Architecting Cloud Native Microservices APIs

A handbook of end-to-end best practices, design considerations and practical trade-offs

Indrajit Nadgir

Table of Contents

Introduction **5**

Cloud strategy for your applications **7**

API design for your cloud native microservices applications **13**

Microservices communication and distributed transactions **29**

Handling distributed data **45**

Microservices application security **57**

Build **68**

Deploy **77**

Service Mesh **93**

Operate **99**

Conclusion **113**

TO AMBIKA

This book would not have been possible without the sacrifices you have to made; the countless weekends I spent on this book are the ones I did not spend with you. Thank you.

Author information

Indrajit Nadgir email:
indrajitnadgir7@gmail.com

Indrajit is a software product architect with decades of expertise in designing and developing enterprise software applications. He has led complex cloud modernisation and digital transformation initiatives for products with multi-million(s) $ size, that impact thousands of customers and millions of end users.

Section 1: Introduction

If you are a product architect who wants to understand all the moving parts of a cloud native microservices application and know the design best practices this book is for you.

In this book we start with what cloud strategy suits your needs and then evaluate best practices, trade-offs & important considerations in microservices API design, transaction management, data distribution strategies and security.

Then we step into what it takes to build, deploy and operate the APIs we just created thus giving you an end-to-end overview of the cloud native microservices application lifecycle.

This is not an introductory level book. For example, this book does not explain some basic concepts. It is assumed you already know all that stuff.

Section 2: Cloud strategy for your applications

Cloud and microservices seem to be the buzz words everywhere in the world of application development. Microservices and cloud have revolutionised the way enterprises approach their digital transformation strategy. There is a paradigm shift in the technology ecosystem with cloud that brings so many benefits with it like high scalability, high availability, fault tolerance and better time to market. But the most important benefit the cloud offers is that it changes CAPEX to OPEX. It is one of the most important advantages that cloud offers and it is a driving force for many CTOs to embrace cloud.

How does cloud convert CAPEX to OPEX?

When you run your applications on-premises, you first invest in setting up a data centre, servers, network and buy software licenses. Then you hire the administrators that will manage your data centre. All of this is upfront cost and what counts as Capital Expenditure (CAPEX).

For example – Let us say you have an e-commerce business application and you set up a data centre with all the hardware and software leased with a certain load in mind. But if your business is far more successful and far more quickly that you imagined, you need to provision more infrastructure. But it could be seasonal or a spike which does not sustain. After a point you have all this additional infrastructure which you do not need anymore. You see the problem? You are to predict the demand in advance and invest before you see any revenue from that demand. The chances that you predicted wrong are far higher that you think.

However, when you move to cloud all this cost is reduced. Cloud providers will provide you the required infrastructure on the go. If you need more you can provision more. This converts the cost model to Operational

Expenditure (OPEX). You end up paying for what you use. Thus, it gives you more control over the costs and brings down the overall cost – Total Cost of Ownership (TCO).

Cloud takes a pay-as-you-go approach. So as your demand increases you can provision more (scale) and if the demand reduces you can reduce your provisions accordingly.

Now that cloud offers all these advantages, should we move all our applications to cloud?

Let us consider the following criteria for a moment:

- You have a small business application created using a fixed technology stack which is unlikely to change.
- The demand for your business/ application is stable and predictable. It does not increase/ decrease drastically or suddenly.
- The teams creating the application are fixed in size and unlikely to grow exponentially.
- It is okay to have a long time to market for your application and frequent releases are not required.
- The applications are not mission critical and some downtime can be accommodated.
- The network traffic is stable and predictable
- You already have a data centre set up with required infrastructure to run your application.

If you see that you tick most of the boxes above then you must consider using or continuing to use a monolithic architecture application. There is nothing wrong with a monolith as long as it is a deliberate architectural choice.

When do you need your applications to be the cloud?

It is essential to take a pragmatic approach when choosing between architectures. The purpose of the architectural choices is to enable a solution to a business problem considering all requirements like time to market, cost, availability, performance, fault tolerance, expected growth or expected increase in load, etc.

The problem is – when you have a hammer everything looks like a nail. And microservices architecture is no different. We have to be careful because microservices architecture will not necessarily solve the problem and sometimes it will create many.

When do you choose to develop your application using microservices?

- Your business is scaling really fast. This is typically observed with fast growth start-ups where there is a sudden explosion of user base and the application must keep up with the sudden increase in the traffic. Microservices architecture is built to do that. Applications built using microservices deployed on cloud infrastructure can be easily orchestrated to scale on demand.
- Scaling: You have a business where the number of users can suddenly increase at a given time of the day or in a given month of the year. For example: you have a tear end clearance sale on your e-commerce website every year in say December, right before Christmas. You expect that the traffic will increase 4-5 folds during that time. You want your application to be able to handle such load automatically.

- Time to market: When your application is a monolith, you build the entire application even when you make a small change to the application. It takes time to build and especially to test. Because now you have to run the whole battery of tests to deploy on production even though your change is small. Develop -> Staging -> Production testing. If it is a product that must be released to an end consumer via a delivery portal, then there is the whole process to get the product to GA (General Availability).
- But with microservices you can build, test and deploy/ deliver each microservice independently. Microservices are designed to be light weight and thus integrate very well with Continuous Integration and Continuous Delivery (CI/CD) platforms. So,the time to market reduces considerably. This is especially an advantage for products which require to make rapid changes to their design during early stages.
- Fault tolerance: With a monolith a failure in one part of the application brings down the whole application. But it is unacceptable when you have a mission critical application where you need fault tolerance to be built in to the application. Microservices are by design fault tolerant since they are independent of each other.
- Zero downtime (High Availability): Another requirement for a mission critical application is to be highly available. A near zero downtime is expected not only during the runtime of the application but also during upgrades and patches. Microservices architecture can ensure high runtime availability. They can be delivered easily with advanced DevOps like using canary releases to ensure smooth upgrades.

- Different parts of the application use different languages or different databases.
- Yours is a very successful product which has evolved beyond recognition. The teams have become large and unmanageable. The source code keeps bloating up and maintaining it is no longer possible in the traditional way. You need to move to microservices architecture immediately. Microservices architecture offers to modularise the application into small self-contained units which can be led by independent teams. Each microservice is designed to be developed, tested and deployed or delivered independently so individual team can choose to do what is best for them thus allowing them to scale.

If your application needs any of the above criteria satisfied then consider microservices architecture. In this book we will discuss many challenges that will arise with microservices architecture and how we can solve them. What the different trade-offs, considerations and best practices are.

As we begin designing a microservices application for cloud, one of the most important areas to start with is the API design. Microservices, most of the times, are made of APIs. That is how they expose their functionality to outside world. In the next section we explore some of the best practices to design our APIs.

Section 3: API design for your cloud native microservices applications

A good API design is a prerequisite to a good microservices application. A well thought out API strategy must be put in place before starting any development work so that the resulting APIs are easy to use and consume, well documented, secure and performant. REST APIs are the most common web interfaces that are available today which can be consumed by various clients like a mobile application or from a web browser. The beauty of REST APIs is the ease with which they can be developed and consumed. It is very easy to develop a REST API but difficult to do it right. Below are a few best practices that will help you to design your APIs so that they can be robust.

Use JSON in HTTP requests and responses

JavaScript Object Notation (JSON) has gained immense popularity over the years because of its simplicity and readability. REST APIs can consume and respond in both XML and JSON formats, but using XML needs a lot of parsing and processing on the client side before consuming it and it introduces considerable overhead. Use of JSON makes it straight forward to consume data especially since there are in-built JavaScript libraries to parse JSON objects to text and vice-versa on both client and server side.

Nouns in the url

Use nouns instead of verbs to name the endpoint URLs. HTTP requests already have verbs in them like GET, POST, PUT and DELETE. Do not add those actions again in the url like

```
/v1/@JaneDoe/get_articles
/v1/@JaneDoe/create_article
```

This will make the urls verbose and can be quite confusing as the APIs get more complex.

Instead use nouns to indicate the resources on which the action is intended and let the standard HTTP verbs indicate the action.

For example:

```
GET /v1/@JaneDoe/articles (to get all articles)
GET /v1/@JaneDoe/articles/:article_id (to read an article with an id)
POST /v1/@JaneDoe/articles (to create an article)
PUT /v1/@JaneDoe/articles/:article_id (to update an article with an id)
DELETE /v1/@JaneDoe/articles/:article_id (to delete an article with an id)
```

Hypermedia As The Engine Of Application State (HATEOAS)

HATEOAS is a REST architectural constraint and a design best practice. With HATEOAS we can send the available actions and resource links embedded in the HTTP response dynamically. The client sending the request does not need to hard code the URLs and thus it can help prevent broken URLs as the endpoints evolve over time. The clients can navigate and change the application state based on the hypermedia links in the responses thus driving the workflow forward.

The server can decide what further resources should be made available to the user based on the request (and other parameters like authorisation, business logic, etc) and add appropriate links.

The client can build menus, buttons, or navigation dynamically based on the response from the server.

For example, consider a GET request to fetch the list of articles by a given author.

```
GET /v1/@JaneDoe/articles?page=3 HTTP/1.1
```

The server can just return the paginated list as requested. But then the client does not know if it can go to the next page or what other actions it can perform from this point onwards after displaying the requested page. And the client has to build the URLs on its own to navigate and perform those actions. The client gets involved in driving the application state and any API changes on the server-side must result in changes on the client-side. It leads to broken links and functionality more often than expected.

HATEOAS can help handle this problem with embedded links which tell the client what actions it can perform next and how.

For example, a HATEOAS response to the GET request above would look like:

```
HTTP/1.1 200 OK
Content-Type: application/hal+json
{
        “meta”:
                {
                        “page”: 1
                        “total_pages”: 5
                }
        “author_name”: Jane Doe,
        “email”: jane_doe@myamazingwebsite.com,
        “data”: [
                {
                        “article_id”: 1,
                        “article_title”: Example Title 1,
"links": {
"self": "/v1/@Jane_Doe/articles/1"
}
                },
                {
                        “article_id”: 2,
                        “article_title”: Example Title 2,
"links": {
"self": "/v1/@Jane_Doe/articles/2"
}
                }
        ],
"links": {
"self": "/v1/@Jane_Doe/articles?page=3",
"first": "/v1/@Jane_Doe/articles?page=1",
"prev": "/v1/@Jane_Doe/articles?page=2",
"next": "/v1/@Jane_Doe/articles?page=4",
"last": "/v1/@Jane_Doe/articles?page=5"
}
}
```

The client knows from the response that it can navigate to the next page, previous page, first page and the last page — all dynamically linked to the navigation buttons in the UI. Further, the users can also click on the individual articles to display the contents of the article.

Although HATEOAS relieves the client from driving the application state, it does not mean the client can be oblivious to the application state. The client still needs to know what different actions are possible from a given state. But HATEOAS helps the client to determine what among those actions are valid and how to handle them — dynamically.

However, on the downside it can make responses quite verbose and many developers find it impractical to handle workflows entirely using HATEOAS. So, due trade-offs need to considered during design.

Secure your APIs

Security is a cross cutting concern in enterprises. So, there will usually be an organisational security policy that guides the application security. It is an industry-wide practice to adhere to 'Defence in Depth' with many layers of security while application & data security being just two layers in it. Security is an elaborate topic per se. I have explained in detail about it in a dedicated section on the security best practices for cloud deployment of APIs in this book.

It is a common practice to have a separate dedicated service for authentication and authorization across the enterprise and use an API gateway to handle incoming requests before they even hit the individual services.

However, it is vital for the service to assume nothing and practice a zero-trust policy. What it means is that while it is not its job to authenticate or authorize requests per se, the service must validate the incoming request token by delegating it to the authorization server and only allow request if the token is valid and has the required set of permissions to access the endpoints.

Many frameworks have provisions to additionally secure methods/ routines/ functions based on the role of the incoming request user.

For example, Spring Security for Java based applications has support for @Secured annotation over a service method that can be used to secure the method to calls from a user with a given role only.

All communication from clients to services must happen over TLS encrypted channel. Set up mTLS for secure communication internally between services. Consider using an API Gateway or a service mesh to address a lot of cross-cutting security concerns.

Either by using the organizational security infrastructure or adding your own to secure your APIs, a security strategy is a must.

Caching

Every request usually hits the database. Every time you fetch something from the DB it involves some latency. To pack better performance in your APIs you can cache the data from previous responses. On the downside the returned data could be out-of-date. So, it is important to have an expiry time set for each object cached after which it is evicted from cache.

Caching can be supported at different levels:

If you use an ORM framework like Hibernate it offers a first-level cache and a second-level cache that can reduce DB hits.

There are in-memory caches like Redis which can cache data and return from the service layer of the application without even hitting the DAO layer. API gateways can be configured to cache and return data without even hitting the application.

Browsers cache webpage content in a local or private cache for a single user for faster load time.

CDN (Content Delivery Network) servers cache images, videos and webpages to reduce latency.

Be sure to use the Cache-Control header to specify how the clients should handle caching of data.

Pagination, Sorting and Filtering

The data returned by a GET request can be very huge. If all of this data is handled all at once from querying the DB to building a JSON response to displaying the data on the UI it can be a performance bottleneck and slow down the application considerably. Especially if it is a popular workflow and has a lot of users performing the same query in the same time period it can be a very bad user experience.

Pagination allows the user to define the page size and the page number they are looking for, so that the server can return the results only for that page. Databases often have a way to specify OFFSET (page number) and LIMIT (page size) on the query so that only the expected subset of data is fetched.

For example:

```
GET /v1/@JaneDoe/articles?page=3
```

displays the page 3 of the list of articles by author Jane Doe.

```
GET /v1/@JaneDoe/articles?page=3&page_size=5
```

limits the number of articles per page to be 5.

Support users to specify how they want the data returned to be sorted.

For example:

```
GET /v1/@JaneDoe/articles?sort=+likes&-published_date
```

displays the list of articles by author Jane Doe sorted by number of likes and then by date of publishing the article. '+' and '-' are used to indicate the ascending and descending orders.

Allow users to filter the content returned based on additional criteria.

For example, let us say a user wants to search for articles by Jane Doe but only if the article contains the words 'cloud' and 'containers'.

```
GET /v1/@JaneDoe/articles?search=cloud,containers
```

or say we want to search for an article with a given title

```
GET /v1/@JaneDoe/articles?title=That%20Wonderful%20Book
```

Documentation

It is very easy to miss this one. Far too many wonderful APIs have been rendered useless because of lack of proper documentations. I cannot stress this enough — document, document, document. If the consumers cannot understand what the API does, what requests are expected or what responses to expect, it cannot be impactful. There are many frameworks to make a developer's life easy while creating meaningful documentation for APIs.

For example — Open API Documentation (Swagger) is a very popular and simple framework to create highly consumable documentation for your APIs. It offers excellent integration into Spring applications and easy documentation using annotations.

Proper request validation and Exception handling: Return standard HTTP error codes

You are working on a critical application and the server responds with a generic 500: Internal Server Error. You try to find out from the error message if there is anything you can do to get it working but the application has suddenly decided that you are its mortal enemy and does not want to talk to you. There is no meaningful detail in the error message that you can use to work around or even understand the problem. And you hear yourself exclaim 'Et tu Brute!'. You do not want the users of your application to go through this experience.

The trick is to validate all incoming requests to see if they are inline with what is expected by the service. Do not assume the users to always send the right kind of requests. There could be null values or invalid values in the request parameters. Be sure to validate it all.

The second part of this equation is to handle all exceptions and converting them into appropriate HTTP Status codes with details.

Do not return the exception trace as it is. Most users cannot read it or cannot make sense out of it. Do not send just the HTTP Status Code and generic error message with it. It is insufficient.

Offer details about why such an error occurred and if there is anything a user can do to work around it.

For example: Let us say a user tries to login without creating an account first. This obviously will lead to a UserNotFoundException from the service. Sending a response with empty body or some cryptic error message will cause more harm than good. Also, returning a response with HTTP Status code 500 with just the exception stack trace does not help much either. To the response add a proper message that says "*The user id which you are using to log in is not yet created. First create your account using 'Register User' link.*". Add the link to 'Register User' page using HATEOAS and send it along with the response. Now the user has a way to understand what went wrong and what the user can do about it.

Logging and Monitoring

Add a logger with different levels of logging to the APIs. In production, log levels are set to ERROR (there should not be many, anyway :)) and in

staging logs typically are a little more verbose at DEBUG level. This should help investigate any problems quickly and address them.

Monitoring and collecting operational statistics of our running application is vital to keep track of how the application is performing and behaving in production. There are several frameworks that help achieve this at an application level and at an infrastructural level.

For example — Spring Actuator provides several endpoints like /health, /metrics, /info, etc that provide details about the application and integrate well with Spring applications so well that monitoring becomes a trivial problem.

Prometheus provides monitoring and alerting using time series data at a container runtime level. Actuator provides Prometheus integration as well which makes it very easy to visualise all data using a tool like Grafana.

Testing

Goes without saying. Unit test each API endpoint thoroughly with all combinations of possible request data including all null values. Be sure to design and develop the endpoints with Inversion of Control (IoC) and Dependency Injection (DI) rather than hard-wiring the instances.

APIs must be tested with Unit tests, Performance tests, static security analysis tests, dynamic security analysis tests (runtime), container security tests, Integration tests, Functional Tests, User Acceptance Testing (UAT), Smoke tests and Regression tests. Automate and ensure good code coverage.

Versioning

Support at least two versions of the API at any point of time. Typically versions can be specified in the endpoint URL like so:

```
GET /v1/articles
GET /v2/articles
```

API changes from vCurrent to vNext must be announced in advance. If any functionality is getting removed, do not remove it in vNext. First deprecate it in vNext and then remove it in vNext + 1. Any deprecations from vCurrent must be included in the announcement.

Usually, the release cycle repeats as follows: announce the End of Service (EoS) for vPrevious, then announce vNext General Availability (GA), then GA vNext, EoS vPrevious, then announce the EoS for vCurrent, then announce vNext + 1 GA and so on.

Use canary releases to deploy new versions. It involves deploying the vNext with all requests still routed to vCurrent. Once all testing is completed, route only 5% of the traffic to vNext and then slowly increase to a 100%.

Backward compatibility

Ensure the API changes are done in a way that is backward compatible. Do not remove information from the return data in a new version but add information as required by the change.

For example

Let us say in v1 of the API, we are displaying the author's name for an article.

So typical response for a GET request like

```
GET /v1/articles?author_name=Jane%20Doe HTTP/1.1
```

would be

```
HTTP/1.1 200 OK
Content-Type: application/hal+json
{
        ...
          "author_name": Jane Doe          ...
}
```

And then we create a new v2 for the API where we want to split the author's name into first name and last name. So, we replace the author_name with first name and last name in v2. Let us say v2 also offers additional information with HATEOAS links. There are new consumers for these links but want to retain the author_name from v1. They cannot upgrade to v2 even though they might want to consume the new links added in v2 because it breaks the previous functionality.

However, adding the newly created fields along with author_name will help retain functionality.

```
HTTP/1.1 200 OK
Content-Type: application/hal+json
{
        ....
        "author_name": Jane Doe,
        "first_name": Jane,
        "last_name": Doe
          .....

}
```

Now the new consumers who want to use first_name and last_name can use those fields and the consumers moving from v1 to v2 who want to continue using author_name but want to use links can safely use v2 without breaking functionality. Obviously, this example is very simple (for brevity) and the real-world problems and use cases tend to be much more complex, but the driving principle remains the same.

Return a single object in response body

When sending response to a REST request many a time it is quite common to make the mistake of sending a list.

For example: It is very tempting to send a list of articles matching the search query for a GET request like /articles?author_name= abc. But consider the implications for a moment. If the endpoint changes and we no longer want to send just a list of articles but also some author info along with it, it becomes annoyingly difficult to do that. The API needs to change in a way that is no longer backward compatible.

So, send an object as a response that wraps the list. This way we ensure that when we need to change the response to include additional information it can be handles elegantly.

Rate limiting, Circuit breakers, Throttling

There are design patterns which help build a robust API that can handle failure elegantly.

Rate limiting an API defines the number of requests a user can send to your API in a given period of time. Any requests beyond the limit will return with HTTP status code 429: Too many requests.

Most API Gateways provide this functionality out of the box.

Circuit breaker is a design pattern to prevent an API from calling a failing operation over and over expecting a success, when it is likely that the operation will continue to fail. Netflix Hystrix is a very popular example. Isto service mesh has a circuit breaker functionality using Virtual Service and Destination Rules.

User loads vary on an API depending upon time of day or business hours, etc. This can put overload on the resources and cause them to fail or partially unavailable. One way to handle this is via autoscaling. But autoscaling involves provisioning of additional resources and that takes some time. The services might be unavailable temporarily during that period. Alternatively, when the resource consumption by APIs reaches a certain level, the requests can be throttled to ensure the APIs can continue to work. One strategy could be to suspend low priority operations until the high priority ones are completed or additional resources are provisioned.

Parting thoughts

Like I said when we began this section, it is very easy to design an API but very difficult to do it right. In this chapter I have outlined a few best practices that will help you consider some important design aspects while architecting your REST APIs for your microservices. Once you have your API designs ready the next problem to solve will be to decide how these APIs will work together to solve a problem. We head into the next section to understand microservices communications and the several trade-offs involved in it.

Section 4: Microservices communication and distributed transactions

One of the very important areas to consider during architecture design phase is – how the microservices are going to communicate with each other. Have a strategy in place that covers all microservice communications.

- Do you need synchronous / asynchronous communication between services?
- Security considerations

  Secure the communication channel between microservices using mutual TLS (mTLS). It can be done using a service mesh that secures the east-west traffic.

- Performance implications

  Inter service communication is inevitable in microservice architecture. However, it is a good idea to limit the chatter over the east-west traffic. During design do consider if the services will become very chatty and what is the performance penalty involved. Maybe it is a case where two services are so tightly coupled in the business context that they are part of the same use case. It might be prudent to merge them into a single service. There is nothing wrong in designing and developing a monolith or a macro-service after due diligence and if it is deliberate. It is not bad architecture. You are just prioritising strong consistency and performance over availability and scalability. Likewise, it is absolutely fine if your business needs you to favour higher availability and be eventually consistent.

Synchronous communication between microservices

Microservices can issue synchronous calls between each other when the use case justifies it. But one needs to be careful about long running operations in another microservice if a synchronous call is made to it from your microservice. Do consider other alternatives like modular monoliths or eventual consistency.

If you choose to use synchronous communication it is pretty straight-forward to issue a call from one microservice to another. It can be via a REST call, for example. However, it is quite expensive to do it and considered an anti-pattern because it increases coupling and reduces availability.

Asynchronous communication between microservices

Microservices are built for asynchronous communication. It makes the design robust and loose coupling provides high availability. Asynchronous communication is typically achieved with a distributed event store acting as an event bus between microservices.

Although asynchronous calls provide better performance, they can create consistency issues. The design leans in favour of eventual consistency. This is discussed in detail in the next section.

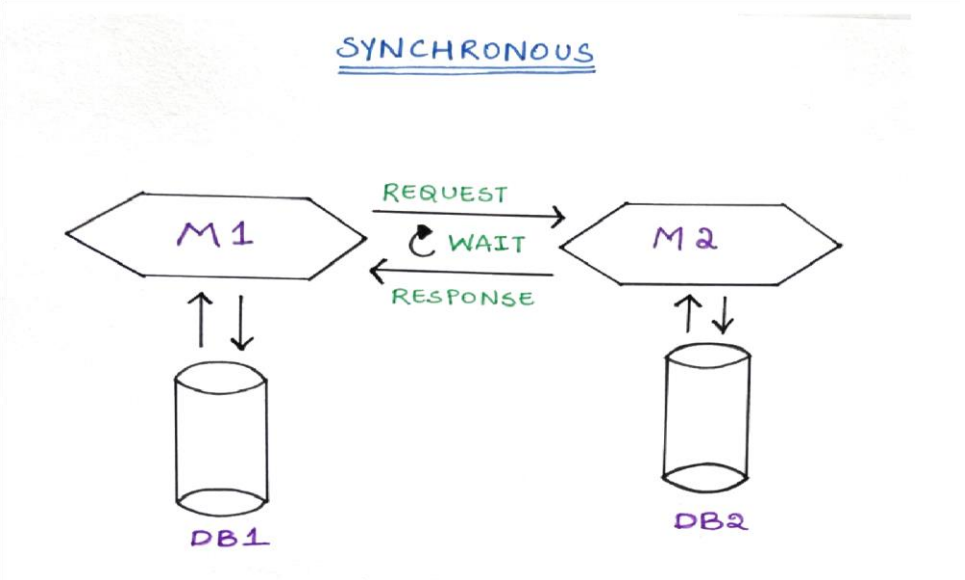


Figure 1: Synchronous communication between microservices

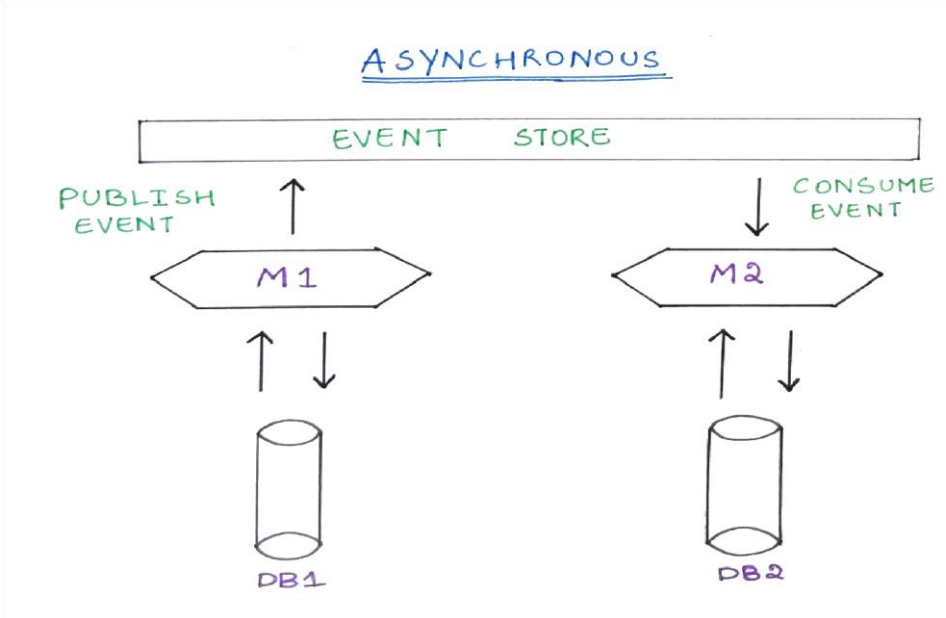


Figure 2: Asynchronous communication between microservices

Transaction Management

In a monolith, transactions are ACID (Atomic, Consistent, Isolated, Durable) compliant. Since monoliths run as individual processes sharing

the same runtime and using a single database instance, transactions can be handled seamlessly. But that is hardly the case with microservices.

Microservices are designed to use their own databases and run as separate processes.

That could mean some part of data will be handled by microservice M1 and committed to database DB1 and some other part of data will be handled by M2 and committed to DB2, as part of the same transaction. Or it could mean that a microservice must write to a combination of multiple databases or message queues/ event stores.

Trade-offs need to be drawn between consistency and modularity. When you split the functionality into multiple microservices you give away strong consistency between operations. You are trying to reduce the affinity between services to make them independent and therefore they operate in an asynchronous fashion. Asynchronous operations tend to be eventually consistent.

If an operation needs to be highly consistent and you are considering the operation to be spread across two or more microservices, then reconsider. It will be prudent to use a modular monolith instead of microservices in this scenario.

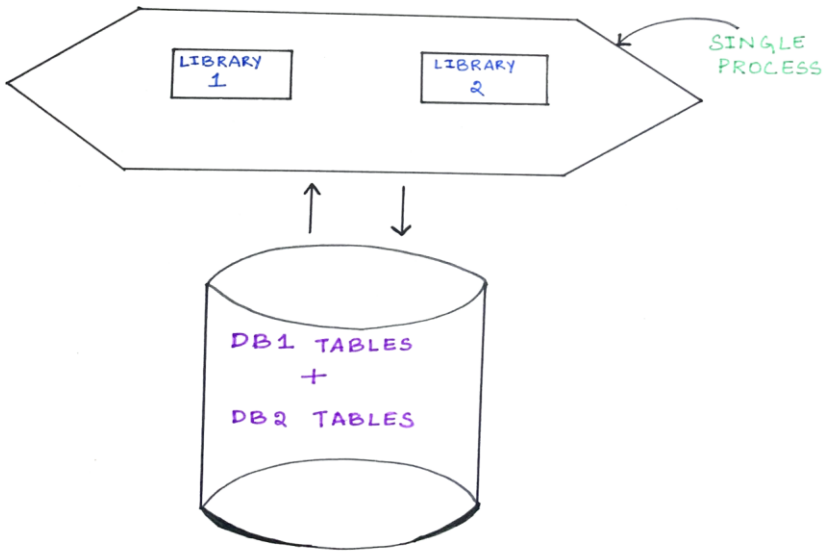


Figure 3: Modular monolith running as a single process with a single database

However, let us say that there is a business case with an operation O1 consisting of 3 steps:

1. Commit to DB1 //needs to be highly consistent
2. Commit to DB2 // needs to be highly consistent
3. Commit to DB3 // book keeping. Can be eventually consistent

The first two steps have very high consistency requirements while the third step is okay to be eventually consistent since it is just logging some information to DB3.

So, we can handle the first two steps in microservice M1 and then put out a message to microservice M2 to handle step 3.

This means that there must be distributed transactions in M1 & M2. Distributed transactions have higher performance penalties and can get quite complex to manage. It is recommended to avoid distributed transactions, ideally. But we don't live in an ideal world, do we?

Two Phase Commit (2PC XA transactions)

One way to handle a distributed transaction is to use XA (eXtended Architecture). XA is a two-phase commit (2PC) protocol by X/Open group standard.

XA transactions are designed to have a transaction manager which controls the commits across resources. Each local transaction has a resource manager attached to the resources.

The resource managers are enrolled into the transaction manager. So, when the transaction manager commits, each resource manager commits its own local transaction. If any resource manager reports a failure, the transaction manager rolls back all the commits.

Using XA, the scenario that we described earlier can be handled as below:

in Microservice M1

```
XATransactionManager tm = new XATransactionManager (XAResourceManager rm1, XAResourceManager rm2, XAResourceManager rm3);
  try {
        tm.begin();
        // Some SQL statements for relational database DB1 (mapped to rm1);
        //Some statements for database DB2 (mapped to rm2);
        // Some messages produced and sent to an event store ES (mapped to rm3);
        tm.commit()
      }
  catch(Exception ex){
        tm.rollback();
       //set your house in order
  }
```

Separately in microservice M2

```
{
  try {
      // Init transaction tx
      tx.begin();
      // Consume the message from the event store ES
      // Process data
      // Execute changes to DB3
      // Produce messages to ES
      tx.commit();
      }
   catch(Exception ex){
        tx.rollback();
   }
}
```

M2 can choose to create its own 2PC transactions if there is a need or it can handle the operation as a set of local transactions with idempotent consumer pattern.

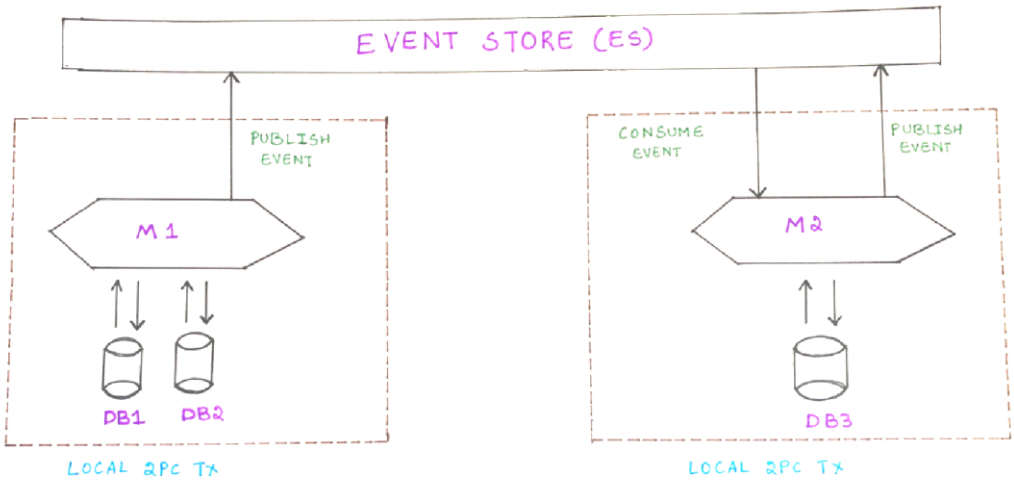


Figure 4: Local 2PC transactions in microservices

M1 is synchronous and highly consistent in its commit to DB1, DB2 and ES. M2 is synchronous and highly consistent (if 2PC is used) in its commit to DB3 and ES. However, the overall system spanning across M1 and M2 is asynchronous and if M2 fails in its local transaction there is no way for M1 to rollback its changes.

But what if the commit to DB3 is required to be strongly consistent?

Let us say the requirement is to commit to all the databases DB1, DB2 and DB3 at once or all must fail. There are use cases like a banking transaction where eventually consistent transactions can create confusion and bad user experience. In such use cases it is better to use a good monolith to design solutions.

If you want synchronous and highly consistent distributed transactions within a micro service you can use 2PC. But if you need it across

microservices then consider a redesign. When you try to achieve consistency across microservices using 2PC, you are using a tool that was not quite meant to do that. Different microservices run as separate processes and thus they cannot guarantee that the local transactions inside each microservice can be globally atomic.

There are frameworks that support distributed transactions across REST APIs. But the associated performance penalty is very high and it impacts availability of the overall system. Add to this the complex corner cases like *'what if the transaction coordinator goes down for a very long period'*, you get the picture. It often leads to strong coupling; unwanted dependencies and the architecture starts to get a little more complex than you would like it.

However, if you are the adventurous type, figure below shows the best-case scenario in a 2PC transaction across microservices.

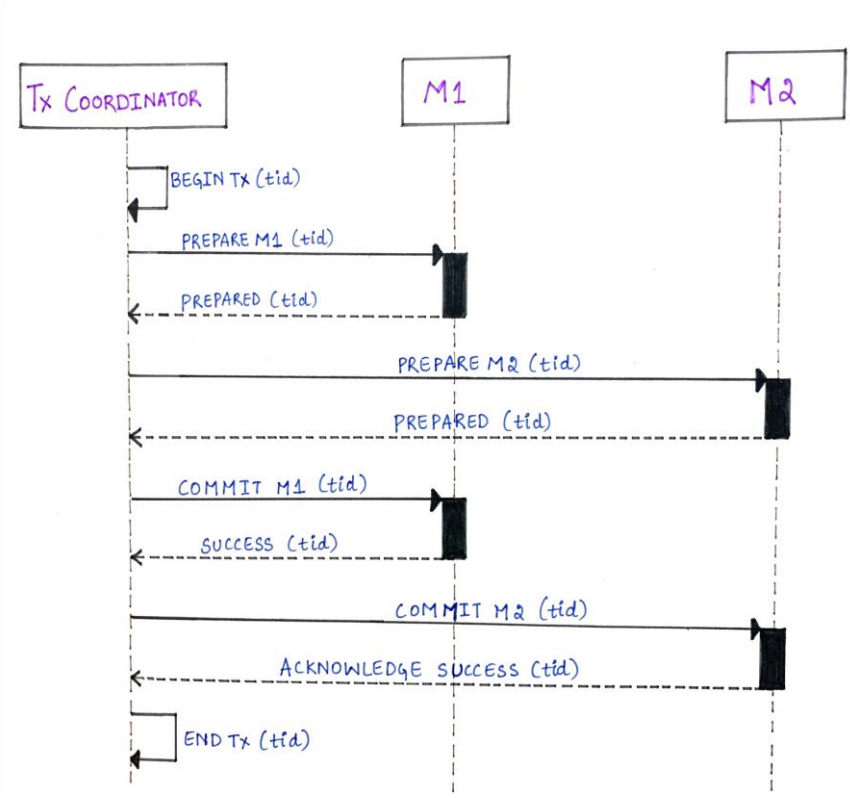


Figure 5: 2PC transaction between microservices – success scenario

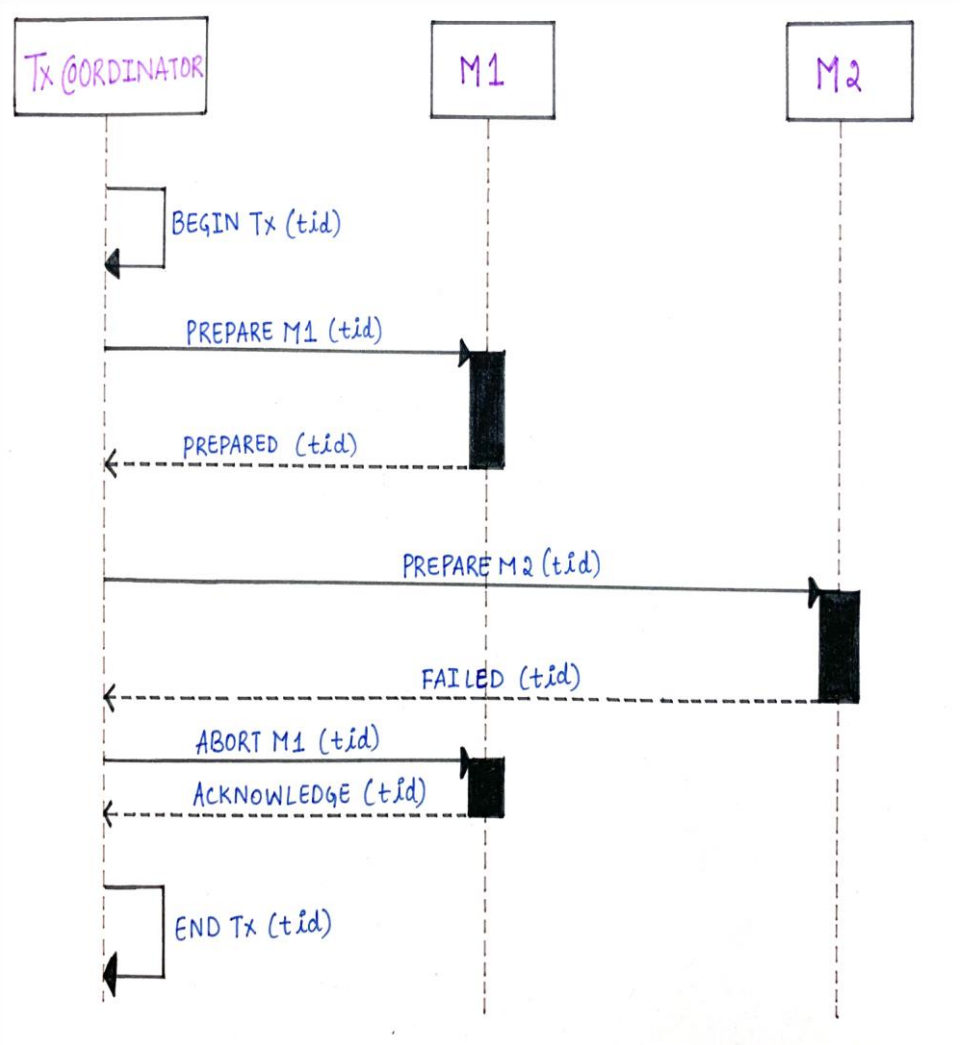


Figure 6: 2PC transaction between microservices – failure scenario

In the best interest of your own sanity, avoid scenarios described in Figure 5 and Figure 6. In most cases it is acceptable for the system to be eventually consistent including local transactions within a microservice. These cases can be supported with microservices executing their individual local transactions and communicating to other microservices about the outcome. And then the services involved in the work flow coordinate with each other to either reach a desired end goal of a successful distributed transaction or in case of a failure in one or more local transactions

compensate the earlier local transactions by issuing a set of transactions to eventually bring the system to a consistent state.

There are a couple of ways to go about it.

Orchestration

One of the services can take up the responsibility of being an orchestrator who leads the interactions between services during the transaction work flow. The orchestrator issues requests to other microservices to execute their end of the transaction or issue requests to compensate in case of a failure.

From the example with microservice M1 and M2 earlier,

Let us modify it a little bit. Microservice M1 commits to DB1, M2 commits to DB2 and M3 to DB3.

Microservice M1 can act as an orchestrator with the work flow as below:

- M1 instantiates the orchestrator O1.
- O1 issues commit to DB1.
- O1 sends a command to M2. This can be, for example, using a message broker with M2 acting as an idempotent consumer with ability to retry its operation.
- M2 commits to DB2.
- O1 acts as a consumer to the result of the operation on M2 (direct response or via a message broker).
- M2 responds with the result of its local transaction.
- If M2 is successful, O1 continues its happy flow. If M2 reports a failure, O1 issues a set of compensating transactions to commit to DB1. That ends the saga.

- If successful, then O1 now issues command to M3 to commit to DB3.
- M3 responds back with a status. If successful that completes the transaction. If not, then O1 issues a set of compensating transactions to commit to DB1. Then issues a compensating command to M2 to compensate the earlier DB2 commit. That ends the saga.

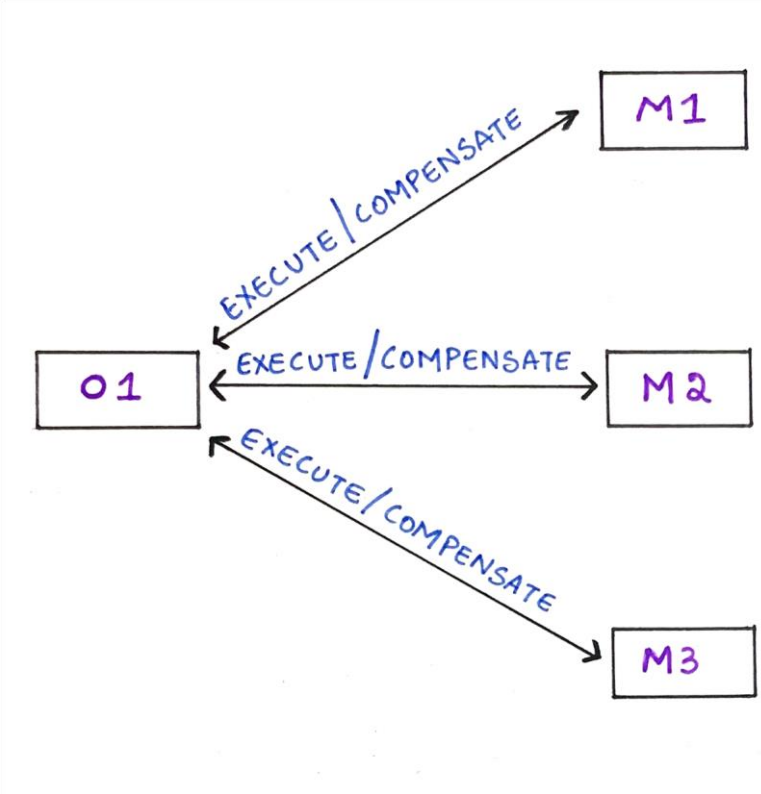


Figure 7: Orchestration with three microservices

Choreography

Choreography is a saga transaction pattern where each microservice executes a local transaction and updates its data source and passes on a relevant message to the event store. Other microservices pick this up and execute their local transactions and the process continues until all

individual microservices participating in a work flow complete their respective pieces of work.

If one microservice fails with an unexpected error then it issues a relevant message to the event store and other microservices issue compensating transactions locally to cancel their earlier commits. There is no centralized control over the entire transaction.

There are a couple of ways to achieve choreography for the same scenario described in orchestration.

Using an event store with 2PC

The participating microservices can generate events to reflect the results of their local transactions using an event store. Other participants can consume those events to either execute their local transactions or compensate.

However, each microservice must handle its own local transaction as a 2PC XA transaction since it involves two steps internally – commit to its own database and then publish an event to the event store.

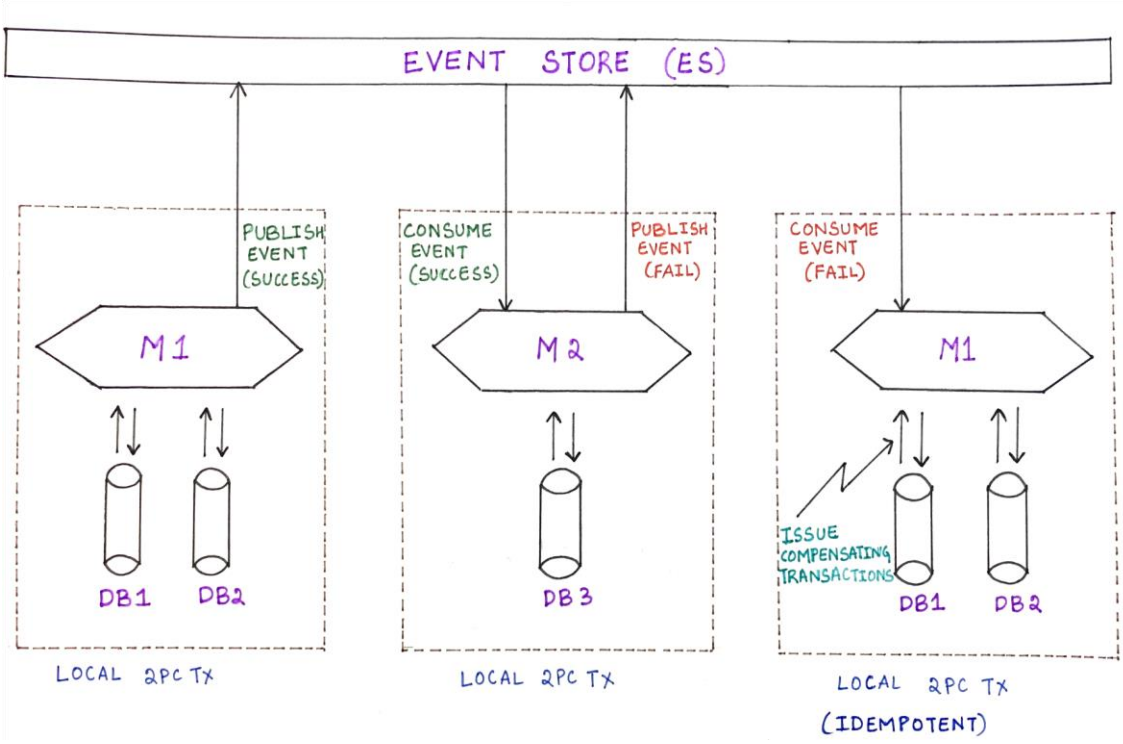


Figure 8: Choreography between two microservices M1 and M2

Use a Change-Data-Capture (CDC) tool like with an event store

2PC is not very convenient for reasons already discussed previously. You can choose to use a change data capture tool like Debezium to capture the database changes and send events to an event store. Other participants can consume those events and execute their local transactions or compensate.

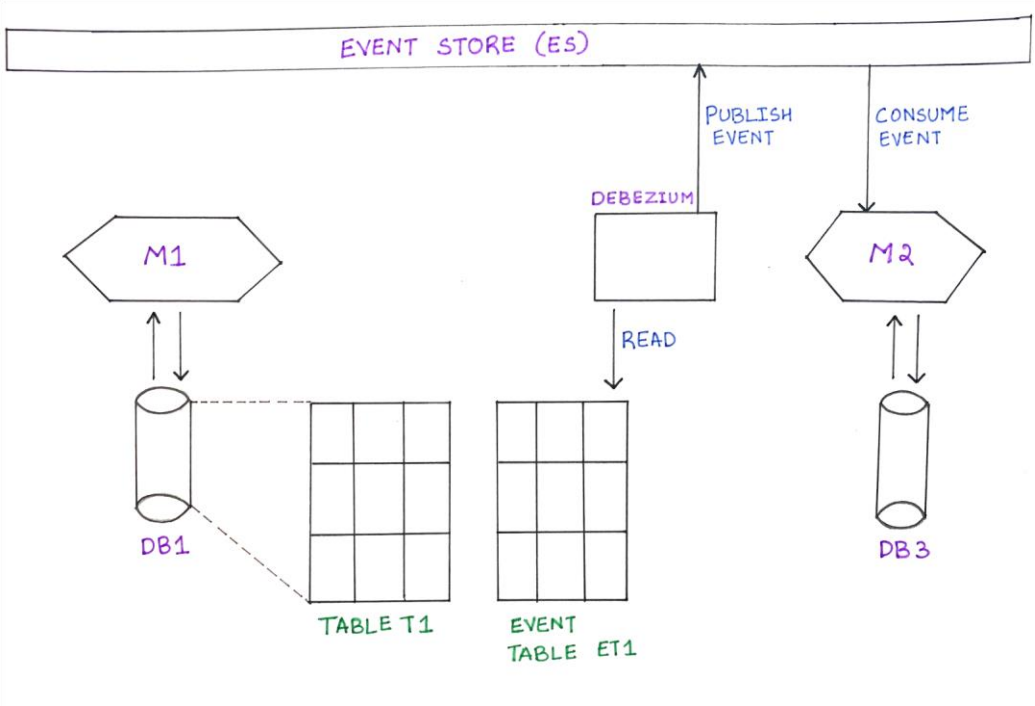


Figure 9: Choreography between microservices M1 and M2 using Debezium instead of 2PC.

Parting Thoughts

Although saga transactions seem very simple at first, there are cases which will come to light during implementation that will make you reconsider your strategy and revise it.

For example: What happens when some other transaction over-writes the data when microservices M1 and M2 are executing a saga. M1 commits to DB1. M2 fails. Meanwhile someone else has over-written the data in DB1. M1 is unable to issue a compensating transaction. Now that is data corruption! This might not be a big deal in social media websites because it represents less than 1% of cases. But for a banking application consistency is everything, even if it is eventual consistency.

Rest assured there are mechanisms to counter such issues and corner cases like for example using semantic locking.

Now that we have handled transactions in microservices we move to the next section to design our data strategy. Data in a microservices application is distributed because each micro service has its own database. It requires novel strategies to be developed to scale the data while making the system fault-tolerant and highly available.

Section 5: Handling distributed data

Data is the most critical aspect of enterprise application software. Be it a monolith or a microservices architecture there could be a need to distribute data. However distributed data strategy is non-trivial and needs to consider several trade-offs.

Especially if you are using microservices there is a chance that you will need to distribute data at some point. It could be because the application has grown exponentially and now the data needs to be split across multiple data sources or you may have a Command Query Responsibility Segregation (CQRS) design pattern implemented where the writes are handled by a different service and the reads are handled by a different service (and hence data is written to one node and read from multiple different nodes).

If you are using a single data source and the data is growing then there is an option to vertically scale it. It does not require you to distribute the data across multiple nodes of data source or horizontally scale it. If you have a traditional application, it is likely using a relational data base. Relational databases (RDBMS) have been around for a while now. They form the data backbone in most enterprises. The data is structured and expressed as a schema with assured ACID compliance.

With vertical scaling, as the data grows it needs stronger and bigger servers (more CPUs, more RAM, etc) to handle the kind of load that comes with volume. However, there is a limit to how much you can scale vertically and it presents performance problems after a threshold. Thus arises the need to scale horizontally.

Also, to improve availability and fault tolerance you need to consider replicating the data.

When you develop a strategy to distribute the data, do consider the trade-offs involved.

The most basic questions we normally start with -

- What is the volume of the data?
- Is the data structured or unstructured?
- What is the nature of data? Is it relational or non-relational? Does it need to be highly consistent?
- What is the performance penalty involved in distributing my data? Is it acceptable?

To handle high growth data (for example consider a social media app where the user data just keeps piling up) or unstructured data, it can be done using a NoSQL database.

The data is stored like a self-contained document without the need for joins. So, it is easier to scale horizontally than in a relational database. NoSQL databases favour high availability over high consistency. If horizontal scalability is your design preference and eventual consistency is an acceptable trade-off consider a NoSQL database.

Distributed data trade-offs: CAP Theorem

For a distributed datastore it is necessary for the system to make trade-offs between Consistency, Availability and Partition Tolerance (CAP). CAP Theorem formalises these trade- offs.

Consistency:

For a distributed data system to be consistent all queries should return the same data irrespective of the node which is queried. It means that the data is instantly replicated and all nodes have the same copy of data.

Obviously, it takes finite time for data to replicate across all nodes which is what contributes to latency. So perfect consistency is impractical. But for the purpose of this discussion, we assume the time taken to replicate is zero.

Availability:

A distributed data system is called highly available if all requests get a valid response even if one or more nodes fail. So, a request can be sent to an alternative node to get a response if a given node is unavailable thus making the system highly available.

Partition Tolerance:

A partition is a scenario when there is a communication failure between nodes. Thus, the nodes are not able to replicate data between themselves until the communication is restored. A partition tolerant distributed system is able to continue to operate even when there is a partition.

Partitions are inevitable in a distributed system. So, it is impractical to design a distributed system that favours consistency and availability and trades off partition tolerance.

CAP theorem states that, in a distributed data system, in the presence of a partition you need to trade-off between consistency and availability.

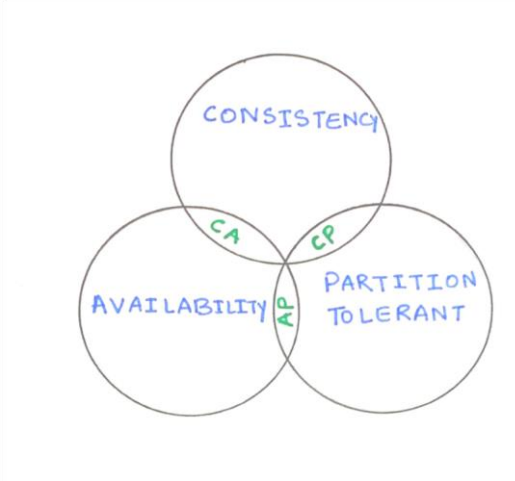


Figure 10: CAP Theorem

So, we predominantly see two kinds of systems:

CP systems:

In a partition tolerant distributed data system, if you favour consistency over availability, it is called a CP system. In a CP system if there is a partition then the nodes which are unreachable are removed from the system until the communication is restored. There are CP systems which make the entire cluster go 'down' or some which prevent writes or some which allow read and write operations only to main node which is inside partition. All are reasonable approaches.

For example: Let us say there are 3 nodes N1, N2 and N3.

An operation writes to N1. The system immediately replicates the data to N2 and N3. So whenever there is a read operation on any node the same data is returned there by ensuring consistency.

But let us say there is a partition with N3. A CP system will remove the node N3 from servicing any requests until the partition is resolved. This is done because N3 might be holding stale data and will not be able to update its copy of data until it is able to communicate with other nodes.

N3 is made unavailable thus trading off availability in favour of consistency.

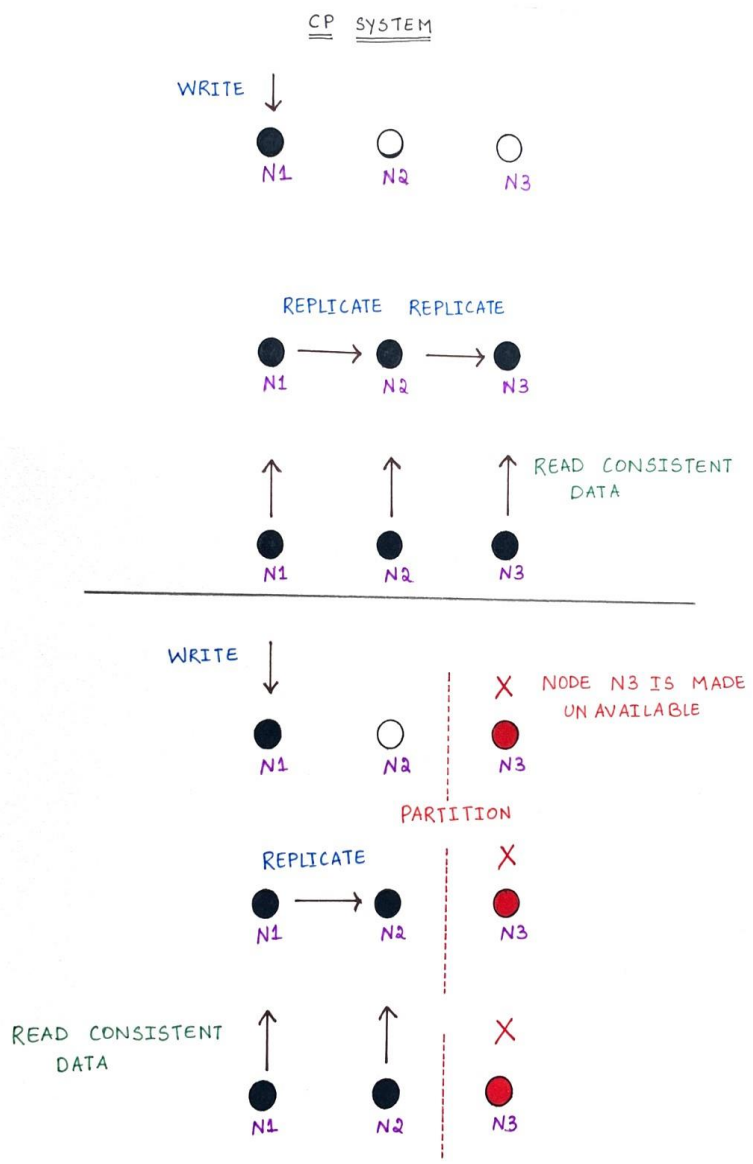


Figure 11: CP system scenario

AP systems:

In a partition tolerant distributed data system, if you favour availability over consistency, it is called an AP system. In an AP system if there is a partition then the nodes which are not reachable continue to operate with potentially stale data. The data might be outdated since there is no communication with other nodes and hence the responses might return inconsistent data. But the system is highly available since none of the nodes is taken out of service.

Let us consider our earlier example with three nodes N1, N2 and N3.

When there is a partition with N3, instead of removing N3 from the system, we let N3 continue to service requests with potentially inconsistent data thus favouring availability over consistency.

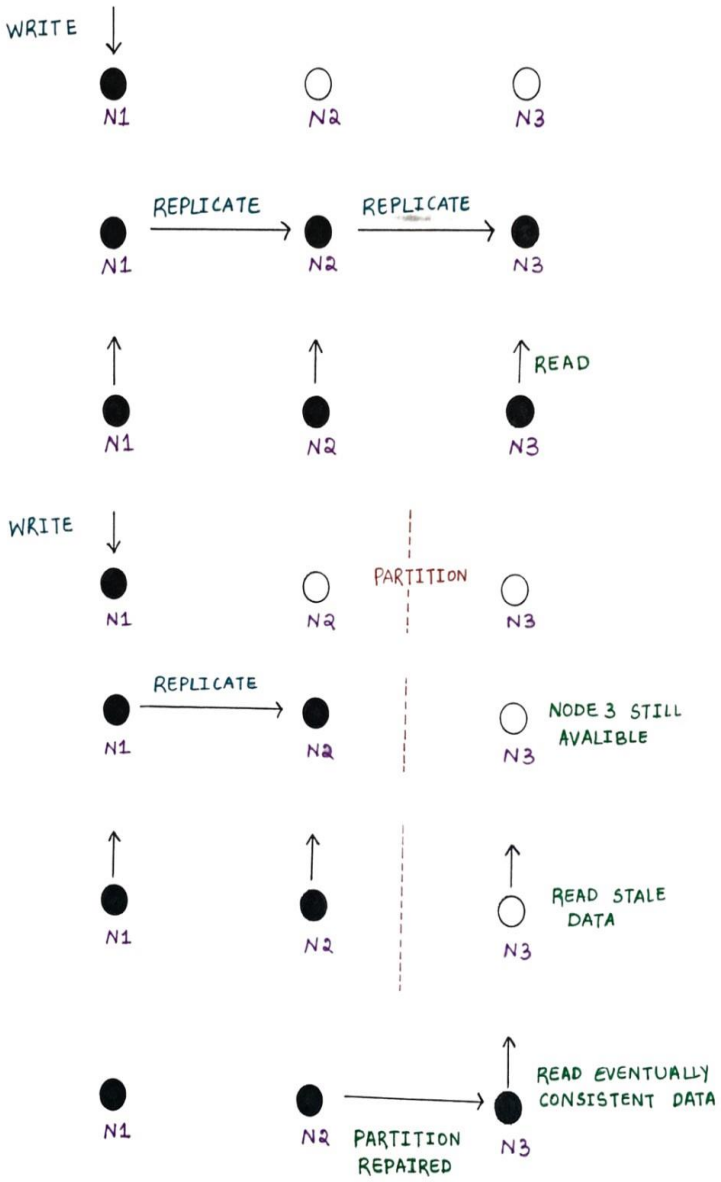


Figure 12: AP system scenario

While CAP theorem talks about the trade-offs between consistency, availability and partition tolerance, there are a few other data distribution strategies worth noting to help your application achieve better scalability, fault tolerance and availability while lowering the latency.

Read Replication

In a typical application the reads are more than the writes to the database. It is called the read write ratio. For applications with a high read write ratio, it is important to ensure that the application instances are able to read seamlessly without latency introduced by writes or other reads.

This can be achieved with read replication of data with the creation of additional nodes which act as replicas for read operation.

The data replication can be done synchronously or asynchronously.

- Synchronous replication
  During a write operation the data is written to a primary node. The success response is sent only after it gets replicated to secondary nodes. This obviously introduces higher latency but ensures high consistency.
- Asynchronous replication
  Once the data is written to the primary node the success response is sent. The replication happens asynchronously between main node and replicas. The latency is lower but it comes at a cost of lower consistency. There is a chance that the secondary node responds with stale data during a read operation because it has not yet received the update from primary node.

There are three ways to achieve read replication:

- Single primary node
  There is a primary node which takes in writes and there are multiple secondary nodes for reads. The data gets replicated from the primary

node to the secondary nodes. This strategy ensures better fault tolerance but low throughput because there is just one primary node. If primary node fails, then one of the secondary nodes is elected to be a primary node and thus ensuring better availability.

- Multiple primary nodes
  To improve throughput, you can set up the system so that there are multiple primary nodes spread across multiple data centres with replicas in each data centre.
- No primary node
  This is a strategy where there are no primary nodes and all nodes are primary nodes. It means all nodes are replicas that can take write operations. This is an asynchronous replication strategy thus providing very high fault tolerance, availability and low latency but compromising on consistency.

Sharding

With read replication you can ensure the system can offer better availability, fault tolerance and low latency. But what about scale? What if in your application the write volume increases due to business expansion and more customers are onboarded?

For example: Let us hypothetically say you are using a relational database and you have 100,000 customers. That corresponds to 100,000 rows in the Customer table. Now, the business grows exponentially and very soon you have a 1 million customers and then 10 million. So, 10 million rows in the same table does not perform as well as 100,000 rows. As discussed

before, you do some vertical scaling with better servers but that hits a ceiling at some point. You need other options.

Sharding offers a way to horizontally scale the database. You create a shard key and write the data to multiple nodes based on the shard key.

For example: A very trivial shard key for the above Customer example could be customer_id % 10. This divides the customers into 10 shards with 1 million customers each for a total 10 million customers. There are many advanced ways to create a shard key that can help split the data writes evenly across shards in an optimal way.

In NoSQL databases sharding is far easier than in relational databases because of the way data is structured. In case of relational databases, it can get quite tricky to shard data. You must be careful about the table joins and query strategy. You can split the tables to co-locate the related data into one shard and perform joins internally to the shard. You can also perform a cross-shard join or a distributed join but be aware of the performance penalty you pay with each trade-off.

Many relational databases offer excellent out-of-the-box support for sharding.

Sharding + Replication

You can use a combination of sharding and read replication to achieve a good trade-off between high availability, high consistency and low latency while ensuring good fault tolerance.

While designing your system for all the trade-offs discussed so far it is important to address security, monitoring and back-up & recovery.

- Security is at the core of all enterprises. All data at rest should be secured using strong encryption. All data in transit should use encrypted channel via TLS.
- Monitor file systems, CPU usage, memory usage, read/ write to disk, etc using standard metrics offered by all enterprise DB.
- Continuously back-up all data for disaster recovery.

Parting thoughts

So far in this book we have designed APIs for our microservices, managed inter-service communications and transactions. In this section we saw various strategies to handle distributed data systems. But the most important aspect of your application is its security which is what we cover in the next section.

Section 6: Microservices application security

With monoliths, applications and data typically reside in an internal data centre and businesses could secure it with one perimeter and have peace of mind. But with cloud all that has changed. But with it comes a change in the way you secure your critical assets as there are a lot more variables involved with cloud than with an in-house data centre. The attack surface is much larger on cloud. Below we about some of the best practices to reduce and control the surface area for potential attacks to secure APIs, applications, services and data.

Encrypt all data exchanged

HTTP transfers data in plain text. It means anyone listening in over the wire can extract all the information that gets exchanged as request and response between the client and the server. This is obviously a major security concern. All information exchanged over the wire must be encrypted. This is achieved with HTTPS and TLS. TLS handshake is used to exchange SSL certificates and establish a secure channel for information exchange. All data transfers only via this secure channel thus making it unreadable to any malicious 3rd party. It is imperative to enforce usage of HTTPS for all API endpoints.

HTTP Strict-Transport-Security response header can also be used to inform the client that all communication to the server must be using HTTPS.

Avoid using GET requests to send sensitive information.

Information sent in GET requests gets cached by browsers and gets logged by servers. Sending passwords or any other sensitive information in the

GET requests can lead to security leaks. Always use POST requests to send such information in the request body.

Authorisation and Authentication

Authentication is used to verify the user identity. Authorisation is used to verify if the authenticated user is allowed to access the said resource on the server to perform a given operation.

Moving from Session based security to Token based security

HTTP and REST APIs are stateless by design. Every request is treated as a new request. This poses a problem. How will the server know if the user is legitimate for every request without storing any state? If the application is monolithic there is a good chance that the user log in details is maintained in a session on the server and a session id is used to manage it. If there is load balancing involved then the sessions are possibly managed via a shared session cache or using sticky sessions.

It works fine until a certain threshold after which it does not scale. As the number of users increases exponentially the server must maintain more and more states which becomes unsustainable after a point. Also, session-based security does not work well with microservices architecture whose primary goal is to scale.

Token based security solves this problem by using tokens to pass information between client and server to enable scaling and not storing the user information on the server-side.

OAuth 2.0 (Open Authorisation) is an industry standard protocol for authentication and authorisation widely used by many enterprise applications. JSON Web Tokens (JWT) are used to encode and send claims as OAuth Bearer tokens in request headers. OAuth 2.0 and JWT form the basis of token-based security.

OAuth 2.0 supports authorization flows that address several scenarios for server-side (web server applications), mobile, device, desktop, client-side (JavaScript), machine-to-machine applications.

Authorisation Code Flow is used with the server-side web applications where the source code is not exposed to general public.

Steps involved in the flow are detailed in figure below.

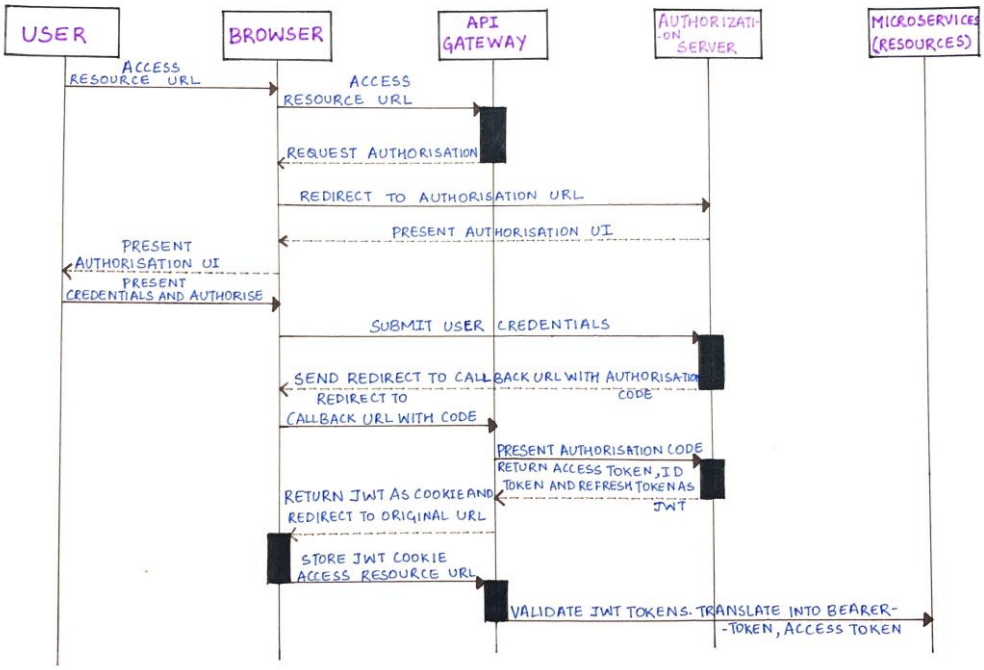


Figure 14: Authorization Code Flow

Apart from the Authorisation Code flow there are other flows supported to cover other scenarios.

Implicit Flow is used with client applications residing on user devices which cannot store client secret securely.

Resource Owner Password Flow is used when the applications are trusted and want to access each other's resources.

Client Credentials Flow is used in machine-to-machine authorisation where the application is authorised rather than the user.

JWT Tokens

JSON Web Tokens (JWT) are extensively used during the OAuth process and Single Sign On (SSO). They are a compact and secure way to transfer data between parties.

A JWT consists of three components – a header, a payload and a signature.

The header contains metadata information about the algorithm used to sign the token, type of the token, etc.

The payload contains the claims, user id, user name, token expiry, issued at, etc

The signature is a one-way hash using a hashing algorithm like SHA-256, header, payload and a secret key. So, the signature could be verified by only the server that issued the token with the secret key that only the server possesses.

There is no way to change the payload of the token since the signatures will not match when the server verifies the token.

Multi-factor authentication

Use multi factor authentication for additional security. It protects the application and the infrastructure in case of a password breach.

Dedicated Auth Server

Use a dedicated and trusted Authorisation Server to handle all authentication and authorization. Do not allow individual services or API gateways to create or validate tokens.

Zero Trust Policy

Microservices must not trust anyone else apart from themselves. Each microservice must validate tokens of incoming requests. Requests must have minimal scopes associated only with the target microservice and access tokens from user requests should not be forward chained to other services. Instead, a new token must be created for the next request in the chain with a defined scope.

This approach helps a fine-grained control over the security of individual services.

API Gateways

API gateways act as entry points to the application. They provide an added layer of security between the client and the services and protect the resources from outside world.

The requests hit the API gateways and then they are transformed and transferred to the actual microservices. So, malicious requests can be spotted and filtered.

API gateways can protect the microservices from DoS (Denial of Service) attacks, SQL injections, and other such security threats because they are designed to add rate limiting, throttling, circuit breaker and handle incoming traffic elegantly.

They can also act as an intermediary to exchange incoming JWT tokens to get the bearer tokens with the Authorisation Server and forward them to the microservices. It is a good practice not to forward the incoming tokens directly to the services.

Service Mesh

While API gateways handle the north-south traffic, service meshes are designed to take care of the east-west traffic (the traffic between services internal to the infrastructure).

Service mesh abstracts a lot of cross cutting concerns and offer capabilities like observability, traffic management and security without having to manage them in individual services. Some of the benefits include mTLS (mutual TLS) to encrypt the traffic between services, routing, access-control, canary deployments, monitoring, logging and even authentication.

Services mesh frameworks are evolving to act as API gateways too. In the future it would not be surprising to see a unified solution.

Audit, Log and Monitor

Audit everything regularly with an expert group guiding the security infrastructure. Monitoring and logging are offered by cloud infrastructure, API gateways, service meshes and independent open-source products.

Container security

Container runtimes can lead to security breaches if deployed with a wide surface area for attack. Use the following strategies to reduce the potential attack surface and to put checks to ensure safety.

- Use a thin host operating system.
- Always run your containers in the no-new-privilege mode.
- Create containers with non-root users.
- Run containers with read-only, where applicable.
- Do not pack in credentials in the image. Use container orchestration volumes or environmental variables to pass such information to container runtime.
- Use a thin base image for creating containers.
- Check containers for any violations in the common security best practices before deploying.
- Scan container images regularly for any vulnerabilities.
- Use a trusted base image to build containers.
- Set up security gates at every step of DevOps like develop, build, test, package, deploy and operate to check of any vulnerabilities.

A more detailed analysis is part of Section 7 on build where containers are discussed.

Defence in Depth

So far, we have covered several best practices to secure the APIs, applications and services running in cloud. But what about securing the cloud itself? How do we do that? Do we set up a strong firewall at the entry point and negotiate all traffic from there? What if there is a leak? Should we not diversify like what we do with our financial investments to mitigate risks? Defence in Depth is a layered security defence strategy to avoid single points of failure. It originates from military strategy where the most critical assets are protected with concentric layers of security. Defence in Depth employs several layers of security mechanisms to safeguard cloud resources.

- Physical Layer – Physically secure the cloud data centres by controlling who has physical access to the data centre facility premises, the floors and the data centre server labs. Most Cloud Service Providers (CSPs) have well defined processes to guarantee robust security and compliance to international standards.
- Access Policy Layer – Define the access policy to the cloud resources with Role Based Access Control (RBAC). Use deny-by-default approach and review access privileges regularly for both the cloud administrative teams accessing cloud resources and the users of applications accessing data and services of the application.
- Perimeter Layer – It is sometimes also called the edge because it is the boundary that separates the cloud from the outside world. It is here where you set up your firewall, Intrusion Prevention System (IPS), Intrusion Detection System (IDS). Most CSPs provide built in

protection against DDoS attacks, SQL injections and popular attack patterns with the help of artificial intelligence.

- Network Layer - Secure the open ports and IP addresses accessible to the external internet. Set up logical boundaries to your network and set up network security groups.
- Compute Layer – Compute layer covers the VMs or the nodes. Secure it with antivirus and anti-malware protections. CSPs provide extensive options to secure the compute layer.
- Application Layer – Build security as an essential part of Software Development Life Cycle (SDLC). Integrate security gates at every step – vulnerability testing, static analysis, dynamic analysis and vulnerability scans. Security context should be built into each microservice with a Zero Trust Policy and Deny by Default Policy. Runtime instances should be monitored, logged and audited.
- Data Layer – It is the most important asset that you are securing and therefore it is at the centre of the concentric layers of security we build. Secure all data at rest and in flight. Encrypt data persisted on the file system. Encrypt data sent from and to users and between services. Use data masking for sensitive data like credit card information.

With these layers acting together as concentric circles of defence layers, we create a formidable deterrent against any attacks and protect our key assets.

Parting thoughts

We have covered several security best practices for your microservices in this section. With this we have come to the end of the sections that deal with the development part of DevOps application lifecycle. In microservices architecture, there are many areas of operations like compute, storage, networking and monitoring that development teams must closely work with. We look into it in the next sections.

Section 7: Build

We have now evaluated key considerations and trade-offs while we design our microservices application. All the topics discussed so far are part of the design, develop and test phase of the DevSecOps lifecycle. Traditionally, development teams for monoliths do not have to worry about very complex build and deploy phases. The build and deploy architectures for monoliths tend to be far simpler compared to microservices.

A good understanding of build infrastructure, deployment architecture, number of instances running, load balancers involved and where to look for logs, is all that you would bother about as development teams with most monoliths.

But in microservices, development teams need to know a lot more about build and deploy, given that it is a distributed architecture. As a development team you want to –

- Define clearly what you want in your final package. So, be involved in creating Dockerfile and how it is used in the build phase to create containers.
- What your container certification and signing requirements are. For example – Do you need your container to be Open Container Initiative (OCI) compliant? Do you need it to be certified?
- Know what your production performance baseline is. That means, you need to define together with your extended team what are the number of requests per second that your instance can handle in

production and what are the number of instances required to run at any given moment. Then create a baseline deployment architecture.

- Define where and how the containers are run. Is the production running it on a bare-metal Kubernetes cluster, hyper scalar managed Kubernetes cluster, some container service, etc. Each one will have a different deployment definition and the development team either create those definitions or be very closely involved in creating them. For example – for microservices deployed on Azure Kubernetes Service (AKS), you would want to create Helm Charts that deploy the containers with all the required Kubernetes service definitions, secrets, configmaps, persistent volumes, roles and role bindings.
- Define a baseline architecture. How does a user request ingress into the application? IS there an application gateway required? What about a service mesh? Do you want your services to be exposed with a public IP or do you need it all to be private IP and only expose functionality via the gateway? It is primarily the job of the application architect to define all this.
- What are the key application parameters you want to measure? Do you know where to look if you need the logs for a given instance of your container?

There are many other considerations, but you get the picture. In microservices architecture, teams need to synergise a lot during the design and architecture discussions. It is necessary that everyone from development, test, operations, site reliability engineering teams are on the same page.

DevSecOps is a vast area on its own and deserves a separate discussion. In this book we are going to focus on sections of DevSecOps that you as an application architect must know and be closely involved in.

Now that you have created your microservices application, it is time to build and containerize it.

In this section we will look at some of the best practices for containerizing your microservices application.

One of the main reasons why teams move to microservices architecture is the time-to-market advantage it offers. Given that microservices are light weight, it translates into smaller builds and thus shorter time-to-market. In this section we are going to look at the build phase of your DevSecOps lifecycle.

Fig 1: DevSecOps – highlight build

Continuous Integration and Continuous Delivery (CI/CD)

Continuous Integration (CI) are a set of best practices for development teams that encourages them to add small code changes, test them and integrate them with a central repository of version control system. Continuous Delivery (CD) is a mechanism to take the code from the central source code repository, compile, test and package it into a deployable form. It sometimes extends its scope to deploying the packages into a test or production environment based on what your needs are.

CI/CD pipeline is a set of tools which automate the end-to-end lifecycle including the pulling of the code from the repository, compile, test, package and deploy stages. Based on individual requirements, the CI/CD

pipelines can vary a lot. But below is a diagram that shows the most common components of a CI/CD pipeline

Fig 2: CI/CD pipeline

The build server pulls the source code from the source code repository on a trigger or by polling or at regular intervals. Then there are several stages for compile, test, package and containerization. You can also add a post build stage to push the container to a registry or to install it on a test environment. There are many excellent choices for the build server component like Jenkins and GitHub Actions.

Though we have endeavoured to be technology agnostic in this book it is prudent to discuss some key technologies involved in CI/CD pipeline for our microservices. Docker containers are a necessary discussion for any microservices architecture. Over the years Docker has become synonymous with containerization. There are other containerization providers like RKT and CRI-O. For the purpose of this book, we will focus on Docker.

Microservices are typically built and run as containers. So, it is expected that the CI/CD pipeline creates a container as its output. Of course, you can customize it to push the container to a container registry or deploy it to a test VM running docker engine or maybe to a staging environment to perform further tests. But for the sake of brevity let us confine to building containers as the final output of the CI/CD pipeline.

Let us first get some definitions out of the way. Containers are lightweight software packages which contain all dependencies including libraries required to execute in an isolated environment. They are different from virtual machines (VM) in the fact that VMs contain full-fledged operating

system (OS image) and thus are heavy and take time to start. On the other hand, containers are bundled only with application and the dependent libraries over a thin base OS container image without the kernel and thus are light weight and can start up really fast.

Docker is a Platform-as-a-Service (PaaS) software that helps you create and run your containers easily. Docker builds images which are read-only files with instructions to create containers. Containers are runnable instances of docker images. Container images are stored on a container registry like Docker Hub.

Dockerfile

When you build your applications, you first build your binaries. For example, you would build a jar file for a Spring Boot application using Maven or Gradle. After this you use a Dockerfile to build an image. Dockerfile is a simple text file which has a set of instructions to build the Docker image.

Dockerfile best practices

Let us start with a simple Dockerfile which creates a container for a Spring Boot application.

```
FROM eclipse-temurin
ARG JAR_FILE=target/*.jar
COPY ${JAR_FILE} application.jar
CMD ["java","-jar","/application.jar"]
```

But this is a very basic Docker file with several bad practices and anti-patterns. Below are some of the best practices to improve it and create a production ready Dockerfile for the said Spring Boot application.

- Use deterministic images with specific version tags and SHA256 hashes. The build needs to be idempotent meaning every time you build the image it should create the same image for the same Dockerfile. For example: Specify which version of the base image you want to use.

```
FROM eclipse-temurin:11
ARG JAR_FILE=target/*.jar
COPY ${JAR_FILE} application.jar
CMD ["java","-jar","/application.jar"]
```

- Use thin images with only the packages that you need. For example: When running a Java application, you just need a JRE and not a full JDK. So, FROM eclipse-temurin:17-jre (size approximately 90MB) is preferred to eclipse-temurin:17 which is a JDK image (size approximately 240MB).

```
FROM eclipse-temurin:11-jre-
alpine@sha256:1135fe43b09ac3d3201b5804aae868291a02d84910c43c7291
4c33d18a34d26346
ARG JAR_FILE=target/*.jar
COPY ${JAR_FILE} application.jar
CMD ["java","-jar","/application.jar"]
```

- Understand the difference between ENTRYPOINT and CMD. The default arguments provided by CMD can be overridden, but not the

ones provided by ENTRYPOINT. So, when you want your container to mandatorily do something use ENTRYPOINT.

```
FROM eclipse-temurin:11-jre-alpine@sha256:35fe43b09ac3d3201b5804aae868291a02d84910c43c72914c33d18a34d26346
RUN mkdir /opt/app
WORKDIR /opt/app
ARG JAR_FILE=target/*.jar
COPY ${JAR_FILE} application.jar
ENTRYPOINT ["java","-jar","application.jar"]
EXPOSE 9082
```

- Use WORKDIR to use a particular directory as an operating point for your commands. For example, when you set the WORKDIR as /opt/app and COPY your jar file, it automatically copies it to the WORKDIR. Likewise, when you execute java -jar application.jar, the jar from /opt/app is what is picked up.
- Use a non-root user with least privilege to run the service. You do not need to run containers as root user.

```
FROM eclipse-temurin:11-jre-alpine@sha256:35fe43b09ac3d3201b5804aae868291a02d84910c43c72914c33d18a34d26346
RUN mkdir /opt/app
RUN addgroup --system nonroot && adduser -S -s /usr/sbin/nologin -G nonroot nonroot
WORKDIR /opt/app
ARG JAR_FILE=target/*.jar
COPY ${JAR_FILE} application.jar
RUN chown -R nonroot:nonroot /opt/app
```

```
USER nonroot
ENTRYPOINT ["java","-jar","application.jar"]
EXPOSE 9082
```

- Minimize the number of layers. RUN, ADD and COPY commands add layers. Combine multiple RUN commands into a single one where it makes sense. Notice how addgroup and adduser are clubbed into a single RUN with &&.
- Use only trusted images. Scan containers for security loopholes and perform a thorough vulnerability assessment. Scans are auto-enabled in most container registries. For example: Microsoft Defender for Containers provides comprehensive vulnerability assessment for Azure Container Registry images. Docker Hub provides a feature to automatically scan for vulnerabilities in the images uploaded to it.
- Use multi stage build. It helps to separate the build time dependencies from the runtime dependencies and to create smaller images by packaging only what is required.
- Use .dockerignore file in the same folder as Dockerfile to include only what you need in the container and ignore the rest.

Parting thoughts

In this section we have covered some key considerations and best practices to containerize your microservices. After the build phase, we now turn our focus to the deploy phase.

Section 8: Deploy

Now that you have built your docker containers and pushed them to a registry, you want to deploy them on cloud and start running them, first in testing environment, then in staging and if all goes well, eventually in production.

Before we start, it is important to emphasize that there are many ways to run an enterprise application in cloud. Major areas to consider are workloads, compute, storage, networking, business continuity and disaster recovery, Identity management, and security.

Workloads and Compute

Enterprise applications could be run on a VM, as a container on a container engine (bare-metal or managed), as a batch application on cloud or be hosted as a serverless function. These are all pretty common ways to run workloads on cloud depending on the nature of the workload and what your goals are.

In this section we will focus on using Kubernetes as orchestration framework for our workloads. The goal is to run cloud-native containerized enterprise applications with high scalability, availability, and with an ability to move across cloud service providers with ease. Kubernetes is open source and offers much finer control over workloads. It is very widely supported by all major cloud service providers which makes it an excellent choice to run our workload.

A typical Kubernetes deployment architecture looks as below
Kubernetes architecture

Kubernetes runs containers in Pods. Pods are the smallest deployable units of computing in Kubernetes. Kubernetes architecture works on control-loop concept. You declare the desired state using workload resources. Kubernetes continuously drives the actual state of the system towards the desired state. Pods are usually not declared directly. You do that using workload resources created by Kubernetes APIs to representing a higher abstraction than Pods like Deployments, StatefulSets or DaemonSets.

Deployments

Deployment is the most common way to run stateless applications. Any Pod that dies is rescheduled and replaced to meet the desired state as defined by the 'Deployment' object. There is no state preserved between Pod reschedules. If the Pods use Persistent Volume Claims (PVC) they are shared across Pods and therefore must have ReadWriteMany or ReadOnlyMany access modes. Pod identities are not sticky.

Some important considerations while planning Deployments for your application –

- Add a start-up probe, readiness probe, or a liveness probe to understand if your container has started, is ready to serve requests or is too busy doing something that it has a broken internal state and needs to restart.
- Add container lifecycle hooks like post-start and pre-stop to execute scripts or to call API endpoints while starting or stopping containers.

- Add Resource Quotas, resource requests & limits and LimitRange to define resource allocation like CPU and memory per namespace and per Pod so that you do not exceed intended consumption.
- You can define multi container Pods. Each Pod can have more than one init containers, side cars, adapters or ambassador containers supporting the main container. But one container Pods are the simplest and easiest to deploy and debug.
- Add a HorizontalPodAutoScaler to scale up or down Pods based on certain conditions like CPU or memory utilization.
- Define an update strategy. If opting rolling updates then define maxUnavailable and maxSurge.
- Define deployment strategy like canary or blue/green. Know that when performing a canary deployment, you are limited by the number of Pod replicas. It means if you want to send 1% of traffic to v2 and 99% to v1, you will need to deploy 100 replicas, irrespective of whether your traffic requires it or not. A better approach is to use a service mesh like Istio.

StatefulSets

A StatefulSet is used to run stateful applications. Pods deployed using StatefulSets have a sticky identity. This means that Pod identities and DNS names remain consistent across reschedules. Volumes are mounted using volumeClaimTemplates which attach a volume per Pod instead of sharing it across all Pods like in case of Deployments. When and if a Pod is rescheduled, the corresponding volume is reattached thus helping to maintain state of applications. The key considerations explained in Deployments are also applicable to StatefulSets.

DaemonSets

A DaemonSet defines Pods which run on all or some (depending upon taints and tolerations) Kubernetes Node. It is useful to run Pods to provide a Node level functionality. A common usecase is to run a log aggregator for every Node to collect logs from /var/log and send it to a database.

Jobs and CronJobs

Jobs are used to run one or more Pods with continuous retries until a given number of Pods succeed.

CronJobs are used to run Jobs on a schedule.

Storage

Pods are ephemeral. When Pods get restarted, they begin anew. So, if you want data stored, retrieved and persisted between Pod restarts you need to use Kubernetes Volumes. Volumes are mounted on to Pods using volumeMounts.

There are many kinds of Volumes supported by Kubernetes.

ConfigMaps

ConfigMaps provide a way to load configuration data on to a Pod. This is a great way to supply your application with information like names of other services that your container needs to call. A ConfigMap resource needs to be defined before mounting it as a volume or adding it as a environment variable into Pods. ConfigMaps are defined for a given namespace and accessible to Pods only within that namespace.

emptyDir

Use an emptyDir where you want to mount a directory to a Pod so that it is accessible to all containers inside the Pod. Containers in the Pod could share information using emptyDir, or it can also be used to cache temporary data or a Pod could use it like a working column for internal calculations. emptyDir lasts only the lifecycle of a Pod and it is deleted when a Pod is deleted. emptyDir is usually provisioned from the underlying host node's filesystem although it can only be made to exist in-memory.

hostPath

Use hostPath to mount a directory from host node's filesystem to your Pod. It is best to avoid this type of volume mount. Very rarely do Pods need to mount volumes from underlying node filesystems. One example where you need to use hostPath volumes is when doing log aggregation using a DaemonSet to collect logs from the /var/log/* location of a Node.

Secrets

Secrets are a way to send sensitive information to Pods. Secrets are Kubernetes resources created and mounted using tempfs. So, they are not persisted to a disk and they are read-only. Just like ConfigMaps, you need to define Secrets before consuming them in Pods. Secrets are defined for a given namespace and can be consumed by Pods in that namespace.

PersistentVolumes (PV) and PersistentVolumeClaims (PVC)

PVs and PVCs together provide a way to abstract the storage from the workloads. Workloads do not need to know the details about the storage,

who is provisioning them and how. A PV is a piece of storage provisioned in the Kubernetes cluster either statically by an administrator or dynamically using StorageClasses. A PVC is a way for a Pod to claim that storage. Together they 'hide' the storage details from Pods.

StorageClasses define the provisioner (usually from the cloud service provider), reclaim policies, the class (like premium or standard) of the storage, etc. A PVC defines the storage requested, the accessMode and the StorageClass from where the storage is requested.

Best practices

Avoid statically provisioning your volumes. Use StorageClasses and dynamically provision for better scalability and on-demand volume expansion.

Back-up your volumes and use encrypted storage. All data on storage is encrypted by default in major cloud service providers.

Define the size and type of storage required by your application. You need to clearly answer questions like -

Do I need ReadWriteMany or ReadWriteOnce?

What size of storage do I need and what do I want to store there (structured or unstructured files)?

Do I want to use standard or premium storage?

Networking

Networking is perhaps the most vital cog in the wheel of the Kubernetes juggernaut. Networking is what connects the individual entities in the cluster and defines how they interact with one another and with the outside world.

Pod is the smallest deployable unit of computing in Kubernetes. Kubernetes assigns an IP to each Pod. So, Pods can 'call' each other using these IPs on a given port to reach an application endpoint. But the problem really is that the Pods are ephemeral which make their IPs ephemeral. That means Pods can be destroyed and recreated. So, let us consider a front-end application reaching a backend microservice. This microservice is deployed as a set of 3 replica Pods using a Deployment. Front-end is connected to one of the replicas (say replica-0) using its IP. Now, replica-0 crashes and it is restarted. Its IP changes because, in principle, it is a new Pod that is reprovisioned on restart. The connection is broken between the front-end and the backend microservice. The front-end will not know the new IP.

To solve this problem Kubernetes introduces the concept of Services. A Service is an outward facing abstraction to expose a group of related Pods running as replicas. A service is given an IP of its own and it is not affected by Pod restarts. The front-end can use the service IP or the Fully Qualified Domain Name (FQDN) to reach the back-end Pods. FQDN for a service with a name 'my-service' in a namespace 'my-ns' will be of the form 'my-service.my-ns.svc.cluster.local'.

You tie Services with Pods using selectors, typically matching labels or matching expressions.

There are several service types to help handle different scenarios –

ClusterIP

This service type is used to create services that are internal to the cluster. You can then choose to reach these services using Ingress or Gateway.

NodePort

Expose the service at each node's IP at a given port.

Load Balancer

Expose the service to external consumers via a publicly accessible IP.

External Name

If you want to configure a service to reach external applications you can set that up with ExternalName service. You create a mapping from the service to a DNS entry for the external app.

Headless Service

You can create a service with clusterIP: None. Kubernetes does not create an IP for the service but maps the service directly with the backing Pod IPs. An array of all Pod IPs is returned and you will need to select the Pod you want to connect to.

For example: Headless services are used with StatefulSets which have 'sticky' Pods. You can choose to connect to one of the Pods using Pod FQDNs which are of the form 'my-pod{0-(N-1)}.my-service.my-ns.svc.cluster.local' where N is the number of replicas.

Ingress

Kubernetes supports creation of Ingress resources to manage http traffic flow into your web applications. You can define the host, path, port and the back-end service that gets called for a given path. You can define all services to be internal to the cluster using ClusterIP service type. Then configure an Ingress resource to map incoming http requests to appropriate services thereby setting up a single entry-point to your cluster.

With Ingress, you get better control over your cluster and better security with a reduced attack surface. You can also set up TLS termination at Ingress.

While you define an Ingress resource, you need to set up an ingress controller in your cluster and declare that in your Ingress object.

Gateways

Kubernetes Gateway resources are similar to Ingress but offer finer control over configurations.

API Gateways

API Gateways add another layer of abstraction to Ingress resources. All major cloud providers offer their own API Gateways. API Gateways are a great way to consolidate configuration for complex web applications that might be distributed across many AKS clusters and say several auto-scaled VM sets for images, videos and other static content.

Additionally, API Gateways offer in-built Web Application Firewalls (WAFs), protection against Distributed Denial-of-Service (DDoS) attacks, and TLS termination. You can define your ingress controller with a private IP. And then wire this IP to the App Gateway. That way everything in the cluster is internal and nothing is exposed to the internet. The only way to access the services would be through the App Gateway. This configuration boosts security by making the attack surface really small and properly policed.

It is a best practice to set up the App Gateway in a separate virtual network from the Kubernetes cluster virtual network but connect them (for example using peering).

Network Policies

Network Policy is a way to control traffic flow into and out of a Pod at OSI layer 3 or 4. By default all traffic is allowed into and from Pods. You can define ingress and egress rules/ policies which define the traffic that is allowed from/ to IP blocks, Pods or namespaces and on what ports.

In a nutshell, Kubernetes networking allows you to –

- Connect containers within a Pod using loopback (like localhost). Pods behave like a VM and containers are like processes running on that VM. So, containers within a Pod cannot not share the same ports.
- Expose application back-ends using Services to external consumers outside the cluster.
- Expose Pods within the cluster, only internally, using Services.

- Use Ingress to expose web applications to outside the cluster at OSI Layer-7. Services operate at Layer-3 and Layer-4.

Security

Security of microservices applications on cloud must be looked at from a holistic point of view rather than a stand-alone basis. When you run microservices, you typically run them as containers on a Kubernetes cluster hosted on a cloud provided by a Cloud Service Provider (CSP). So, it becomes important to understand the baseline security considerations before deploying your applications.

Security on cloud is shared responsibility between the CSPs, the cluster operations team and the development and testing teams. Kubernetes proposes the 4Cs model for application security that includes Cloud, Cluster, Container and Code. Code, which represents your application and data security is covered in the Section 6: Securing your microservices applications. Container security is covered in Section 7: Build.

Cluster Security

Kubernetes cluster security, although is not a direct development responsibility, you do have some important contributions to make there. There are Kubernetes resources created by you and your team which can affect the cluster security if configured improperly.

When you write Kubernetes yaml files for deployment of Pods or other resources, keep in mind below considerations –

- Security Context is a property set in the manifest YAML file that defines the context for pod or container or volume security. For example, you can run a pod as a non-root user or set the root file system as read-only. This helps to reduce the attack surface drastically. If you have set a non-root user in your container image the Kubernetes security context property for runAsUser will override it.
- Know the configuration set for the pod security admission controller to understand what pod security standards are defaults. Typically, this is a cluster wide configuration set by your operator, but knowing it helps you to configure the right pod security admission labels in namespaces and pod security context.
- For Pods which need to read or create other Kubernetes resources, create Service Accounts with appropriate Role Based Access Control (RBAC) role bindings. Use least privilege permissions while setting roles and role bindings to service accounts.

Kubernetes User Authentication, Authorization (RBAC) and Admission Controllers configuration is more operations driven than development driven and therefore not discussed in this book. But to know more about the best practices around these topics, Kubernetes documentation is an excellent resource.

Cloud Security

Most elements in cloud security are provided by the cloud service providers (CSPs). But it helps to know the key aspects in cloud security that are relevant to you, just so that you can enable them.

Compute

- Antimalware and antivirus protection must be installed in all compute instances.
- If your application stores app data on the compute machine, take regular back up of your application data from your VMs.
- Encrypt OS and data disks.

Storage

- Encrypt data at rest and data in transit.

Networking

- Use virtual networks and network security groups or network access control lists (depending upon the CSP nomenclature) to ensure isolation and access control at OSI layer -4.
- Use application gateways and application firewalls to expose APIs to external internet. Secure all other parts of your application by making it private to the cluster where it runs.
- Use VPNs to connect to your on-premises network.

Identity and Access Management (IAM)

All major Cloud Service Providers (CSPs) offer excellent IAM functionality to set up user identity and access control of cloud resources based on roles. Be sure to understand what your CSP offers and what the best practices are to take full advantage of the infrastructure. CSP specific IAM discussions are out of scope of this book.

If you are using bare metal Kubernetes cluster, your organization will need to set up your Kubernetes authentication using OIDC (recommended for

production clusters) or client certificates. Then use Kubernetes RBAC for access control.

With all layers including cloud, cluster, container and code secured you must achieve what is called ‘Defence in depth’ which is discussed at the end of Section 6: Securing your microservices applications.

Multi-tenancy

Kubernetes provides Namespaces to create logical isolation between workloads. So, if you are using development namespace and a staging namespace you can use a single cluster with different security policies (like RBAC or Pod Security Policies) enforced on different namespaces. This helps to optimize costs while not compromising on security or operational efficiency.

Business Continuity and Disaster Recovery

You deploy your production workloads on a Kubernetes cluster to ensure high reliability. If a Pod went down it is reprovisioned and restarted. But what happens when an entire cluster or worse a region goes down. To ensure business continuity, choose to deploy your production workloads in multiple clusters in multiple regions and availability zones.

Take regular back-ups of any persistent data storage and replicate the data between regions asynchronously to ensure eventual consistency.

Baseline architecture

With all the due considerations above, we arrive at a baseline architecture for deploying and running your microservices on cloud. We make certain assumptions with the baseline like you are using a CSP and you are the administrator for the virtual network where the services are getting deployed on your Kubernetes cluster. In the real world, these roles are often split into multiple personas if not teams. But nonetheless it is a starting point, which will help you and your team to understand the complete picture of what is required to run a production workload. You can then customize it based on your individual requirements.

The terminology used here is intended to be CSP agnostic. If you are a GCP or AWS user you will replace the term Virtual Network with VPC or if you are an Azure user you will be familiar with VNet. But the concept remains the same.

Baseline architecture.

Conclusion

In this section, we have learnt what the key considerations are to define the baseline architecture for Kubernetes orchestration of microservices. With that said there is also an option to introduce a service mesh into your baseline architecture. We evaluate how a service mesh can influence your design in the next section.

Section 9: Service Mesh

When you deal with a monolithic application you are more focused on the traffic that goes in and out of the application. This traffic is called the north-south traffic. Your security goals are to secure the application and the data (in transit and at rest).

Microservices are by design distributed. While the merits of microservices architecture are many, it brings with it new challenges. Challenges like — how are these microservices going to communicate with each other, is the communication expected to be encrypted, observability (metrics, logs, traces), traffic routing between microservices, and other things.

Most of these new challenges are cross-cutting concerns that can be addressed outside the application to eliminate boiler plate configurations as much as possible. And this is how service meshes were born. A service mesh is a (pretty aptly named) solution that manages the east-west traffic. i.e., the traffic between services.

There are several open-source solutions out there with the top three slots taken by Istio, Linkerd and Consul. Istio is by far the most popular one and is the one we are going to use for evaluation.

Istio comes packed with many useful features for traffic management, security and observability. Istio works by injecting an envoy proxy into each pod as a side car.

Traffic management

Istio's Gateway, Virtual Service and Destination Rules provide creative options to manage traffic between services. You can set up routing rules, traffic split between different versions of your service, circuit breakers, load balancing and rate limiting.

If you are doing canary releases on your production, you will know that using primitive Kubernetes resource definitions is very cumbersome. Firstly, the percentage split of traffic between versions is tied to replicas. For example, if you want to direct 10% traffic to a new version you need 10 replicas, 9 old and 1 new. But if you want to split 1% traffic to new version you need 100 replicas. It is not viable on production. Istio makes life on production much simpler due to its ability to route traffic without relying on number of replicas. You can also split traffic based on users like only users belonging to a user group will route to v2 and others route to v1.

I found the traffic management aspect of Istio to be very mature. The value proposition and the return on investment add up.

Security

When services communicate with each other there could be requirements to secure it with encryption, especially if it is a multi-tenant cluster.

Istio provides Peer Authentication, Request Authentication and Authorization Policy resources to handle authentication and authorization. It has very good in-built support for certificate

management, mutual TLS, token validation, and to allow or deny a request based on authorization conditions.

Peer Authentication

Out of the box support for mTLS is a very useful feature indeed. It has its uses like in a multi-tenant cluster. But evaluate if you really need it. If your cluster is single tenant where you have full control, most applications are better off with a public Application Gateway supported by a Web Application Firewall (WAF) while the application and all its services are private. So, no part of your application is exposed to internet and the only entry point is via an App gateway. It is secure enough for most businesses.

Additional encryption imposes latency / performance penalty. Evaluate carefully if it is necessary.

For example: A service has a 99-percentile latency of 90ms and the acceptable latency is only 100ms. If you add inter service communication encryption it introduces a delta. You need to benchmark the additional latency and if it is a necessary trade-off for added security. Not that more security is bad, but there is always a trade-off.

Request Authentication and Authorization

Your service must handle request authentication and authorization to some extent to enforce a robust zero-trust policy. While it might be a cross-cutting concern and delegating it to a service mesh might make life easier. There sure are arguments, and good ones, in favour of service mesh led security. But the stakes are way too high to let security go out of scope of

your application. With zero-trust policy, your application must not trust anyone apart from itself. You can use some network policies or Istio authorization policies to control what services are allowed to interact with what services and when, to optimize performance.

However, there are some cases where it might be tolerable for applications to delegate all their auth workflows to a service mesh. You need to evaluate what works for you.

In Istio some customisation to authentication and authorization to cover some corner case scenarios are not supported. For example: You want to validate if a user request has got a token and return 401 if it does not. This use case is supported but then you cannot add a WWW-Authenticate header to let the client know what to do. Well, there is still a way to add an EnvoyFilter to do that, but then it gets more complex from there.

Observability

For a distributed architecture it is very important to have a way to log, collect metrics and enable tracing.

Istio comes with addons for Prometheus, Grafana, Jaeger and Kiali. For access logs, Istio can be configured using the Telemetry API to log envoy access logs to stdout.

Although observability is well supported, more is expected in production like easy integration with tools like Datadog or Dynatrace. So, if your

observability scenarios have specific requirements, it is important to confirm that those are supported.

Performance

While Istio and for that matter any service mesh offers great benefits it does come with a performance penalty. Every request needs to be routed through envoy proxy and that adds to latency.

It is recommended to benchmark for throughput and latency for all your use cases before investing in a service mesh.

Parting thoughts

Like any technology service mesh is not a one size fits all solution. Some might find all features useful; most will find some features useful and some will find that they are better off without a service mesh.

Should you feel the need to use some of the capabilities offered by Istio or other service meshes be sure to confirm the use case is supported, benchmarked and latency requirements are met.

It is a relatively new technology which is still evolving. It has good community support and a promising future. With that said we move on to our next section on Observability.

Section 10: Observability

In microservices, owing to its distributed architecture, it becomes difficult to pin point a problem to a given service and to debug it. Therefore, it is necessary to observe the entire system from end-to-end, especially in production, to effectively troubleshoot.

Is this not a responsibility of an SRE (Site Reliability Engineer)?

Yes and no. You as an application architect or developer are equally responsible for how your application performs in real time. You are a stake holder and need to be involved in the decision-making process of what is measured and how it is reported. You obviously do not want to be woken up at midnight to be told that the application has started crashing and you have no idea where to look because you did not configure the monitoring systems. So, a good first-hand knowledge of your observability set up is in order.

So, then what is observability?

Observability is the ability to infer internal state of a system with external attributes that it exposes or provides. Metrics, logs and traces form the three pillars of observability. The key objective of observability is to ensure that the system is performing and operating as per expectation.

Metrics

Metrics are key performance indicators (KPIs) of your services which are collected over a period of time. They can indicate how well the system is performing, like if there is a delay in servicing requests, number of requests failing, etc.

Depending upon their nature different services might expose different metrics.

In synchronous web based transactional microservices some of the KPIs will include Apdex score or User satisfaction score, error rate, latency (say, time taken to service 95% of requests), and request rate (shows if the traffic is increasing).

In asynchronous event driven microservices, KPIs will include total number of messages processed, error rate, latency, throughput (number of messages processed successfully per second), number of messages being processed at a given time, and last time a massage was processed successfully. Similarly, if a microservice is sending messages KPIs may include total number of messages sent or last time a message was sent.

For batch jobs, a very important KPI will be last time when the batch job ran successfully.

Some other system metrics that will be of interest are CPU usage, memory usage, server stats (like total number of active sessions, number of server requests, time taken by requests), cache stats (like total requests, misses, etc), and garbage collection stats.

There are a lot of possible metrics that can be collected. It really depends on what you want to track and to what extent. The metrics detailed above are probably the bare minimum you must track and care must be taken not to overdo it. Otherwise, you will be introducing unnecessary overhead.

4 golden signals

Latency, Traffic, Errors and Saturation are considered 4 golden signals of observability. You must measure these for any system to understand how it is performing.

Latency

Latency is a measure of 'slowness'. It is the time taken to send response to a request. The response could be a success or a failure. Typically, a failure response is quicker, so it is important to separate the two.

Traffic

Traffic is a measure of 'how many users are accessing your service' over a period of time. For a web application it is measured by HTTP requests per second.

Errors

Errors is a metric that measures the number of failures in your service. It can be tracked with a metric like so –

logback_events_total{level="error",} 0.0

You can even link this with the latest trace id that caused the error with an Exemplar, so that it is possible to jump to the corresponding trace id in the Zipkin UI and analyse further. More about Exemplars is explained later in this section.

Saturation

Saturation indicates how 'busy' your service has been. If a service is too busy servicing current requests while more requests are piling up in the queue, it can overwhelm the system and eventually lead to failure.

Saturation can be measured by CPU usage or memory usage or by measuring any resource that is limited and can get exhausted if the service consumes way too much of it.

From a resource perspective this is a measure of 'utilization'. You need to make a decision – at what levels of resource utilization can your service safely function.

There are many metrics monitoring systems out there in the market both paid and opensource and each one works differently. Some of them like Prometheus scrape an endpoint from the client at regular intervals (pull) while others like Datadog or Dynatrace expect a client push. There are other differences like dimensionality and whether the rate aggregation is client-side or server-side.

Monitoring systems come with alerting mechanism where you can configure for alerts to be sent on some conditions like on error responses or if certain number of requests take more than the threshold time to process. Alerts are a hallmark of a good observability system. Most times it is impossible to keep looking at all metrics for all services. Alerts are an inevitable and important part of the eco-system if you want to build good production observability.
Prometheus architecture

Logs

Logs are events emitted from services which are persisted to a file for later reference. They are usually accompanied by a timestamp, a trace id and the details of the event itself. Logs are intended to help you stitch back the context for what, where, when and why. In a microservices distributed

architecture each service logs its own events. In the larger context where the request pans across multiple services, it might get difficult to open a random service log and figure out the problem, should there be one. That is why it is important to evaluate the problems at a high-level leveraging metrics and traces and then when you have zeroed in suspect services you deep dive into logs to check for details. It probably takes several iterations of going from metrics to traces to logs and back.

While logs are very easy to generate and useful to debug, excessive logging can cause overhead in production. A popular and effective strategy is to log only errors in production.

Logs written by different services running as containers go into separate files. When you run your containers in an orchestrated platform like Kubernetes, it is recommended to stream your logs to stdout or stderr. The logs are written to the default Kubernetes log location usually /var/log/pods/{pod-id}/{container-id} on the node.

It becomes important to be able to visualize all logs corresponding to your application in one place – in other words log aggregation. Logstash and Fluentd are popular log aggregation frameworks. Other solutions like Datadog and Grafana provide their own agents to enable log aggregation.

Let us consider Fluentd for example. It can be configured to run as a DaemonSet in each Kubernetes node. When different containers write to their respective logs, Fluentd can collect logs from various sources and send them to a persistent store like ElasticSearch with Kibana UI or an event streaming platform like Kafka or for that matter Influxdb with Grafana UI.

FluentD architecture.

Tracing

Traces provide an overview of the application flow and causality. When a request goes through many microservices either synchronously or asynchronously, it crosses over many contextual boundaries. It is important to know what services were invoked by a given operation, the order in which they were invoked and how much time was spent in each service handling the request. This gives a good understanding of the command flow and the latency that the application incurs for a given operation.

Every request flow is recorded in a 'Trace' and has a trace id associated with it and is usually injected into the header of the request. The trace id is the root of the span tree that spreads across multiple services. The tracer collects metadata information in what is called as a 'Span'. Each trace consists of multiple spans. Each span exists within the contextual boundary of a service. When the trace moves on to the next service in the command flow, it is then a new span that records the corresponding information in it. Every span has a span id associated with it along with the trace id. This way it is possible to trace back the origin of a request and the path it has followed to reach this point.

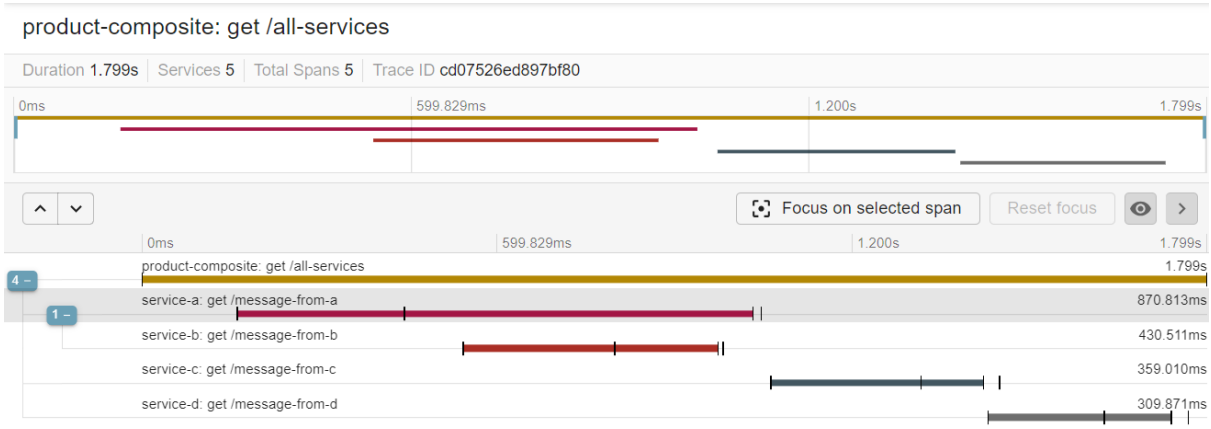
Zipkin and Jaeger are two very popular tracing solutions that are OpenTracing compatible. For the purpose of this section, we will use Zipkin to demonstrate with examples how tracing can help measure performance and request flows.

Distributed tracing with Zipkin involves a tracer that runs on each instance of your instrumented service and record useful timing and metadata information. This information is then sent by the Reporter to the Zipkin

server via a Transport. The Zipkin server consists of aa collector which collects the data from the Reporter and stores it in an internal in-memory database. It is possible to persist this data from in-memory database to an external database (recommended for production) like Elastic Search, Cassandra or MySQL. The server also provides with APIs that can be queried for the data we just stored and an UI that can be used to visualize and analyse.

Fig: Zipkin architecture.

Let us consider scenario where a product-composite which calls service - a, service-c and service-d. service-a internally calls synchronously service-b. The trace in Zipkin UI looks as below.

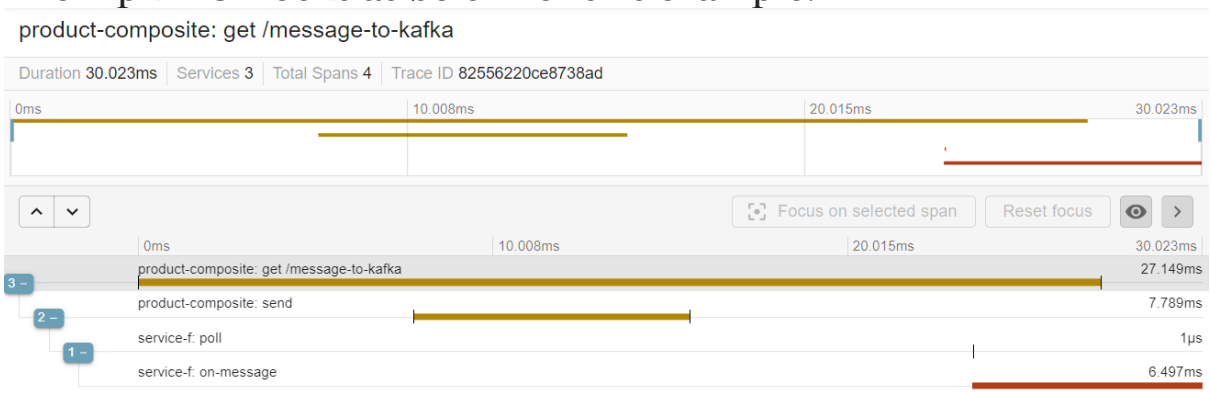


It is very easy to analyse from the Zipkin UI, what services were called from a given service and how much time each call took. All tracing is done with very little overhead and no boilerplate code is added to your application.

Let us take for example an asynchronous call from product-composite service to service-f using Kafka as a distributed messaging platform. The trace id is injected as a header in the Kafka message before sending it. It is

then extracted in the consumer and a new span is created for the tracing to continue. This is particularly useful while tracing distributed transactions like saga transactions spread over multiple services.

The Zipkin UI looks as below for this example:



The transaction starts in product-composite service. A message is created in product-composite and sent to a Kafka topic. This message is read asynchronously in service-f and handled to complete transaction. However, since the transaction spans over product-composite and service-f, the trace id is propagated across services. It is easy to visualize this with Zipkin (or any other tracing service of your choice) as shown by the example above.

Best practices

Metrics, Logs and Traces, although useful individually to some extent, need to function together when you want a full picture of the system. They cannot help you to understand the root cause of an issue you are facing in the production unless you use all of them in a meaningful way.

So, how can you leverage the three pillars of observability to get the best value from them?

Exemplars

This is where exemplars come in. Exemplars are a way to introduce external data into a metric. One of the most common exemplars is to add the trace id of a request flow into the timer histogram bucket of a function which is in the request path. This way you can hop to the right trace in your tracing system from the metric of interest. Explore the trace and find out where in the request flow the problem lies. Once you have identified the service where the root cause exists, you can explore the logs corresponding to the trace id to further deep dive into the problem.

Tools like Grafana provide good exemplar integration across metrics, traces and logs.

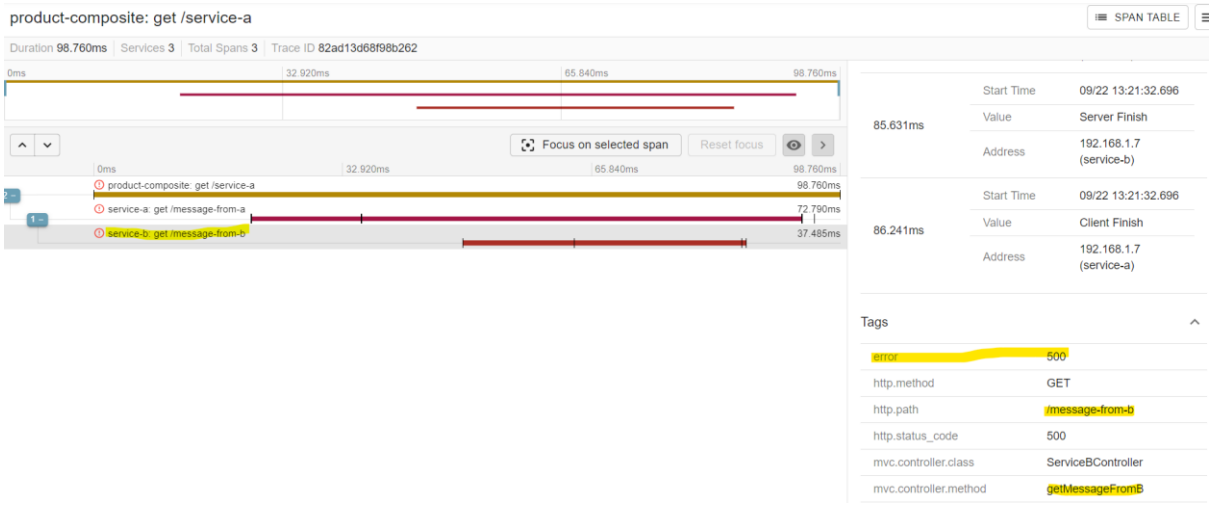
To understand how it works better, let us consider for example the following scenarios.

Scenario 1: You have a request flow through product composite, then hit service-a and then service-b. Service-b fails and return error response via the product-composite.

You notice that there is a spike in the error metrics. You see that there is a trace id mapped to the error, thanks to the exemplar associated with it. This is the trace id of the latest request flow that failed.

```
http_server_requests_seconds_count{exception="RuntimeException",method="GET",out
come="SERVER_ERROR",status="500",uri="/service-a",} 1.0 #
{span_id=" d94711a5a025d1c1",trace_id=" 82ad13d68f98b262"}
```

Hop to the trace id in Zipkin UI. You notice that service-a calls service-b properly, but service-b causes a failure and return a 500 error.



You can further open the logs for service-b and grep with trace-id and or span-id. Now you see all the corresponding logs pertaining to the error.

Now you see how you can deep dive from a high-level error count metric information all the way up to the log event for the exception that led to the failure. It is very important to be able to do this since unless you use metrics, traces and logs in combination with one another you will not be able to take full advantage of the available information at hand.

Scenario 2: You have a request flow through product composite, then hit service-a and service-b. Service-b takes 1-2s to handle request where the expected threshold is 200ms and tolerating threshold is 1s.

This scenario can be observed with multiple metrics. For example, for a request that took 1.45s to complete -

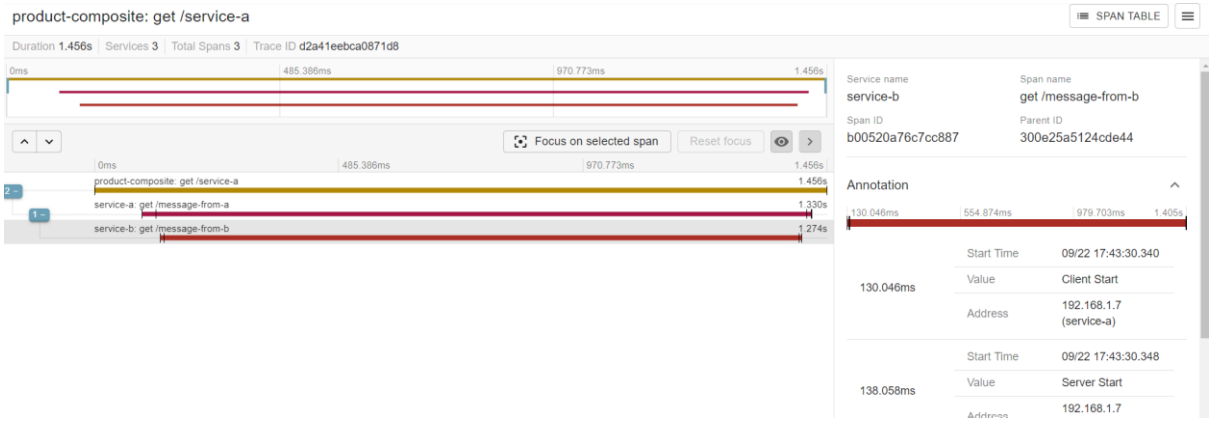
The corresponding timer bucket will start filling up.

```
product_composite_http_requests_seconds_bucket{uri="/service-a",le="1.4537915",}
1.0 #
{span_id=" b00520a76c7cc887",trace_id=" d2a41eebca0871d8"}
```

The gauge that measures ‘max time taken by a request’ starts showing up higher value. In this case 1.45s

```
product_composite_http_requests_seconds_max{uri="/service-a",} 1.4537915
```

Now, you want to find out where the delay is. So, you hop to the trace id in Zipkin UI



You can easily see that the service-b call has taken over 1.4s. There lies the bottle-neck. You can do further debugging from the logs and find out where exactly the problem is.

```
2023-09-22 17:43:30.355  INFO [service-b,d2a41eebca0871d8,b00520a76c7cc887,true] 18144 --- [nio-9082-exec-5] o.i.o.serviceb.Servi
BController        : Request recieved at microservice B
2023-09-22 17:43:31.606  INFO [service-b,d2a41eebca0871d8,b00520a76c7cc887,true] 18144 --- [nio-9082-exec-5] o.i.o.serviceb.Servi
BController        : Response from microservice B is: This is a sample response from microservice B
```

Notice that the between the two logs the time elapsed is about 1.4s. Sometimes it will not be this obvious. There are other ways like using a timer to determine the problem area and adding more targeted instrumentation in the testing environment to check for issues.

Scenario 3: You have a service-f which consumes a message from a Kafka topic and then produces another message to another topic. There is a

defect in the processing logic. So, some messages are not being handled and an exception is getting thrown. For those messages, no outgoing messages are being produced by your service.

When you look at service-f metrics, it is pretty easy to see that some messages have failed to be processed. You already have set up alerts for it, may be. Finding out what messages have not been processed is the first order of the day.

Looking at log events is one way to find out the trace id of latest message that failed, assuming there is an exemplar associated with it. It will also show you the number of messages that have failed.

Also, look for the tags in the timer histogram buckets that show failure like outcome="SERVER_ERROR". Find the trace id associated with it.

Look at the last time a message was handled successfully. Together with the rate of messages handled, this data will tell you at what rate your service is failing approximately (like one in 10 message or 6 in 10 messages).

Hop to the trace id in Zipkin UI. Find out where it is failing. Use logs to map the right trace id with the log and dig deeper.

Some of the other good practices include -

- Logs and Traces will grow with the increase in the request traffic. So, it is important to ensure that you do not overwhelm your production systems with meta data.
- In production log only what is important that will help you to meaningfully understand what is going on.

- Traces must be sampled at a proper frequency that will help you to get enough requests sampled to understand the request flow and latency information in production environment.
- Metrics do not grow based on traffic flow. But it is important to define what you want to measure and observe. Too many metrics can get overwhelming and difficult to make sense out of.

Overall architecture

Section 11: Conclusion

With this we come to the end of our book. In this book I have tried to address some of the design challenges you will face while you architect your microservices applications and the trade-offs involved. I have detailed the best practices and important considerations through various scenarios covering cloud strategy, microservices APIs design, transaction management, distributed data strategies and finally security of your microservices.

With microservices you will deal with cloud infrastructure, Kubernetes clusters, compute, storage and networking considerations and trade-offs as you plan your application deployment. I have covered those topics from a development team's view point in the later half of this book.

Observability is a critical piece of the puzzle which is addressed at the end of this book to wrap it up.

I hope you find the book useful and I wish you all the very best!