



Lesson 2: Architecture Consideration for Cloud Native Applications

[Introduction](#)

[Design Considerations](#)

[Monoliths vs Microservices](#)

[Trade-offs](#)

[Best Practices for Application Deployment](#)

[Amorphous Applications](#)

Introduction

- It's important to identify the requirements and structure of an application before building it
- Monoliths, microservices, trade-offs

Design Considerations




- While working on context discovery, it's important to identify the necessary functionalities of the application and the available resources that can enable its buildout
- **Stakeholders:** Understand who requires and sponsors the application
- **Functionalities:** What all functionalities are necessary for the application?
- **End Users:** Who is this application/tool for?
- **I/O Process:** Understand how the input and output flow works. Is the output dependent on input given by the user? How does the output present itself?
- **Engineering Teams:** Which team is skilled enough to build the application and implement the project?

- While talking about the available resources, it's a good practice to list out the following: the number of people who can work on this/ability to hire contractors, financial resources, time frames, and internal knowledge (any particular skills/languages that need to be known beforehand)

Monoliths vs Microservices

- The goal of both architectures is to provide value to customers and make it easy to adjust and extend the existing functionalities
- Both architectures have 3 (three) main layers:
 1. UI (User Interface Layer) – It handles all the HTTP requests triggered by the users and returns a response
 2. Business Logic Layer – The main component that represents a collection of functionalities that provide a service to the customer
 3. Data Layer – Accessing and storing objects through our application execution by using a database

Monolithic vs Microservices

 Properties	 Monolithic	 Microservices
<u>Structure</u>	- All layers are part of the same unit	- An application is broken down into independent, smaller units
<u>Repository + Binaries</u>	- One repository is maintained and a single binary is created for an application	- Each unit maintains a separate repository and binary
<u>Communication</u>	- All units share the same existing resources (CPU, memory)	- All units communicate via API's - Each unit can be implemented in a different programming language
<u>Use Case</u>	- Is a good place to start if one doesn't have a lot of money	- Requires more time, money, and effort - Is usually adopted by big companies who have the resources for it

Trade-offs

Understanding more aspects that help us decide which architecture we must choose:

Trade-offs

<u>Property</u>	Monolith	Microservices
<u>Development Complexity</u>	- One programming language - One repository - Sequential development (in order to add new features, we have to backtrack and make sure everything is compatible with it)	- Multiple programming languages - Multiple repositories - Concurrent development (in order to add new features, we can just add it to that module)
<u>Scalability</u>	- Replicates the entire stack which can lead to overconsumption of resources	- Replication of single units can be done hence the resources are scaled on-demand and are not wasted
<u>Time to Deploy</u>	- Single delivery pipeline is needed - Whenever a change is made, the entire app is deployed. In case of failure of a functionality, the entire app will fail and the availability is reduced - Low velocity at scale	- Multiple delivery pipelines is needed - Whenever a change is made, only that functionality is deployed. In case of failure of a functionality, the entire app will not fail and the app will be available 24x7 to the user - High velocity at scale
<u>Flexibility</u>	Low (in case a change of programming language is required, the entire application will have to be rewritten)	High (in case a change of programming language is required, only one specific unit will have to be rewritten)
<u>Operational Cost</u>	- Low initial cost - High cost at scale	- High initial cost - Low cost at scale
<u>Reliability</u>	- Recovery of the entire stack - Low visibility (combined metrics and logs for the entire application)	- Recovery of the failed unit - High visibility (individual metrics and logs for each unit)

Best Practices for Application Deployment

- **Health Check**: an HTTP endpoint that uses *insert logic* to check the status ("ok" "error") of a certain component or application. Usually found at `"/healthz"` or `"/status"`
- **Metrics**: Should be gathered for individual services (helps identify what resources the application requires to be fully operational). Can contain information like amount of resources used (CPU, Memory), amount of

requests made/left, amount of users and active users, etc. Usually found at an HTTP endpoint like "/metrics"

- **Logs:** Used to record operations. Helpful for rectifying errors mostly. Log aggregated tools: Fluentd, Splunk
- **Tracing:** Usually present in the application layer, it helps a developer record when a function is invoked and what requests are made
- **Resource Consumption:** CPU, Memory, Network Throughput that the application uses

Amorphous Applications

- After building an application, it's important to maintain it and act on customer feedback
- In order to do so, we need to ensure the structure of the project is enabling and not blocking any future feature development
- While maintaining the application, in case we have to extend any functionality, we can easily do this using microservices. However, with monoliths, we might need to add an abstraction layer and ensure that the application can handle the new services
- When it comes to adding new services, it's better to prioritize extensibility (seen in microservices) over flexibility (seen in monoliths)
- Understand that the project architecture is not a static manifest. It's not like "once made, now no more changes are needed 👍"
- The project architecture is always evolving and growing, amorphous, and is in constant movement

▼ Operations that can be performed to change the project architecture during maintenance to increase longevity of the project:

- **Split:** Divide a single complex service into two or more less complex services
- **Merge:** Combine two or more granular services into one service
- **Replace:** If the programming language is being changed (maybe you found a better, more efficient solution using a different method)
- **Stale:** Archive/Remove the service if it provides no business value