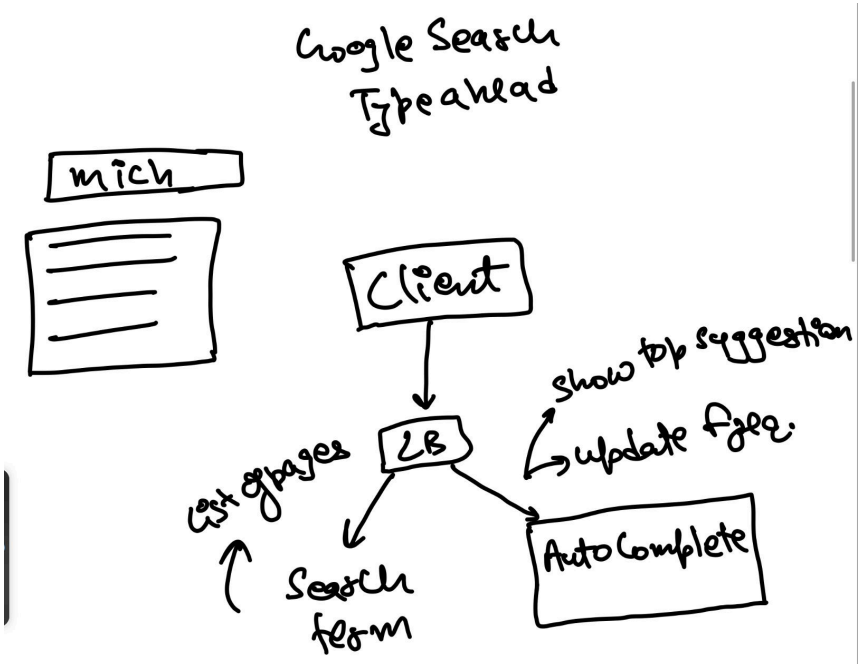
Agenda for microservices classes

- What are microservices ?
    - Theory for microservices
    - What are services ?
    - What are microservices ?
    - Why are they even needed ?
    - Principles of microservices
- Microservices design patterns
    - Distributed Transactions
        - [Anti-pattern] Two phase locking
        - Saga pattern
        - CQRS
    - Observability
    - Circuit breaker pattern
        - Situation of service degradation
    - Event Sourcing
- We would not be implementing anything, we will be just understanding microservices, it's issues and design pattern behind it.

Services

- What are services ?

If you remember, we discussed Search typeahead in one of the classes. If you think about Google Search, there are 2 things happening from the client side.
When I type, I request for most likely typeahead suggestions. And then step 2 is actually searching for something where I expect to see search results.



- You need typeahead to show a suggestion, but when you click on search then the next step is that it takes you to the page where you have results of websites that may contain or they are about the search term.
- This means google is not just the autocomplete system, but there is some other place as well where you send the search term and it returns you the pages which match the search term.
- For a given website with multiple functionalities, it's possible that there are not one but multiple services, where a service is a combination of:
    - App servers
    - Database
    - Caching
    - Other more components
- Service is a combination of all of those, as a service it exposes some APIs to the external world.
    - Just like we had two APIs in search typeahead

    - Api could be for:
        - Fetching something
        - Making an update

Think of the search typeahead service. In the search typeahead service, we supported 2 APIs.

- getTopXSuggestions(search_prefix)
- updateFrequency(search_term)
  These APIs could be called by anybody. It could be called by the client or another service.

Definition of a service

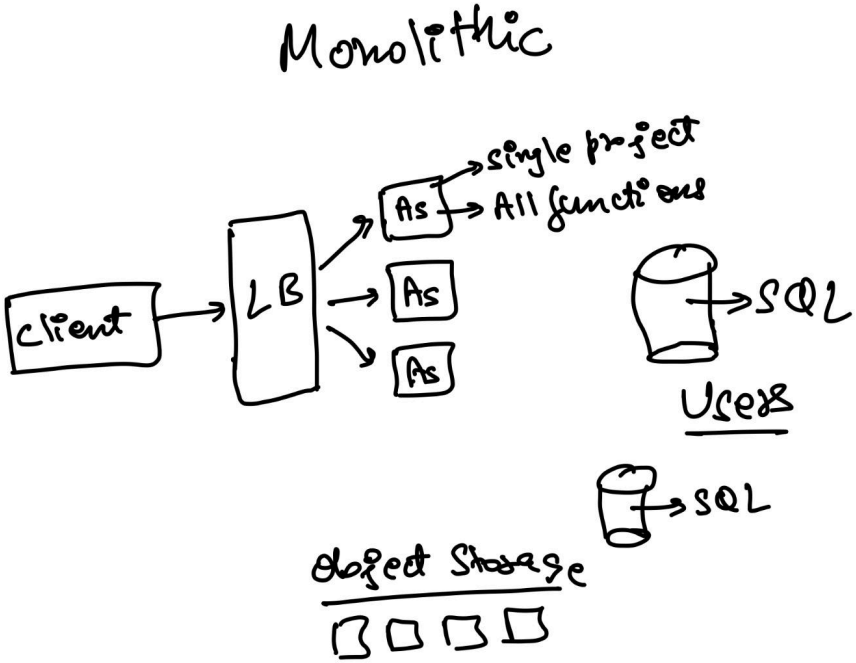
- A service is a combination of application layer and storage layer which exposes certrain API to do certain job.
- In this google example we had two different services
    - For doing autocomplete
    - For giving actual search results
        - Whenever a search happens, this makes a call to the autocomplete service to update the frequency of new search term.

More elaboration

- Service contains APIs
    - APIs are how you talk to a service and ask for information, or ask it to update/change/add some information. Remember REST APIs from previous classes?
- Analogy
    - Imagine a restaurant, it contains:
        - Kitchen
        - Cooks
        - Meals
        - Waiters
        - Ingredients
        - and many other components
    - But to talk to the restaurant or order the food, we would be talking to waiters only.
    - So here waiters are APIs and the entire restaurant setup is service.

Monolithic Architecture

Imagine if my entire site had only one service. A big service and hence the name monolith.



- Consider any website, if the entire website is built in a way where you have a client; the client talks to the load balancer, and we have app servers which are running a single project code.
    - There is no concept of services.
    - Everything is inside these appservers, all kinds of business logic.
    - There is one single codebase, with all codebase compiled in one single executable file.
      Note that a single codebase does not mean a single file. Things can still be broken down across files, across functions. You can still make the code modular. Apply the design patterns we have learnt in LLD. Just that every appserver is able to handle all kinds of requests coming to the website.
- After that there are databases, they could be multiple or single - for different use cases. DBs could be sharded or not sharded.

- As long as everything is part of the same codebase, it is called monolithic architecture.
- Monolithic is good if:
    - Project is small
    - Not very complicated

Need for microservices / problems with monolithic

- As the codebase becomes bigger with time, it is harder for new employees/engineers to understand the codebase.
- It will be difficult for them to learn and start contributing and adding features or debugging because it takes time to understand such a large codebase. A single change could have multiple dependencies.
- In order to make a small change in one function and make it reflect we have to recompile the whole codebase, then make the new executable file and finally redeploy it.
    - For all minor changes we have to do large project compilation + redeployment
    - Project compilation time or build time become slower.
- As all appservers are running all files in the project, if one use case is very very slow, it ends up affecting everyone. Because your app servers are busy executing the slow code.
    - Then most of the requests will be stuck, so one function being slow basically slows down everything else.
    - Failure/slowness in one component affect everyone.

However we should continue using monolithic architecture unless we start facing problems/challenges with it.

- Having microservices also creates a lot of challenges. It is expensive and slow. We will discuss this later. So, do not shift from monolith unless absolutely required.

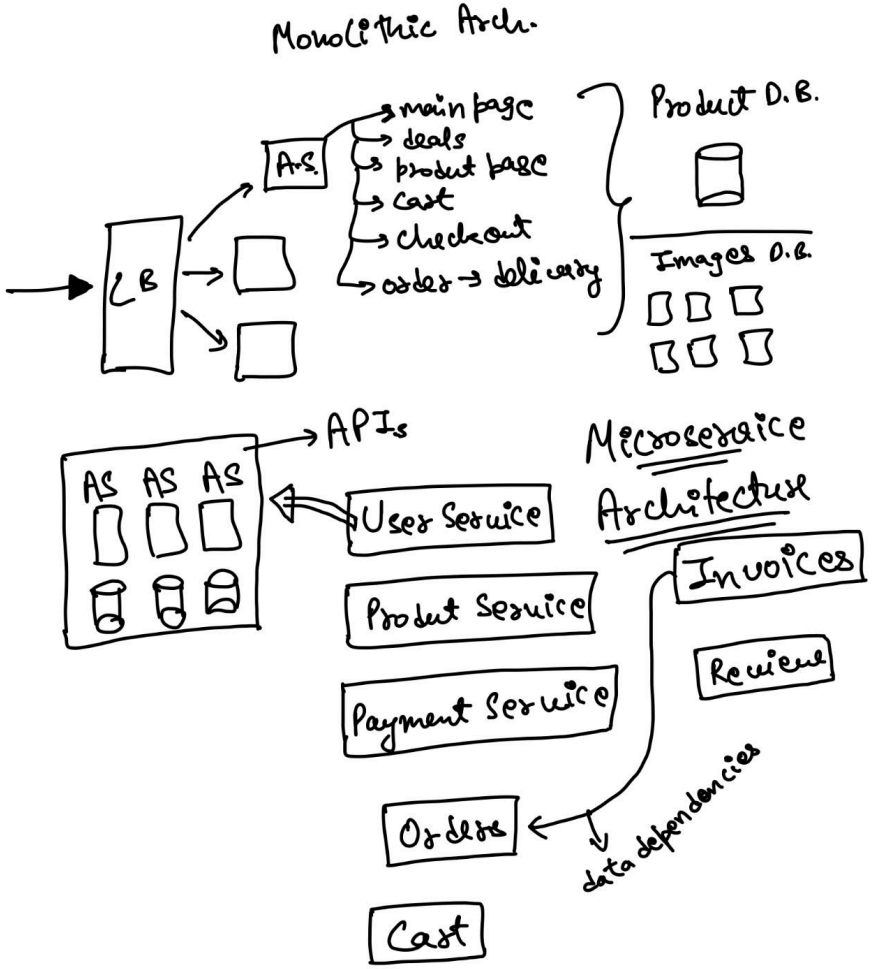
Service Oriented Architecture Vs Microservices

- Breaking the project down in multiple parts, if things are becoming too big.
- This can be done in two fashion:
    - Service oriented architecture
    - Microservices

- Service oriented Architecture
    - We build multiple services
    - It is not mandated that every service has it’s own database.
    - Different services might share one or more database
    - Calls are synchronous in nature
        - It means a service will wait for the response, when it makes a call to another service.
- Microservices
    - **Separation of concern**
        - Okay with duplicating data, but require an independent /separate database.
        - Tech stack / code deploy process / scaling can all the independent. More on this later.
    - Services can be called in both synchronous and asynchronous fashion.
        - Event driven architecture can happen.
        - Across services, in such cases, eventual consistency is what’s feasible instead of immediate consistency.

Flipkart Example:

Let’s take an example of Flipkart to elaborate the difference between monolith and microservices.



- Flipkart monolithic architecture
    - We have appservers running massive project
    - The project internally has all the code that is required to show:
        - Flipkart main page
        - Deals running

        - Product page
        - Cart
        - Checkout page
        - Order Delivery page
    - All the logic is inside single project
    - We may have multiple databases like product db, image db etc.
    - Any appserver, for example, will be running this large executable that has the business logic of all of the above.
- Flipkart microservices architecture

Now imagine Flipkart at the scale it is. There are thousands of developers working on the team. Different experiments are tried everyday. Are there some components which are actually independent and can be different services?
For example, information about a user's profile (their name, address, profile pic, etc.) are not needed most of the time. Most of the time you search for products, or you look at the details of the product.
So, can we maybe separate out User details as another service?
Very similarly, the entire payment flow is again independent. Maybe, Payment can also be separated as a service. Same for notifications (email/SMS). Same for Invoice generation.

Maybe, Flipkart can be broken down into following microservices:

- UserService for showing/updating user profile
- ProductService for showing product page and search functionality - Maintains inventory, full text search on product, all product details.
- PaymentService for completing payments
- OrderService for tracking and updating order after the order is placed.
- Cart service
- Invoice service for generating invoices
- Review service for product reviews

Note that you should not break into services that are very dependent on each other. For example,

- If product search and product description are closely linked to each other - and they both essentially need same information, then best to keep them in the same service.
- Very similarly, not a great idea to break down user and user_login / user_authentication into 2 different services, if user authentication is heavily dependent on user microservice for data.

That being said, dependencies cannot be made zero. Which is why microservices need to talk to each other. A single transaction might be across multiple microservices.

- Like Invoice service depends on order service as invoices are generated after order is placed. So, order service would make a call (directly/indirectly) to ensure invoices are generated.

Client to Microservice interaction

Now, you might have 2 questions. What's happening from the client. Will my page become really slow if I have to talk to multiple microservices?
And second, how would a client talk to a microservice?
Let's go one by one.
Imagine the Facebook.com page.

- Left side has menu items.
- Centre of the page is news feed.
- Right of the page are ads, events and "people you may know" section.
- Top of the page has your name, picture.

All of the above could be different microservices.

- News Feed
- Ads
- Events
- People you may know
- User

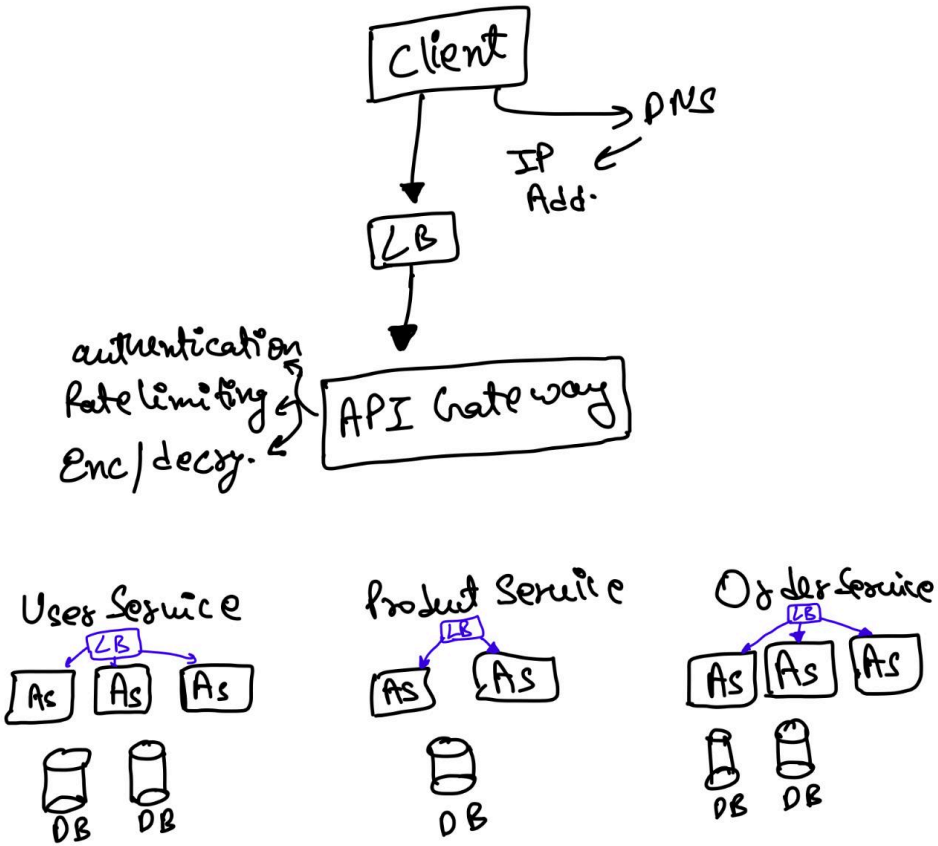
In a way, when the page is loaded, in step 1, basic skeleton of the page is fetched. And then the page itself fires async requests in parallel to all above microservices to get data.
Imagine, requests going to (get req for ease of understanding):

- /newsfeed/<user_id>
- /events/<user_id>
- /ads/<user_id>
- /pymk/<user_id>
- /user/<user_id>

Now, onto the second part. Which is, in DNS, I store the IP for a load balancer. How do we directly connect to a microservice then?

API Gateway

- How would load balancer know to send which request to which app server ?



- We have a client, it would do DNS lookup, find a load balancer, so we basically reach load balancer.
    - Load balancer is working on network layer
    - It has no information about the microservices.
    - Hence, we need a set of machines which are aware of what microservices exist, and how to route to that microservice.
    - Those are called API Gateways. They sit below the Load Balancer and work at the application layer in networking.
- API gateway can be used for more than routing

- Firstly, API gateway is not one single machine. You need multiple machines to ensure high availability. But all API Gateways have all the logic (so LB can send the request to any of the API Gateways).
- Given API Gateways work on application layer, you can also use them to do:
  - Authentication: Validate if session tokens are stored on the server side, and hence it's a valid session. Reject the request if not.
  - Rate limiting: Let user configure IP based, service based, API based rules, which if violated rate limit is hit and request is rejected. Refer to the class called "Popular Interview Questions" for more details.
  - Encryption/decryption

Properties of Microservices

- **Separation of Concern**
  - Tech stack and DB could be different for different microservices. One microservice could be built in Java Spring Boot with MySQL, whereas the other could be built with NodeJS and MongoDB. You have the liberty to choose what works best in your case.
    - For example, a ProductService requires full text search, needs to be fast, and is schema less given so many categories. So, you might choose to build it in Java Spring Boot with MongoDB for storing product details, and ElasticSearch for the fulltext index.
    - At the same time, for UserService, you can choose to use Python with MySQL. No sharding needed.
  - Completely independent code deployment process. If a microservice does not change often, it need not be deployed often. If a microservice does change often, then only that needs to be deployed more frequently.
  - Testing is also easier. Because every microservice is independent, and the expected response is clear for use cases, testing becomes easier. I know that changes here will not affect any other services as long as my APIs respond with what's expected.
  - Scaling is independent. If UserService has less traffic, maybe it only has a few appservers and a small DB. However, if ProductService has a lot of traffic, it might have a lot of appservers, sharding in MongoDB, and a large ElasticSearch cluster.

    - Added robustness. If there is a critical bug or an issue in a particular microservice, it only affects that microservice. Other microservice can isolate themselves (We will discuss circuit breaker pattern for this).
- Cons of microservices
    - Adds more layers or latency because of more network calls.
        - In monolithic, request goes to load balancer then to app server then to database.
        - With microservice, request goest to load balancer, then to API gateway, then to load balancer of one of the microservices, then to app server and then to database.
        - But in between if we require some more information then again we have to call some other microservices.
        - Due to this, latency increases.
    - More expensive
        - Regardless of the traffic, for each microservice we need at least one app server and one database. That means if I have 20 microservices, I need at least 20 appservers, 20 DB machines, a few machines for API gateway, etc.
        - So maintaining a microservice is more expensive.
    - Logging/tracing is harder
        - As requests go through many microservice so if a failure occurs we have to look for logs of each microservice which makes tracing harder.
    - Data inconsistency
        - Imagine two microservices: orderService and paymentService. Both have the same attribute that is payment_done or payment_pending.
        - If paymentService the attribute is updated then it should also be updated at orderService.
        - But there is no guarantee that they will always remain in sync as they are stored in different places.
        - You can only ensure eventual consistency. Immediate consistency across services is very hard.

Ok, now that we understand the basic tenets of Microservices, following are some key questions we should explore answers to. So, the remaining part of this lecture and the next lecture will do exactly that.

- How do you do observability in microservices? How to debug / find source of a failure.

- If a transaction has to happen across multiple microservice, how do we ensure it happens reliably, and as a transaction. How do we do distributed transactions in microservices?
    - Are there ways we can reduce latency? What can we do to address consistency issues?
- If a microservice fails, then how do we ensure that other microservices do not go down because of it? How do we still keep other functionalities running?

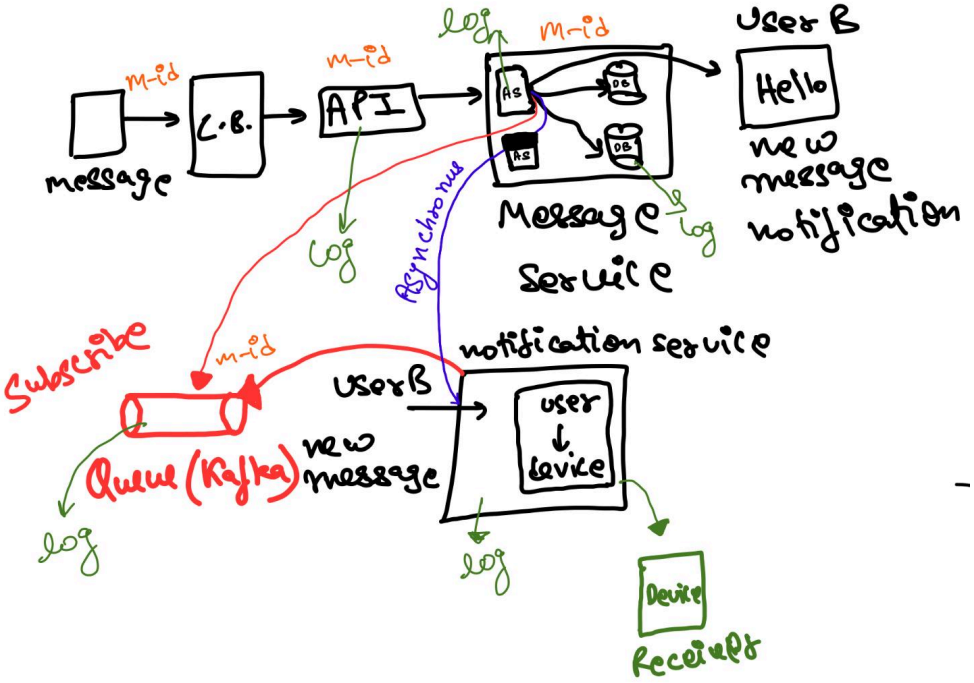
Observability

- What is observability in general ?
    - Imagine you have built a project and deployed it. You might expect that everything should work fine.
    - But in reality the code does not always work fine. There are things that may go wrong.
        - There might be bugs in the code.
        - There might be corner cases that are missed.
        - There could be some problem with the machine your code is running on.
        - There could be some problem with the network.
    - Code failing is fine. However, we need to know when things go wrong.
        - Maybe some form of alarm or notification
    - If we do get notified about an ongoing issue, we would also need to find out what exactly is wrong. This is called diagnosis.
        - It helps in verify the fix
- Observability can be defined in three steps:
    - How do I quickly detect that there is a problem
    - How do I diagnose and fix the problem
    - How do I verify if the problem is fixed

Now, how do we do observability in Microservices. The key challenge being that if a request goes through multiple microservices and some of these calls can be async, how do we detect and diagnose.

- There are three things which you have to take care of.

- Tracing



- Let’s take the example of messenger case study.
- Let's say somebody sends the message, it goes to load balancer, API gateway and then message service where I have app servers and databases.
- After the message service gets the message, the message is stored for both sender and recipient. However, I also want to send the notification, so maybe asynchronously the appserver also asks to send the notification to User B async.
- Let's say we have another service as notification service, which sends the user a notification that it has a new message.
- It only needs to maintain user to device mapping. When it is asked to send the notification, it sends the same.

        - Now there are few challenges, firstly, if I am making an async call to notification service, then there is some probability that the call fails.
        - Also we cannot keep retrying indefinitely. So we might end up not sending the notification at all.
        - To solve that in some cases instead of directly calling microservices, maybe we put the things on queue, like Kafka, so we publish that there is a message sent and the notification service subscribes to the same.
        - This is one format, basically the start of an event driven architecture.
        - In this case we have multiple hops.
        - Let's say we sent a message and the receiver never got the notification. That means there is a problem.
        - We have to trace exactly where the problem occurs, like on which component the message went, and on which it failed.

Distributed Tracing

- For tracing we need:
    - Unique Id/ Trace Id, which gets passed along for the same request to all microservices. In this case, we can generate a messageId and put it in the message object. If the message object is passed everywhere, then we can log messageID also everywhere .
    - If we have logs everywhere, and we are able to bring all logs at a central place , then we will be able to search by this common Id.
    - Where the logs are absent, that is the point of failure.
    - To implement this we can use splunk, elastic search. Basically, any database that can do fulltext search.
    - Example - elastic search logstash kibana, which keeps stashing logs in elastic search and kibana is a UI on top of elastic search which helps us to see and search those logs.

Alarms/Detection

However, Distributed tracing is step 2. First, I need to be notified that there is something wrong. How would that happen?

For that I need aggregated numbers somewhere. Example:

- How many messages reach every service.
- What is the failure rate on every service, every machine.
- What's the CPU like there?
- What's the latency - maybe P90, P95 latency.
- What percentage of the user see higher than 1 second latency.

If you have used graphs in Google Analytics, then imagine that.

A bunch of metrics which will tell me if something is wrong. Depending on your application, you will need to think carefully about metrics that are important and how you measure them.
However, how do I store these metrics? It cannot be logs. Logs are unreliable and best effort. Plus I only need numbers per minute or per 5 minutes. Logs might be overkill.
So, I need someplace to publish stats.
This is typically done through a time series DB.
Timeseries DB usually has a stat collection job running on every machine. It basically aggregates key to value on every machine, every minute and then sends it to the central DB. Your application code basically, just publishes a key and value.

For example,

```
// code to store message
  ods.publish("messagestore.latency", 50);
  return;
```

Your central DB (Timeseries DB) is good at aggregation for keys. Bringing stats together from all machines.
Answering queries like:

- Give me values for a given key, minute wise.
- Or Give me values for a given key, 5 min wise.
- Or Give me values for a given key, day wise.

In each of the above queries, there can be an aggregation rules (basically, how do I merge values within the 5 min or a day). Aggregation can happen as sum, max, min, avg, p95 and so on.
Most clouds give you an inbuilt time series DB as a service. Example, AWS CloudWatch.

Once you have the metrics and the graphs, then you can write rules to check a violating value and then send emails, or make IVRS calls. There are paid services like PagerDuty that can automate that for you - or you can write your own code.