Maya Hussein

CSCE-3701-01

October 2022

# Architectural Patterns

# 1 Introduction

The software architecture of a system illustrates the design of the system's overall structure and behavior, allowing the stakeholders to gain better comprehension of their system ilities and tradeoffs. Since most systems revolve around sourcing similar purposes, architectural patterns were designed to provide known solutions to most common problems. These patterns include but are not limited to: (a) Microservices, (b) Saga, (c) Strangler, (d) Throttling, (e) Sharding, all of which will be discussed in this paper.

# 2 Microservices Pattern

A Microservice architecture aims to distribute its services on the cloud using an interface. Unlike monolithic applications, a microservice architecture is designed as separately deployed units, which allows for higher scalability, better decoupling, and swifter deployment through pipeline delivery. Microservices architecture was issued mainly to allow for continuous delivery. Monolithic systems were tightly coupled as a single-deployment unit, hindering maintenance, constant updates, and deployment of the software. Because of the microservices multiple deployment units, components can now be programmed, tested, updated, and deployed individually without having to en-

tirely affect the other service. Moreover, microservices architectures resolve the service-oriented architectures complexity and difficulty of implementation by simplifying the connectivity and accessibility to service components. [1] As shown in the diagram, the architecture acts as a distributed system,

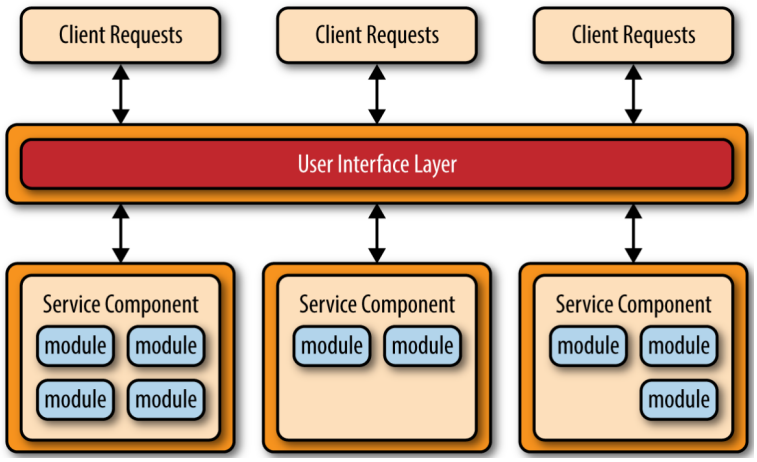


Figure 1: Microservices Architecture Diagram [1]

where each component is deployed as a separate unit. Every component can contain a module or several modules that interact with one another. It is also noted that a component interacts with other components through the interface. Because of its component singularity, microservices have higher agility and scalability, easier maintenance, and independent deployment, which in return allow faster release into the market.[2] However, these advantages do come with tradeoff, including a higher degree of complexity, high network traffic, and therefore limited reuse of code, difficulty in global testing, points

of failure. [1]

For its high agility and scalability, microservice architecture can be utilized in (a) internet of things, as it provides small scale and lightweight applications that are easily manageable, (b) cloud computing as the architecture supports deployment in small units, and (c) web applications as it provides high scalability. For instance, multi-national corporations such as Amazon, Netflix, and Uber currently adopt microservice architectural design. Amazon initially utilized service-oriented architecture; however, after instantiating services like Amazon AWS and Apollo and shifting to microservice architecture, its market worth has become $1.6 trillion as of August 2022. Additionally, Netflix has used Amazon's AWS service, which was deployed using microservice architecture. In 2013, Netflix's API Gateway handled around 2 billion API requests daily, which were managed by 500 cloud-hosted microservices, and by 2017 the architecture has hosted around 700 microservices. In return, Netflix makes around $8 billion a year. Interestingly, Uber has transitioned from a static mySQL database, and a monolithic architecture into microservices through an API Gateway. As a advantage of the transition, programmers were able to fix issues faster, and without affecting other components, improving the speed, quality, and deployment rate.[2]

# 3 Saga

Saga is a variant of the microservice architecture that manages data consistency across its services in distributed scenarios. Saga uses a sequence of transactions that updates each component and then sends a trigger to drive the next event.[3] In case of event failure, different transactions are produced to replace the failing node. A transaction is a unit of logic, where an event occurs to a specific entity, and its command is used to store all its required information for execution or triggering the next event. Because distributed systems requires all participants to commit before a transaction proceeds, which is not always supported in databases, Saga offers transaction management using local transactions, which is executed by one participant allowing pivot to occur to distributed microservices. [5]

A method to coordinate transactions with multiple participants without a centralized node is choreography. Choreography allow each transaction to direct events that drive other transactions in other components.[4]

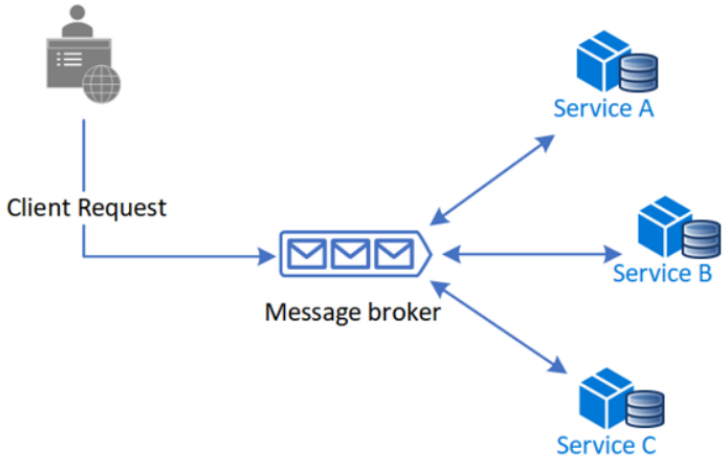


Figure 2: Saga Architecture Diagram
[4]

The diagram shows how saga choreography can be implemented on a system. An event is first requested by the client. The broker then drags the request and publishes a transaction. This transaction then drives another local transaction in another service. Afterwards, as shown a response is reported from this service's local transaction into another transition using choreography. [3]This technique is useful for software that require a limited number of participants. Because the tasks are distributed across several participants it limits the points of failure without requiring any additional implementation or maintenance. On the contrary, the transition between commands can be confusing for participants as it might be difficult to identify the transaction is for which service.[3] Moreover, testing might be complex because all components must run to stimulate a transaction. Databases like Amazon Dy-

namoDB, Google Workflows, and Uber all utilize saga in their microservice architectures as means to transition between transactions and components without having the microservice's databases faults. [5]

# 4 Strangler

As time progresses, systems become harder to maintain as the development tools and technologies are overruled by newer technologies and on the contrary, the systems become more complex with higher scalability and external factors making it complex to add new features. Strangler allows the software to replace certain functionalities with newer services using a façade that accepts requests from old systems. This façade directs the requests either to the legacy or the new component. Slowly, all existing features can be added into the updated program while consumers are using the interface. [6]

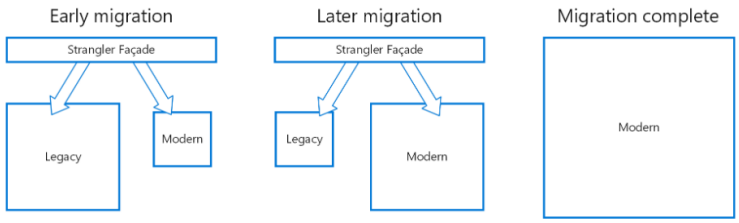


Figure 3: Strangler Architecture Diagram
[6]

The diagram shows how features are deployed over time in stages where the

strangler façade is redirecting the features to newer components gradually. It is also noted that after migration stages, the last model captures the all the new features from all stages. This architecture reduces the risk of failure as the migration occurs over an extended period of time. It also ensures that the system functions normally on the old components as new services are deployed.[6] However, the system requires constant and careful routing will transitioning from the legacy to the updated system. It also requires a rollback plan if the new feature fails as the system needs to be able to go back to the old feature.[7]

Strangler architecture can be used to implement object-oriented systems as it can allow class migrations swiftly and without significant loss of data. It is also used in database most corporations need to run a second backup system that can track and maintain the data. Whenever an update is made to the legacy system, the system records these changes and reports them to the consumer. If enough updates are made, the admin can route the system directly into the second database. Moreover, strangler figs are also utilized in web applications as it can transform Java server-based services into dynamic web-based apps. The legacy code will eventually be transformed into web services, and the server-based application can be completely disconnected.[7]

# 5 Throttling

Throttling architecture is used to handle varying load on the cloud. It is used to control the rate of data flow into the target. Also known as Rate Limiting, the architecture allows the system to use resources up to a certain limit and throttles the remaining. Some of these strategies include rejecting the requests or degrading the functionality. [8]

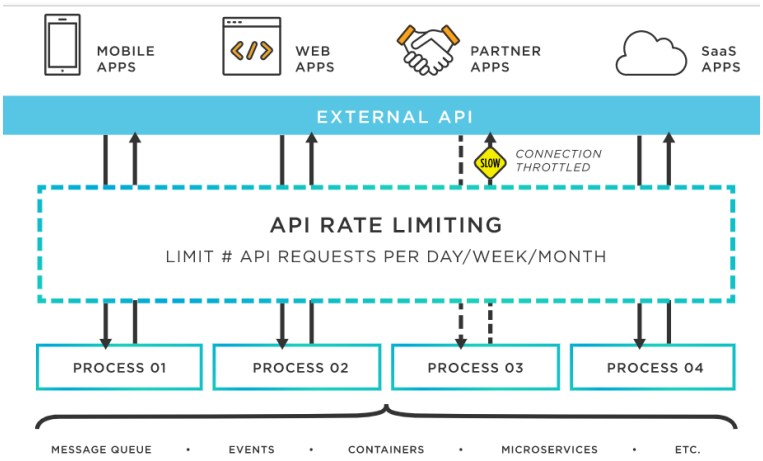


Figure 4: Throttling Architecture Diagram [8]

As shown in the diagram, the API Rate Limiter restricts the amount of requests that are being processed into the system. Also note that the throttling connction can also slow down the data until a request is handled to enhance the reliability and usability of the system. All requests must pass through the API before the process to ensure that the system is not overwhelmed. As a result, the architecture prevents major points of failure and attacks as they need to pass through the API first. It is also used to restrict expenses by applying usage rates. A throttle ensures that the maximum target is reached and is not overpassed. On the other hand, if the system is not redundant, the system can be degraded when the targets are slowed down because of the throttled connection. Moreover, not any engineer is capable of enforcing this architecture, specific expertise is required. It is also difficult to implement in isolation as in needs constant connection to the API. [8]

Some applications of this architecture include implementing throttle over a database as it ensures the maximum targe is reached. Moreover, the throttle API is used in Meta on Instagram using Instagram Basic Display API implementing throttling architectrure[9], Shopify which uses Shopify API that is based on a throttling algorithm with graphQL[10], and OAuth of Google Support which has a standard rate limit and can be increased through requests that take up to 5 business days.

# 6 Sharding

When data is hosted on a single database, the server might face storage limitations, or lack of sufficient computer resources. It can also be affected by network bandwidth or its geography. Sharding aims to divide the data into sections (shards), where each has the same data schema, but on a different dataset. A shard is a data-storing unit that can contain different types of data for many entities acting as a storage node. Because of the partitions, sharding ensures that the data is organized . Ensuring that sharding logic directs the application to the appropriate shard, the logic can be implemented by the data-storage system or the data access code of the application. [12]

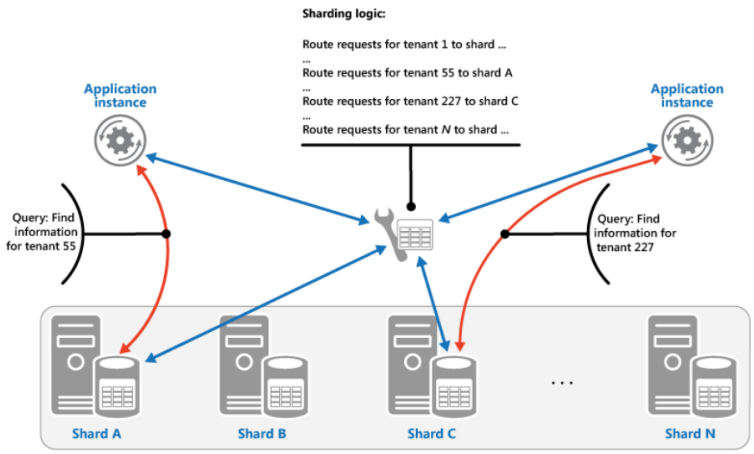


Figure 5: Sharding Architecture Diagram [12]

The diagram shows how all application instances are directed to the shard-

ing logic node which in return sends the event to its appropriate shard for execution. This creates a level of abstraction which as shown supports a high level of control on the instances. Such features allow system scaling by adding new shards running on additional storage nodes. It can also improve performance by load balancing as the sharding logic distributes the events among shards. Moreover, on the cloud, shards can be allocated closer to its users accessing the data. [11]

Consequently, sharding is used in data encryption with recognition to ShardingSphere Proxy, which encrypts and decrypts data. Applications do not normally need any code refactoring; the code is simply sent to the Proxy in plaintext where it is encrypted and later sent to the database. It is also used in distributed database systems, where users interact with a Proxy to reach the logic table that is sharded into many servers. Most importantly, it supports social applications such as apps hosted by Meta. Teams there were already designing their own customized solutions using a shard manager as a generic platform to be able to handle points of failure, but with minimal load balancing, which led to optimal reliability, and operation overhead. However, because it is a solution to most database scaling issues, it can add complexity to the system, and as a result become cost inefficient. It can also compromise the databases' referential integrity as database management systems don't

always support foreign keys across different database servers. [11]

# References


[1] Mark Richards. *Software Architecture Patterns*. O'Reilly, 2022.

[2] *Microservice Architecture Style*. Microsoft Azure, 2022

[3] *Saga Distributed Transactions Pattern*. Microsoft Azure, 2022.

[4] Mehmet Ozkaya. *Saga Pattern for Microservices Distributed Transactions*. Medium, 2021.

[5] Chris Richardson. *Saga*.Microservices.io. 2021.

[6] *Strangler Fig pattern*. Microsoft Azure, 2022.

[7] Marty Abbott. *Strangler Pattern*.AFK Partners, 2019

[8] *Throttling Pattern*. Microsoft Azure, 2022.

[9] Zameer Masjedee. *Implementing API Rate Limits in Your App*. Shopify, 2021.

[10] *Rate Limits*. Facebook Developers, 2022.

[11] Rakesh Kalra.*Disadvantages of Sharding*. Think Scholar, 2019.

[12] *Sharding Pattern*.Microsoft Azure. 2022.