PAMCA601: Cloud Computing

Module 3 : Cloud Computing Programming Paradigms

Courtesy : *Ming Lian , Dogules E Comer & Other Sources of Internet*

The Map Reduce Programming Paradigm

- *MapReduce* is a programming model for data processing

- The power of MapReduce lies in its ability to scale to 100s or 1000s of computers, each with several processor cores

- How large an amount of work?

  - Web-Scale data on the order of 100s of GBs to TBs or PBs

  - It is likely that the input data set will not fit on a single computer's hard drive

  - Hence, a distributed file system (e.g., Google File System- GFS) is typically required

Motivations

- Motivations
  - Large-scale data processing on clusters
  - Massively parallel (hundreds or thousands of CPUs)
  - Reliable execution with easy data access
- Functions
  - Automatic parallelization & distribution
  - Fault-tolerance
  - Status and monitoring tools
  - A clean abstraction for programmers
    - Functional programming meets distributed computing
    - A batch data processing system

Commodity Clusters

- MapReduce is designed to efficiently process large volumes of data by connecting many commodity computers together to work in parallel

- A *theoretical* 1000-CPU machine would cost a very large amount of money, far more than 1000 single-CPU or 250 quad-core machines

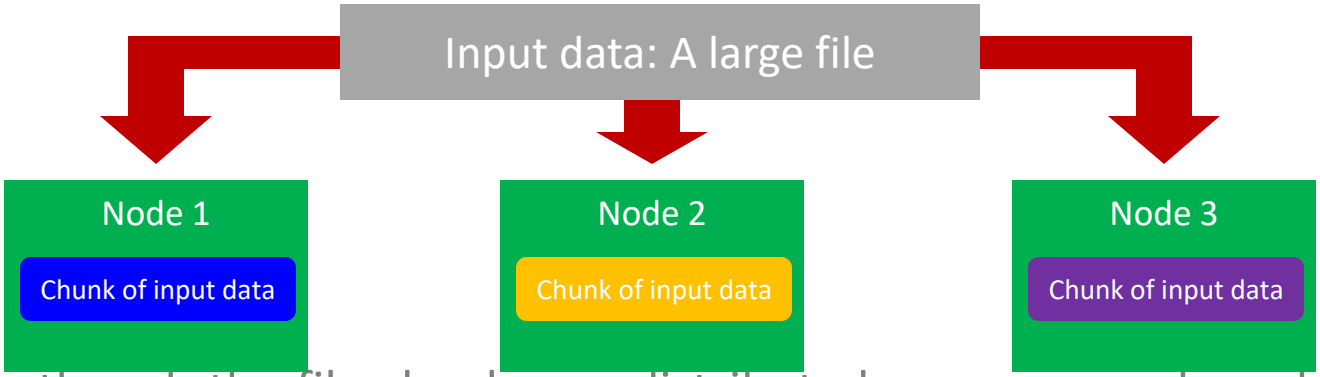
- MapReduce ties smaller and more reasonably priced machines together into a single cost-effective *commodity cluster*

Isolated Tasks

- MapReduce divides the workload into multiple *independent tasks* and schedule them across cluster nodes
- A work performed by each task is done *in isolation* from one another
- The amount of communication which can be performed by tasks is mainly limited for scalability reasons
- The communication overhead required to keep the data on the nodes synchronized at all times would prevent the model from performing reliably and efficiently at large scale

Data Distribution

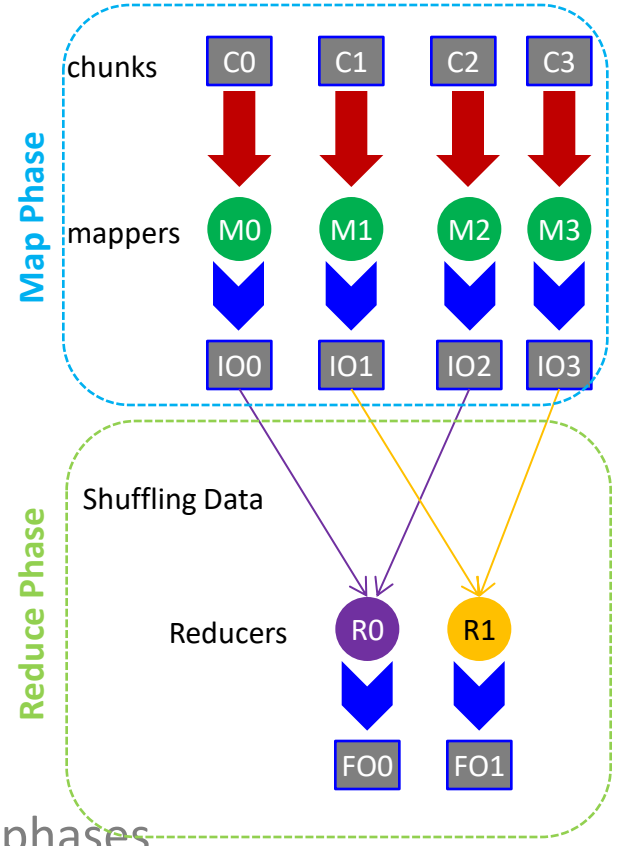
- In a MapReduce cluster, data is distributed to all the nodes of the cluster as it is being loaded in
- An underlying distributed file systems (e.g., GFS) splits large data files into chunks which are managed by different nodes in the cluster



- Even though the file chunks are distributed across several machines, they form *a single namesapce*

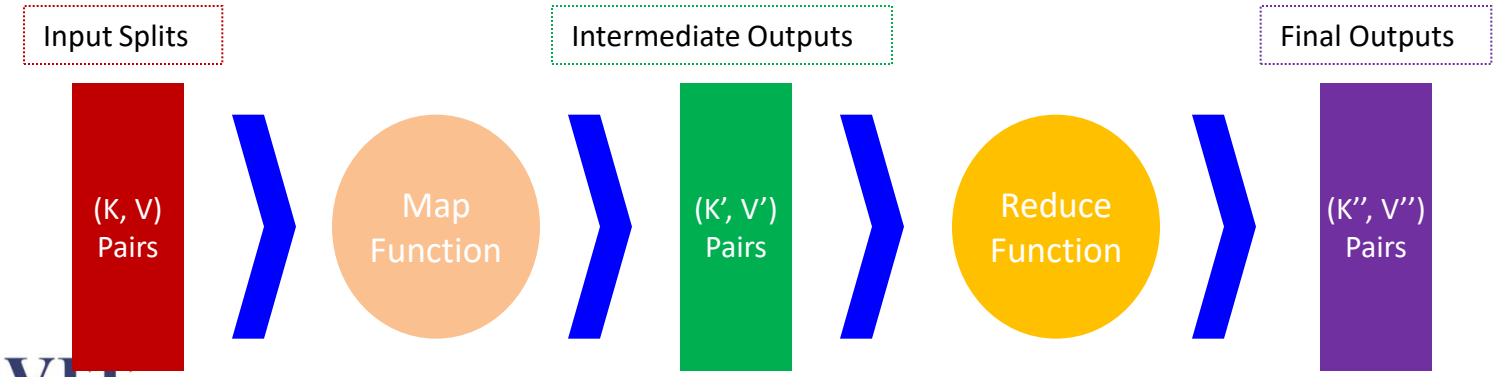
MapReduce: A Bird's-Eye View

- In MapReduce, chunks are processed in isolation by tasks called *Mappers*
- The outputs from the mappers are denoted as intermediate outputs (IOs) and are brought into a second set of tasks called *Reducers*
- The process of bringing together IOs into a set of Reducers is known as *shuffling process*
- The Reducers produce the final outputs (FOs)
- Overall, MapReduce breaks the data flow into two phases, *map phase* and *reduce phase*



Keys and Values

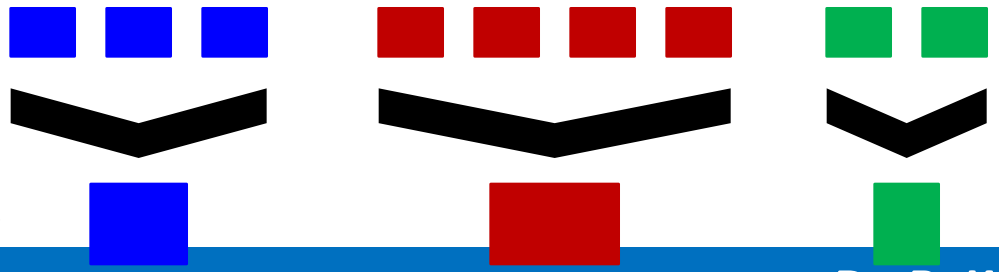
- The programmer in MapReduce has to specify two functions, the *map function* and the *reduce function* that implement the Mapper and the Reducer in a MapReduce program
- In MapReduce data elements are always structured as key-value (i.e., (K, V)) pairs
- The map and reduce functions receive and *emit* (K, V) pairs



Partitions

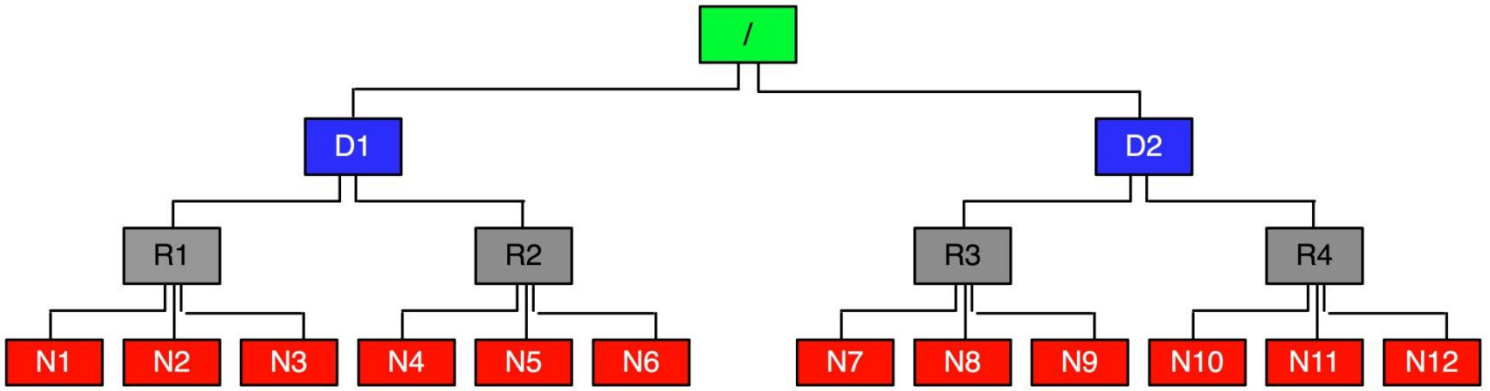
- In MapReduce, intermediate output values are not usually reduced together
- *All values with the same key are presented to a single Reducer together*
- More specifically, a different subset of intermediate key space is assigned to each Reducer
- These subsets are known as *partitions*

Different colors represent different keys (potentially) from different Mappers



Partitions are the input to Reducers

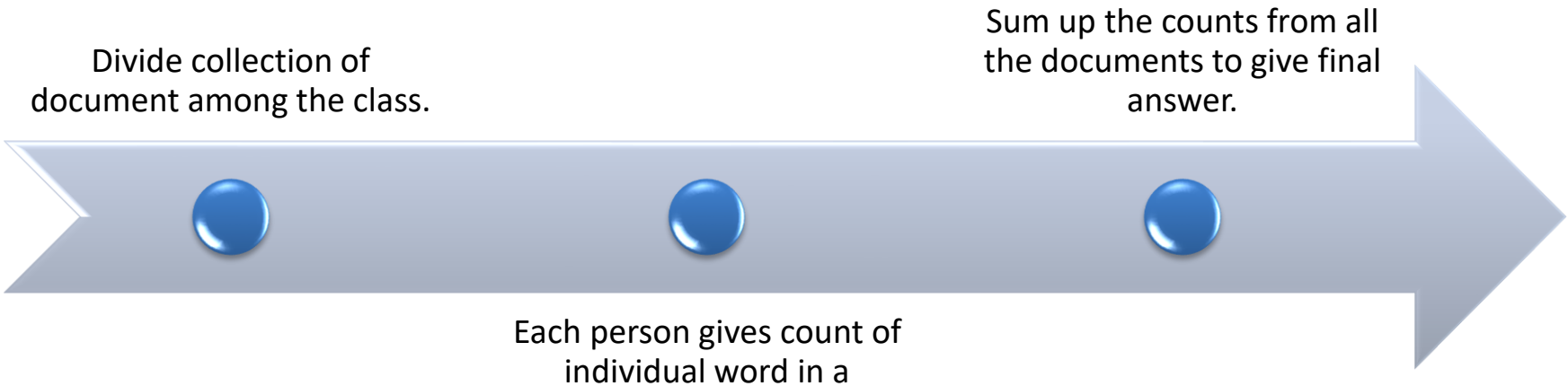
Network Topology In MapReduce



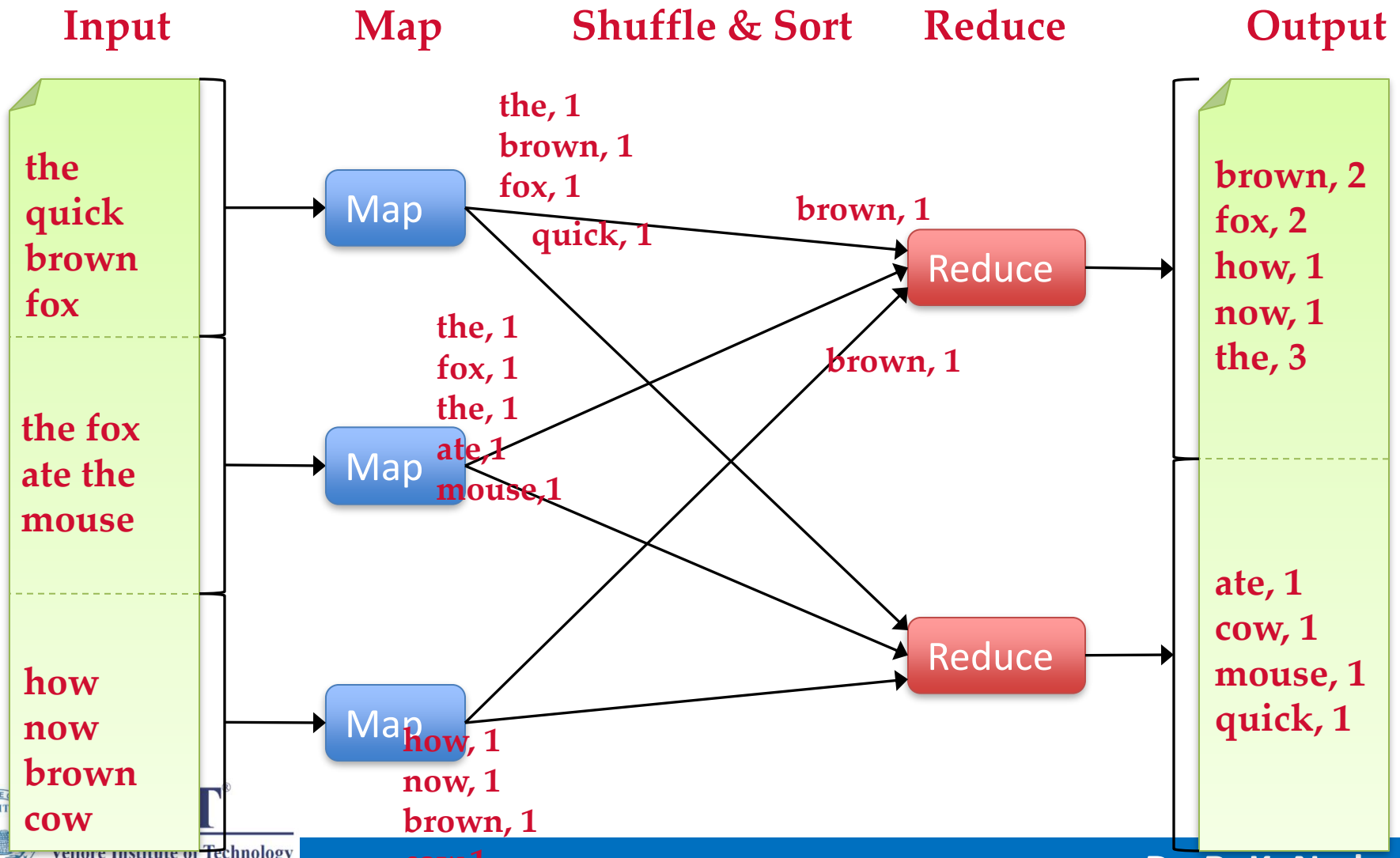
- MapReduce assumes a tree style network topology
- Nodes are spread over different racks embraced in one or many data centers
- A salient point is that the bandwidth between two nodes is dependent on their relative locations in the network topology
- For example, nodes that are on the same rack will have higher bandwidth between them as opposed to nodes that are off-rack

Example: Word counting

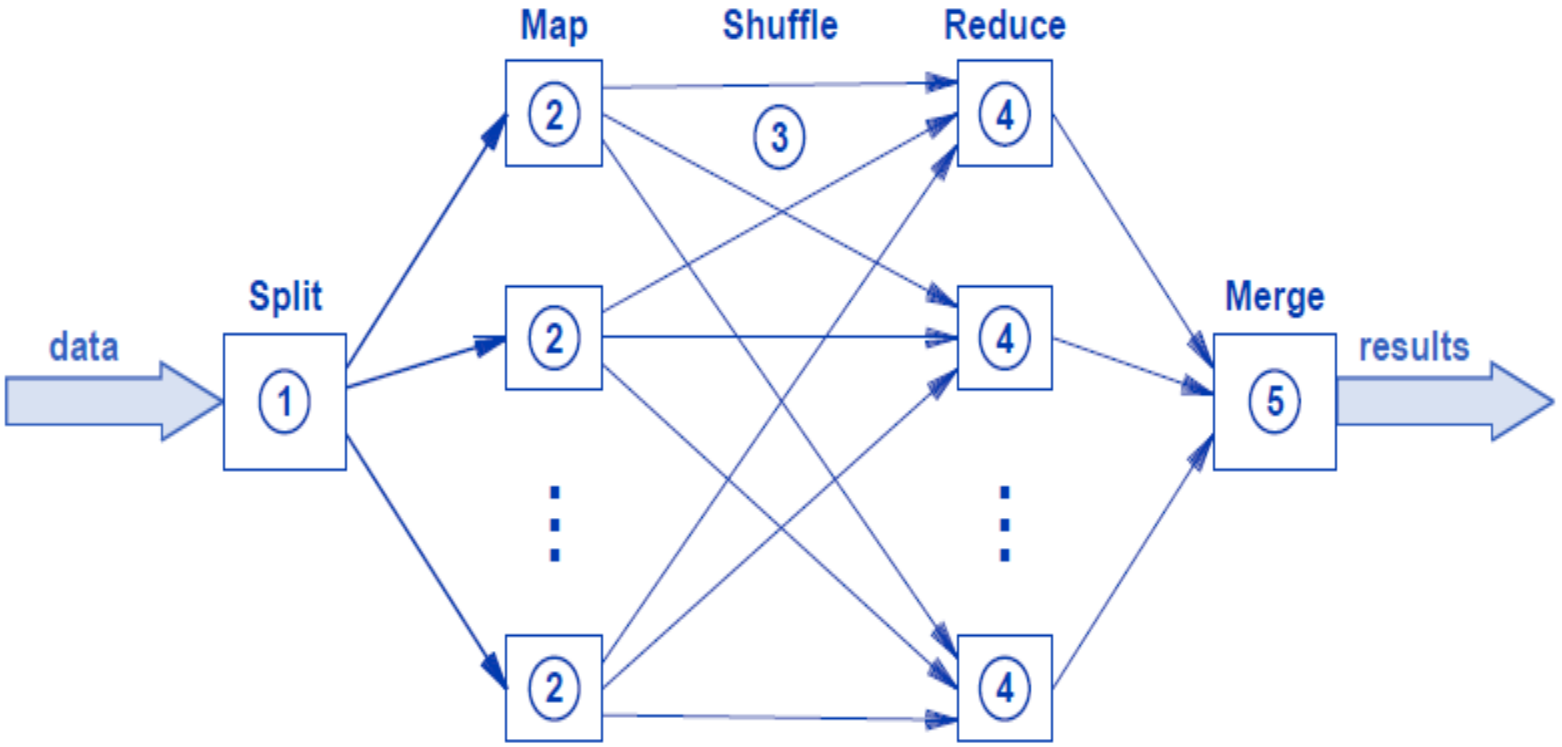
Consider the problem of counting the number of occurrences of each word in a large collection of documents”

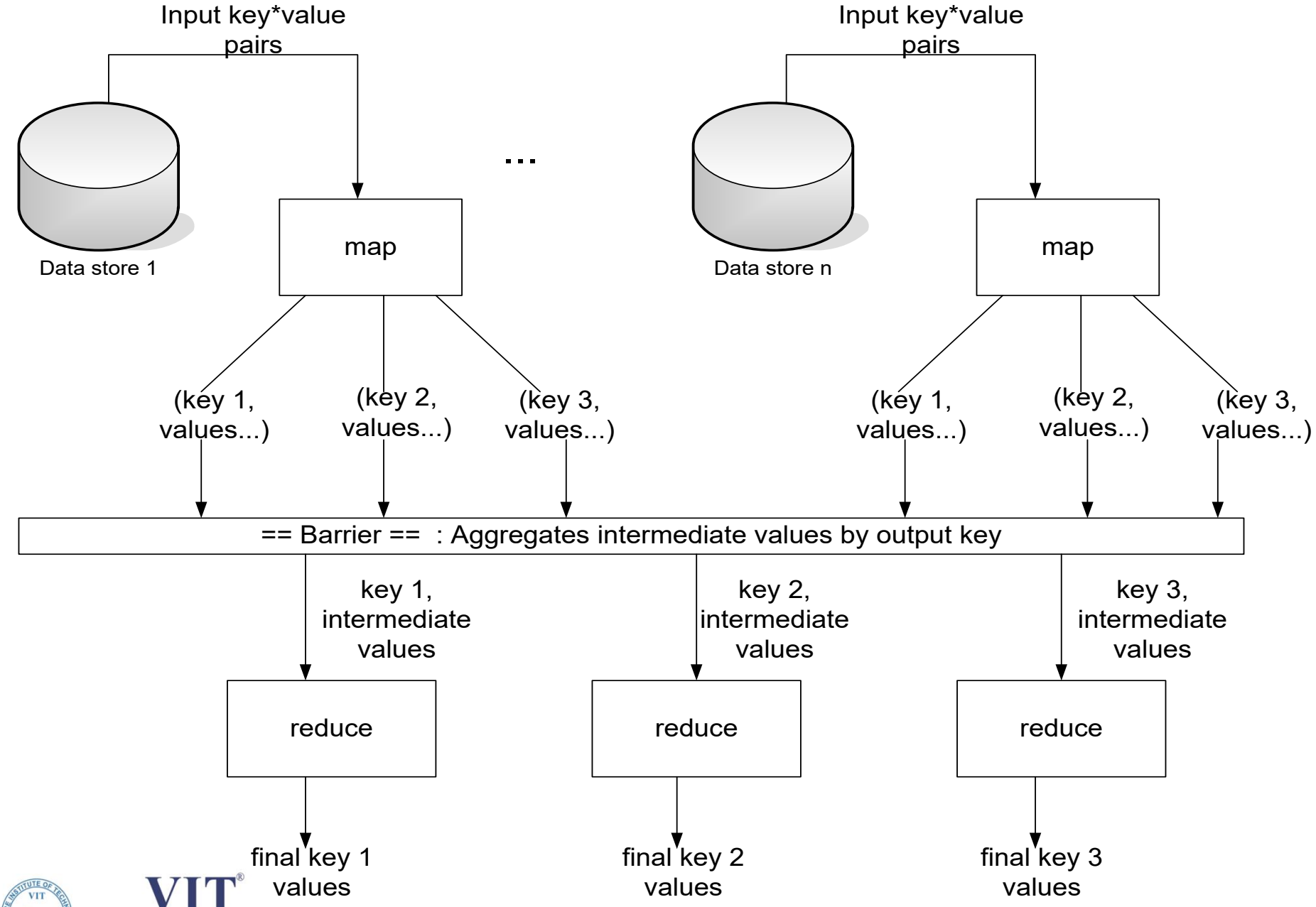


Word Count Execution

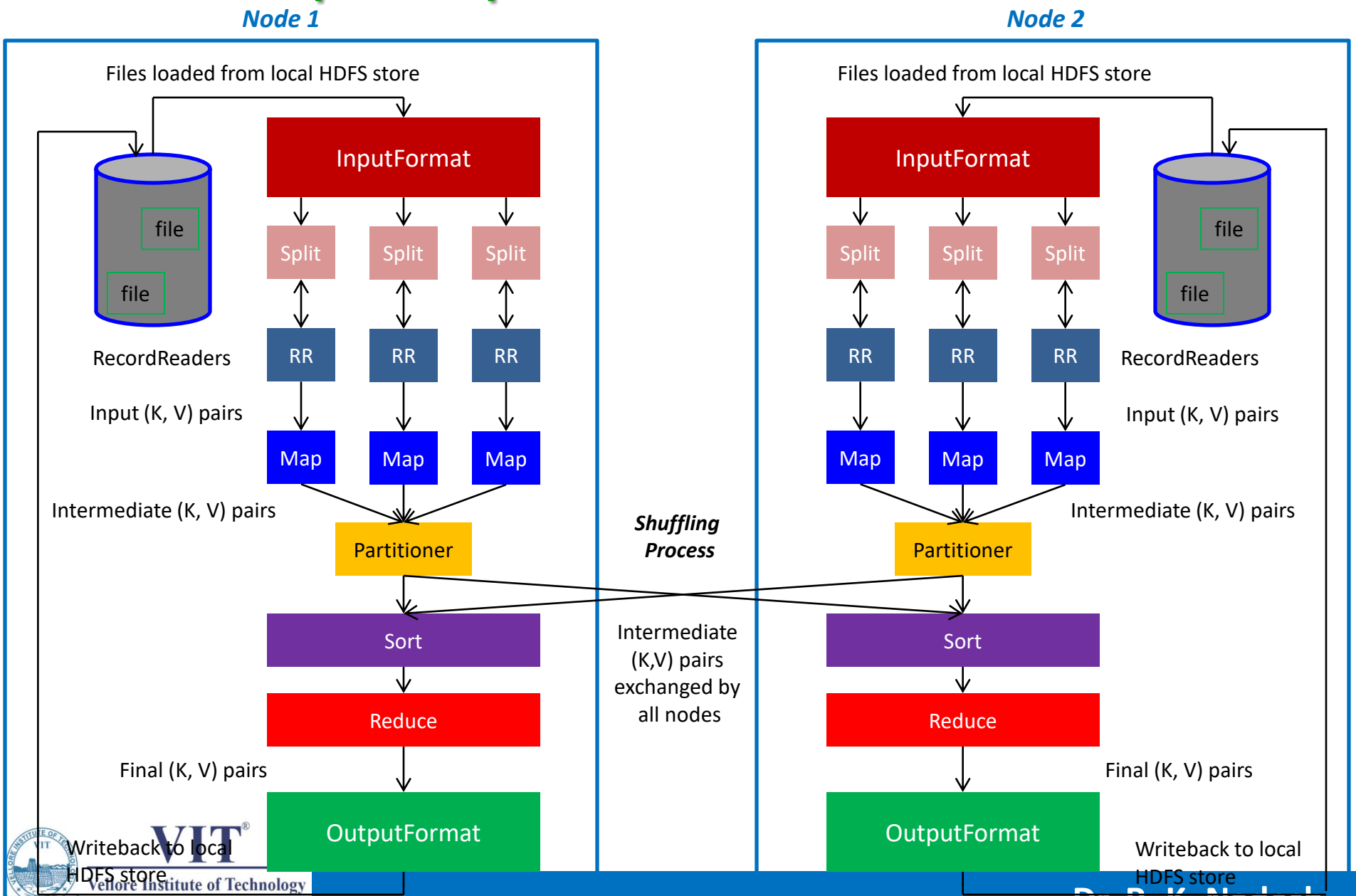


The five conceptual steps of Map Reduce processing



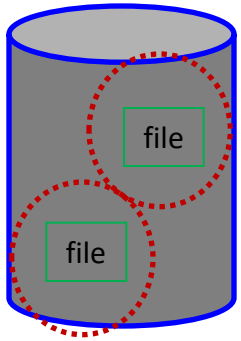

Input key*value pairs
Input key*value pairs
Data store 1
Data store n
map
map
(key 1, values...)
(key 2, values...)
(key 3, values...)
(key 1, values...)
(key 2, values...)
(key 3, values...)
== Barrier == : Aggregates intermediate values by output key
key 1, intermediate values
key 2, intermediate values
key 3, intermediate values
reduce
reduce
reduce
final key 1 values
final key 2 values
final key 3 values

Hadoop MapReduce: A Closer Look



Input Files

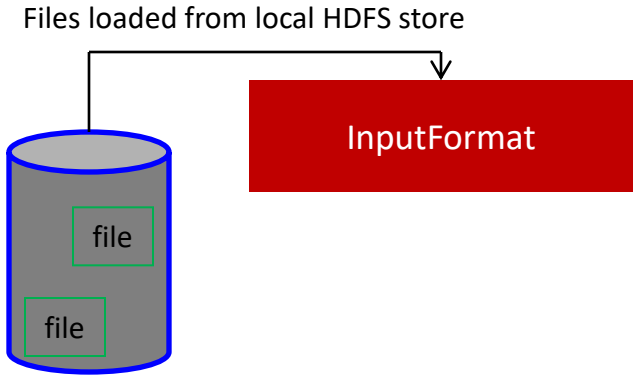
- *Input files* are where the data for a MapReduce task is initially stored
- The input files typically reside in a distributed file system (e.g. HDFS)
- The format of input files is arbitrary
  - Line-based log files
  - Binary files
  - Multi-line input records
  - Or something else entirely



InputFormat

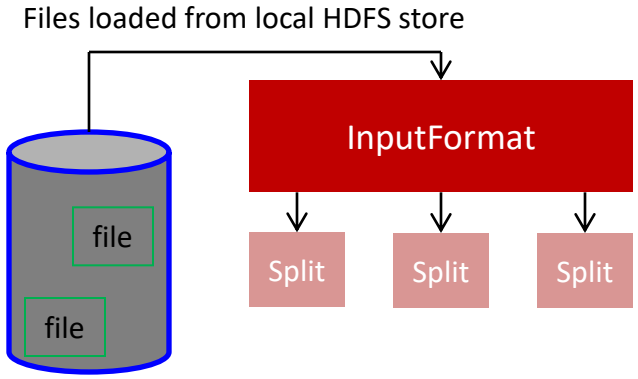
- How the input files are split up and read is defined by the *InputFormat*

- InputFormat is a class that does the following:
  - Selects the files that should be used for input
  - Defines the *InputSplits* that break a file
  - Provides a factory for *RecordReader* objects that read the file



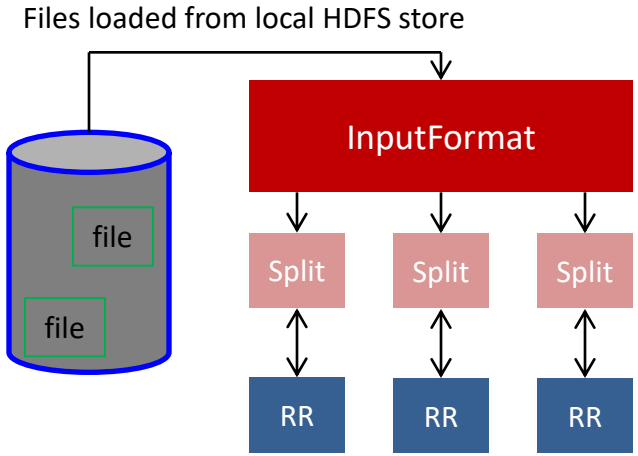
Input Splits

- An *input split* describes a unit of work that comprises a single map task in a MapReduce program
- By default, the InputFormat breaks a file up into 64MB splits
- By dividing the file into splits, we allow several map tasks to operate on a single file in parallel
- If the file is very large, this can improve performance significantly through parallelism
- Each map task corresponds to a *single* input split



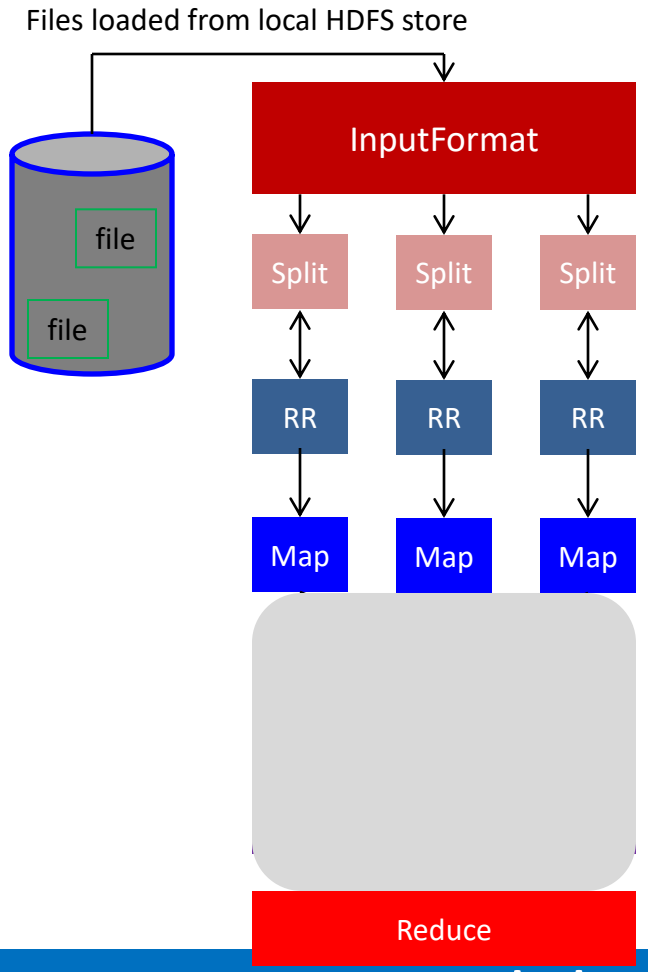
RecordReader

- The input split defines a slice of work but does not describe how to access it
- The *RecordReader* class actually loads data from its source and converts it into (K, V) pairs suitable for reading by Mappers
- The RecordReader is invoked repeatedly on the input until the entire split is consumed
- Each invocation of the RecordReader leads to another call of the map function defined by the programmer



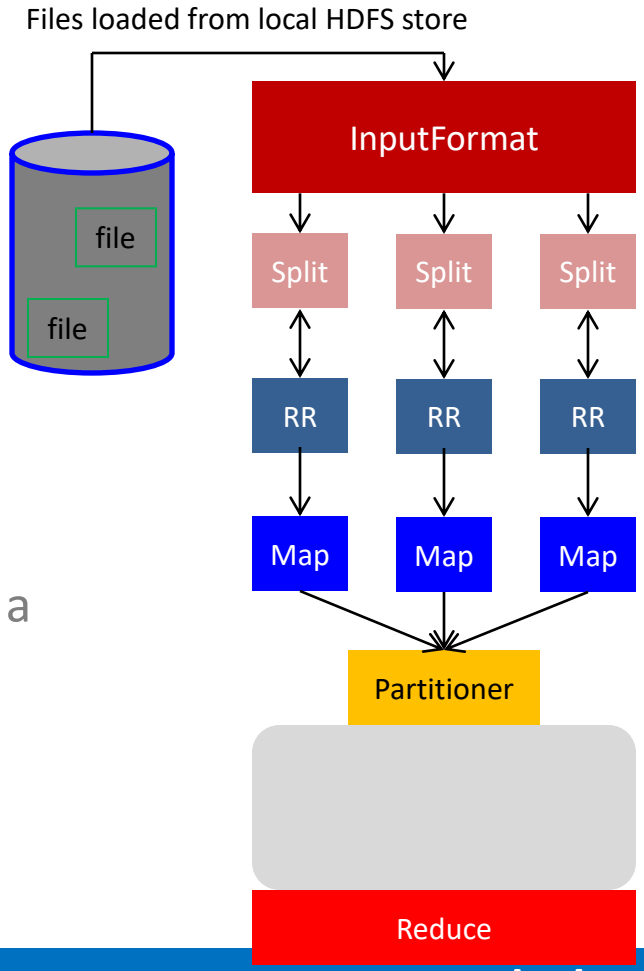
Mapper and Reducer

- The *Mapper* performs the user-defined work of the first phase of the MapReduce program
- A new instance of Mapper is created for each split
- The *Reducer* performs the user-defined work of the second phase of the MapReduce program
- A new instance of Reducer is created for each partition
- *For each key in the partition assigned to a Reducer, the Reducer is called once*



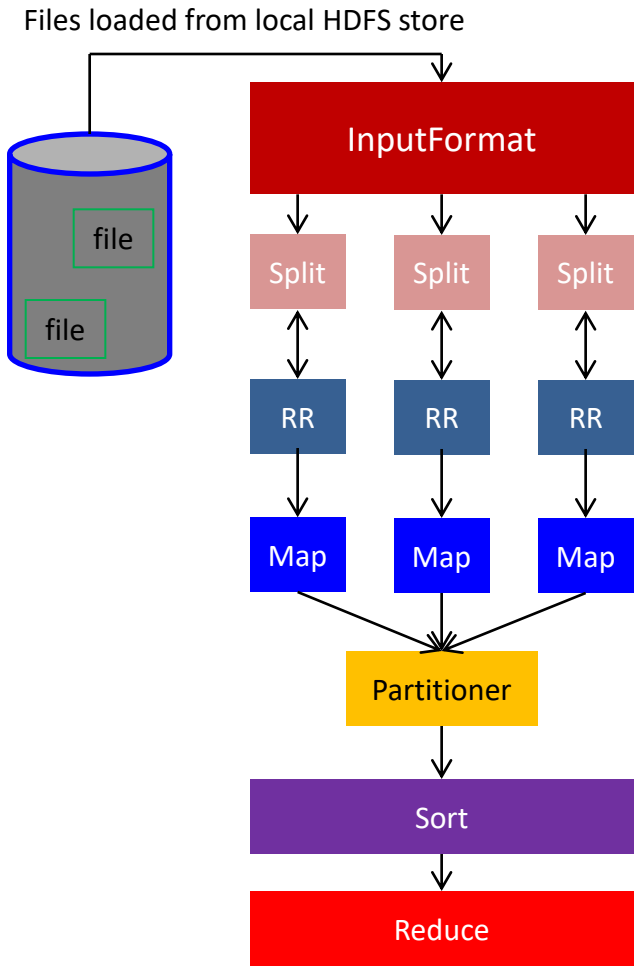
Partitioner

- Each mapper may emit (K, V) pairs to *any* partition
- Therefore, the map nodes must all agree on where to send different pieces of intermediate data
- The *partitioner* class determines which partition a given (K,V) pair will go to
- The default partitioner computes *a hash value* for a given key and assigns it to a partition based on this result



Sort

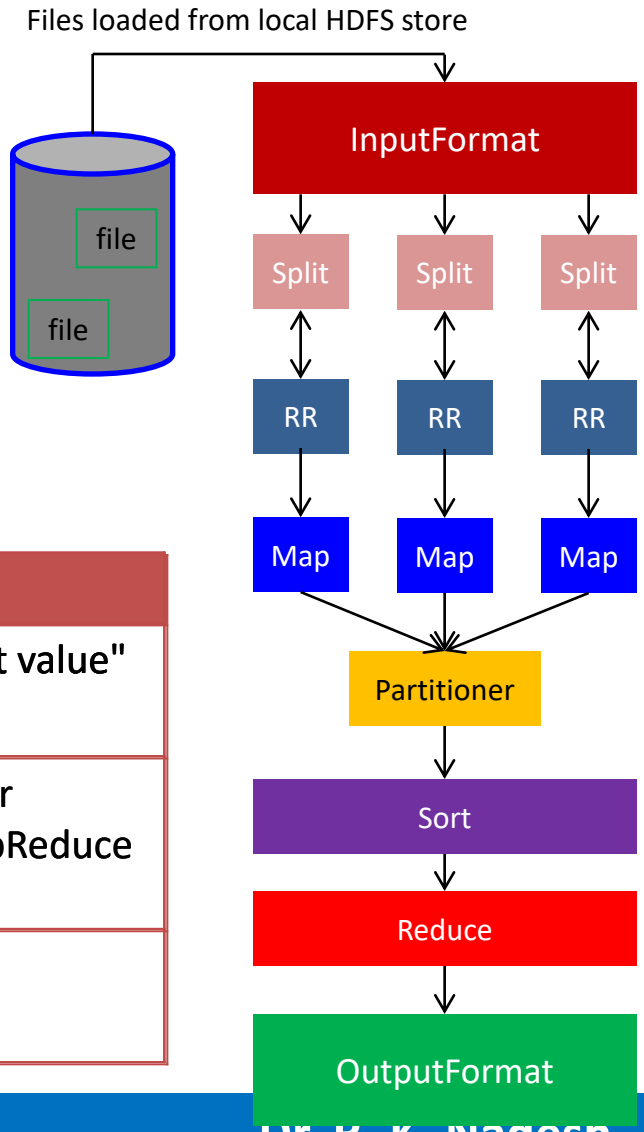
- Each Reducer is responsible for reducing the values associated with (several) intermediate keys
- The set of intermediate keys on a single node is *automatically sorted* by MapReduce before they are presented to the Reducer



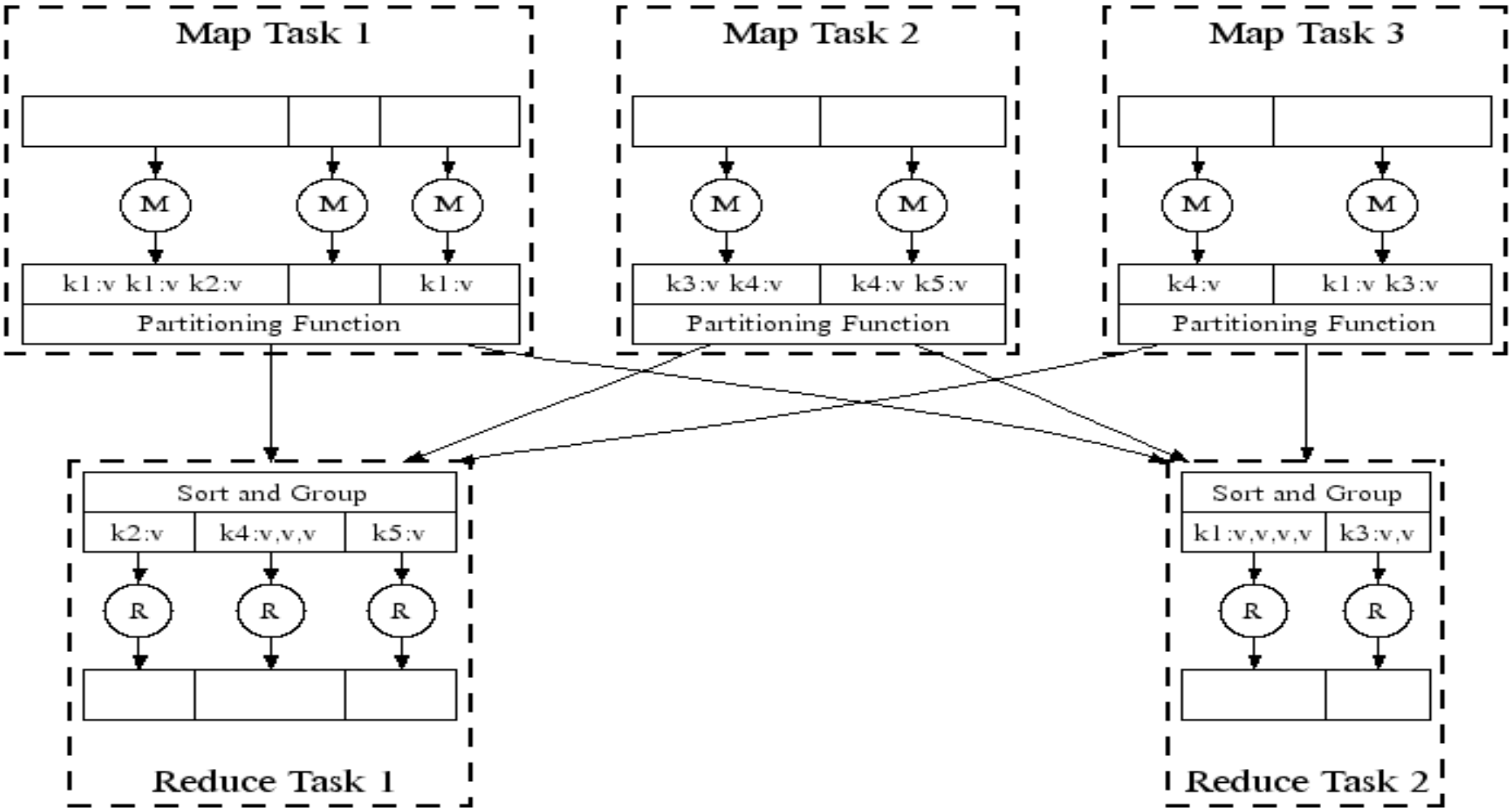
OutputFormat

- The *OutputFormat* class defines the way (K,V) pairs produced by Reducers are written to output files
- The instances of OutputFormat provided by Hadoop write to files on the local disk or in HDFS
- Several OutputFormats are provided by Hadoop:

OutputFormat	Description
TextOutputFormat	Default; writes lines in "key \t value" format
SequenceFileOutputFormat	Writes binary files suitable for reading into subsequent MapReduce jobs
NullOutputFormat	Generates no output files

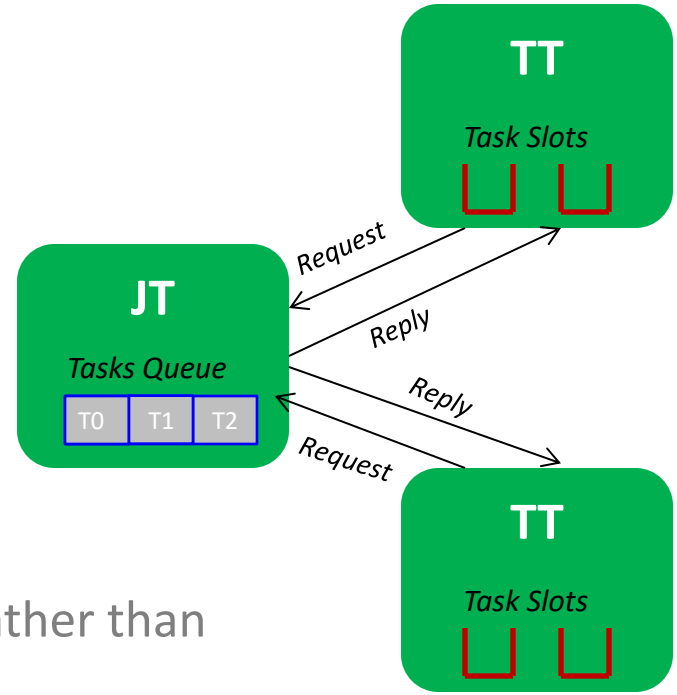


Parallel Efficiency of Map Reduce



Task Scheduling in MapReduce

- MapReduce adopts a *master-slave architecture*
- The master node in Map Reduce is referred to as *Job Tracker* (JT)
- Each slave node in MapReduce is referred to as *Task Tracker* (TT)
- MapReduce adopts a *pull scheduling* strategy rather than a *push one*
  - I.e., JT does not push map and reduce tasks to TTs but rather TTs pull them by making pertaining requests



Map and Reduce Task Scheduling

- Every TT sends a *heartbeat message* periodically to JT encompassing a request for a map or a reduce task to run

I. Map Task Scheduling:

- JT satisfies requests for map tasks via attempting to schedule mappers in the *vicinity* of their input splits (i.e., it considers locality)

II. Reduce Task Scheduling:

- However, JT simply assigns the next yet-to-run reduce task to a requesting TT regardless of TT's network location and its implied effect on the reducer's shuffle time (i.e., it does not consider locality)

Job Scheduling in MapReduce

- In MapReduce, an application is represented as a *job*
- A job encompasses multiple map and reduce tasks
- MapReduce in Hadoop comes with a choice of schedulers:
  - The default is the *FIFO scheduler* which schedules jobs in order of submission
  - There is also a multi-user scheduler called the *Fair scheduler* which aims to give every user a fair share of the cluster capacity over time

Fault Tolerance in Hadoop

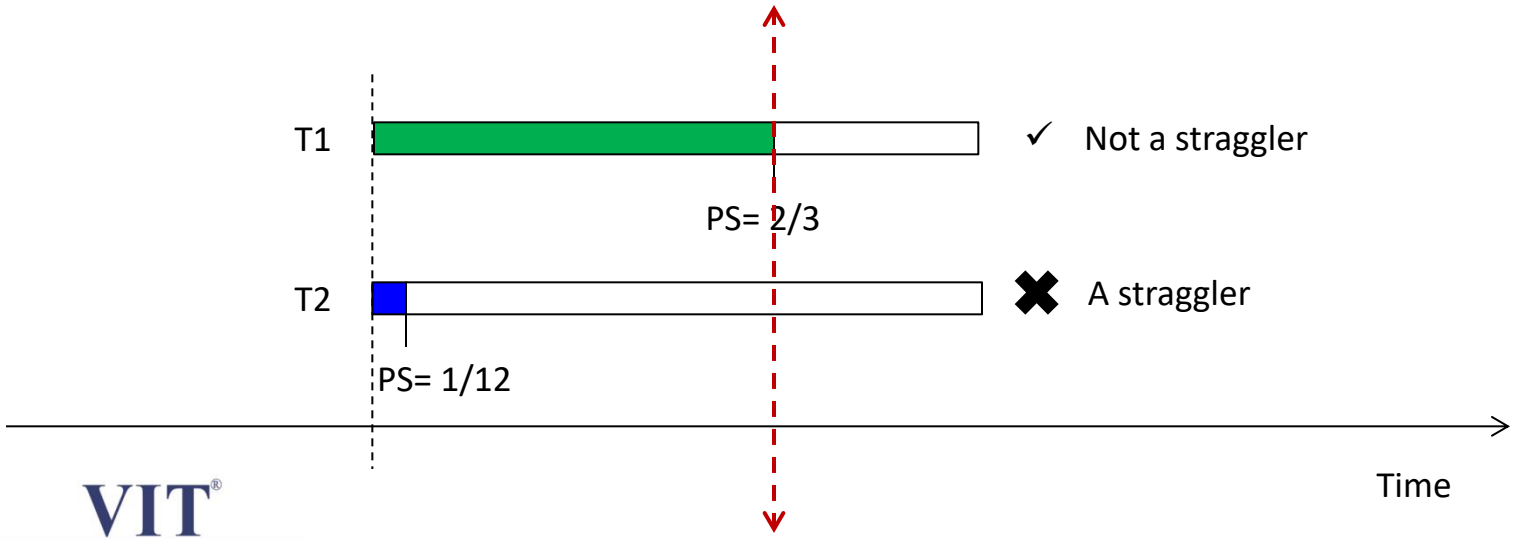
- MapReduce can guide jobs toward a successful completion even when jobs are run on a large cluster where probability of failures increases
- The primary way that MapReduce achieves fault tolerance is through *restarting tasks*
- If a TT fails to communicate with JT for a period of time (by default, 1 minute in Hadoop), JT will assume that TT in question has crashed
  - If the job is still in the map phase, JT asks another TT to re-execute *all Mappers that previously ran at the failed TT*
  - If the job is in the reduce phase, JT asks another TT to re-execute *all Reducers that were in progress on the failed TT*

Speculative Execution

- A MapReduce job is dominated by the slowest task
- MapReduce attempts to locate slow tasks (*stragglers*) and run redundant (*speculative*) tasks that will optimistically commit before the corresponding stragglers
- This process is known as *speculative execution*
- Only one copy of a straggler is allowed to be speculated
- Whichever copy (among the two copies) of a task commits first, it becomes the definitive copy, and the other copy is killed by JT

Locating Stragglers

- How does Hadoop locate stragglers?
  - Hadoop monitors each task progress using a *progress score* between 0 and 1
  - If a task's progress score ***is less than*** (average – 0.2), and the task has run for at least 1 minute, it is marked as a straggler



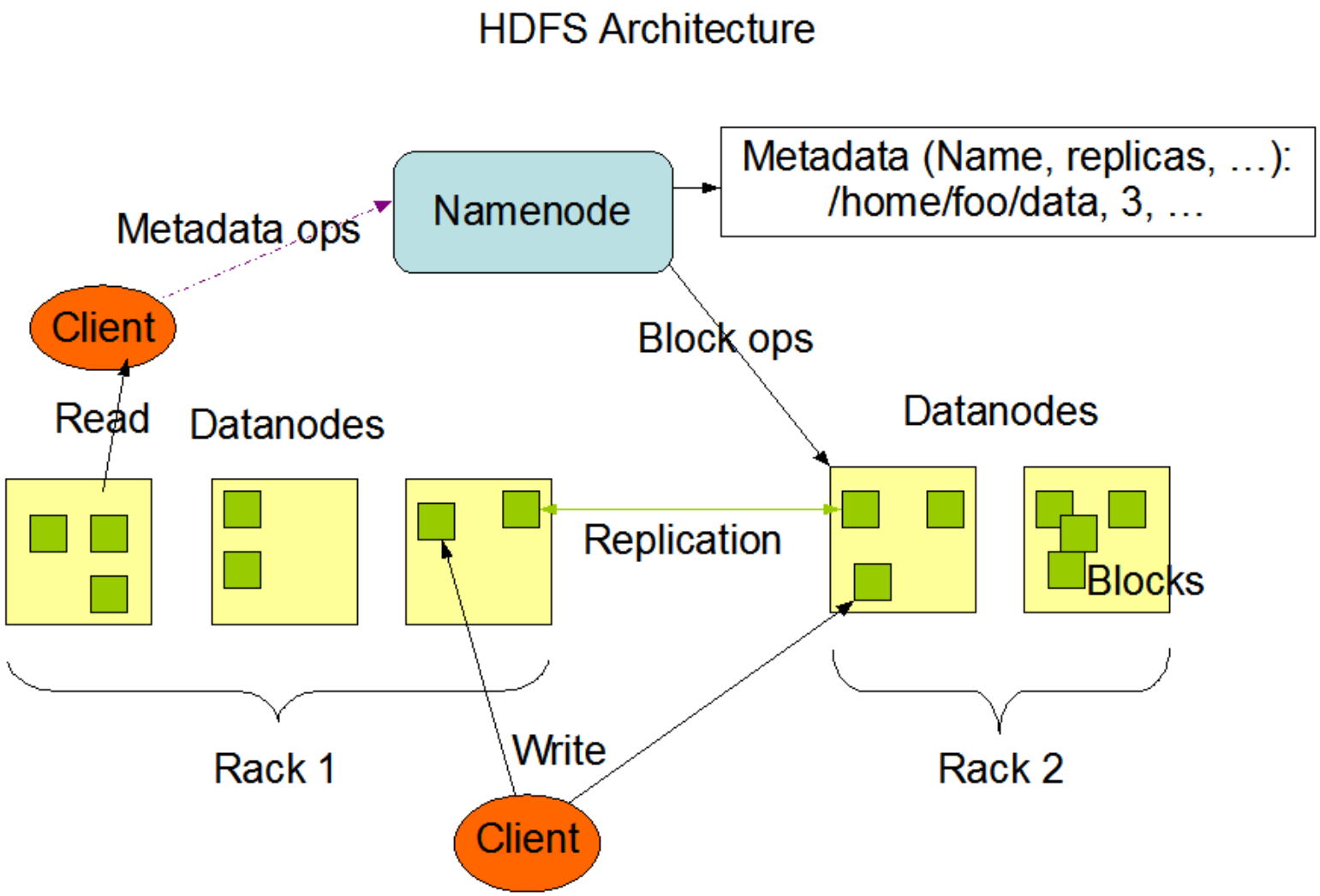
Hadoop Distributed File System(HDFS)

- Very Large Distributed File System
  - 10K nodes, 100 million files, 10PB
- Assumes Commodity Hardware
  - Files are replicated to handle hardware failure
  - Detect failures and recover from them
- Optimized for Batch Processing
  - Data locations exposed so that computations can move to where data resides
  - Provides very high aggregate bandwidth

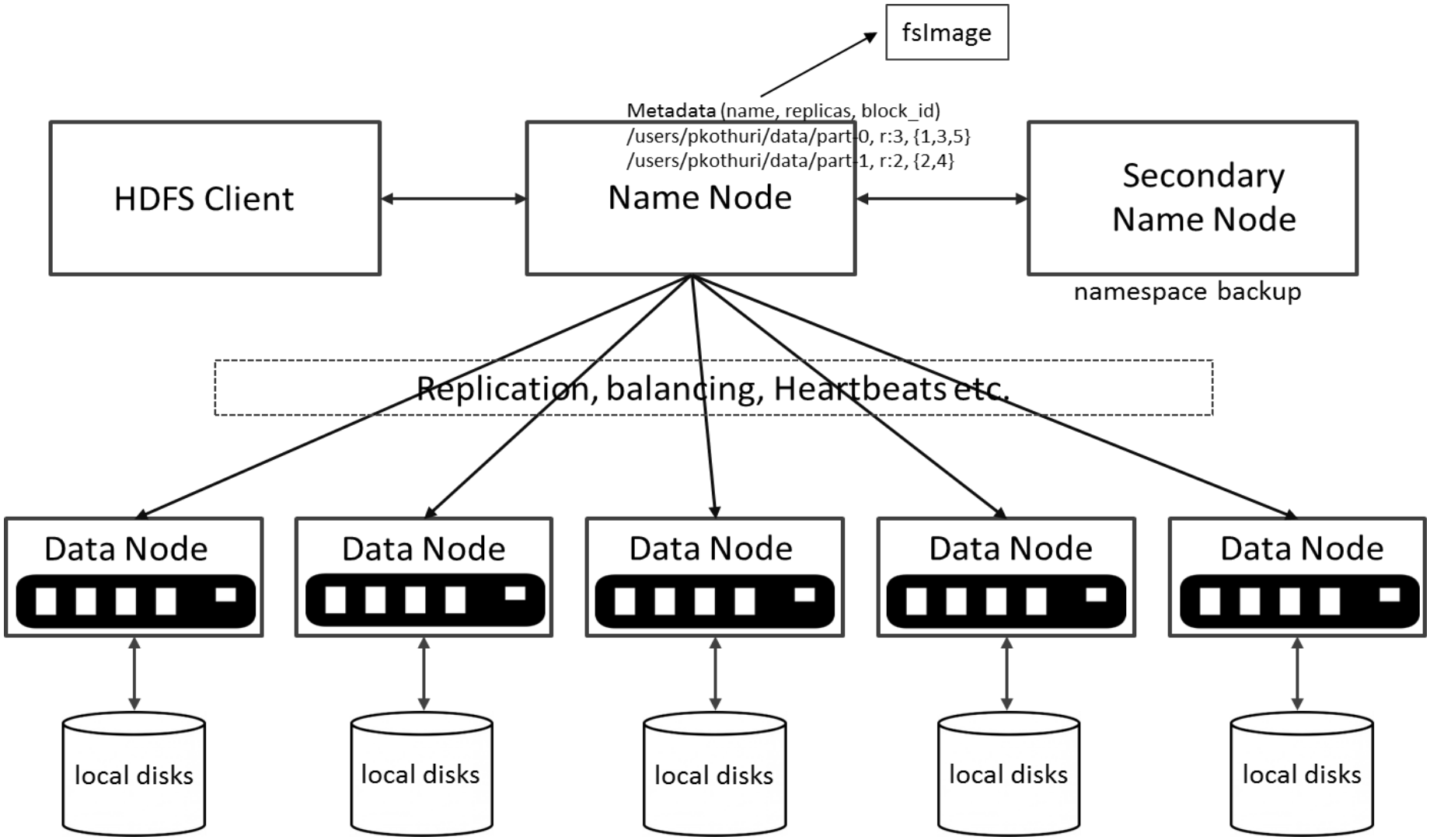
What's HDFS ?

- HDFS is a distributed file system that is fault tolerant, scalable and extremely easy to expand.
- HDFS is the primary distributed storage for Hadoop applications.
- HDFS provides interfaces for applications to move themselves closer to data.
- HDFS is designed to 'just work', however a working knowledge helps in diagnostics and improvements.

HDFS Architecture



HDFS Architecture



HDFS – Data Organization

- Each file written into HDFS is split into data blocks
- Each block is stored on one or more nodes
- Each copy of the block is called replica
- Block placement policy
  - First replica is placed on the local node
  - Second replica is placed in a different rack
  - Third replica is placed in the same rack as the second replica

Functions of a NameNode

- Manages File System Namespace
  - Maps a file name to a set of blocks
  - Maps a block to the DataNodes where it resides
- Cluster Configuration Management
- Replication Engine for Blocks

NameNode Metadata

- Metadata in Memory
  - The entire metadata is in main memory
  - No demand paging of metadata
- Types of metadata
  - List of files
  - List of Blocks for each file
  - List of DataNodes for each block
  - File attributes, e.g. creation time, replication factor
- A Transaction Log
  - Records file creations, file deletions etc

DataNode

- A Block Server
  - Stores data in the local file system (e.g. ext3)
  - Stores metadata of a block (e.g. CRC)
  - Serves data and metadata to Clients
- Block Report
  - Periodically sends a report of all existing blocks to the NameNode
- Facilitates Pipelining of Data
  - Forwards data to other specified DataNodes

Block Placement

- Current Strategy
  - One replica on local node
  - Second replica on a remote rack
  - Third replica on same remote rack
  - Additional replicas are randomly placed
- Clients read from nearest replicas

Heartbeats

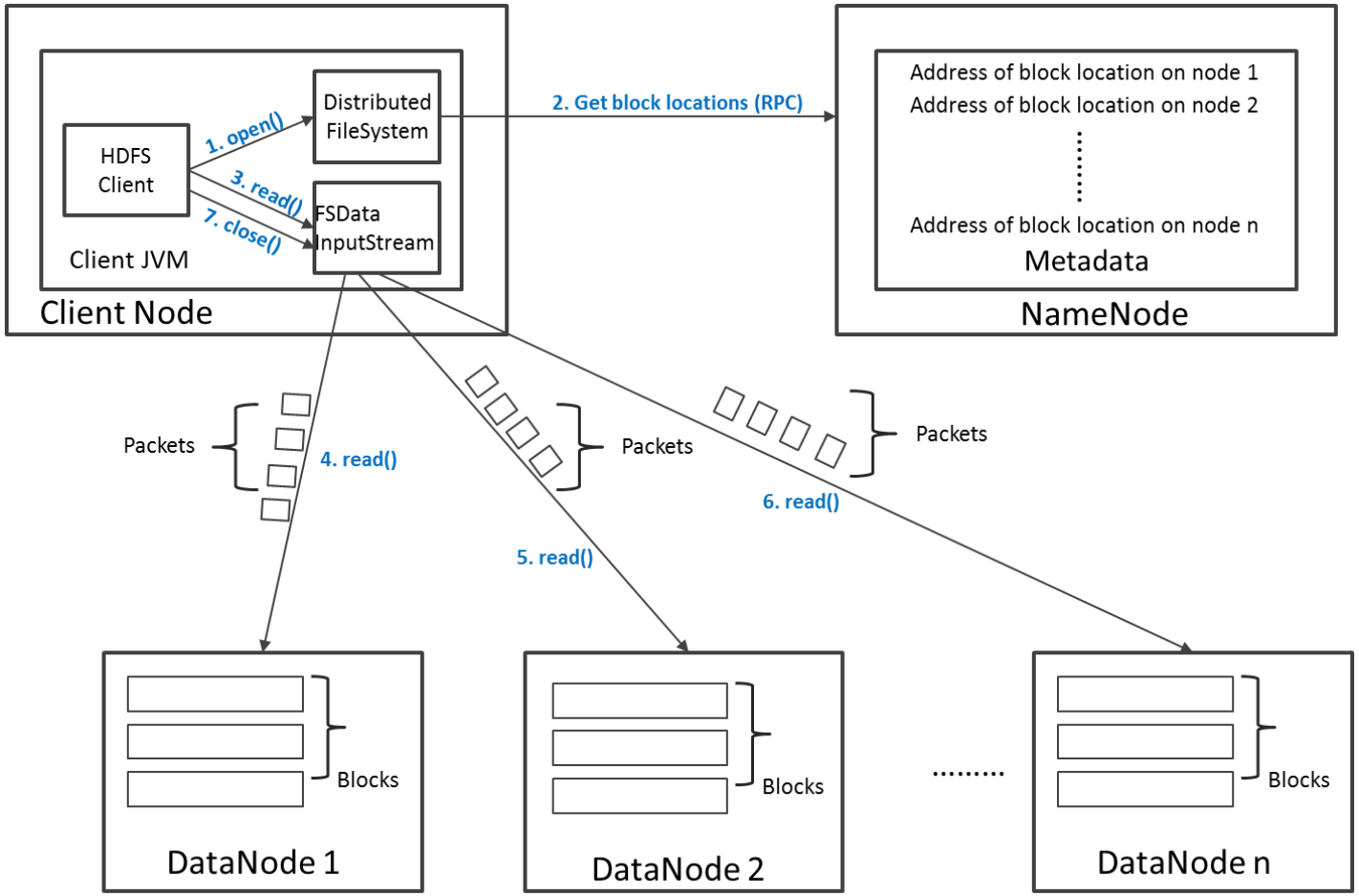
- DataNodes send hearbeat to the NameNode
  - Once every 3 seconds

- NameNode uses heartbeats to detect DataNode failure

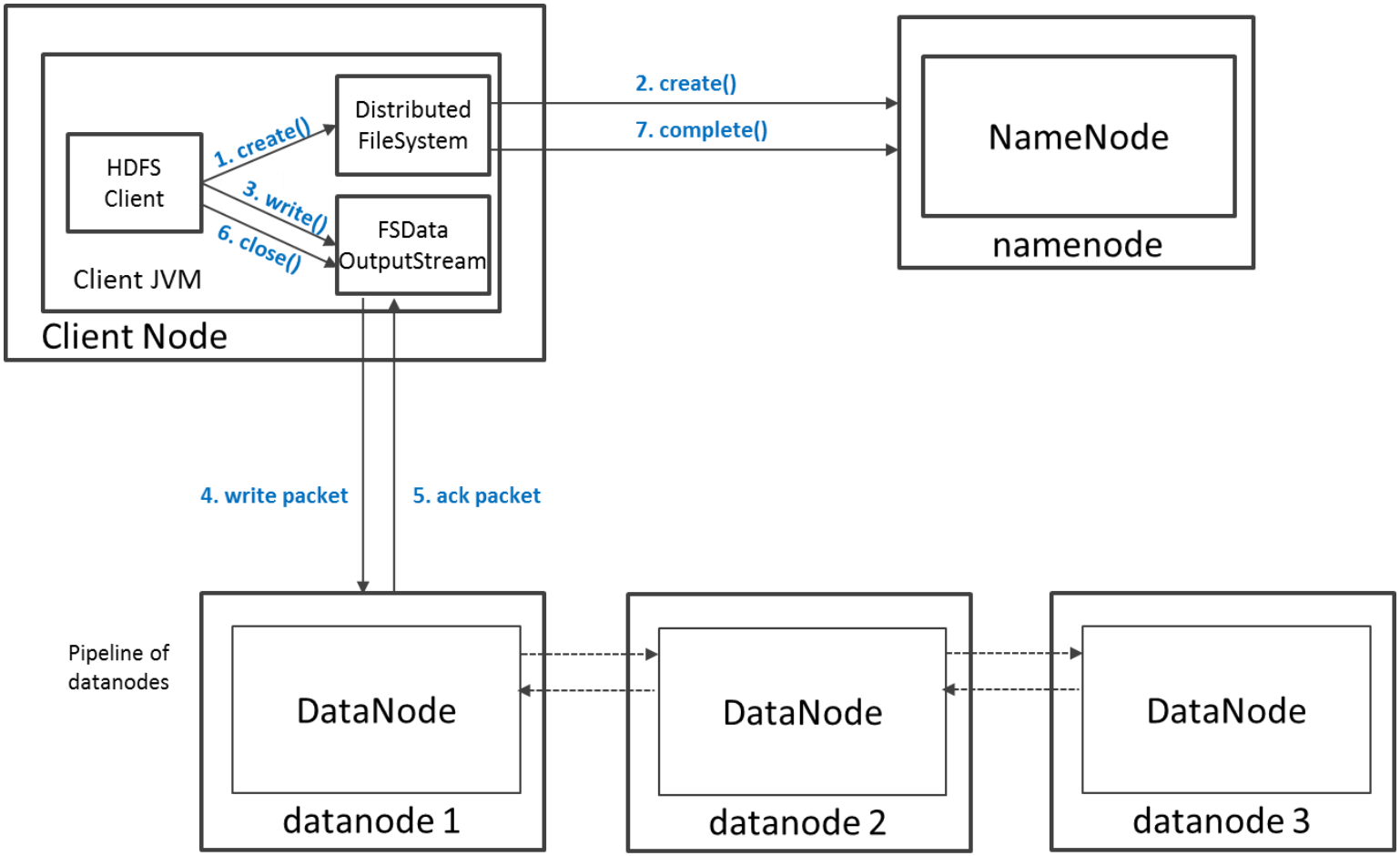
Replication Engine

- NameNode detects DataNode failures
  - Chooses new DataNodes for new replicas
  - Balances disk usage
  - Balances communication traffic to DataNodes

Read Operation in HDFS



Write Operation in HDFS



Unique features of HDFS

- Failure tolerant - data is duplicated across multiple DataNodes to protect against machine failures. The default is a replication factor of 3 (every block is stored on three machines).

- Scalability - data transfers happen directly with the DataNodes so your read/write capacity scales fairly well with the number of DataNodes

- Space - need more disk space? Just add more DataNodes and re-balance

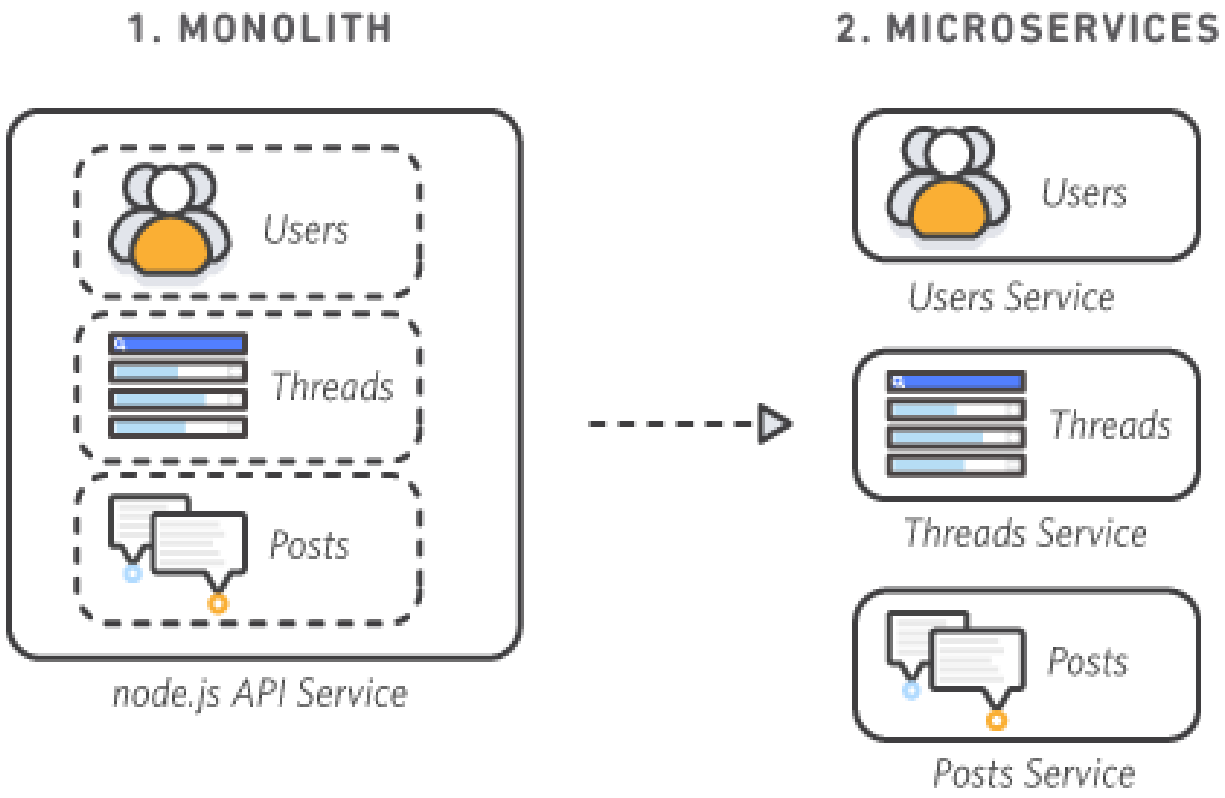
- Industry standard - Other distributed applications are built on top of HDFS (HBase, Map-Reduce)

HDFS is designed to process large data sets with write-once-read-many semantics, it is not for low latency access

Microservices

- **Microservices are an architectural style that develops a single application as a set of small services.**
- **Each service runs in its own process.**
- **The services communicate with clients, and often each other, using lightweight protocols, often over messaging or HTTP.**
- These services are owned by small, self-contained teams.
- Microservices architectures make applications easier to scale and faster to develop, enabling innovation and accelerating time-to-market for new features.

Monolithic vs. Microservices Architecture



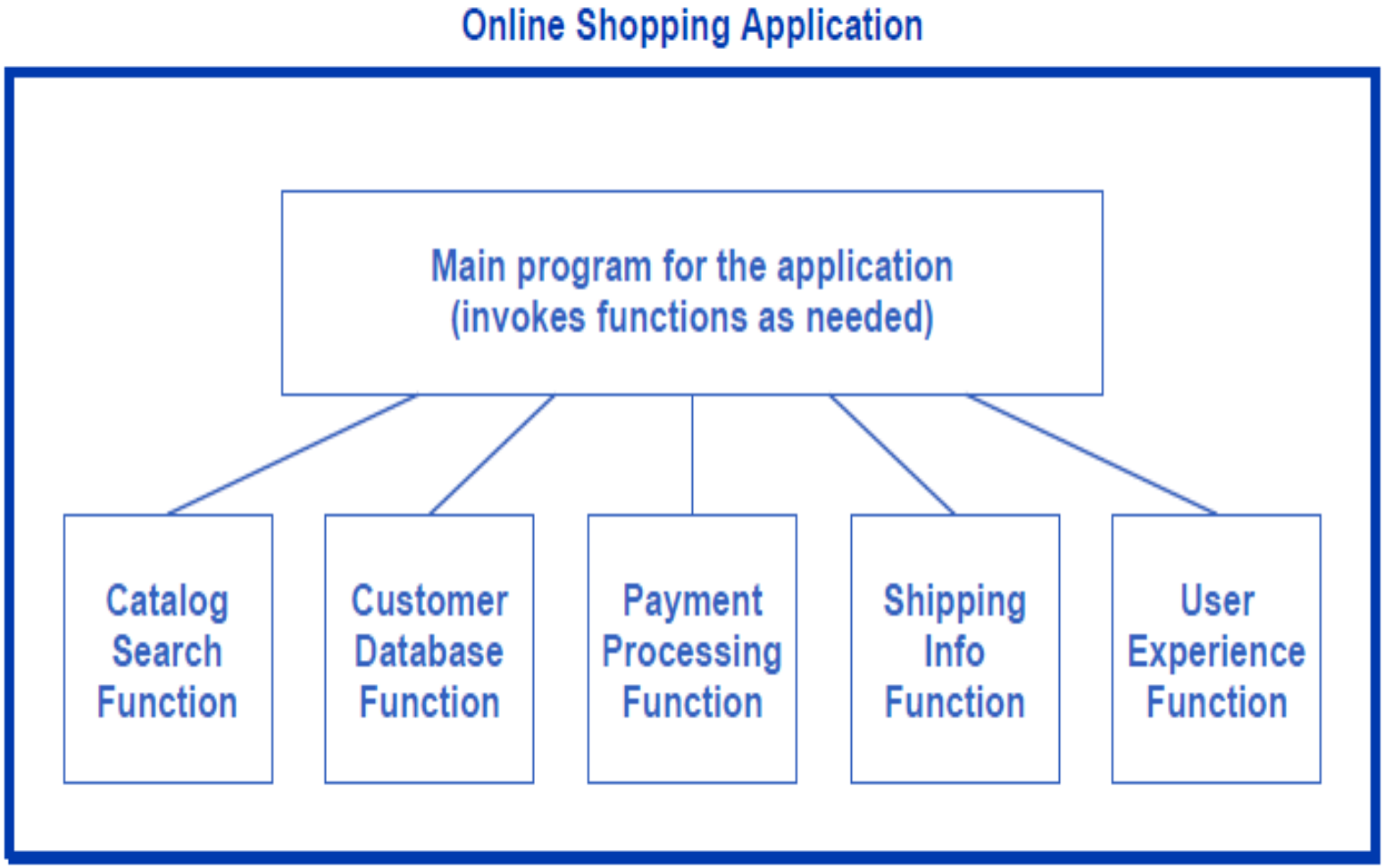
Monolithic vs. Microservices Architecture

- With monolithic architectures, all processes are tightly coupled and run as a single service.
- This means that if one process of the application experiences a spike in demand, the entire architecture must be scaled.
- Adding or improving a monolithic application's features becomes more complex as the code base grows.

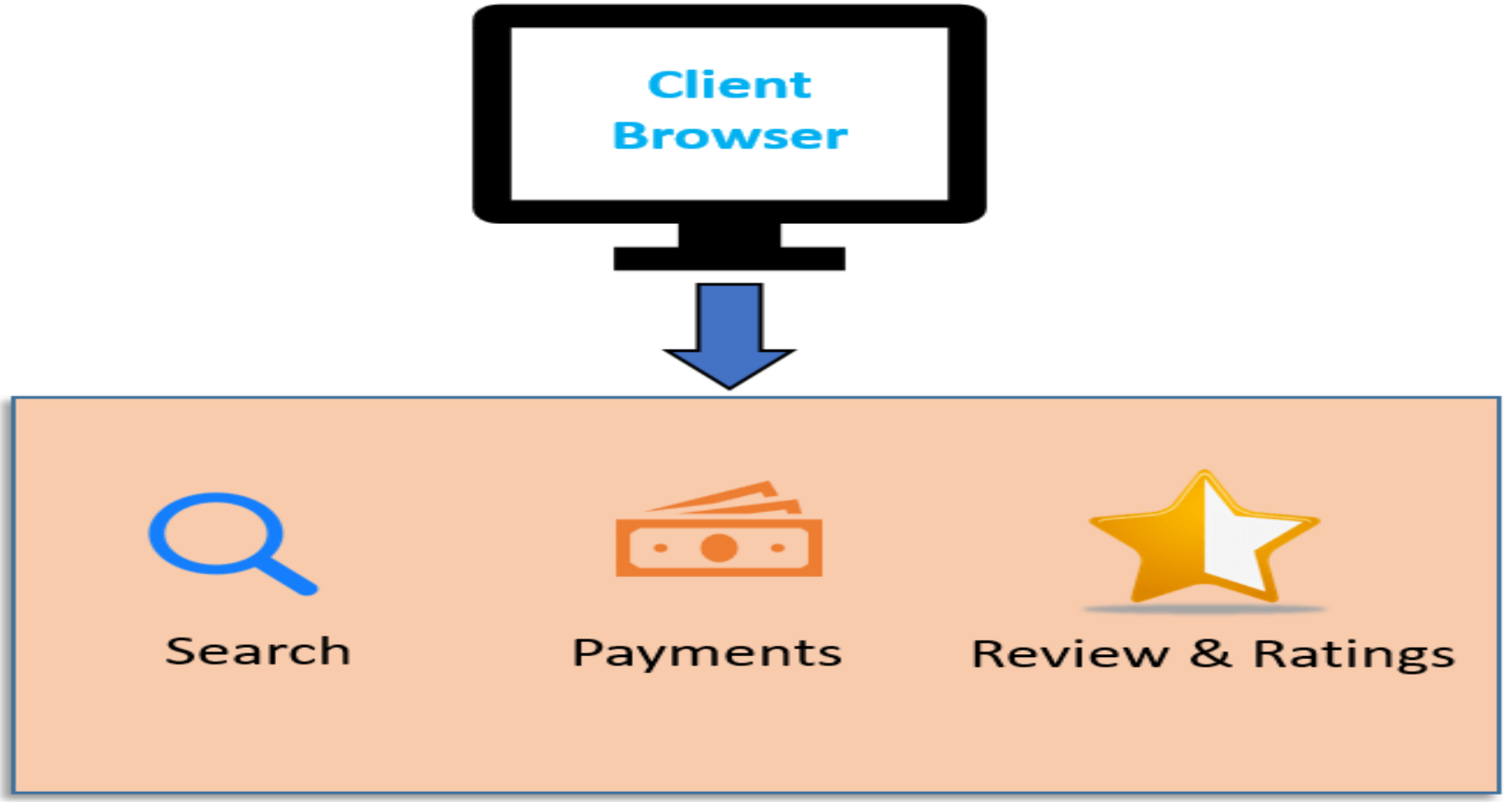
Monolithic vs. Microservices Architecture

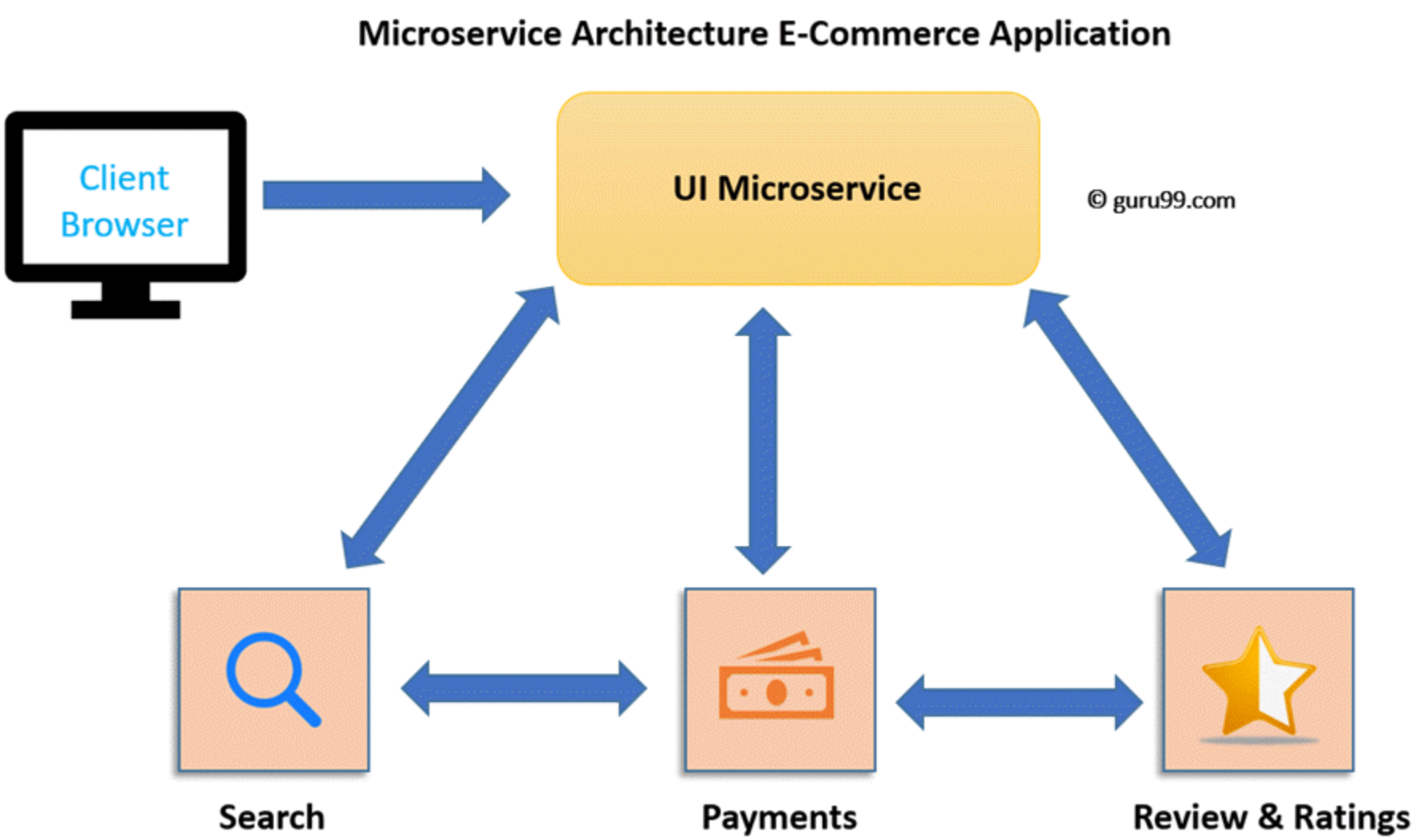
- With a microservices architecture, an application is built as independent components that run each application process as a service.
- These services communicate via a well-defined interface using lightweight APIs.
- Services are built for business capabilities and each service performs a single function.
- Because they are independently run, each service can be updated, deployed, and scaled to meet demand for specific functions of an application.

Monolithic application with all the functions



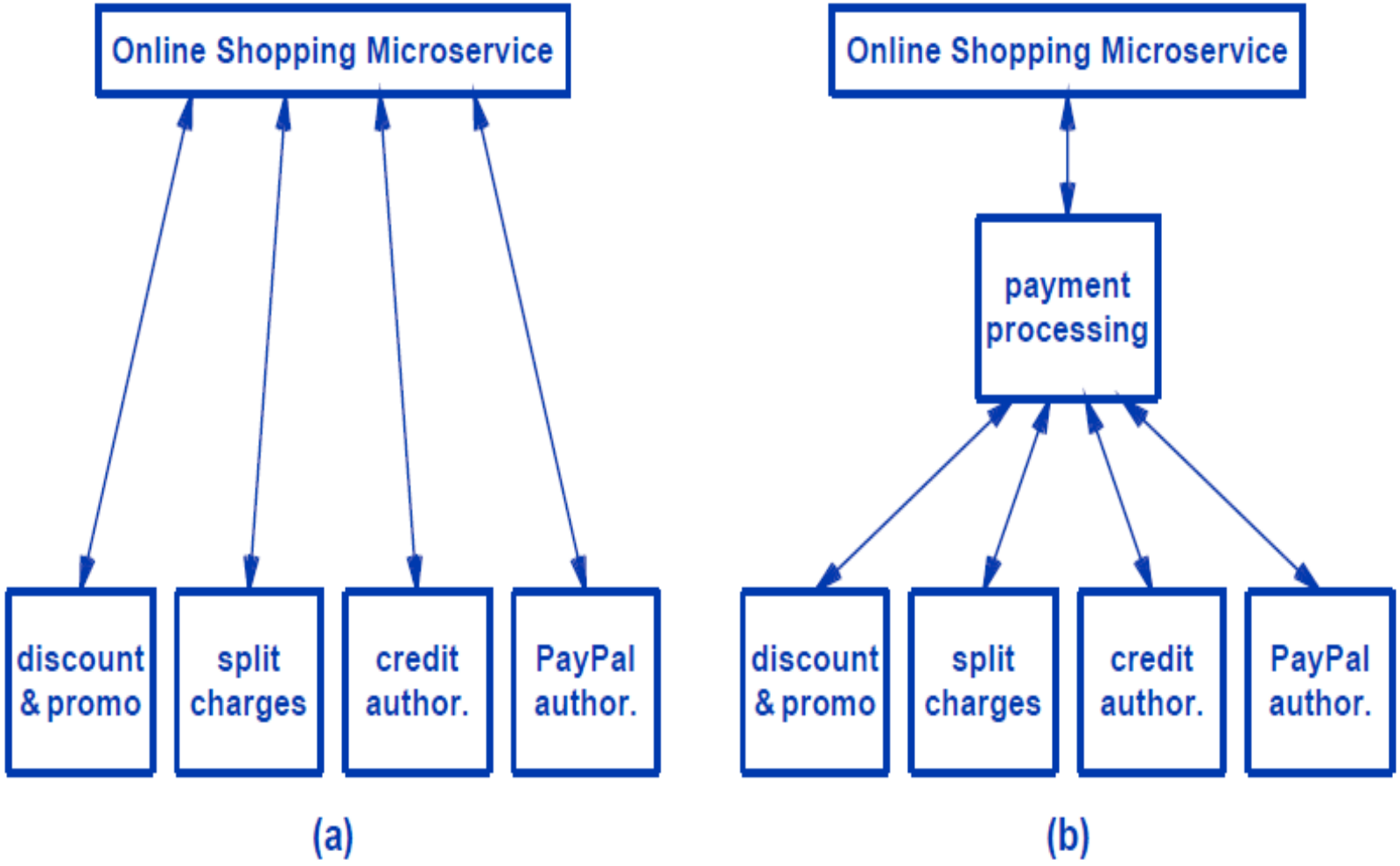
Monolithic Architecture E-Commerce Application




Microservice Architecture E-Commerce Application
Client Browser
UI Microservice
© guru99.com
Search
Payments
Review & Ratings

Disaggregated into a set of microservices

Separate and Intermediate Micro Services



Advantages(software development)

- Smaller scope and better modularity
- Smaller teams
- Less complexity
- Choice of programming language
- More extensive testing

Communication Protocols Used For Micro services

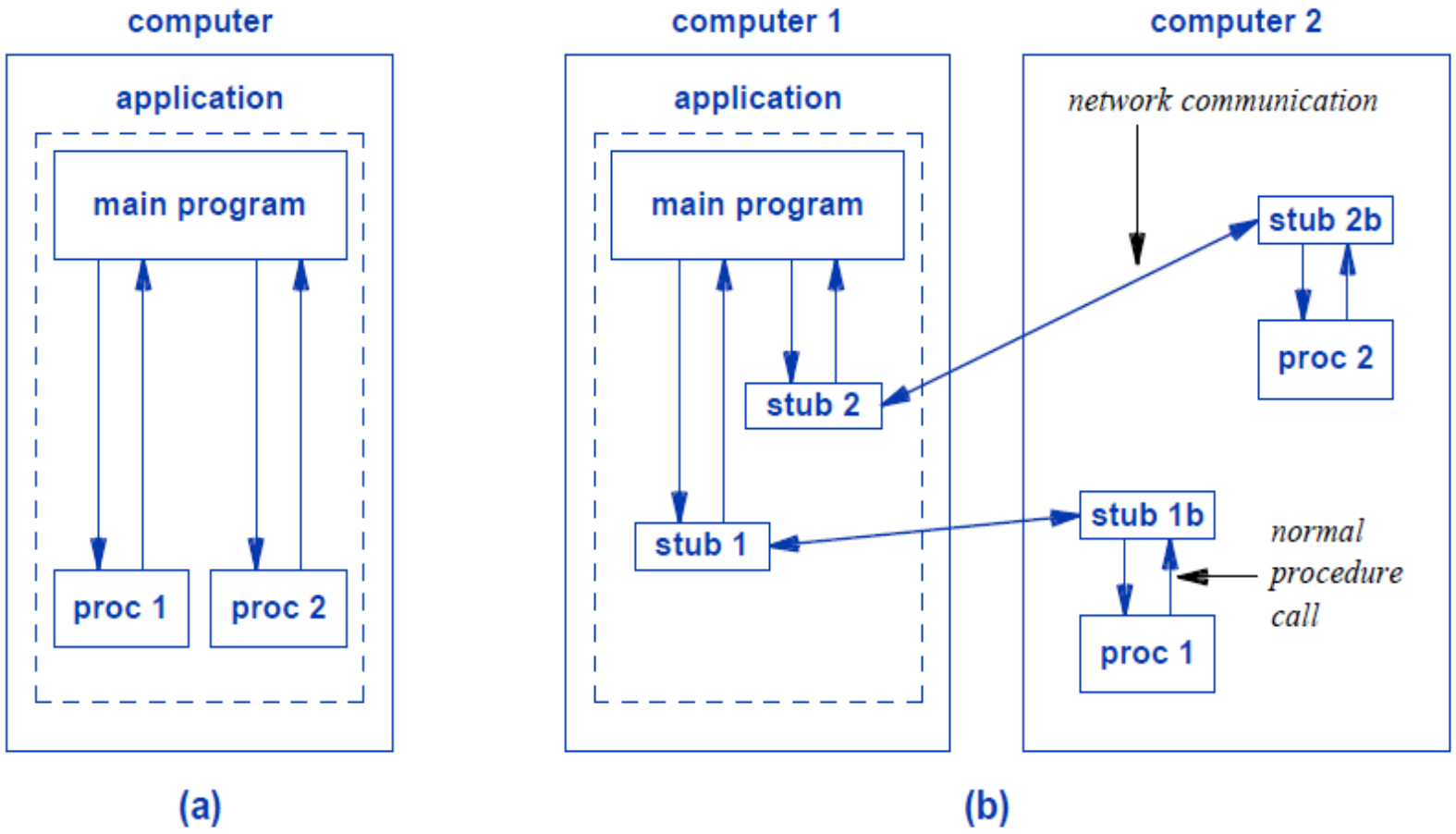
- Micro services use the *Transmission Control Protocol* (*TCP*), with TCP being sent in *Internet Protocol* (*IP*) packets.
- HTTP –The *Hypertext Transfer Protocol* used in the Web
- gRPC –An open source high-performance, universal RPC framework

(gRPC was originally created at Google and then moved to open source)

Six basic operations that can be used in HTTP messages

Operation	Meaning
GET	Retrieve a copy of the data item specified in the request
HEAD	Retrieve metadata for the data item specified in the request (e.g., the time the item was last modified)
PUT	Replace the specified data item with the data sent with the request
POST	Append the data sent with the request onto the specified data item
PATCH	Use data sent with the request to modify part of the data item specified in the request
DELETE	Remove the data item specified in the request

gRemote Procedure Call (RPC)



Distinction between traditional RPC and gRPC

- A traditional RPC follows a request-response interaction: each call to a remote procedure causes a single message to travel over the network to the computer containing the remote procedure and a single message to travel back.

- gRPC extends remote procedure call to allow a remote procedure to stream multiple data items in response to a request.

Communication Among Microservices

- Request-response (REST interface)

REST or RESTful to describe the *request- response* style of interaction used on the Web, a web browser sends a request to which a webserver responds.

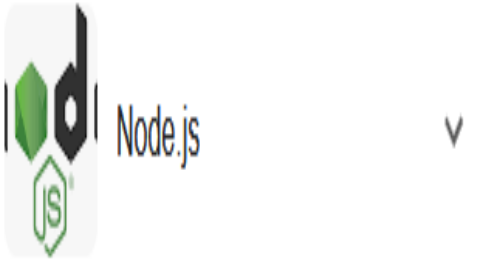
- Data streaming (continuous interface)

When using a streaming interface, an entity establishes a network connection with the microservice and sends a single request. The microservice sends a sequence of one or more data items in response to the request (i.e., the microservice streams a sequence of data items).

Some micro services follow the *publish-subscribe* variant of data streaming

The name arises because the senders are said to “publish” information on various topics, and receivers are said to “subscribe” to specific topics.

Technologies for microservices



Serverless Computing

- Serverless architecture is largely based on a Functions as a Service (FaaS) model that allows cloud platforms to execute code without the need for fully provisioned infrastructure instances.

- FaaS, also known as Compute as a Service (CaaS), are stateless, server-side functions that are event-driven, scalable, and fully managed by cloud providers.

- *Scale to infinity*

- *Scale to zero*

AWS Lambda

- AWS Lambda lets you run code without provisioning or managing servers. You pay only for the compute time you consume - there is no charge when your code is not running.
- With Lambda, you can run code for virtually any type of application or backend service - all with zero administration.
- Just upload your code and Lambda takes care of everything required to run and scale your code with high availability.
- You can set up your code to automatically trigger from other AWS services or call it directly from any web or mobile app.

Examples of serverless

Coca-Cola:

- Soft drink giant Coca-Cola has enthusiastically embraced serverless after its implementation in vending machines resulted in significant savings.

- Whenever a beverage is purchased, the payment gateway makes a call to the AWS API Gateway and triggers an AWS Lambda function to complete the transaction.

- Since vending machines must communicate with headquarters for inventory and marketing purposes, the ability to pay per request rather than operating at full capacity had a substantial impact on reducing costs.

Serverless services

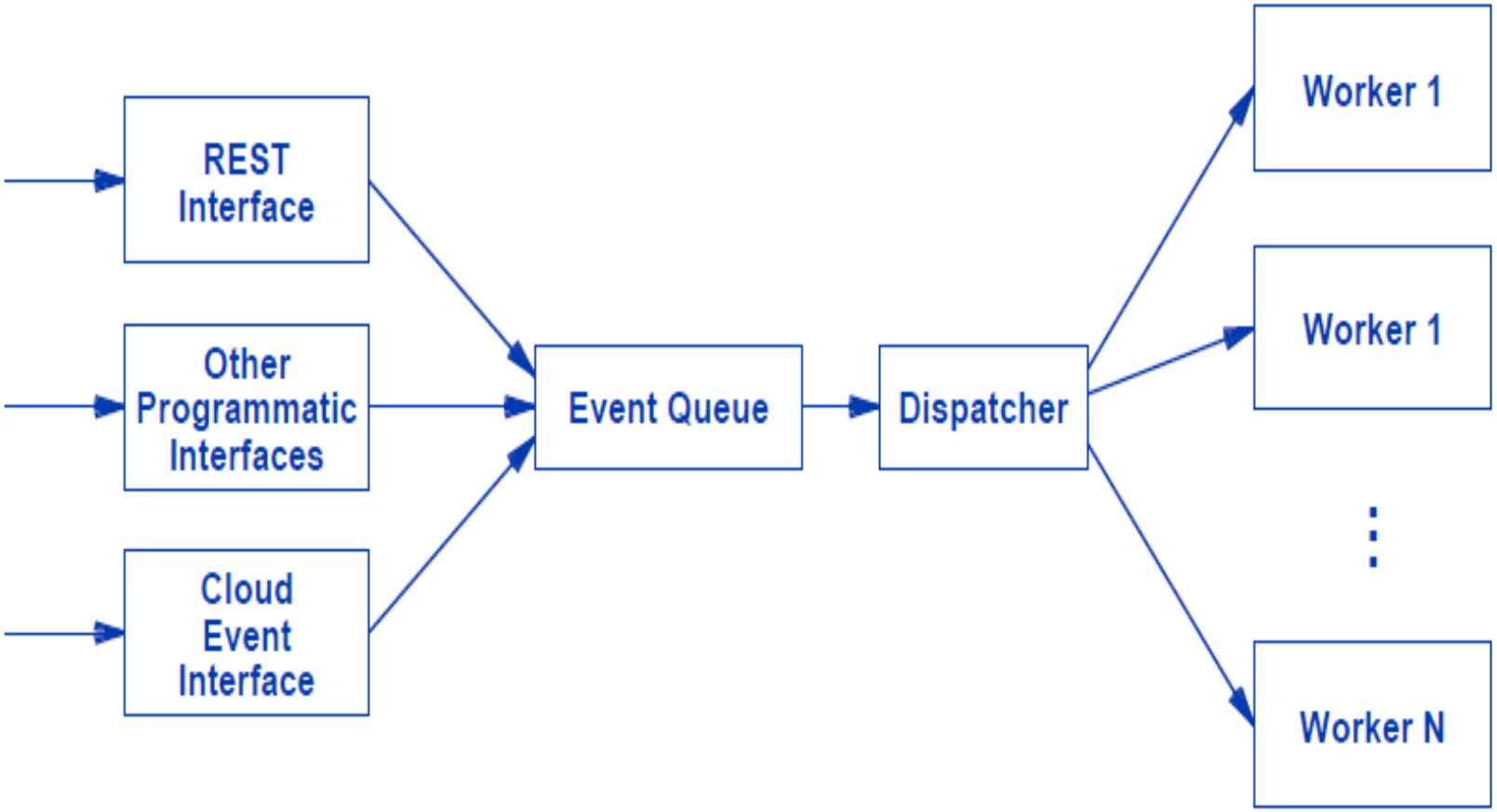
- AWS Lambda, Microsoft Azure Functions, Google Cloud Functions and IBM OpenWhisk are all well-known examples of serverless services.

- The convenience and cost-saving benefits associated with on-demand auto-scaling resources, and only paying for services as they're needed cloud providers.

Stateless Servers And Containers

- Serverless computing focuses on running a single, stateless function in each container (i.e., FaaS)
- It uses containers and can run on multiple physical servers, a server less computing system requires server code to be stateless.
- To handle microservices, containers are designed with a short lifetime: the container starts, performs one function, and exits. Thus, state information does not persist for more than one client connection
- Serverless systems count each server access as an event.

Architecture Of A Serverless Infrastructure

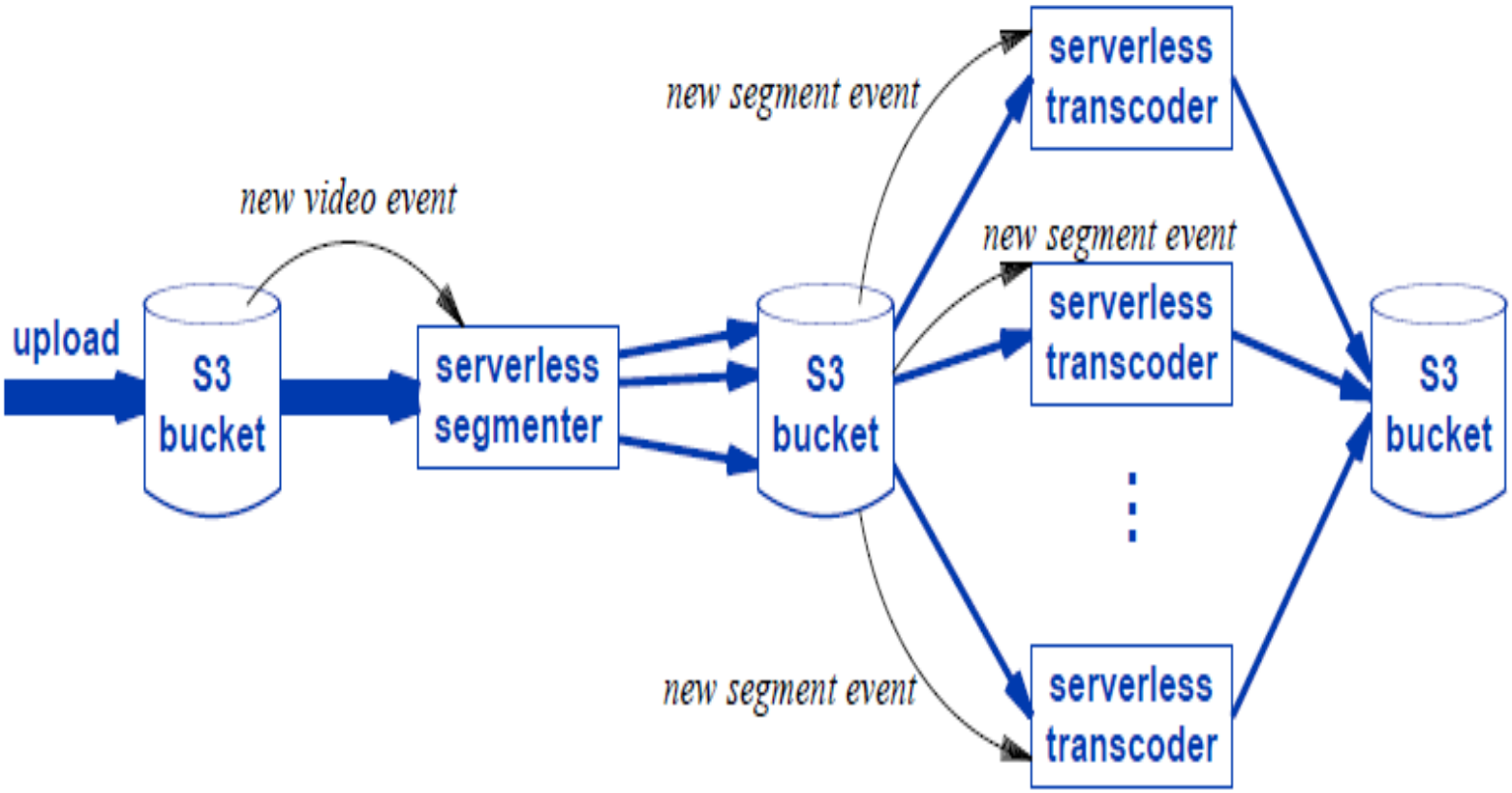


Example Of Serverless Processing

- Netflix uses the AWS *Lambda* event-driven facility for *video transcoding*, a step taken to prepare each new video for customers to download

Step	Action Taken
1.	A content provider uploads a new video
2.	A serverless function divides the new video into 5-minute segments
3.	Each segment is given to a separate serverless function for processing
4.	The processed segments are collected and the video is available for customers to access

Netflix transcoding system

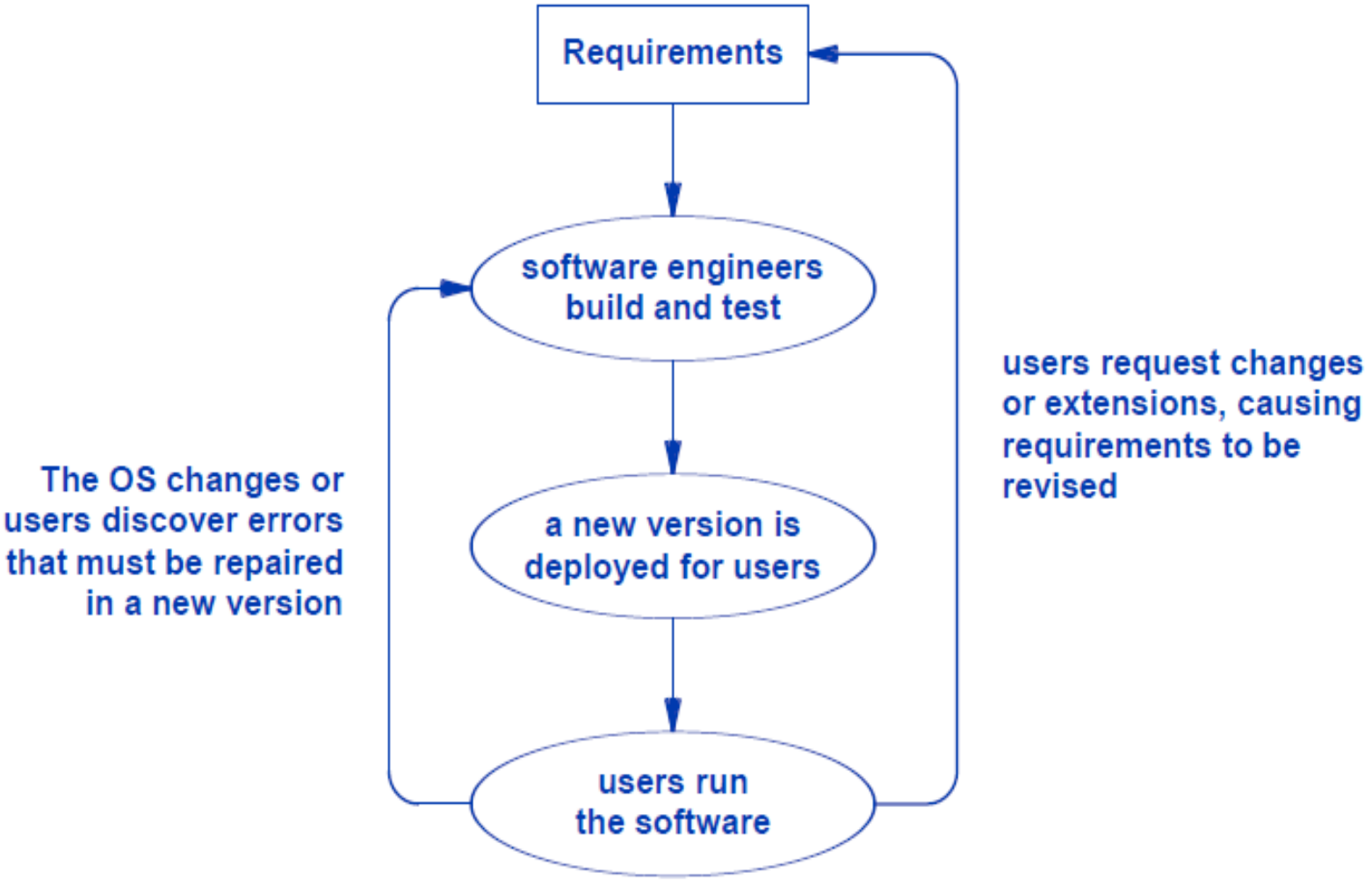


Software Creation And Deployment

- Step 1. An initial engineering effort creates, tests, and deploys a new piece of software, making it available for use.

- Step 2. Authorized users invoke the software as needed, possibly over a span of many years

Realistic Software Development Cycle



Large Software Projects And Teams

- Development Team

A list of requirements, create a design, specify all the pieces, choose a programming language (or languages) and programming tools, and write the code.

- Quality Assurance Team

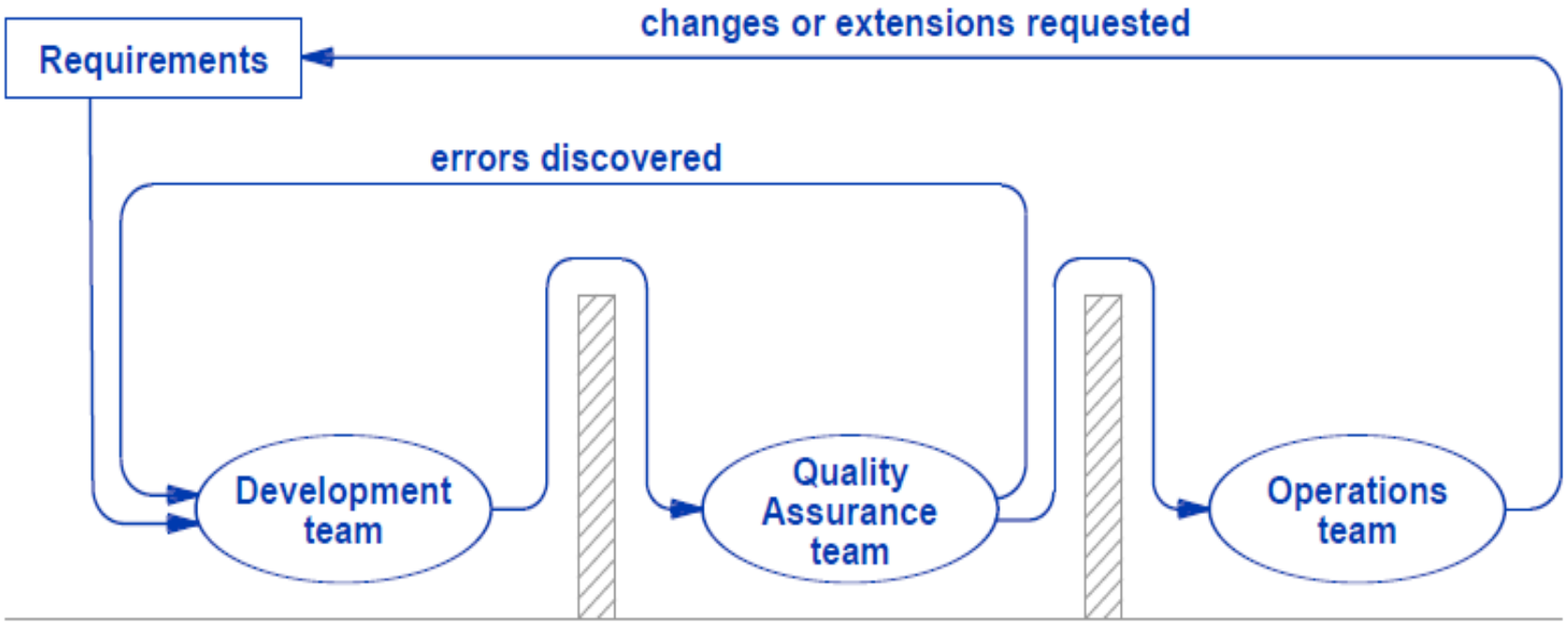
Extensive testing to ensure the software fulfills the requirements and works correctly

- Operations Team

Handles deployment, including configuring the set of users authorized to use the software, network addresses, and related details. Monitoring the execution, restarting after failure, planning back- ups, and creating multiple instances to handle scale.

Disadvantages Of Using Multiple Teams

- Conflicting management objectives
- Competition and finger pointing
- Long development cycle

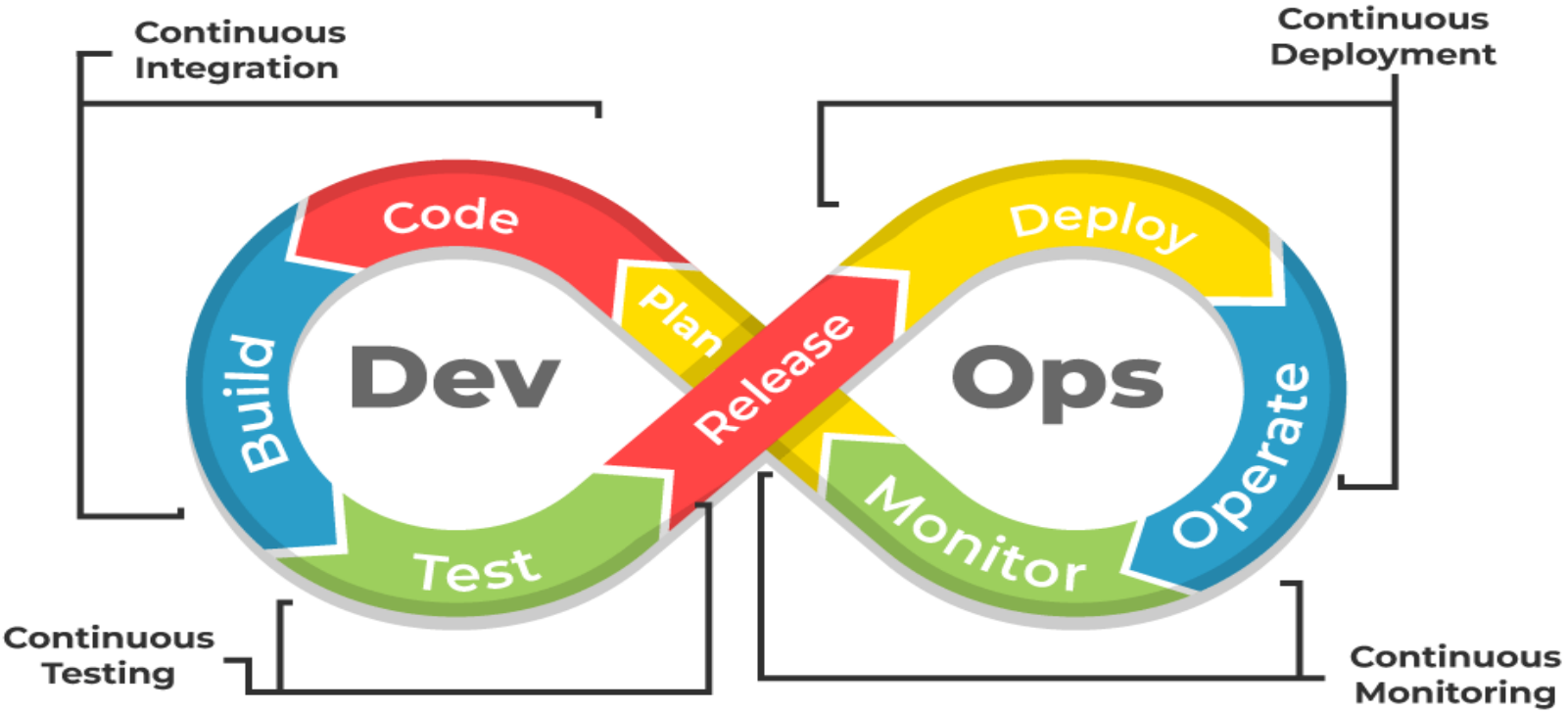


DevOps Approach

- **DevOps** is a collection of two words, "**Development**" and "**Operations**".
- Emphasizes collaboration between development and operations teams to streamline the entire software delivery lifecycle.
- The *DevOps* methodology adopts and extends the Agile approach.
- DevOps breaks down silos between development and operations teams to enable seamless communication, faster time-to-market, and improved customer satisfaction.

DevOps Approach

- Every phase of the software development lifecycle, including planning, coding, testing, deployment, and monitoring, is heavily automated in DevOps.

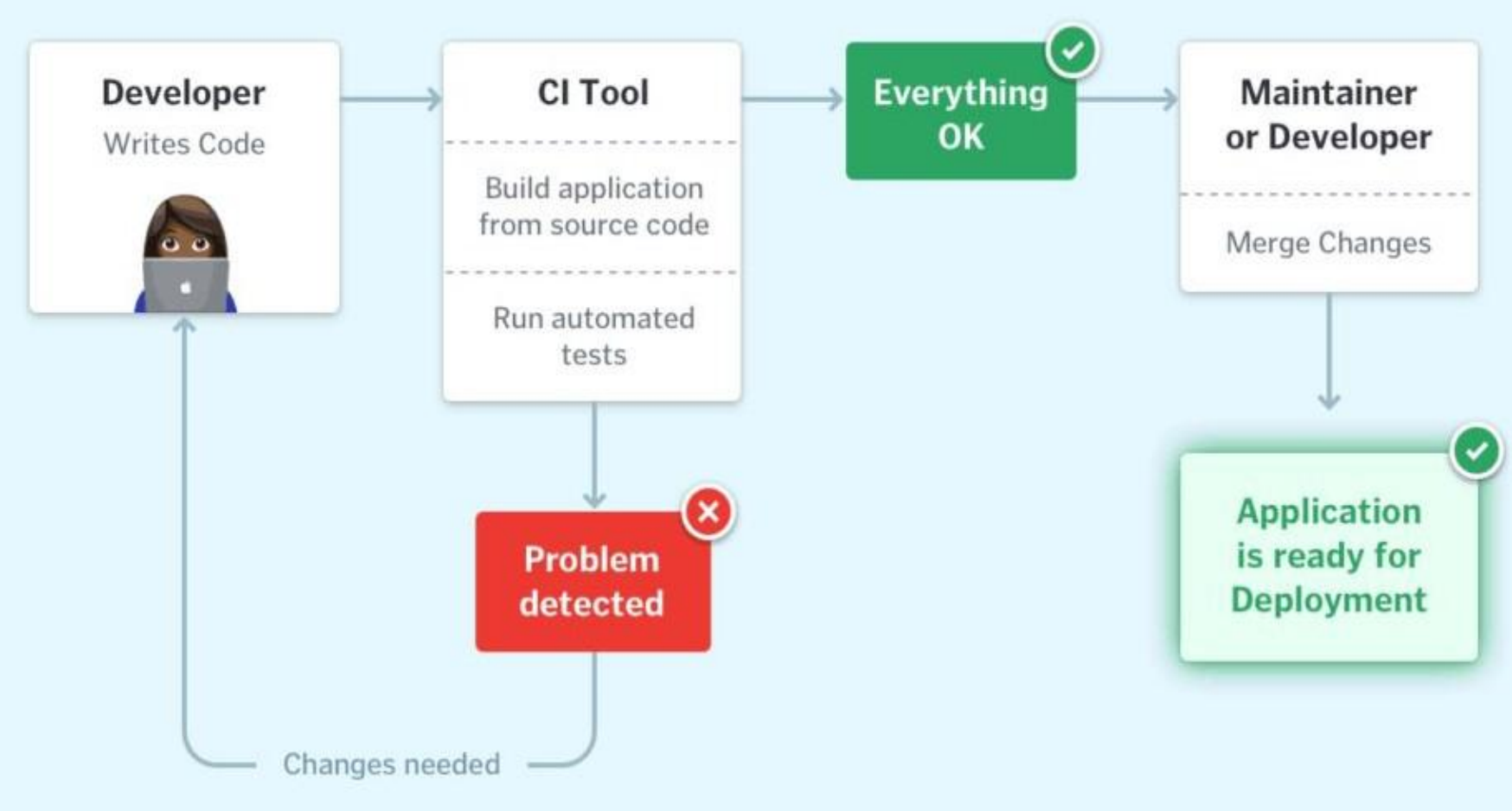


Continuous Integration

- Development team makes small changes in the software continuously rather than waiting to collect changes into a major release.
- *To accelerate the software development cycle, DevOps divides software into small pieces that can be changed and tested quickly.*

DevOps Lifecycle

How CI Works?


Developer
Writes Code
CI Tool
Build application from source code
Run automated tests
Everything OK
Maintainer or Developer
Merge Changes
Application is ready for Deployment
Problem detected
Changes needed

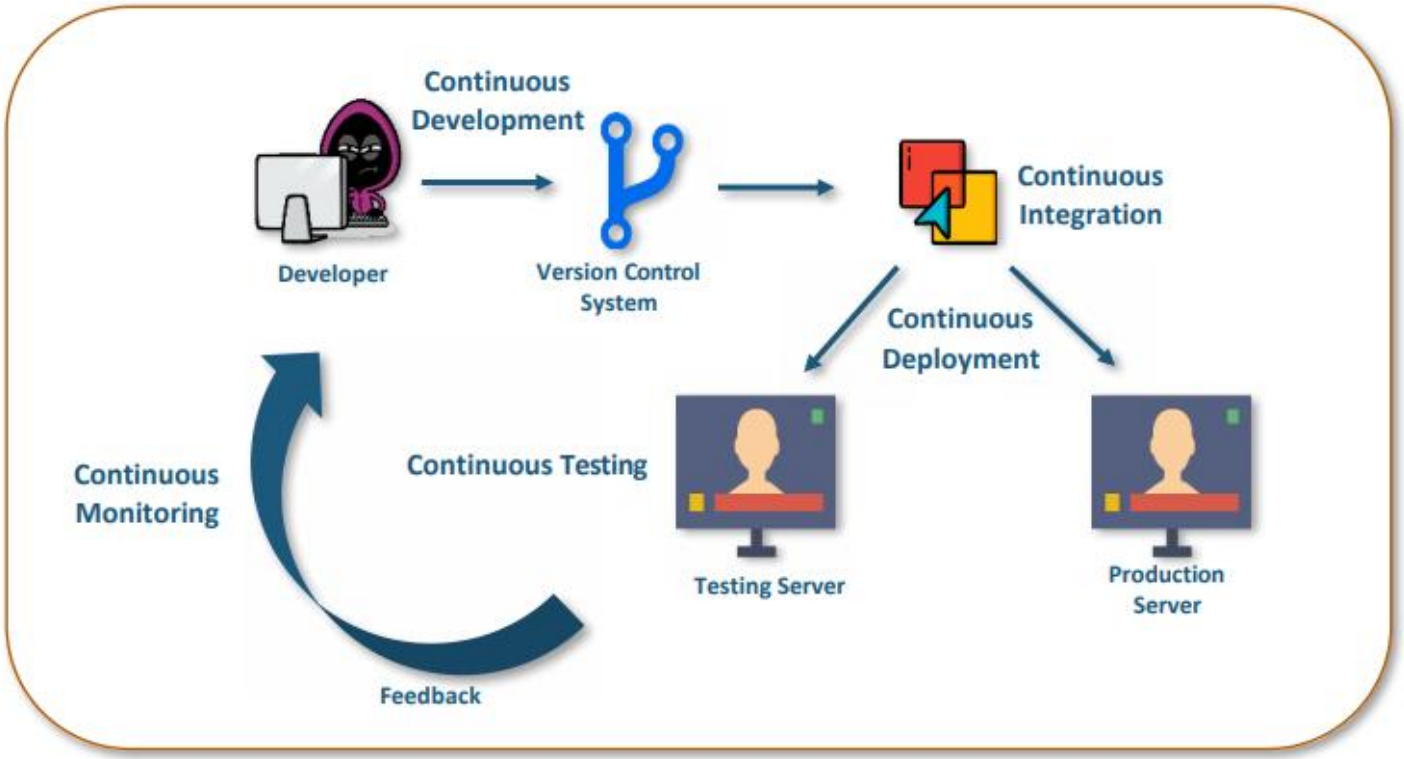
Continuous Delivery (CD): Deploying Versions Rapidly

- In a traditional environment, the Operations team chooses a specific time and date to deploy each major software release.
- To minimize disruption, the team schedules a release to occur when use is minimal, such as at night or on a weekend.
- The idea of scheduling a time for each major release does not work well in a cloud environment.
- Think of releasing a new version of microservice *M*. Many other services may depend on *M* being available.

CD or Continuous Delivery

- Process that begins after Continuous Integration (CI).
- All the codes that are from CI are taken for production.
- The CD process begins by developing, building, and testing the CI.
- **CD** or **Continuous Delivery** is carried out after Continuous Integration to make sure that we can release new changes to our customers quickly in an error-free way.
- This includes running integration and regression tests in the staging area (similar to the production environment) so that the final release is not broken in production.
- Continuous Delivery automates the entire software release process.
- Some popular CD tools are AWS CodeDeploy, Jenkins, and GitLab.

Automated CI & CD



Tools for DevOps

- Jenkins
- Docker
- Ansible
- Puppet
- Chef
- Git
- Kubernetes
- Terraform
- Nagios
- Bamboo
- Selenium
- Splunk
- Vagrant
- Prometheus
- GitLab
- Gradle
- CircleCI
- Jira

Cloud Orchestration

- Industry uses the term *orchestration* to refer to an automated system that coordinates the many subsystems needed to configure, deploy, operate, and monitor software systems and services.

- Cloud orchestration is likewise useful in cloud provisioning, empowering organizations to automate the delivery of important cloud resources to their end users.

- Orchestration in which an automated system configures, controls, and manages all aspects of a service.

A move to containers

- **Rapid creation**
  less time to create a container than to create a VM

- **Short lifetime**
  Unlike a VM that remains in place semi-permanently once created, a container is ephemeral.

  A container resembles an application process: a typical container is created when needed, performs one application task, and then exits.

- **Replication**
  Replication is key for containers.

When demand for a particular service increases, multiple containers for the service can be created and run simultaneously, analogous to creating multiple concurrent processes to handle load.

When demand for a service declines, unneeded container replicas can be terminated.

Orchestrator

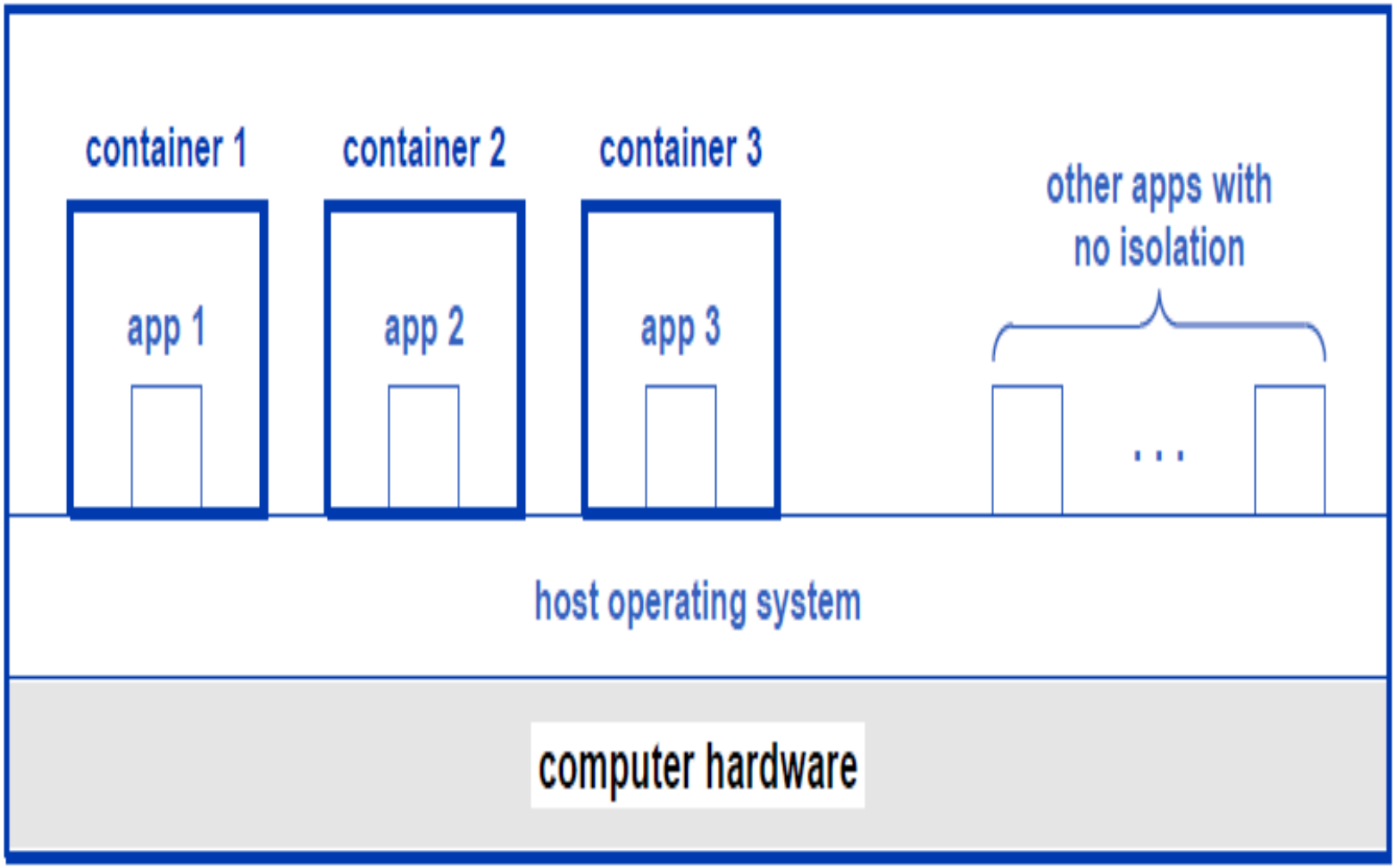
- Dynamic scaling of services
- Coordination across multiple servers
- Resilience and automatic recovery

Container Approach For Isolated Apps

- When an app uses operating system mechanisms to enforce isolation, the app remains protected from other apps.

- *Each container provides isolation, which means an application in one container cannot interfere with an application in another container.*

Container Approach For Isolated Apps



Docker

- Docker is a set of platform as a service products that use OS-level virtualization to deliver software in packages called containers.
- **Docker provides the ability to package and run an application in a loosely isolated environment called a container.**
- Using Docker, you can quickly deploy and scale applications into any environment and know your code will run.
- **Docker is primarily a platform for building, packaging, and running containerized applications.**
- **It provides tools for creating and managing container images, as well as for running containers on a single host.**

Docker Containers

- Tools that enable rapid and easy development of containers.
- An extensive registry of software for use with containers.
- Techniques that allow rapid instantiation of an isolated app.
- Reproducible execution across hosts.

Development tools

- The Docker technology provides an easy way to develop apps that can be deployed in an isolated environment.

- Docker uses a high-level approach that allows a programmer to combine large pre-built code modules into a single image that runs when a container is deployed.

- A separate image file must be created for each app. When an imagefile runs,a container is created to run the app. We say the app has been *containerized*.

Extensive registry of software

- *Docker Hub*, an extensive registry of open source software that is ready to use.
- The registry enables a programmer to share deployable apps.
- A user (or an operator) can combine pieces from the registry in the same way that a conventional program uses modules from a library.

Rapid instantiation

- Docker uses an early binding approach that combines all the libraries and other run-time software that will be needed to run the container into an image file.
- Unlike a VM that must wait for an operating system to boot, a container can start instantly.
- Early binding approach means a Docker container does not need the operating system to perform extra work when the container starts.
- As a result, the time required to create a container is impressively small.

Reproducible execution

- Once a Docker container has been built, the container image becomes *immutable.*
- The image remains unchanged, independent of the number of times the image runs in a container.
- Furthermore, because all the necessary software components have been built in, a container image performs the same on any system.
- As a result, container execution always gives reproducible results.

Docker Containers

- *Docker technology makes app development easy, offers a registry of pre-built software, optimizes the time required to start a container, and guarantees reproducible execution.*

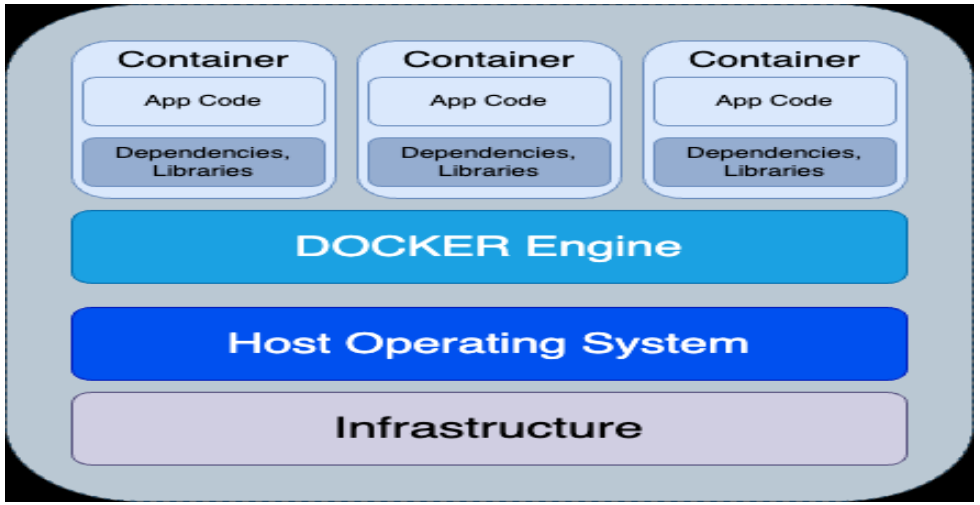
Docker Terminology And Development Tools

Docker Term	Meaning	Analogous To
image	A file that contains binary code for a container along with its dependencies	a.out file (MacOS/Linux) .exe file (Windows)
layer	One piece of software added to an image as it is built	a software module
container	An instance of an image	an executing process
Dockerfile	A specification for how to build an image	Makefile (Linux)
docker build	A command that constructs an image according to a Dockerfile	"make" program (Linux)
docker run X	A command that runs image X as a container	launch app X

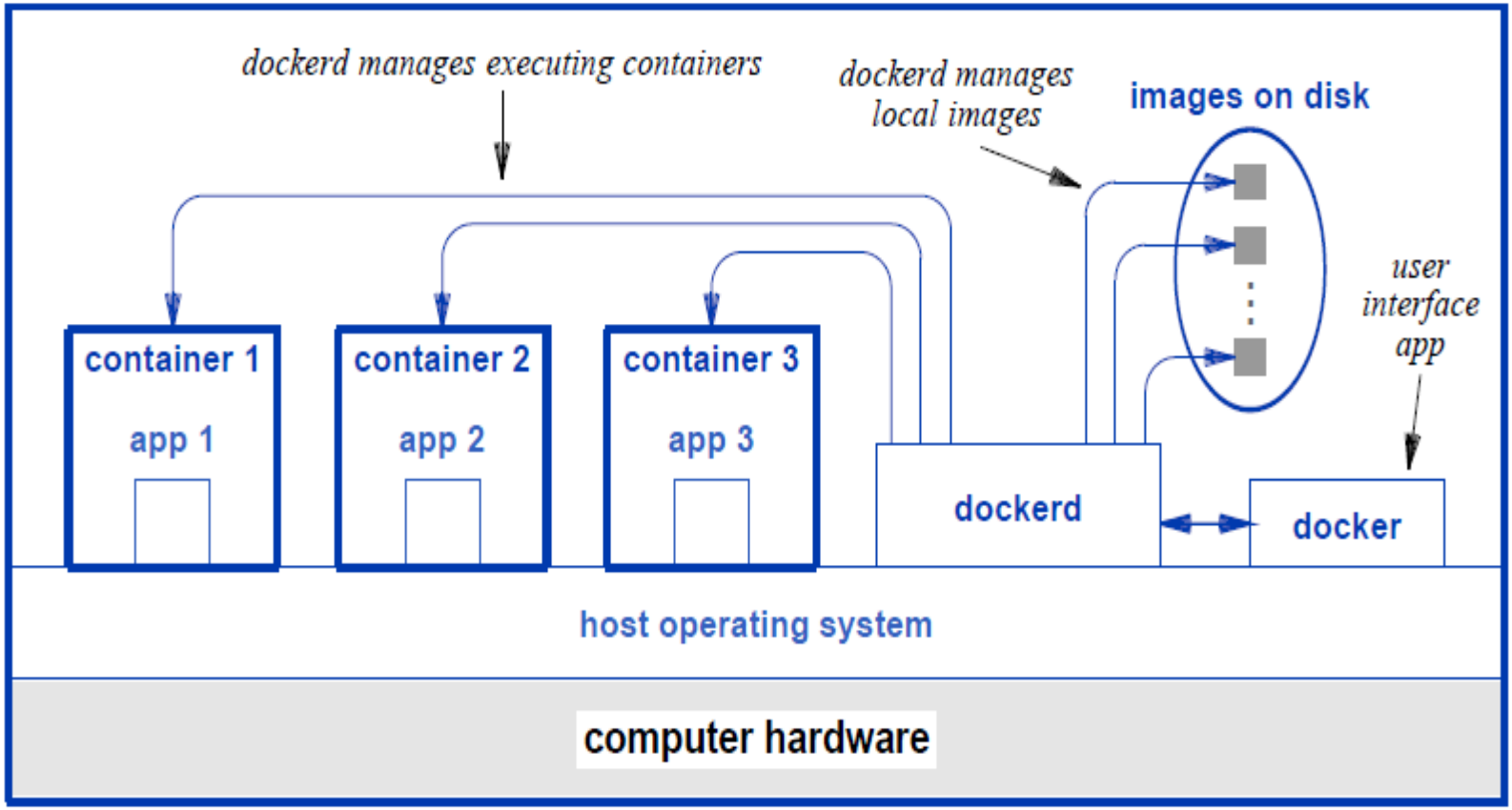
Docker Software Components

- Containers operate under the control of an application known as a *Docker daemon* (*dockerd*).

- Docker provides a user interface program, Collectively, the software is known as the ***Docker Engine*.**



Docker daemon, *dockerd*



Docker daemon, *dockerd*

- The dockerd program, which remains running in background at all times, contains several key subsystems.

- dockerd provides two interfaces through which a user can make requests and obtain information

  RESTful interface

  CommandLineInterface(CLI)

Docker commands

Command	Meaning
docker build .	Read Dockerfile in the current directory and follow the directives to construct an image
docker images	List all images on the local machine
docker run	Create and start a container running a specified image
docker ps	List all currently running containers on the system
docker pull	Download a copy of an image from a registry
docker stop	Stop one or more running containers
docker start	Restart one or more stopped containers

How Does Docker Work?

Docker Engine uses a Client Server Architecture.

Kubernetes (K8s)

- Kubernetes, often abbreviated as K8s(kates), is an open-source container orchestration platform designed to automate the deployment, scaling, and management of containerized applications. across a **cluster of machines**.

- Originally developed by Google and later donated to the Cloud Native Computing Foundation (CNCF)

- It focuses on automating container orchestration, scaling, and high availability.

Key features and concepts of Kubernetes

- **Container Orchestration**:

- Kubernetes provides a way to manage and orchestrate containers, primarily Docker containers, but it can also support other container runtimes.

- It handles the deployment and scaling of containers, ensuring high availability and efficient resource utilization.

Cluster Management

Kubernetes organizes containers into clusters, which are made up of one or more nodes (physical or virtual machines).

Clusters provide a framework for deploying and managing containerized applications.

Service naming and discovery

- Kubernetes allows a service to be accessed through a domain name or an IP address.
- Once a name or address has been assigned, applications can use the name or address to reach the container that runs the service
- Typically, names and addresses are configured to be global, allowing applications running outside the data center to access the service.

Load balancing

- Kubernetes does not limit a service to a single container.
- Instead, if traffic is high,
- Kubernetes can automatically create multiple copies of the container for a service, and use a *load balancer* to divide incoming requests among the copies.

Storage orchestration

- Kubernetes allows an operator to mount remote storage automatically when a container runs.

- The system can accommodate many types of storage, including local storage and storage from a public cloud provider.

Optimized container placement

- When creating a service, an operator specifies a cluster of servers(called *nodes*) that Kubernetes can use to run containers for the service.

- The operator specifies the processor and memory (RAM) that each container will need.

- Kubernetes places containers on nodes in the cluster in a way that optimizes the use of servers.

Automated recovery

- Kubernetes manages containers.
- After creating a container, Kubernetes does not make the container available to clients until the container is running and ready to provide service.
- Kubernetes automatically replaces a container that fails, and terminates a container that stops responding to a user-defined health check.

  (**Self-Healing**)

Rolling Updates and Rollbacks

- Kubernetes supports rolling updates, enabling zero-downtime updates for applications.
- A user can create a new version of a container image, and tell Kubernetes to start replacing running containers with the new version
- If issues arise during an update, Kubernetes allows for rollbacks to the previous working version.

Management of configurations and secrets

- Kubernetes separates management information from container images, allowing users to change the information needed for configuration and management without rebuilding container images.

- Kubernetes allows one to store sensitive information, such as passwords, authentication tokens, and encryption keys.

Limits On Kubernetes Scope

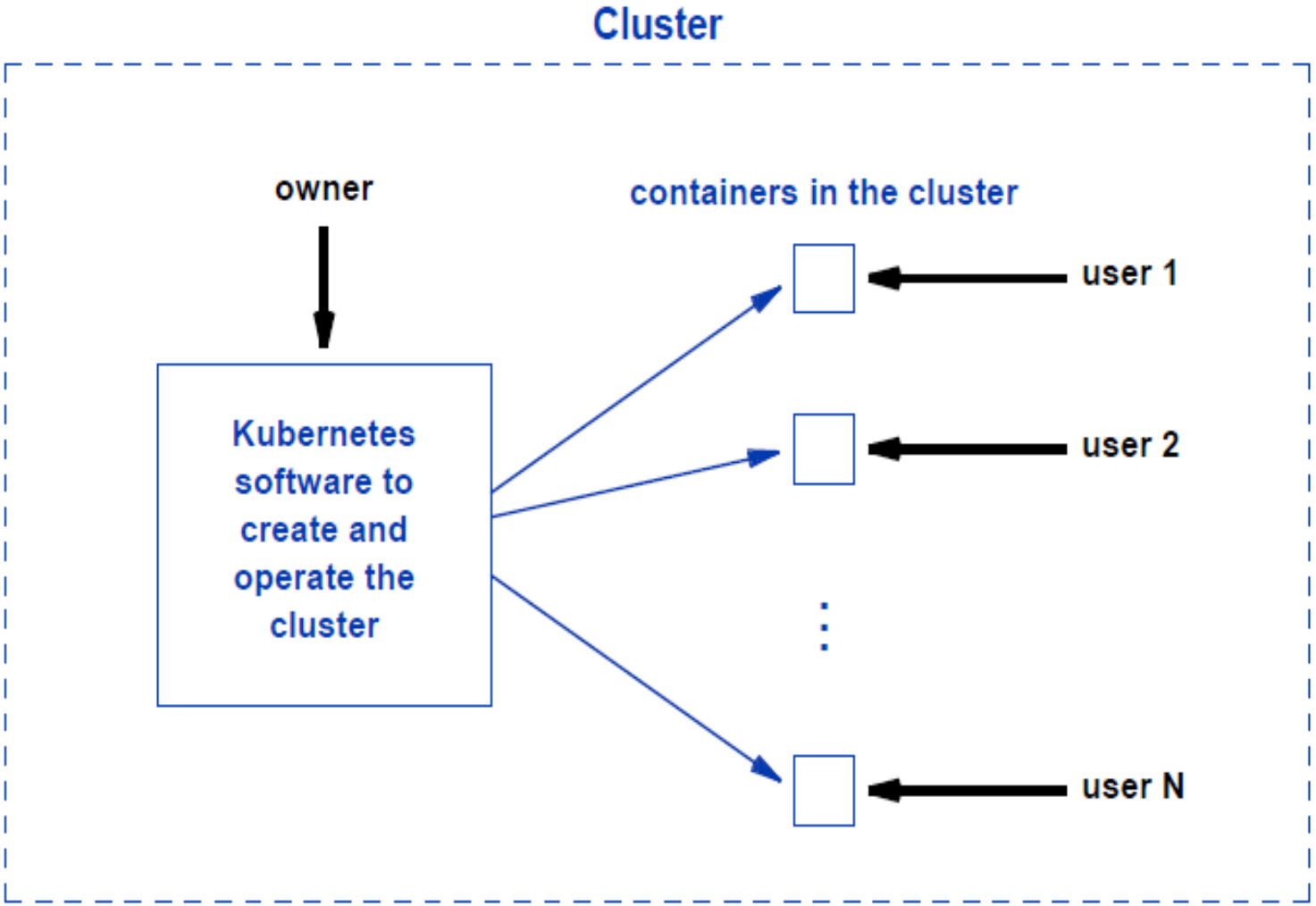
- Does not focus on a specific application type
- Does not manage source code or build containers
- Does not supply event-passing middleware
- Does not handle monitoring or event logging

Kubernetes Cluster Model

- Kubernetes uses the term *cluster* to describe the set of containers plus the associated support software used to create, operate, and access the containers.

- The number of containers in a cluster depends on demand, and Kubernetes can increase or decrease the number as needed.

Conceptual organization of a Kubernetes cluster



Kubernetes Pods

- “Pod" is the smallest deployable unit that can hold one or more containers.
- Pods are the basic building blocks of the Kubernetes object model and represent a single instance of a running process in a cluster.
- Containers within the same Pod share the same network namespace, which means they can communicate with each other using local host.
- It's used to group one or more containers that should be deployed together on the same host. These containers share the same network and storage resources, making them tightly coupled.

Pod Creation, Templates, And Binding Times

- Kubernetes uses a late binding approach in which a programmer creates a *template* for the pod (sometimes called a *pod manifest*) that specifies items to use when running the pod.

- A template assigns a name to the pod, specifies which container or containers to run, lists the network ports the pod will use, and specifies the version of the Kubernetes API to use with the pod.

- A template can use *yaml* or *json* formats

An example template for a pod that runs one container and uses port 8080.

```
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
  labels:
    purpose: web-server
spec:
  containers:
  - name: example-pod
    image: f34cd9527ae6
    ports:
    - containerPort: 8080
```

Init Containers

- An init container is a specialized type of container that runs and completes its tasks before the main application containers within the same Pod start.
- **Init containers are designed to perform initialization and setup tasks that are necessary for the proper functioning of the application containers.**
- Init containers are useful for ensuring that your application's prerequisites are met before it begins its execution.
- They help manage complex initialization processes and dependencies, making it easier to run applications with specific requirements in Kubernete.
- **Eg : An init container can check the external storage and either exit normally if the storage is available, or exit with an error status if the storage is unavailable.**

Kubernetes Terminology: Nodes And Control Plane

- Kubernetes uses the term *Control Plane* (also known by the term *master*) to refer to the software components that an owner uses to create and operate containers.

- When the control plane software runs on a node, the node is known as a ***master node***.

- Kubernetes uses to run containers **a *Kubernetes node***. Some sources use the term ***worker node***.

master node and worker node

Kubernetes cluster

- Kubernetes cluster runs a single copy of the control plane components on a single master node.

- It is possible to create a high-availability cluster by running multiple copies of the control plane software on multiple master nodes.

Control Plane Software Components

Component	Purpose
API server	Provides an interface to the control plane
Scheduler	Chooses a node that will run a pod
Cluster State Store	Stores state information for the cluster
Controller Manager	Handles controllers that operate the cluster
Cloud Controller Manager	Handles cloud provider interactions

Control Plane Software Components

- *API server* : labeled the *kube-api-server*, the APIserver provides an interface to all the control plane components.
- *Scheduler* :*kube-scheduler*, the Scheduler handles the assignment of pods to nodes in the cluster.
- *Cluster State Store* : The Cluster State Store holds information about the cluster, including the set of nodes available to the cluster, the pods that are running, and the nodes on which the pods are currently running.
- When something changes in the cluster, the Cluster Store must be updated.
- **etcd**: A distributed key-value store that stores the cluster's configuration data, state information, and metadata. Etcd helps maintain consistency and reliability in the cluster.

Control Plane Software Components

- Controller Manager : **kube-controller-manage**r, the Controller Manager component consists of a daemon that remains running while the cluster exists.

- Kubernetes includes useful controllers, such as a *Replication Controller*, *Endpoints Controller*, and *Namespace Controller*

- Controller Manager performs housekeeping functions such as garbage collection of events, terminated pods, and unused nodes.

Control Plane Software Components

- The **Cloud Controller Manager (CCM)** is a component in a Kubernetes cluster responsible for interacting with a cloud provider's APIs to manage and control cloud-specific resources and functionalities.

- CCM is an optional component, and its use depends on whether your Kubernetes cluster is running in a cloud environment and needs to integrate with that cloud provider's services.

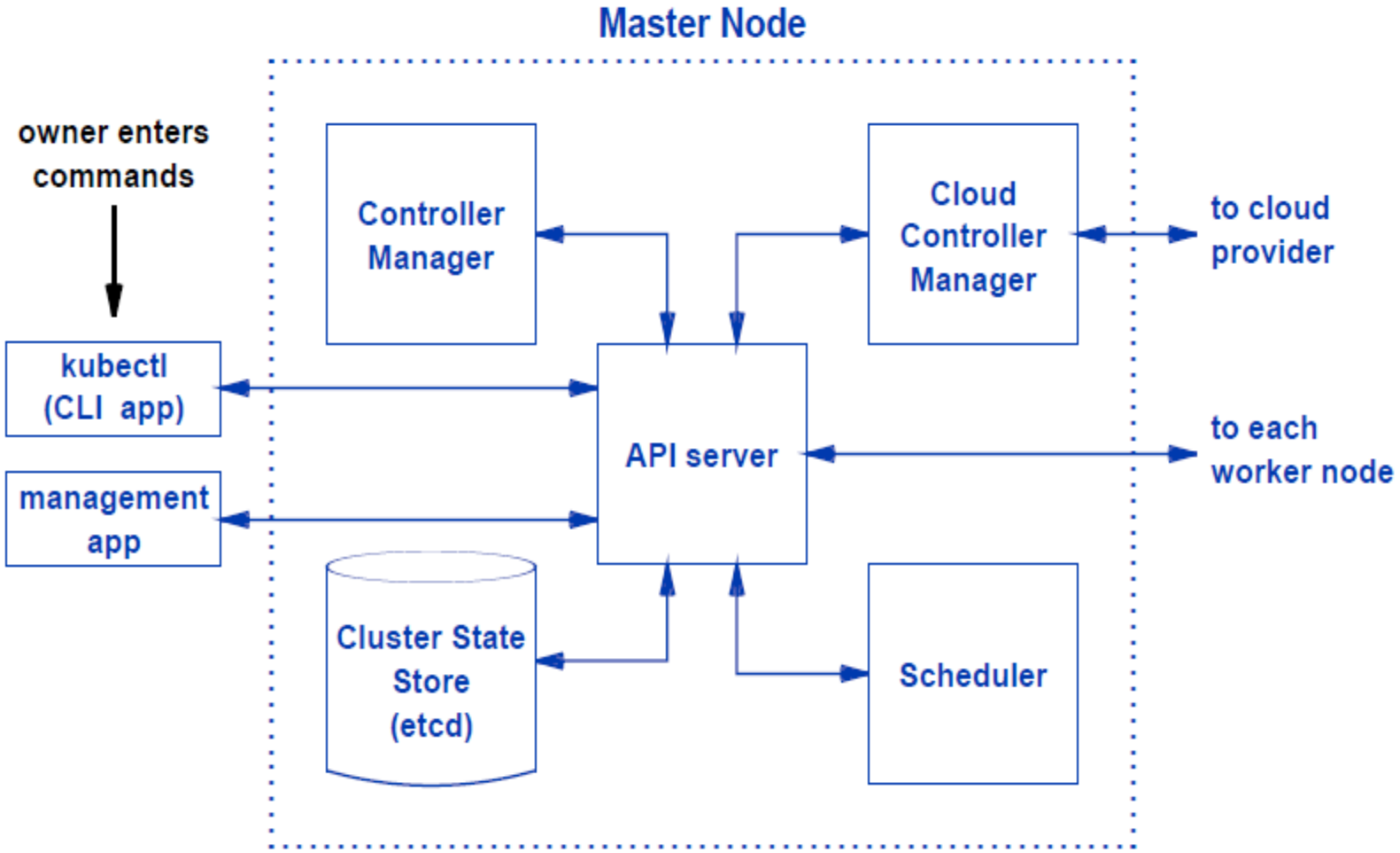
Command-Line Interface (CLI) (*command-line app*)

- Perform various tasks related to Kubernetes cluster management, resource configuration, monitoring, and troubleshooting.

- These command-line tools are used by administrators, developers, and operators to interact with and manage Kubernetes clusters.

- **kubectl** is the primary command-line tool for interacting with Kubernetes clusters. It allows users to create, update, delete, and manage Kubernetes resources, inspect cluster status, and perform various administrative tasks.

Communication Among Control Plane Components



Worker Node Software Components (also known as Minions or Nodes)

- Each worker node runs software components that control and manage the pods running on the node.

Component	Purpose
Service Proxy	Configures the network on the node (i.e., iptables)
Kubelet	Uses container software to run and monitor pods
Container Runtime	Standard container software (e.g., Docker Engine)

Service Proxy

- Also known as a **"Kube Proxy" or "Kubernetes Proxy,"** is a network service responsible for facilitating network communication to and from Pods within the cluster.
- Its primary role is to enable internal network connectivity among Pods, expose services to the network, and manage routing and load balancing for those services.
- iptables is a powerful and flexible Linux kernel-level firewall tool that can be configured to filter, modify, and route network packets.
- iptables helps in managing network traffic and implementing various network-related features and policies within the cluster.

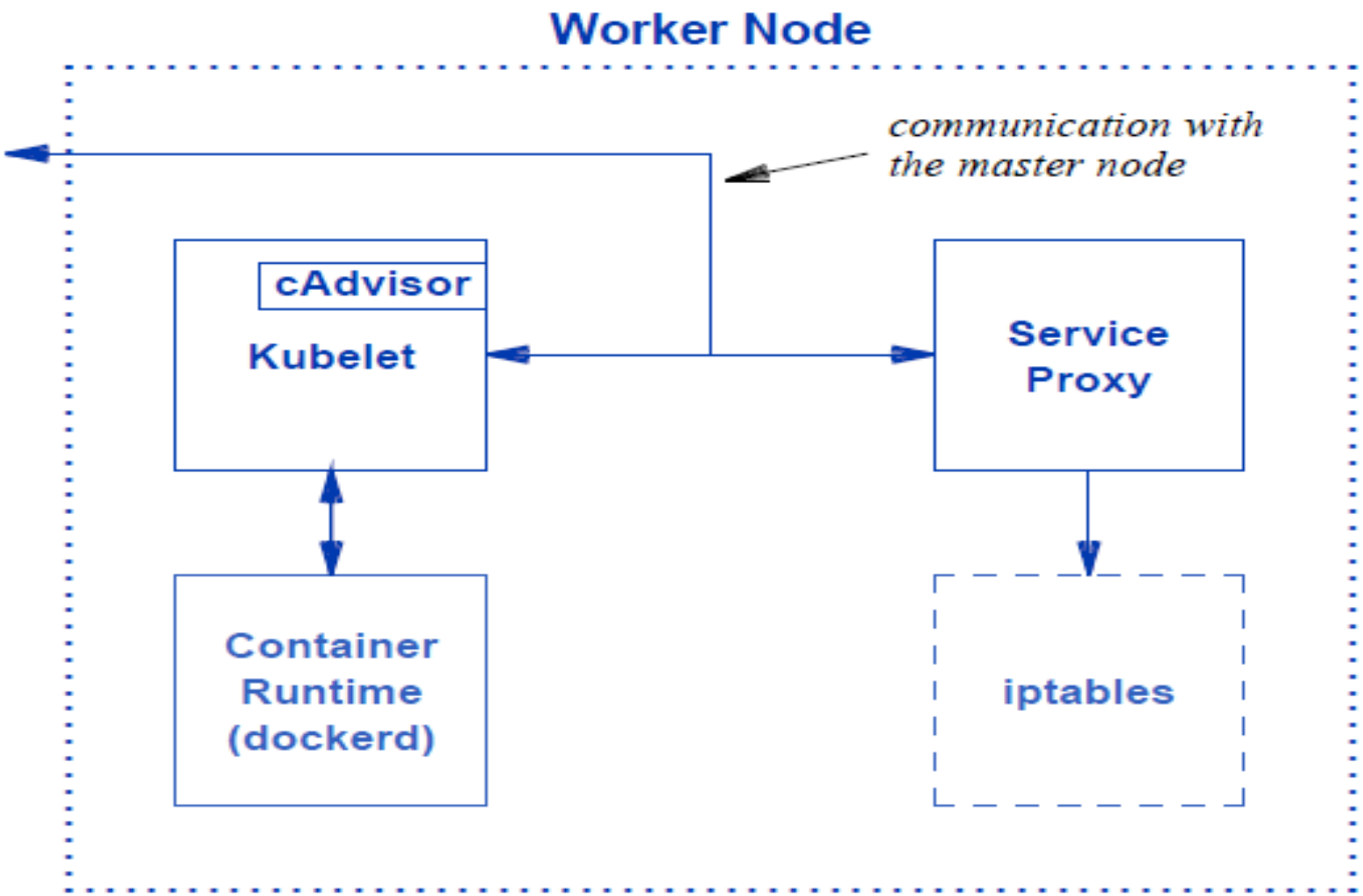
Kubelet

- The Kubelet is a critical component in Kubernetes that ensures that containers are running as expected and that the desired state, as defined in the Pod specifications, is maintained.
- It communicates with the control plane components, such as the API server, to synchronize its actions with the cluster's desired state.
- The Kubelet is an essential part of the orchestration and management of containers in a Kubernetes cluster.
- Kubelet includes a copy of the *cAdvisor* software that collects and summarizes statistics about pods.

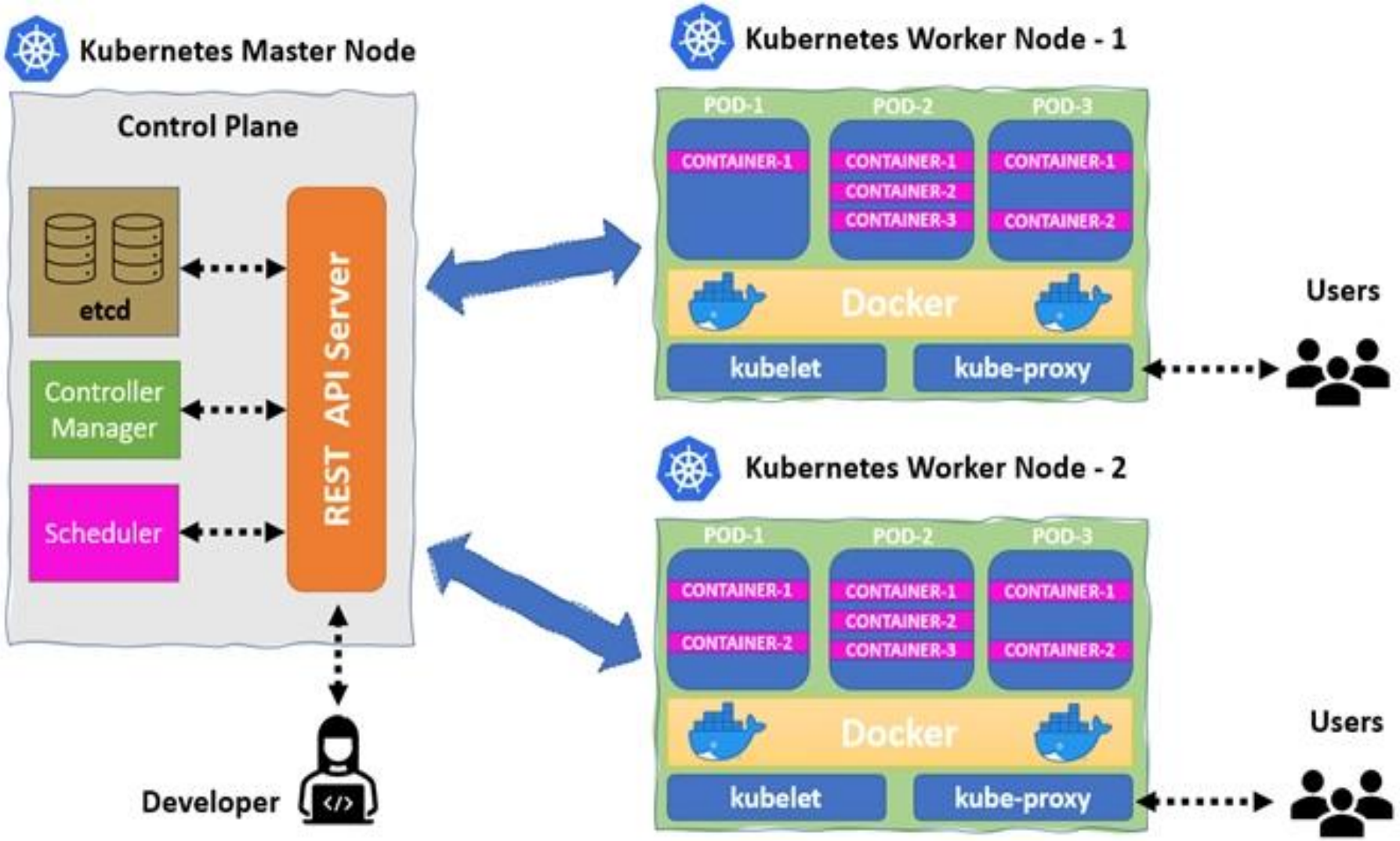
Container Runtime

- **When Kubernetes needs to deploy containers, Kubelet interacts with the Container Runtime system to perform the required task.**

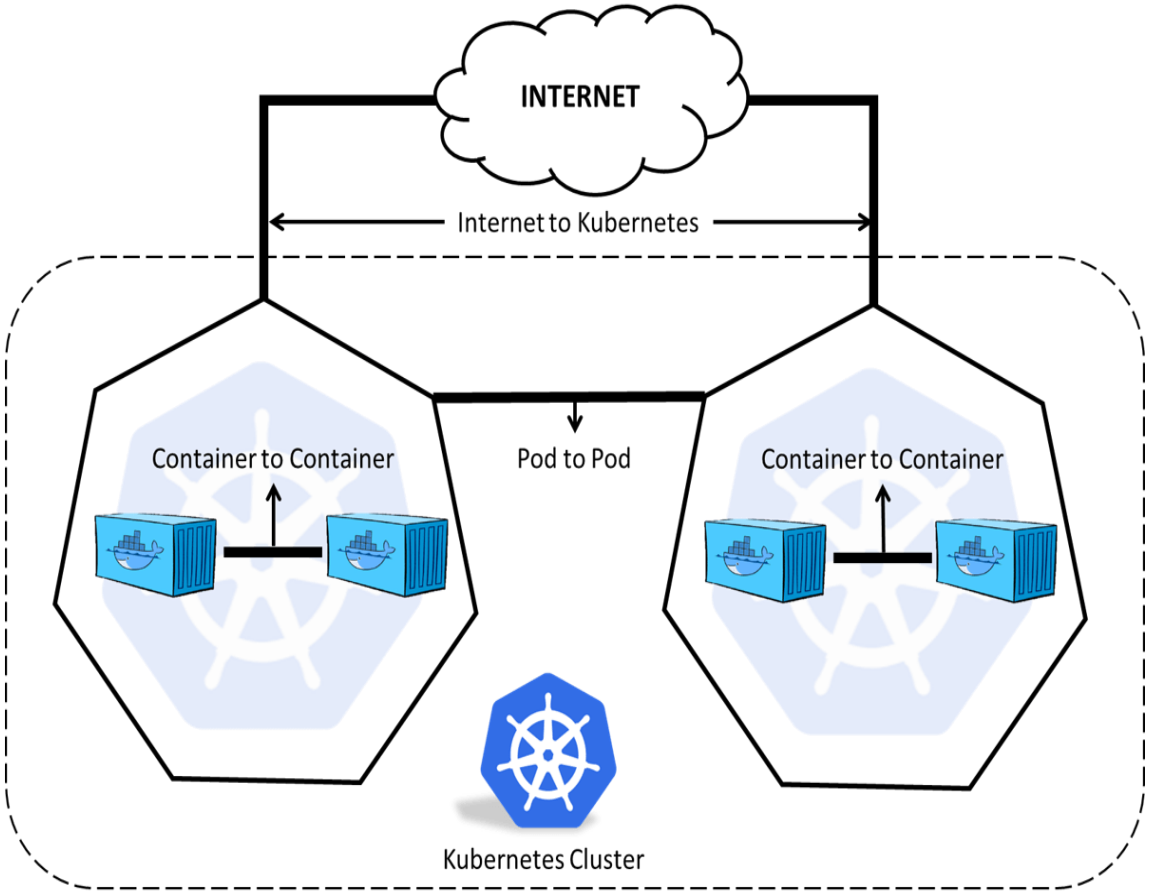
Software components on a worker node and the communication among them.



Overview of Kubernetes Architecture



Kubernetes Networking Model



Summary

- MapReduce
- Hadoop
- Microservices
- Serverless Computing
- Devops
- Containers
- Docker
- Kubernetes
- Cluster Model
- Master Node And Worker Node
- Kubernetes Architecture