International Journal of Multidisciplinary Research and Growth Evaluation.

Navigating Software Complexity: Guidelines for Choosing Scalable Architecture Styles

Sachin Shridhar Padhye
Katy, Texas, U.S

* Corresponding Author: **Sachin Shridhar Padhye**

Article Info

ISSN (online): 2582-7138
Volume: 06
Issue: 03
May-June 2025
Received: 15-04-2025
Accepted: 16-05-2025
Page No: 1620-1626

Abstract
Architecture is broadly defined as a science of designing and building structure. It is process of planning, designing, and constructing the structure. In software field structure is the software application. In simple term, Software is computer programs and data that tells computer to perform series of actions using underlying hardware. In real world, there are multiple computer programs executed in chronological order. In real world, software applications which are solving any business requirement is very complex where multiple software applications, commonly called as building blocks are integrated with each other. Software architecture is encompassing the high-level organizations of its components, how they interact and principles guiding their design. This paper present guidelines on architecture where data integrity is extremely important. Correct and adequate data should be presented and maintained in application.

DOI: https://doi.org/10.54660/.IJMRGE.2025.6.3.1620-1626

Keywords: Software Architecture, Microservices, Data, Webhook, gRPC, Kafka, Messaging System

1. Introduction

As per IEEE-1471, software architecture defined as fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution. Architecture style gives generic framework which standardize the software architecture. Architecture style defines organization of the components and communication between them in particular way. It serves as a high-level blueprint for the system, influencing its overall design and behavior. Software application is developed implementing various architecture style and architecture pattern. E.g. Typical Web application uses Layered architecture style, implementing MVC pattern, where it has separate frontend service, data bases and back-end services where business logic is written which controls flow of the data from front end to database and vice versa with processing as per the business requirement. But there could be one monolithic backend service or multiple microservices. Also, within the microservices communication can be handled by peer-to-peer communication or publish-subscribe pattern. In this paper we are going to discuss architecture style to overcome the limitation of the architecture style.

In first part, we discuss how to approach for microservice architecture style. Each architecture has it's characteristics which may be suitable at stage of the business. When any business is start up that time development team is very small and number of user of the applications are small, also there is uncertainty on success of the business. If organization goes implementing the micro services, it may be impractical to implement microservice architecture with limited resources and budget. Alternative approach is to start with monolithic architecture style and move slowly towards micro service architecture. It also recommends best practices for communication.

In second part, paper discuss about architecture pattern which try to solve "Timeout Antipattern". Short time out may fail legitimate requests prematurely and long timeouts can result into poor user experience and delay error responses. This issue occurs when Synchronous request implemented through event driven architecture style or Asynchronous manner.

Part-I

Typical software application has three main component a) User Interface b) Backend Services c) Databases. Software Architecture Style standardizes how these components are organized and connected to produce reliable, scalable software application. There is other components of the software architecture e.g API Gateway, load balancers etc but for this paper we consider above three components. As shown in below figure there are two main software architectures style. A) Monolithic Architecture B) Microservice Architecture.

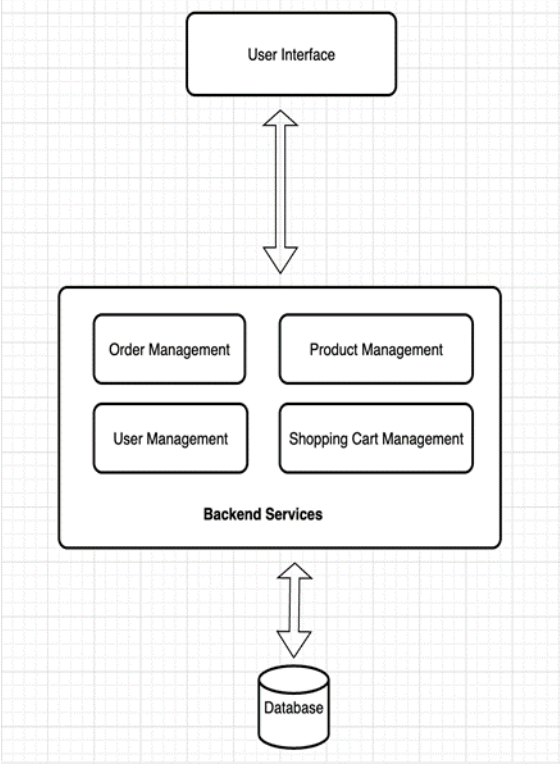

Fig 1: Monolithic Architecture

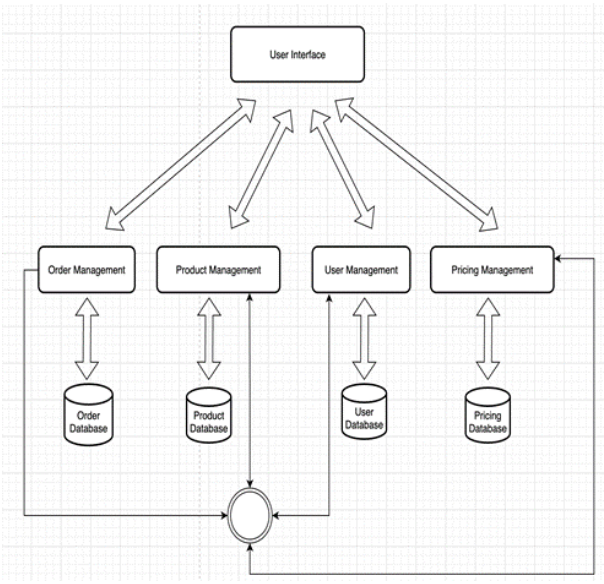

Fig 2: Microservice Architecture

Software application can be defined as a process which collects the data either from the user or other application, applies the business rules on the data which changes the data and then persist the data. Basic functionality of an application is accessing the persisted data (read) or create new data or updated existing data (write), which are known as read and write operations respectively. The place where data is saved in called database. Database can be categories in different types, the way data is saved, E.g. In memory database, relational database, NoSQL database. Base of the software application is database connectivity; this drives the performance of the application. For this paper we will consider relational database. Each application needs connection to the database to read and write data. Each database allows maximum number of connections, also max number of connections on single node is driven by hardware resources (CPU,I/O, Memory). So this Max number of connections directly dictate number of concurrent requests application can serve. E.g. Oracle database can handle 1528 concurrent connections, SQL servers 32,767 and My SQL has default 151 but it can be set through my_connections settings. Hence max number of database requests can be concurrently executed is number of maximum database connections allowed by the database.

In monolithic architecture these connections are shared among the multiple backend services so number of concurrent requests can be executed is less than or equal to available max connections. Additionally multiple backend process can access same table and try to update same row in different request which puts lock on table row, hence even though free connections are available one of the requests must wait until another request releases the lock. In complex applications and use case some time it may cause dead lock.

Microservice architecture solves this problem, each service connects to its own database, hence number of requests can be executed concurrently equal to number of concurrent connections database can handle. Also intra process communication is handle through SAGA pattern like orchestration or choreography, hence not a possibility of database row lock.

Microservice has a bigger landscape, it comes with its own challenges. Sometime larger concurrent connections available are not even required during initial life of an applications. As studies shows a typical user clicks 7-8 clicks per seconds, and each clicks generates API call. During initial life of an application there are 50 concurrent users then it will generate 350-400 requests per seconds, which means every request is made in approximately every 2.87 milliseconds. For ~400 concurrent requests, 400 concurrent database connections are required, so with Oracle database ~73% of connections will be unused. Also, during initial life of a software product, i.e. in startup environment, development team size is small. Hence it is recommended to start with monolithic architecture development and migrate to micro service architecture during later life cycle.

While designing monolithic application, if it is kept in mind that it could be split in multiple micro services then it will be less effort during redesigning or restructuring the application. The principle we are proposing is to design monolithic application consist multiple modules. Module refers to business services, each module will have its own database schema, and each module will communicate with each other through messaging queue. Technically, multiple modules are built, packaged together and deployed as a single package. But down the line when it needs to be decomposed in microservices then only packaging changes. Database is independent of the application, it will be as simple as copying

entire schema to new database instances. Below diagram shows modular monolithic architecture.

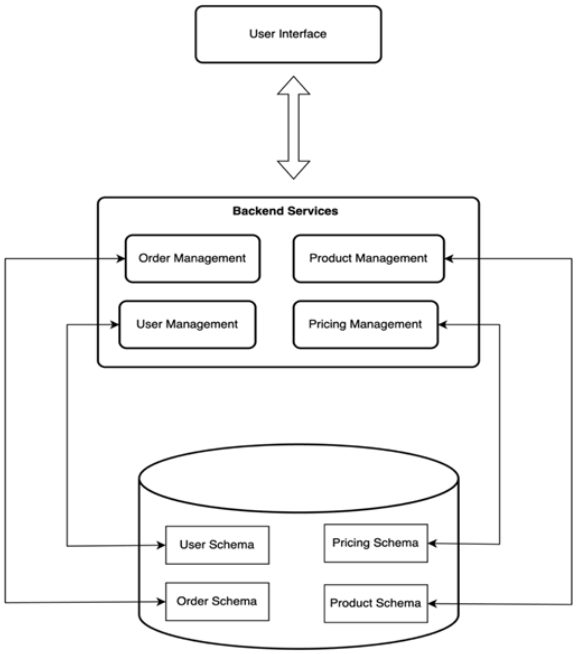

Fig 3: Modularize Monolithic

In ideal microservice word, one table per database is expected but it is very difficult to implement in complex software application. Hence some guidelines recommended to design the database.

Data consists of multiple attributes; Data is changed when one or more attributes of the data is updated. Based on how frequently data is changed in the lifecycle of an application it can be broadly categories in two types a) Inconstant Data b) Stable Data.

Inconstant Data changes frequently by user action. Each Inconstant data is updated based on business rules and set of stable data. Example of the inconstant data is Orders created in e-commerce website; shipments created for custom clearance.

Stable Data doesn't change very frequently. Inconstant data changed into stable data over lifecycle of the application. Stable data served as a master data for an application. Example of the stable data is Products and Sku available on website for purchase.

There are few properties of the Inconstant Data and stable Date.

- Inconstant Data eventually becomes stable data over life cycle of an application. e.g. When customer places an order on e-commerce platform, order can be modified after submitting it up to certain time frame defined by the business process. Customer can cancel the order or order new product until order is shipped. Once it is shipped and received by the customer it cannot be updated.
- Inconstant Data can be created from the stable data. E.g. Order is created from the product data. It is recommended to copy the stable data in inconstant data to make sure new version of the stable data won't impact inconstant data.
- When stable is changed, new version of the stable data is created instead of modifying the properties of the stable data.
- Stable data changes very rarely hence they are mostly read only. This makes them good candidate for the caching.

First step is to decide how this data is classified, if they are under different classification then they can be in separate schema. E.g. Order is transactional data modified by the customers more frequently hence it classified as Inconstant data vs products purchased by the customers are very rarely changes hence it can be classified as a stable data.so both Order and Product will be in their own schema. Below table classifies most common data objects in e-commerce domain.

Inconstant Data	Stable Data
Order	Product
Items	Sku
Inventory	Price
Shipping Group	Promotions
Payment	Coupons
	User

Secondly, next questions need to be asked is that if two data elements in same classification are related, if answer is yes then it can be in same schema and realized as a table structure. If they are not, then it can be in separate schema. e.g. Inventory and order are not actually related, inventory is maintained against the product and as orders are placed inventory is decremented so it can be kept in separate schema. Same thing is for Product and User, hence product and user can be in the separate schema.

Lastly, to check business specific functional requirement to decide. E.g. for some organizations pricing of product changes very frequently like 2-3 times in day, few organizations run very frequent promotions, hence better to separate out in the separate schema. Based on this different schema can be created as follow.

Order Schema	Inventory Schema	Product Schema	Price Schema	User Schema
Order Items Shipping Group Payment	Inventory	Product Sku	Price Promotion Coupon	User

Common Library

Tools and framework put a structure around the application code which makes application consistent, reduces the duplicate work. Micro service architecture is based on the principle of the duplication as each service is independent and managed by separate team but if there are basic framework or pattern is available then building new service just few minutes work. Also, code is structured in specific pattern, so it becomes maintainable. E.g. 100+ services each with different style of code vs 100+ services with similar pattern, it allows the shuffling of the developer among the team as per the requirement and bandwidth. Such tools and frameworks can be maintained as separate library.

Object Relation Mapping (ORM) Tools

Object Relation Mapping tools is the abstraction layer sits on top of the relational database. It converts relational data in Objects so that it can be easily accessed in service layer. It handles SQL statement creation and execution. Software developer just focus on writing business logic and code to

write getting SQL connection, creating SQL statements, mapping data from the DTO to SQL statements, executing statements, mapping query result back to DTO is handled by the OR tool. ORM tools provide basic API for Create, Read, Delete, Update (CRUD).

There are few missing elements of ORM

1) Dynamically creating Search criteria for listing API. One of the common requirements of the application is to search and filter the items based on user provided inputs. OR tools API generally requires defining method in interface to find results on search criteria. If there are 10 filters on item, then it is impractical to write so many methods. OR tools provide a way to construct the query using Criteria Builders or query language. E.g. Hibernate provides Hibernate Query Language (HQL). Library can be written to convert System for Cross-Domain Identity Management (SCIM) filter for selecting item in dynamic query.
2) Using ORM Tool API, one update statement is executed per item which may update multiple columns. But there could be requirement of updating 50-100 rows of a table base on some condition. E.g. There is a requirement of consolidation of shipment, there 50 shipments with multiple line items, these line items need to be consolidated in new shipment, also it can be removed from consolidated shipment and when it is removed it needs to go back to original shipment. Hence, we need to update 200 lines items with new shipment id and maintain reference to old shipment id. There are two ways of doing this one is to execute 50 SQL update statement vs execute 1 SQL statement with CASE-WHEN option. Executing 50 SQL is not a viable solution because it's time complexity of O(n), meaning as number of rows increases it will increase number of statements to be executed, whereas on other hand second solution will always execute 1 SQL statement. Hence there is a need to provide a library which will construct complex SQL statements.

Microservice Communication

Important aspect of the microservice architecture is intra service communication. There are two main type of the communications 1) Point to Point communication 2) Multipoint communication.

Point to Point communication

In this type of communication service sends message or requests directly to another service. This type of communication is typically used to pull the data from the other service, or some action needs be performed before performing other action i.e. Synchronous communication. E.g. When customer places an order, it needs to submit order service needs to check if inventory is available to accept an order. gRPC is popular communication protocol used for point-to-point communication.

gRPC is a high-performance, open-source, universal Remote Procedure Call (RPC) framework developed by Google. gRPC uses HTTP/2 and data transmission is done in byte stream, offering a lightweight and efficient communication mechanism**.** gRPC protocol works on Server / Client communication model. Server owns the service, and it publish contract which is used by the client. There are two ways by which this can be adopted. One way is creating proto library per microservice which will be added by the caller microservice. Another way is to create one library for all service, and it is included by every other microservice. As shown below, there five micro services and five proto libs. If service-2 wants to communicate with service 1 then it will include service-proto-lib-1. Also, if service-5 wants to call service-1 and service -2 then it will include service-proto-lib-1 and service-proto-lib-2.

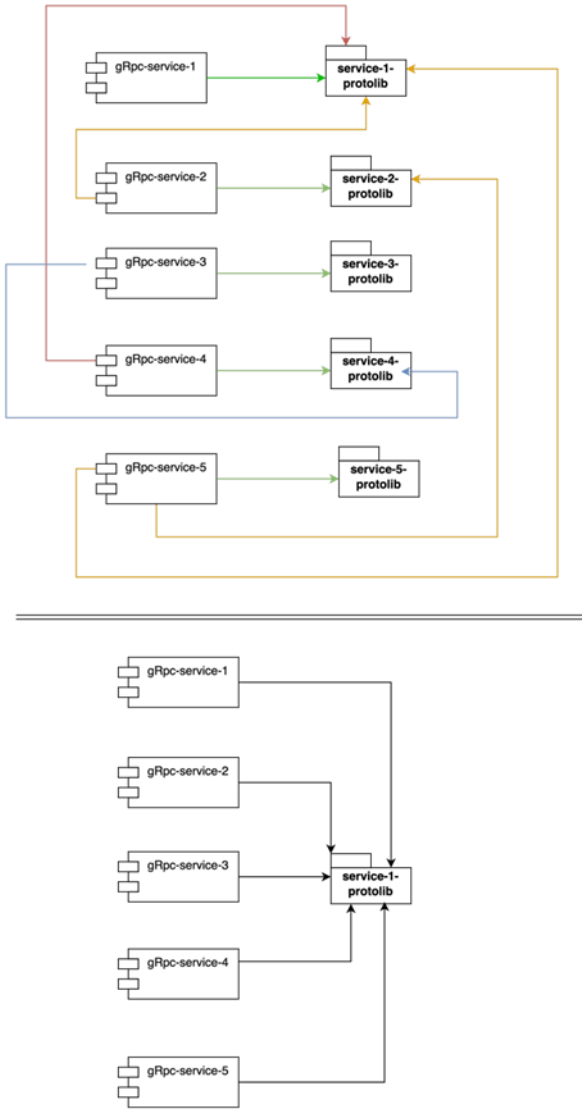

Fig 4: Proto lib per Microservice v/s One Proto lib for all service

There are several advantages of maintaining proto library per microservice.

1. Each proto lib is own by corresponding micro service, so ownership of the library is clearly defined. In case of the other approach where one library approach library owns by multiple teams.
2. In one library approach, as application grows size of the lib increases which increases build and deployment time of all the service, which is not a case in multi library approach.
3. Redundant information is included in each microservices in case of the one library approach. E.g. Microservice-1 doesn't need to communicate with microservice-5 but it will sill include microservice-5 proto and client details.
4. One library is strong coupled solution where entire

application service definition and descriptions are available to all the services.

Multipoint communication

In this type of communication one service publishes the message and other services are subscribed to read the message. Generally, there is a messaging system which may have topics or queue or both. Difference between topic and queue is that topic can have multiple subscribers whereas queue has only one subscriber. Subscribers will receive message and process it. E.g. After order is confirmed successfully, Order service publishes message. Email service subscribe it to send email order confirmation email, oms-integration service subscribes it to trigger order transformation process. Kafka, rabbit MQ is popular messaging queues used for multipoint communication.

There can be two approaches available. In first approach each service will have two topics, in-topic and out-topic where each service is message producer which produces message on out-topic, also each service is consumer of in-topic. Then there will be centralized service which will consume out-topic from each service and then use config map to forward message to destination in-topic from each service. This approach is based on Choreography Saga design pattern.

In second approach, each microservice will be listening to dedicated Kafka 'service-topic'. If one service needs to communicate to other services, then it must pragmatically produce message to corresponding topic. This is tight coupling approach which follows Orchestration saga pattern of the microservice. Implementing new business requirement needs code changes in multiple services.

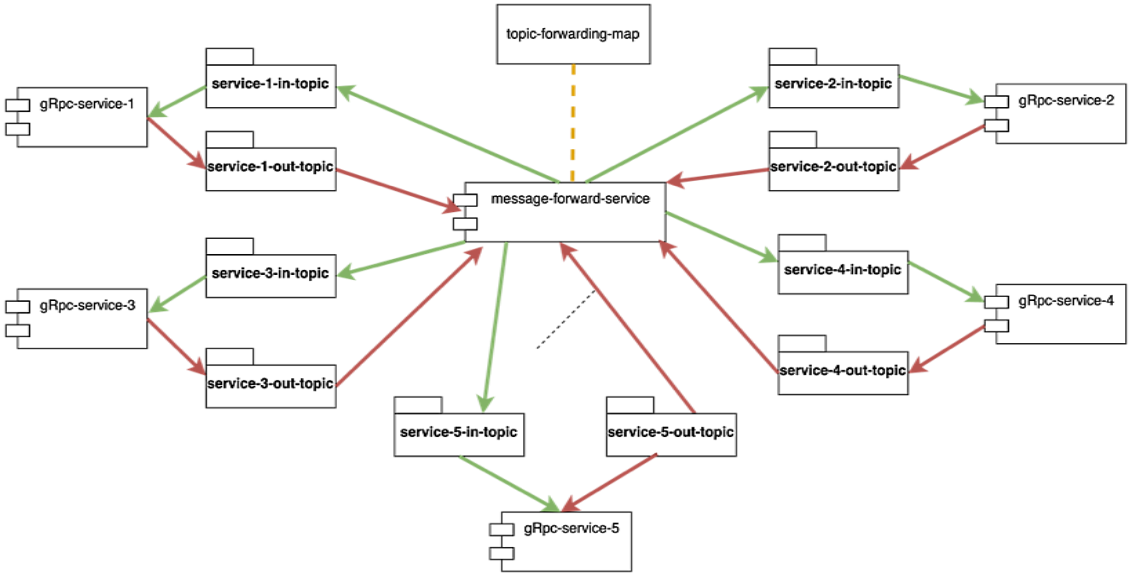

Fig 5: Multipoint communication 'Choreography Saga' pattern.

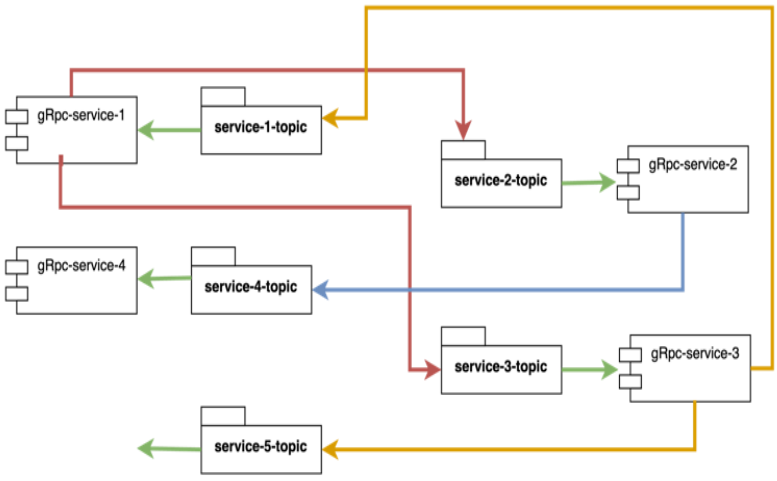

Fig 6: Multipoint communication 'Orchestration Saga' pattern

Part-2

At very high level any data driven web application has four type of requests which are Create, Read, Update and Delete requests. Read and Delete requests are very simple and efficient as they work on unique key. Create and update requests can be complicated as it may requires some calculation and derive some additional data. E.g. In e-commerce when customer increase quantity of an item then application must check, is there any applicable promotions or discounts qualified then apply the promotions and recalculate the order price. Setting a request time out is common solution which is anti-pattern because if time out is small then legitimate request fails prematurely, in event driven architecture style it may continue processing and updates data

behind the scene but caller may get API failure, on other hand if time out is at higher side then API will be long running Either ways long or short timeouts brings bad user experience because user must wait on the UI for longer time and failure. The best design principle is segregate user entered data vs calculated / derived data.

To give better user experience, Idea is to make API call asynchronous, using the web socket connection browser will get response back once response is created and processed successfully. Below figure shows the life cycle of the Async response processing.

As shown in Fig 7, below are the steps which will be executed in the lifecycle of the request.

1. Once customer is logged in user will have unique customer id assigned, then web browser will open WebSocket with backend service. Backend service will save WebSocket session in cache. Cache stores WebSocket session against customer id.
2. Web browser will send individual request to individual micro service along with customer id. Microservice will create token for the request and send it back to web browser.
3. Once microservice is done processing the request it will produce the response and send it to other micro service to log it and WebSocket service with token and client id.
4. WebSocket service will receive the response message and fetch the WebSocket session based on the client id and sends the response back to web browser and send message to request / response management service to mark it delivered or clear.
5. Once Web browser receives the response it can delete token and update UI models.

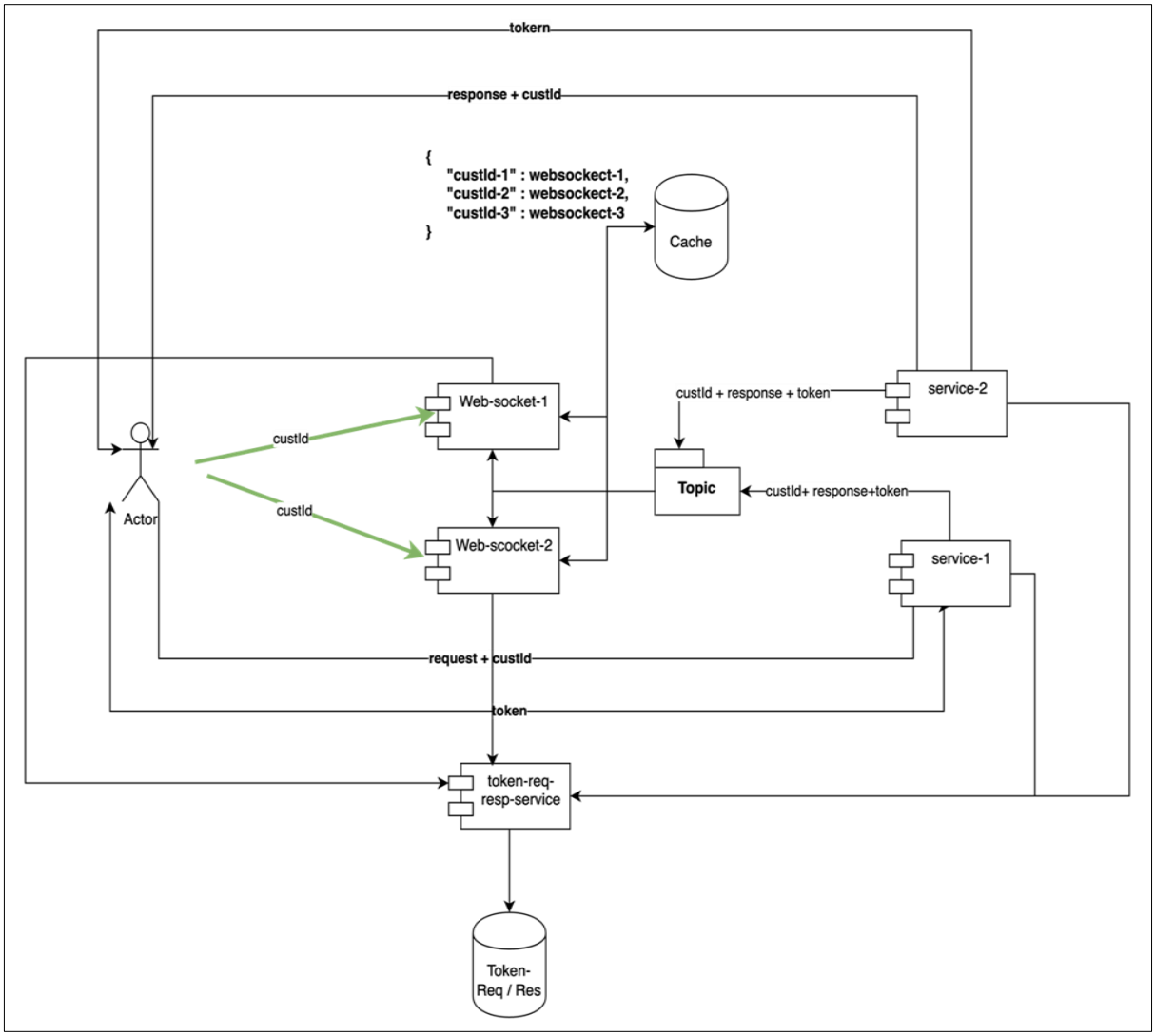

Fig 7: Asynchronous Request

Conclusion

The paper describes Microservice architecture style. First it explains, In startup environment when there is limitation in recourses to invest in Microservice architecture, one can start with Monolithic architecture then slowly migrate to microservice architecture. Base layer of any architecture is database, keeping future migration in mind based on the classification of the data separate database schema should be created in single database.

Next guideline is to create common library for basic framework which can be used across multiple backend services which brings uniformity across multiple backend services.

Also, paper discuss about various communication types between various microservices. It describes advantages and disadvantages of the various communication types and recommends architecture style for communication. Finally, it describes architecture style for web application using

websocket which drives high throughput because of Asynchronous implementation of API.

References

1. Li S, Zhang H, Jia Z, Zhang Z, Zhang C, Li J. Understanding and addressing quality attributes of microservices architecture: A Systematic literature review. Inf Softw Technol. 2021 Mar;131:106449.
2. De Lauretis L. From Monolithic Architecture to Microservices Architecture. In: 2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW). 2019 Oct 27-30.
3. Ghofrani J, Lübke D. Challenges of Microservices Architecture: A Survey on the State of the Practice. ZEUS. 2018;1-8.
4. Surianarayanan C, Ganapathy G, Pethuru R. Essentials of Microservices Architecture: Paradigms, Applications, and Techniques. Taylor & Francis; 2019.
5. Ren Z, Xu X, Wan J, Zhang W, Zhao J. Migrating web applications from monolithic structure to microservices architecture. In: *Proceedings of the 10th Asia-Pacific Symposium on Internetware.* 2018.
6. Quick Start [Internet]. gRPC; 2024 Nov 25 [cited YYYY MMM DD]. Available from: https://grpc.io/docs/languages/java/quickstart
7. Core Concepts, Architecture and Lifecycle [Internet]. gRPC; 2024 Nov 12 [cited YYYY MMM DD]. Available from: https://grpc.io/docs/what-is-grpc/core-concepts
8. Overview [Internet]. Protocol Buffers Documentation; [cited YYYY MMM DD]. Available from: https://protobuf.dev/overview
9. Apache Kafka®: Basic Concepts, Architecture, and Examples [Internet]. Confluent; [cited YYYY MMM DD]. Available from: https://developer.confluent.io/courses/apache-kafka/events
10. Intro to Apache Kafka®: Tutorials, Explainer Videos and More [Internet]. Confluent; [cited YYYY MMM DD]. Available from: https://developer.confluent.io/what-is-apache-kafka