International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395-0056
Volume: 11 Issue: 08 | Aug 2024 www.irjet.net p-ISSN: 2395-0072

Fortifying Financial Systems: Exploring the Intersection of Microservices and Banking Security

Sumit Bhatnagar[1], Roshan Mahant[2]

[1]*Individual Researcher, New Jersey, USA*
[2]*LaunchIT Corp, Urbandale, IA USA*

Abstract- As part of their digital transformation, financial service companies can greatly benefit from the implementation of a microservice architecture. We can build a service-oriented architecture (SOA) application using the architecture to enhance its overall performance and maintainability. This enables the application to consist of several smaller components that operate independently and simultaneously. In the financial services industry, the accuracy of artifact states holds immense significance. Given that an inaccurate artifact state or anomalous artifact operation(s) could potentially ruin the entire application, it is crucial to conduct an analysis of the artifact operations within each microservice during the design process. In this study, we present a technique for identifying anomalies through the characteristics of artifacts associated with the microservice architecture. Following this, we identify the properties of artifacts. As a result of technological improvements and shifting client expectations, the financial sector is currently going through a tremendous upheaval. Among these advancements, microservices architecture has emerged as a key enabler of agility, scalability, and innovation in banking systems. This paper delves into the relationship between microservices and banking security, emphasizing the use of microservices to strengthen financial systems in the face of growing cyber security risks. The microservices architecture simplifies the development, deployment, and scalability of big financial applications by separating them into smaller, self-contained services. This modular approach not only makes the system more reliable and helps separate problems, but it also makes continuous integration and delivery (CI/CD) easier, which lets you respond quickly to changes in the market and new rules. However, the implementation of microservices introduces new security challenges, such as controlling inter-service communication, ensuring data integrity, and safeguarding against attack detection and protection.

Keywords -microservices, artifact anomaly, workflow

INTRODUCTION

In recent years, microservice architecture has garnered a lot of interest due to the numerous advantages it offers, particularly in applications that are both sophisticated and large. The strategy is based on the idea of dividing an application into a number of smaller services that serve a specific purpose. This makes the process of development and maintenance simpler, more efficient, and more scalable. Microservice architecture is especially helpful in the context of financial systems, which are frequently big and complicated and call for high levels of reliability and fault tolerance. We will discuss the advantages of utilizing microservice architecture in financial systems, along with the challenges to successfully implement this strategy, in the following paragraphs. Compared to organizations that specialize in financial technology, the majority of traditional financial services have become less innovative over the course of the past few years. This is mostly due to the fact that these services rely heavily on their big, monolithic legacy systems. To a large extent, these systems are highly stiff, and they frequently become maintenance headaches due to their limited adoption of new technologies and performance [1]. Overall, they are quite rigid. The adoption of system design that is more adaptive to changes, such as new technologies and faster performance requirements, is something that organizations that provide financial services are contemplating in order to maintain their competitive edge [2].An example of a service-oriented architecture (SOA) is the microservice design, which breaks down large applications into smaller, autonomous parts called microservices that can operate in tandem. [1, 2]. Microservices are also known as microservices. The ability to scale and operate a microservice independently at smaller granularities is one of its defining characteristics. which allows for a significant improvement in both the system's performance and its ability to be maintained. Every microservice, for instance, has the ability to adopt a variety of technological platforms that are optimally suited to the performance levels that it requires.

The process of developing applications that use the microservice architecture frequently uses workflow models. These models assist in both the development of a microservice and the planning of its partnership with other service units. Conducting an analysis of artwork is absolutely necessary to ensure accurate workflow execution. We define artifacts as the data or objects specified, employed, or referenced by the operations within a workflow. Despite the well-behaved workflows,

conflicts between the operations could potentially lead to erroneous states. The distribution of microservices across multiple locations necessitates a thorough examination of the artwork. This is because an inaccurate artifact state has the potential to corrupt the entire system. For instance, the process of message passing might propagate an improper state from one microservice to other microservices.

In this paper, the authors provide methods for identifying suspicious artifacts inside microservices. This research's topic is "Fortifying Financial Systems: The Intersection of Microservices and Banking Security." Initially, they determine the quality of the artworks by analyzing the microservices' features. They first show two methods that use these properties to detect abnormalities, and then they turn a microservices workflow model into the appropriate SP-tree structure.

Using the detection result, the designer can make adjustments to the service application to prevent the system from experiencing a failure state. E.g., below: Digital Banking Architecture

Microservices in finance Sector

Microservices and layered architectures are the most common types of architectures for core banking systems. As a result, it is essential to disclose their most important characteristics, as well as their strengths and weaknesses, in order to create an excellent fintech solution and make an informed decision. Therefore, let's begin with a stratified structure.

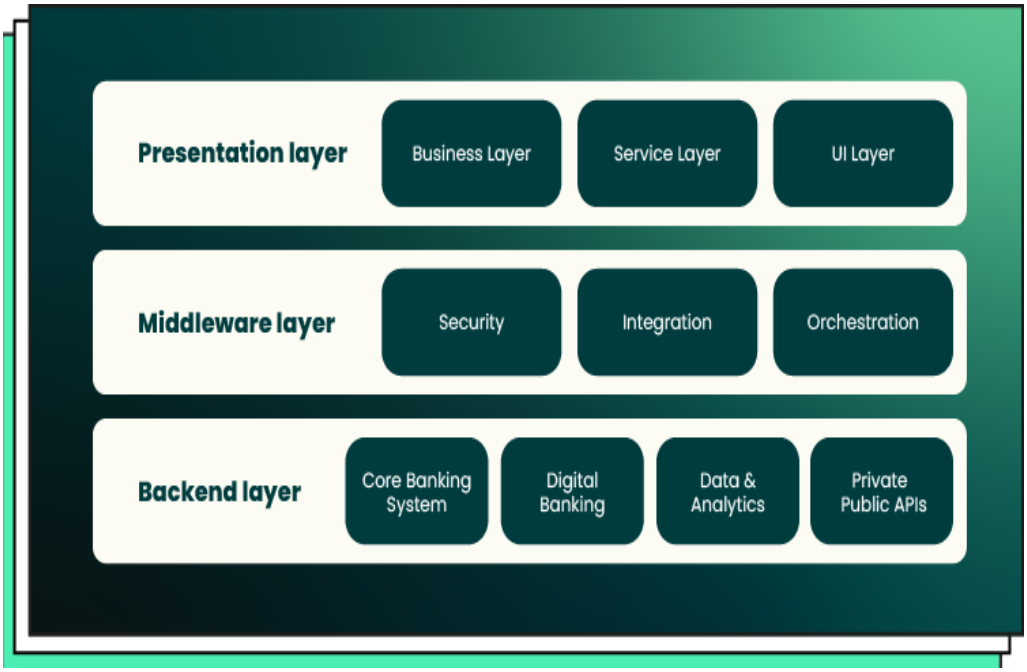


Fig.1 Layered architecture

The presentation layer, often known as the front end, middleware, and backend, are the components that make up a standard tiered design, as depicted in figure 1. Modularity, scalability, and maintainability are among its most important characteristics. These characteristics make it simple to implement changes and add new features. Each layer of this design represents a different module, simplifying the development of this architecture. Additionally, we can apply different scales to each layer separately. At long last, it is feasible to improve one layer without having an effect on the other layers.

A number of benefits, including ease of creation, clarity, and simplicity, are associated with this digital bank design. Therefore, we can use it to build large and complex banking solutions, improve existing ones, and incorporate new features.

The performance overhead and rigidity of a layered design are two reasons why it is considered to be a weakness. Communication between levels may complicate the process of setting up and carrying out functions. In addition to this, the layered method is typically stiff, which makes it difficult to make changes in a single layer.

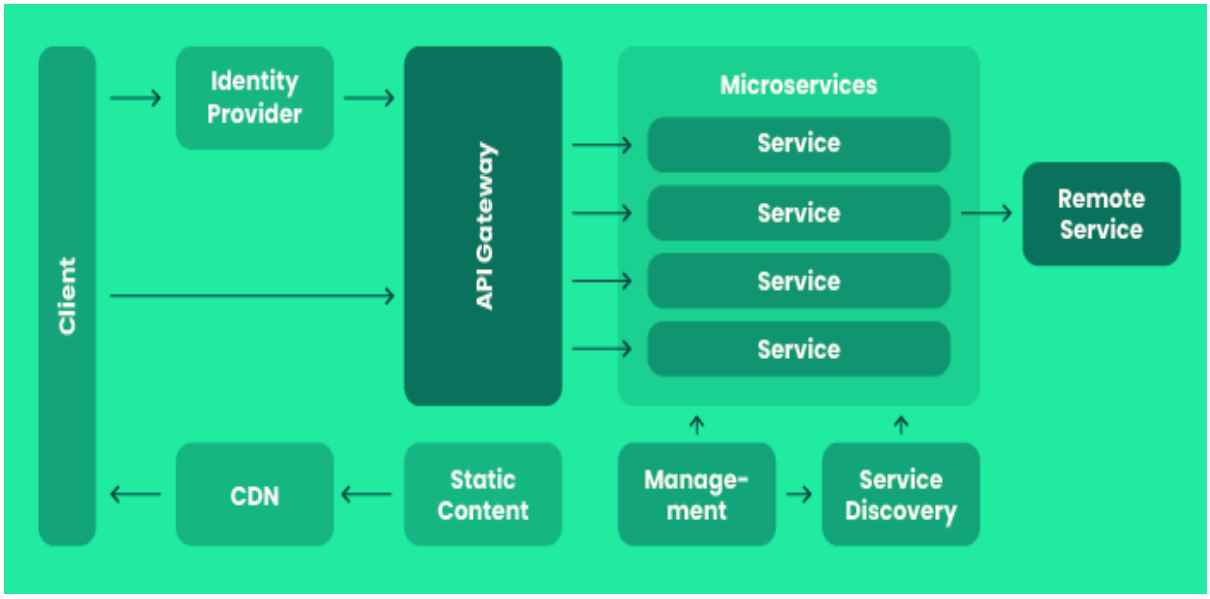


Fig.2 Microservices-based architecture

Figure II depicts the microservices-based architecture. Figure II displays a variety of services that are deployable independently. Each service performs a specific business function, and these services communicate with each other. Numerous digital banks construct their banking applications using microservices architecture. Now, let's explore the factors that contribute to its popularity. In addition to providing the largest prospects for scalability, microservices also make it possible to integrate with a wide variety of third-party services in a seamless manner.

Benefits of Using Microservice Architecture in Financial Systems

Scalability

The scalability of microservice architecture is among the most important advantages that use this design. Sometimes, financial systems are difficult to understand since they are made up of a wide variety of components that need to be controlled and maintained. Microservice architecture makes it possible to divide and partition these components with ease, which makes it simpler to scale the system up or down according to the requirements of the circumstance. The general scalability and flexibility of the system is improved as a result of this functionality, which enables developers to tweak or add new features to the system without causing disruption to the system as a whole.

Agility

Microservice design in financial systems offers a number of benefits, one of which is more agility. The ability to swiftly react to changing market conditions and customer requirements is an essential quality for financial systems. By enabling developers to rapidly adjust and add new features to the system, microservice architecture makes it possible for financial systems to be more adaptable and sensitive to circumstances that are always shifting.

Fault Tolerance

Financial systems are required to have a high degree of reliability and tolerance for errors. It is of the utmost importance that the system be able to get back up and running without any interruptions in the event that it has a breakdown or an outage. This helps to lessen the impact of any failures or outages that may occur. Microservice design offers fault isolation and containment. The testing and monitoring of the system is also simplified by this approach, which enables developers to quickly discover and resolve any problems that may arise.

Technology Heterogeneity

One of the advantages of microservice architecture is its ability to support the utilization of many technologies. Financial systems typically consist of a variety of components, each constructed using a variety of technologies or computer languages. Microservice architecture enhances the system's overall performance and scalability by allowing the construction of each component using the most appropriate technology.

Challenges of Using Microservice Architecture in Financial Systems

Implementing microservice architecture presents challenges despite its benefits. These challenges include:

Complexity

When working with huge and complicated financial systems, microservice architecture can be particularly difficult to understand and implement. The complexity of the system increases in proportion to the number of microservices that are present, making it more difficult to manage and maintain. It is necessary to practice careful planning and management in order to guarantee that the system will continue to be scalable, maintainable, and secure.

Testing

In addition, testing might be a difficult task when dealing with microservice architecture. It is necessary for us to perform unique tests on each microservice in order to verify that it functions appropriately and integrates with the other microservices that are present in the system. Performing this might be a time-consuming process that calls for meticulous organization and coordination in order to guarantee that the testing is comprehensive and efficient.

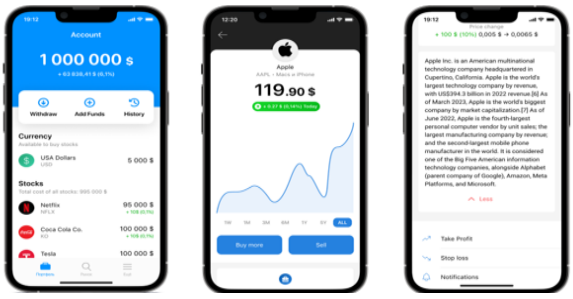
Data Consistency

The use of microservice architecture in financial systems may pose challenges in terms of maintaining data consistency. It is difficult to guarantee that the data is consistent throughout the entire system because each microservice could have its own data store or database.

Careful design and management are required to ensure data synchronization and consistency across all microservices. There is a kind of software design known as microservices architecture. This form of architecture divides an application into a collection of small, independent services. Every service handles its own process and communicates with others via well-defined APIs. In order to design and maintain complex systems, developers have the ability to develop, launch, and scale these services independently of one another. This provides a variety of benefits.

Finbox as a case study

By easing the introduction of new white-label solutions, Finbox helps financial institutions and fintechs increase their customer base and revenue. One of these is an investment product.



Through the use of the Java Spring Framework, we initially developed and constructed the backend using the Java Spring Framework in a monolithic architecture. Using the PostgreSQL database, they were able to construct everything into a single application. This strategy expedited the implementation of business logic and guaranteed the consistent achievement of goals. On the other hand, it contributed to issues in terms of performance and scalability. Because of a specific use case, the FIX protocol was experiencing a period of high load. Since we wanted to avoid program crashes, we had to devise workarounds.

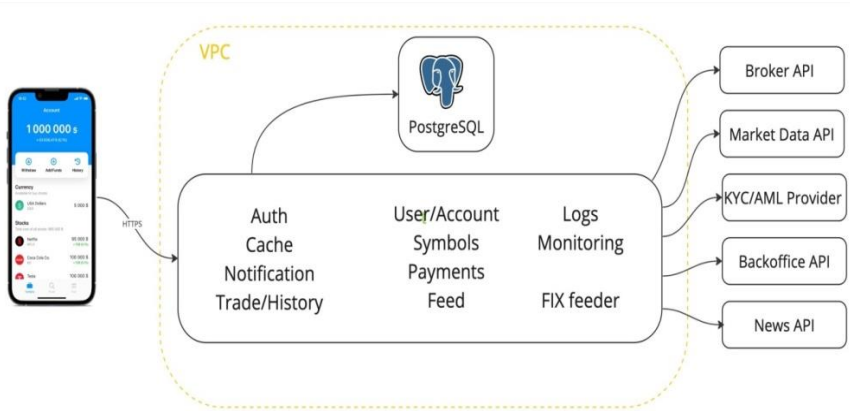


Fig.3Monolithic architecture

Authentication and authorization, caching, and notification services are some of the components that the financial services platform offers to banks and fintech companies in order to assist them in expanding their user base and capitalization, trade history, user account management, symbol management, payment gateway, data feed, log monitoring, FIX feeder, broker API, market API, KYC/AML service, back-office API, and news feed. The platform was initially constructed using a monolithic architecture that included the Java Spring Framework and PostgreSQL, as depicted in Fig. 1. Because of the high load on the FIX protocol, it faced performance and scalability issues. To address these, the platform transitioned to a microservices architecture with asynchronous task processing via Kafka, leveraging HTTPS and web sockets for stability. This transition allowed for independent scaling and better resource utilization, resulting in improved response times, throughput, and system stability, while also ensuring compliance and efficient user and trade management. As shown in Fig. 3, Monolithic architecture

The next iteration aimed to prevent issues with the high load and frequent failures of the FIX Feeder, while also enhancing the scalability of the API Gateway layer. This time, the challenge lay at the core of the system. Given the high degree of coupling in the software, any modification would result in crushes occurring in unexpected locations. Ineffective horizontal scaling was a result of errors that occurred during the design and implementation of the architecture.[1]

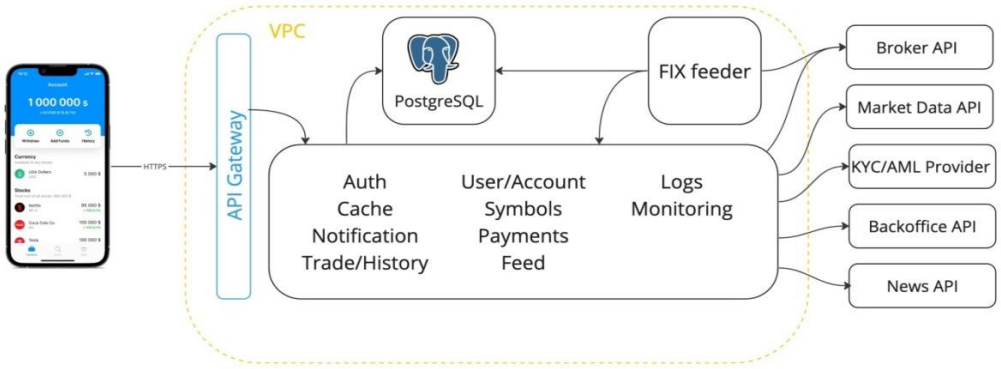


Fig.4 Microservices architecture

After learning from these two phases, we made the decision to transition to a fully decoupled microservices architecture, utilizing Kafka for asynchronous task processing. This system is currently quite flexible and scalable, able to accommodate changes in both the codebase and the architecture, as demonstrated in Figure 5: Microservices architecture. The core microservices handle the primary functionality, allowing for dynamic scaling of each service. On the other hand, the peripheral is responsible for managing connectivity with devices and APIs provided by third parties. They decided to switch from the FIX protocol to HTTPS and Web Sockets because of the stability and usability of these protocols.

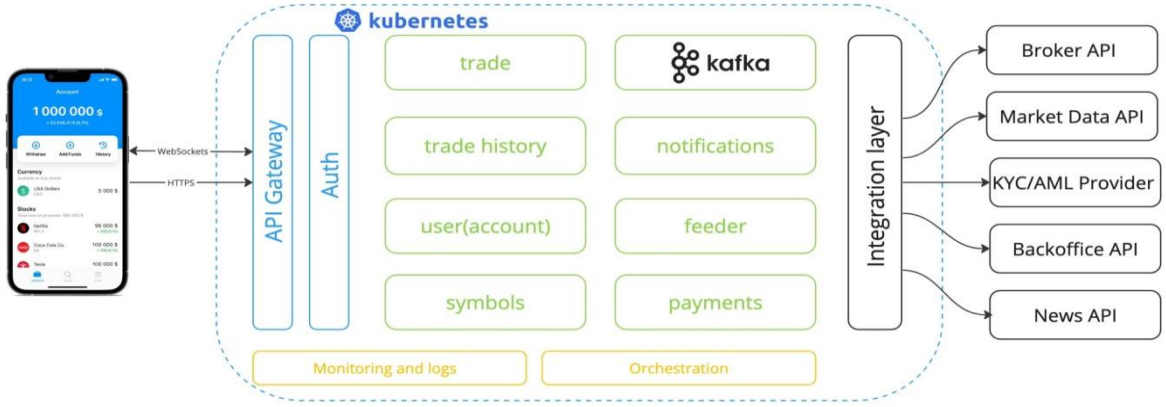


Fig.5 Microservices architecture integration model for finance industry

Response time is a critical performance metric that measures the time taken to process a request from start to finish. In a monolithic architecture, the response time can be longer due to the tightly coupled nature of the components, leading to potential bottlenecks. Transitioning to microservices architecture aims to reduce response time by decoupling components and enabling parallel processing. [1]

Monolithic Architecture

$RT_{monolithic} = \frac{1}{N} \sum_{i=1}^{N} RT_i$

Where:

- RTi is the response time for each request iii
- N is the total number of requests

Microservices Architecture:

$RT_{microservices} = \frac{1}{M} \sum_{j=1}^{M} RTj$

Where:

RTj is the response time for each request jjj

M is the total number of requests

Improvement in Response Time (IRT):

$IRT = RT_{monolithic} - RT_{microservices}$

Load (L)

Theory:

The load is the number of messages or requests handled by the system per unit time. A high load on a single component in a monolithic architecture can lead to performance degradation. Increasing both performance and reliability can be accomplished by distributing the load among a number of different microservices. L_{HTTPS}+WebSockets $= \frac{\text{Number of Messages}}{\text{time period}}$

ANOMALIES IN MICROSERVICE ARCHITECTURE

A microservice is defined as a programmable unit that is surrounded by contexts and has the ability to communicate with every other microservice through message forwarding. [3]. On various platforms, such as physical machines, virtual machines, or containers, we can manage, deploy, and scale different microservices independently [4]. An application with a microservice architecture can be built by assembling several distributed microservices, each of which handles its own process and communicates with others through message-based protocols. This design has at least three features: it is context-bound, smaller than a typical service in service-oriented architecture (SOA) [5], and operationally independent. Based on the observation of these traits, the following is a list of artifact attributes that can be located and identified: Through the use of a messaging mechanism, the independence feature ensures that the artifacts included within each microservice are independent of those contained within another microservice using a messaging mechanism. This facilitates the transfer of the artifact from one microservice to another. It is possible for each microservice to have its own unique context because of the limited context characteristic. When a message passes an artwork from one microservice to another, each microservice's context can give it a different name. However, the artifact itself remains separate and autonomous. The smaller size and independent properties of each microservice make it possible for it to readily expand horizontally [6]. This allows it to concurrently operate on multiple instances. This scenario allows for the simultaneous presence of identical independent artifacts on multiple processes in different locations. Using the above information about artifacts, finding strange artifacts within each microservice of the application (intra-micro service) can help check the accuracy of the artifact states sent between microservices and keep the system from failing. In the context of a microservice's related workflow, we can examine an artifact's erroneous state without regard to processes external to the microservice. When we exchange an artifact via a message and send it to other microservices or service units, we can interpret the act of receiving a message as writing on the artifact, and we can also consider each parameter it represents as a reading on the artifact. The following definition [7] identifies an artifact as having an intra-micro service artifact anomaly if it exhibits abnormal Read, Write, and Kill operations. (1) A concurrent artifact anomaly occurs when two parallel operations on the same artifact fall into one of the following pairs: (Kill, Write), (Kill, Read), or (Write, Write). A continuous artifact anomaly arises when two successive operations on the same artifact end, read, write, or terminate. written, written, or written.

Microservice, Structured Workflow, and SP-Tree

Using the microservice architecture [2], they are able to model the design of a microservice as a workflow that is contained within a pool. This enables the microservices to communicate with one another via message passing. Figure 6 provides a visual representation of an example of the microservices paradigm.

The workflow modeling process employs a structured workflow model (SW). They use this model to avoid stalemate or unwanted recurrence. [8] This paradigm guarantees that a joint node accompanies each split node at a specific level, and that nodes within the split-joint

structure lack any edges connecting to nodes beyond the structure. Each joint node is located at the same level as the split node. To streamline linear computation, researchers utilized a specialized tree structure known as a sequential and parallel tree (SP-tree) [9]. The sequential structure positions the child nodes on the left side, while the parallel structure positions them on the right side. In this structure, each SW node is associated with a matching SP-tree node N. When considering its structure, an SP tree can be defined as follows: A child positioned to the left of the N symbolizes the immediate successor that is 0½V in the southwest direction. Software uses split nodes like AND, XOR, and Loop to express split-joint structures. The first node in the software structure initiates a new branch as a right-side child. The structure's direct heir is represented by a youngster on the left side. To symbolize a road that is empty, we build a node that says "Empty" in Figure 7.

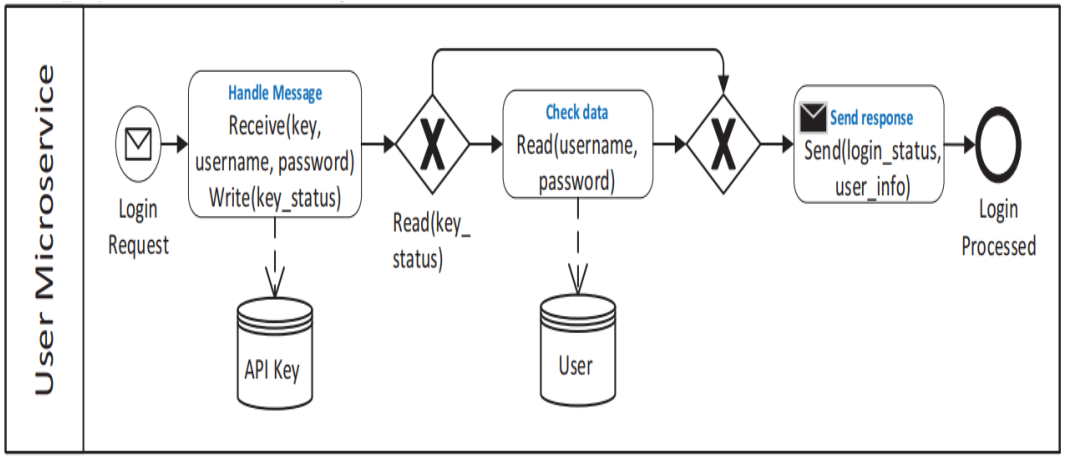


Fig. 6: An example of microservice modeled in SW

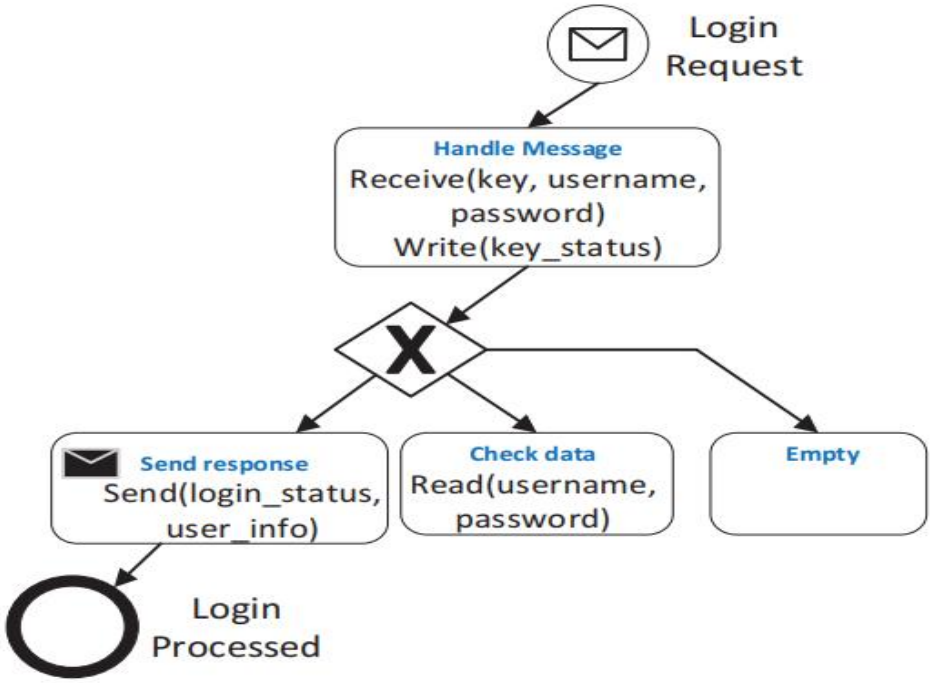


Fig. 7: An example for transforming an SW into a SP-Tree

Detecting Anomaly in A Microservice

Detection of Continuous Anomaly

One way to characterize a continuous anomaly is with a three-tuple (v, m, n). In this scenario, artifact v is used on node m before moving on to node n. Because m precedes n, there is no processing of v between m and n. Each possible combination of m and n operations is an operation pair: (Kill, Read), (Kill, Kill), (Write, Kill), or (Write, Write). Detecting continuous abnormalities in the SP-tree is the purpose of Algorithm 1, which is derived from [9]. The algorithm employs a leftmost depth-first technique, incorporating a recursive procedure to traverse the SP-tree node by node. Once you have arrived at a node: (1) If the node in question is an activity node, the algorithm will gather information about the flow of data. It will then identify any anomalies within the current node. In the case of split nodes, the algorithm will recursively find anomalies within the current node and any child nodes on the right side, using the data flow information. After that, it will either return the data flow from the right side or call its successor recursively if one exists. Upon completion of the task, we examine all the SP tree nodes.

Algorithm 1:

Continued Detection of Abnormalities (SAD)

Application: SP-tree node Produced: SP-tree element

A set of continuous abnormalities switch is a global variable. {

cDVH(³6WDUW´):

The left-hand child must initiate data flow information return SAD.

cDVH(³(QG´);

return node = Null;

cDVH(³$cWiYiW\´);

return ProcessActivityNode(node);

cDVH(³;25´):

return ProcessXorNode(node);

cDVH(³$1D´);

return ProcessAndNode(node);

cDVH(³/RRS´):

return ProcessLoopNode(node);

}

For **ProcessActivityNode** Analysis of anomalies in an activity node primarily entails the following five steps: (2) Locate the assemblage of artifacts that possess either a Kill or a Write operation prior to reaching that specific node, referred to as LK and LW sets, respectively. (3) use this collection to identify any continuous anomalies; and (4) find the collection of artifacts in that node that have the first and last operations, which are called SA sets. (1) call the left-side child recursively; (2) call the left-side child iteratively; (3) update the LK and LW based on the SA sets used in the successor node; and (6) return the data flow for analysis appropriately if there is no left-side child. For **ProcessXorNode** they can divide the process of discovering continuous anomalies into two independent components: (1) the expression, and (2) the branches. This section provides a detailed explanation of both these components. When discussing an XOR structure, there is an operation that involves artifacts; hence, it is possible for an anomaly to occur as a result of its action in the nodes that came before it. Following are the four steps that make up the first part: The process involves four steps: (1) identifying the node's SA sets; (2) determining its LK and LW; (3) identifying the oddities in the XOR node's expression; and (4) altering LK and LW based on the SA sets, thereby identifying the oddities in the XOR branches. There are four steps involved in this process. The three steps that make up the second part are listed below:

Find the problem and get the data flow by calling each child on the right side multiple times; (2) change LK and LW based on the data flow from step (1), which is used to find the problems in the child on the left side. which might be thought of as a successor to the XOR structure in SW. (3) If there is no child on the left side, then the data flow should be returned for processing in the appropriate manner. Otherwise, the recursive call should be performed to the child on the left side.

For **ProcessAndNode** The technique bypasses expressions in AND split nodes and goes straight to the right-hand children to identify the continuous abnormality. The detection process consists of four steps: Using the flow data gathered in step (2), they proceed as follows: First, they determine the LK and LW for each child on the right side of the AND node. Next, we check for any anomalies and collect data through mutual recursive calls. Finally, we update the variables LK and LW. Next, make a recursive call to the child on the left side. If there is no child on the left side, return the data flow for analysis in the appropriate manner. [10]

For **ProcessLoopNode** Combining all iterations into one reduces the need for duplicate computation due to the loop body's recurring structure and control, making iterating more efficient for finding continuous abnormalities in a loop. Eliminating unnecessary computations will help achieve this. They categorize

continuous anomalies associated with a loop based on the location of the two processes that make up the loop. This is because two actions taken on the same artifact constitute a continuous anomaly. If the first operation takes place outside of the loop and the second occurs inside the loop, we can refer to the anomaly as pre-loop-end. They refer to the anomaly as an intra-loop anomaly when both procedures occur inside the loop.

An irregularity with a first operation occurring inside the loop and a second operation occurring outside of it is known as a tailed-loop anomaly.

The following six steps comprise the formula: Start by obtaining the loop node's SA sets. 2. Acquire LK and LW components. 3. Look for any irregularities in the loop node before it ends. 1. 4. Repeatedly invoking the right-hand child will return any abnormalities in the loop body, whether they occur before or after the loop ends. 5. Update LK and LW to identify the tail loop anomaly. 5. Using a recursive function, call the left-hand child. Take the data flow back to analysis if the left side does not include any children, return the data flow for analysis.[11]

Detecting of Concurrent Anomaly

If two nodes in the SW are close to each other (m and n), they can run at the same time. The NCP that came before this one is an AND split node. Using an AND split node for each of the two nodes m and n, we test artifact V extensively. Consider the following as a three-tuple: (v, m, n). Hence, it has the option to select (Write, Write), (Kill, Read), or (Kill, Write) as the pair of processes in m and n. Using Algorithm 2, they can identify concurrent anomalies in an SP tree by employing a leftmost depth-first strategy. [12]. This algorithm uses a recursive approach to explore the SP tree node by node. When the algorithm reaches a node The algorithm will return a union containing the artifact(s), their operation, and any other elements within the current node. This set, named a_set, not only returns the value from the recursive call to its child, but also returns the value itself. On the right side, there is a separate collection known as branch_a_set that is associated with each AND node's distinct branches. This set encompasses all a_sets derived from the top node to its final successor in this branch. After gathering all the branch_a_sets, the algorithm scrutinizes each pair of branch_a_sets to identify any contemporaneous anomalies.

Algorithm 2:

Detection of Concurrent Anomalies (CAD)

Application: SP-tree node Results: a_set{}

International variables: collection of simultaneous outliers

When the node type is not Start, End, or AND, then gather all objects and their operations and save them in

a_set;

}

switch(type of node) {

cDVH (^{3}Start´):

return CAD(child in the left side);

cDVH (^{3}End´):

return {};

cDVH (^{3}Activity´):

if(child in the left side exist){

return (a_set ω CAD (child in the left side));

}else{

return (a_set);

}

cDVH(^{3}Loop´):

a_set= a_set ω CAD (child in the right side);

return (a_set ω CAD (child in the left side));

cDVH(^{3};25´):

for(int i=0; i<a number of immediate child in the right side; i++) {

a_set= a_set ω CAD (ith child in the right side);

}

return (a_set ω CAD (child in the left side));

cDVH(^{3}$1D´):

branch_a_set[];

for(int i=0; i<a number of immediate child in the right side; i++) {

branch_a_set[i] = CAD (ith child in the right side);

a_set= a_set ω branch_a_set[i];

}

for (int i =0; i<branch_a_set.size; i++) {

for (int j=i+1; j<branch_a_set.size; j++) { Detect the concurrent anomaly by comparing branch_a_set[i] and branch_a_set[j];

}

}

return (a_set ω CAD (child in the left side));

Algorithm 2's operations: When the algorithm reaches a node, it will first determine the type of node that it is currently examining and then perform a detection that corresponds to that kind. When the node is a Start, the algorithm calls the child on the left side of the tree, which alternates between levels. The answer is an empty set, indicating that the SP-tree for the end node has ended. This method takes an activity node and returns its A_set plus the value of the recursive call to its leftmost child, if any. If the activity node does not have a child, the procedure directly returns its a_set. [13]

The method initiates a recursive call to retrieve all the a_sets from the child on the right and its successor, if any. The method then performs a union operation on all of these sets. Next, the method joins the values that were returned with the a_set of the Loop node. A loop node, for example, could handle this. The procedure finishes by returning the union of the current a_set and the value received by a call to the child on the left that proceeded backwards, taking care of the child on the left. To create a union, the technique takes all the a_sets from the children of an XOR node and puts them together. The recursive call collects all the a_sets from each direct child on the right and any subsequent children. Next, it merges the returned value with the a_set of the XOR node. Finally, it takes care of the left child in a way that is similar to what happens in the loop node [14].

It functions similarly to the XOR node, joining all the a_sets associated with each child on the right side of the AND node. It makes a recursive call for each direct child on the right. This call retrieves all a_sets from that child and its successor, if any. We also save the returned number in a separate branch. It will figure out the ongoing anomaly in each pair of branch sets once it has dealt with all the children on the right side. The left kid is taken care of last, like a loop node. It's [15]

CONCLUSION

When it comes to financial institutions and fintech companies, using a microservices design is essential because the financial sector is always changing. This method has many advantages, including more scalability, better fault separation, faster development, and more customization. Additionally, it ensures the framework meets strict security and compliance standards. Transitioning from a monolithic architecture to a microservices architecture significantly enhanced the financial services platform's performance, scalability, and stability. The use of Kafka for asynchronous processing and HTTPS/WebSockets for communication ensured a more resilient and efficient system, capable of handling high loads and providing a superior user experience. To summarize, the incorporation of microservice architecture is not only a technological movement, but rather a transformative journey for financial institutions. In the dynamic financial industry, it is crucial for attaining operational excellence, increasing client delight, and maintaining a competitive advantage. Businesses may secure their future growth and success in the digital age by adopting this architectural paradigm, which puts them at the forefront of financial innovation. Microservice architecture can provide significant advantages to financial service sectors as they undergo digital transformation. Distributed microservices, each operating independently as its own process and interacting with other microservices through message-based protocols, can compose a microservice-based financial application. During the application design process, it is critical to analyze the artifact operations that occur within each microservice. An anomalous artifact operation can lead to an inaccurate artifact state, potentially corrupting the entire application. This article delves into the attributes of artifacts within a company's microservice design, taking into account the characteristics of microservices. These characteristics provide a method to prevent the system utilizing the microservice architecture from entering a state of failure. In order to guarantee that the artifacts sent to other microservices are accurate, this method employs the detection of artifact abnormalities within each microservice. They can categorize the intra-microservice artifact anomaly into two types: continuous and concurrent, where two sequential or parallel operations on the same artifact lead to abnormal behavior. We present a method to detect intra-micro service artifact anomalies with a SP-tree structure for each micro service. The presented methods show that they can detect intra-microservice anomalies.

References

1. https://medium.com/firstlineoutsourcing/microservices-architecture-in-financial-systems-benefits-challenges-and-use-cases-b388ed01f8a3

2. https://forbytes.com/blog/digital-banking-architecture/

3. F. J. Wang and F. Fahmi, "Constructing a Service Software with Microservices," in the Proceedings of 2018 IEEE World Congress on Services (SERVICES), pp. 43-44, 2018.

4. S. Newman, Building Microservice: Designing Fine-Grained Systems, O'Reilly Media, 2015.

5. F. Fahmi, P.-S. Huang and F.-J. Wang, "Improving the Detection of Sequential Anomalies Associated with a Loop," in the Proceedings of 2019 IEEE 43rd Annual Computer Software and pplications Conference (COMPSAC), pp. 127-134, 2019.

6. S. X. Sun, J. L. Zhao, J. F. Nunamaker, and O. R. L. Sheng,374-391, 2006.

7. P. S. Huang, F. Fahmi, F. J. Wang, "Improving the Detection of Artifact Anomalies in a Workflow Analysis", under submission review.

8. N. Dragoni, et al., Microservices: Yesterday, Today, and Tomorrow, Present and Ulterior Software Engineering, pp. 195-216, 2017.

9. N. Dragoni, et al., Microservices: how to make your application scale. International Andrei Ershov Memorial Conference on Perspectives of System Informatics, pp. 95-104, 2017.

10. T. Erl, Service-Oriented Architecture: Analysis and Design for Services and Microservices, Prentice Hall, 2016.

11. D Temporal Structural Workflow Computer Software and Applications Conference Workshops 2014 IEEE 38th International, pp. 480-485, 2014.

12. Chen, P. Qi, Y. Hou, D. Causeinfer: Automated end-to-end performance diagnosis with hierarchical causality graph in cloud environment. *IEEE Trans. Serv. Comput.* 2016, *12*, 214–230.

13. Lin, J.Chen, P. Zheng, Z. Microscope: Pinpoint performance issues with causal graphs in micro-service environments. In Proceedings of the International Conference on Service-Oriented Computing, Hangzhou, China, 12–15 November 2018; Springer: Berlin/Heidelberg, Germany; pp. 3–20.

14. Chen, H. Chen, P. Yu, G. A framework of virtual war room and matrix sketch-based streaming anomaly detection for microservice systems. *IEEE Access* 2020, *8*, 43413–43426.

15. Meng, L.Ji, F. Sun, Y.Wang, T. Detecting anomalies in microservices with execution trace comparison. *Future Gener. Comput. Syst.* 2021, *116*, 291–301.