Cloud Platforms

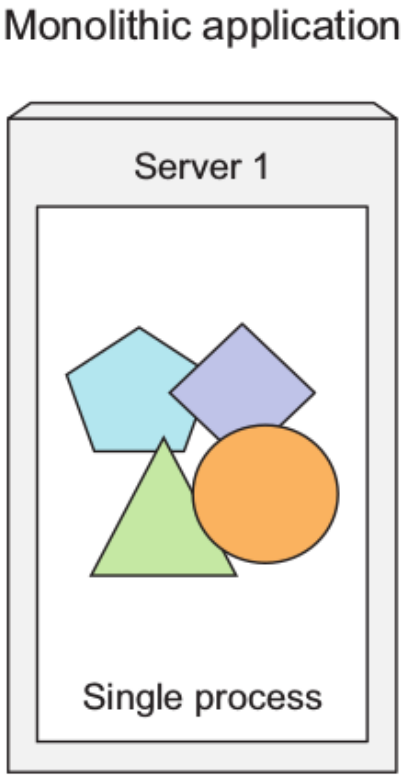
Lightweight cloud computing platforms for DevOps

Reference:

- ***‘Kubernetes in Action’ by Marko Luksa***

Software development - BACK

- Some years back, most software applications were big monolites, running as a single process or a small number of processes
- They have slow-release cycles and are updated infrequently since the development is complex and unpractical:
  - Developers usually package up the whole system and hand it over to the ops team, the team of system administrators responsible for managing the infrastructures (the servers)
- Changes to one part of the application requires a redeployment of the whole application
- Over time the lack of boundaries between the software components increased the complexity of the development and maintenance, thus deteriorating the quality of the whole application
- Monolithic applications usually requires a small number of power servers

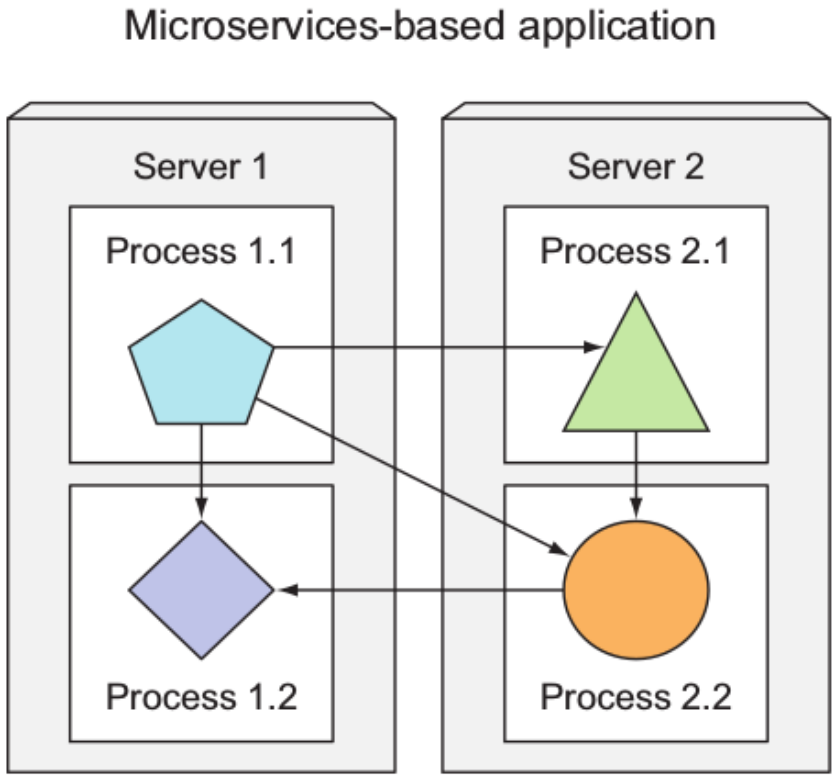


Legacy Systems

- Those systems – named legacy systems – are still up and running today (even though they are slowly replaced)
- If convenient they are cloudified, it means that physical servers are replaced by VMs
- This process simplify the maintenance of the infrastructure, the software development process and maintenance is still complex
- When the application has to deal with an increasing load, resources are added to the servers, i.e. vertical scalability is adopted, as it does not require modification to the original architecture of the application
- Vertical scaling, however, has limits and it is not always feasible

Software development - NOW

- Cloud computing radically changed the way software applications are developed and deployed
- Big monolithic legacy applications are now ***broken down into independent components***, named services or microservices
- Services are decoupled from each other, and they can be developed, deployed, updated and scaled individually
- Each service/microservice runs as an independent process and communicates with the others via well-defined interfaces



Service independency

- This structure enables a more rapid application lifecycle: components can be changed quickly and easily, as often as necessary to keep up with rapidly changing business requirements
- Services communicate through synchronous protocols (e.g. SOAP, HTTP/RESTful) or asynchronously via message exchange (e.g. message queue). Such protocols are simple and well known by developers
- Services can be developed separately, a change in one does not require changes or the redeployment of other services (provided that the interface does not change)

Application development process

- This new architecture also changed the whole application development process and, in particular, how applications are managed and deployed in production
- In the past, the application lifecycle was managed by two different (usually separated teams):
  - The **development team (dev team)**, whose task was to design and program the applications software
  - The **operations team (ops team)**, which took care of deploying and maintaining the application
- *Now organizations are adopting a different approach: the same team that develops the application also take part in its deployment and its maintenance for its whole lifetime*
- ***This approach is called DevOps***

DevOps

- The main advantage of this approach is to have developers more involved in running the application in production
- This leads to a better understanding of both the users' needs and issues and also the problems in deploying the application
- The deployment process is streamlined, it means that releases can be performed more often: ideally you can have developers deploying their application themselves without having the ops team involved (continuous development)
- Such practice, however, requires developers to have a deep knowledge of the details of the underlying infrastructure, which is not always feasible or desirable

NoOps

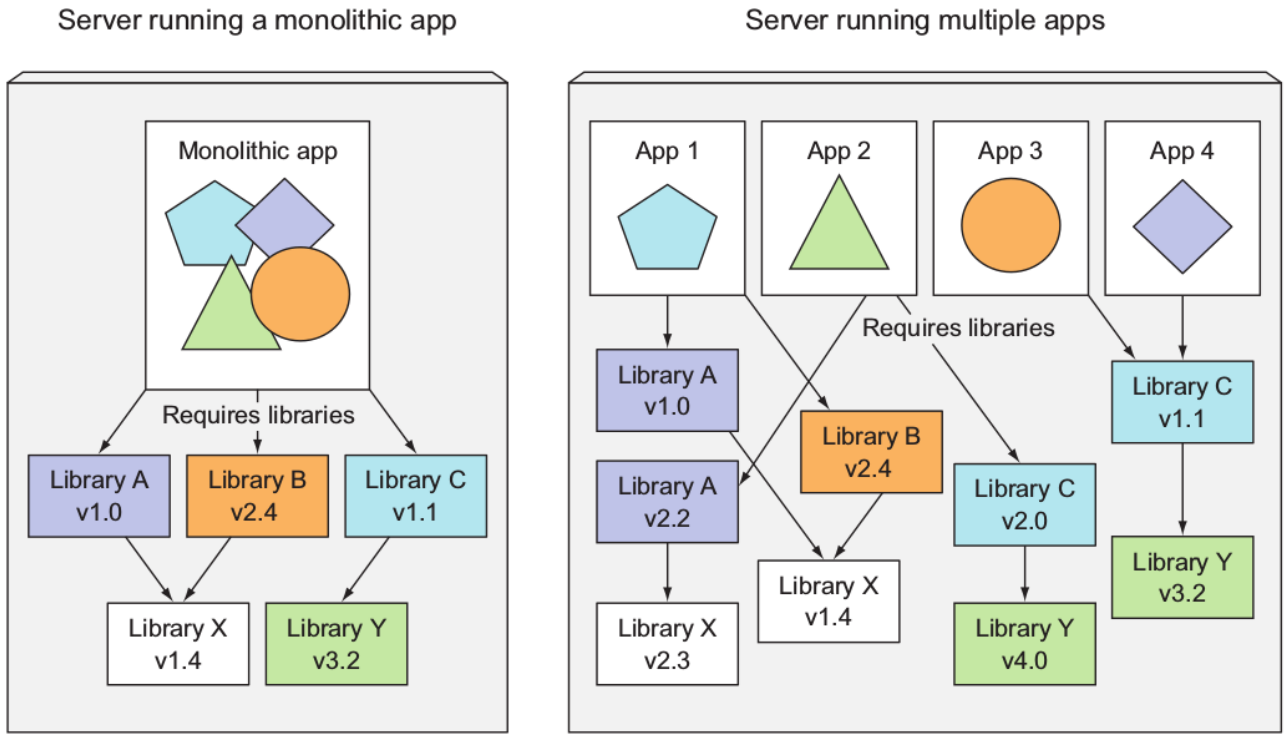
- Even though developers and system administrators work towards achieving the same goal, they usually have different background, experiences and motivations
- Even if the DevOps model is considered a target for its advantages, its implementation is hard as the members of the dev team hardly have the background and experience to take care of the infrastructure
- An ideal solution would be to put in practice a NoOps approach, in which developers deploy applications themselves without knowing anything about the infrastructure and without having to deal with the peculiarities of deploying an application
- *A solution to enable automatic service deployment and maintenance would make feasible to implement a DevOps or NoOps approach*
- You would still need to have a ops team to manage the infrastructure, but developers would be allowed to deploy their software easily

Service deployment and management

- Regardless of the knowledge required for application deployment and management, a service based architecture has other drawbacks
- When an application consists of a small number of components, their management is simple
- When the number of components increases, the complexity of deployment operations increases
- Manual configuration and deployment is tedious, error prone and unfeasible when the number of services reaches a certain number
- *A solution to automate the deployment and configuration of components is required to manage large applications with a large number of services*

Service Dependencies

- As mentioned, services are developed as separate components independently by different teams
- Due to their independence, it is common to have different teams using different libraries
- Each team select their own library, when the same library is the same it might happen that different versions are selected
- In addition to this, different teams want to have the freedom to replace the library whenever the need arises
- Deploying different services/application with crossed dependencies can be a nightmare for the ops team

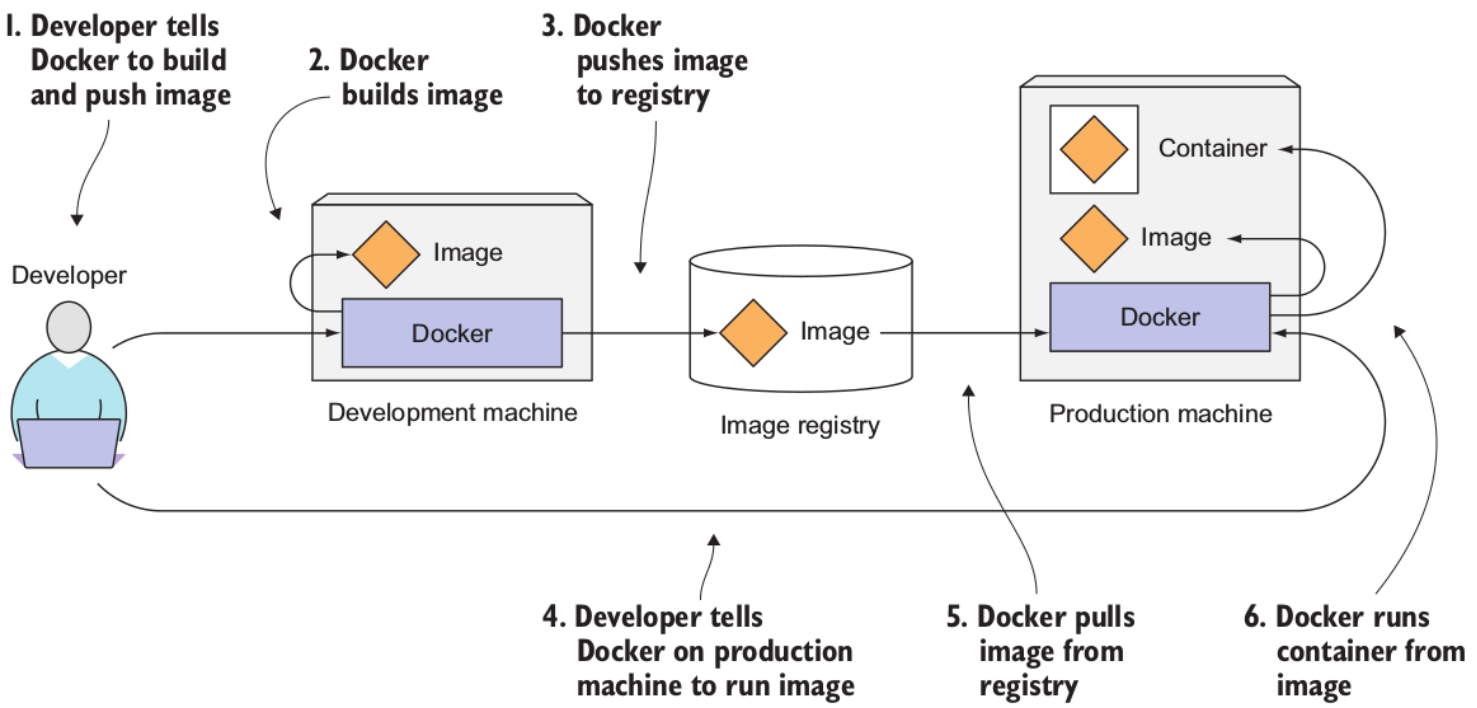


Consistent Application Environment

- Regardless of the number of individual components, one of the biggest problems that developers and operations teams have is to deal with the differences in the environment they run the applications
- *A method to simplify such operations is to deploy applications in an ideal confined environment where all the libraries are pre-installed and the real hardware on which the software is running is somehow abstracted*
- This environment can be created through full virtualization, i.e. running he software in a VM
- This solution, however, is expensive as full virtualization has a significant overhead (virtualization itself, guest OS, etc)
- In addition to this, using VMs has an overhead in terms of configuring the VM and its guest OS by configuring it and by installing the proper libraries

Lightweight Virtualization

- *Lightweight virtualization mechanisms, like containers, can be a solution to the high overhead required by full virtualization*
- Containers can be used to prepare a virtualized environment in which libraries and dependencies are pre-installed and configured to run a certain application or a service, which is part of an application
- In addition, containers can be easily shipped and installed on different machines using ad-hoc distribution systems that take care of distributing and installing images on different machines
- An example of this is Docker



Kubernetes

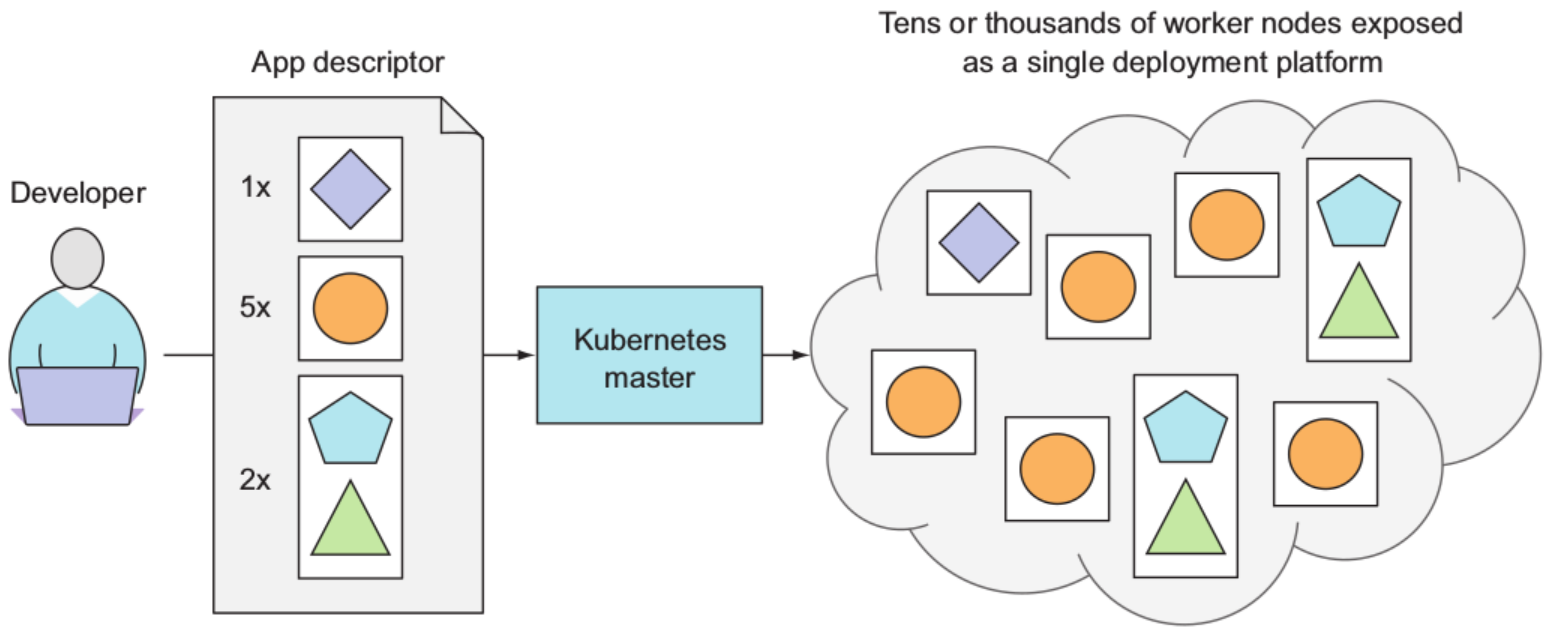
- Kubernetes is a software systems developed by google that allows to easily deploy and manage containerized applications on top of it
- A containerized application is an application whose components (independent services) are deployed and shipped as different containers
- Kubernetes aims at simplifying completely the overall process of deploying and managing the containers composing the application, taking care of aspects regarding scalability, failure tolerance, resource management, etc.
- The goal is to offer a *lightweight IaaS infrastructure to deploy applications in a simple and effective manner*, so developers can take care of it without having to deal many of the aspects
- The operation team is still required, in this case, however, it is responsible to maintain and run the Kubernetes infrastructure

Kubernetes Core Operations

- Kubernetes is a distributed platform that is composed of a **master** node and a varying number of **worker** nodes
- Kubernetes nodes can be installed directly on the physical servers or they can be installed on a cloud platform on top of VMs
- Kubernetes master node exposes an interface to developers, which can request to execute a specific application
- The developer has to submit an app descriptor which only includes the list of components that compose the app
- For each component the configuration is specified
- The Kubernetes master takes care of deploying the app components on the worker nodes according to the description provided by the developer in a complete automatic manner

Kubernetes Core Operations

- Through the interface the developer can specify that certain services must run together consequently they are developed on the same working node
- Others will be spread around the cluster according to the status of each working node



Kubernetes core operations

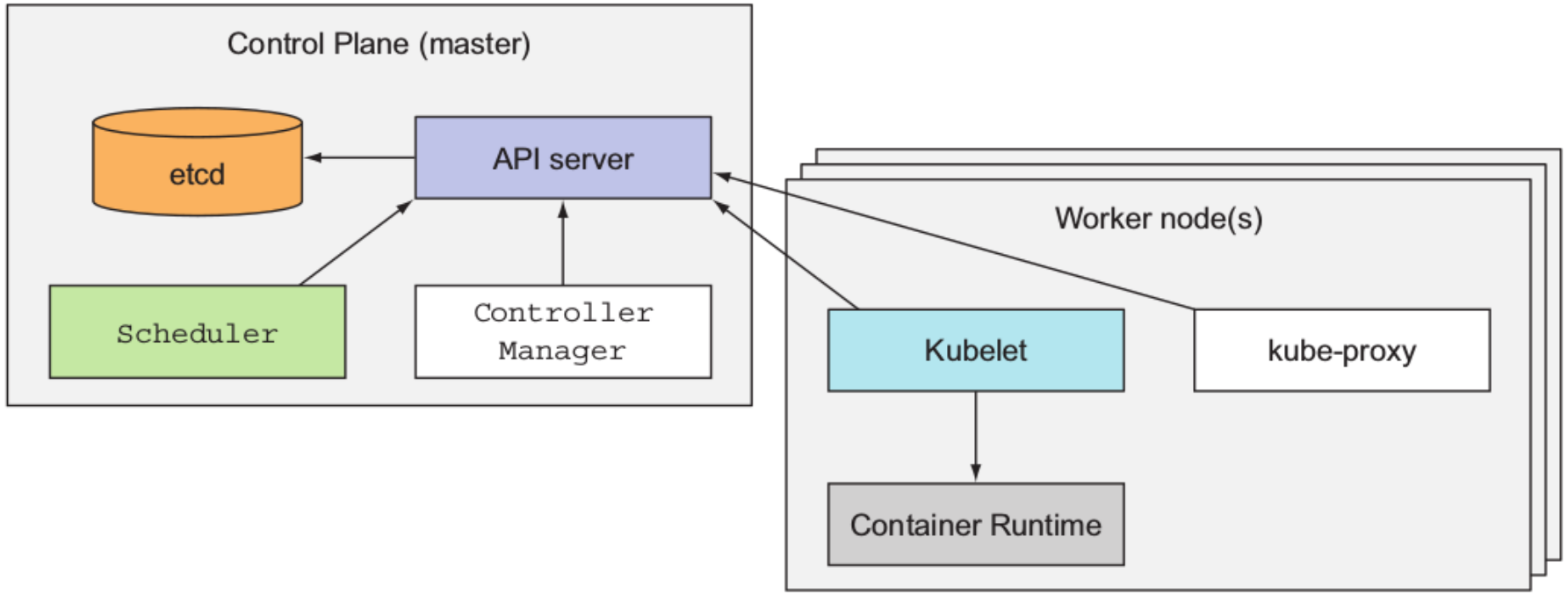
- In addition to the configuration of each component the developer must specify the desired number of instances in order to ensure scalability
- The platform will take care of replicating each component and load-balance the traffic among them in order to ensure scalability
- The platform takes care of monitoring the status of each component and handle failures: in case one instance fails (or fails the working node on which it is executed) the platform takes care of replacing the instance with a new one

Advantages

- Kubernetes relieves applications developers from having to implement infrastructure-related functionalities into their applications
- The platform takes care of implementing functionalities like scaling, load-balancing, self-healing
- Consequently application developers can focus on developing applications without wasting time figuring out how to integrate the application within the infrastructure
- The duties of the ops team are simplified: they are relieved from taking care of deploying the components, configuring them and fix failures, as all those functionalities are handled by the platform itself
- In addition to this Kubernetes can achieve a better utilization of the resources available in the infrastructure

Cluster Architecture

- The master node implements the control plane, which is responsible for controlling the cluster. Each component of the control plane can be installed on a single master node, can be split across multiple nodes or they can be replicated in order to ensure high availability.

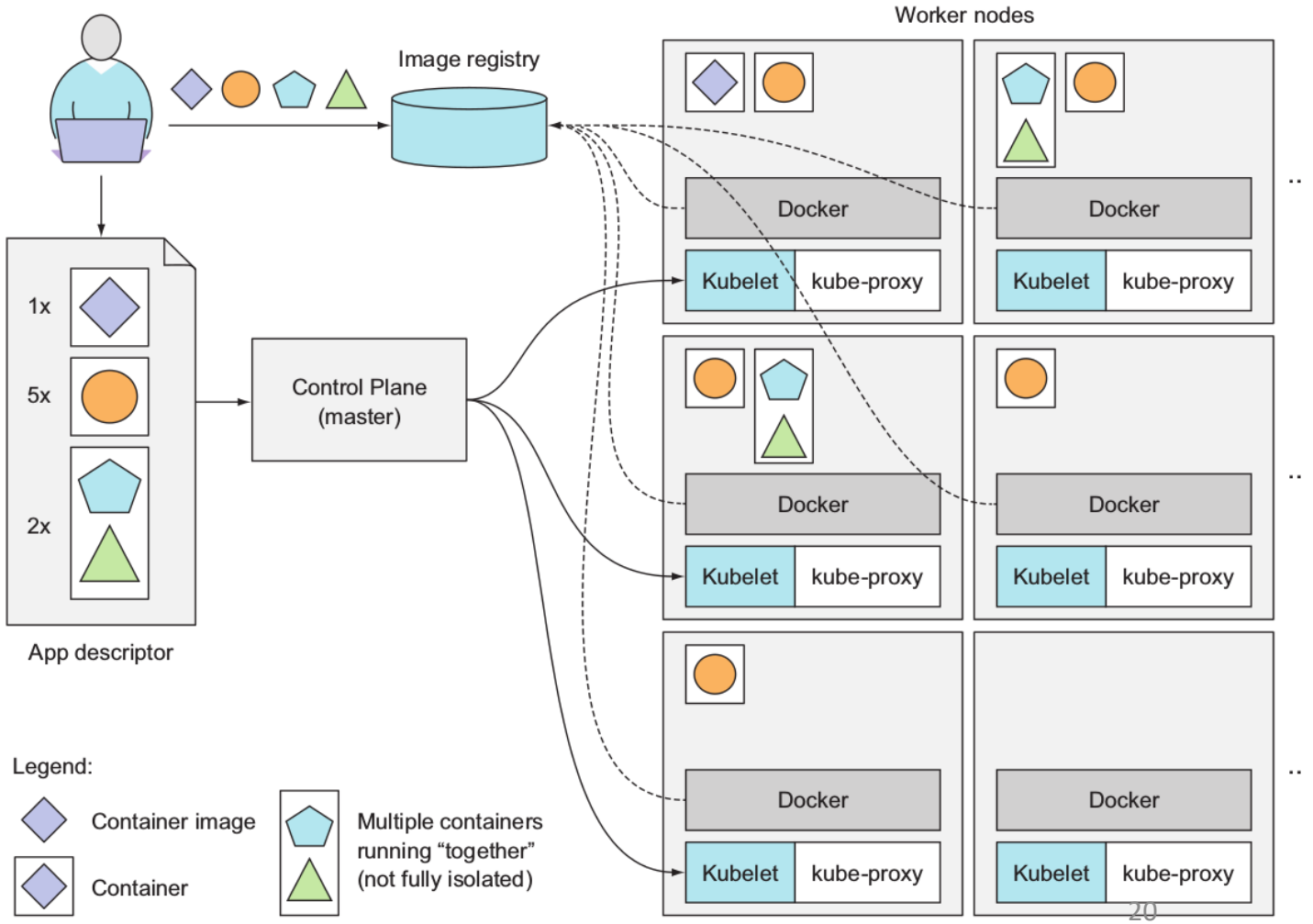


Cluster Architecture

- The control plane has the following components:
  - **API server**, which exposes an interface for developers
  - **Scheduler**, which schedules apps
  - **Controller manager**, which performs cluster-level functions such as replicating components and handling node failures
  - **Etcd**, which is a reliable distributed data store
- The worker nodes instead include the following components:
  - **Container runtime**, a container runtime environment (like Docker or others) that takes care of running the containers
  - **Kublet**, which talks with the API server and manages containers on the node
  - **Kubernetes Service Proxy**, which takes care of balancing the network traffic between application components and forward requests, while containers are migrated across working nodes

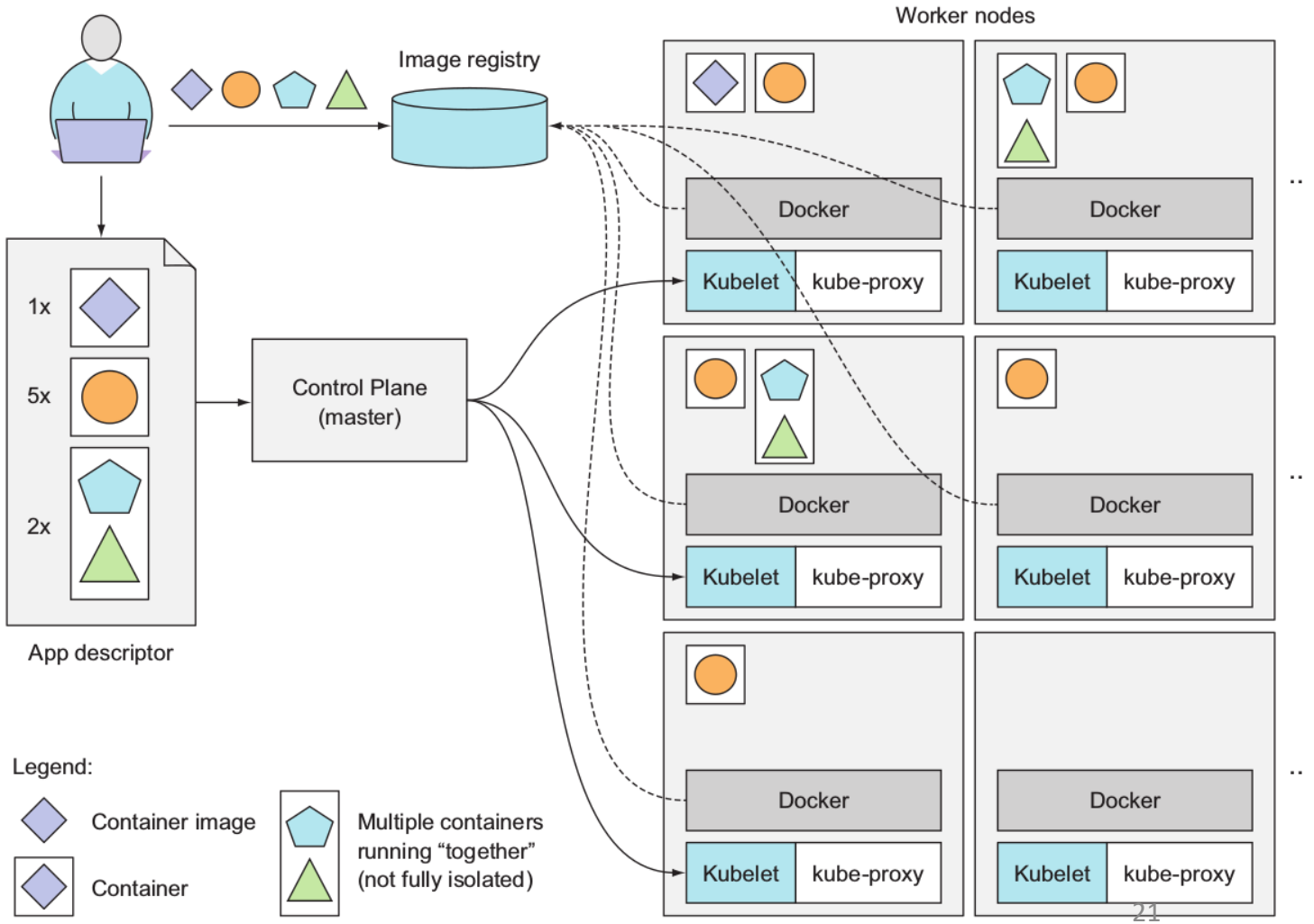
Application Deployment

- The application description provided by the developer lists the container that composes the application
- Containers are grouped together into **pods**
- A pod is a group of containers that must run on the same worker nodes, in other words they must be deployed together and shouldn't be isolated (a pod can be one container)
- For each pod/single container the developer specifies the configuration (e.g. from which image it must be instantiated) and the number of replicas to be deployed



Application Deployment

- After submitting the descriptor to the control plane of the master node, the scheduler schedules the specified number of replicas of pods to the available worker nodes
- The kublets running on the worker node takes care of controlling Docker to pull the container images from the repository and run them



Management and Maintenance

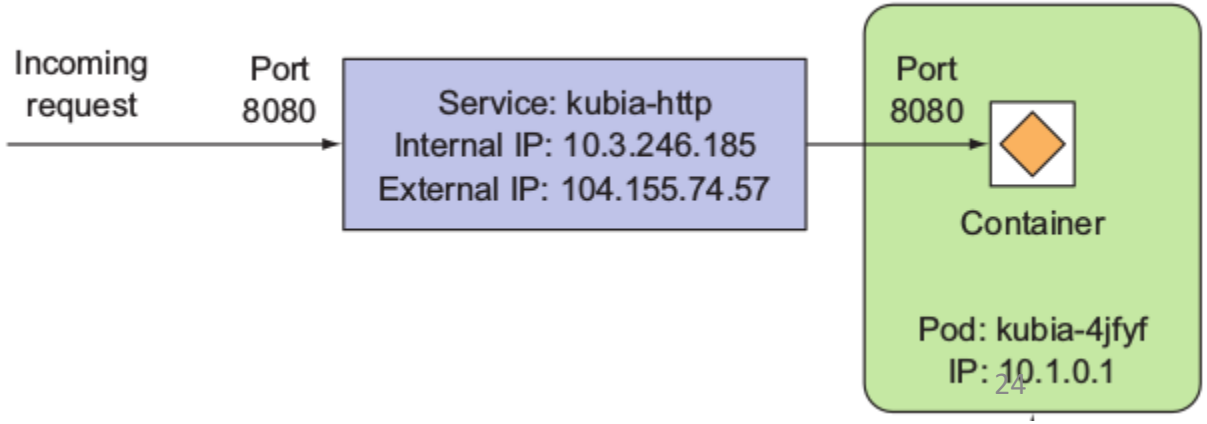
- Once the application is running, Kubernetes continuously makes sure that the deployed state of the application always matches the description provided by developers
- If one or more instance stops working properly (e.g. the application crashes) Kubernetes will take care of restarting it
- If the instance, or a whole working node, becomes inaccessible, Kubernetes will select a new node for all the containers that were running on the node
- While the application is running the developer can decide to increase/decrease the number of instances, in this case the platform can take care of adding/removing instances at runtime
- The developer can even let the platform decide the optimal number of instances of a certain container, in this case the platform will adjust the number of running instances based on real-time metrics like CPU load, memory consumption, query per second, etc.

Container Migration

- The platform automatically takes care of instantiating containers and migrating them when required (e.g. when a working node fails)
- When a container provides a service to external client, the platform has to take care of forwarding requests as the container is moved across the platform
- What it is usually performed is that the platform exposes a specific service with a single public IP address, regardless of the number of containers and their location
- The platform with the kube-proxy component takes care of connecting to the containers and forwarding the traffic in a proper manner
- In addition, the component takes care of implementing load-balancing policies to forward the traffic in a balanced manner in order to ensure scalability

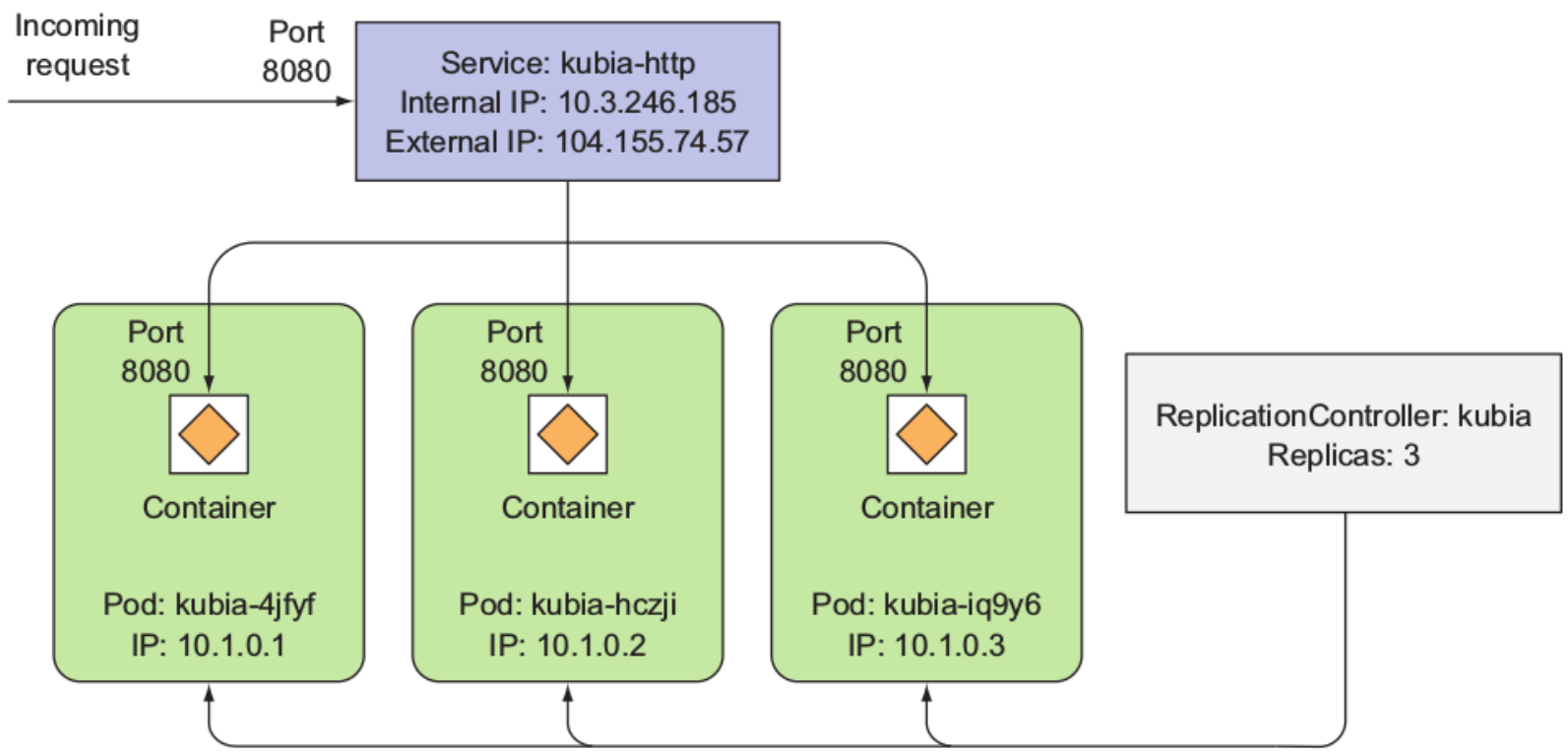
Exposing Applications

- Pods are connected to a local private networks, in order to allow communication among them
- Each pod gets its own IP address, however, this address is internal to this network and it is not accessible from outside
- Such local private network allows the communication among containers running on the same or different workers
- If a service needs to be exposed to external networks (e.g. because it exposes the web or the REST API interface of the application) the developer needs to define a service object
- A service object instructs the platform to associate with a pod an external IP address and port so that the service exposed by the pod can be reached from external hosts
- The service takes care of forwarding the request coming to that external IP address to the local IP address of one of the associated pod instances



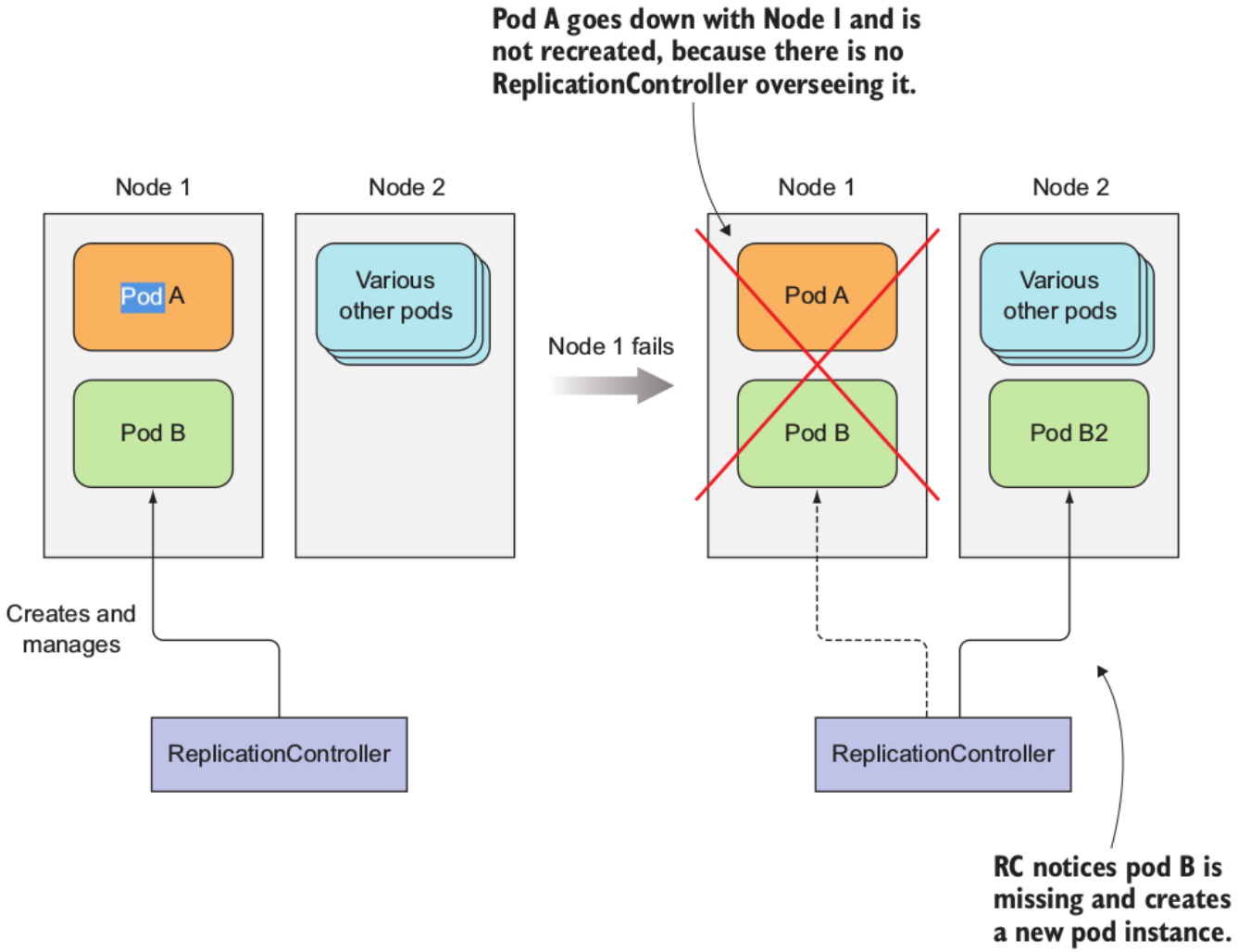
Load Balancing

- A service can be connected to a pod that requires multiple redundant instances for load balancing
- In this case the service spreads the requests across the different instances
- To each pod the platform associates a replication controller for taking care that the proper number of instances is always running



Failure handling

- The replication controller is also responsible for monitoring the instances of a pod to detect failures
- If the failure of a working node is detected, the replication controller detects it and react by creating a new pod instance



Pod design

- The general rule to group containers into pods is that developers should always run containers in separate pods unless a specific reason requires them to be part of the same pod, so they are deployed on the same working node
- For instance an application might have latency requirement in the data exchange between two container, in this case it is convenient to group them in the same pod so they do not communicate over the network