Microservices
UNIT 2
Chapter 2: System Design and Operations

Q1) What is Independent Deployability?

- Each microservice must be deployable completely independent of any other microservice.
- Independent deployability allows us to perform selective or on-demand scaling;
- if a part of the system experiences high load we can re-deploy or move that microservice to an environment with more resources, without having to scale up hardware capacity for the entire, typically large, enterprise system.
- Scaling hardware resources on-premise can be extremely costly—we have to buy expensive hardware in anticipation of the usage rather than in response to actual usage.
- Team can deploy safe microservices to a public/ private cloud, where scaling of resources is significantly cheaper.
- Two different teams would be responsible for the development of separate microservices (e.g., Customer Management and Shipment Management).
- If the first team, which is responsible for the Customer Management microservice, needs to make a change and rerelease, but Customer Management cannot be released independent of the Shipment Management microservice, we now need to coordinate Customer Management's release with the team that owns Shipment Management.
- Such coordination can be costly and complicated, since the latter team may have completely different priorities from the team responsible for Customer Management.
- Eliminating costly cross-team coordination challenges is indeed a significant motivation for microservice adopters.

Q2) Why we need for More Servers?

- To ensure independent deployability, we need to develop, package, and release every microservice using an autonomous, isolated unit of environment.
- In reality, lightweight packaging solutions, such as JAR, WAR, and EAR archives in Java don't provide sufficient modularity and the level of isolation required for microservices.
- WAR files still share system resources like disk, memory, shared libraries, the operating system, etc.
- Case in point: a WAR or EAR file will typically expect a specific version of Java SDK and application server to be present in the environment.
- They may also expect specific versions of OS libraries in the environment.
- One application's environmental expectations can be drastically different from another's, leading to version and dependency conflicts if we need to install both applications on the same server.
- For companies that have been using microservice architecture for a number of years, it is not uncommon to develop and maintain hundreds of microservices.
- Let's assume you are a mature microservices company with about 500 microservices.
- To deploy these microservices in a reliable, redundant manner you will need at least three servers/VMs per each microservice, resulting in 1,500 servers just for the production system.
- Typically, most companies run more than one environment (QA, stage, integration, etc.), which quickly multiplies the number of required servers.
- The deployment unit universally used for releasing and shipping microservices is a container.

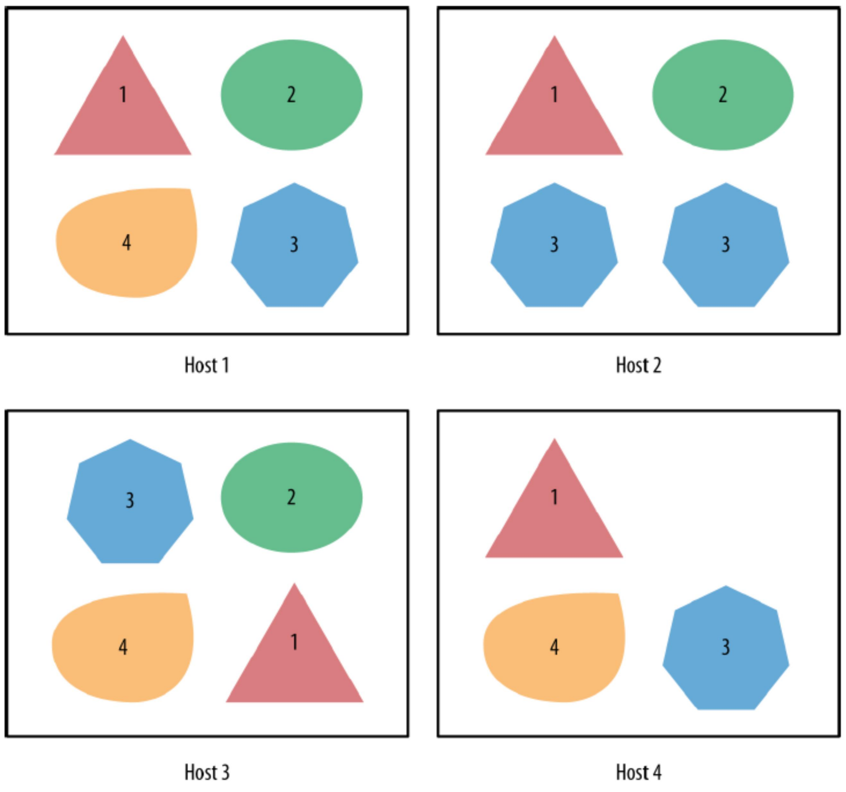
- If you have never used containers before, you can think of a container as of an extremely lightweight “virtual machine.” The technology is very different from that of conventional VMs.
- It is based on a Linux kernel extension (LXC) that allows running many isolated Linux environments (containers) on a single Linux host sharing the operating system kernel, which means we can run hundreds of containers on a single server or VM and still achieve the environment isolation and autonomy
- Containers provide a modern isolation solution with practically zero overhead.
- Currently the most widely deployed container toolset is Docker, so in practice Docker and containers have become somewhat synonymous.

Q3)What are Docker and Microservices ?

- Containers were not created for microservices.
- They emerged as a powerful response to a practical need: technology teams needed a capable toolset for universal and predictable deployment of complex applications.
- Indeed, by packaging our application as a Docker container, which assumes prebundling all the required dependencies at the correct version numbers, we can enable others to reliably deploy it to any cloud or on-premise hosting facility, without worrying about target environment and compatibility.
- The only remaining deployment requirement is that the servers should be Docker-enabled.
- In comparison, if we just gave somebody our application as an executable, without prebundled environmental dependencies we would be setting them up for a load of dependency pain.
- Alternatively if we wanted to package the same software as a VM image, we would have to create multiple VM images for several major platforms, since there is no single, dominant VM standard currently adopted by major players.
- Linux containers use a layered filesystem architecture known as union mounting.
- This allows a great extensibility and reusability not found in conventional VM architectures.
- Centralized registries, discovery services, and community oriented platforms such as Docker Hub and GitHub further facilitate quick adoption and education in the space.
- One core principle is prominently outlined in the Docker documentation itself:
- “ Run only one process per container. In almost all cases, you should only run a single process in a single container. Decoupling applications into multiple containers makes it much easier to scale horizontally and reuse containers.”

Q4)Explain the Role of Service Discovery.

- How you distribute your services across your available hosts will depend on your business and technical needs and very likely may change over time.
- Hosts are just servers, they are not guaranteed to last forever.
- Ex: what the nonuniform distribution of your services may look like at some point in time if you have four hosts with four containers.



- We have Microservice 1 deployed on all four hosts, Microservice 2 is only on hosts 1–3.
- Keep in mind that the deployment topology may change at any time, based on load, business rules, which host is available, and whether an instance of your microservice suddenly crashes or not.
- If we allocated an IP per microservice, the IP address allocation + assignment would become too complicated.
- Instead, we allocate an IP per host (server) and the microservice is fully addressed with a combination of:
- 1. IP address (of the host)
- 2. Port number(s) the service is available at on the host
- This is where service discovery enters the microservices scene.
- We need some system that will keep an eye on all services at all times and keep track of which service is deployed on which IP/port combination at any given time, so that the clients of microservices can be seamlessly routed accordingly.
- As an architect, we need to decide how much automation "magic" we want from our tools versus how much control we need to retain for ourselves.

Q5)Discuss the Need for an API Gateway.

- A common pattern observed in virtually all microservice implementations is teams securing API endpoints, provided by microservices, with an API gateway.
- API gateway features:
- **1. Security:**
- Things can go horribly wrong securitywise when there are many moving parts.
- We certainly need some law and order to keep everything in control and safe.
- APIs provided by microservices may call each other, may be called by "frontend," i.e., public-facing APIs, or they may be directly called by API clients such as mobile applications, web applications, and partner systems.
- To make sure we never compromise the security of the overall system, the widely recommended approach is to secure invocation of "public-facing" API endpoints of the microservices-enabled system using a capable API gateway.

- Sometimes, certain microservices are deemed “internal” and excluded from the security provided by an API Gateway, as we assume that they can never be reached by external clients.
- This is dangerous since the assumption may, over time, become invalid.
- It’s better to always secure any API/microservice access with an API gateway.
-
- **2. Transformation and Orchestration:**
- The Web is a distributed system.
- Due to its distributed nature it uses “chatty” interfaces.
- Those are interfaces where you need to make many calls to get the data required for a single task.
- Example: Shipping, Inc.’s microservice : Developing an “intelligent” inventory management system.
- The purpose of the new system is to analyze properly anonymized data about millions of shipments passing through Shipping, Inc., combine this insight with all of the metadata that is available on the goods being shipped, determine behavioral patterns of the consumers, and utilizing human + machine algorithms design a “recommendation engine” capable of suggesting optimal inventory levels to Shipping, Inc.’s “platinum” customers.
- If the team is building this system using a microservice architecture, they could end
- up creating two microservices for the main functionality:
- 1. Recommendations microservice, which takes user information in, and responds with the list containing the recommendations—i.e., suggested stock levels for various products that this customer typically ships.
- 2. Product Metadata microservice, which takes in an ID of a product type and retrieves all kinds of useful metadata about it.
- Let’s say the page size is 20, so 20 suggestions at a time. the user-interface team will have to make 21 HTTP calls: one to retrieve the recommendations list and then one for each recommendation to retrieve the details, such as product name, dimensions, size, price, etc.
- The user-interface team is not happy. They wanted a single list; but instead are forced to make multiple calls (the infamous “N+1 queries” problem).
- Additionally, the calls to the Product Metadata microservice return too much information which is an issue for, say, mobile devices on slow connections.
- A capable API gateway will allow you to declaratively, through configuration, create API interfaces that can orchestrate backend microservices and “hide” their granularity behind a much more developer-friendly interface and eliminate chattiness.
- In our example scenario, we can quickly aggregate the N+1 calls into a single API call and optimize the response payload.
- This gives mobile developers exactly what they need: a list of recommendations via a single query, with exactly the metadata they required.
- The calls to backend microservices will be made by the API gateway.
- Good API gateways can also parallelize the twenty calls to the Product Metadata microservice, making the aggregate call very fast and efficient.
-
- **3. Routing:**
- However, directly providing tuples of the IP/port combinations to route an API client is not an adequate solution.
- A proper solution needs to abstract implementation details from the client.

- An API client still expects to retrieve an API at a specific URI, regardless of whether there's a microservice architecture behind it and independent of how many servers, Docker containers, or anything else is serving the request.
- An API gateway hide the complexities of routing to a microservice from the client apps.
- An API gateway can interface with either HTTP or DNS interfaces of a service discovery system and route an API client to the correct service when an external URI associated with the microservice is requested.

Q6)Explain with example Monitoring and Alerting.

- Let's take Consul as an example.
- Not only does Consul know how many active containers exist for a specific service, marking a service broken if that number is zero, but Consul also allows us to deploy customized health-check monitors for any service.
- This can be very useful. Indeed, just because a container instance for a microservice is up and running doesn't always mean the microservice itself is healthy.
- We may want to additionally check that the microservice is responding on a specific port or a specific URL, possibly even checking that the health ping returns predetermined response data.
- In addition to the "pull" workflow in which Consul agents query a service, we can also configure "push"-oriented health checks, where the microservice itself is responsible for periodically checking in, i.e., push predetermined payload to Consul.
- If Consul doesn't receive such a "check-in," the instance of the service will be marked "broken."
- This alternative workflow is very valuable for scheduled services that must run on predetermined schedules.
- It is often hard to verify that scheduled jobs do run as expected, but the "push"-based health-check workflow can give us exactly what we need.
- Once we set up health checks we can install an open source plug-in called Consul Alerts, which can send service failure and recovery notifications to incident management services such as PagerDuty or OpsGenie.
- These are powerful services that allow you to set up sophisticated incident-notification phone trees and/or notify your tech team via email, SMS, and push notifications through their mobile apps.