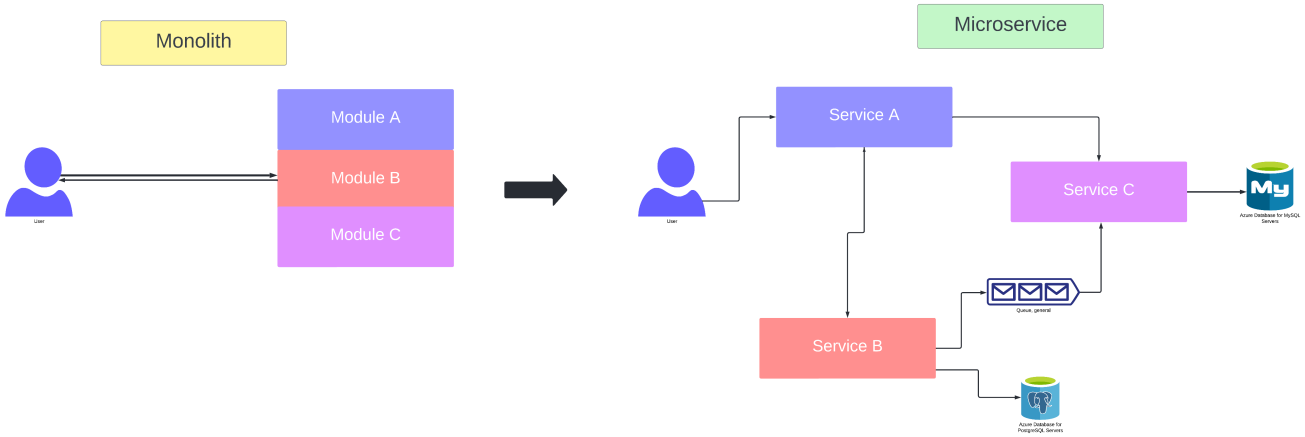
INTERVIEW READY

Moving from Monoliths to Microservices



Before we start discussing about how we are going to migrate let's first discuss briefly about monolith and microservice architecture.

Monoliths are large code bases which contain all the logic required for you to run your application. All the relevant code is kept in a single repository.

Whereas in a **Microservice architecture** our application is made of small services communicating with each other using protocols like HTTP or GRPC.

Before we start our discussion on how to migrate let's discuss some advantages monolith have over microservice

1. **Monoliths are better for small team** Microservice architecture is more complex. It is might a waste of time for small teams to deal with the complexity of the microservices architecture. For small teams unified application is more manageable in comparison to splitting code up into microservices.

2. **Less chances of breaking change** Let's take a scenario. We have two modules **Profile** and **Auth**. Initially the profile ID was integer but we changed it to string. So the Profile module has function **getProfile(string ProfileID)** but the auth service has a function call **getProfile(id)** where id is int.

   In **Monolith** architecture Auth module communicates Profile module via function call. The parameters that we are passing is clear. So if the parameters are invalid then we can catch them at **compile time**.

   However in **microservice** architecture Auth module has no idea that the function parameters have changed. So all communication with profile service will break.
   Since the code and expectation in monoliths are at one place it is much harder to have breaking changes.

But why should we migrate from Monolith to Microservice architecture ?

1. **Separation of concerns** In a microservice architecture we can make changes to one service without affecting other services. However in monoliths since all our code is at one place making changes will affect the entire system.

2. **Engineering is easier** Working with microservice architecture can be easier since the developer needs to understand only a particular microservice providing the functionality he/she will be working on and not the whole system.
   For e.g., an engineer working with the auth service is not concerned with the functionalities of profile service.

3. **Easy deployments** Each service in a microservice architecture is self contained. So if we want to redeploy a particular service we can do so without affecting the entire system. Also as services are smaller in size compared to monoliths their startup and deployment time are relatively less.

4. **Better scalability**
   Each service can be scaled independently. So it is more cost effective and saves time compared to monoliths.

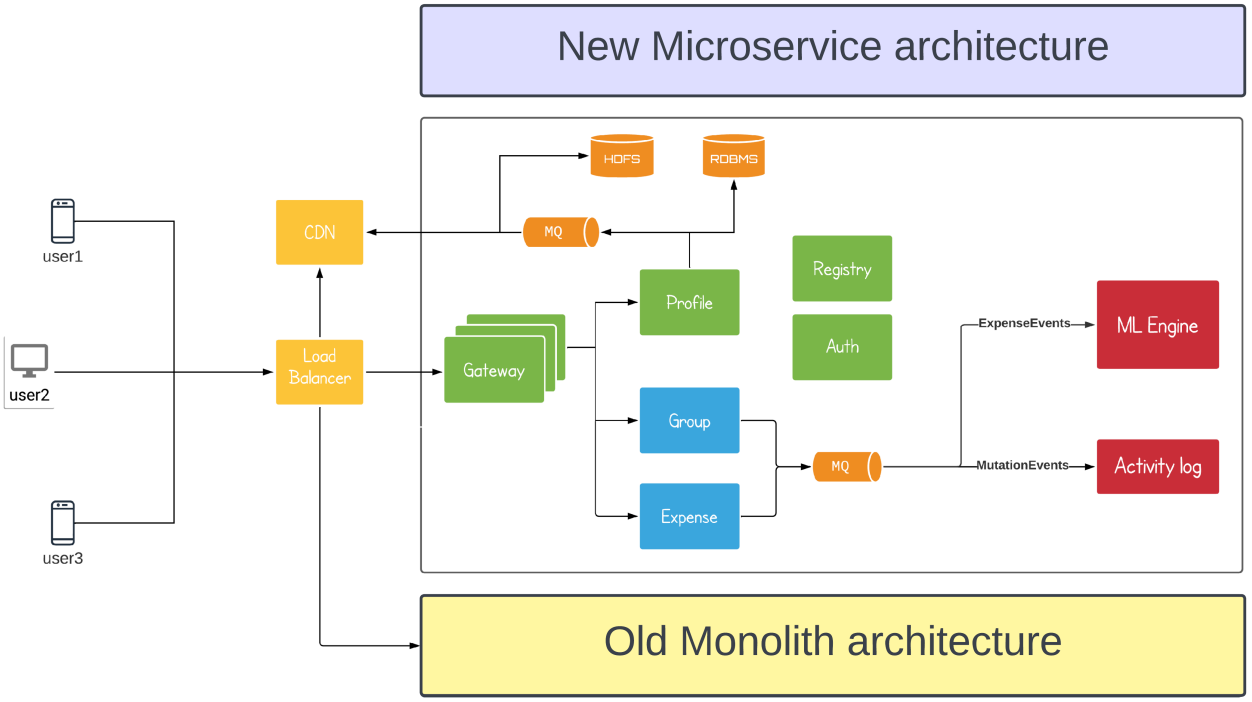
5. **Great for large teams** Microservice is a great choice when we have a large team. Each team can work on a service which makes the development process much efficient.

6. **More flexibility** Each service can be written in different language and tech stack. This allows teams to pick up a tech stack that is the most appropriate.

Now let's discuss the fun part

How are we going to make a smooth migration.

Our first approach can be that we write the entire microservice architecture. But we will have our old monolith as well. Then we will route some of our users (let's say 20%) to our new architecture. If it works fine then we can route more and more users until we route all users to the new architecture.

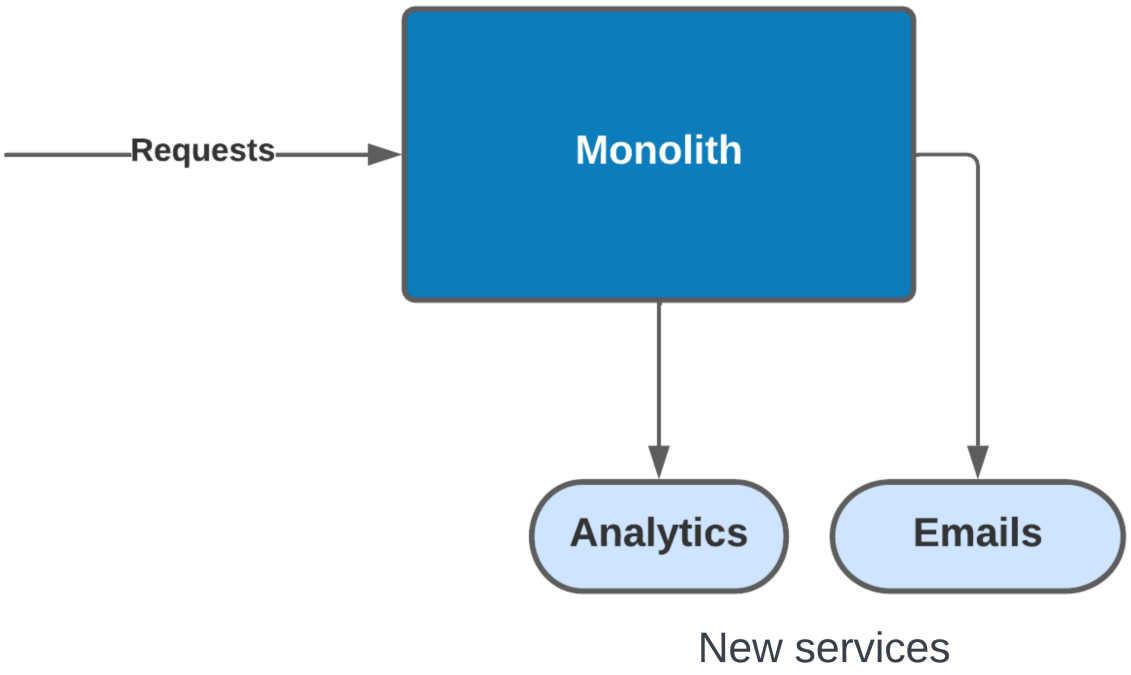


So is this approach optimal? No.

The problem with this approach is the huge engineering challenge. We need to **write all microservices**, **make sure databases are correctly configured** and then **redirect user**. So we need to invest a lot of resource in the very beginning.

So is there any better approach? Yes

Whenever we want to add a new module to our application, instead of adding it to the monolith we can make it a single service. This approach is better because we need less investment in the beginning and the migration will be much smoother.



Okay, now we have added a new service but,

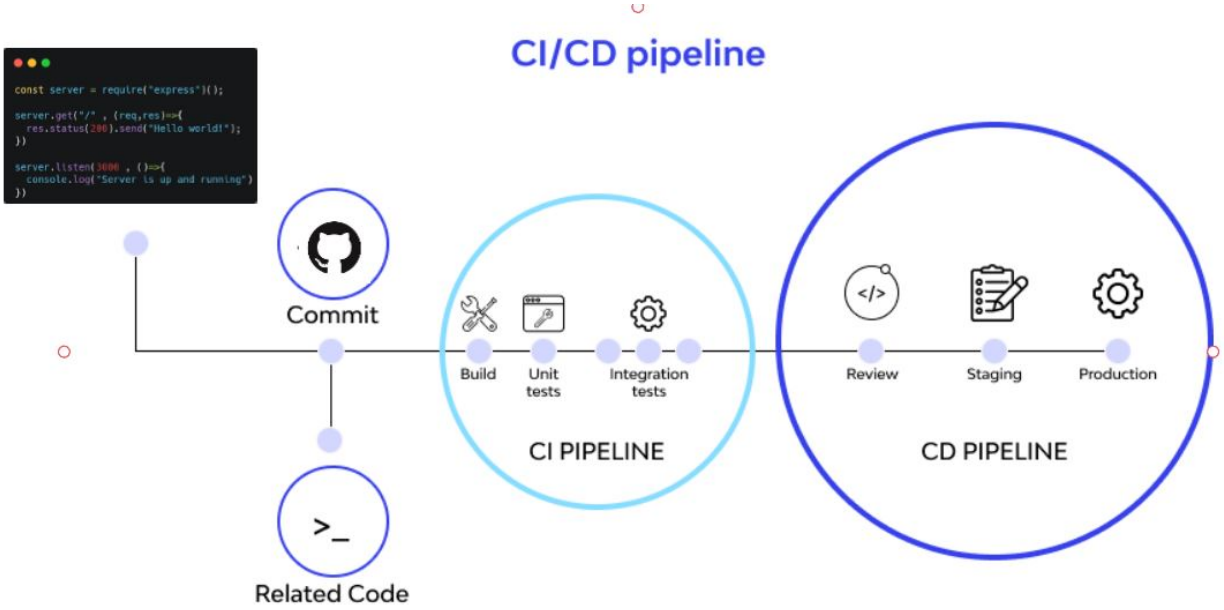
What are the infrastructure requirements to make the migration ?

- **API Contract** First, we need to define how the services are going to talk to each other. We can do this by defining API contracts for each service. But **what is an API Contract?**. In simple terms it tells a service, how to construct a valid object so that it can send it to other service.

  We can also go for a better approach. Instead of making the service go through the contract before every call we can **implement a library for each service**. Developers can use the function defined in the library to make network calls. This library is called the client library.

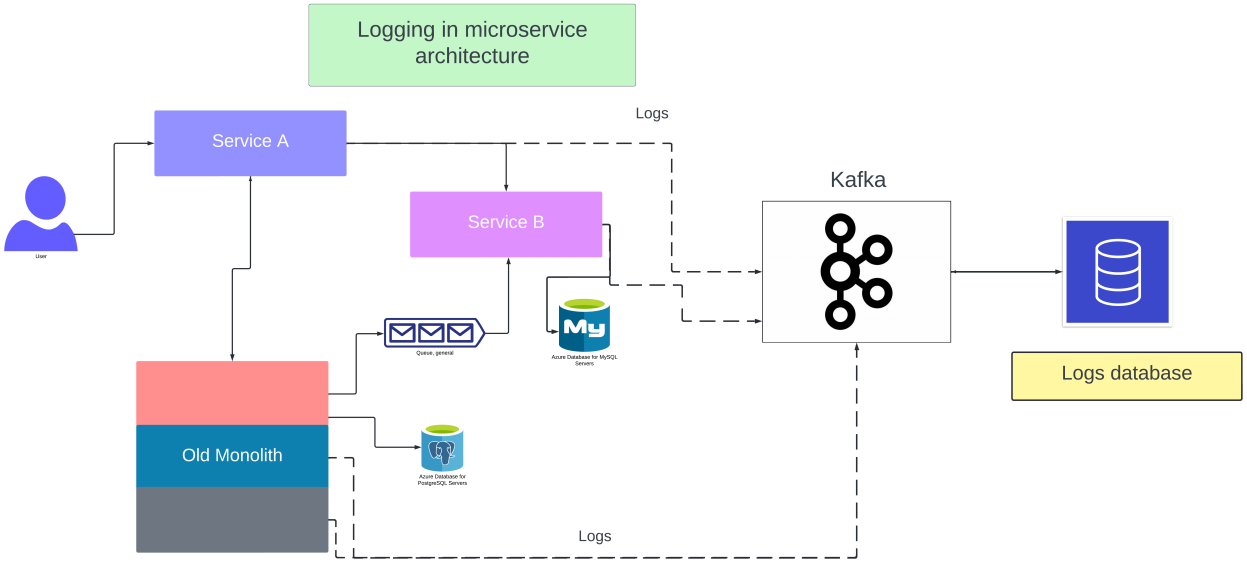
- **Router** We also need to route requests from one service to another. We can use something like a service registry here. It maps the the methods to handlers. So whenever we get a request having a particular method we know to which service we need to send the request.

- **Simplifying deployment** As we increase the number of services in our system the deployment process should not become complex. To simplify the deployment process we can have automated deployments (*e.g., If we push code to main then it is tested and deployed automatically*). We can use CI and CD pipelines, containers and service dashboards to automate our integration, development and deployment.



- **Communication between services** Let's try to understand this point with few examples. For some services we might need quick response, for that we can use request-response architecture. For other services whether the message reaches the other end or not is not that critical. In such cases we might use a message queue. For sending bulk data to any service we can use batch processing.
  As you can see, we choose communication process for each pair of services as per our need.

- **Logging**
  A request can be routed from service A to service B and so on. If there is a separate logger in each service then in case of any error or failure it will be a nightmare to find the bug. So we need a logging service that logs events in every service in our system to a separate database.
  We can use something like Apache Kafka and Lucene.



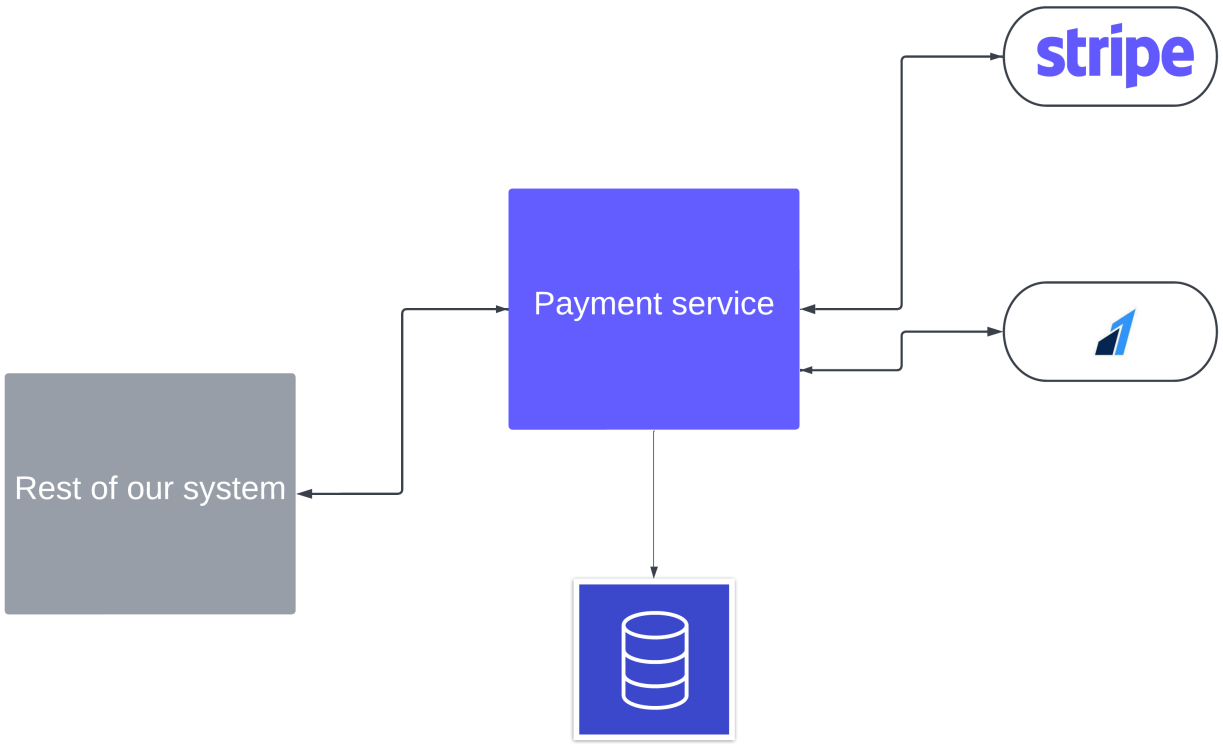
Now let's discuss about some practical considerations

- **Each service must be the single source of truth for the data it is responsible for.** Other services can ask for the data but will not store it. For this we need a **dedicated data store for each service.**

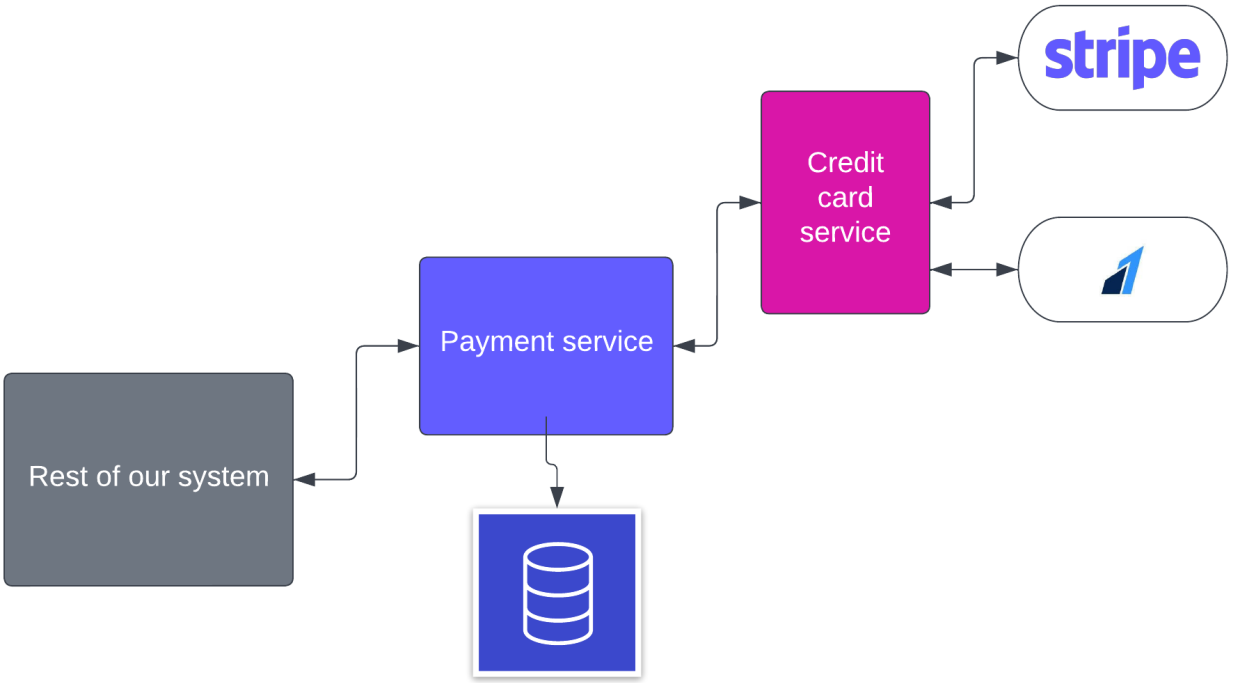
- **Splitting the service into smaller ones.** Once we have converted our monolith to microservice then we might be tempted to keep breaking down each service into smaller and smaller service. But before breaking down the service we must ask ourselves these three question
  - Is the new service being used by any other service excluding the original service?
  - Is there sufficient decoupling between the original and new service?
  - Is the business requirements of the new service separate from the original service?

  If the answer to all these questions are **YES**, then we can and should break our services.
  In simple terms we must condense business responsibility to a single service.

  Let's try to understand this point with an example:
  Initially we have this system:



But we want to separate a smaller service, one that handles credit card transactions with third party services like Razorpay, stripe etc. So it looks something like this:

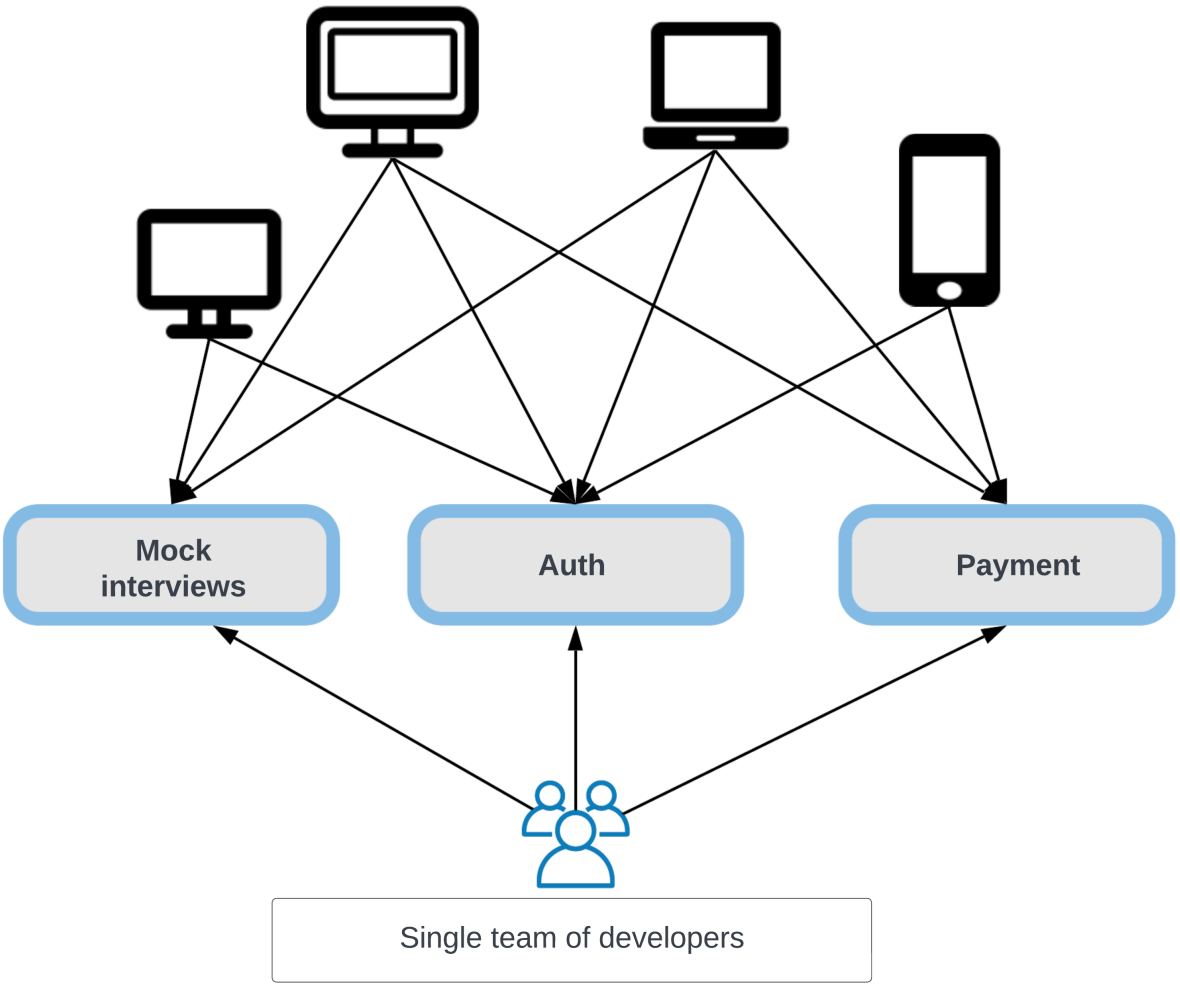


Now let's ask ourselves the 3 questions:

- Is the credit card service used by any other service except payment? **No**
- When there is a change in payment service requirement will there be change in credit card requirement? **Yes**. So is there sufficient decoupling? **No**
- Are the business requirement of payment and credit card service different? **No**

So **we shouldn't split the service into smaller ones.**

- **Initial Infrastructure cost** Microservices are expensive. Cost of logging, building a deployment infrastructure etc is not worth it for a small team. So now the question arises when should we move to microservice architecture?



We can get a rough idea form the table below: | Startup | Medium scale companies |Large organization| |---|---|---| |*1 developer can work on at most 2 microservices**|*1 developer can work on at most 1 microservices**|*2-4 developers can work on at most 1 microservices* |

That's it for now!

You can check out more designs on our video course at InterviewReady.