



BITS Pilani

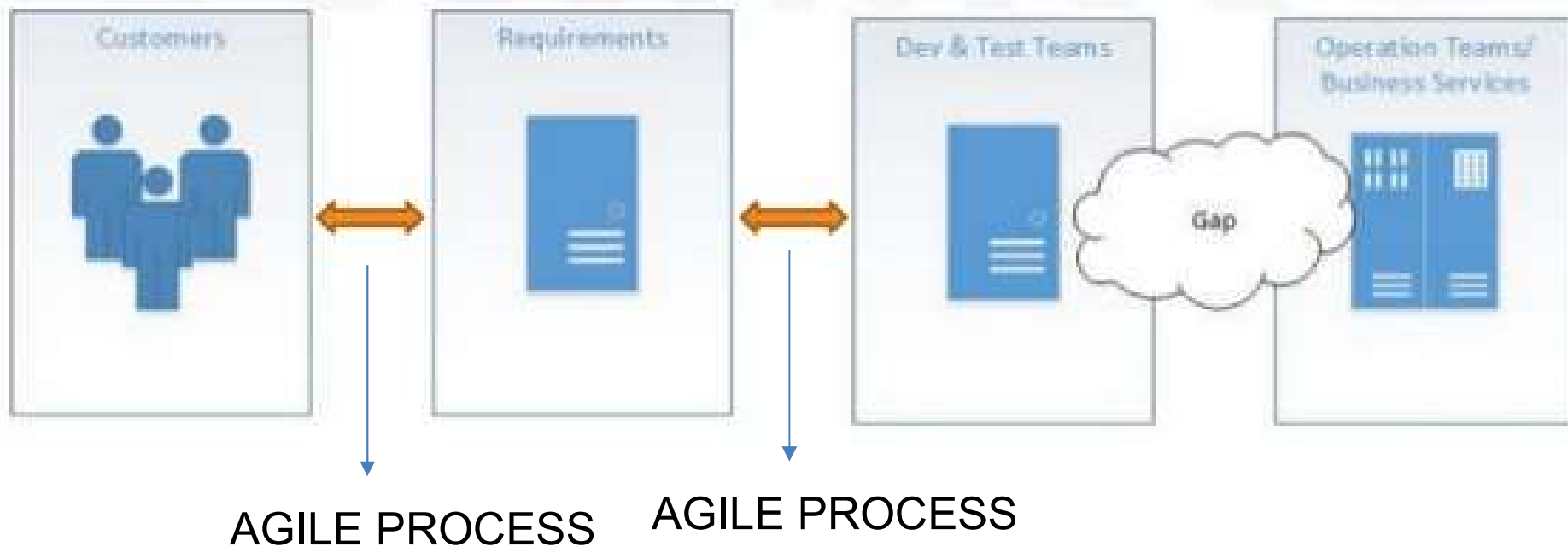
Pilani Campus

DevOps – CI/CD

Problems of Delivering Software



Why DevOps? – Delivery Challenges



Dev and Ops dialogue



Dev:

- Put the current release live, NOW!
- It works on my machine
- We need this Yesterday
- You are using the wrong version



Ops:

- What are the dependencies?
- No machines available...
- Which DB?
- High Availability?
- Scalability?



Need for DevOps -> Faster Deployment Cycles



COMPANY	DEPLOY FREQUENCY	DEPLOY LEAD TIME	RELIABILITY	CUSTOMER FEEDBACK
AMAZON	23,000/day	minutes	high	high
GOOGLE	5,500/day	minutes	high	high
NETFLIX	500/day	minutes	high	high
FACEBOOK	1/day	hours	high	high
TWITTER	3/week	hours	high	high
TYPICAL ENTERPRISE	once every 9 months	months or more	low/medium	low/medium

From "The Phoenix Project"

DevOps

- DevOps is the process of alignment of IT Development and Maintenance Operations with better and improved communication.
- Microsoft defines DevOps as “Union of People, Process and Products to enable continuous delivery of value to the customers”.
- Continuous Integration (CI), Continuous Delivery and / or Continuous Deployment (CD) is a major part of DevOps methodology.



<https://www.redhat.com/en/topics/devops/what-is-ci-cd#:~:text=Continuous%20delivery%20usually%20means%20a,environment%20by%20the%20operations%20team.>

DevOps



Definition of DevOps:

“DevOps is a set of practices intended to reduce the time between committing a change to a system and the change being placed into normal production, while ensuring high quality”

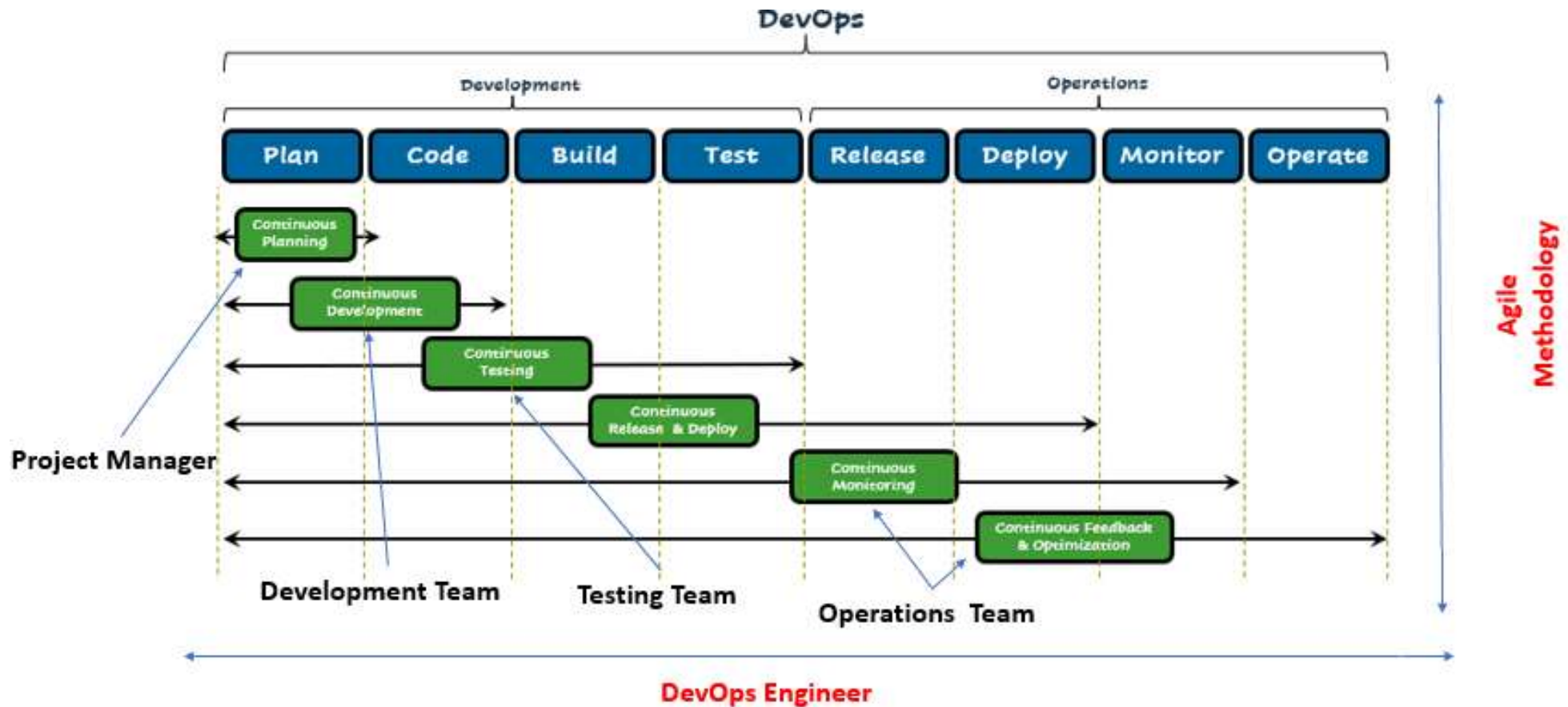
Implications of this definition

- Practices and tools
- Do not restricted scope of DevOps to testing and development

DevOps



DevOps – Generic Methodology





Continuous Integration (CI)

- Code changes are frequently merged into the main branch
- Automated build and test processes ensure that code in the main branch is always production-quality
- Automation process for developers
- Successful CI -> Build, Test and Merge to shared repository

Goals:

- ✓ Faster identification and bug fix
- ✓ Improve software quality
- ✓ Reduce time to release new software

Continuous Delivery and / or Continuous Deployment (CD)



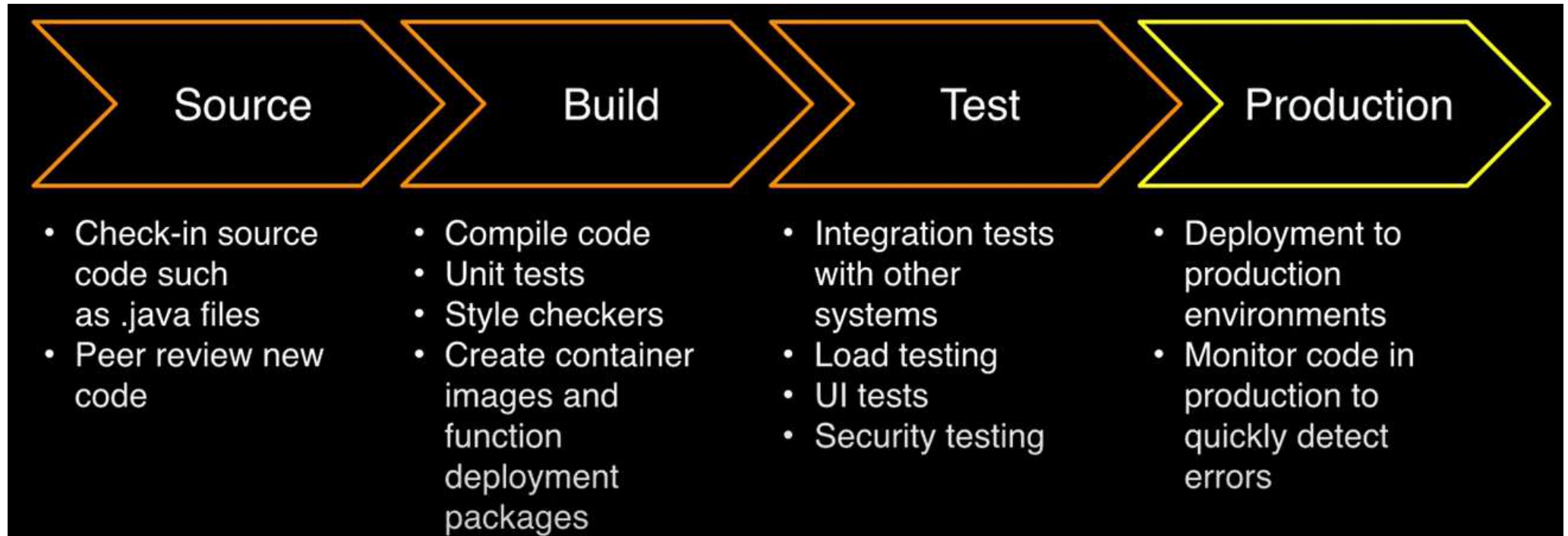
Continuous Delivery

- Automates the software release process
- Every revision triggers an automated flow that builds, tests, and then stages the update into repository like GitHub or Container Registry
- Manual decision to deploy to production
- Goal -> Have a codebase always ready for deployment to Prod environment

Continuous Deployment

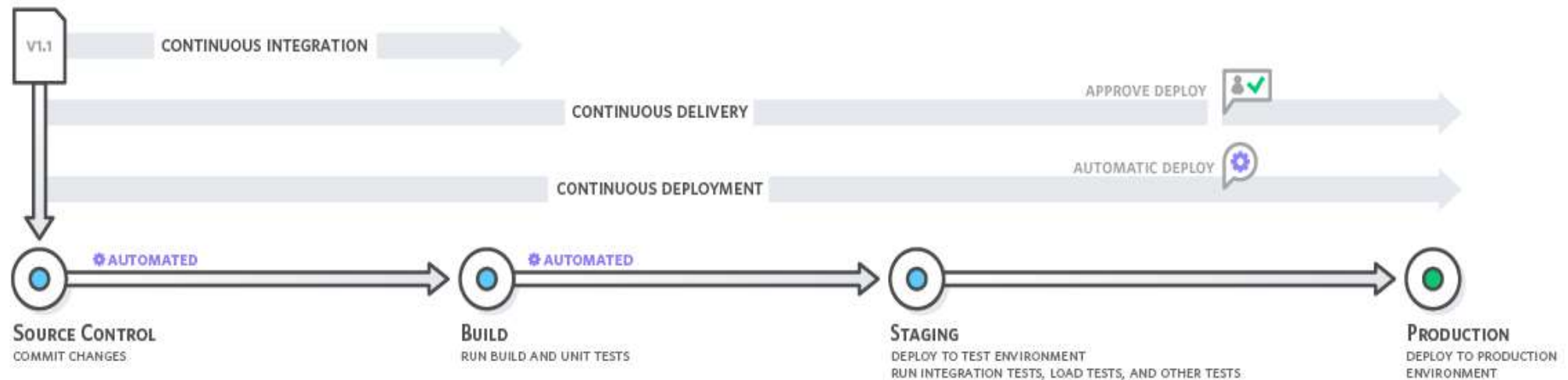
- Automates the deployment process
- May contain Automation testing
- Release code changes to production environment

Release Process Stages



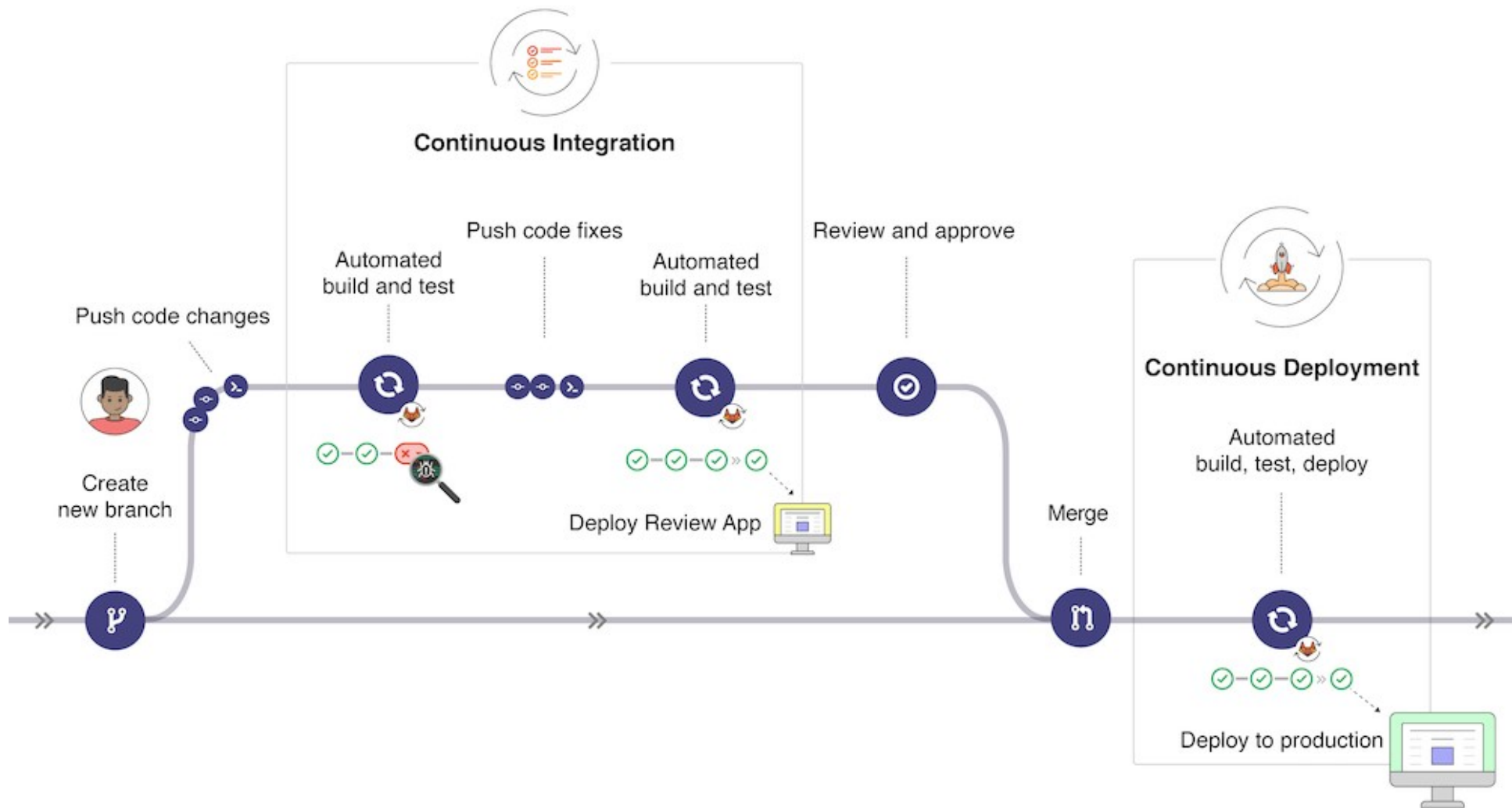
CI / CD Pipeline

A CI/CD pipeline is a series of steps that must be performed in order to deliver a new version of software



<https://aws.amazon.com/devops/continuous-integration/>

CI / CD Pipeline



CI/CD pipeline for Monolith vs Microservices

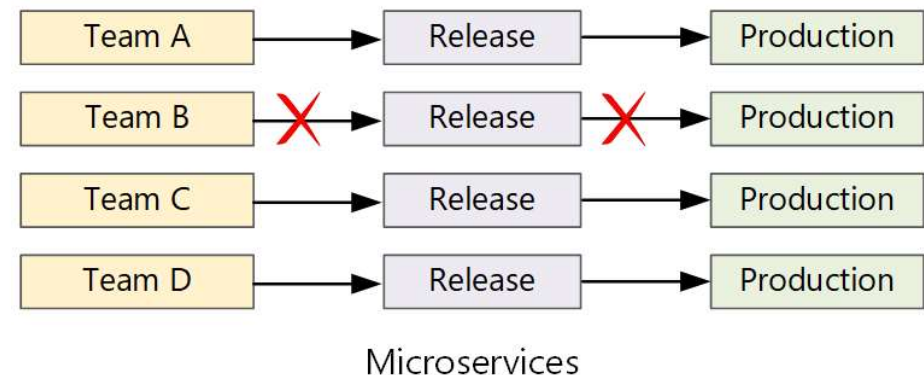
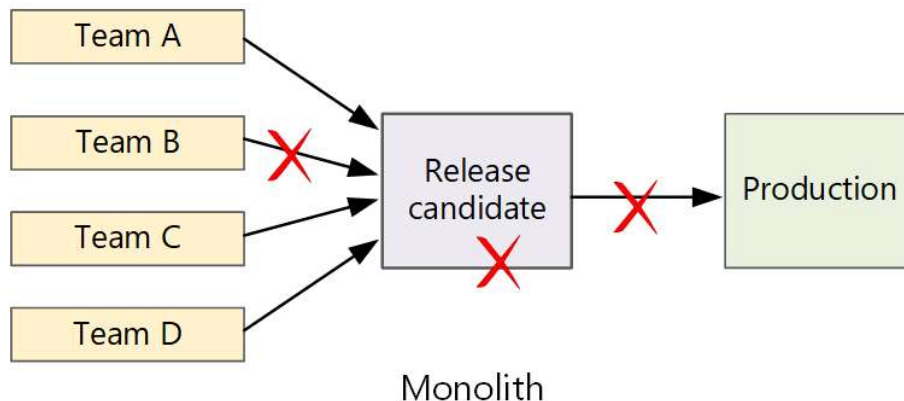


Monolith

- Single Build Pipeline
- Bug fixing delays release of features

Microservice

- One service, one pipeline
- High release velocity and reliability



<https://learn.microsoft.com/en-us/azure/architecture/microservices/ci-cd>



CI/CD Process for Microservices

- Each team can build and deploy the services that it owns independently, without affecting or disrupting other teams.
- Before a new version of a service is deployed to production, it gets deployed to Dev/Test/QA environments for validation. Quality gates are enforced at each stage.
- A new version of a service can be deployed side by side with the previous version.
- Sufficient access control policies are in place.
- For containerized workloads, you can trust the container images that are deployed to production.





DevOps - Tools

Popular Tools

- Jenkins
- GitLab
- GitHub Actions
- Azure DevOps
- AWS DevOps [CodeCommit, CodeBuild, CodeDeploy, CodePipeline services]
- Spinnaker - CD Platform for Multi-cloud environment
- CircleCI, TravisCI, GoCD etc.

Tools that appear in CI/CD workflow










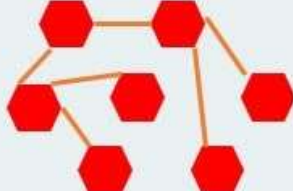
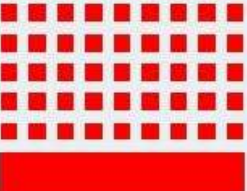

- Infrastructure automation - Ansible, Chef, Puppet, AWS CloudFormation, AWS SAM
- Container Runtime - Docker
- Container Orchestration - Kubernetes

Benefits of DevOps – CI/CD



Evolution of Process, Architecture, Application Development



	Development Process	Application Architecture	Deployment and Packaging	Application Infrastructure
~ 1980	Waterfall 	Monolithic 	Physical Server 	Datacenter 
~ 1990				
~ 2000	Agile 	N-Tier 	Virtual Servers 	Hosted 
~ 2010				
Now	DevOps 	Microservices 	Containers 	Cloud 



BITS Pilani

Pilani Campus

APIs and Gateways

API



- Application Programming Interface
- API is a contract between a service and its clients
- The nature of the API definition depends on which IPC mechanism you're using.
- APIs invariably change over time as new features are added, existing features are changed, and old features are removed
- Microservices architecture adopts API-First approach
- Ex: Google Map API



Introduction



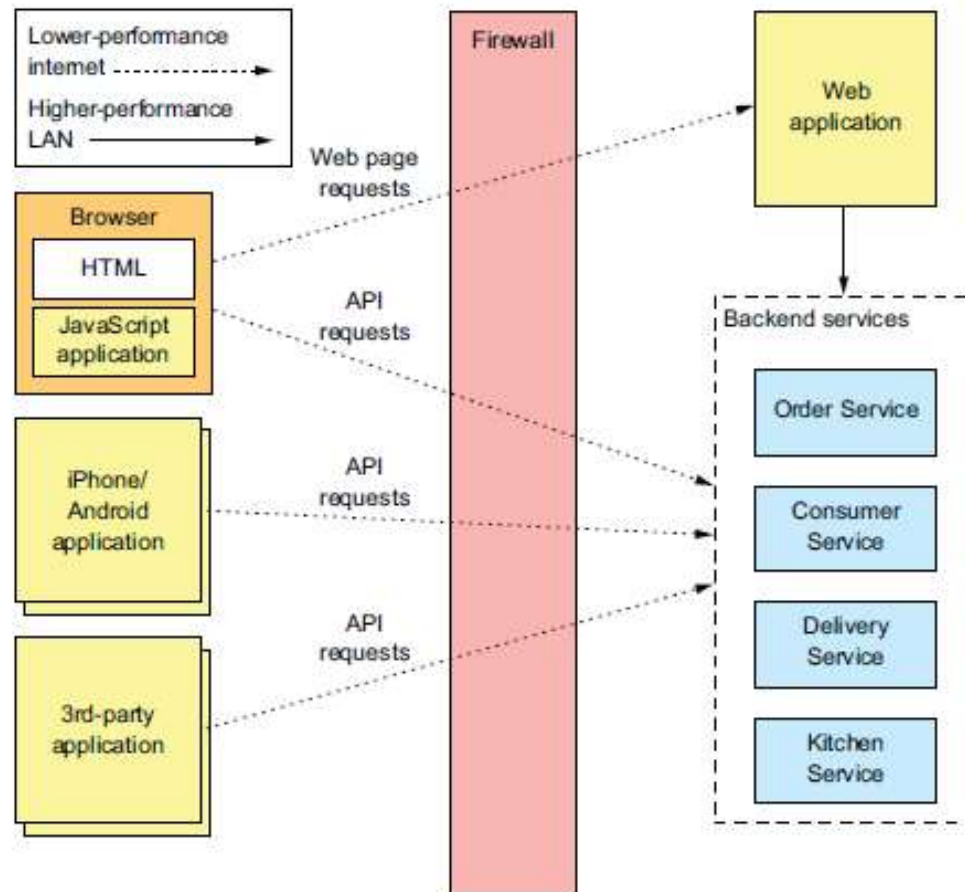
Let's imagine that you are developing a native mobile client for a shopping application.

The product details page displays a lot of information

- Number of items in the shopping cart
- Order history
- Basic Product Information
- Customer reviews
- Low inventory warning
- Shipping options
- Various suggested items



Approach 1 - Direct Client-to-Service communication



- Clients outside the firewall access the services over the lower-performance internet/mobile network
- Clients inside the firewall use a higher performance LAN

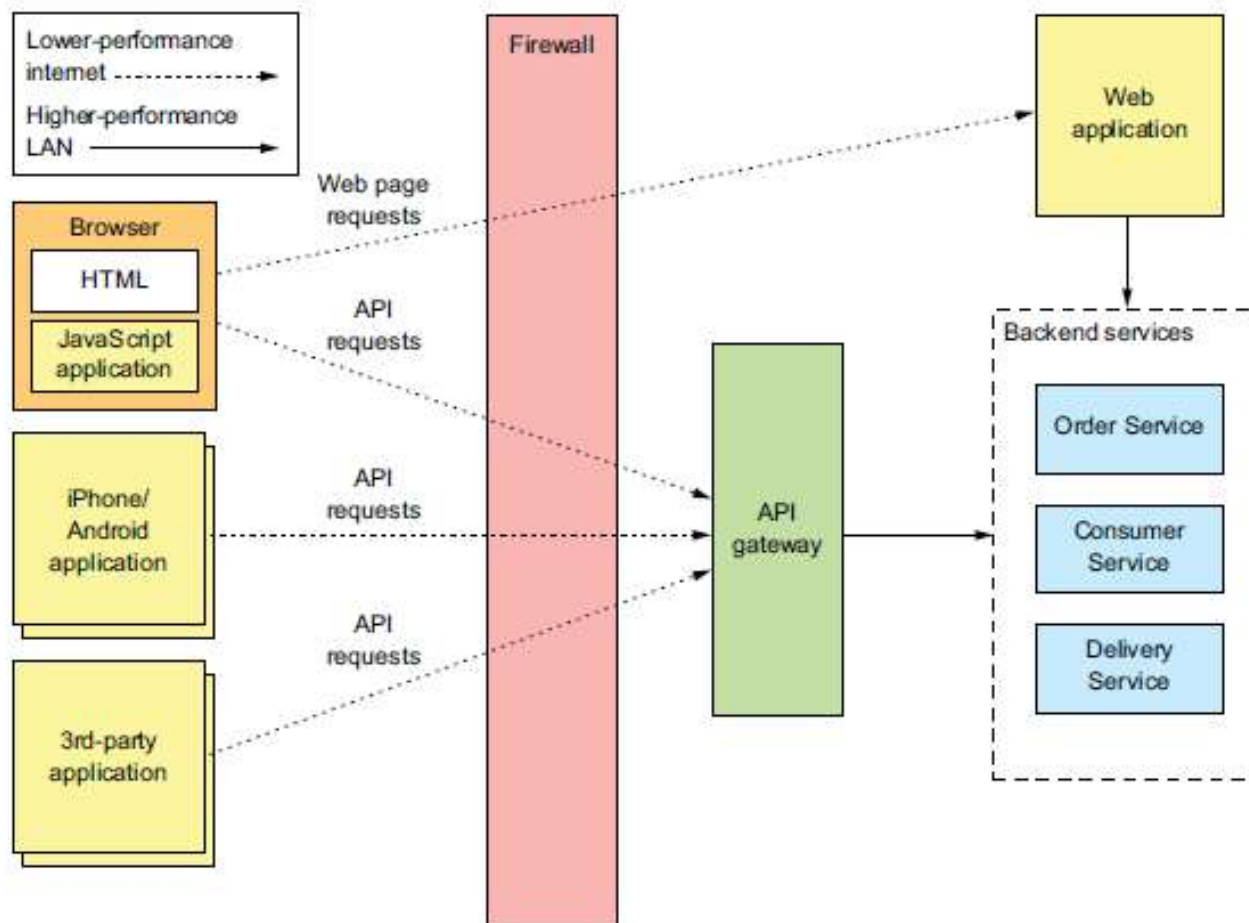
Solution – Introduce a Boundary Layer



- A boundary layer provides a façade over the complex interactions of your internal services.
- The boundary layer provides an abstraction over internal complexity and change.
- The boundary layer provides access to data and functionality using a transport and content type appropriate to the consumer.

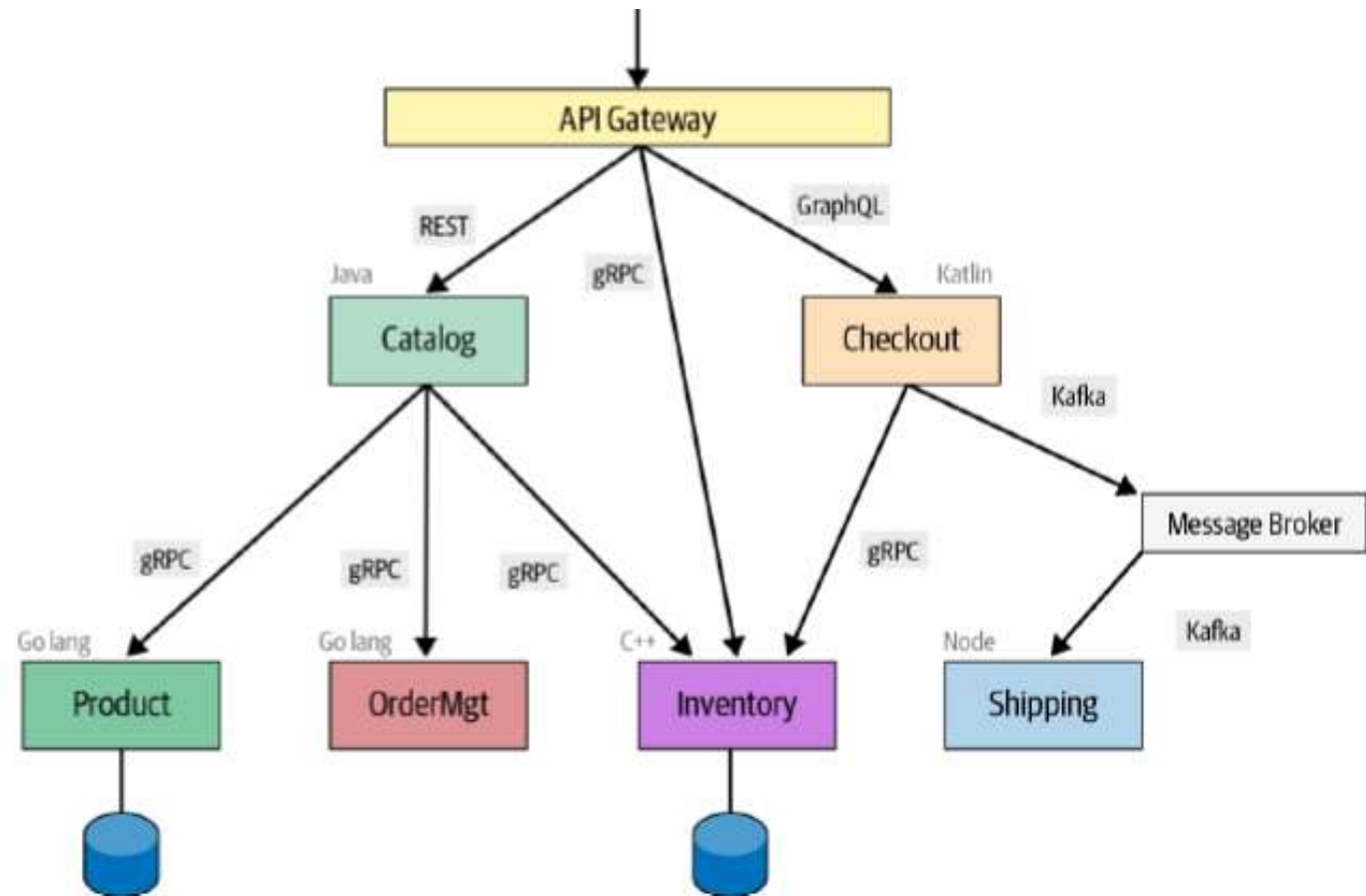
Pattern : API Gateway

Implement a service that's the entry point into the microservices-based application from external API clients.



Key features of API Gateway

- Request Routing
- Protocol Translation
- API Composition
- Implementing Edge Functions
- Authentication and authorization
 - Rate limiting
 - Caching
 - Collect logs and metrics



API Gateway Architecture

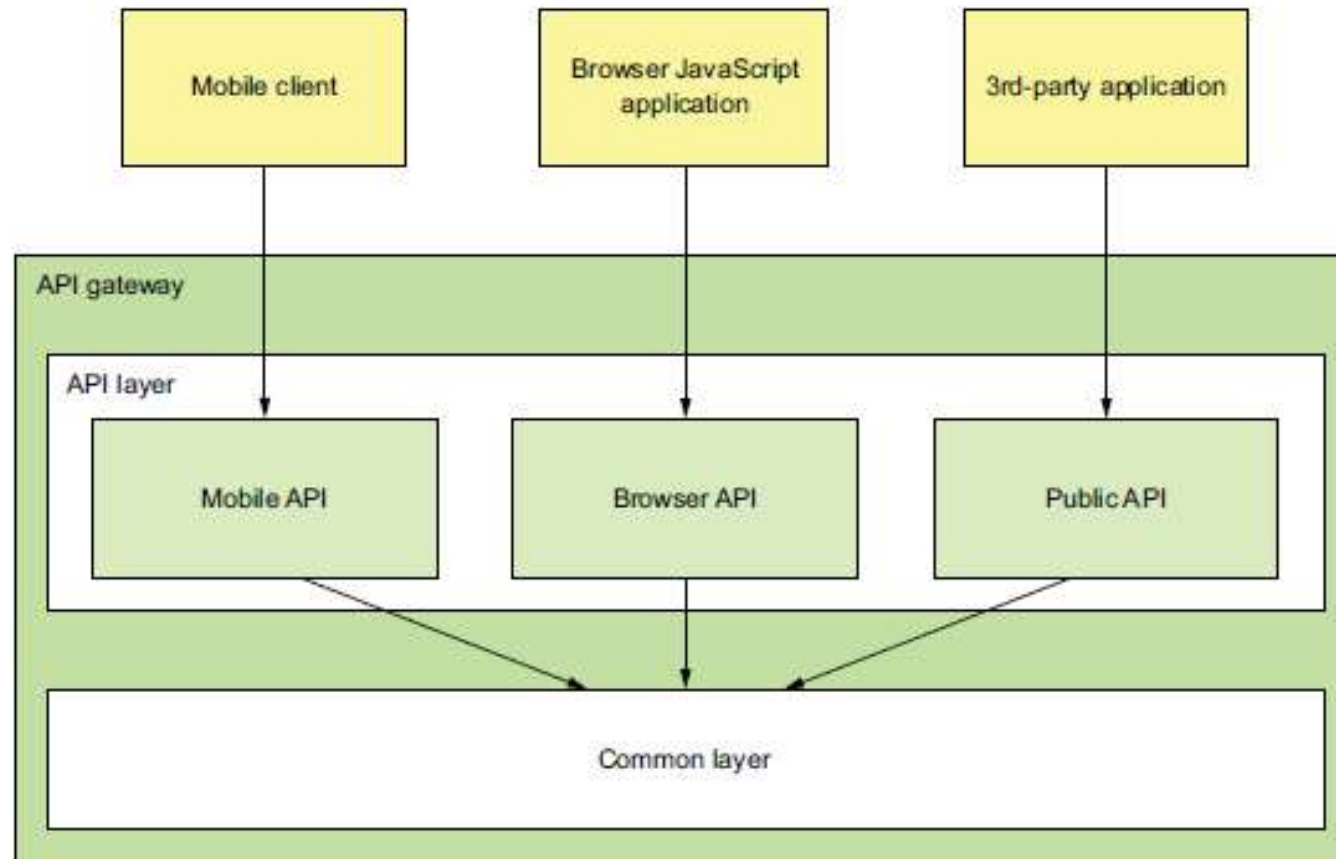


Figure 8.5 An API gateway has a layered modular architecture. The API for each client is implemented by a separate module. The common layer implements functionality common to all APIs, such as authentication.

Benefits and Drawbacks of API Gateway



Benefits

- It encapsulates internal structure of the application.
- It also simplifies the client code

Drawbacks

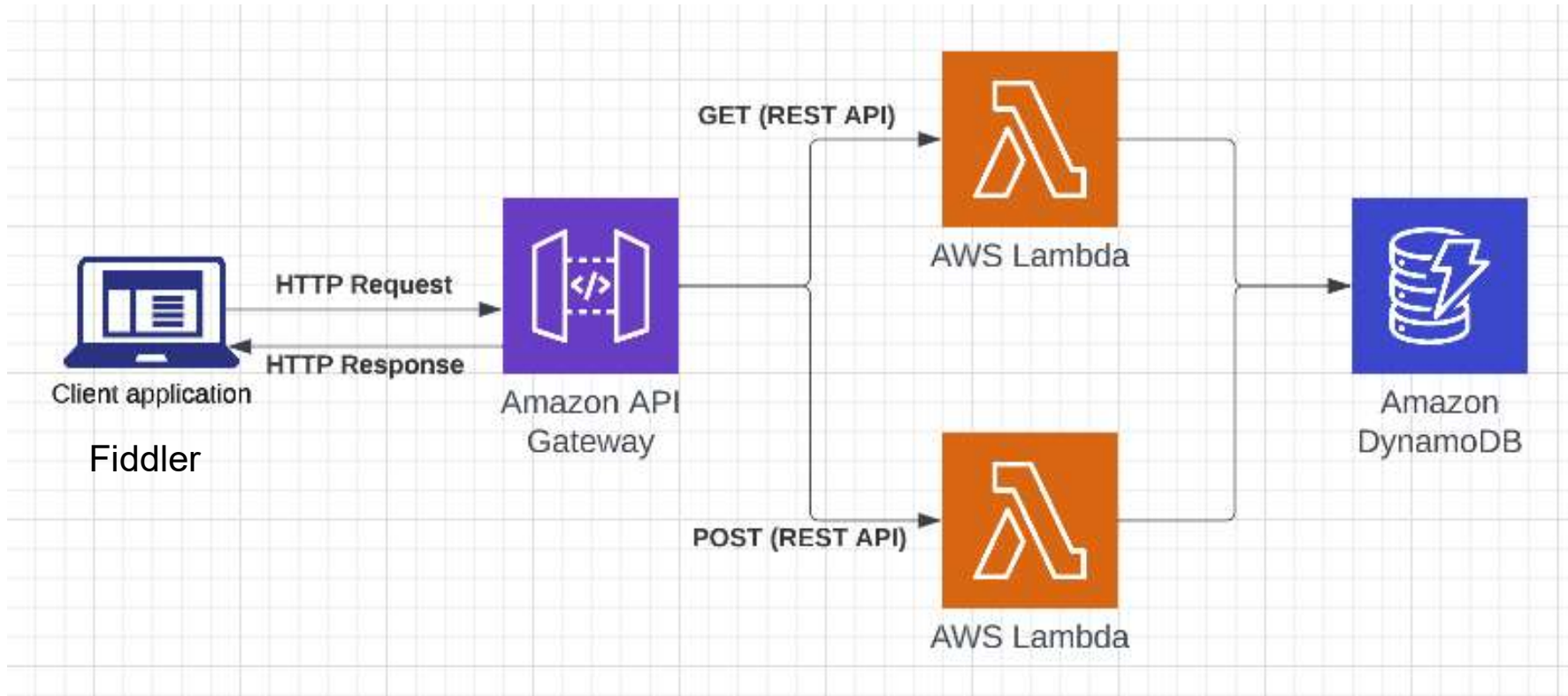
- Another component that must be developed, deployed, and managed.
- API gateway can become a development bottleneck



Implementing an API Gateway

- Using an off-the-shelf API gateway - AWS API Gateway, Kong
- Developing your own API gateway using either an API gateway framework or a web framework -NETFLIX ZUUL

Demo of API Gateway implementation in AWS





BITS Pilani

Pilani Campus



Deployment, Scaling and Availability: Custom, Managed, Containers

What is Scalability



- Scalable is the term used to describe software systems that can accommodate growth
- It is a non-functional requirement
- Capability to handle growth in “some dimension” of its operations

Operational dimensions can include

- number of simultaneous users or requests a system can process
- amount of data a system can effectively process and manage

Scale Up vs Scale Down



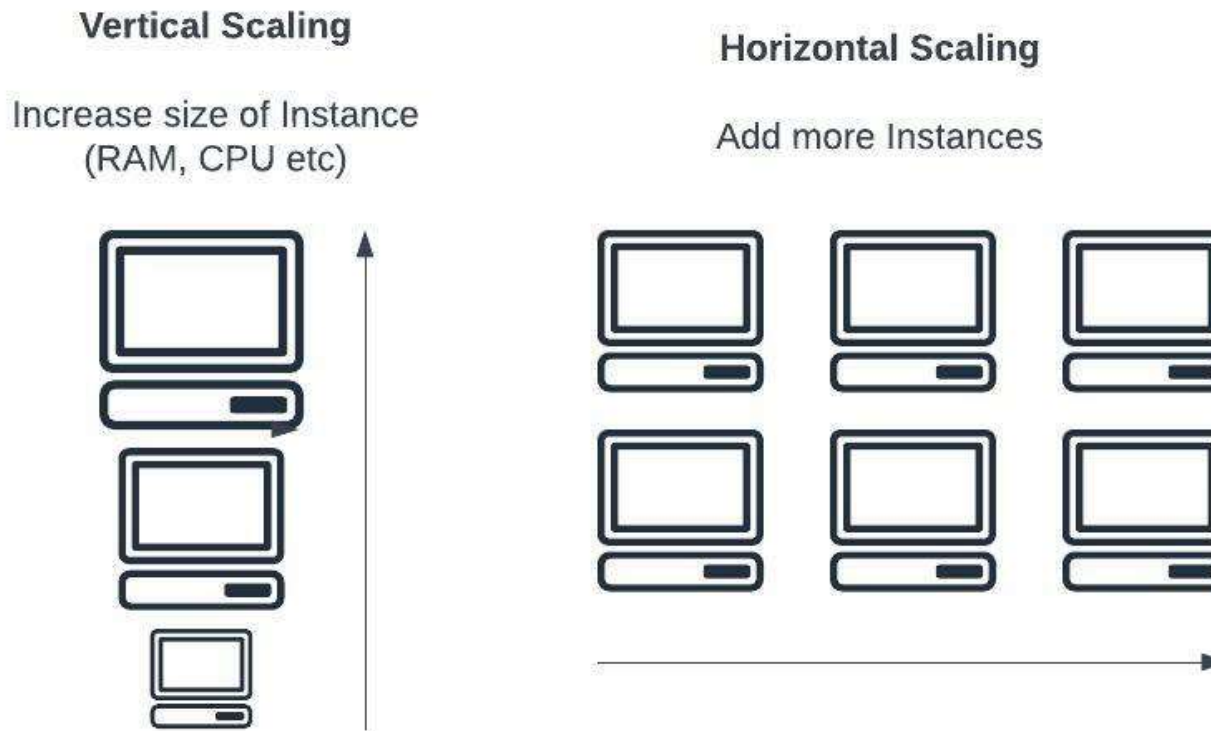
- Scaling Up refers to increasing the size or capacity of a system
- Scaling Down refers to reducing the size or capacity of a system

Ex: Adding more Physical Servers, RAM, CPUs, Virtual Machines, Containers etc.

Vertical vs Horizontal Scaling



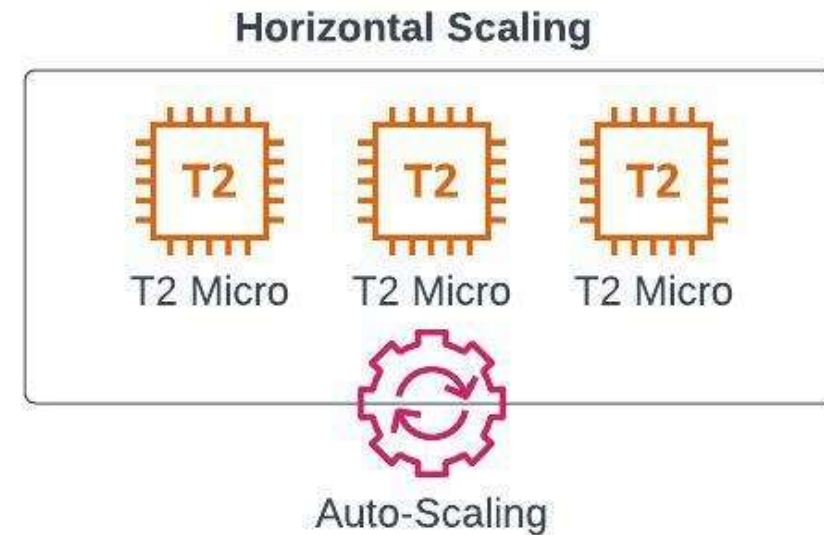
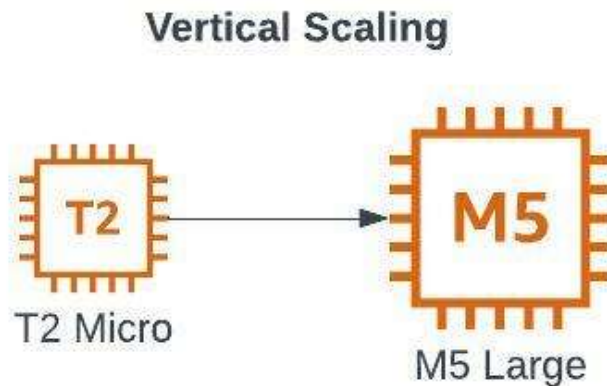
- Vertical Scaling (Scaling Up) is defined as increasing a single machine's capacity with the rising resources in the same logical server or unit
- Horizontal Scaling (Scaling Out) is an approach to enhance the performance of the server node by adding new instances of the server to distribute the workload equally



Vertical vs Horizontal Scaling



- Vertical Scaling - Adding more power to an existing ec2 instance
- Horizontal Scaling - Adding more ec2 instances



Deployment Patterns for Applications on Cloud

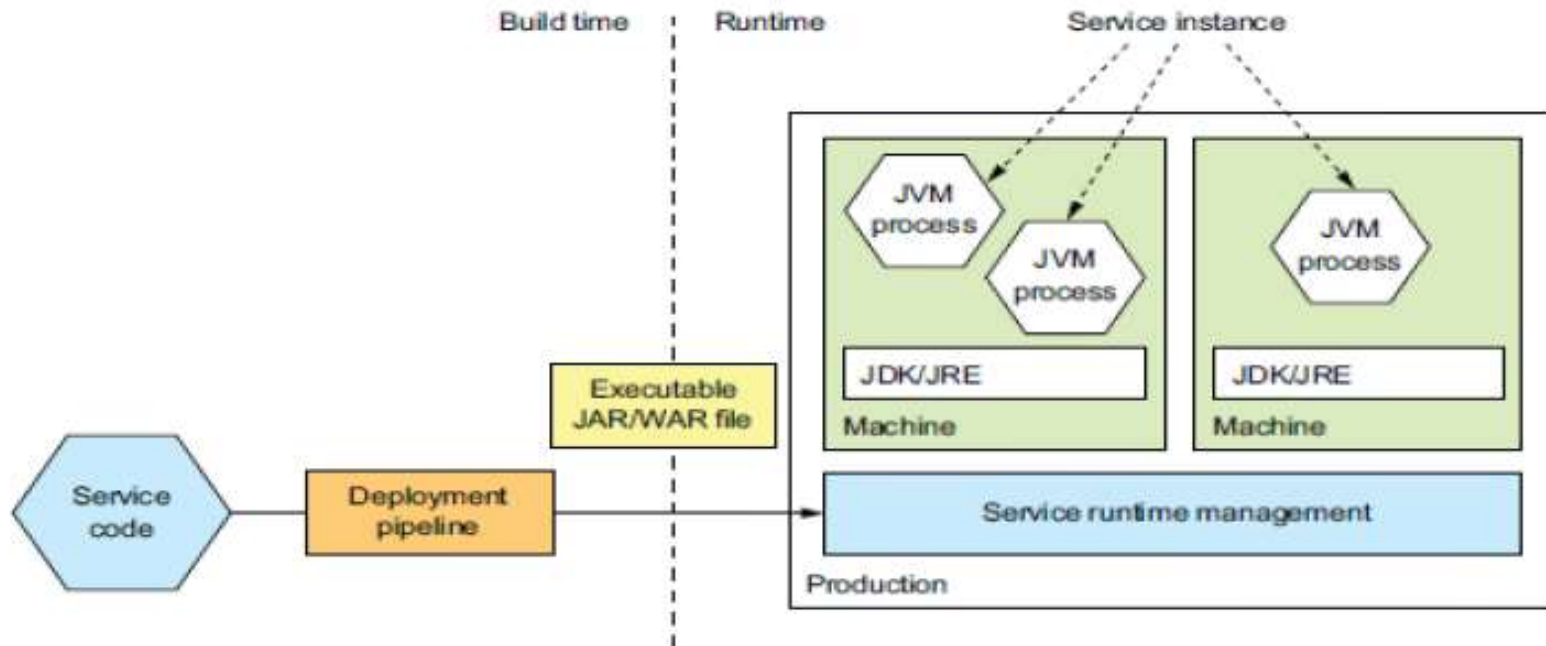


1. Language specific packaging format pattern
2. Service as a Virtual Service pattern
3. Service as a Container pattern

1. Language specific packaging format pattern



1. **Deploying services using the Language-specific packaging format pattern:** Deploy a language-specific package into production.



What if each service is developed in different programming language?? Other than Java?



2. Service as a Virtual Machine Pattern

- Package the service as a virtual machine image and deploy each service instance as a separate VM (Ex: T1.Micro EC2 instance create as VMI)
- Ex: Netflix packages each service as an EC2 AMI (Amazon Machine Image) and deploys each service instance as a EC2 instance.

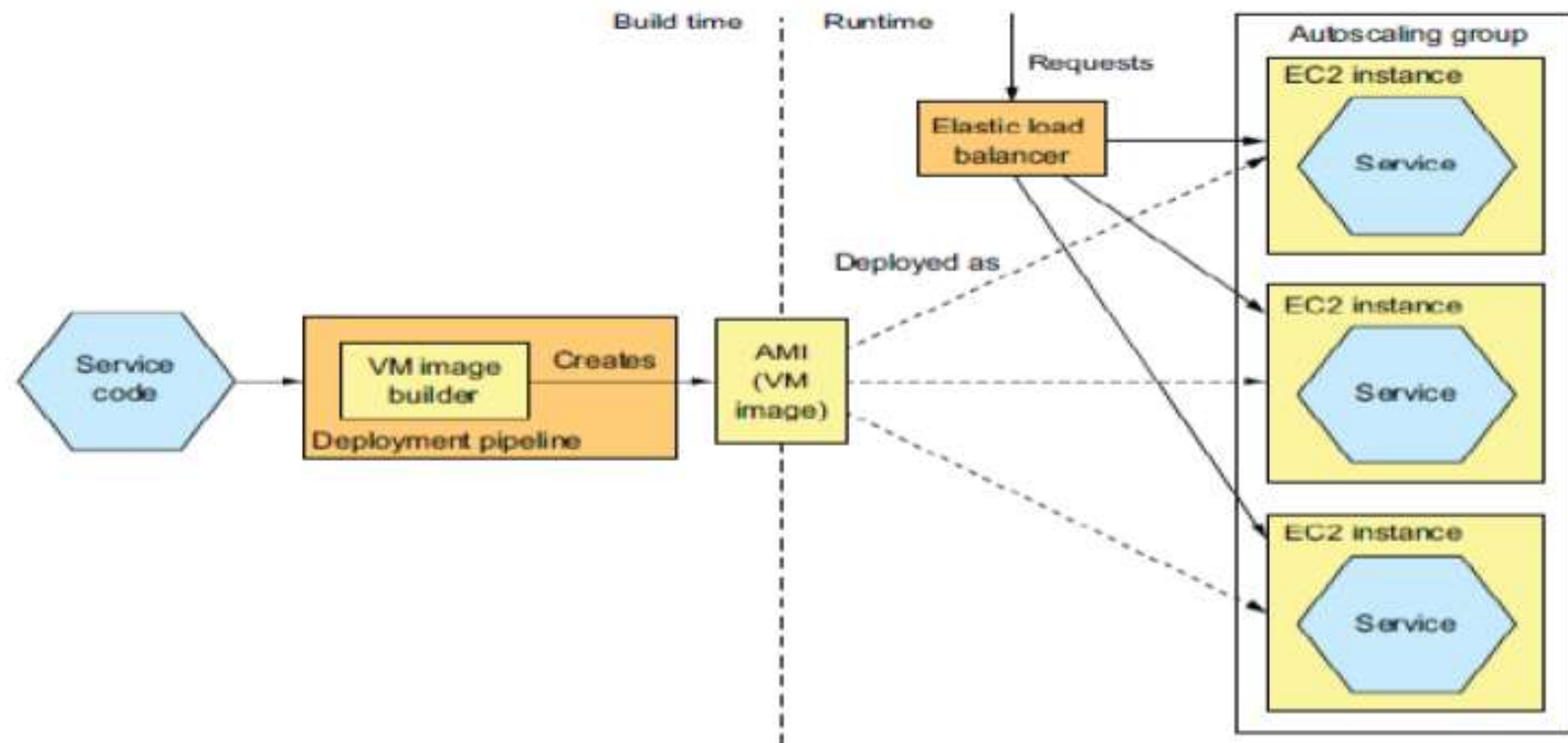
How to create an AMI from EC2 instance:

<https://docs.aws.amazon.com/toolkit-for-visual-studio/latest/user-guide/tkv-create-ami-from-instance.html>

<https://microservices.io/patterns/deployment/service-per-vm.html>

2. Service as a Virtual Machine Pattern

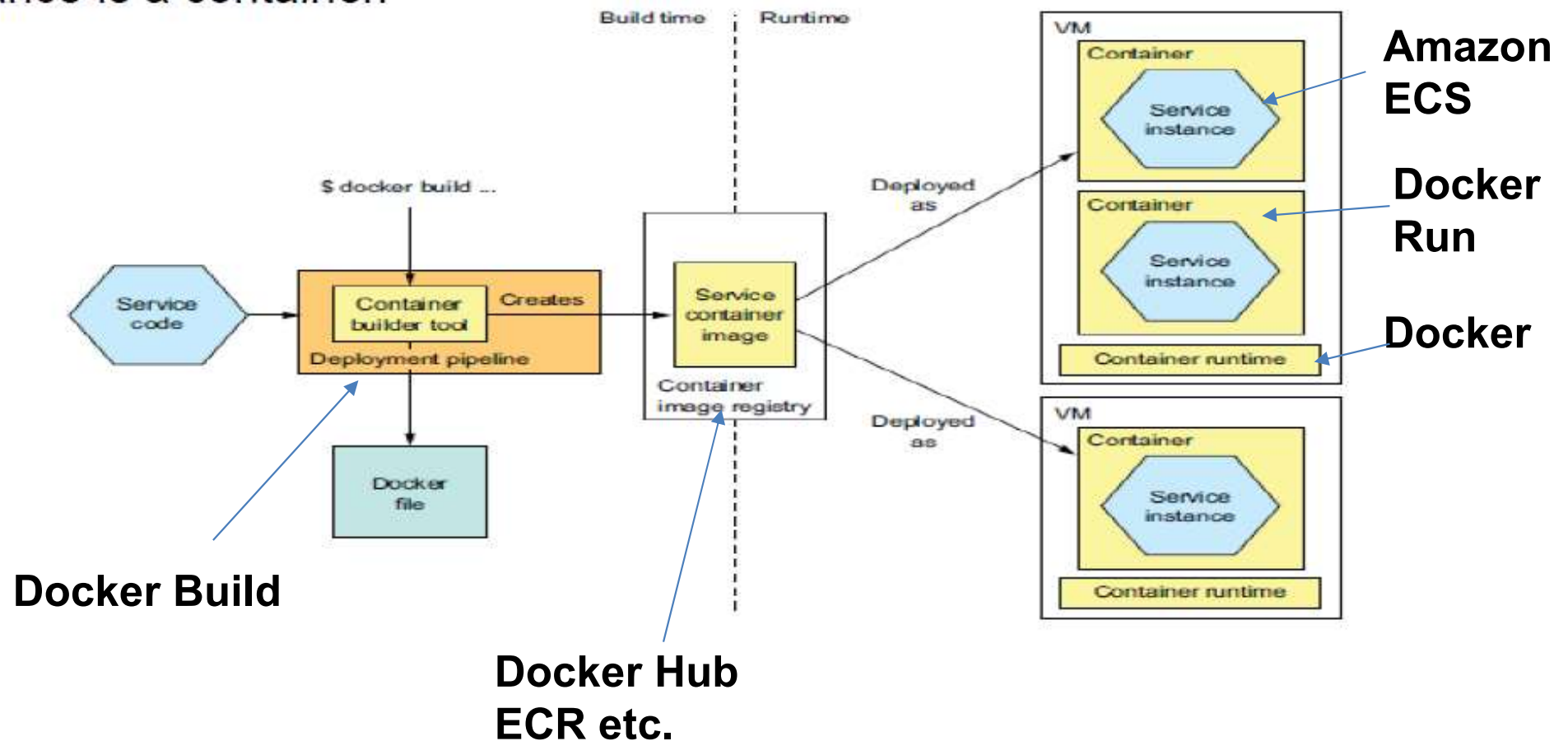
Deploy services packaged as VM images into production. Each service instance is a VM.



<https://microservices.io/patterns/deployment/service-per-vm.html>

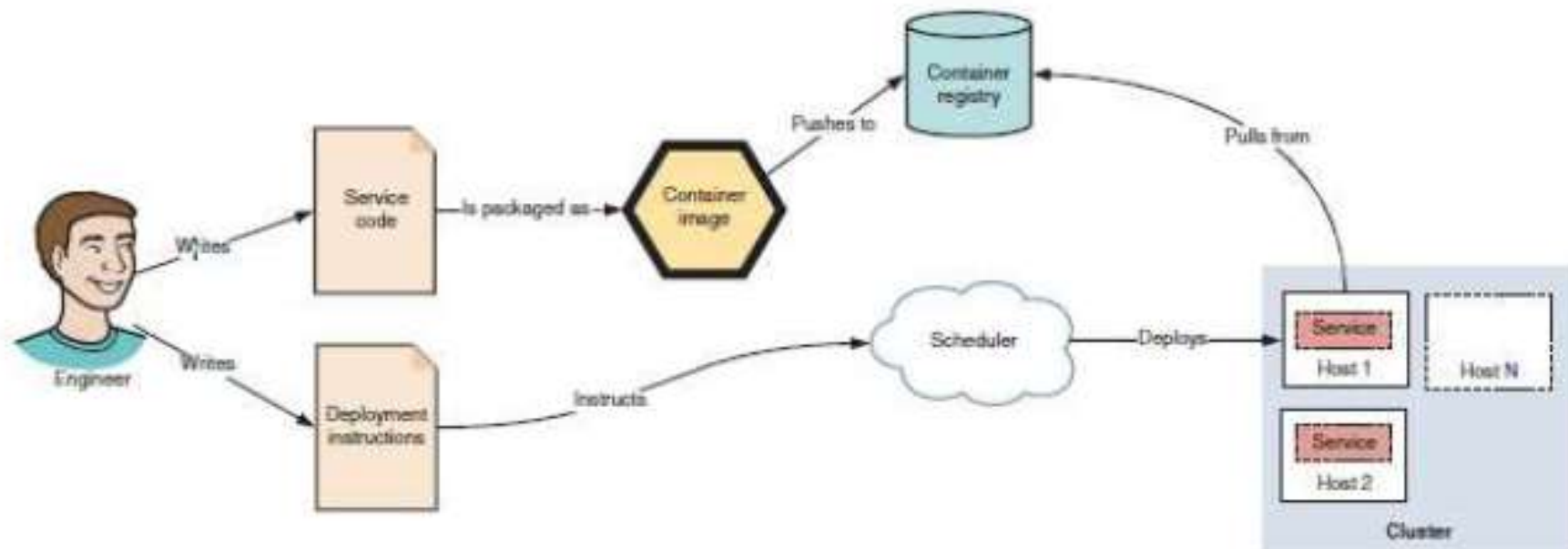
3. Service as a Container Pattern

Deploy services packaged as container images into production. Each service instance is a container.



3a. Deploy Services using Docker

- Build an image for a service
- Run multiple instances or containers of your image
- Push your image to a shared repository, or registry





Service to Host Models

Three common models for deploying services to underlying hosts:

- Single service to host
- Multiple services to host
- Multiple scheduled services per host/container scheduling.

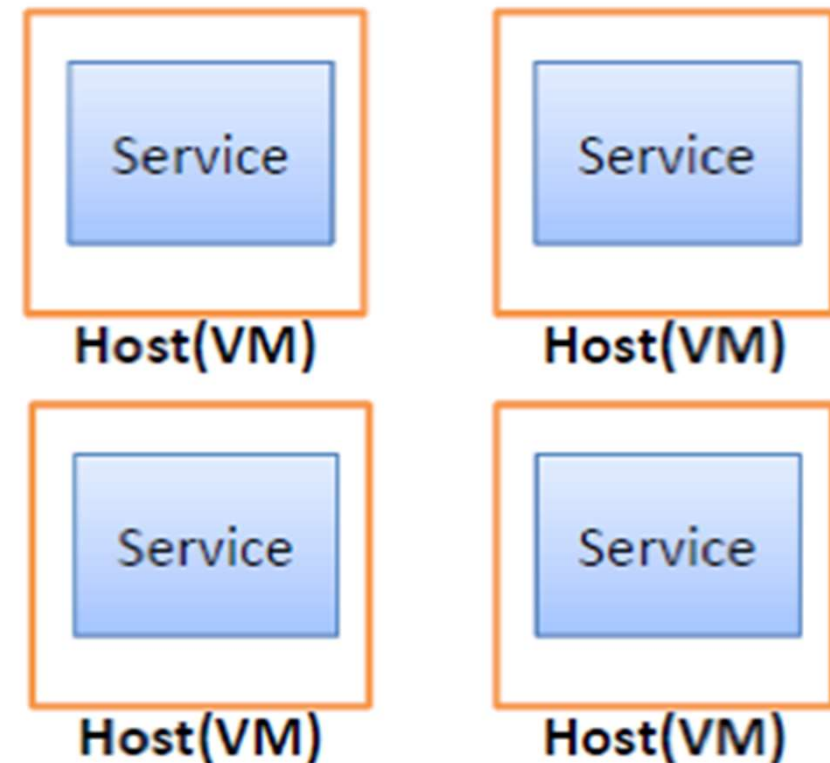
Single Service Instance per Host Pattern

Benefits

- Service instance Isolation
- Easy to manage, monitor and deploy
- No conflicting resource requirements

Drawbacks

- Less efficient resource utilization



Multiple Service Instances per Host Pattern

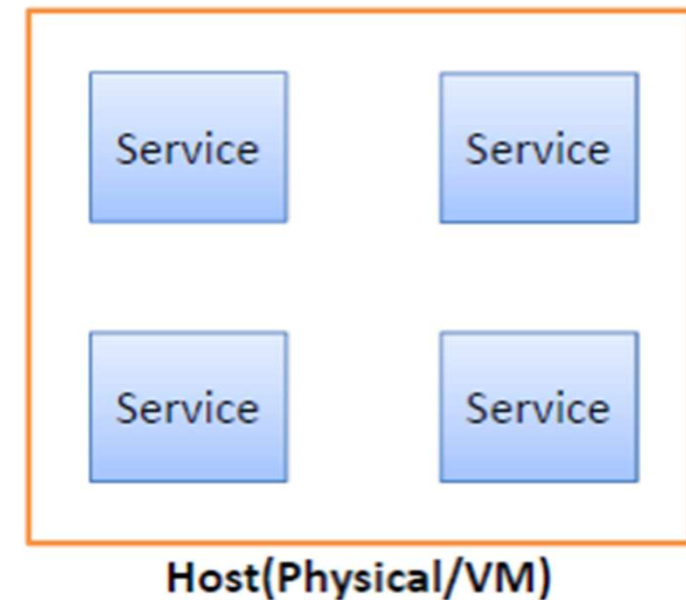
Deploying multiple services to a single host in production is synonymous with deploying multiple services to a local dev workstation or laptop.

Benefits

- Attractive from a host management point of view
- Cost

Challenges

- Monitoring
- Isolation of instances
- Resource consumption

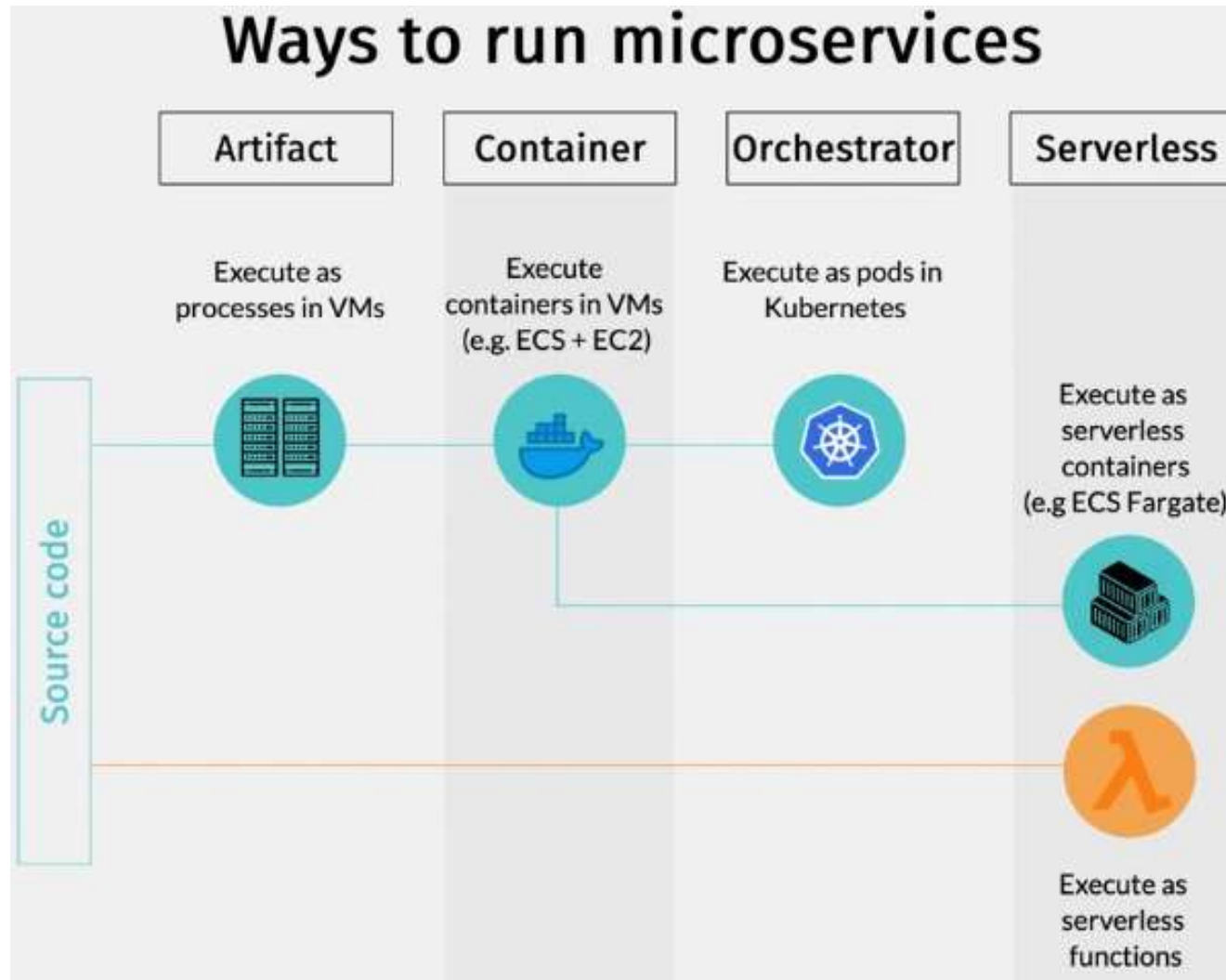




Multiple scheduled services per host

Container scheduler is a software tool that abstracts away from underlying hosts by managing the execution of atomic, containerized applications across a shared pool of resources

Ways to run Microservices



<https://semaphoreci.com/blog/deploy-microservices>



Ways to run Microservices

From simple to complex, here are the five ways of running microservices:

1. Single machine, multiple processes: buy or rent a server and run the microservices as processes.
2. Multiple machines, multiple processes: the obvious next step is adding more servers and distributing the load, offering more scalability and availability.
3. Containers: packaging the microservices inside a container makes it easier to deploy and run along with other services. It's also the first step towards Kubernetes.
4. Orchestrator: orchestrators such as Kubernetes are complete platforms designed to run thousands of containers simultaneously.
5. Serverless: serverless allows us to forget about processes, containers, and servers, and run code directly in the Cloud [Ex: AWS Lambda function].

<https://semaphoreci.com/blog/deploy-microservices>



BITS Pilani

Pilani Campus



Scaling Strategies for Applications on Cloud



Scaling Strategies for Apps on Cloud

Five approaches

- 1.Vertically Scaling the entire Cluster
- 2.Horizontally Scaling the entire Cluster
- 3.Horizontally Scaling Individual Microservices / Application
- 4.Elastically Scaling the entire Cluster
- 5.Elastically Scaling the Individual Microservices / Application

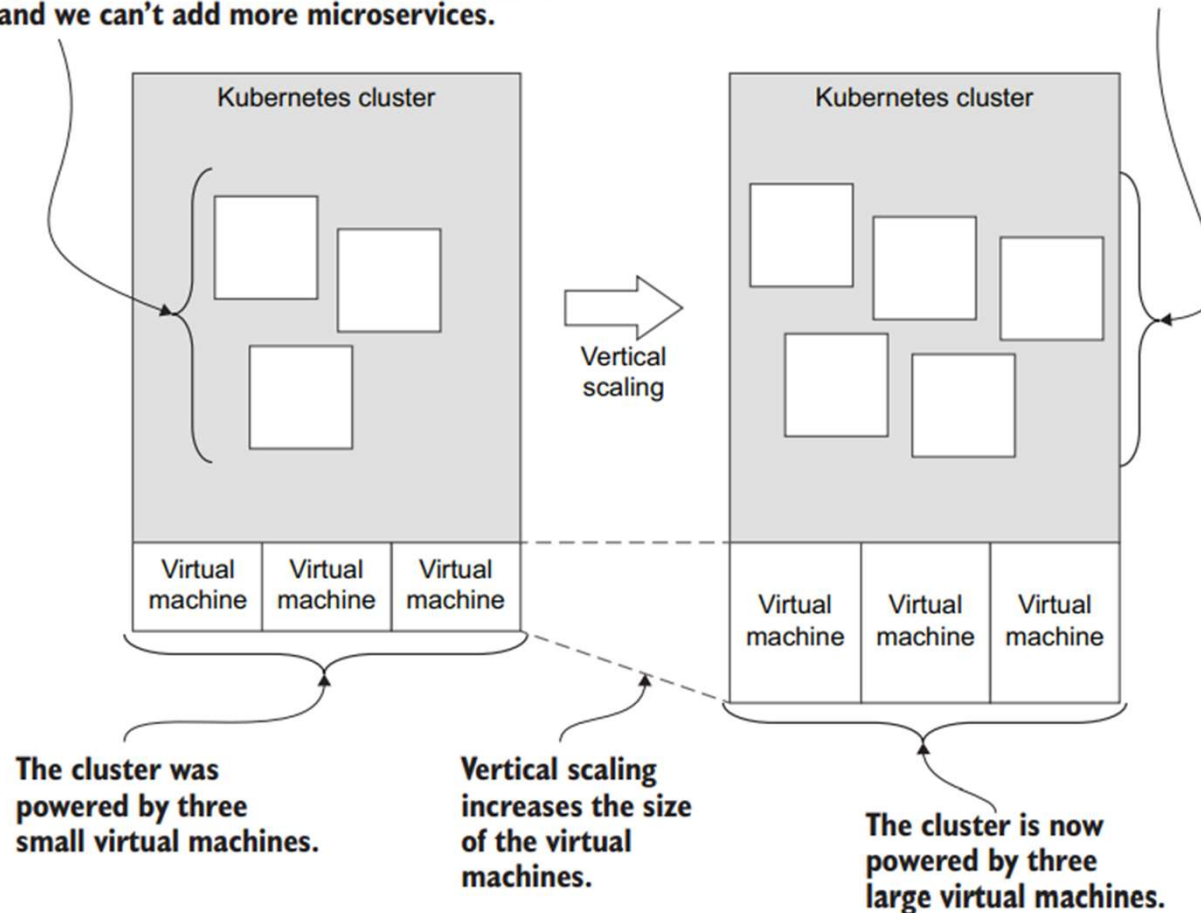
<https://thenewstack.io/scaling-microservices-on-kubernetes/>

1. Vertical Scaling of Cluster

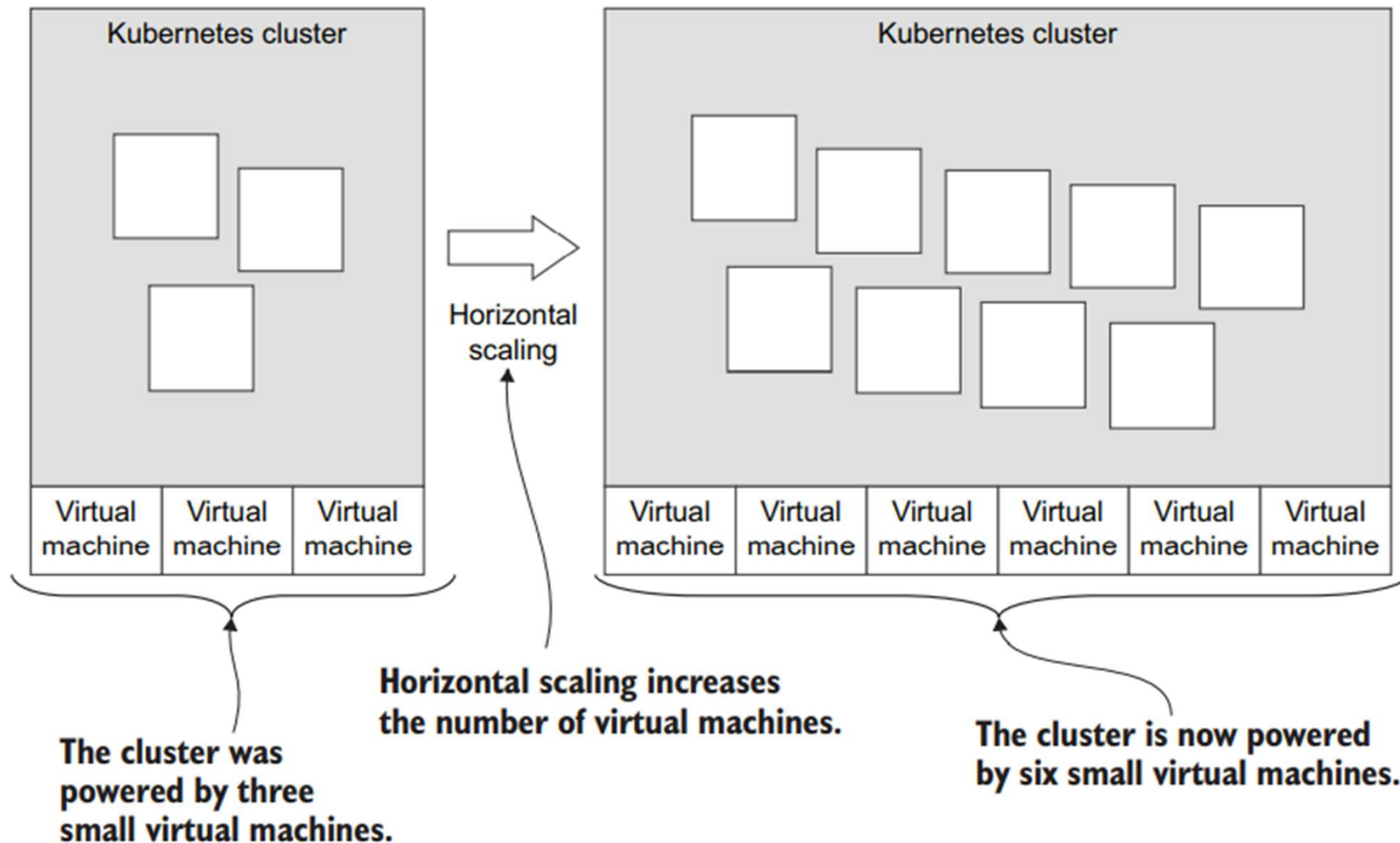
The number of microservices hosted by the cluster is limited by computing capacity.

At some point we will exhaust the capacity of our cluster and we can't add more microservices.

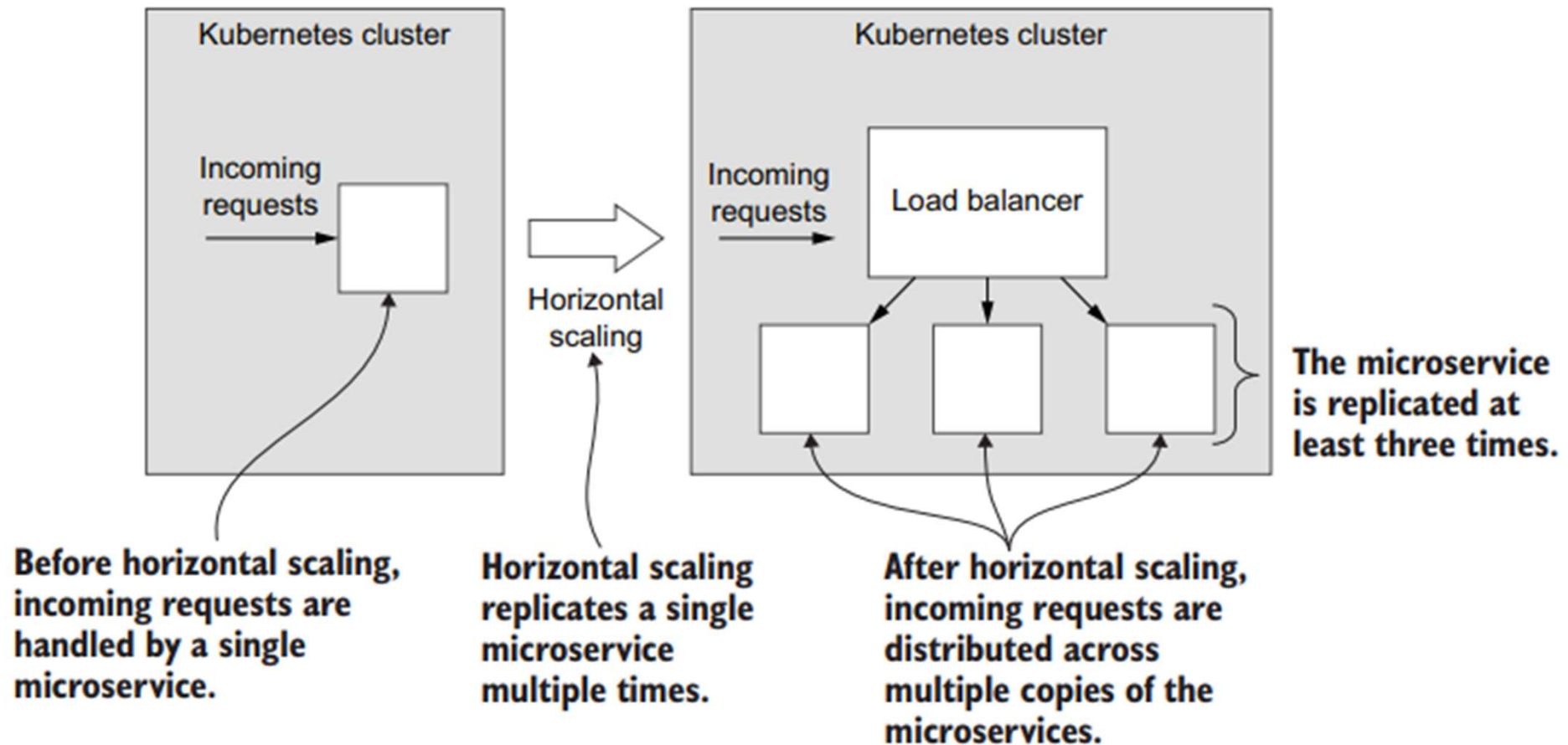
After vertical scaling we have more computing capacity and we can add more microservices.



2. Horizontal Scaling of Cluster



3. Horizontal Scaling of Individual Microservice



4. Elastic Scaling for the Cluster

This is a technique where we automatically and dynamically scale our cluster to meet varying levels of demand.

- Whenever a demand is low, Kubernetes can automatically deallocate resources that aren't needed.
- During high-demand periods, new resources are allocated to meet the increased workload.
- This generates substantial cost savings because, at any given moment, we only pay for the resources necessary to handle our application's workload at that time.

```
default_node_pool {  
  name = "default"  
  vm_size = "Standard_B2ms"  
  
  enable_auto_scaling = true  
  
  min_count = 3  
  
  max_count = 20  
}
```

Enables Kubernetes cluster autoscaling

**Sets the minimum node count to 3.
This cluster starts with three VMs.**

**Sets the maximum node count to 20.
This cluster can automatically scale
up to 20 VMs to meet demand.**

Enabling auto-scaling for cluster with Terraform

5. Elastic Scaling of Individual Microservice



The number of replicas for the microservice is expanded and contracted dynamically to meet the varying workload for the microservice (bursts of activity)

```
resource "kubernetes_horizontal_pod_autoscaler" "service_autoscaler" {  
  metadata {  
    name = var.service_name  
  }  
  
  spec {  
    min_replicas = 3  
    max_replicas = 20  
  
    scale_target_ref {  
      kind = "Deployment"  
      name = var.service_name  
    }  
  }  
}
```

Sets the range of instances for this microservice.
It starts at 3 instances and can scale up to 20 to
meet variable levels of demand.

Enabling auto-scaling of individual Microservice with Terraform



BITS Pilani

Pilani Campus



THANK YOU



-
1. Demo of API Gateway implementation in AWS Cloud
 2. Demo of Continuous Build and Deployment using AWS SAM (Serverless Application Model)

**Thank
You!**