Microservices
UNIT 1

Chapter 3: Designing Microservice Systems:

Q9) The Systems Approach to Microservices:

•A microservices system encompasses all of the things about your organization that are related to the application it produces.
•This means that the structure of your organization, the people who work there, the way they work, and the outputs they produce are all important system factors.
•Equally important are runtime architectural elements such as service coordination, error handling, and operational practices.
•In addition to the wide breadth of subject matter that you need to consider, there is the additional challenge that all of these elements are interconnected—a change to one part of the system can have an unforeseen impact on another part.
•With all of the interconnected parts and the complex emergence that results, it is very difficult to understand how the parts work together.
•In particular, it is difficult to predict the results that can arise from a change to the system.
•So, they do what scientists have always done—they develop a model.
•The models mathematicians develop to study complex systems allow them to more accurately understand and predict the behavior of a system.
•Model-based approach can help all of us conceptualize our system of study and will make it easier for us talk about the parts of the system.

Q10) A microservice design model?

•A microservice design model comprised of five parts: Service, Solution, Process and Tools, Organization, and Culture.

●1. Service:

•The services form the atomic building blocks from which the entire organism is built.

●2. Solution:

•A solution architecture is distinct from the individual service design elements because it represents a macro view of our solution.
•When designing a particular microservice your decisions are bounded by the need to produce a single output—the service itself.
•Conversely, when designing a solution architecture your decisions are bounded by the need to coordinate all the inputs and outputs of multiple services.

●3. Process and Tools:

•The system behavior is also a result of the processes and tools that workers in the system use to do their job.
•Choosing the right processes and tools is an important factor in producing good microservice system behavior.

●4. Organization:

•How we work is often a product of who we work with and how we communicate.
•From a microservice system perspective, organizational design includes the structure, direction of authority, granularity, and composition of teams.

●5. Culture:

•We can broadly define culture as a set of values, beliefs, or ideals that are shared by all of the workers within an organization.
•Your organization’s culture is important because it shapes all of the atomic decisions that people within the system will make.

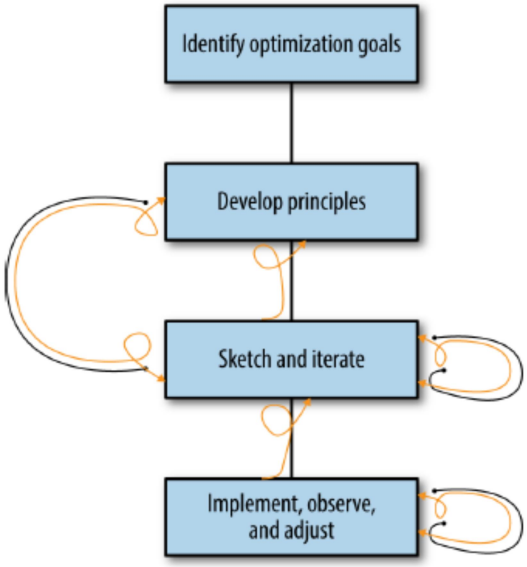
Q11) Standardization and Coordination?

- Policies, governance, and audits are all introduced as a way of policing the behavior of the system and ensuring that the actors conform.
- In other words, some parts of the system are standardized.
- No matter how many rules, checks, and governance methods you apply you are always at the mercy of actors in a system that can make poor decisions.
- Standardization is the enemy of adaptability and if you standardize too many parts of your system you risk creating something that is costly and difficult to change.
- **1. Standardizing process:**
- By standardizing the way that people work and the tools they use, you can influence the behavior in a more predictable way.
- For example, standardizing a deployment process that reduces the time for component deployment may improve the overall changeability of the system as the cost of new deployments decreases.
- The Agile methodology is a great example of process standardization.
- Where change should be introduced in small measurable increments that allow the organization to handle change easier.
- In addition to process standardization, most companies employ some form of tool standardization as well to define the types of tools their workers are allowed to utilize.
- **2. Standardizing outputs:**
- Output standardization is way of setting a universal standard for what that output should look like.
- Any deviation from the standard output is considered a failure.
- In a microservices system, a team takes a set of requirements and turns those into a microservice.
- So, the service is the output and the face of that output is the interface (or API) that provides access to the features and data the microservice provides.
- In the microservices context, output standardization often means developing some standards for the APIs that expose the services.
- Ex: HTTP interface
- **3. Standardizing people:**
- You can also decide to standardize the types of people that do the work within your organization.
- For example, you could introduce a minimum skill requirement for anyone who wants to work on a microservice team.
- Standardizing skills or talent can be an effective way of introducing more autonomy into your microservices system.
- If only the best and brightest are good enough to work within your system, be prepared to pay a high cost to maintain that standard.
- **Standardization trade-offs:**
- But keep in mind that while they aren't mutually exclusive, the introduction of different modes of standardization can create unintended consequences in other parts of the system.
- the decision to standardize the team's output has had unintended consequences on the team's work process.
- This happens because standardization is an attempt to remove uncertainty from our system, but comes at the cost of reducing innovation and changeability.
- The benefit of standardization is a reduction in the set of all possible outcomes.
- It gives us a way to shape the system by setting constraints and boundaries for the actions that people within the system can take.

- But this benefit comes at a cost.
- Standardization also constrains the autonomy of individual decision-makers.

Q12) A Microservices Design Process?

- the secret to great design is using the right design process.
- a good designer employs a process that helps them continually get closer to the best product.



- **Set Optimization Goals:**
- identify the goals that make sense for your particular situation.
- A smaller set of optimization goals is easier to design for.
- A single optimization goal provides the most clarity and has a higher likelihood of succeeding.
- For example, a financial information system might be optimized for reliability and security above all other factors.
- That doesn't mean that changeability, usability, and other system qualities are unimportant—it simply means that the designers will always make decisions that favor security and reliability above all other things.
- **Development Principles:**
- Principles outline the general policies, constraints, and ideals that should be applied universally to the actors within the system to guide decision-making and behavior.
- The best designed principles are simply stated and easy to understand.
- **Sketch the System Design:**
- a good approach is to sketch the important parts of your system design for the purposes of evaluation and iteration.
- serialize some of the abstract concepts from your head into a tangible form that can be evaluated.
- The goal is to sketch out the core parts of your system, including organizational structure (how big are the teams? what is the direction of authority? who is on the team?), the solution architecture (how are services organized? what infrastructure must be in place?), the service design (what outputs? how big?), and the processes and tools (how do services get deployed? what tools are necessary?).
- **Implement, Observe, and Adjust:**
- Good designers make small system changes, assess the impact of those changes, and continually prod the system behavior toward a desired outcome.

- It is more realistic to gain essential visibility into our system by identifying a few key measurements that give us the most valuable information about system behavior.
- In organizational design, this type of metric is known as a key performance indicator (KPI).
- Identify the right KPIs, be able to interpret the data, and make small, cheap changes to the system that can guide you back on the right course.

Q13) The Microservices System Designer?

- The microservices system designer should be able to enact change to a wide array of system concerns.
- Organization, culture, processes, solution architecture, and services are significant concerns for the system designer.
- You could decide that the system boundaries should mirror the boundaries of the company.
- The microservices system designer or software system designer is responsible for all the elements of the bounded system.
- The implication is that there is a world within the system and world outside of these borders.
- The system designer's task is to introduce small changes within the system in order to produce behavior that will align with the desired goal.
- Responsibilities are segregated among specialists who may not share the same objectives:
- The solution architect focuses on the coordination of services, the team manager focuses on the people, and the service developer focuses on the service design.
- Someone or some team must be responsible for the holistic view of the entire system for a microservices system to succeed.

CHAPTER 4: Establishing a Foundation:

Q14) Goals and Principles?

- It is important to have some overall goals and principles to help inform your design choices and guide the implementation efforts.
- The more autonomy you allow, the more guidance and context you need to provide to those teams.
- **Goals for the Microservices Way:**
- It is a good idea to have a set of high-level goals to use as a guide when making decisions about what to do and how to go about doing it.
- Ultimate goal in building applications in the microservices way: finding the right harmony of speed and safety at scale.
- It might take a very long time for you to find that perfect harmony of speed and safety at scale if you are starting from scratch.
- So, you don't need to reinvent established software development practices.
- Instead, you can experiment with the parameters of those practices.
- the four goals to consider:
- 1. Reduce Cost: Will this reduce overall cost of designing, implementing, and maintaining IT services?
- 2. Increase Release Speed: Will this increase the speed at which my team can get from idea to deployment of services?
- 3. Improve Resilience: Will this improve the resilience of our service network?
- 4. Enable Visibility: Does this help me better see what is going on in my service network?

- **1. Reduce cost:**
- The ability to reduce the cost of designing, implementing, and deploying services allows you more flexibility when deciding whether to create a service at all.
- In the operations world, reducing costs was achieved by virtualizing hardware.
- For microservices, this means coming up with ways to reduce the cost of coding and connecting services together.
- Templated component stubs, standardized data-passing formats, and universal interfaces are all examples of reducing the costs of coding and connecting service components.
-
- **2. Increase release speed:**
- Increasing the speed of the "from design to deploy" cycle is another common goal.
- A more useful way to view this goal is that you want to shorten the time between idea and deployment.
- Sometimes, you don't need to "go faster," you just need to take a shortcut.
- When you can get from idea to running example quickly, you have the chance to get feedback early, to learn from mistakes, and iterate on the design more often before a final production release.
- One place where you can increase speed is in the deployment process.
- By automating important elements of the deployment cycle, you can speed up the whole process of getting services into production.
-
- **3. Improve resilience:**
- It is also important to build systems that can "stand up" to unexpected failures.
- In other words, systems that don't crash, even when errors occur.
- Ex 1: One of the ways DevOps practices has focused on improving resilience is through the use of automated testing.
- By making testing part of the build process, the tests are constantly run against checked-in code, which increases the chances of finding errors in the code.
- Ex 2: Blue-green deployment: a new release is placed in production with a small subset of users and, if all goes well during a monitoring phase, more users are routed to the new release until the full userbase is on the new release.
- If any problems are encountered during this phased rollout, the users can all be returned to the previous release until problems are resolved and the process starts again.
-
- **4. Enable visibility:**
- Improve the ability of stakeholders to see and understand what is going on in the system.
- We often get reports on the coding backlog, how many builds were created, the number of bugs in the system versus bug completed, and so on.
- But we also need visibility into the runtime system.
- The DevOps practices of logging and monitoring are great examples of this level of runtime visibility.
- "If it moves graph it. If it matters, alert on it".
- Most effort to date has been to log and monitor operation-level metrics (memory, storage, throughput, etc.).

Q15) Operating Principles?

- Unlike goals, which are general, principles offer more concrete guidance on how to act in order to achieve those goals.
- Principles are not rules—they don't set out required elements.
- Instead, they offer examples on how to act in identifiable situations.

- Principles can also be used to inform best practices.
- **Ex1: Netflix**
- Netflix's cloud architecture and operating principles :
- **1. Antifragility:**
- Netflix works to strengthen their internal systems so that they can withstand unexpected problems.
- "The point of antifragility is that you always want a bit of stress in your system to make it stronger."
- "Simian Army" set of tools, which "enforce architectural principles, induce various kinds of failures, and test our ability to survive them".
-
- **2. Immutability:**
- Allows system to "scale horizontally."
- "Red/Black pushes": Although each released component is immutable, a new version of the service is introduced alongside the old version, on new instances, then traffic is redirected from old to new.
- After waiting to be sure all is well, the old instances are terminated.
-
- **3. Separation of Concerns:**
- (SoC) in the engineering team organization.
- Each team owns a group of services.
- They own building, operating, and evolving those services, and present a stable agreed interface and service level agreement to the consumers of those services.
- an organization structured with independent selfcontained cells of engineers will naturally build what is now called a microservice architecture.
-
- **Ex2: UNIX**
- **1. Make each program do one thing well.**
- To do a new job, build afresh rather than complicate old programs by adding new features.
- **2. Expect the output of every program to become the input to another.**
- Don't insist on interactive input.
- **3. Design and build software, even operating systems, to be tried early.**
- Don't hesitate to throw away the clumsy parts and rebuild them.
- **4. Use tools in preference to unskilled help to lighten a programming task.**

Q16) Suggested principles?

- **1. Do one thing well:**
- Many microservice implementations adopt the essential message—"do one thing well," which leads to the challenge of deciding what constitutes "one thing" in your implementation.
- For some, "one thing" is managing user accounts.
- **2. Build afresh:**
- It may be better to build a new microservice component rather than attempt to take an existing component already in production and change it to do additional work.
- **3. Expect output to become input:**
- For Unix systems, this leads to reliance on text strings as the primary data-passing medium.
- On the Web, the data-passing medium is the media type (HTML,JSON, etc.).

- you can even use HTTP's content-negotiation feature to allow API providers and consumers to decide for themselves at runtime which format will be used to pass data.
- **4. Don't insist on interactive input:**
- This means humans don't need to be engaged every step of the way—the scripts handle both the input and the output on their own.
- Reducing the dependency on human interaction in the software development process can go a long way toward increasing the speed at which change occurs.
- **5. Try early:**
- You will learn your mistakes early.
- It is also a way to encourage teams to get in the habit of releasing early and often.
- The earlier you release, the earlier you get feedback and the quicker you can improve.
- **6. Don't hesitate to throw it away:**
- Being willing to throw something away can be hard when you've spent a great deal of time and effort building a component.
- However, when you adopt the "try early" principle, throwing away the early attempts is easier.
- It is also important to consider this "throw it away" principle for components that have been running in production for a long time.
- Over time, components that did an important job may no longer be needed.
- **7. Toolmaking:**
- You sometimes need to build the "right tool" for the job.
- Companies that created their own developer and deployment tool chains in order to improve their overall developer experience.
- Sometimes these tools are built from existing open source software projects.
- Sometimes the tools are, themselves, passed into open source so that others can use them and contribute to improving and maintaining them.
- The important element here is to recognize that, in some cases, you may need to divert from building your solution and spend some time building tools to help you build that solution.

Q17) Platforms?

- From a microservice architecture perspective, good platforms increase the harmonic balance between speed and safety of change at scale.
- With a platform we pass from the conceptual world to the actual world.
- The challenge is that it seems every company is doing this their own way, which presents some choices to anyone who wants to build their own microservice environment.
- Do you just select one of the existing OSS platforms?
- Do you try to purchase one? Build one from scratch?
- "microservice concerns": divide them into two groups: shared capabilities and local capabilities.
- **1. Shared Capabilities:**
- Shared capabilities are platform services that all teams use.
- These are standardized things like container technology, policy enforcement, service orchestration/interop, and data storage services.
- It is important to note that shared services does not mean shared instance or shared data.
- **The following is a quick rundown of what shared services platforms usually provide:**
- **1] Hardware services:**
- In some companies there is a team of people who are charged with accepting shipments of hardware , populating those machines with a baseline OS and common software for

monitoring, health checks, etc., and then placing that completed unit into a rack in the "server room" ready for use by application teams.
- Another approach is to virtualize the OS and baseline software package as a virtual machine (VM).
- **2] Code management, testing, and deployment:**
- Once you have running servers as targets, you can deploy application code to them.
- That's where code management (e.g., source control and review), testing, and (eventually) deployment come in.
- Most microservice shops go to considerable lengths to automate this part of the process.
- For example, the Amazon platform offers automation of testing and deployment that starts as soon a developer checks in her code.
- **3] Data stores:**
- It is usually not effective for large organizations to support all possible storage technologies.
- Even today, some organizations struggle with providing proper support for the many storage implementations they have onsite.
- It makes sense for your organization to focus on a select few storage platforms and make those available to all your developer teams.
- **4] Service orchestration:**
- The technology behind service orchestration or service interoperability is another one that is commonly shared across all teams.
- There is a wide range of options here.
- Many of the flagship microservice companies (e.g., Netflix and Amazon) wrote their own orchestration platforms.
- **5] Security and identity:**
- This often happens at the perimeter via gateways and proxies.
- There are also a number of security products available. Shared identity services are sometimes actually external to the company.
- **6] Architectural policy:**
- These are services that are used to enforce company-specific patterns or models—often at runtime through a kind of inspection or even invasive testing.
-
- **Local Capabilities:**
- Local capabilities are the ones that are selected and maintained at the team or group level.
- One of the primary goals of the local capabilities set is to help teams become more self-sufficient.
- This allows them to work at their own pace.
- It is common to allow teams to make their own determination on which developer tools, frameworks, support libraries, config utilities, etc., are best for their assigned job.
- Finally, it is important that the team making the decision is also the one taking responsibility for the results.
- **the common local capabilities for microservice environments:**
- **1. General tooling:**
- A key local capability is the power to automate the process of rolling out, monitoring, and managing VMs and deployment packages.
- **2. Runtime configuration:**
- A pattern found in many organizations using microservices is the ability roll out new features in a series of controlled stages.
- This allows teams to assess a new release's impact on the rest of the system.

- **3. Service discovery:**
- These tools make it possible to build and release services that, upon install, register themselves with a central source, and then allow other services to "discover" the exact address/location of each other at runtime.
- This ability to abstract the exact location of services allows various teams to make changes to the location of their own service deployments without fear of breaking some other team's existing running code.
- **4. Request routing:**
- Once you have machines and deployments up and running and discovering services, the actual process of handling requests begins.
- All systems use some kind of request-routing technology to convert external calls (usually over HTTP, Web- Sockets, etc.) into internal code execution (e.g., a function somewhere in the codebase).
- **5. System observability:**
- A big challenge in rapidly changing, distributed environments is getting a view of the running instances—seeing their failure/success rates, spotting bottlenecks in the system, etc.

Q18) Culture?

- Culture is important because it not only sets the tone for the way people behave inside an organization, but it also affects the output of the group.
- The code your team produces is the result of the culture.
- Three aspects of culture that you should consider as a foundation for your microservice efforts:
- **1.Communication:**
- Research shows that the way your teams communicate (both to each other and to other teams) has a direct measurable effect on the quality of your software.
- Focus on Communication:
- "organizational metrics are significantly better predictors of error-proneness" in code than other more typical measures including code complexity and dependencies.
- The process of deciding things like the size, membership, even the physical location of teams is going to affect the team choices and, ultimately, the team output.
- **2. Team alignment:**
- The size of your teams also has an effect on output.
- More people on the team means essentially more overhead.
- Aligning Your Teams:
- Robin Dunbar found that social group sizes fall into predictable ranges.
- "The various human groups that can be identified in any society seem to cluster rather tightly around a series of values (5, 12, 35, 150, 500, and 2,000)."
- These groups each operate differently.
- The first (5) relies very much on a high-trust, low-conversation mode: they seem to understand each other without lots of discussion.
- Dunbar found that, as groups get larger, more time is spent on maintaining group cohesion.
- "adding [more people] to a late software project makes it later."
- "two-pizza team" rule: Any team that cannot be fed by two pizzas is a team that is too big.
- **3. Fostering innovation:**
- Innovation can be disruptive to an organization but it is essential to growth and long-term success.

- A simple definition of innovate is “to do something in a new way; to have new ideas about how something can be done.”
- A common challenge is that the innovation process can be very disruptive to an organization.
- Sometimes “changing the way we do things” can be seen as a needless or even threatening exercise—especially if the change will disrupt some part of the organization.
- Fostering innvotation means setting boundaries that prevent teams from taking actions that threaten the health and welfare of the company and allowing teams to act on their own within these safe boundaries.