:10.1145/3120459.3120483.

Jamshidi, P. et al. (2018) ‘Microservices: The journey so far and challenges ahead’, IEEE Software, 35(3), pp. 24–35. doi:10.1109/ms.2018.2141039.

Jun Gui and Hebiao Yang (2010) ‘Realization of EAI based on service-oriented architecture’, 2010 International Conference on Educational and Information Technology [Preprint]. doi:10.1109/iceit.2010.5607607.