MicrovsMono/Scalability&FaultTolerance

Monoliths – Deployment as a single unit, if something fails then the whole deployment fails, good monoliths can also be designed by decoupling the data models and business flows. Takes a lot of time since everybody is working on the same code, and the code may become super large and might become a pain.

Microservices- way of architecting software such that they are loosely coupled, and can be scaled and deployed independently, everything has one responsibility. Can also identify faults easily, some services can be read heavy or write heavy.

We can use microservices, we need proper tooling, we need to make sure we have proper monitoring, circuit breakers, lots of testing , ordering of systems also will be there becuz of dependencies. We need to check if we need to scale a service. We need to define how we will communicate between services.

Scalability- performing large number. of concurrent users without degrading their experience, real systems are also cost efficient, so the system design is also cost efficient.

Fault tolerance- no matter what happens , even if the servers go down the data doesn't become inconsistent.

No failure should break the consistency of data

When should we think of scalability- of we are working at a large scale comp then we have to think from day 1, but in a new comp we don't have to optimize for performance, we just have to make it work.

Look at every single line of code and assume it may crash and see how your system reacts.

Think about how your system reacts to failures.

If our system is down our users should not now and we should provide appropriate responses to users.

Implement circuit breakers, give appropriate error messages. Have a fallback plan showing default messages to the user.

How to check if our system is scalable-

Know the limits of your machine, know how many requests one machine can handle.

Load test on a machine and know benchmarks.

U will be able to find out bottlenecks in our code this way

How to load test without affecting production traffic- Do it during when you have less traffic, testing stateless components are easy, like mainly the only thing that is affected is our database , if we feel our database is going to crash dude to load testing then we have to stop. At least some load testing will happen in production especially if there is a major event.

How to replicate traffic- there are open source tools, which can replace our logs , cloud providers provide built in load testing. You can write a script which just iterates through the log file and then fires requests.

How To Make Sure we are testing the right thing - in most cases the databases will always be the bottleneck so test our database first,so understand the limitations of the language first, like whether it is a single threaded or multi threaded language , if it's multi threaded make sure the critical sections are protected. If concurrency is happening make sure u are not losing out in consistency.

For every service there are APIs that need to always run , so make sure u load test these endpoints , these are the critical reads and critical writes.

Different Types of Testing-

Longevity Tests- tests memory leaks, ex. If u put a key in redis without setting TTL, restarting a machine after u reach a certain memory threshold would also work.

Security Testing- should not leak user data, prowler- cloud custodian are great tools

Like there are packages which can inject network latency so we can use these testing.

Chaos Testing - We don't want to tie our servers with cloud providers, so in chaos Testing we randomly delete a load balancer or a server. But when the servers goes down we need to handle it gracefully and our data must remain consistent.

Key metrics for monitoring databases -

How much disk space is being used, cache hit and miss rate , which queries are being frequently hit, slow queries, i/o consumption.

Key metrics for api servers- monitor CPU and memory , monitor network also if it is network heavy, how many api requests are we getting without it throttling. We need to know how many times we have to throttle. For asynchronous workers see how many workers will be there in the queue and check how much time it takes to process these

requests, also based on the service measure the cpu or memory, if it is something like encryption or decryption, video processing it would be a cpu heavy job