Prerequisites

Summary

For this course, students should be comfortable with:

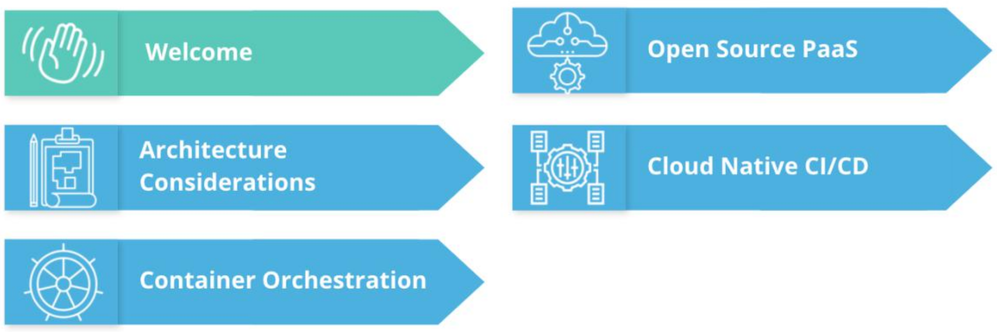
- **web application development with Python**
- using the **CLI** or command-line interface
- using **git commands**
- creating a **DockerHub account**

These pre-requisites will be used throughout the course.

Course Outline

Summary

Throughout the course, we will walk through a realistic example of applying good development practices and containerizing an application, before it's released to a Kubernetes cluster using an automated CI/CD pipeline.



Microservice Fundamentals course outline

This **course** has the following lessons:

- Welcome
- Architecture Considerations
- Container Orchestration
- Open Source PaaS
- Cloud Native CI/CD

In the first lesson, we will cover:

- Introduction to Cloud Native
- CNCF and Cloud Native tooling
- Stakeholders
- Tools, Environment & Dependencies

CNCF Cloud Native Computing Foundation

An overview of the Cloud-Native Ecosystem. Then Architectural models to be considered while deploying an application. Thereafter covering about

a) design pattern such a monoliths and microservices as well as best practices to adapt at the implementation stage to optimize an application.

b) How to **package** an **application using Dockers** and **deploy** it to **Kubernetes cluster** using i**mperative** and **declarative configurations.**

c) **Deploy** a **Kubernetes cluster** using **K3s**.

d) **Evaluate** the platform as a service or a **PaaS solutions** and how we can **use Cloud Foundry** to **deploy** an **application without worrying about the underlying infrastructure.**

e) How to **use Cloud-Native tooling** to **construct a CI/CD pipeline**.

f) Deep-dive into **GitHub** actions and **Argo CD** as **deployment mechanisms** and **template configuration managers such as Helm**.

First lessons highlights

a) An introduction to Cloud-Native and the principles that it advocates.

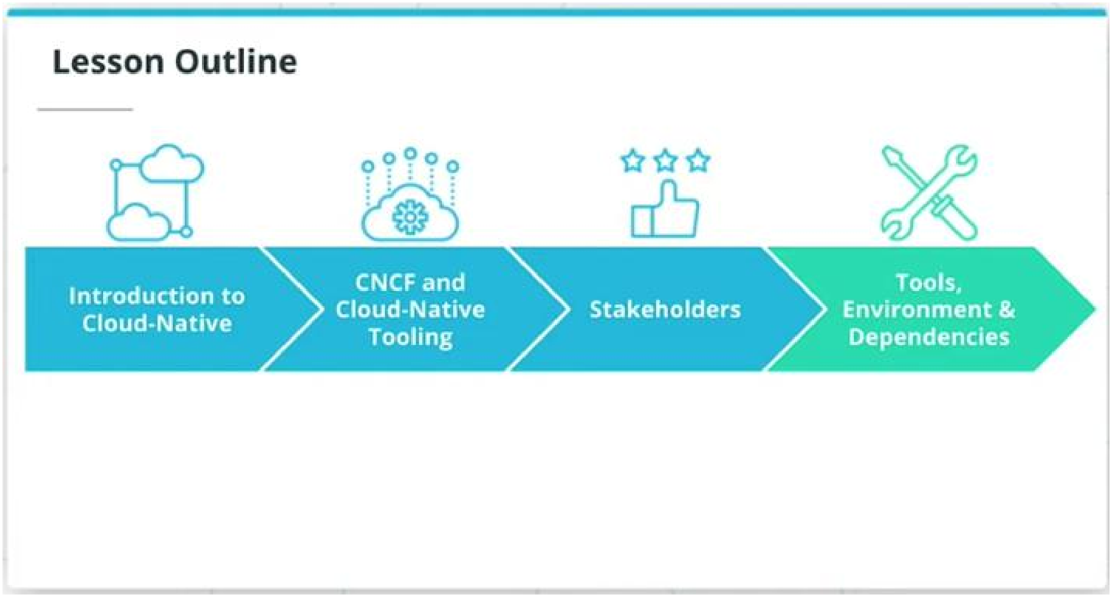
b) Containers and how these are closely correlated with a microservice-based architecture.

c) Various tools in the Cloud-Native ecosystem.

d) Kubernetes as a framework to manage containers at scale and it integration with tools that provide networking, storage, service mesh, traceability, and many more

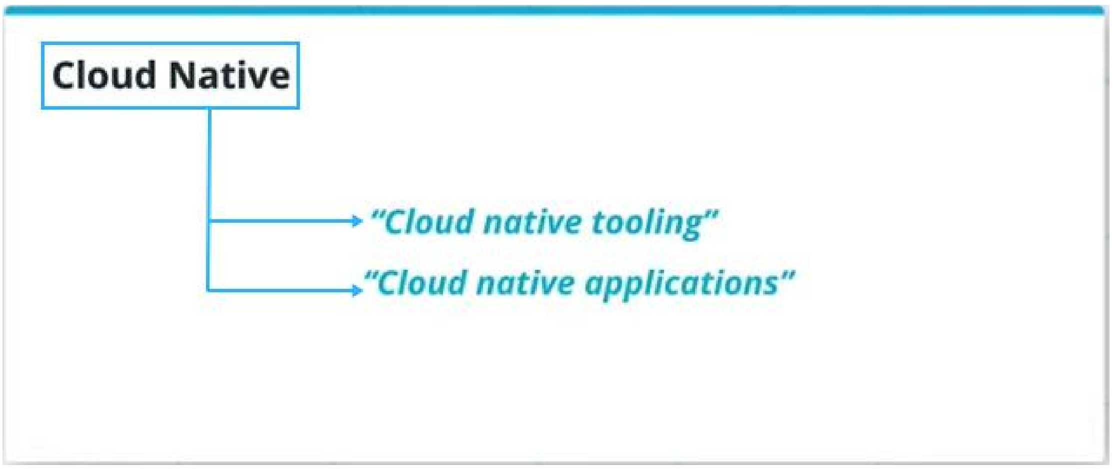
e) Main stakeholders that will consider **adoption of Cloud-Native tooling**. We will evaluate what are the key points to consider from a technical and business perspective.

f) We will have list of tools applications that we will need to install on our machines.


Lesson Outline
Introduction to Cloud-Native
CNCF and Cloud-Native Tooling
Stakeholders
Tools, Environment & Dependencies

Introduction to Cloud-Native

Transcript:

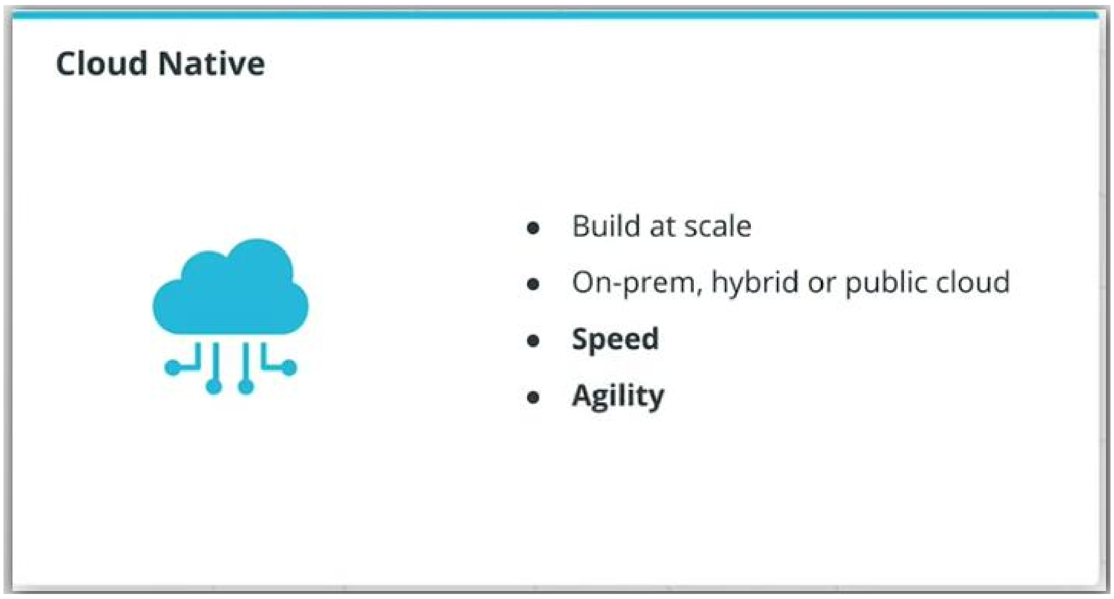


Cloud Native

"Cloud native tooling"

"Cloud native applications"

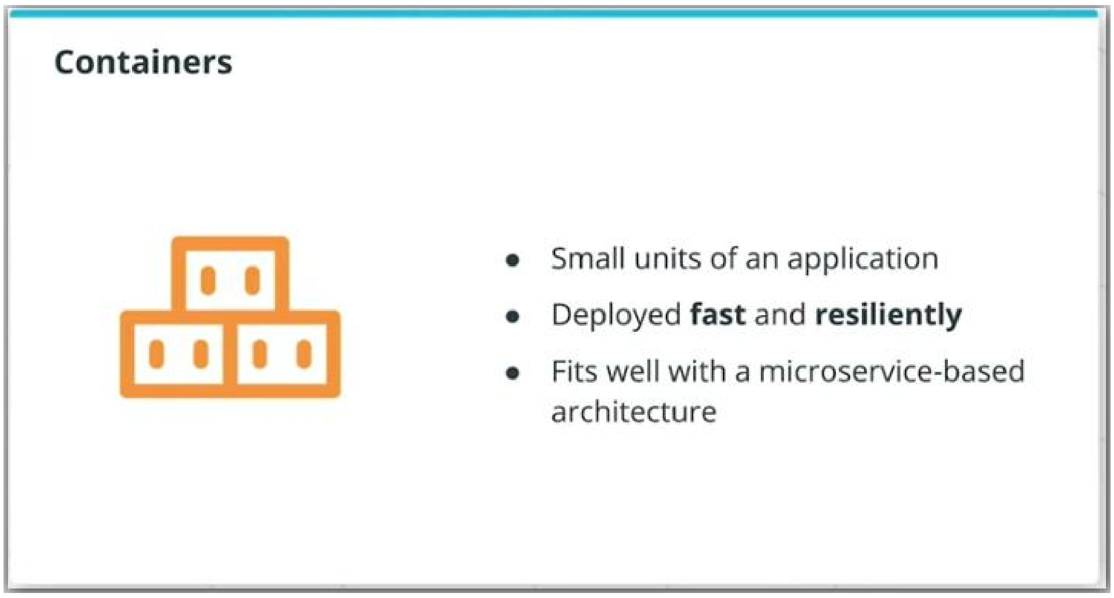
In past years, you might have heard about cloud native tooling or a cloud native applications. But what does it actually mean?



Cloud Native

- Build at scale
- On-prem, hybrid or public cloud
- **Speed**
- **Agility**

Cloud native refers to the set of practices that empowers an organization to build and manage applications at scale while using private, hybrid and public cloud providers. However, the key to cloud native deployment is speed and agility,or how quickly an organization can respond to change and increase its feature velocity.



Containers

- Small units of an application
- Deployed **fast** and **resiliently**
- Fits well with a microservice-based architecture

Usually, containers are closely associated with the cloud native terminology. Containers are small and manageable units that have an application running inside. A container can be deployed fast, resiliently and easily managed. A such, often a microservice–based architecture is chosen for the cloud native software.
As this represent a collection of small independent services that can be easily containerized.

Summary

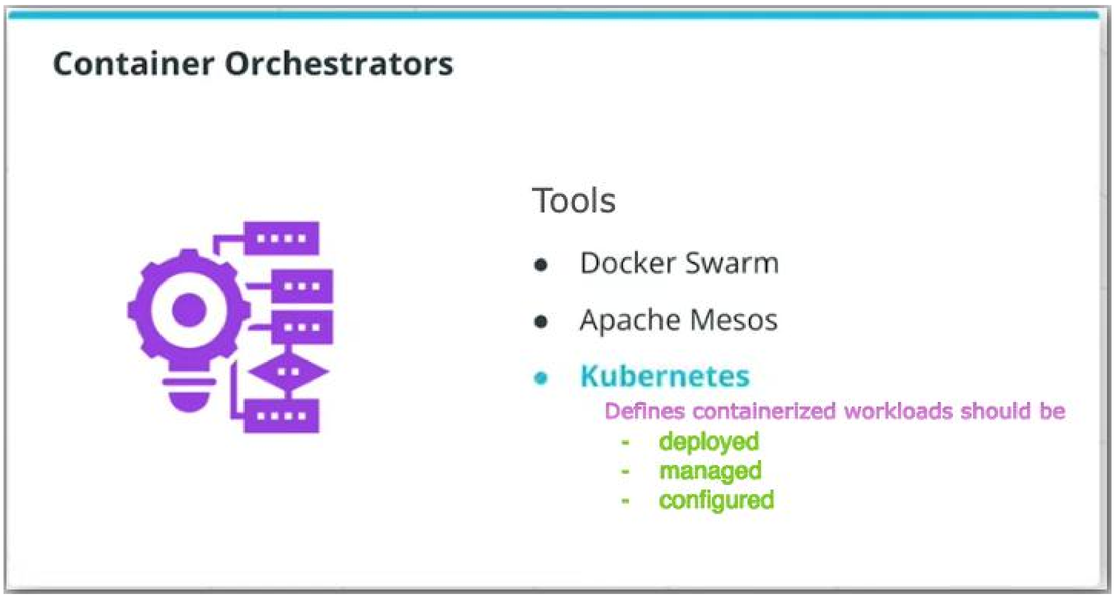
Cloud–native refers to the set of practices that empowers an organization to **build and manage applications at scale**. They can achieve this goal by using private, hybrid, or public cloud providers. In addition to scale, an organization needs to be agile in integrating customer feedback and adapting to the surrounding technology ecosystem.

Containers are closely associated with cloud–native terminology. Containers are used to run a single application with all required dependencies. The main characteristics of containers are easy to manage, deploy, and fast to recover. As such, often, a microservice–based architecture is chosen in tandem with cloud–native tooling. Microservices are used to manage and configure a collection of small, independent services that can be easily packaged and executed within a container.

CNCF and Cloud–Native Tooling

CNCF refers to **Cloud Native Computing Foundation**

Transcript:

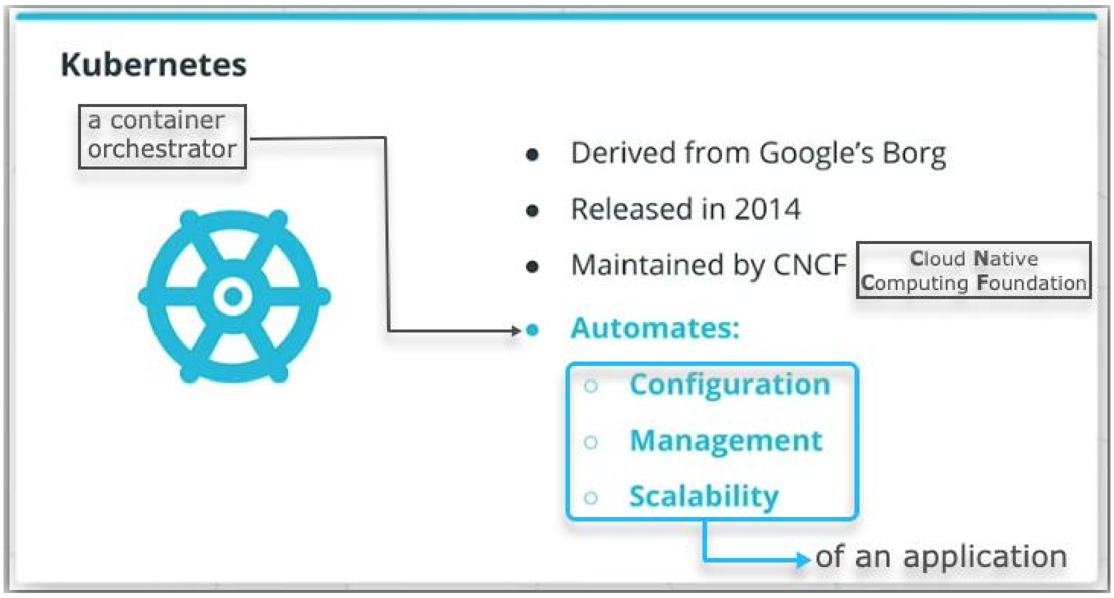


Container Orchestrators

Tools

- Docker Swarm
- Apache Mesos
- Kubernetes
  Defines containerized workloads should be
  - deployed
  - managed
  - configured

With the pairs of containers, it was imperative to introduce a tool that would managed containers at scale. Over time, multiple tools appeared on the radar, such as Docker Swarm, Apache Mesos, and Kubernetes. However, Kubernetes took the lead in defining how containerized workloads should be deployed, managed, and configured.



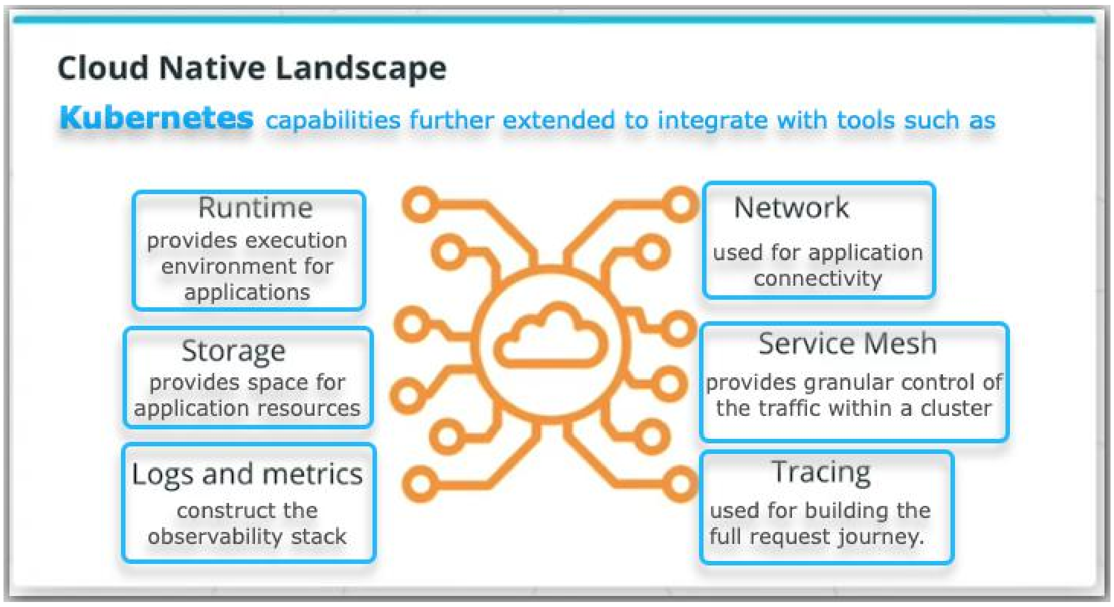
Kubernetes

a container orchestrator

- Derived from Google's Borg
- Released in 2014
- Maintained by CNCF (Cloud Native Computing Foundation)
- **Automates:**
  - **Configuration**
  - **Management**
  - **Scalability**

of an application

Kubernetes derives from Borg at Google open–source software that orchestrate containers. It had its first initial release in 2014, and it's now maintained under the CNCF umbrella or Cloud Native Computing Foundation. Overall, Kubernetes is a container orchestrator that automates the configuration, management and scalability of an application.



Cloud Native Landscape

Kubernetes capabilities further extended to integrate with tools such as

Runtime provides execution environment for applications

Storage provides space for application resources

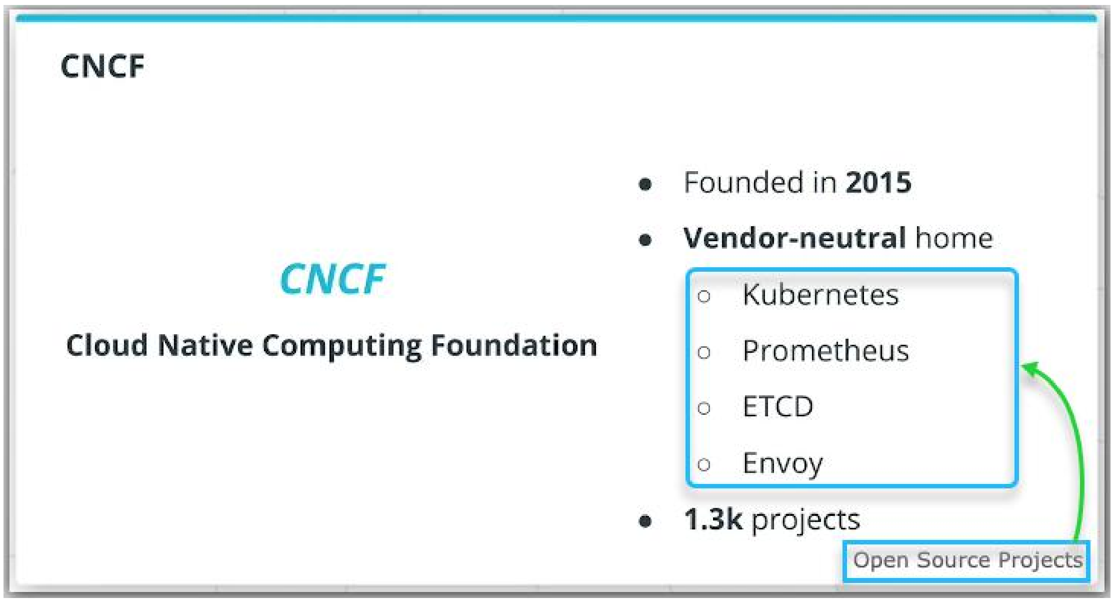
Logs and metrics construct the observability stack

Network used for application connectivity

Service Mesh provides granular control of the traffic within a cluster

Tracing used for building the full request journey.

Overtime, Kubernetes capabilities were further extended to integrate with tools that provide functionalities such as runtime for application execution environment, network, for application connectivity, storage for application resources, service mesh for granular control of the traffic within a cluster, logs and metrics to construct the observability stack, and tracing for building the full request journey and many more.



CNCF

CNCF

Cloud Native Computing Foundation

- Founded in **2015**
- **Vendor-neutral** home
  - Kubernetes
  - Prometheus
  - ETCD
  - Envoy
- **1.3k** projects

Open Source Projects

A large part of these tools are maintained under the CNCF umbrella. CNCF was founded in 2015, and it provides a vendor–neutral home to open source projects, such as Kubernetes, Prometheus, ETCD, Envoy and many more. Currently there are more than 1,300 projects associated with the CNCF.

Summary

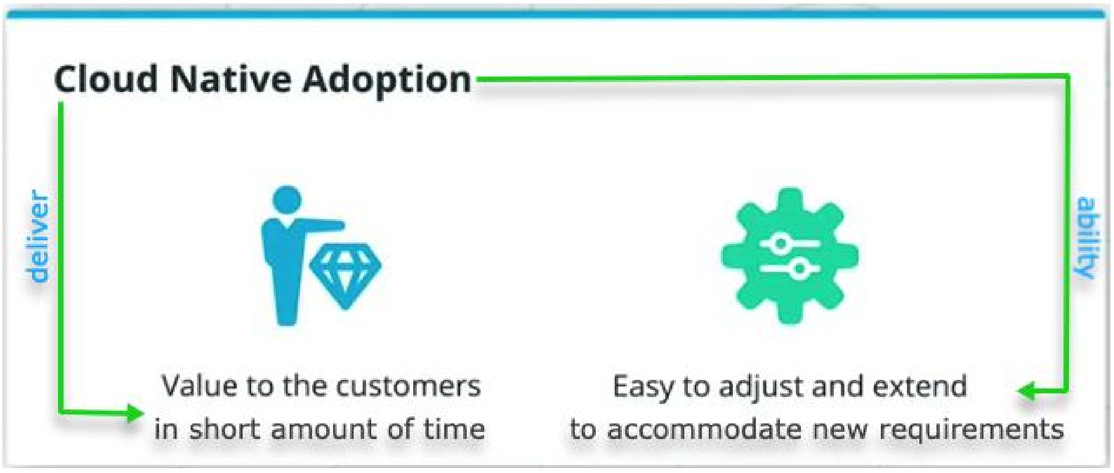
Kubernetes had its first initial release in 2014 and it derives from Borg, a Google open–source container orchestrator. Currently, Kubernetes is part of **CNCF** or **Cloud Native Computing Foundation**. CNCF was founded in 2015, and it provides a **vendor–neutral home to open–source projects** such as Kubernetes, Prometheus, ETCD, Envoy, and many more.
Overall, Kubernetes is a container orchestrator that is capable to solutionize the integration of the following functionalities:

- Runtime
- Networking
- Storage
- Service Mesh
- Logs and metrics
- Tracing

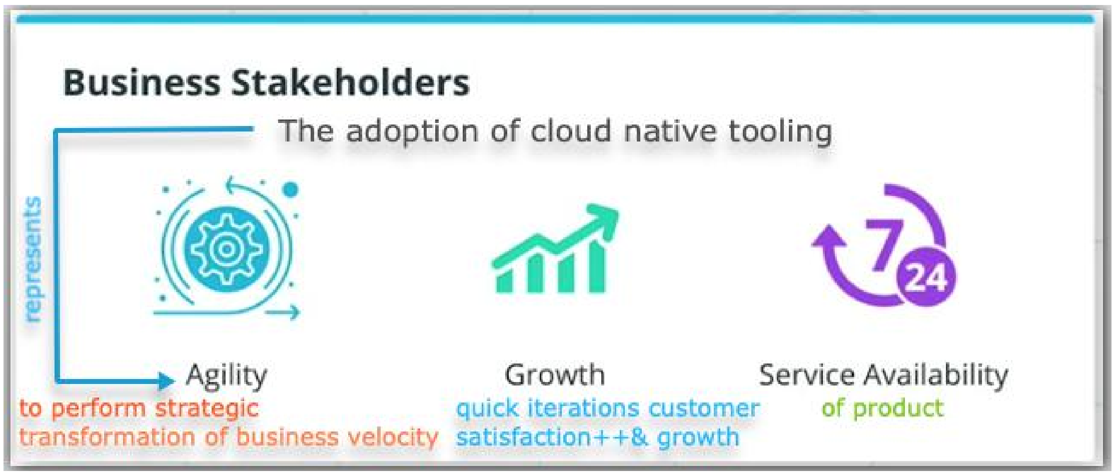
Stakeholders

Transcript:

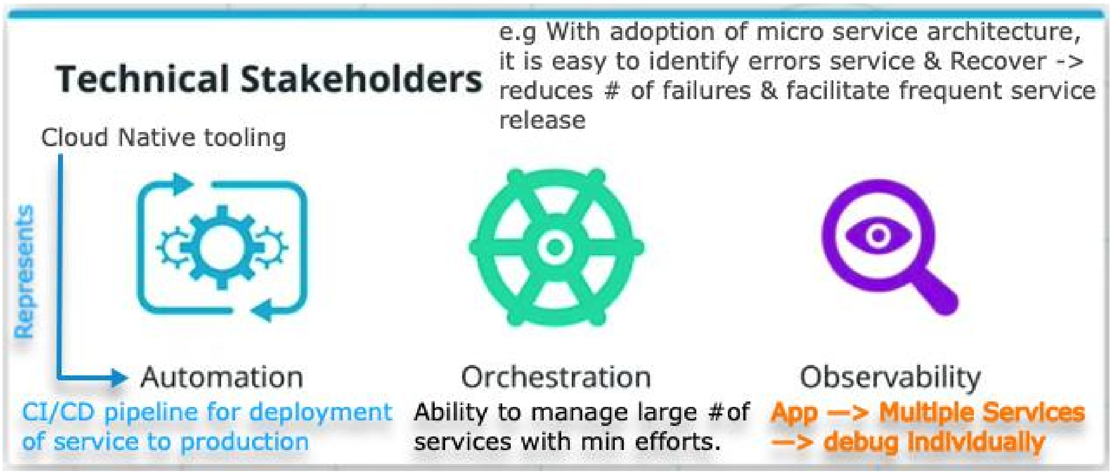
When adopting cloud native tooling and principles, we need to first consider the main stakeholders. In this section, we'll evaluate what are the key points that are assessed from a business and technical perspective before integrating an open source project. Cloud native tooling has a wide adoption in the industry lately.


Cloud Native Adoption
deliver
ability
Value to the customers in short amount of time
Easy to adjust and extend to accommodate new requirements

This is due to its ability to deliver value to customers in a short amount of time, and the ability to easily extend to accommodate new requirements. This represent the core reasons why an organization needs to trial and assess cloud native technologies.


Business Stakeholders
The adoption of cloud native tooling
represents
Agility
to perform strategic transformation of business velocity
Growth
quick iterations customer satisfaction++& growth
Service Availability
of product
7
24

From a business perspective, the adoption of cloud native tooling represents; agility, to perform strategic transformations for acceleration of business velocity, growth, which represents quick iterations that can increase customer satisfaction and lead to the growth of customer base, and service availability, which ensures the product is available to customers 24/7.


Technical Stakeholders
e.g With adoption of micro service architecture, it is easy to identify errors service & Recover -> reduces # of failures & facilitate frequent service release
Cloud Native tooling
Represents
Automation
CI/CD pipeline for deployment of service to production
Orchestration
Ability to manage large #of services with min efforts.
Observability
App —> Multiple Services —> debug Individually

For example, with adoption of micro service architecture, it is easier to identify an error service and recover it. This reduces the blast radius of a failure, and facilitates the frequent service release. From a technical perspective,

the adoption of cloud native tooling represents automation, which is strongly encouraged with the cloud native tooling. For example, constructing a pipeline to deploy a service to production without human intervention.
Orchestration, which refers to the ability to manage thousands of services with minimal effort. Usually, the introduction of a container orchestrator such as **Kubernetes** is fundamental, and observability that encompasses the segregation of an application to multiple services, and the ability to troubleshoot and debug each component independently.

Summary

An engineering team can use cloud-native tooling to enable quick delivery of **value to customers** and **easily extend** to new features and technologies. These are the main reasons why an organization needs to adopt cloud-native technologies. However, when trialing cloud-native tooling, there are two main perspectives to address: business and technical stakeholders.

From a **business perspective**, the adoption of cloud-native tooling represents:

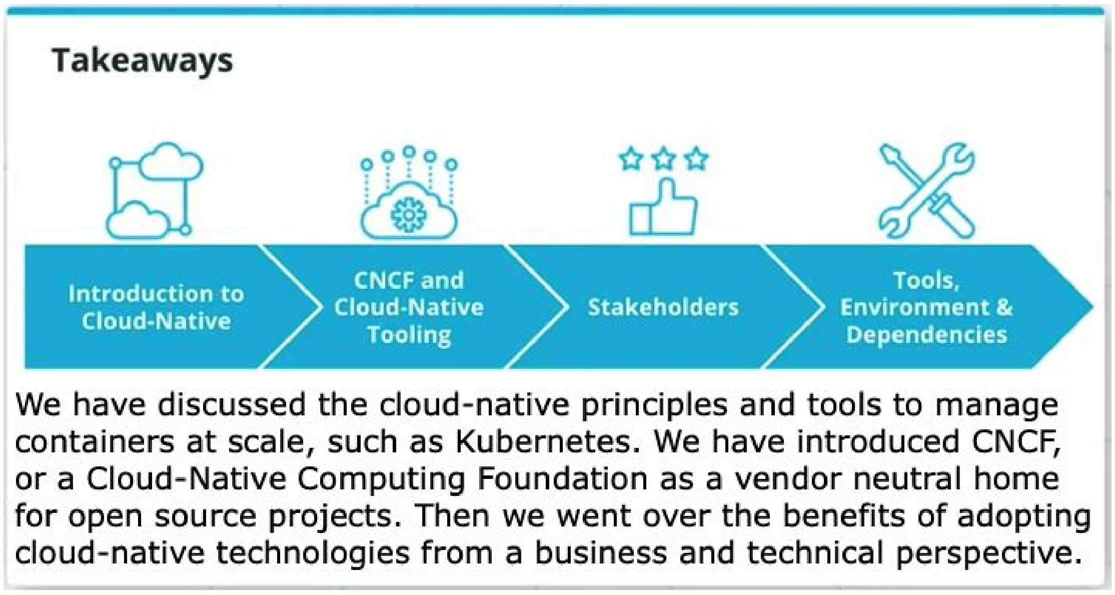
- Agility – perform strategic transformations
- Growth – quickly iterate on customer feedback
- Service availability – ensures the product is available to customers 24/7

From a **technical perspective**, the adoption of cloud-native tooling represents:

- Automation – release a service without human intervention
- Orchestration – introduce a container orchestrator to manage thousands of services with minimal effort
- Observability – ability to independently troubleshoot and debug each component

Recap

In this lesson, we have discussed the cloud-native principles and tools to manage containers at scale, such as **Kubernetes**.
We have introduced CNCF, or a Cloud-Native Computing Foundation as a vendor neutral home for open source projects.
Then we went over the benefits of adopting cloud-native technologies from a business and technical perspective.

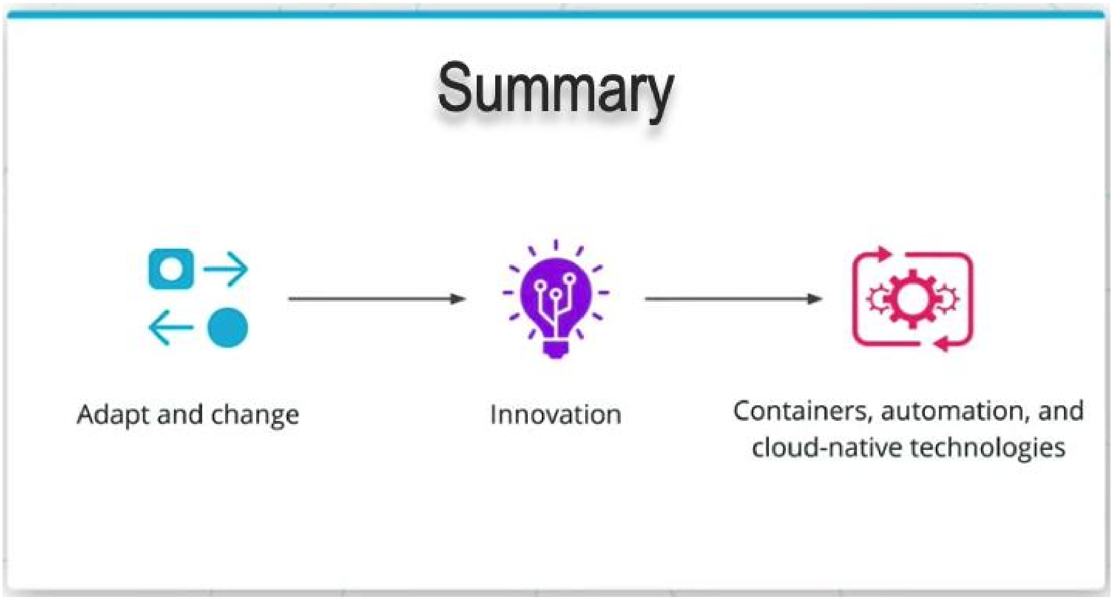

Takeaways

Introduction to Cloud-Native
CNCF and Cloud-Native Tooling
Stakeholders
Tools, Environment & Dependencies

We have discussed the cloud-native principles and tools to manage containers at scale, such as Kubernetes. We have introduced CNCF, or a Cloud-Native Computing Foundation as a vendor neutral home for open source projects. Then we went over the benefits of adopting cloud-native technologies from a business and technical perspective.

Summary

Every organization aims to succeed! This is represented by providing customer value and the ability to be responsive to the surrounding ecosystem. Coincidentally, this is closely correlated with technological innovation, which translates into the adoption of containers, automation, and usage of cloud–native tooling.
By completing this course you will be equipped to lead the adoption of cloud–native tooling and principles within an organization.



Summary

Adapt and change
Innovation
Containers, automation, and cloud-native technologies

Lesson 2:

Architecture Consideration for Cloud Native Applications

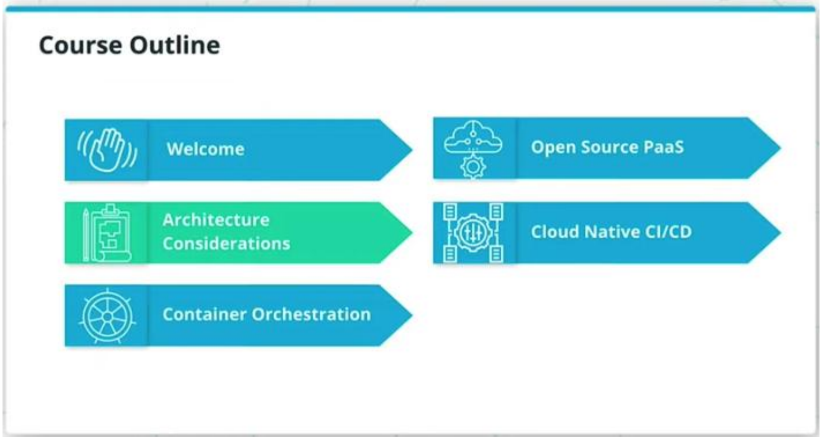
1. INTRODUCTION

Welcome to the Architecture Consideration lesson.

Before building an application, it is important to have a list of requirements which will define the structure and design of the application. It's very common for developers to spend a lot of time and effort building an application only to find out that it has fundamental flows when it is accessed by hundreds of customers, or migration process needs to be kicked off.

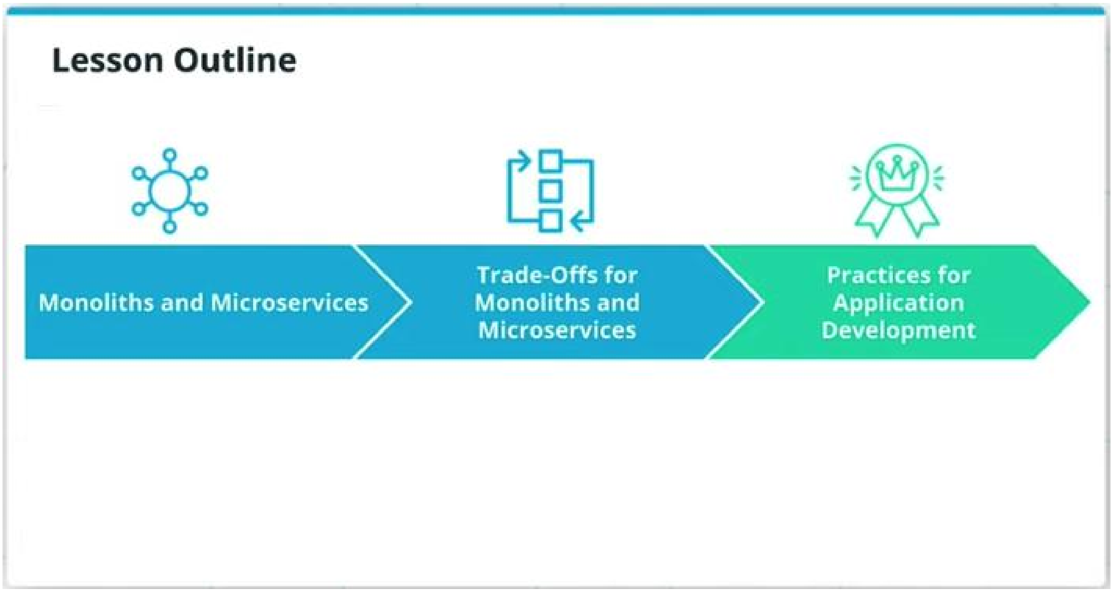
Before building an application, it is important to define **the structure and design of the project and understand the implied trade-offs.**

This lesson covers design patterns such as monoliths and microservices, as shown below in figure including trade-offs to examine with each model. In addition, we will go through best practices to adopt at the implementation stage, to optimize the visibility, management, and troubleshooting of an application.



In this lesson, you'll first learn how to structure a project using monolithic and microservice based architectures. Also, we will examine how requirements and available resources can define which architecture is most suitable for your project. Then, we will learn about the benefits and drawbacks of each architecture and reflect on how these trade–offs can impact the design of an application and its road map. Finally, you will learn and apply some of the best practices during the application development.

These are relevant for both microservice and monolith architectures, as the focus on optimization of visibility, management and troubleshooting of an application.
By the end of this lesson, you should be able to evaluate the most suitable architecture for an application, considering functional requirements, available resources, and time frame.



Lesson Outline

Monoliths and Microservices
Trade-Offs for Monoliths and Microservices
Practices for Application Development

Summary

Welcome to the Cloud Native Fundamentals course!

Before building an application, it is common to go through a design phase to identify the main requirements and structure of an application. In correlation with the available resources, a team will choose the most suitable project architecture.

In the industry, usually, the two main approaches that are usually referenced are monoliths and microservices. In this lesson, we will explore each architectural model and the implied trade-offs. As well, we will cover good development practices to be considered if an application is targeted for containerization. These practices are valid for both monolith and microservice architectures.

Monoliths and Microservices
Trade-Offs for Monoliths and Microservices
Practices for Application Development

Architecture Considerations lesson outline

What has been covered so far:

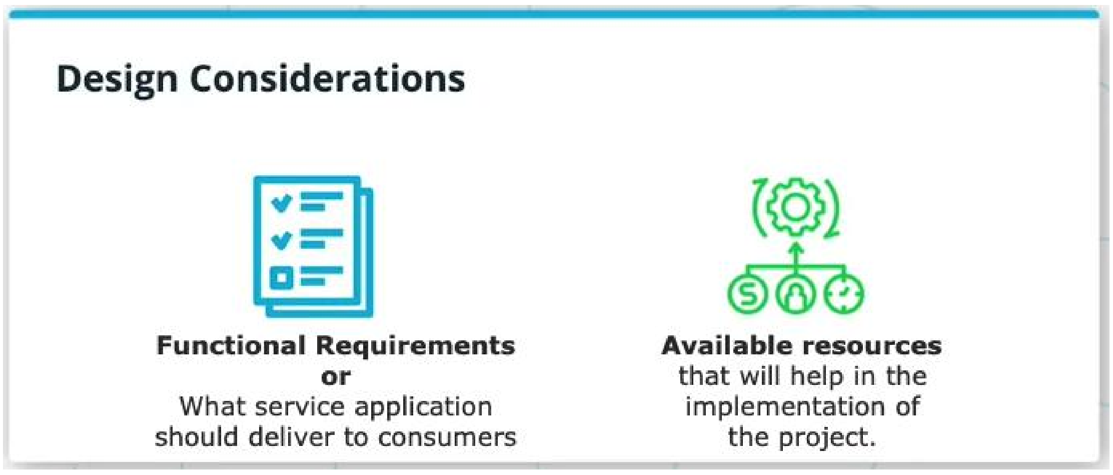
- Monoliths and Microservices
- Trade-offs for Monoliths and Microservices
- Practices for Application Development

2. DESIGN CONSIDERATIONS FOR CLOUD–NATIVE APPLICATIONS

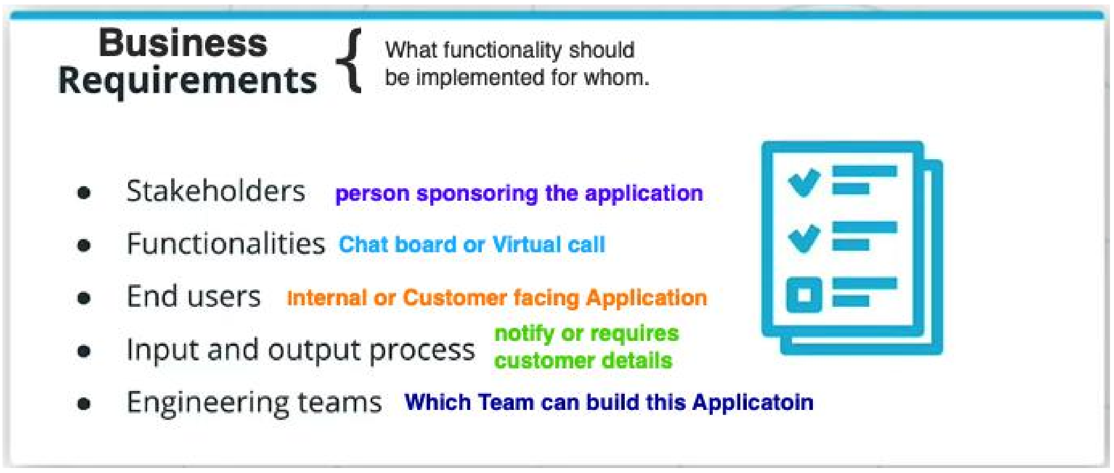
Transcript:

There are a few things to consider when designing a cloud–native application. When deciding to create a product, it is paramount to allocate time for context discovery that help us define the overall structure and design of the components.

This practice enables the build out of maintainable projects where new features can be added with minimal engineering effort. The first step in the contexts discovery process is to list the functional requirements or what service the application should deliver to consumers, the second step is to list all of the available resources that will help with the implementation of the project, these two data points will contrary the application architecture, making it easier to choose between monoliths and microservices–based architecture.



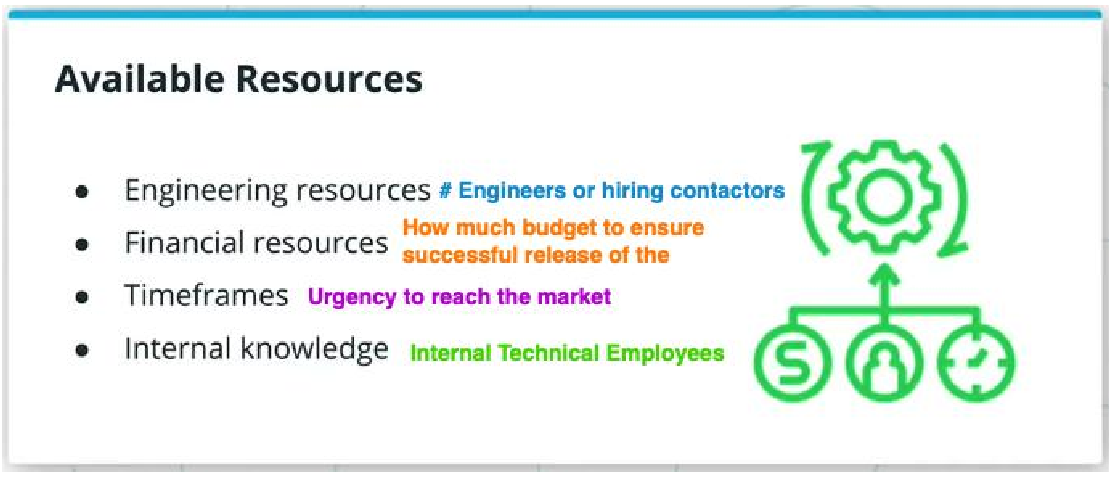
For Contest discovering, it is paramount to gather and understand the **business requirements**, what functionality should be implemented and for whom, for example, a good starting point is to answer questions like, who are the stakeholders? Such as identifying the personas that require and sponsor this application, for example, the marketing team requiring a new tool to customize notifications for customers, what functionalities are needed? For example, should these provide a chat board or a virtual call functionality? Who are the end consumers or customers? For example, is it an internal tool for employees, or is it a customer–facing application? How inputs and outputs should be processed, such as, should the application send a notification as part of the output, or does it require any customer detail to execute successfully? Or what engineering team can build the application or understanding which teams has the skills and time frame to implement the project?



Business Requirements { What functionality should be implemented for whom.

- Stakeholders person sponsoring the application
- Functionalities Chat board or Virtual call
- End users Internal or Customer facing Application
- Input and output process notify or requires customer details
- Engineering teams Which Team can build this Applicatoin

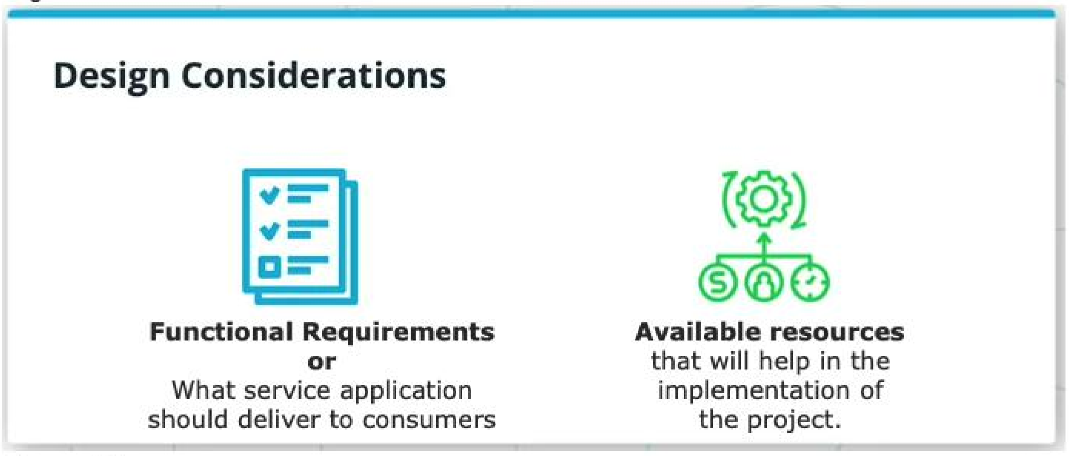
It is equally important to define the available resources for an organization or what can facilitate or block the product release, for example, a good starting point is to list the available engineering resources, such as the number of engineers that can work on the project, or ability to hire contractors. Financial resources, for example, how much can the business spend to ensure a successful release of the product. Time frames or define if there is any urgency to reach the market, an internal knowledge of a programming language or a tool which can facilitate the development of an application.



Available Resources

- Engineering resources # Engineers or hiring contactors
- Financial resources How much budget to ensure successful release of the
- Timeframes Urgency to reach the market
- Internal knowledge Internal Technical Employees

Figure 1.0

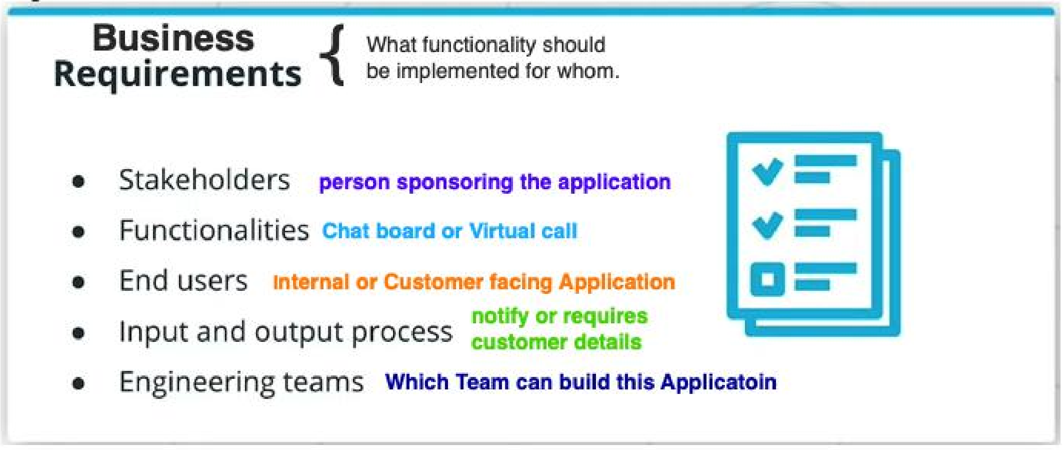


Design Considerations

Functional Requirements
or
What service application
should deliver to consumers

Available resources
that will help in the
implementation of
the project.

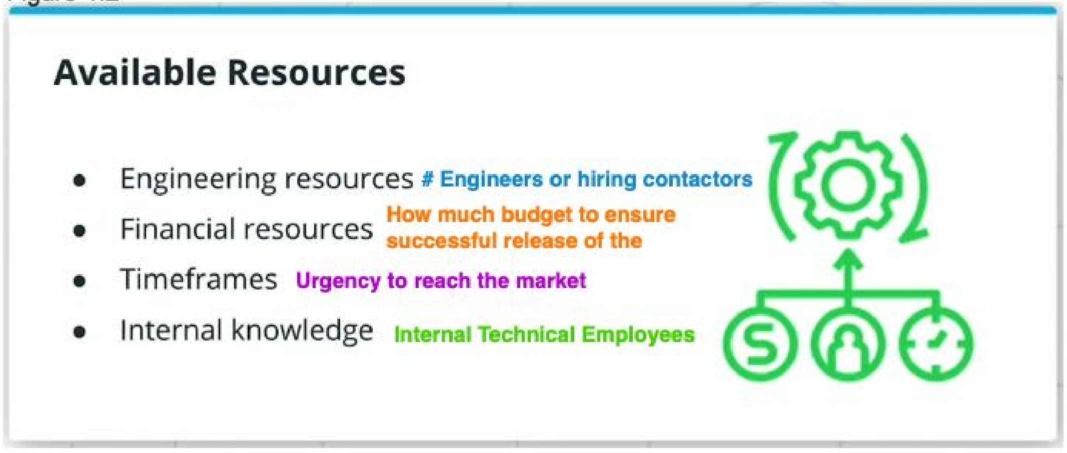
Figure 1.1



Business Requirements { What functionality should be implemented for whom.

- Stakeholders **person sponsoring the application**
- Functionalities **Chat board or Virtual call**
- End users **Internal or Customer facing Application**
- Input and output process **notify or requires customer details**
- Engineering teams **Which Team can build this Applicatoin**

Figure 1.2



Available Resources

- Engineering resources **# Engineers or hiring contactors**
- Financial resources **How much budget to ensure successful release of the**
- Timeframes **Urgency to reach the market**
- Internal knowledge **Internal Technical Employees**

Summary

When building an application it is important to allocate time for context discovery. This includes listing all necessary functionalities of the application and enumerating any resources that can enable its buildout. This phase sets the fundamentals of the project. If properly implemented, it can enable the creation of services that are scalable, resilient, and extensible.

The first step in the context discovery process is to list the functional requirements, or what application capabilities should deliver to the end-users. For example, a good starting point is to expand on the following:

- ➢ Stakeholders
- ➢ Functionalities
- ➢ End users
- ➢ Input and output process
- ➢ Engineering teams

The second step is to enumerate the available resources that facilitates the implementation of the project. For example, a good starting point is to list available:

- ➢ Engineering resources
- ➢ Financial resources
- ➢ Timeframes
- ➢ Internal knowledge

Having a good understanding of functional requirements and available resources can lead to a simpler choice between monolithic and microservice-based architectures.

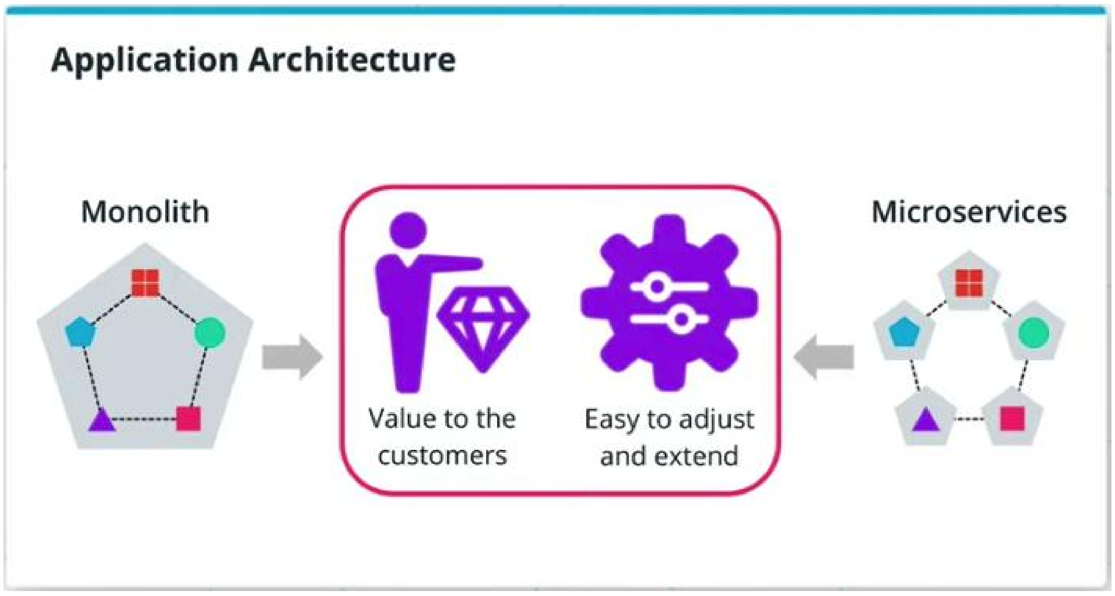
3. MONOLITHS AND MICROSERVICES

Transcript:

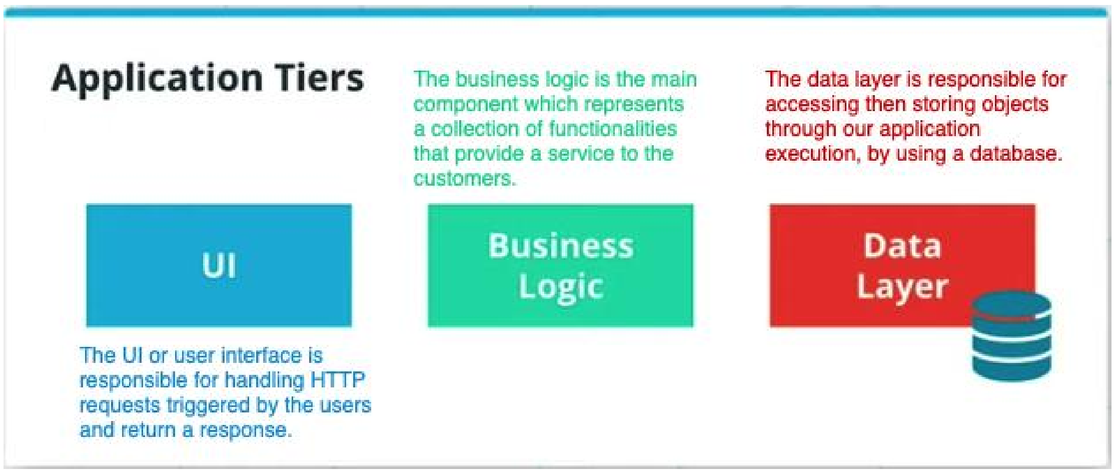
Once we have collected and analyzed the application requirements and available resources, we can move to the design phase. This is the step where you need to choose the most suitable Application Architecture. In this section, we'll cover how a simple booking application can be designed using both Monoliths and Microservice based models.

When an engineering team decides which architecture should be chosen for an application, usually two distinct models are referenced, monoliths and microservices.

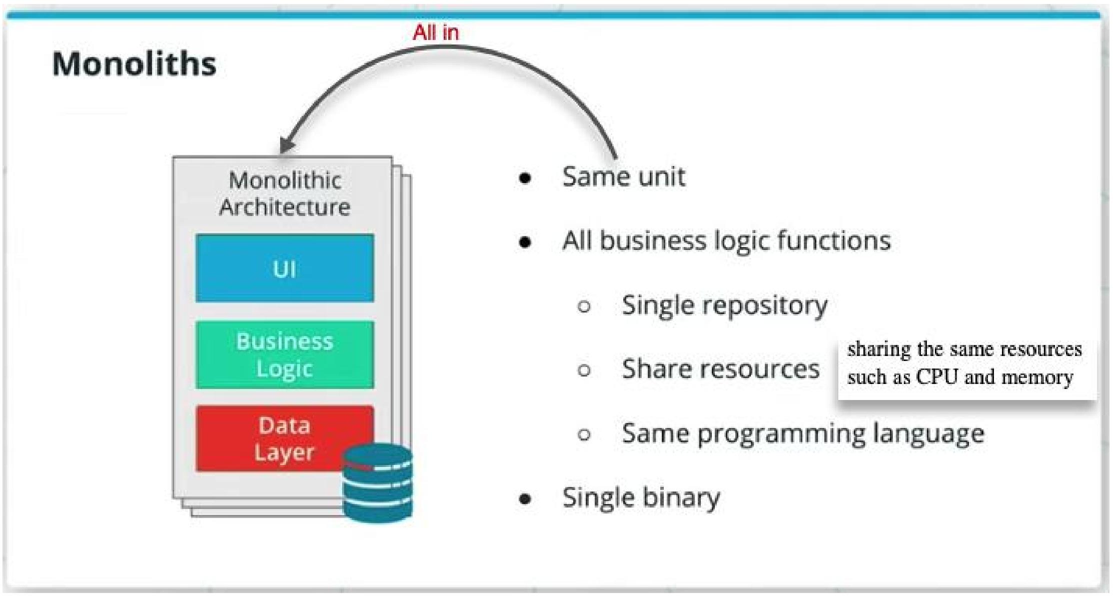
Regardless of the adopted structure, the end goal is to have a well–designed application that can provide value to customer for a satisfactory user experience, and at the same time allows the engineering team to easily adjust and extend the existing functionalities.



Let's look how an application is going to be structured while using a monolith or microservice design.
At a macro level, an application is composed of three main tiers, UI, Business Logic, and Data Layer.
The UI or user interface is responsible for handling HTTP requests triggered by the users and return a response.
The business logic is the main component which represents a collection of functionalities that provide a service to the customers.
The data layer is responsible for accessing then storing objects through our application execution, by using a database.

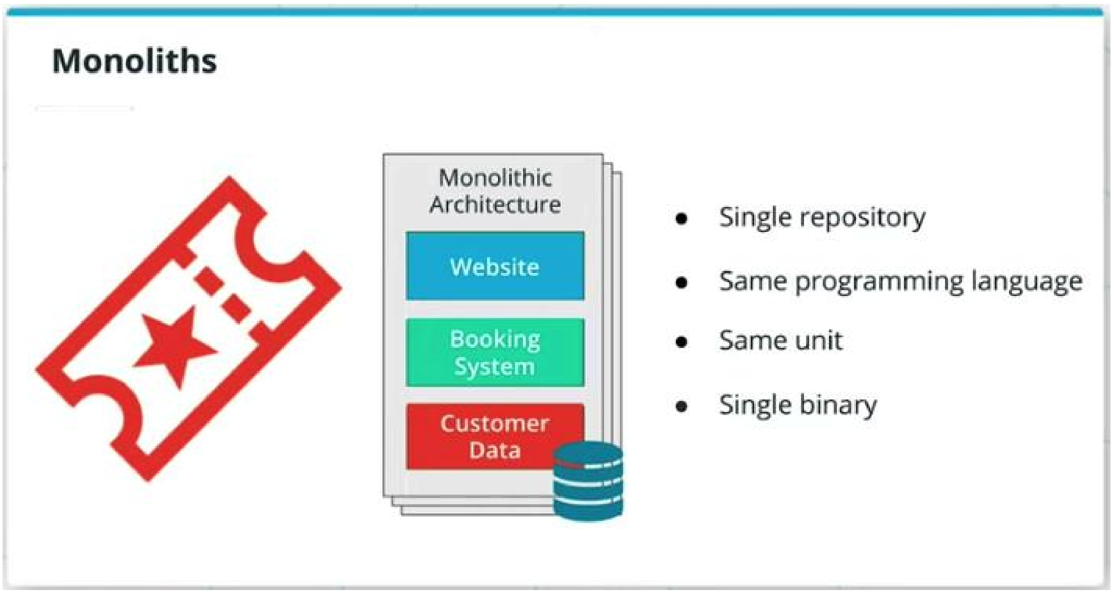

Application Tiers
The business logic is the main component which represents a collection of functionalities that provide a service to the customers.
The data layer is responsible for accessing then storing objects through our application execution, by using a database.
UI
Business Logic
Data Layer
The UI or user interface is responsible for handling HTTP requests triggered by the users and return a response.

In a monolithic approach, all of this components are part of the same unit. In this case, all business logic functions are developed and managed in a single repository, sharing the same resources such as CPU and memory, and revolving around one programming language.



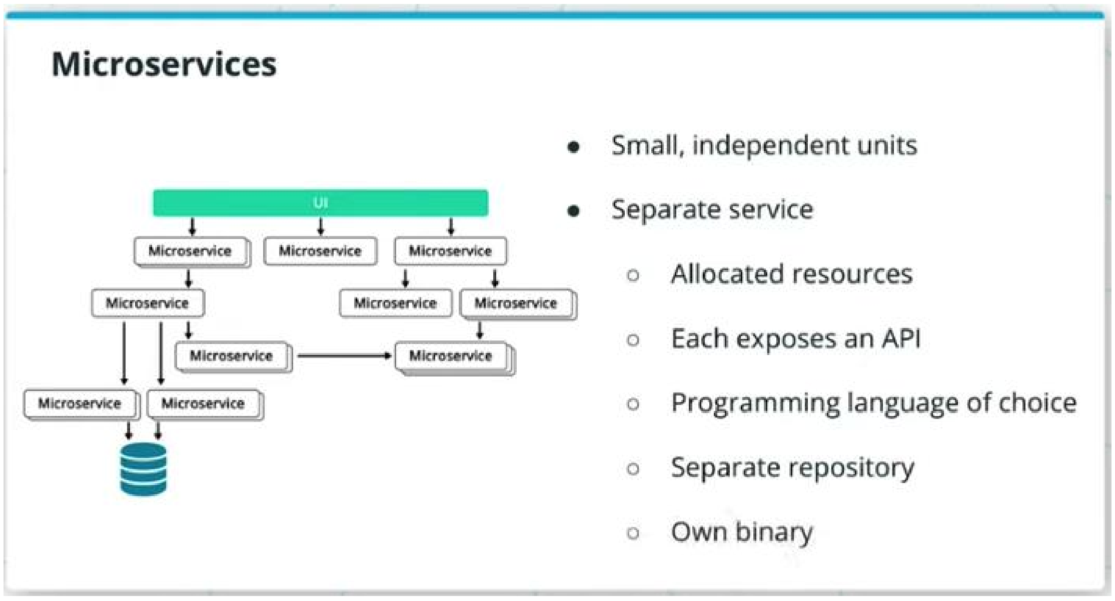
- Same unit
- All business logic functions
  - Single repository
  - Share resources
  - Same programming language
- Single binary

The packaging, distribution and deployment of a monolith application are represented by a single artifact or binary, which will include the code for the entire stack. Imagine we are developing a booking system application that allows customers to buy tickets online. In this example, the UI is the website, which will be the main point of interaction with the user. The business logic contains the code that implements the booking functionalities, such as searching, booking, payment, and so on. This functions are managed together in a single repository using the same programming language, for example, Java or Go, and the last layer is the data layer containing functions that store and retrieve customer data. With the monolith architecture, all of these tiers are managed as a unit and packaging, distribution and release of this application are represented by single binary.


Monoliths

- Single repository
- Same programming language
- Same unit
- Single binary

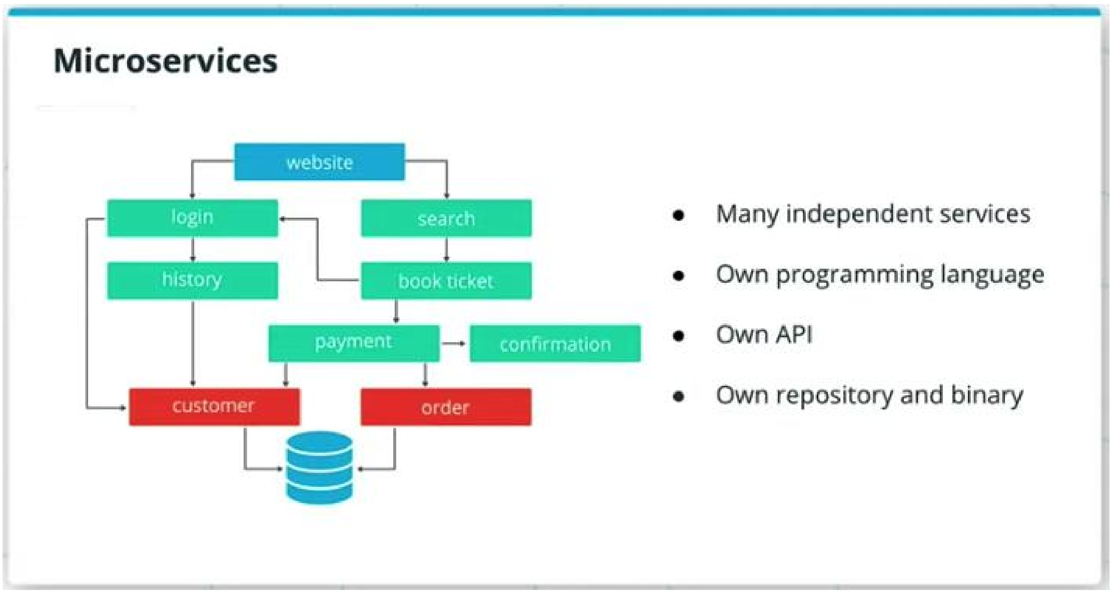
On the other side, the microservices approach aims to breakdown the application into smaller, independent units. Each functionality represents a separate service containing its logic and its own allocated resources for CPU and memory. In addition to these, each service exposes an API or application programming interface for interaction and communication with other available units.
Every unit is written in the programming language of choice, which is the best suited for context of implementation. This enables concurrent development cycles as multiple teams can work on building multiple services at the same time. When it comes to the packaging, distribution, and deployment, each service is composed of a separate repository with its own binary, that contains the code and the dependencies for that unit alone.



Microservices

- Small, independent units
- Separate service
  - Allocated resources
  - Each exposes an API
  - Programming language of choice
  - Separate repository
  - Own binary

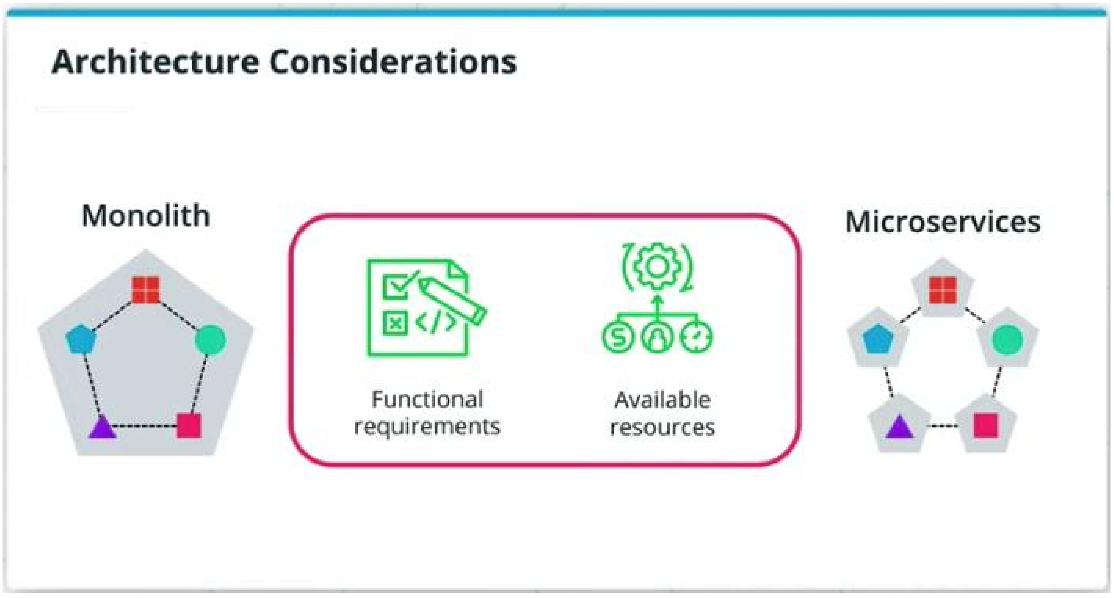
Now, let's build an online booking website using the microservices approach. In this case, they remains the website, which will be the main point of interaction with the user. However, the business logic is decomposed into many independent units. One for each service, for example, logging, payment, confirmation, orders, and many more. Each service is written in its own programming language. As such one team can use Go language for the payment service. While at the same time, another team can use Python to build the login capabilities. This enable us multiple teams to work concurrently on different functionalities. It is not worthy that every unit has its own well-defined API, which is used for interaction and communication with other available units, and lastly, the data layer contains functions that store and retrieve customer and order data. As expected, each service has its own repository and it will be deployed using its own binary.



Microservices

- Many independent services
- Own programming language
- Own API
- Own repository and binary

Choosing an architecture for an application is highly impacted by the functional requirements such as product features, stake holders or dependencies and available resources such as engineering teams, time frame, finances and internal knowledge of a programming language or tool.



Architecture Considerations

If the organization has unlimited resources and powerful engineering force, then a micro service approach can be easily adopted. However, if a team is considering to build a new application, but only has few engineers available and a tight time frame, then a monolithic structure will help them hit the ground quicker.

Summary

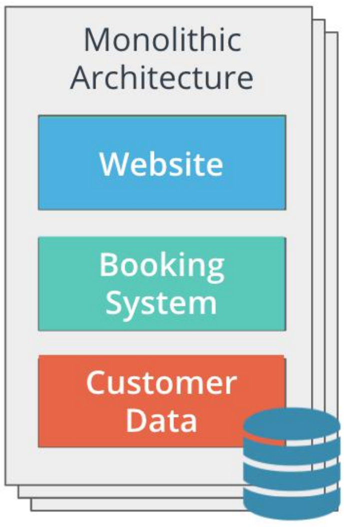
Prior to an organization delivers a product, the engineering team needs to decide on the most suitable application architecture. In most of the cases 2 distinct models are referenced: monoliths and microservices. Regardless of the adopted structure, the main goal is to design an application that delivers value to customers and can be easily adjusted to accommodate new functionalities.

Also, each architecture encapsulates the 3 main tires of an application:

- UI (User Interface) – handles HTTP requests from the users and returns a response
- Business logic – contained the code that provides a service to the users
- Data layer – implements access and storage of data objects
- Monoliths

In a monolithic architecture, application tiers can be described as:

- part of the same unit
- managed in a single repository
- sharing existing resources (e.g. CPU and memory)
- developed in one programming language
- released using a single binary

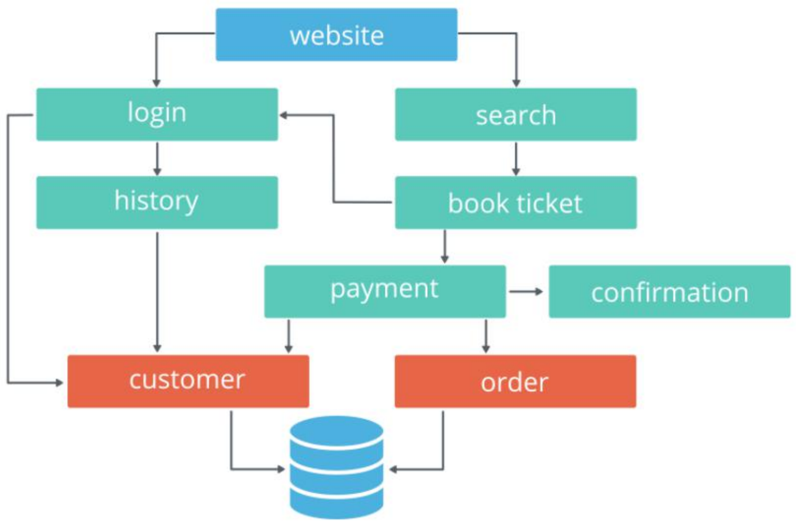


A booking application referencing the monolithic architecture

Imagine a team develops a booking application using a monolithic approach. In this case, the UI is the website that the user interacts with. The business logic contains the code that provides the booking functionalities, such as search, booking, payment, and so on. These are written using one programming language (e.g. Java or Go) and stored in a single repository. The data layer contains functions that store and retrieve customer data. All of these components are managed as a unit, and the release is done using a single binary.

In a microservice architecture, application tiers are managed independently, as different units. Each unit has the following characteristics:

- managed in a separate repository
- own allocated resources (e.g. CPU and memory)
- well-defined API (Application Programming Interface) for connection to other units
- implemented using the programming language of choice
- released using its own binary



A booking application referencing the microservice architecture

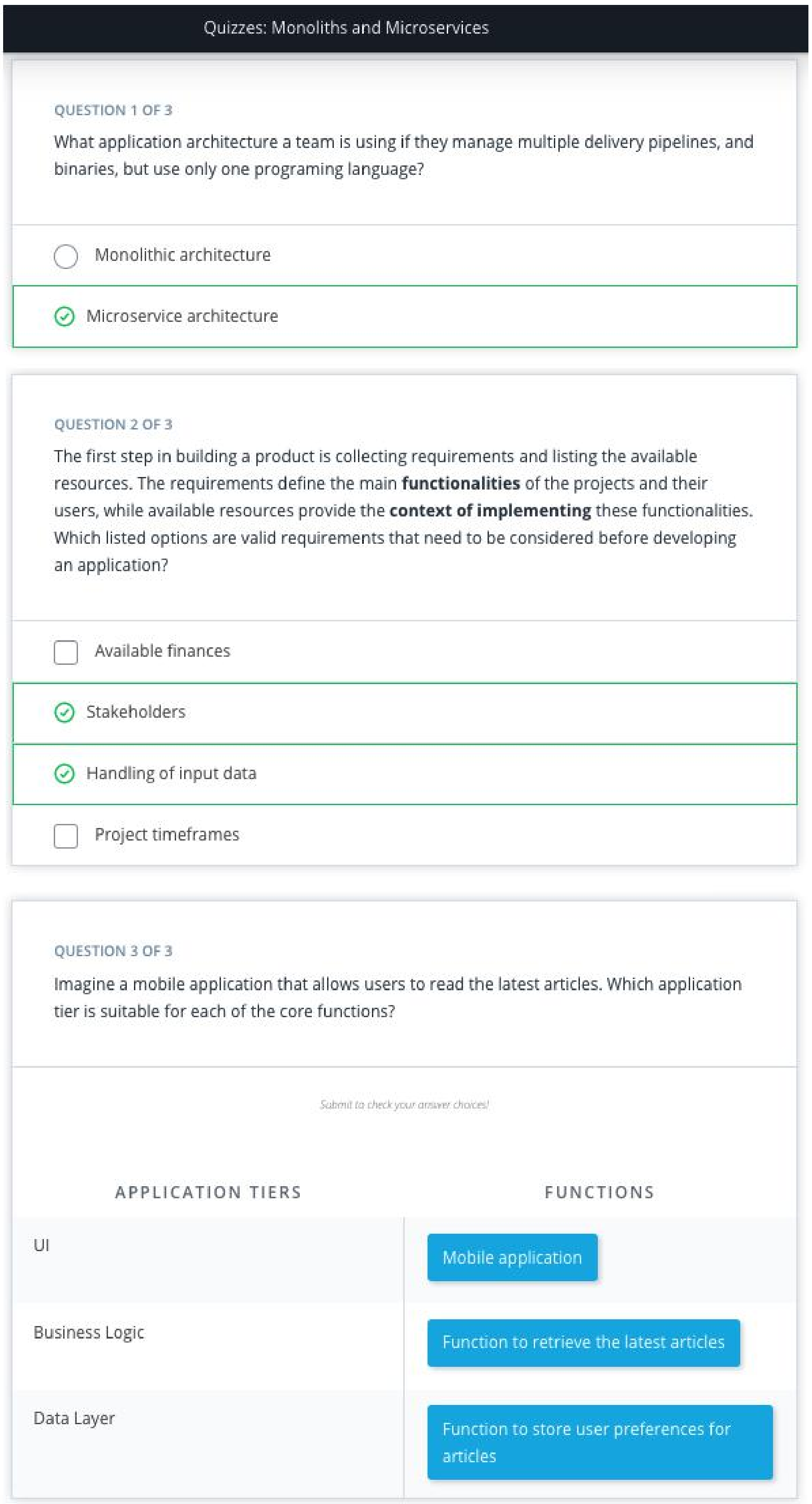
Now, let's imagine the team develops a booking application using a microservice approach.

In this case, the UI remains the website that the user interacts with. However, the business logic is split into smaller, independent units, such as login, payment, confirmation, and many more. These units are stored in separate repositories and are written using the programming language of choice (e.g. Go for the payment service and Python for login service). To interact with other services, each unit exposes an API. And lastly, the data layer contains functions that store and retrieve customer and order data. As expected, each unit is released using its own binary.

New terms

- Monolith: application design where all application tiers are managed as a single unit
- Microservice: application design where application tiers are managed as independent, smaller units

4. Quizzes: Monoliths and Microservices



Quizzes: Monoliths and Microservices

QUESTION 1 OF 3

What application architecture a team is using if they manage multiple delivery pipelines, and binaries, but use only one programing language?

- ○ Monolithic architecture
- [✓ Microservice architecture]

QUESTION 2 OF 3

The first step in building a product is collecting requirements and listing the available resources. The requirements define the main **functionalities** of the projects and their users, while available resources provide the **context of implementing** these functionalities. Which listed options are valid requirements that need to be considered before developing an application?

- ☐ Available finances
- [✓ Stakeholders]
- [✓ Handling of input data]
- ☐ Project timeframes

QUESTION 3 OF 3

Imagine a mobile application that allows users to read the latest articles. Which application tier is suitable for each of the core functions?

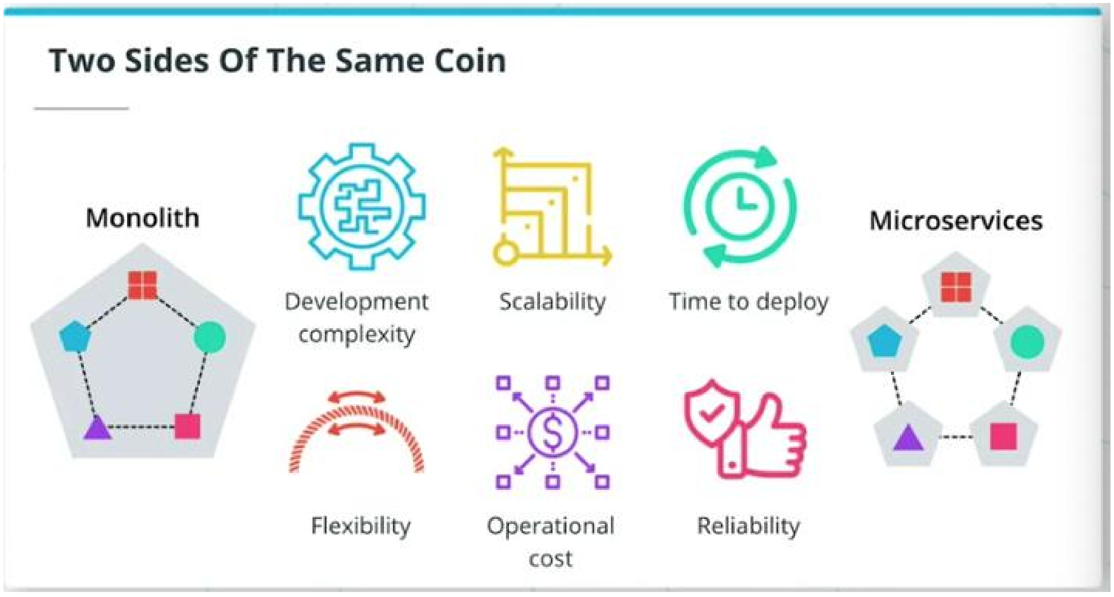
Submit to check your answer choices!

APPLICATION TIERS	FUNCTIONS
UI	Mobile application
Business Logic	Function to retrieve the latest articles
Data Layer	Function to store user preferences for articles

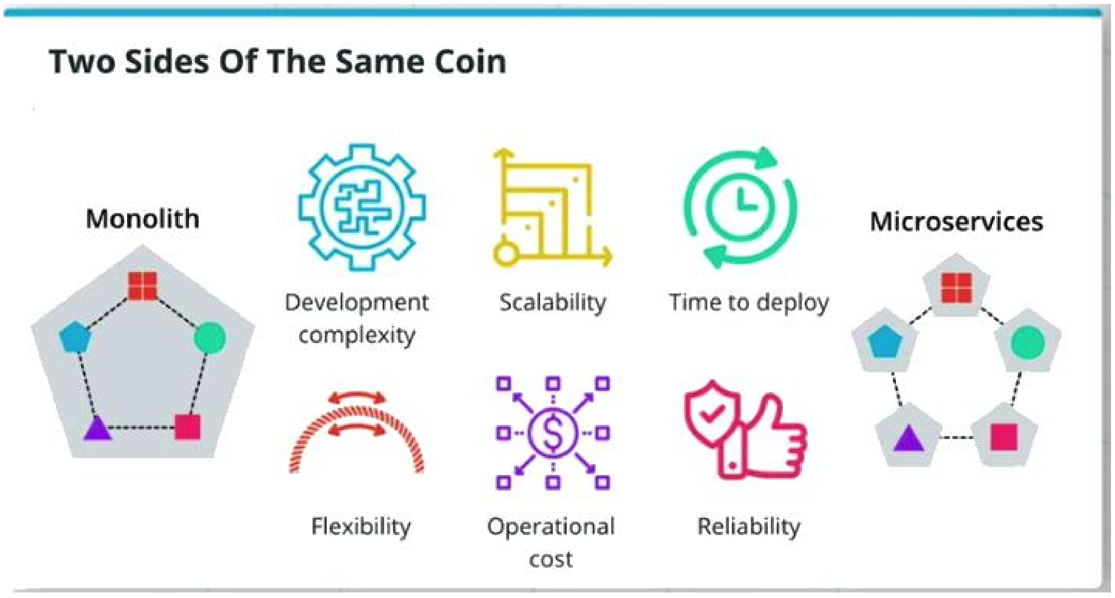
5. Trade-offs for Monoliths and Microservices Architectures

Transcript: Up to this stage, we have explored how an application can adopt a monolithic or microservice-based architecture. However, each approach has a set of trade-offs that needs to be considered thoroughly as this will impact the longevity of the project. There is no right or wrong structure for an application, there are just **different requirements** and **available resources**.

Let's look at each trade-off in a bit more detail. So far, we have learned about two main architectural patterns, monoliths and microservices. The choosing a model is heavily defined by the functional requirements and available resources. However, this represents just one side of the coin.



While deciding on the structure of a project, it is equally important to understand the trade-offs implied by each architecture, such as development complexity, scalability, time to deploy, flexibility, operational cost, and reliability. Let's look at this in more detail.



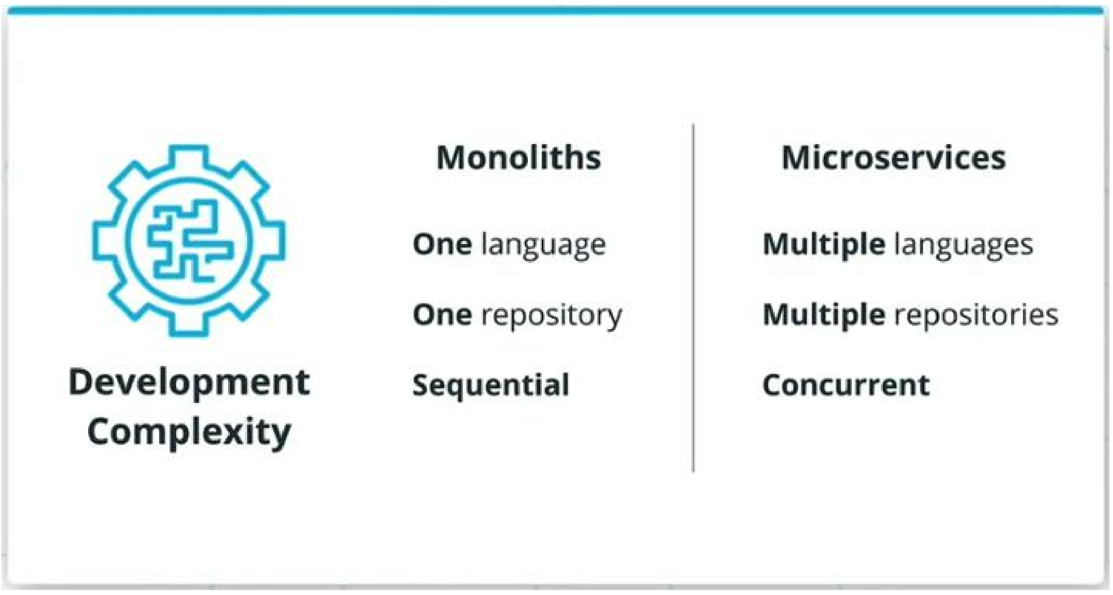
Transcript:

Development complexity addresses the effort required to deploy and manage an application.

Let's compare the development complexity for monoliths and microservice architectures.

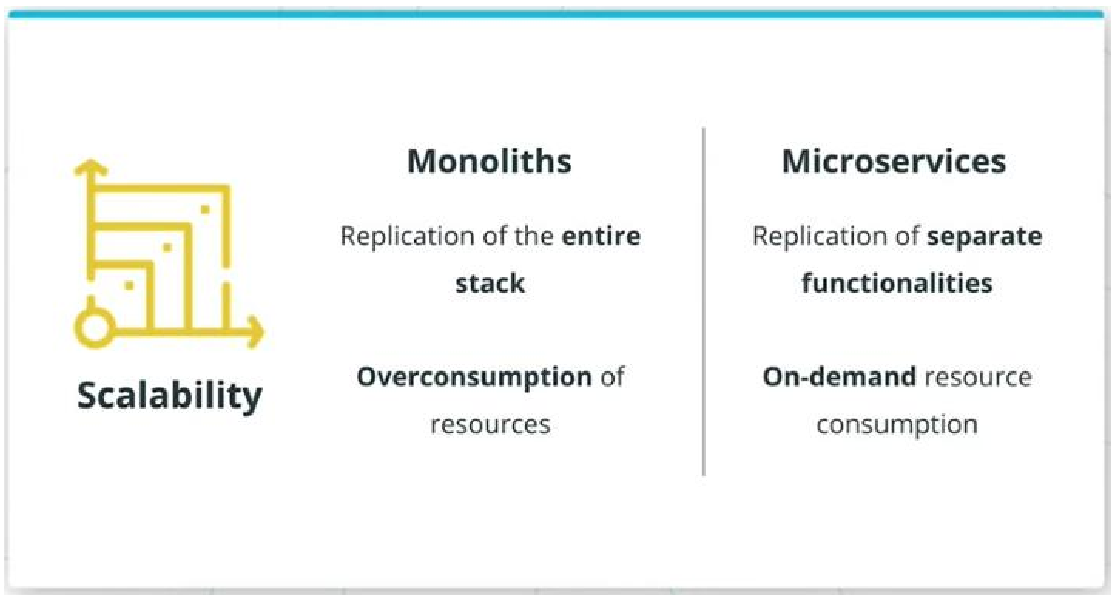
Starting with a programming language, the monolith usually revolves around a single framework or language. On the other side, microservices can be deployed using multiple languages.

Similarly, when it comes to the management of the code repositories, the monolith can be stored in one repository while each microservice require its own separate codebase.
While if the monolith seems to be on the brighter side, it is worth noting that the development complexity scale is radically when more functionalities are added to the project.
Additionally, each model enables an engineering pattern. With microservices, each team can work on a functionality dependently in a concurrent manner.
In a monolithic approach, the development is sequential, usually a release requiring changes to multiple functionalities to ensure backwards compatibility.



	Monoliths	Microservices
Development Complexity	**One** language	**Multiple** languages
	One repository	**Multiple** repositories
	Sequential	**Concurrent**

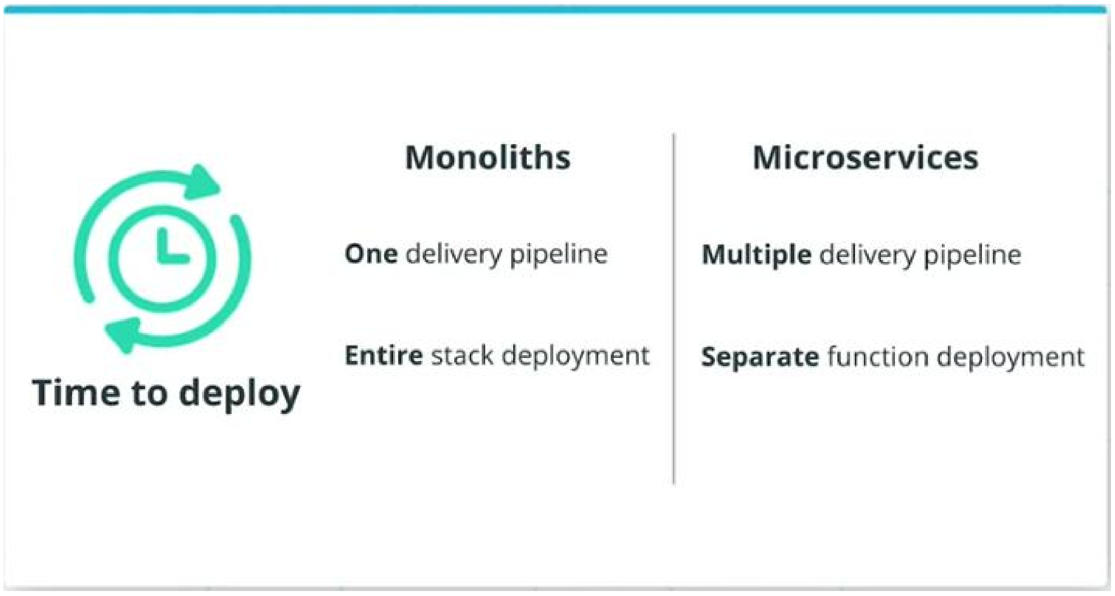
Scalability captures how an application scale is under load. For example, observing the application behavior when incoming request are increasing exponentially, or there is a sudden demand in one particular service. The scalability of a monolithic application implies the replication of the entire unit with all the functionalities. As such, in the case of the Web Store, when the payment service needs to be scaled, we would also have to scale the customer login, order shipping, and AVRA components. This is heavy on the resource consumption such as CPU and memory, as we are using more space than actually needed.
With a microservices architecture, we can identify clearly which unit require scaling and we can replicate that component alone. Consequently, we have a better usage of the platform resources as we only use the memory and CPU that is actually needed.



Scalability	Monoliths	Microservices
	Replication of the **entire stack**	Replication of **separate functionalities**
	Overconsumption of resources	**On-demand** resource consumption

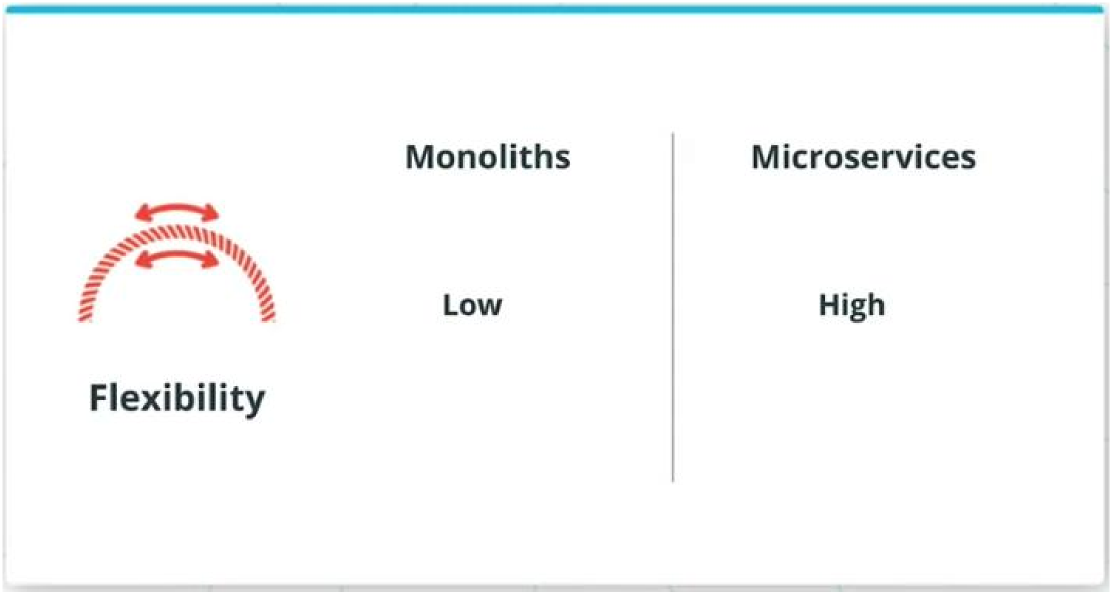
Time to deploy refers to the ability to build a delivery pipeline and ship features. As expected with a monolithic approach, only one delivery pipeline is required as all components are developed in the same code repository. On the other side, the microservices will need a delivery pipeline per service which usually coincides with one pipeline per code base. When developing new features in a monolithic architecture, the entire unit or application will be deployed. This can be disastrous if a release fails as it will take down the entire application.
There is a higher risk of violating the zero downtime principle, which aims that application should be available to consumers 24x7.
This however, is not the case with microservices,

as each functionality can be deployed independently without affecting the availability of our components. As such, there is less risk to take down the entire application within your release.
Consequently, microservices allow an increase velocity of future development as we can have more releases with flies risk while with monolith this is at a lower rate as ever release implies the redeploy of the entire stack.



	Monoliths	Microservices
Time to deploy	**One** delivery pipeline	**Multiple** delivery pipeline
	Entire stack deployment	**Separate** function deployment

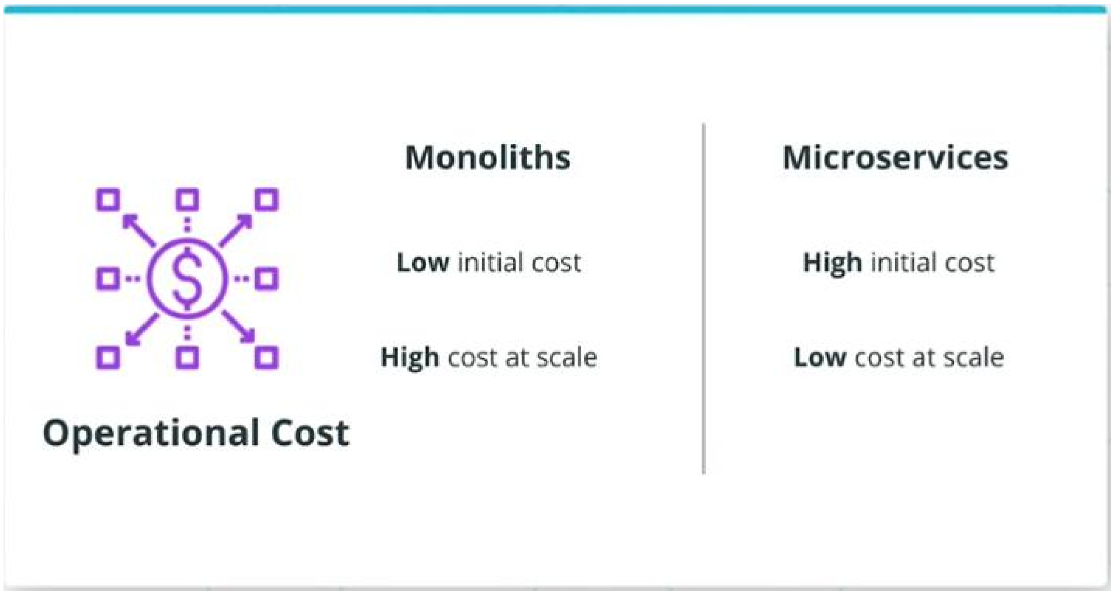
Flexibility implies the ability to incorporate new technologies and adapt to new principles and tooling. There are moments when it's better to use a different programming language for service or modify your application for a specific platform. This operations would represent a big beg moment with monolith as the entire project would have to be rewritten or restructured. At the same time, microservices are purposely built to be loosely coupled and allows independent changes to services. Your writing, or redesigning one functionality is more achievable than redesigning the entire stack.



	Monoliths	Microservices
Flexibility	Low	High

Operational coast encapsulates the necessary resources to build, deploy and release a product. A low initial cost to spin up an application is certainly inciting.
This is the case with a monolithic architecture where only one code base is needed and there's only one delivery pipeline. On the other side, the microservices require the maintenance of multiple code bases and delivery pipelines and the dependencies for programming languages that comes with it.

However, the showcase is only the genesis of a product development. The picture is flipped when new functionalities are added and the application needs to be scaled to cope with high customer demand. Maintaining a monolith or time imposes more complexity and it will consume more resources when replicated. While with microservices, that operational cost is directly proportional to the required resources at the time.
Scalability is perform individually per unit and adding new components is a defect to operation.

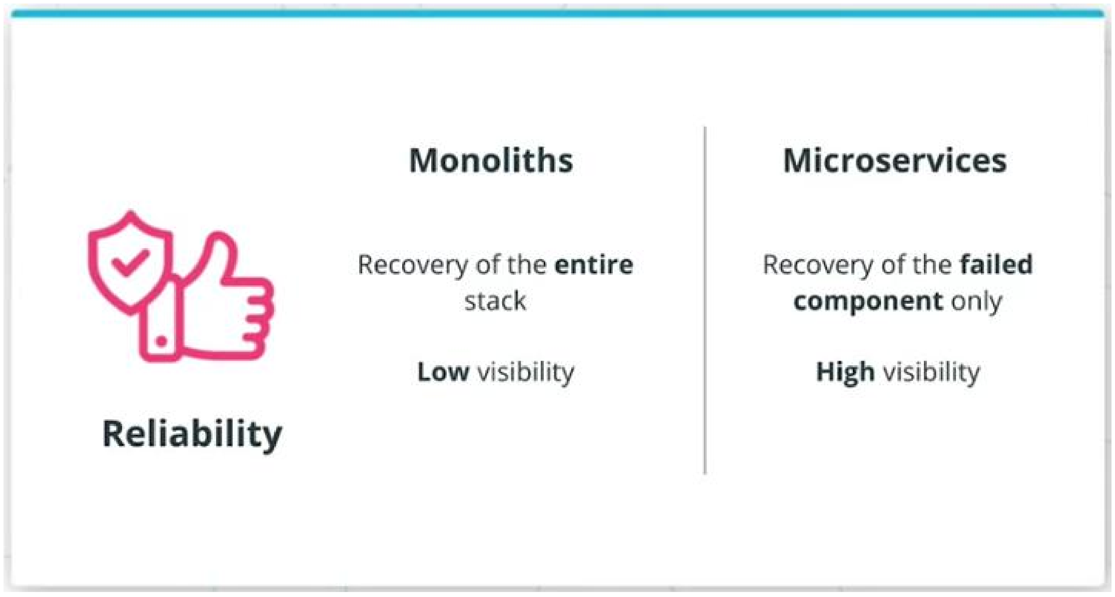


Operational Cost

Monoliths	Microservices
Low initial cost	**High** initial cost
High cost at scale	**Low** cost at scale

Reliability refers to the ability to recover from failure and waste monitoring application. Microservices are represented by a distributed amount of functions that

interact with each other via the network. If a component fails then that component alone will need recovery while if a monolith is in a failed state then the entire stack will need to be troubleshooted and recovered.
Additionally, with microservices, it is possible to have representative metrics and logs of a separate unit, while with monoliths getting granular visibility for each functionality is difficult as all of the metrics and logs for the entire application will be aggregated together.



	Monoliths	Microservices
Reliability	Recovery of the **entire** stack **Low** visibility	Recovery of the **failed component** only **High** visibility

As you can observe, monoliths and microservices have trade-offs to be considered when you start a project and maintain it. However, it is important to truly understand the application requirements when choosing an architecture as this will define the best approach.

There is no golden path to design a product,
but a good understanding of the trade-offs will provide a clear road map that enforces extensibility and optimization. Regardless if monolith or a microservices architecture is chosen, as long as the project is coupled with an efficient delivery pipeline, the ability to adopt new technologies and easily add features, the path to Cloud-native development is certain.

Summary

Development Complexity

Development complexity represents the effort required to deploy and manage an application.

- **Monoliths** - one programming language; one repository; enables sequential development
- **Microservice** - multiple programming languages; multiple repositories; enables concurrent development

Scalability

Scalability captures how an application is able to scales up and down, based on the incoming traffic.

- Monoliths - replication of the entire stack; hence it's heavy on resource consumption
- Microservice - replication of a single unit, providing on-demand consumption of resources

Time to Deploy

Time to deploy encapsulates the build of a delivery pipeline that is used to ship features.

- **Monoliths** – one delivery pipeline that deploys the entire stack; more risk with each deployment leading to a lower velocity rate
- **Microservice** – multiple delivery pipelines that deploy separate units; less risk with each deployment leading to a higher feature development rate

Flexibility

Flexibility implies the ability to adapt to new technologies and introduce new functionalities.

- **Monoliths** – low rate, since the entire application stack might need restructuring to incorporate new functionalities
- **Microservice** – high rate, since changing an independent unit is straightforward

Operational Cost

Operational cost represents the cost of necessary resources to release a product.

- **Monoliths** – low initial cost, since one code base and one pipeline should be managed. However, the cost increases exponentially when the application needs to operate at scale.
- **Microservice** – high initial cost, since multiple repositories and pipelines require management. However, at scale, the

cost remains proportional to the consumed resources at that point in time.

Reliability

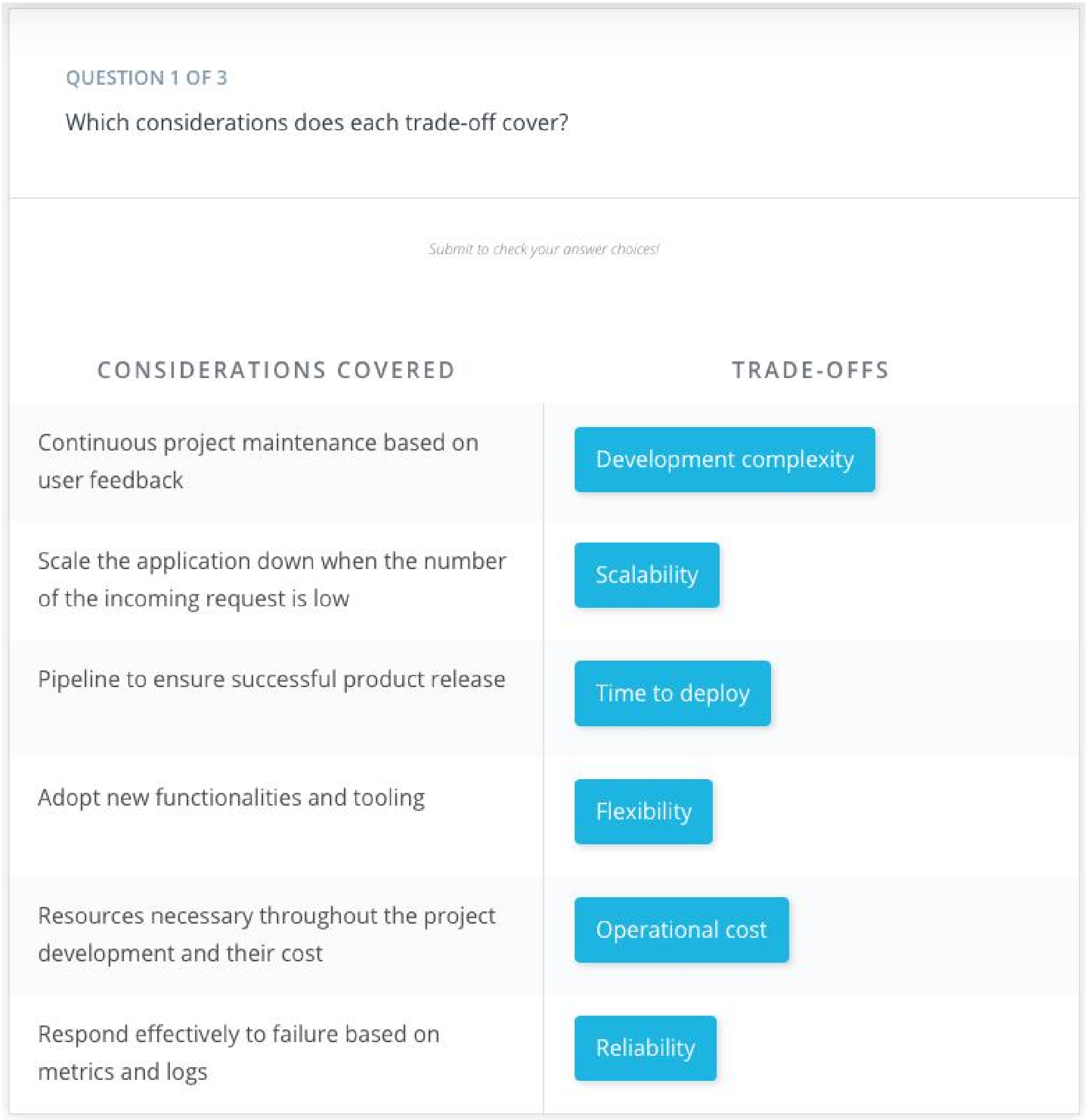
Reliability captures practices for an application to recover from failure and tools to monitor an application.

- Monoliths – in a failure scenario, the entire stack needs to be recovered. Also, the visibility into each functionality is low, since all the logs and metrics are aggregated together.
- Microservice – in a failure scenario, only the failed unit needs to be recovered. Also, there is high visibility into the logs and metrics for each unit.

Each application architecture has a set of trade-offs that need to be considered at the genesis of a project. But more importantly, it is paramount to understand how the application will be maintained in the future e.g. at scale, under load, supporting multiple releases a day, etc.

There is no "golden path" to design a product, but a good understanding of the trade-offs will provide a clear project road-map. Regardless if a monolith or microservice architecture is chosen, as long as the project is coupled with an efficient delivery pipeline, the ability to adopt new technologies, and easily add features, the path to could-native deployment is certain.

6.0 Quizzes: Trade-offs for Monoliths and Microservices

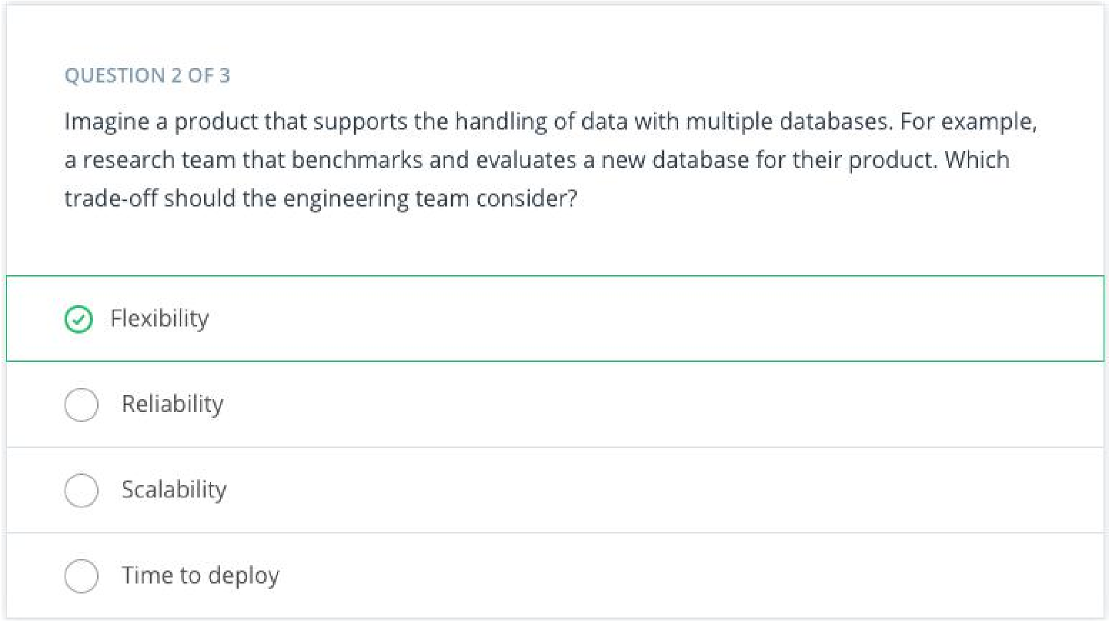


QUESTION 1 OF 3

Which considerations does each trade-off cover?

Submit to check your answer choices!

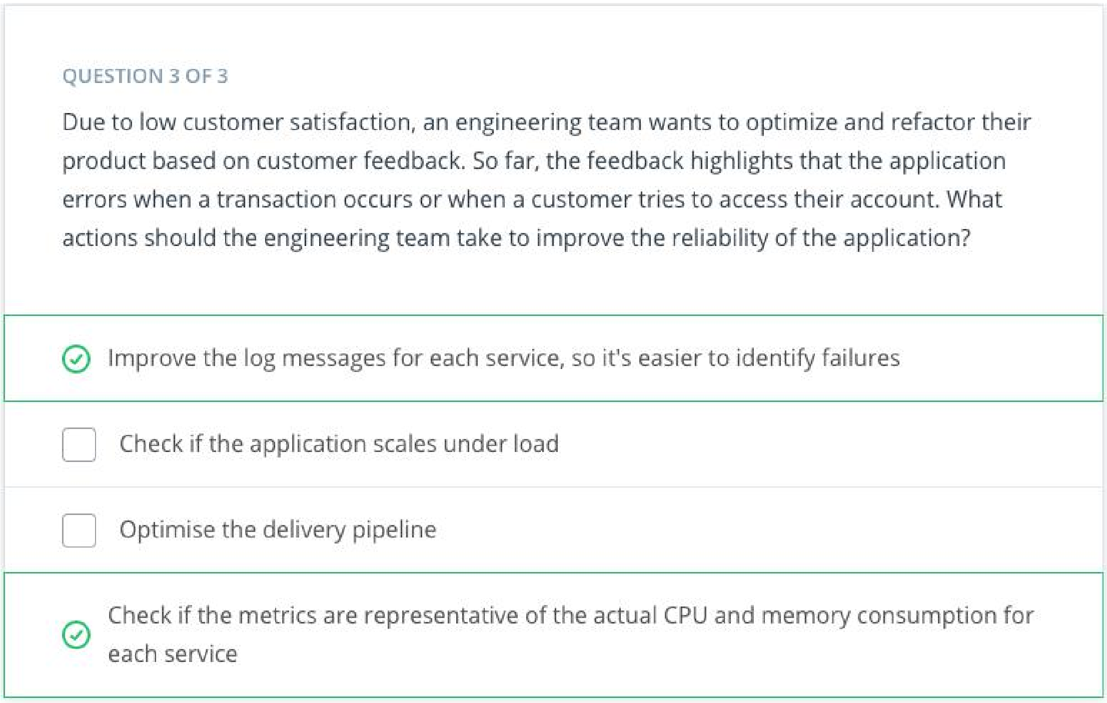
CONSIDERATIONS COVERED	TRADE-OFFS
Continuous project maintenance based on user feedback	Development complexity
Scale the application down when the number of the incoming request is low	Scalability
Pipeline to ensure successful product release	Time to deploy
Adopt new functionalities and tooling	Flexibility
Resources necessary throughout the project development and their cost	Operational cost
Respond effectively to failure based on metrics and logs	Reliability



QUESTION 2 OF 3

Imagine a product that supports the handling of data with multiple databases. For example, a research team that benchmarks and evaluates a new database for their product. Which trade-off should the engineering team consider?

- [✓ Flexibility]
- Reliability
- Scalability
- Time to deploy



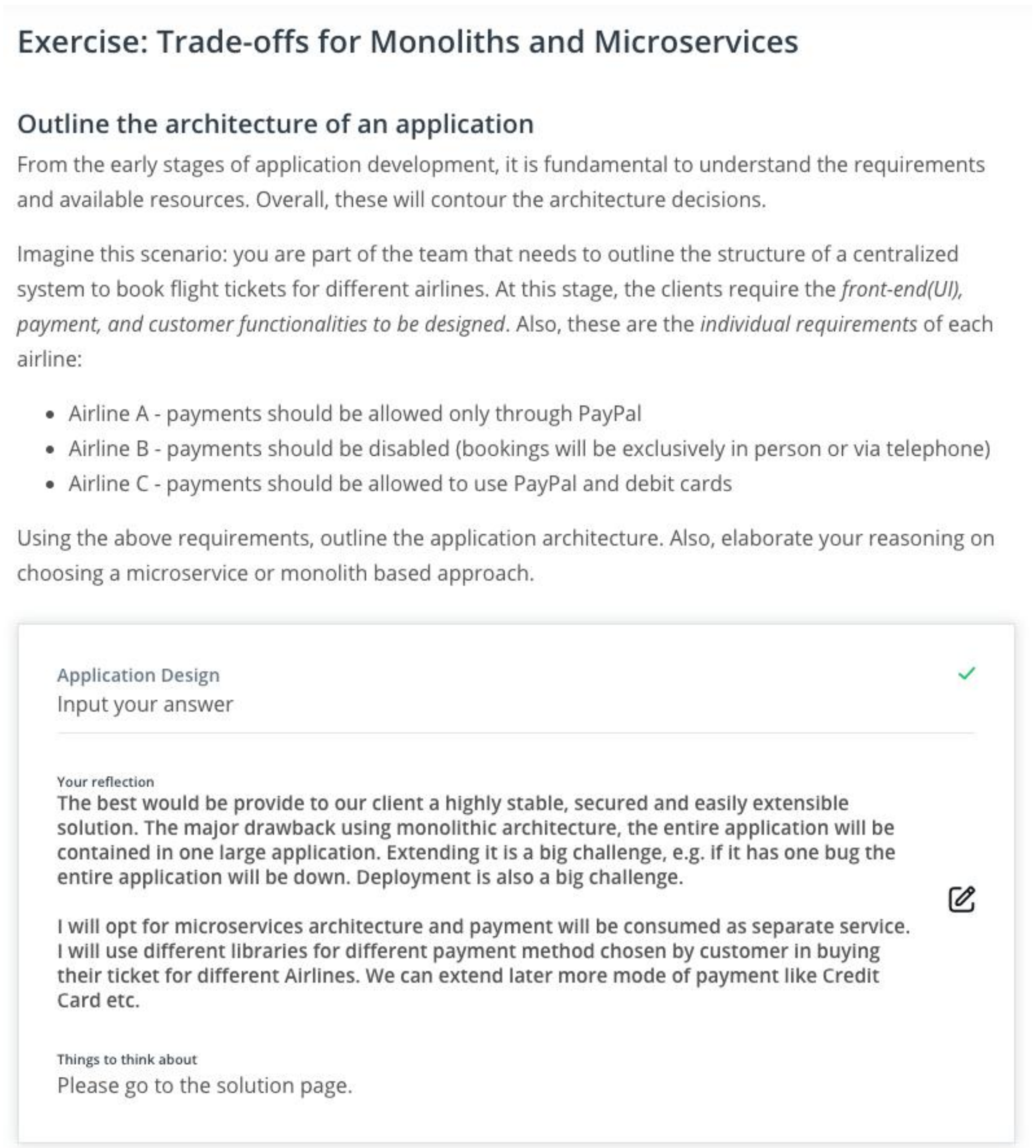
QUESTION 3 OF 3

Due to low customer satisfaction, an engineering team wants to optimize and refactor their product based on customer feedback. So far, the feedback highlights that the application errors when a transaction occurs or when a customer tries to access their account. What actions should the engineering team take to improve the reliability of the application?

- ✓ Improve the log messages for each service, so it's easier to identify failures
- ☐ Check if the application scales under load
- ☐ Optimise the delivery pipeline
- ✓ Check if the metrics are representative of the actual CPU and memory consumption for each service

Reliability refers to the ability to respond quickly to failure, by having insightful logs and metrics

7. Exercise



Exercise: Trade-offs for Monoliths and Microservices

Outline the architecture of an application

From the early stages of application development, it is fundamental to understand the requirements and available resources. Overall, these will contour the architecture decisions.

Imagine this scenario: you are part of the team that needs to outline the structure of a centralized system to book flight tickets for different airlines. At this stage, the clients require the *front-end(UI), payment, and customer functionalities to be designed*. Also, these are the *individual requirements* of each airline:

- Airline A - payments should be allowed only through PayPal
- Airline B - payments should be disabled (bookings will be exclusively in person or via telephone)
- Airline C - payments should be allowed to use PayPal and debit cards

Using the above requirements, outline the application architecture. Also, elaborate your reasoning on choosing a microservice or monolith based approach.

Application Design ✓
Input your answer

Your reflection
The best would be provide to our client a highly stable, secured and easily extensible solution. The major drawback using monolithic architecture, the entire application will be contained in one large application. Extending it is a big challenge, e.g. if it has one bug the entire application will be down. Deployment is also a big challenge.

I will opt for microservices architecture and payment will be consumed as separate service. I will use different libraries for different payment method chosen by customer in buying their ticket for different Airlines. We can extend later more mode of payment like Credit Card etc.

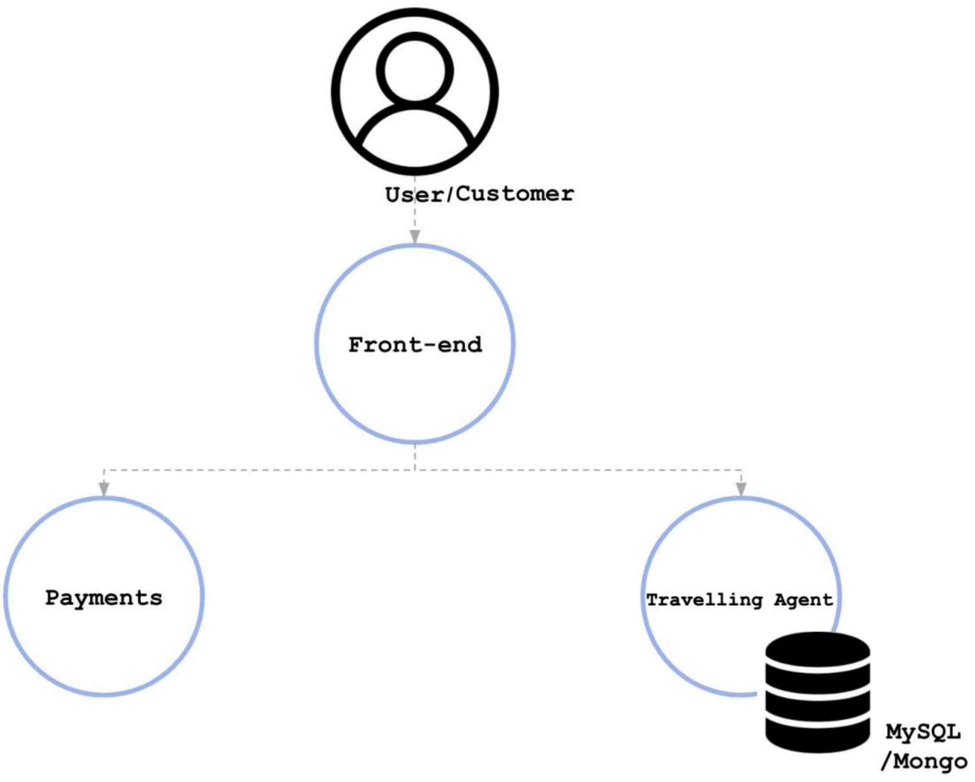
Things to think about
Please go to the solution page.

8. Solution: Monoliths and Microservices

Summary

Given the scenario, it is paramount to choose an architecture that would be replicable and scalable. For example, if thousands of customers access the payment service in the same time frame, then this particular service should be scaled up. In a **monolith architecture,** scaling up creates a replica of everything, including front-end and customer services, in addition to the payment service. This will also consume more resources on the platform, such as CPU and memory, and takes longer to spin up. On the other side, **a microservice** is a lightweight component that requires fewer resources (CPU and memory) and less time for provisioning. For this example, a microservice-based architecture is chosen, based on considerations that the application is a central booking system for multiple airlines, that implies a high load. The main components are:

- Front-end – entry-point for the user, where they will choose their airline or choice
- Customer – requires a database (MySQL or Mongo) to store the customer details
- Payments – to implement PayPal and Debit based operations

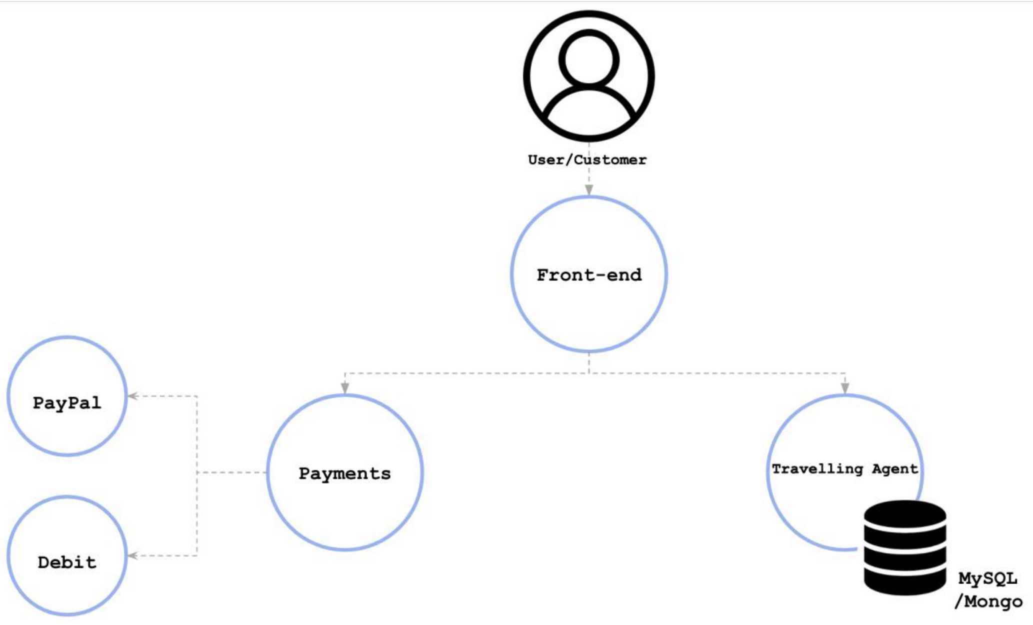


Flight booking application using microservice architecture

Additionally, the "payments" microservice is capable of handling multiple payment systems. Interaction with the PayPal interface and management of debit card APIs are fundamentally different. The "payments" component is a monolith that can be divided into multiple parts.

Payments:

- PayPal – handling all the PayPal payments
- Debit – handling all the debit card payments



So far, we went through requirements collection and structure design of an application.

9 Best Practices For Application Deployment

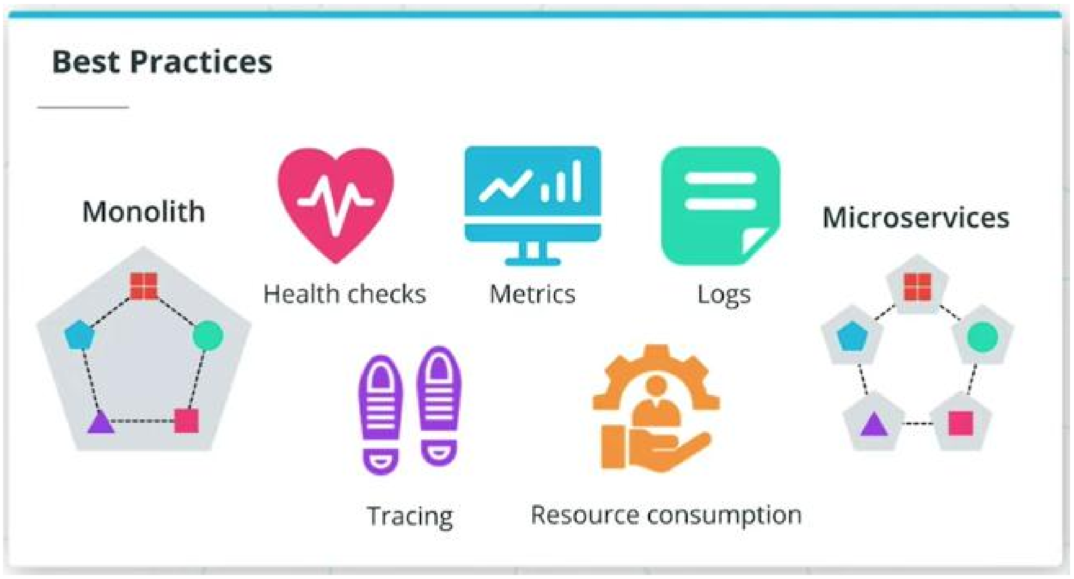
By now, you should be comfortable to choose the most suitable architecture for your project based on the requirements and involved trade offs.

The next step is the implementation phase.

In this section, we will present some of the best practices to use when developing an application that aims to be containerized.

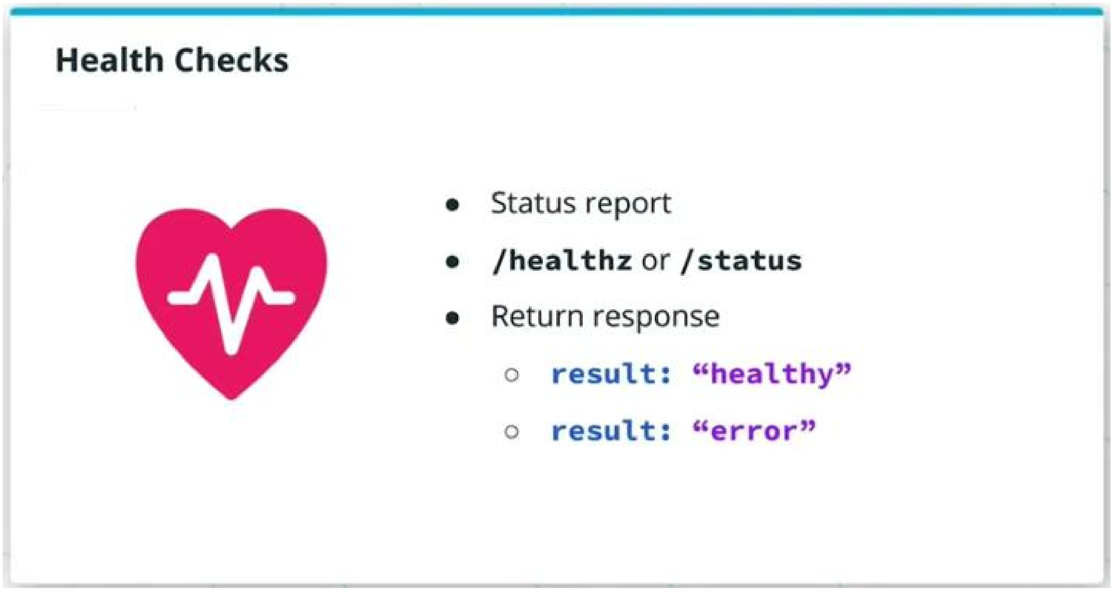
Best Practices For Application Deployment

These practices will focus on increasing visibility and improving resource consumption for a service. Whilst the requirements that contran [Computer programming language in which instructions are written at the compiler level, thereby eliminating the need for translation by a compiling routine or **Control Translator**] an application might differ, with both microservices and monoliths, it is possible to apply the same practices at the implementation stage. These practices are focused on **health checks**, **metrics**, **logs, tracing, and resource consumption**. The purpose of these practices is to increase visibility and improve resource consumption for a service.



In the following sections, we will take a look at these practices in more detail. You'll have a chance to apply and explore some of these practices in the coming exercises. Also, I'll provide a solution to these exercises, making sure you have a point of reference that you can use for your final project.

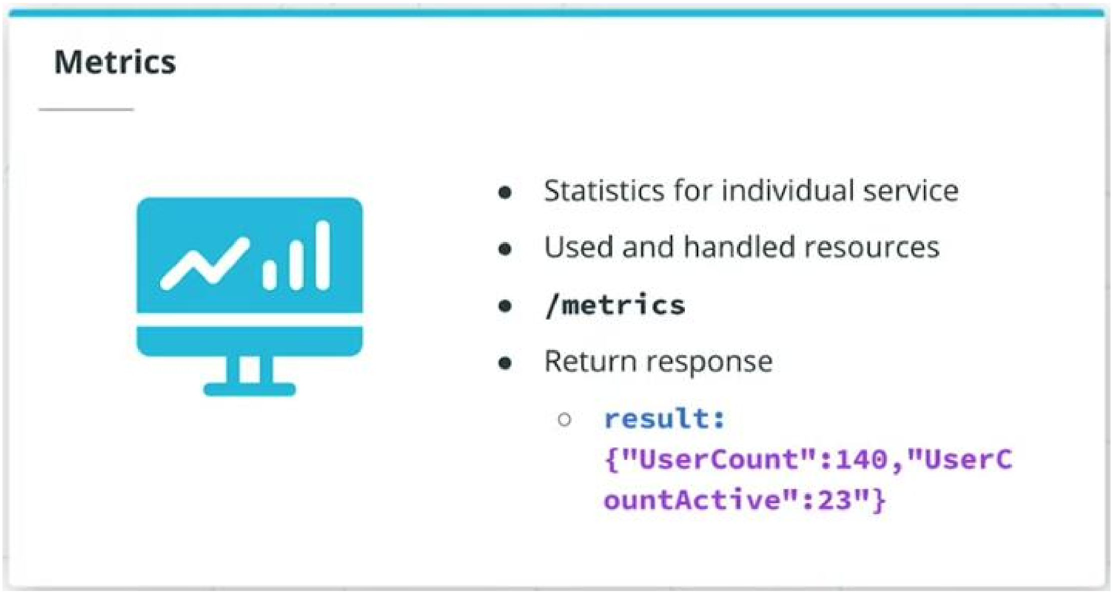
Health checks are the fundamental signal that showcases the status of an application. It usually is represented by an HTTP endpoint such as forward **/healthz or /status.** Once the logic is implemented to check the status of a component, the response will be returned via the design HTTP endpoint. It will report if the application is healthy or in an error state.



Health Checks

- Status report
- `/healthz` or `/status`
- Return response
  - `result: "healthy"`
  - `result: "error"`

Metrics are necessary to understand the behavior of an application. The statistics should be gathered for individual services as it will increase the visibility of what resources the application requires to be fully operational, for example, the ***amount of CPU, memory, or network throughput.*** At the same time, a service can report on the number of resources it is capable to handle, for example, amount of requests, active users, or the amount of login(s). Usually, the metrics are consumed via an HTTP endpoint such as forward slash (**/**) metrics, which returns the statistics for the service, for example, the amount of users or the amount of active users.



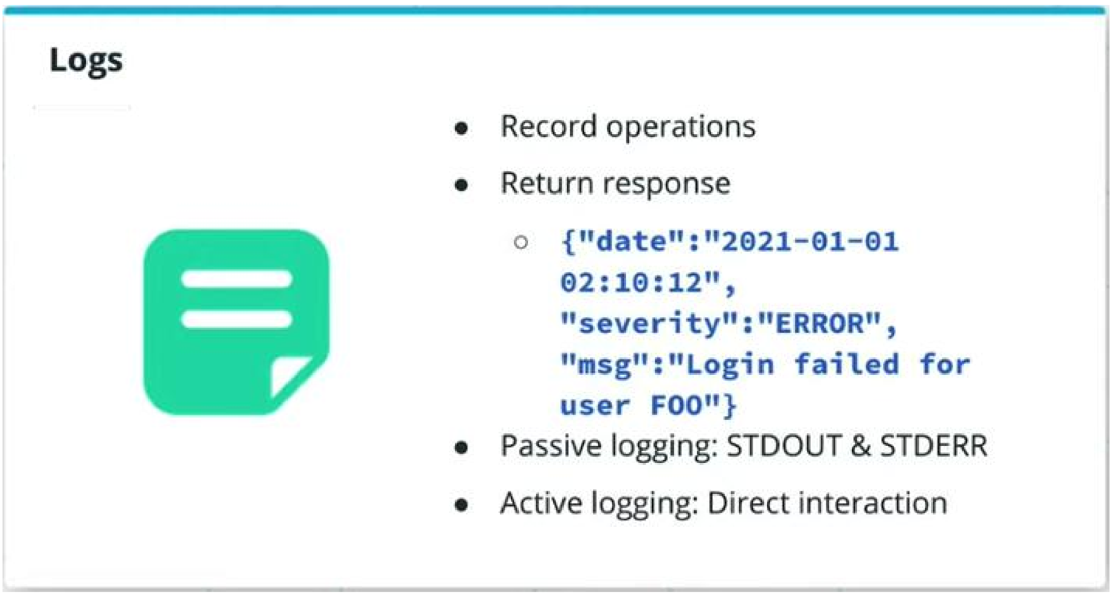
Metrics

- Statistics for individual service
- Used and handled resources
- `/metrics`
- Return response
  - `result: {"UserCount":140,"UserCountActive":23"}`

Log aggregation is the nucleus of any troubleshooting and debugging process for a project. **Logs are used to record operations that are performed by a service**, for example, whether a user has successfully logged into a service or a user encountered an error while reaching a particular function. In this case, we have an error log line with a timestamp recorded on January 1st, 2021, and the log message mentions that the user "FOO" failed to log in. Usually, the logs are **collected from STDOUT, or standard out, and STDERR**, or standard error, through a passive logging mechanism. This means that the application should send the command output and errors to the shell. This will then be collected by a **log aggregated tool, such as Fluentd or Splunk, and sent to a back-end storage or a database.**

However, the application can send the logs records to the back-end storage or a database. In this case, an **active logging technique** is used, as the l**ogs transmission is handled directly by**

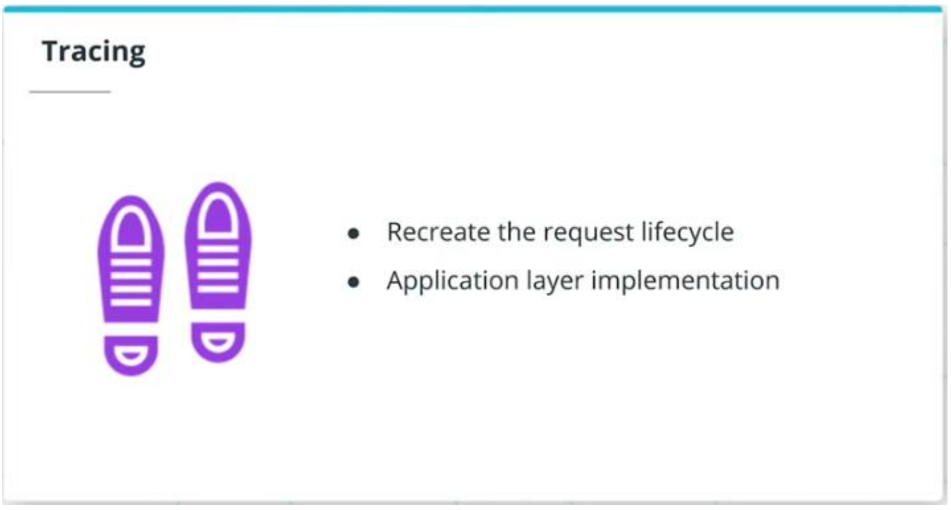
the service without a log aggregated tool such as **Fluentd or Splunk.**



Logs

- Record operations
- Return response
  - {"date":"2021-01-01 02:10:12", "severity":"ERROR", "msg":"Login failed for user FOO"}
- Passive logging: STDOUT & STDERR
- Active logging: Direct interaction

Tracing is capable of showing a holistic picture of how different services are invoked to fulfill a request. With the tracing library integrated, it is possible to **recreate and analyze the life cycle** for request and identify key metrics within an application. Usually, tracing is **implemented at the application layer**, where the developer will be able to record when a function is invoked.

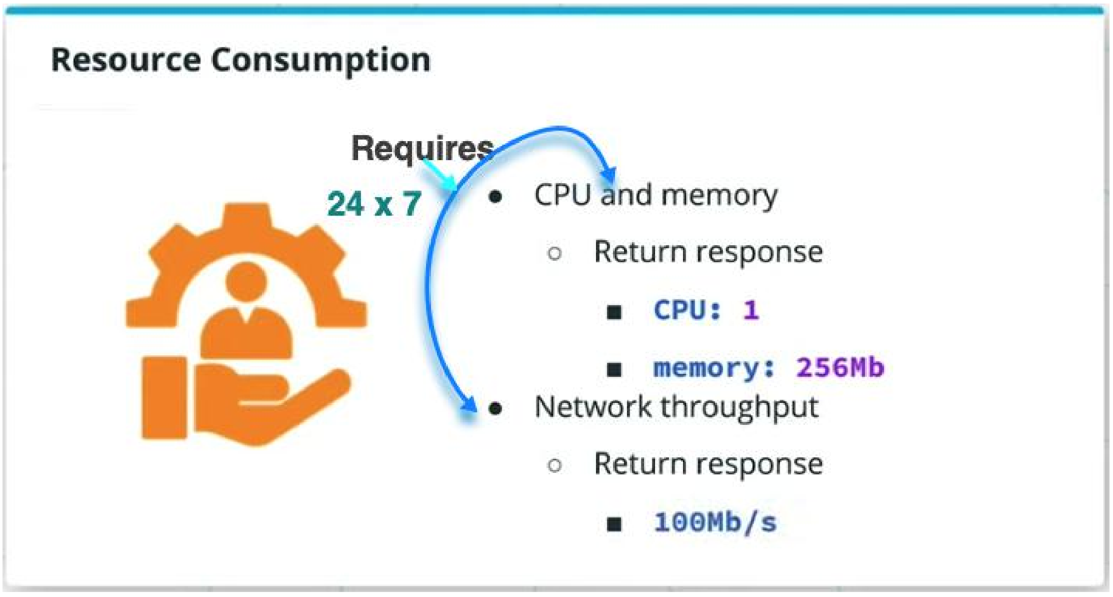


Tracing

- Recreate the request lifecycle
- Application layer implementation

Resource consumption refers to the amount of CPU and memory an application requires to be fully operational. For

example, an application might require an entire CPU and 256 megabytes to execute successfully. This includes the network throughput as well, making sure that the application has enough bandwidth to handle multiple requests. For example, this application has the bandwidth to serve 100 megabytes per second. Having awareness of the application boundaries is fundamental to ensure service availability 24/7.
This will reduce the chances of an application to starve on CPU or be killed by insufficient memory capacity.



Resource Consumption

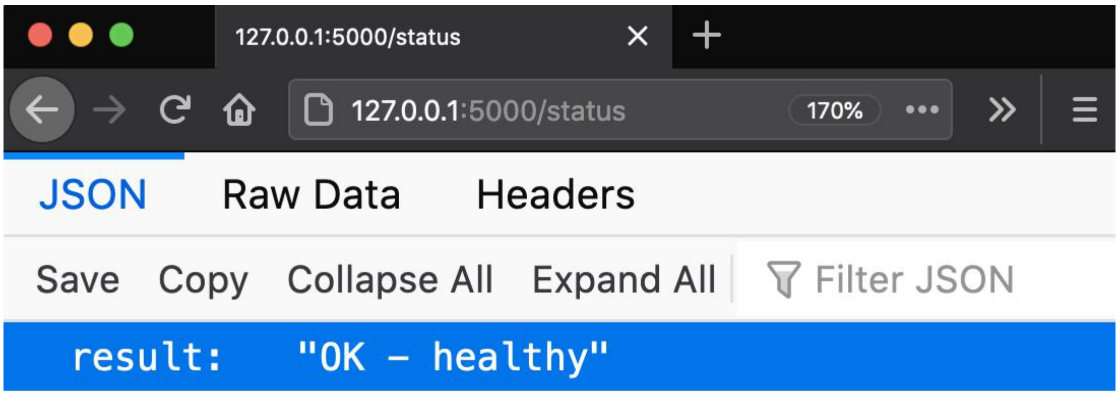
- CPU and memory
  - Return response
    - `CPU: 1`
    - `memory: 256Mb`
- Network throughput
  - Return response
    - `100Mb/s`

Summary

Using the knowledge acquired so far, you should be able to choose the most suitable architecture for an application, based on requirements, available resources, and involved trade-offs. The next stage consists of building the application. Regardless of the chosen architecture, a set of good development practices can be applied to improve the application lifecycle throughout the release and maintenance phases. Adopting these practices increases resiliency, lowers the time to recovery, and provides transparency of how a service handles incoming requests. These practices are focused on health checks, metrics, logs, tracing, and resource consumption.

Health Checks

Health checks are implemented to showcase the status of an application. These checks report if an application is running and meets the expected behavior to serve incoming traffic. Usually, health checks are represented by an HTTP endpoint such as /healthz or /status. These endpoints return an HTTP response showcasing if the application is healthy or in an error state.

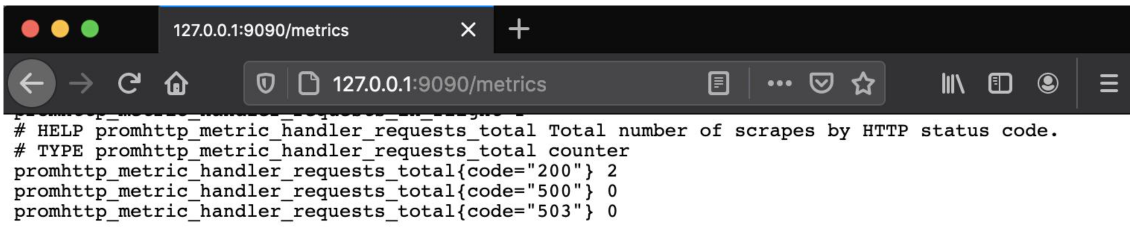


```
result:    "OK - healthy"
```

/status health check that showcases that the application is healthy

Metrics

Metrics are necessary to quantify the performance of the application. To fully understand how a service handles requests, it is mandatory to collect statistics on how the service operates. For example, the number of active users, handled requests, or the number of logins. Additionally, it is paramount to gather statistics on resources that the application requires to be fully operational. For example, the amount of CPU, memory, and network throughput. Usually, the collection of metrics are returned via an HTTP endpoint such as /metrics, which contains the internal metrics such as the number of active users, consumed CPU, network throughput, etc.



```
# HELP promhttp_metric_handler_requests_total Total number of scrapes by HTTP status code.
# TYPE promhttp_metric_handler_requests_total counter
promhttp_metric_handler_requests_total{code="200"} 2
promhttp_metric_handler_requests_total{code="500"} 0
promhttp_metric_handler_requests_total{code="503"} 0
```

`/metrics` endpoint that list of metrics counting the amount requests by the HTTP code returned

Logs

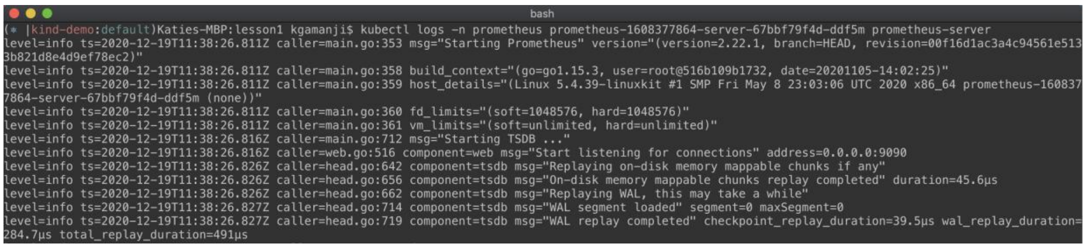
Log aggregation provides valuable insights into what operations a service is performing at a point in time. It is the nucleus of any troubleshooting and debugging process. For example, it is essential to record if a user logged in successfully into a service, or encountered an error while performing a payment.

Usually, the logs are collected from STDOUT (standard out) and STDERR (standard error) through a passive logging mechanism. This means that any output or errors from the application are sent to the shell. Subsequently, these are collected by a logging tool, such as Fluentd or Splunk, and stored in backend storage. However, the application can send the logs directly to the backend storage. In this case, an active logging technique is used, as the log transmission is handled directly by the application, without a logging tool required.

There are multiple logging levels that can be attributed to an operation. Some of the most widely used are:

- DEBUG – record fine-grained events of application processes
- INFO – provide coarse-grained information about an operation
- WARN – records a potential issue with the service
- ERROR – notifies an error has been encountered, however, the application is still running
- FATAL – represents a critical situation, when the application is not operational

As well, it is common practice to associate each log line with a timestamp, that will exactly record when an operation was invoked.



```
bash
(⎈ |kind-demo:default)Katies-MBP:lesson1 kgamanji$ kubectl logs -n prometheus prometheus-1608377864-server-67bbf79f4d-ddf5m prometheus-server
level=info ts=2020-12-19T11:38:26.811Z caller=main.go:353 msg="Starting Prometheus" version="(version=2.22.1, branch=HEAD, revision=00f16d1ac3a4c94561e513
3b821d8e4d9ef78ec2)"
level=info ts=2020-12-19T11:38:26.811Z caller=main.go:358 build_context="(go=go1.15.3, user=root@516b109b1732, date=20201105-14:02:25)"
level=info ts=2020-12-19T11:38:26.811Z caller=main.go:359 host_details="(Linux 5.4.39-linuxkit #1 SMP Fri May 8 23:03:06 UTC 2020 x86_64 prometheus-160837
7864-server-67bbf79f4d-ddf5m (none))"
level=info ts=2020-12-19T11:38:26.811Z caller=main.go:360 fd_limits="(soft=1048576, hard=1048576)"
level=info ts=2020-12-19T11:38:26.811Z caller=main.go:361 vm_limits="(soft=unlimited, hard=unlimited)"
level=info ts=2020-12-19T11:38:26.816Z caller=main.go:712 msg="Starting TSDB ..."
level=info ts=2020-12-19T11:38:26.816Z caller=web.go:516 component=web msg="Start listening for connections" address=0.0.0.0:9090
level=info ts=2020-12-19T11:38:26.826Z caller=head.go:642 component=tsdb msg="Replaying on-disk memory mappable chunks if any"
level=info ts=2020-12-19T11:38:26.826Z caller=head.go:656 component=tsdb msg="On-disk memory mappable chunks replay completed" duration=45.6µs
level=info ts=2020-12-19T11:38:26.826Z caller=head.go:662 component=tsdb msg="Replaying WAL, this may take a while"
level=info ts=2020-12-19T11:38:26.827Z caller=head.go:714 component=tsdb msg="WAL segment loaded" segment=0 maxSegment=0
level=info ts=2020-12-19T11:38:26.827Z caller=head.go:719 component=tsdb msg="WAL replay completed" checkpoint_replay_duration=39.5µs wal_replay_duration=
284.7µs total_replay_duration=491µs
```

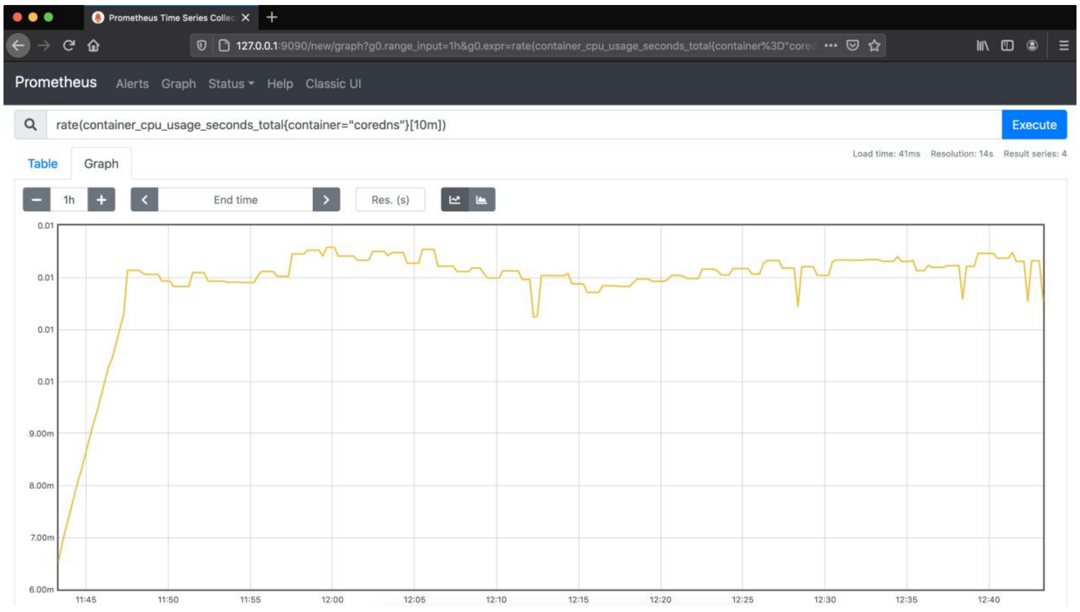
Multiple INFO log lines recorded when a Prometheus service started

Tracing

Tracing is capable of creating a full picture of how different services are invoked to fulfill a single request. Usually, tracing is integrated through a library at the application layer, where the developer can record when a particular service is invoked. These records for individual services are defined as spans. A collection of spans define a trace that recreates the entire lifecycle of a request.

Resource Consumption

Resource consumption encapsulates the resources an application requires to be fully operational. This usually refers to the amount of CPU and memory that is consumed by an application during its execution. Additionally, it is beneficial to benchmark the network throughput, or how many requests can an application handle concurrently. Having awareness of resource boundaries is essential to ensure that the application is up and running 24/7.



```
rate(container_cpu_usage_seconds_total{container="coredns"}[10m])
```

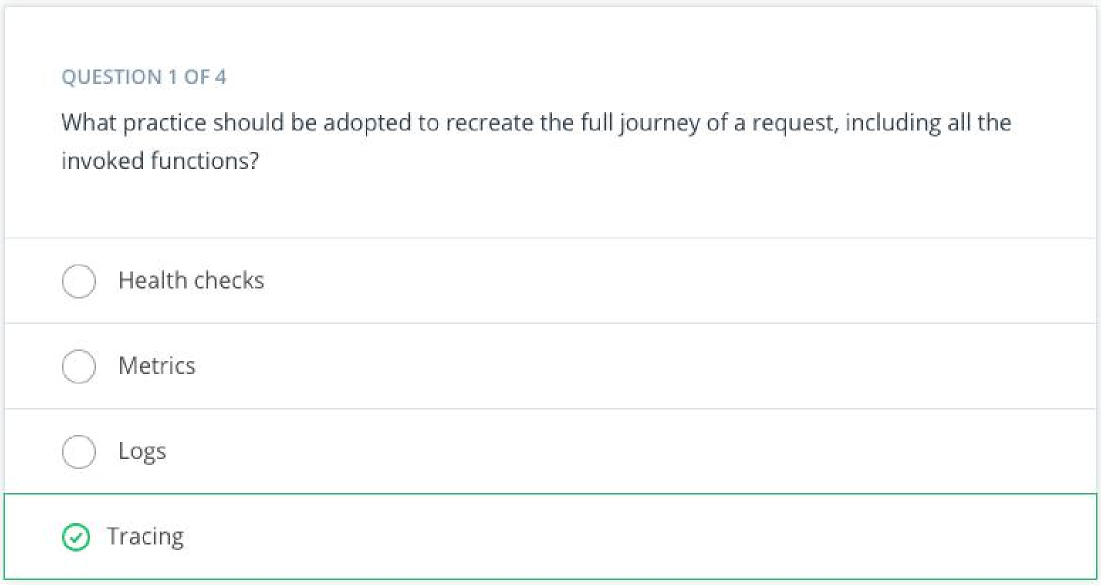
A GRAPH SHOWCASING THE CPU CONSUMPTION OF THE COREDNS CONTAINER

Further reading

- Health Checks – explore the core reasons to introduce health checks and implementations examples
  Url: https://microservices.io/patterns/observability/health-check-api.html
- Prometehus Best Practices on Metrics Naming – explore how to name, label, and define the type of metrics
  Url: https://prometheus.io/docs/instrumenting/writing_exporters/#metrics
- Application Logging Best Practices – read more on how to define what logs should be collected by an application
  Url: https://logz.io/blog/logging-best-practices/
- Logging Levels – explore possible logging levels and when they should be enabled
  Url: https://www.tutorialspoint.com/log4j/log4j_logging_levels.htm
- Enabling Distributed Tracing for Microservices With Jaeger in Kubernetes – learn what tools can be used to implement tracing in a Kubernetes cluster

Url:https://containerjournal.com/topics/container-ecosystems/enabling-distributed-tracing-for-microservices-with-jaeger-in-kubernetes/

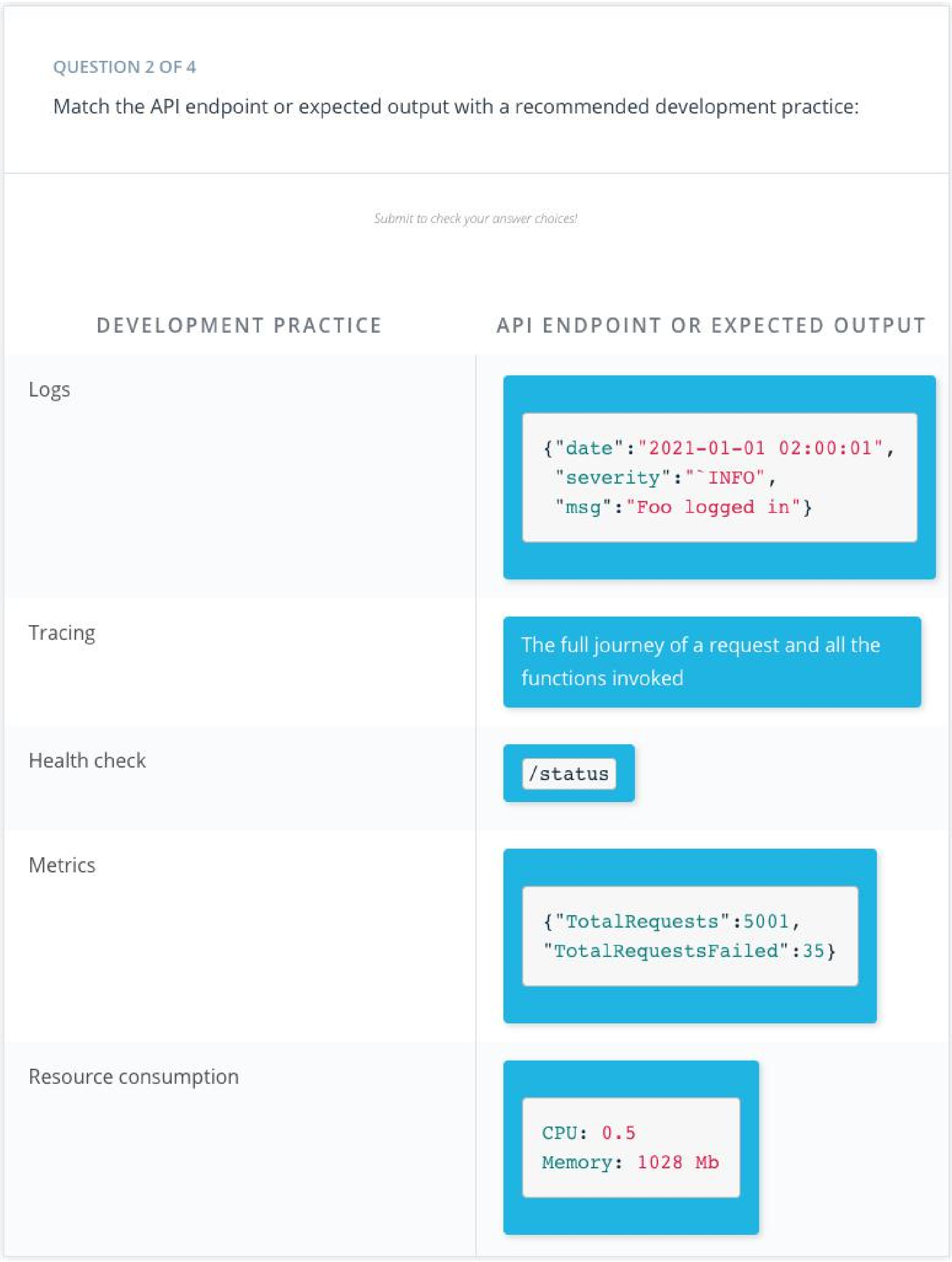
Quizzes: Best Practices For Application Deployment



QUESTION 1 OF 4

What practice should be adopted to recreate the full journey of a request, including all the invoked functions?

- Health checks
- Metrics
- Logs
- [✓ Tracing]

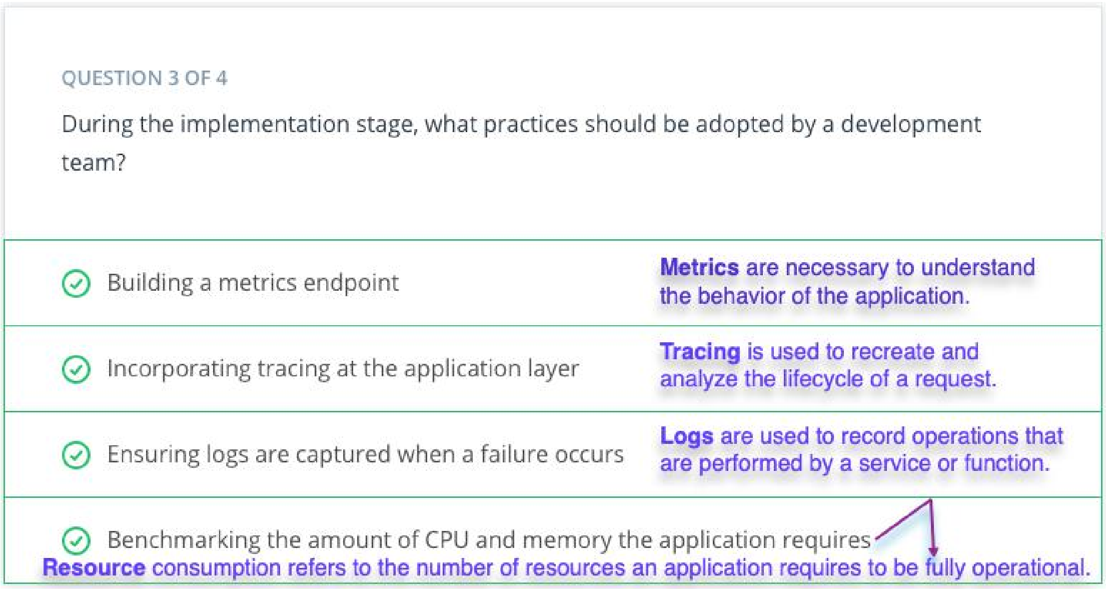


QUESTION 2 OF 4

Match the API endpoint or expected output with a recommended development practice:

Submit to check your answer choices!

DEVELOPMENT PRACTICE	API ENDPOINT OR EXPECTED OUTPUT
Logs	`{"date":"2021-01-01 02:00:01", "severity":"`INFO", "msg":"Foo logged in"}`
Tracing	The full journey of a request and all the functions invoked
Health check	`/status`
Metrics	`{"TotalRequests":5001, "TotalRequestsFailed":35}`
Resource consumption	`CPU: 0.5` `Memory: 1028 Mb`

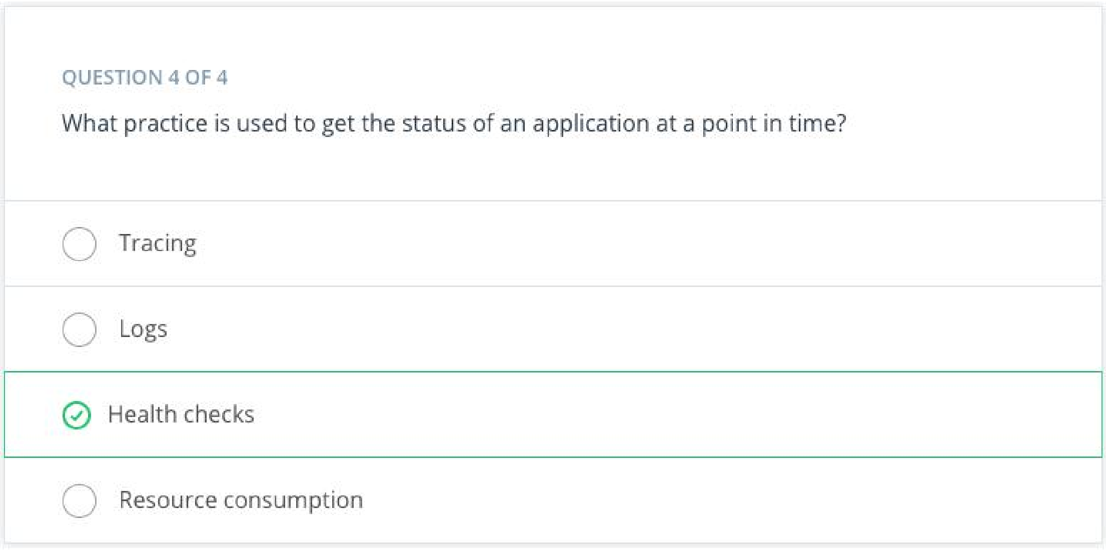


QUESTION 3 OF 4

During the implementation stage, what practices should be adopted by a development team?

- [✓ Building a metrics endpoint] — **Metrics** are necessary to understand the behavior of the application.
- [✓ Incorporating tracing at the application layer] — **Tracing** is used to recreate and analyze the lifecycle of a request.
- [✓ Ensuring logs are captured when a failure occurs] — **Logs** are used to record operations that are performed by a service or function.
- [✓ Benchmarking the amount of CPU and memory the application requires]

Resource consumption refers to the number of resources an application requires to be fully operational.



QUESTION 4 OF 4

What practice is used to get the status of an application at a point in time?

- Tracing
- Logs
- [✓ Health checks]
- Resource consumption

Exercise

Extend the Python Flask application with `/status` and `/metrics` endpoints, considering the following requirements:

- Both endpoints should return an HTTP 200 status code
- Both endpoints should return a JSON response e.g. `{"user": "admin"}`. (Note: the JSON response can be hard coded at this stage)
- The `/status` endpoint should return a response similar to this example: `result: OK – healthy`
- The `/metrics` endpoint should return a response similar to this example: `data: {UserCount: 140, UserCountActive: 23}`

Tips: If you get stuck, feel free to check the solution video where detailed operations are demonstrated.

In this video, we'll have a walk–through solution for the exercise of creating new endpoints for a Python application by using the Flask framework.

Currently, we have our simple Python application which resides in the app.py file. If we just open that particular file by using the py command, we'll be able to see that we have a Flask application, and then we have a couple of routes such as slash

status, slash metrics, and we have our main route, which is going to return a ''Hello World!'' application. Having the exercise, we had to create a status endpoint and a metrics endpoint.
If we go to the status endpoint, then we return a hard-coded response, which is going to be a JSON response with the main result being ''Okay-healthy'' and it's going to return a 200 HTTP code, meaning that the request was successful.
A very similar thing is going to be done with the slash metrics endpoint, where we hardcode a JSON response. Where we return the status being successful, that can return a code zero which means again, the requests has been successful.
Then we can have metrics for the user account or the amount of user which are active. Again, we return at 200, which ensures that the request was successful. The main route is just returning a ''Hello World!'' application. Pretty much this is the content of our application with multiple endpoints. If we exit this particular file, to actually run this application, we're going to use a Python command.
Python already has been installed on your machine as a prerequisite for this course. If we do a Python dash dash version, you'll be able to see that currently,
I'm running Python in version 3.9.5 The last thing we need to do is to test our new impotence.
Going to invoke Python and then our application which is app.py. If you press "Enter," then we will see that we have an application running on localhost on port 5,000.

This port is the default port which is going to be opened by the Flask web server. Let's actually see our application in the browser. If I open Chrome Firefox at the moment,
and if I go to localhost, which is going to be on 127.0.0.1 on port 5,000 and click "Enter", then we'll see our main "Hello World!" message, which we expect from our application.
However, we've created two new endpoints a status and the metrics. Let's check those as well. If we navigate back to our bar and after the port 5,000 we input a forward slash status,
we'll be able to see our hard coded result with, "Okay–healthy". If we test our metrics endpoint, so we go back to our navigation bar and type metrics and click "Enter", we'll be able to see the hard–coded response with our User Account being 140, and the user, which are active here set to 23 and of course we can see the status of this request being successful as well.

This is pretty much how it would be able to create multiple endpoints for an application and test them using a local browser such as Google Chrome or Firefox.

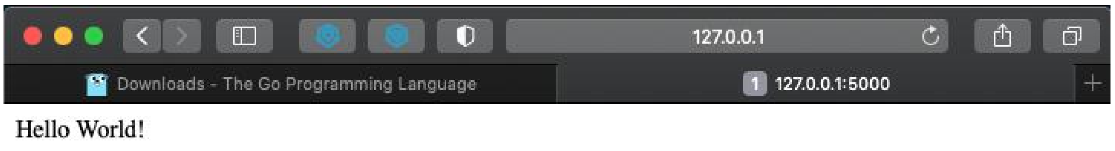
Write Code with me: Original

```
from flask import Flask

app = Flask(__name__)

@app.route("/")
def hello():
    return "Hello World!"

if __name__ == "__main__":
app.run(host='127.0.0.1')
```

Routing is the mechanism of mapping the URL directly to the code that creates the webpage. It helps in better management of the structure of the webpage and increases the performance of the site considerably and further enhancements or modifications become really straight forward. In python routing is implemented in most of the web frame works. We will see the examples from flask web framework

Routing in Flask

The route() decorator in Flask is used to bind an URL to a function. When the URL '/' is mentioned in the browser, the function hello() is executed to give the result.



Hello World!

```
atulsaxena@Atuls-iMac python-helloworld % python3 app.py
 * Serving Flask app 'app' (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production
deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [29/Jun/2021 16:54:14] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [29/Jun/2021 16:54:14] "GET /favicon.ico HTTP/1.1" 404 -
```

Let’s first import json from flask

```
From flask import json
```

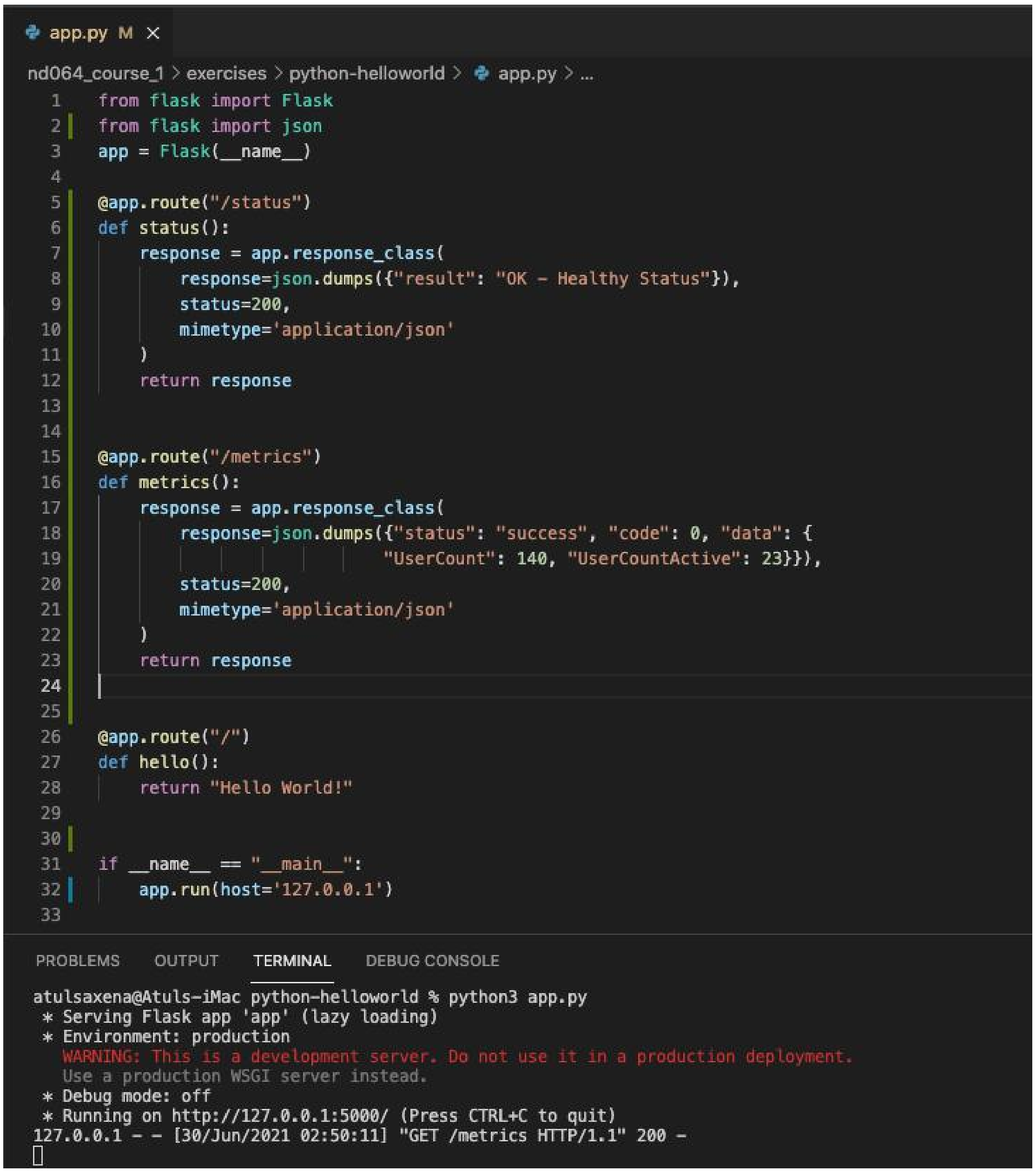
Let’s create end points for `/status`

```
@app.route(“/status”)
def status():
    response = app.response_class(
            response=json.dumps({“result” : “OK - Healthy Status”}),
            status=200,
            mimetype=‘application/json’
    )
    return response
```

Let’s create the end point for `/metrics`

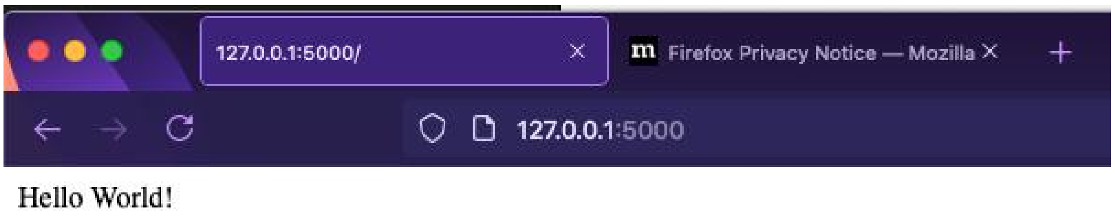
```
@app.route("/metrics")
def metrics():
response = app.response_class(
response=json.dumps({"status": "success", "code": 0, "data": {
"UserCount": 140, "UserCountActive": 23}}),
status=200,
mimetype='application/json'
)
return response
```

Complete Code in the following figure


app.py M
nd064_course_1 > exercises > python-helloworld > app.py > ...
PROBLEMS
OUTPUT
TERMINAL
DEBUG CONSOLE

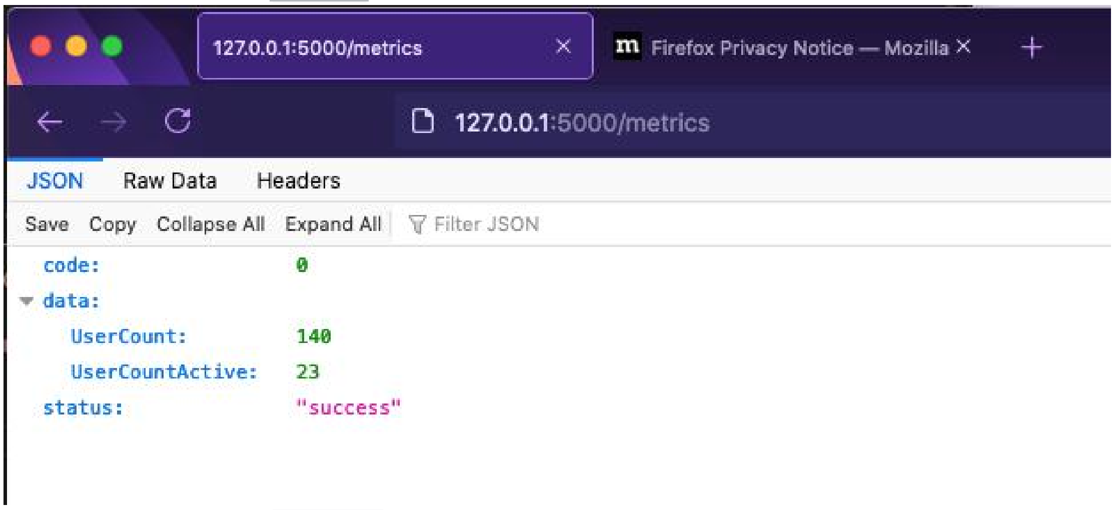
```python
from flask import Flask
from flask import json
app = Flask(__name__)

@app.route("/status")
def status():
    response = app.response_class(
        response=json.dumps({"result": "OK - Healthy Status"}),
        status=200,
        mimetype='application/json'
    )
    return response


@app.route("/metrics")
def metrics():
    response = app.response_class(
        response=json.dumps({"status": "success", "code": 0, "data": {
                        "UserCount": 140, "UserCountActive": 23}}),
        status=200,
        mimetype='application/json'
    )
    return response


@app.route("/")
def hello():
    return "Hello World!"


if __name__ == "__main__":
    app.run(host='127.0.0.1')
```

```
atulsaxena@Atuls-iMac python-helloworld % python3 app.py
 * Serving Flask app 'app' (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [30/Jun/2021 02:50:11] "GET /metrics HTTP/1.1" 200 -
```



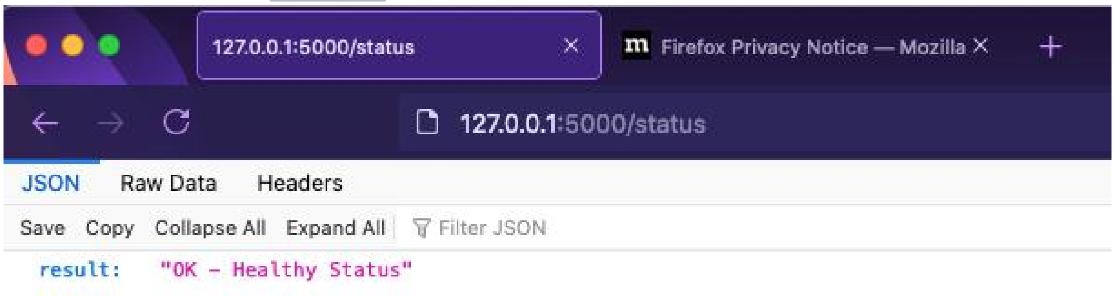
Hello World!

With end point /status



```
code:             0
data:
  UserCount:      140
  UserCountActive: 23
status:           "success"
```

With end point /metrics



```
result:   "OK - Healthy Status"
```

Logging in Python

Whenever we write a code there are chances, we may find some bugs in our programs i.e we call them errors or exceptions which hinder the normal execution of our applications. Logging comes very handy in software development process especially debugging and running. We may figure it out the bug in our small programs, but in real world, where programs are huge and it is near to impossible to find a bug manually. Though it is possible but it consumes programmer's lot of time.

Python has an in-built logging module for our use which solves this problem of ours. Logging is very useful tool and makes our programmer’s life easier in debugging our programs if they encounter any error.

We need logging in our programs

```
import logging
```

The logging module helps us to write status message to a file or other output streams. The file can contain other information including, which part of the code is executed and what problems have arisen.

With logging module, we can use “logger” to log messages. We have 5 kind of log messages, which depicts the severity of the events

The following 5 levels in increasing order of the severity of the events.

- **debug:** To give detailing information. It is used when diagnosing problems
- **info:** To confirm that things are working correctly as expected.
- **warning:** To give some message which informs us of an issue which can be cause problem in future.
- **error:** To give an error message, that the application or software has failed to perform some function.
- **critical:** It informs of a serious problem indicating that the program may stop performing.

The logging module provides us with a default logger, which allows us to proceed without much configuration.

E.g.

```
Import logging

logging.debug(‘My debug message, but it will not appear on terminal’)
logging.info(‘My info message, but it will not appear on terminal’)
logging.warning(‘My warning message on terminal’)
logging.error(‘My error message on terminal’)
logging.critical(‘My critical message on terminal’)
```

Output

```
WARNING:root:My warning message on terminal
ERROR:root:My error message on terminal
CRITICAL:root:My critical message on terminal
```

Exercise:

```
from flask import Flask
from flask import json
import logging

app = Flask(__name__)


@app.route("/status")
def status():
response = app.response_class(
response=json.dumps({"result": "OK - Healthy Status"}),
status=200,
mimetype='application/json'
)
# logging.warning("I warn you, please write better code")
app.logger.info('Status Request Successfully!')
return response


@app.route("/metrics")
def metrics():
response = app.response_class(
response=json.dumps({"status": "success", "code": 0, "data": {
"UserCount": 140, "UserCountActive": 23}}),
status=200,
mimetype='application/json'
)
# logging.info('I told you that')
app.logger.info('Metrics Request Successfully')
return response


@app.route("/")
def hello():
app.logger.info('Main Request')
app.logger.debug('My debug message')
app.logger.info('My info message')
app.logger.warning('My warning message')
app.logger.error('My error message')
app.logger.critical('My critical message')
return "Hello World!"


if __name__ == "__main__":
logging.basicConfig(filename="app.log", level=logging.DEBUG)
app.run(host='127.0.0.1')
```

Output in app.log file as shown below

```
INFO:werkzeug: * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
INFO:app:Status Request Successfully!
INFO:werkzeug:127.0.0.1 - - [01/Jul/2021 00:52:41] "GET /status HTTP/1.1" 200 -
INFO:app:Metrics Request Successfully
INFO:werkzeug:127.0.0.1 - - [01/Jul/2021 00:53:05] "GET /metrics HTTP/1.1" 200 -
INFO:app:Main Request
DEBUG:app:My debug message
INFO:app:My info message
WARNING:app:My warning message
ERROR:app:My error message
CRITICAL:app:My critical message
INFO:werkzeug:127.0.0.1 - - [01/Jul/2021 00:53:15] "GET / HTTP/1.1" 200 -
```

Reflection on Design Consideration for an Application
What are your opinions on the core considerations for choosing a microservice vs monolith methodology?

In my opinion, **monolithic architecture** is an old traditional way of building an application because it posses a number of challenges associated with handling of huge code base, adopting new technology, scaling, deployment and implementing new changes.
On other hand, adopting **microservices architecture** is latest trend these days, it offers visible benefits like increase in scalability, flexibility, agility etc. In this approach, the entire functionality splits into independently deployable modules which communicate with each other using APIs. Each service has it's own space and can be updated, deployed, and scaled independently.

Things to think about
Contouring the architecture for an application is not a simple task. It requires a thorough understanding of the requirements and available resources.

15.Edge Case: Amorphous Applications

In the previous sections, we have explored how to choose a suitable architecture for an application and how to apply some of the best development practices. However, this is only the start of the application lifecycle. After an engineering team has successfully released a product, with both monolith and microservices, the next phase in the application lifecycle is maintenance. In this edge case, we will explore commonly used maintenance operations after a product is released.

Transcript:

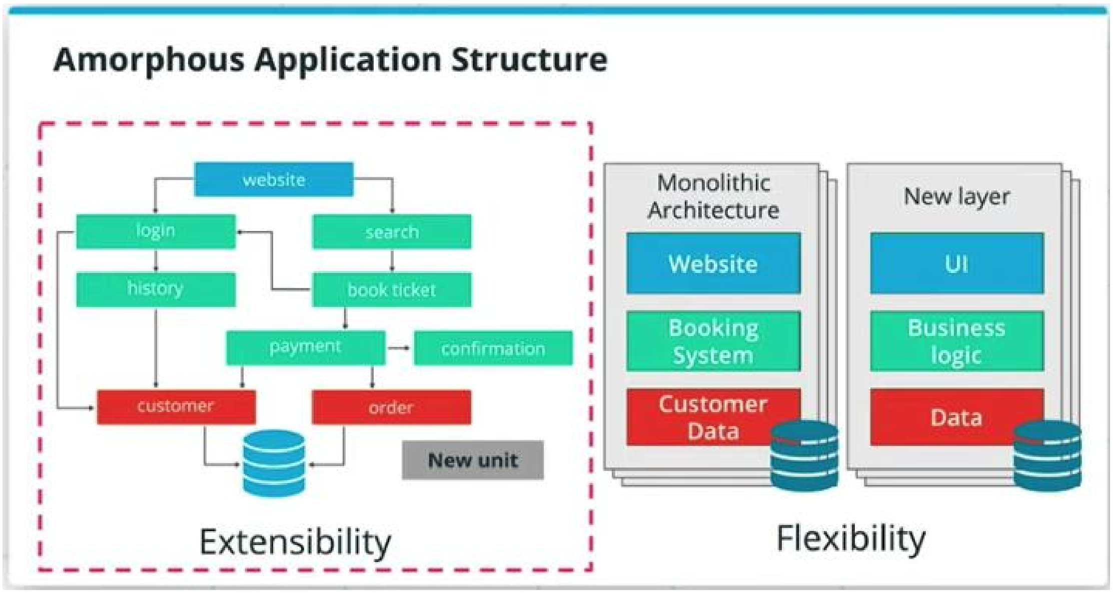
So far, we have traversed the requirements gathering, design, and implementation stages for a project. We need to consider how we'll maintain the application, and iterate on customer feedback.

In this section, the aim is to highlight the organic growth of a project as the structure is not static, but rather an amorphous manifest that is in constant movement. As developers, we should always consider and analyze if the structure of the project is enabling and not blocking the new feature development.

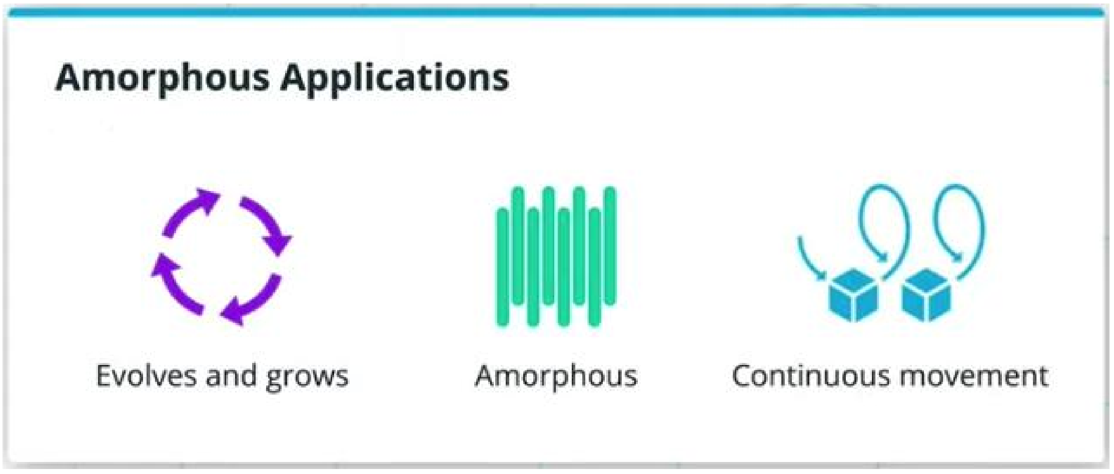
👉 The main reason to design a product is to identify its main components and how these interlink to provide a cohesive user experience.

In the maintenance phase, the product needs to be adjusted and extended with new functionalities. With micro-services, we can easily add a new unit and extend the horizon of application capabilities. However, with monoliths, we might require to introduce an entire abstraction layer, making sure that the application can handle new services.

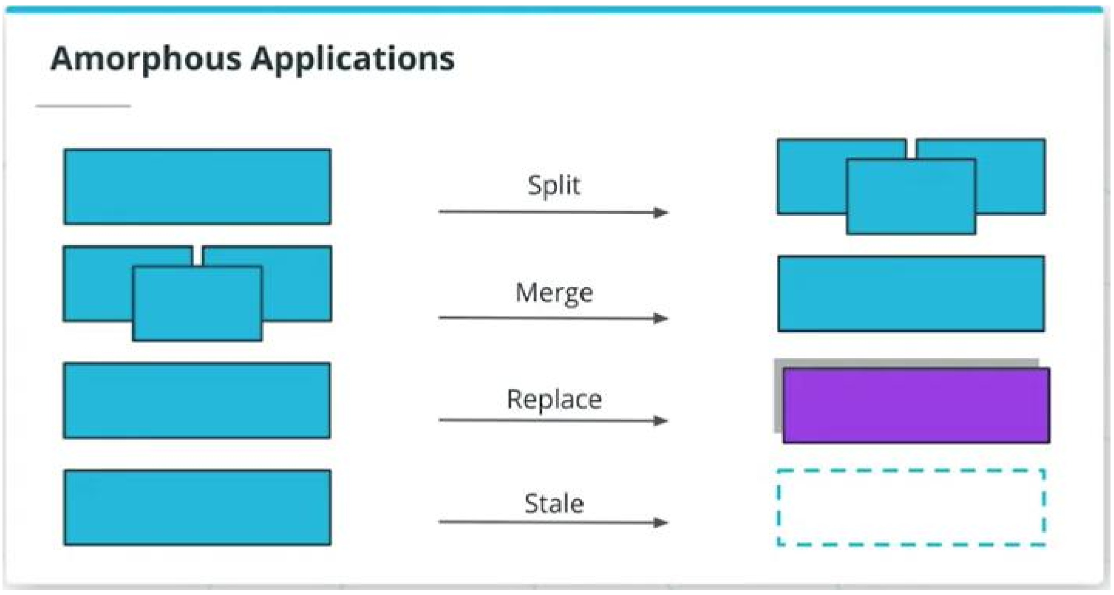
In general, it is always a good principle to focus on extensibility rather than flexibility. That means that it is more efficient to manage multiple services with well-defined and simple functionality, as in the case of micro-services, rather than add more abstraction layers to integrate and support new services, as we've seen with the monoliths.



Extensibility and flexibility should be in constant consideration by the application developers. The project architecture is not a static manifest. It evolves and it grows based on the user feedback and recurring feedback iterations. Consequently, the structure of the project is amorphous and it is in continuous movement.



In the maintenance phase, the project structure can change, especially the following operations are performed. A split operation, which should be considered if a monolith or a micro-service covers too many functionalities. It is more suitable to have simpler units to manage rather than have a single and more complex service. The merge operation should be performed if units are too granular or perform closely interlinked operations e.g. having two separate services for log output and log format, when these could be covered by a single service. It provides little development advantage to have this union segregated. Hence a merge of these functionalities is beneficial. A replace operation should be examined if a more efficient implementation is identified for a service e.g. rewriting the unit using a different programming language or library to optimize and extend a process. Finally, a stale operation should be considered for services that are no longer providing any business value e.g. services that were used to perform a one–off migration process. These units are usually archived or removed to reduce the number of managed services. Performing any of these operations will increase the longevity and continuity of a project. But more importantly, you can observe that the structure of a project, it's not static. It's amorphous and it evolves based on the new requirements and customer feedback.

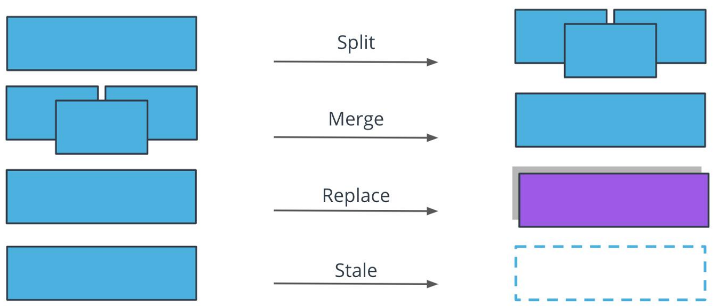


Summary

Throughout the maintenance stage, the application structure and functionalities can change, and this is expected! The architecture of an application is not static, it is amorphous and in constant movement. This represents the organic growth of a product that is responsive to customer feedback and new emerging technologies.

Both monolith and microservice–based applications transition in the maintenance phase after the production release. When considering adding new functionalities or incorporating new tools, it is always beneficial to focus on **extensibility rather than flexibility**. Generally speaking, it is more efficient to manage multiple services with a well–defined and simple functionality (as in the case of microservices), rather than add more abstraction layers to support new services (as we've seen with the monoliths). However, to have a well–structured maintenance phase, it is essential to understand the reasons an architecture is chosen for an application and involved trade–offs.

Some of the most encountered operations in the maintenance phase are listed below:



Application operations to occur in the maintenance phase

- A **split** operation – is applied if a service covers too many functionalities and it's complex to manage. Having smaller, manageable units is preferred in this context.
- A **merge** operation– is applied if units are too granular or perform closely interlinked operations, and it provides a development advantage to merge these together. For example, merging 2 separate services for log output and log format in a single service.
- A **replace** operation – is adopted when a more efficient implementation is identified for a service. For example, rewriting a Java service in Go, to optimize the overall execution time.
- A **stale** operation – is performed for services that are no longer providing any business value, and should be archived or deprecated. For example, services that were used to perform a one-off migration process.

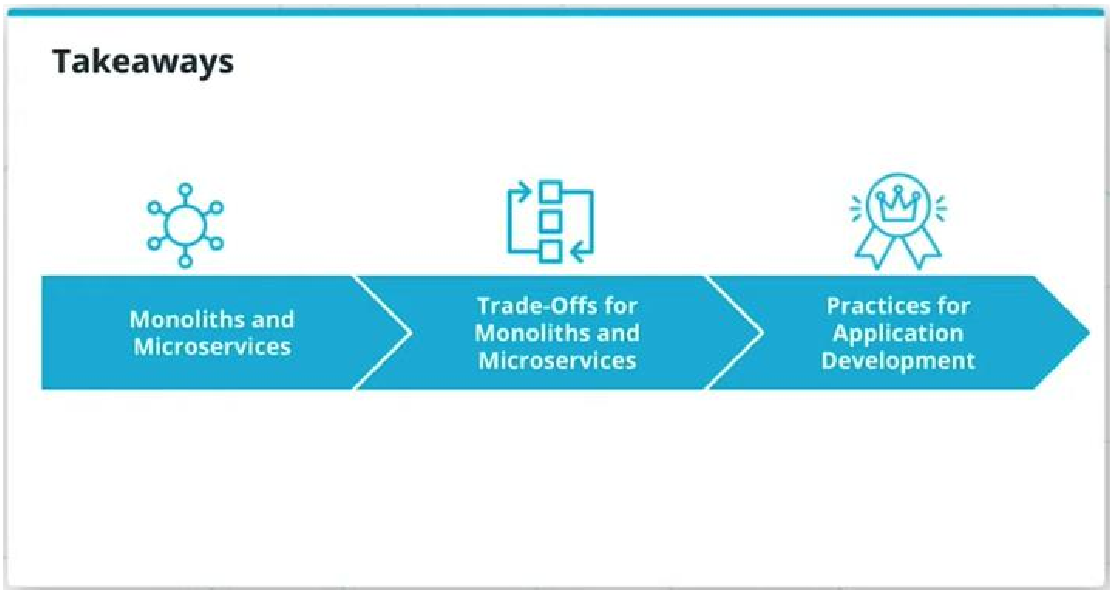
Performing any of these operations increases the longevity and continuity of a project. Overall, the end goal is to ensure the application is providing value to customers and is easy to manage by the engineering team. But more importantly, it can be observed that the structure of a project is not static. It amorphous and it evolves based on new requirements and customer feedback.
Watch how **Monzo is managing thousands of microservices** and evolves their ecosystem: Modern Banking in 1500 Microservices

Lesson 2 Conclusion

Transcript:

In this lesson, you have learned how to select a monolith or microservice–based architecture, considering their requirements and available resources within an organization.

We also covered the involved trade–offs for each architecture, such as development complexity, scalability, time to deploy, flexibility, operational cost, and reliability. Finally, you have learned that regardless of the chosen architecture, there is a suit of best practices that can be applied to enable cloud–native development of a project such as health checks, metrics, logs, tracing, and resource consumption.


Takeaways
Monoliths and Microservices
Trade-Offs for Monoliths and Microservices
Practices for Application Development

Summary

In this lesson, we have covered how to build an application using monolith and microservice–based architecture. The choice of an application structure is highly impacted by available resources, requirements, and involved trade–offs. But more importantly, we have covered development practices that should be considered to optimize an application's resilience, time to recovery, and traceability.

Overall, in this lesson, we covered:

- Monoliths and Microservices
- Trade–offs for Monoliths and Microservices
- Practices for Application Development

Glossary

- Monolith: application design where all application tiers are managed as a single unit
- Microservice: application design where application tiers are managed as independent, smaller units