Use Case Scenarios

Who?

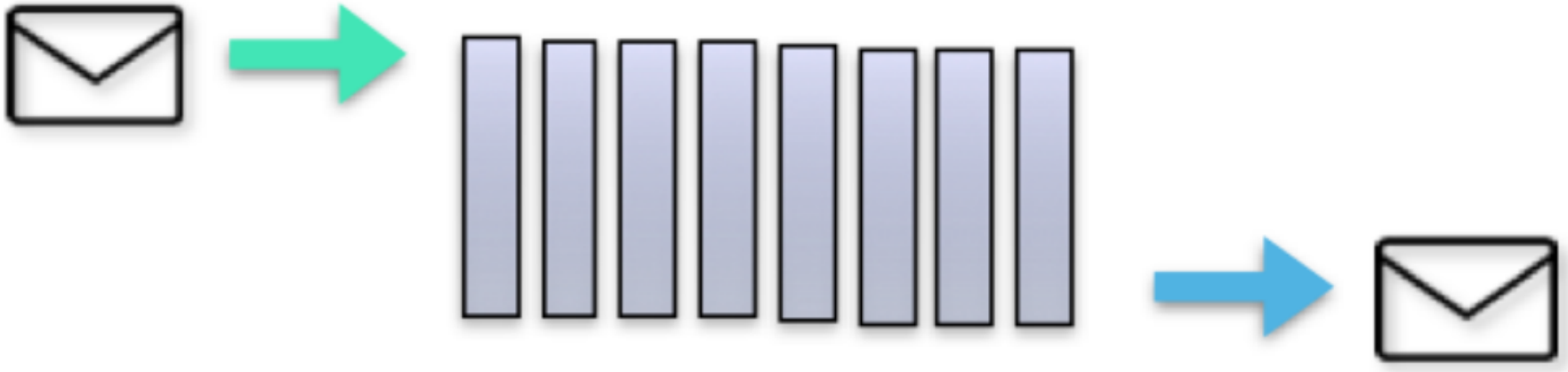
↓

Action?

↓

What?

Publish-subscribe Messaging System



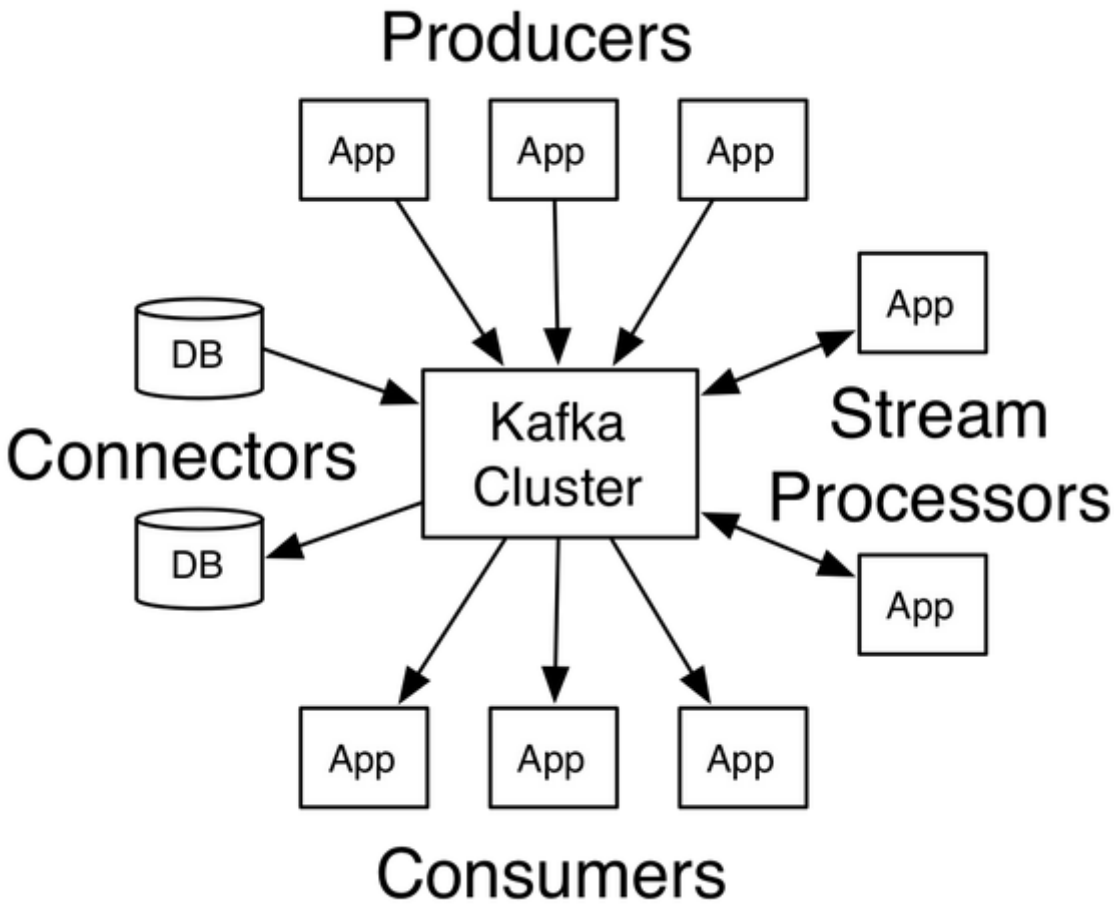
Why Messaging System?

- Loose couplings between modules
- Queuing data for later delivery
- Asynchronous processing
- Reliable load balancing

LinkedIn

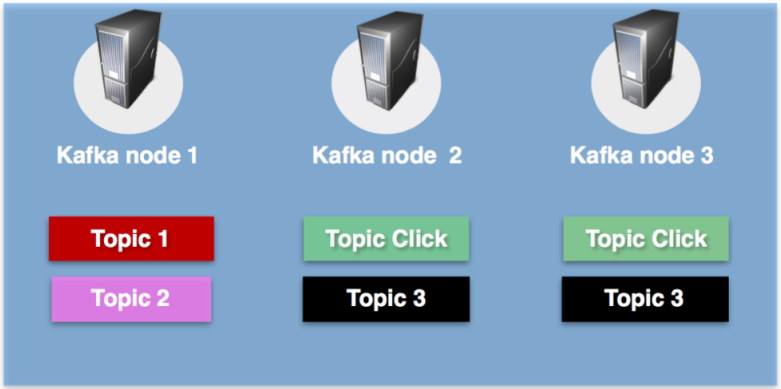
APACHE kafka®

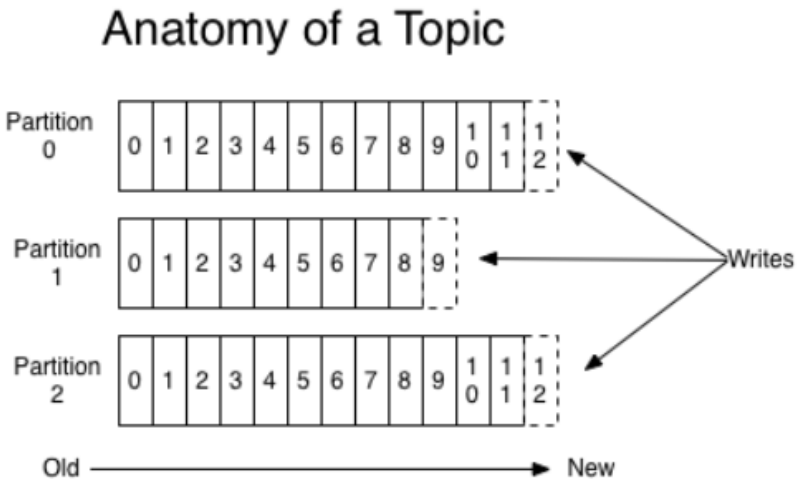
A distributed streaming platform


Producers
App
App
App
DB
Connectors
DB
Kafka
Cluster
App
Stream
Processors
App
App
App
App
Consumers

Topic

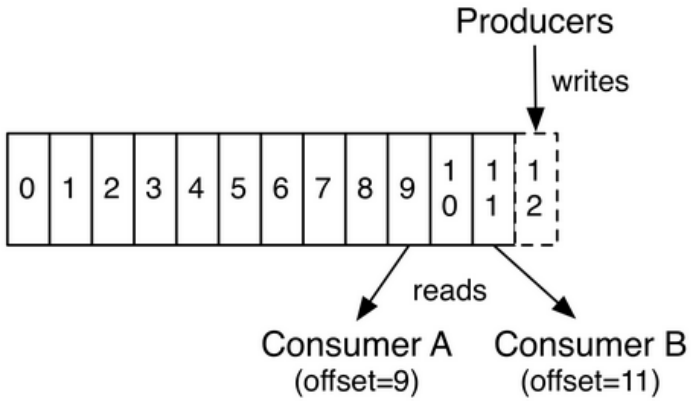
- **A Topic** is a category/feed name to which messages are stored and published.



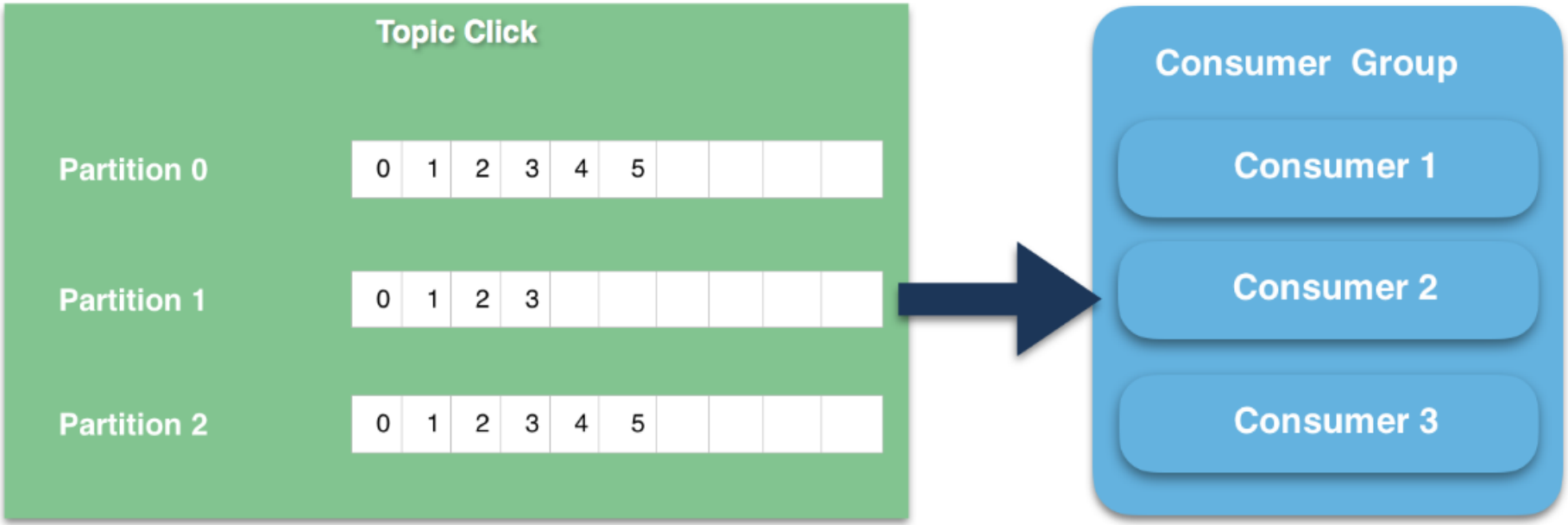


Topic Partition

- Topics are divided into a number of partitions, which contains messages in an unchangeable sequence.
- Each message in a partition is assigned and identified by its unique **offset.**
- A topic can also have multiple partition logs. This allows for multiple consumers to read from a topic in parallel.



Consumers and consumer groups



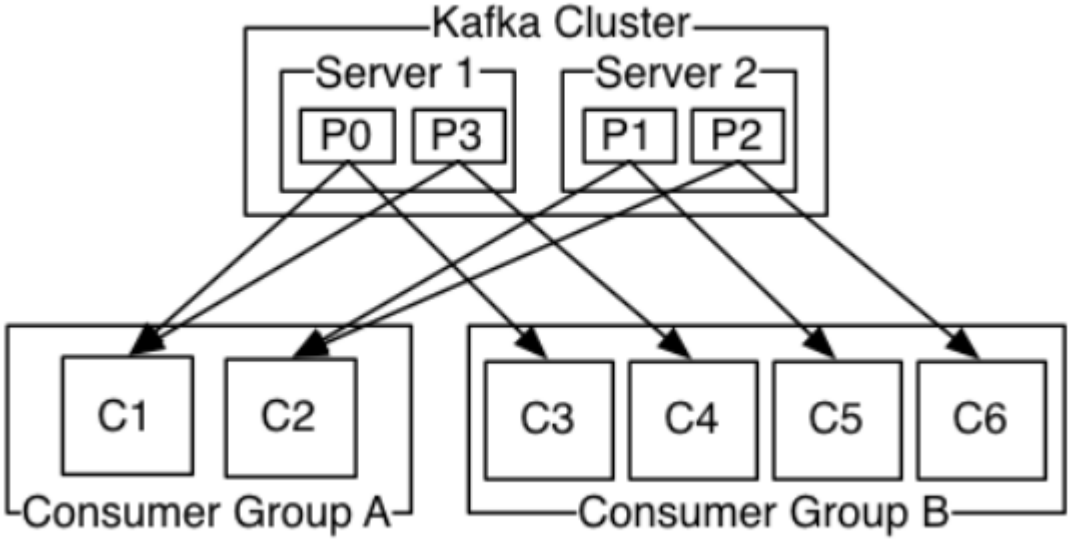
Consumers

- Consumers can read messages starting from a specific offset and are allowed to read from any offset point they choose. This allows consumers to join the cluster at any point in time.

- Consumers pull messages from topic partitions. Different consumers can be responsible for different partitions.

Consumer Group

- Consumers can join a group called a consumer group.
- A consumer group includes the set of consumer processes that are subscribing to a specific topic.
- Each consumer in the group is assigned a set of partitions to consume from. They will receive messages from a different subset of the partitions in the topic.
- Kafka guarantees that a message is only read by a single consumer in the group.

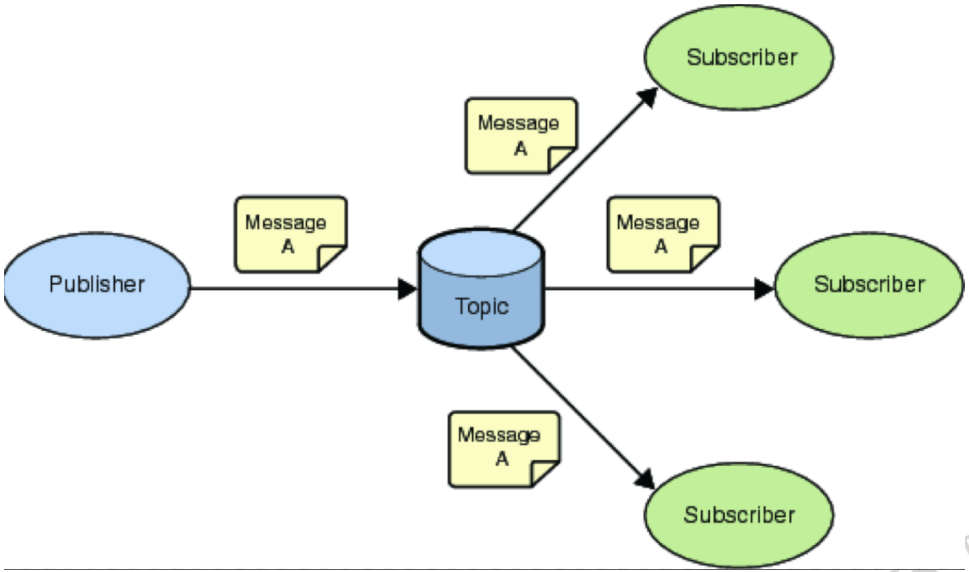

Kafka Cluster
Server 1
Server 2
P0
P3
P1
P2
C1
C2
C3
C4
C5
C6
Consumer Group A
Consumer Group B

Two Models for Messaging: Queuing

- In a queue, a pool of consumers may read from a server and each record goes to one of them
- It allows you to divide up the processing of data over multiple consumer instances, which lets you scale your processing.
- Unfortunately, queues aren't multi-subscriber—once one process reads the data it's gone

Two Models for Messaging: Pub-Sub

- The record is broadcast to all consumers
- Allows you broadcast data to multiple processes, but has no way of scaling processing since every message goes to every subscriber.



Kafka Messaging Models

- The consumer group concept in Kafka generalizes these two concepts.
  - As with a **queue** the consumer group allows you to divide up processing over a collection of processes (the members of the consumer group).
  - As with **publish-subscribe**, Kafka allows you to broadcast messages to multiple consumer groups.

Message Consumption

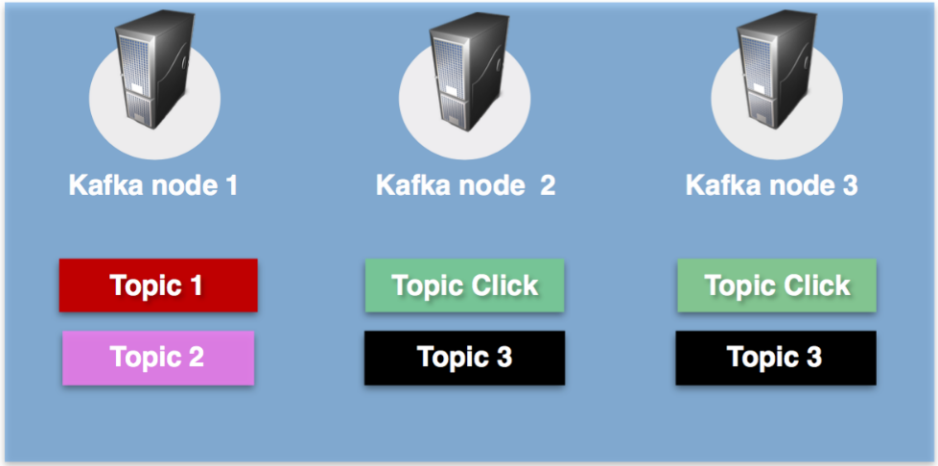
- Messages published to the cluster will stay in the cluster until a configurable retention period has passed by.

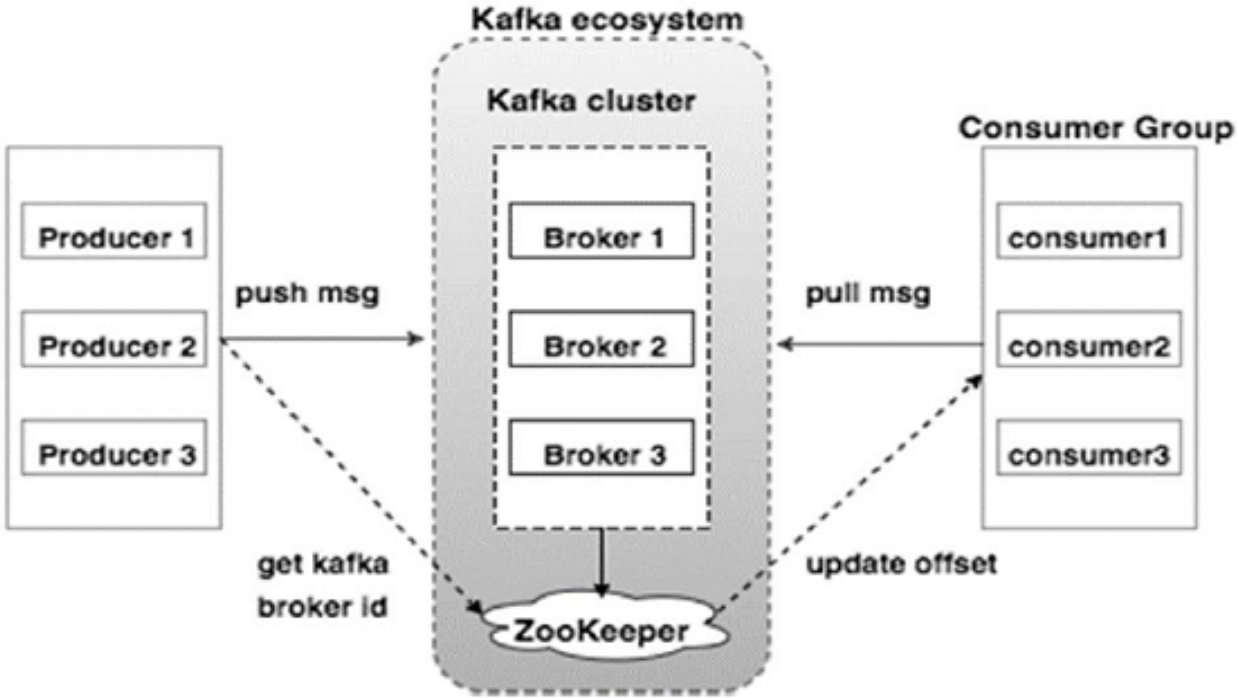
- Kafka retains all messages for a set amount of time, and therefore, consumers are responsible to track their location.

- Data/messages are never pushed out to consumers, the consumer will ask for messages when the consumer is ready to handle the message.

Replication

- Implemented at the partition level. The redundant unit of a topic partition is called a replica.
- Each partition usually has one or more replicas meaning that partitions contain messages that are replicated over a few Kafka brokers in the cluster.
- The *click-topic* is replicated to Kafka node 2 and Kafka node 3.




Kafka ecosystem
Kafka cluster
Consumer Group
Producer 1
Producer 2
Producer 3
push msg
Broker 1
Broker 2
Broker 3
pull msg
consumer1
consumer2
consumer3
get kafka
broker id
ZooKeeper
update offset

Broker

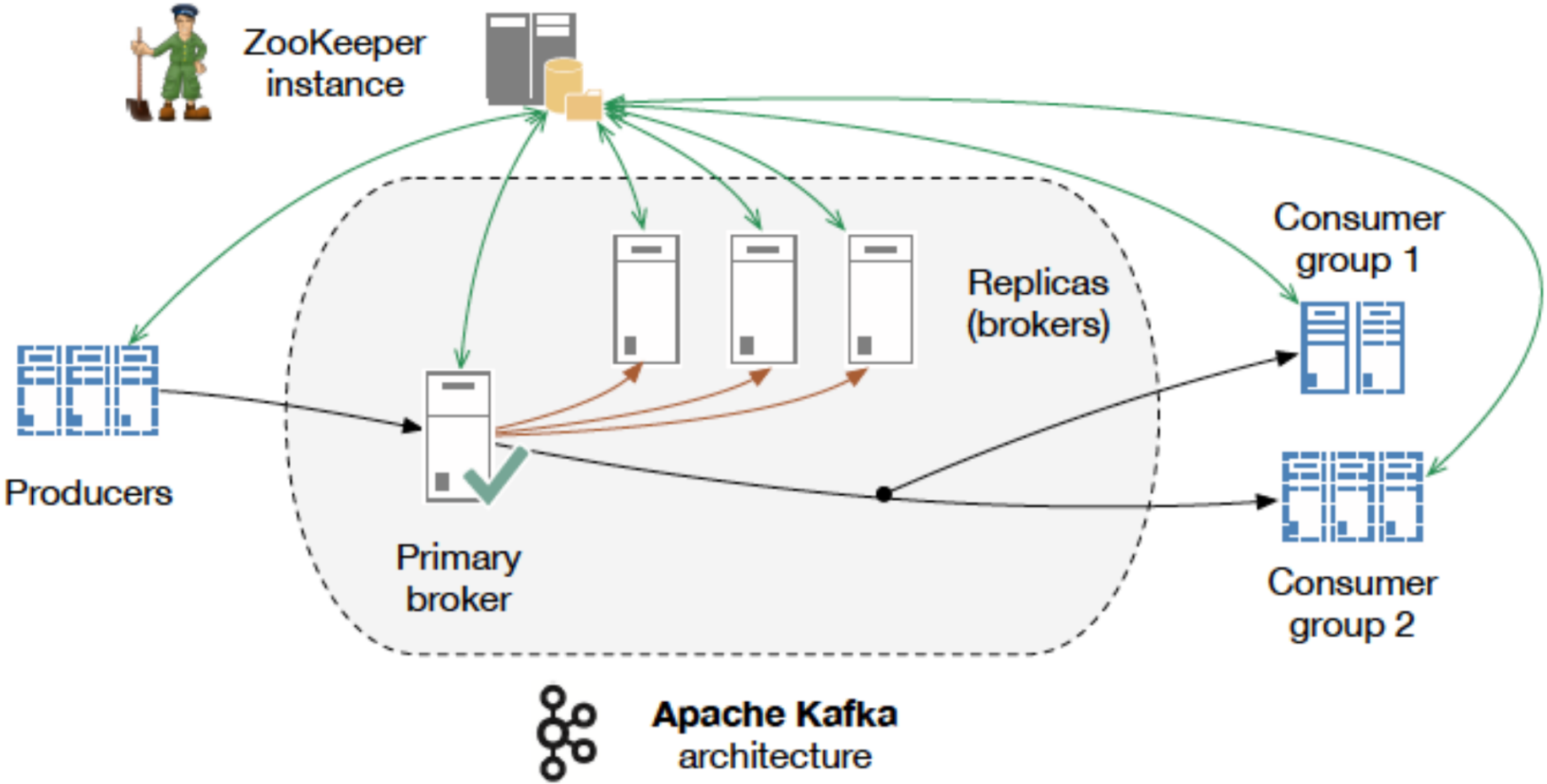
- Kafka cluster typically consists of multiple brokers to maintain load balance.
- Kafka brokers are stateless, so they use ZooKeeper for maintaining their cluster state.
- One Kafka broker instance can handle hundreds of thousands of reads and writes per second and each broker can handle TB of messages without performance impact.
- Kafka broker leader election can be done by ZooKeeper.

ZooKeeper

- Apache Zookeeper, which is a distributed configuration and synchronization service.
- Zookeeper serves as the coordination interface between the Kafka brokers and consumers.
- The Kafka servers share information via a Zookeeper cluster. Kafka stores basic metadata in Zookeeper such as information about topics, brokers, consumer offsets (queue readers) and so on.

ZooKeeper (cont.)

- ZooKeeper is used for managing and coordinating Kafka broker.
- ZooKeeper service is mainly used to notify producer and consumer about the presence of any new broker in the Kafka system or failure of the broker in the Kafka system.
- As per the notification received by the Zookeeper regarding presence or failure of the broker then producer and consumer takes decision and starts coordinating their task with some other broker.


ZooKeeper instance
Replicas (brokers)
Consumer group 1
Producers
Primary broker
Consumer group 2
Apache Kafka architecture


Passive Scheduling: Kafka
Proactive Scheduling: K8S

Kubernetes

- Kubernetes is a platform for hosting Docker containers in a clustered environment with multiple Docker hosts
- Provides container grouping, load balancing, auto-healing, scaling features
- Project was started by Google
- Contributors == Google, CodeOS, Redhat, Mesosphere, Microsoft, HP, IBM, VMWare, Pivotal, SaltStack, etc

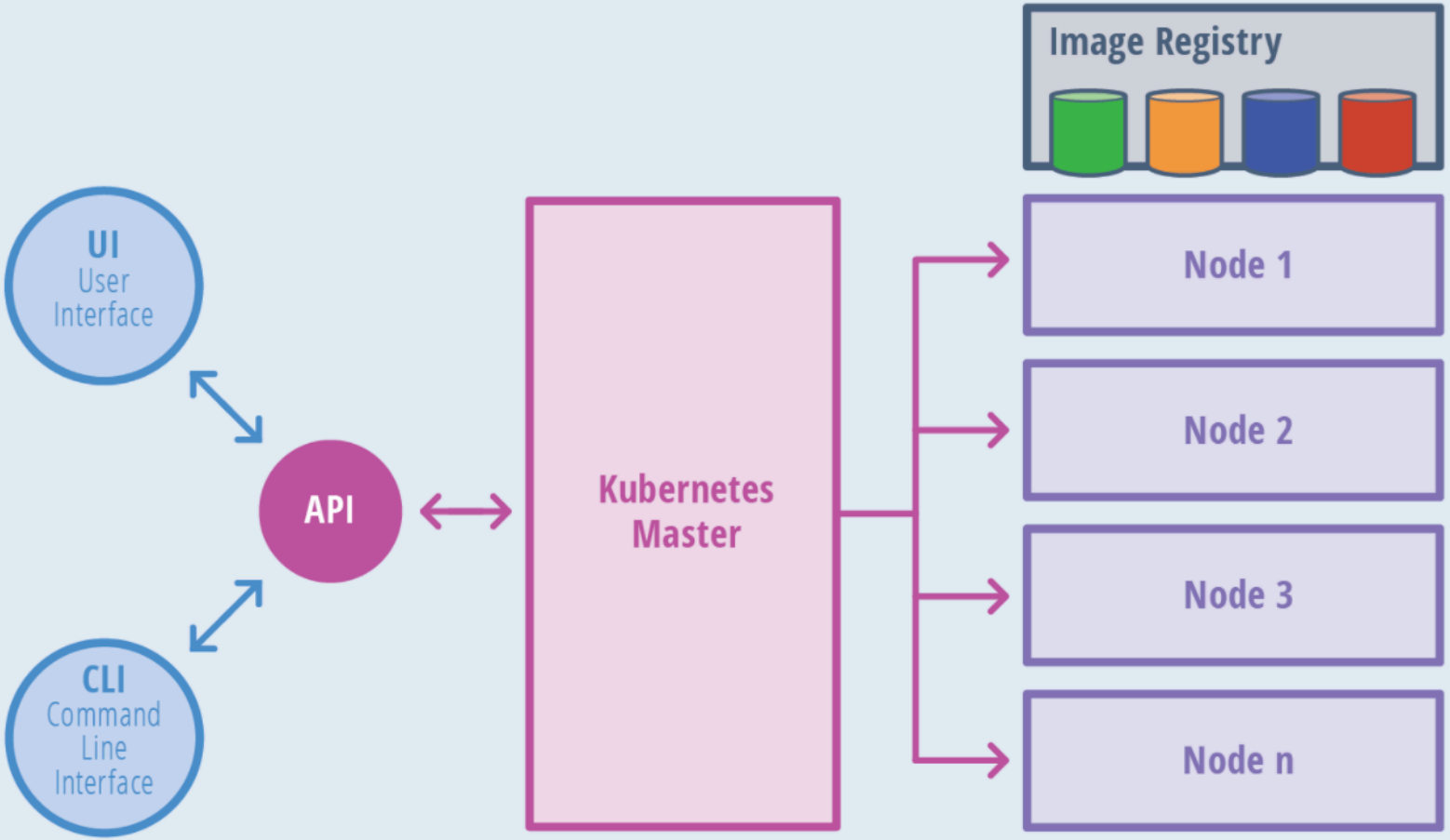
Managing Containers

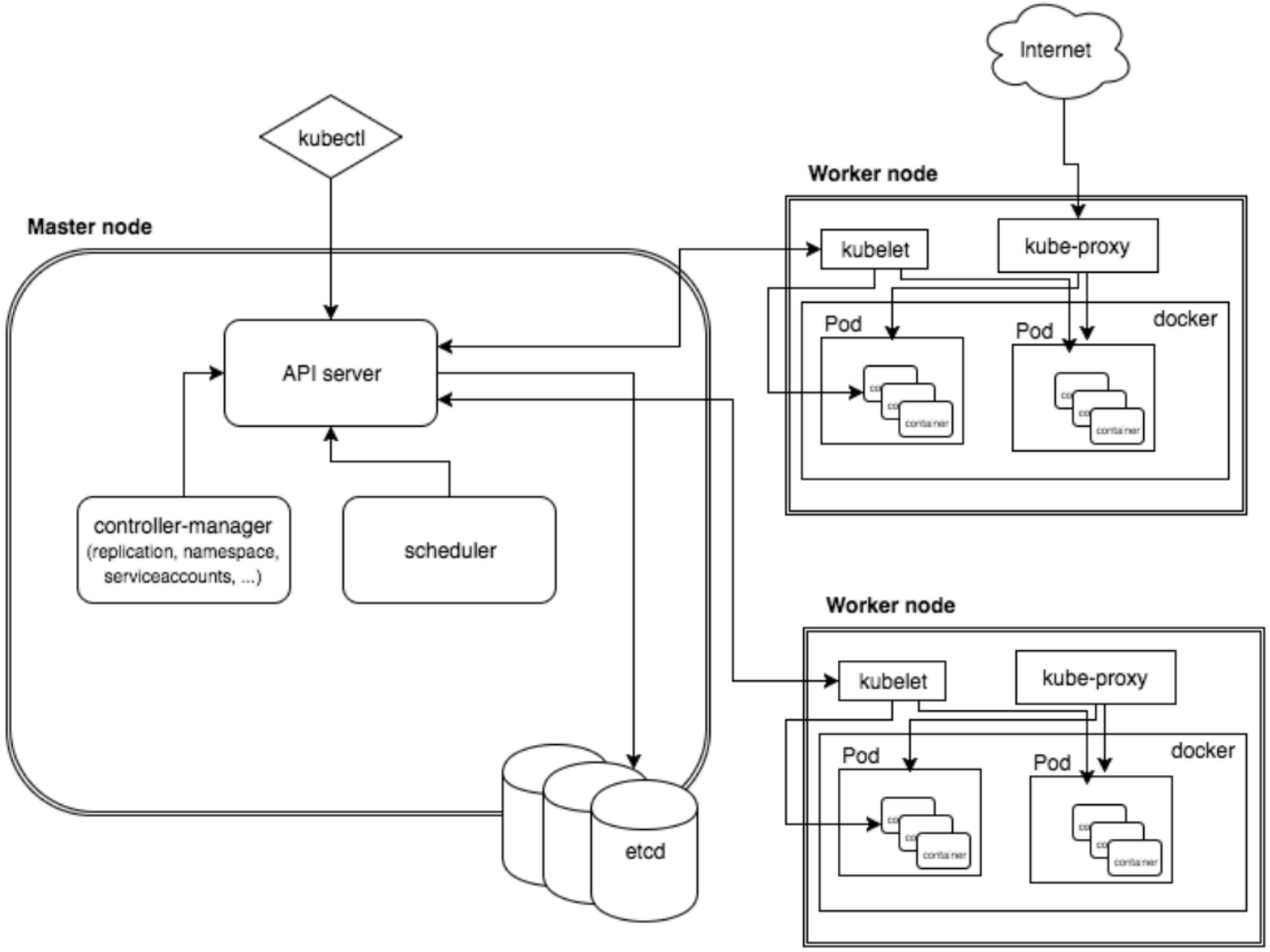
- Greek for pilot or helmsman of a ship
- Open Source cluster manager from Google
- Managing containerized applications across a cluster of nodes
- Kubernetes is:
  - lean
  - portable
  - extensible
  - self-healing
- Has Master and Node (Minion) components
- Easy Service deployments, updates and scalability
- Can run basically on every Linux platform

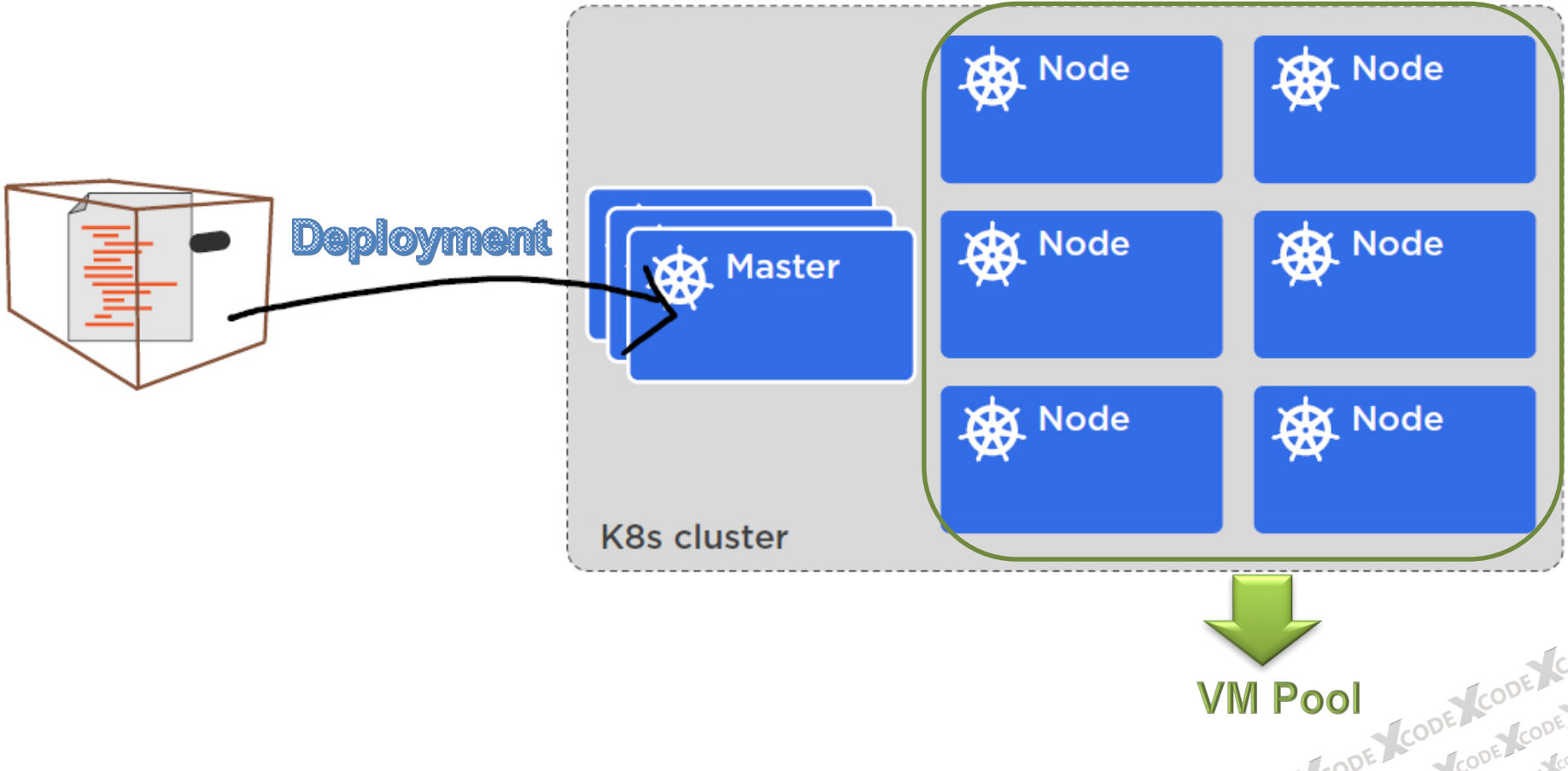
Key Concepts of Kubernetes

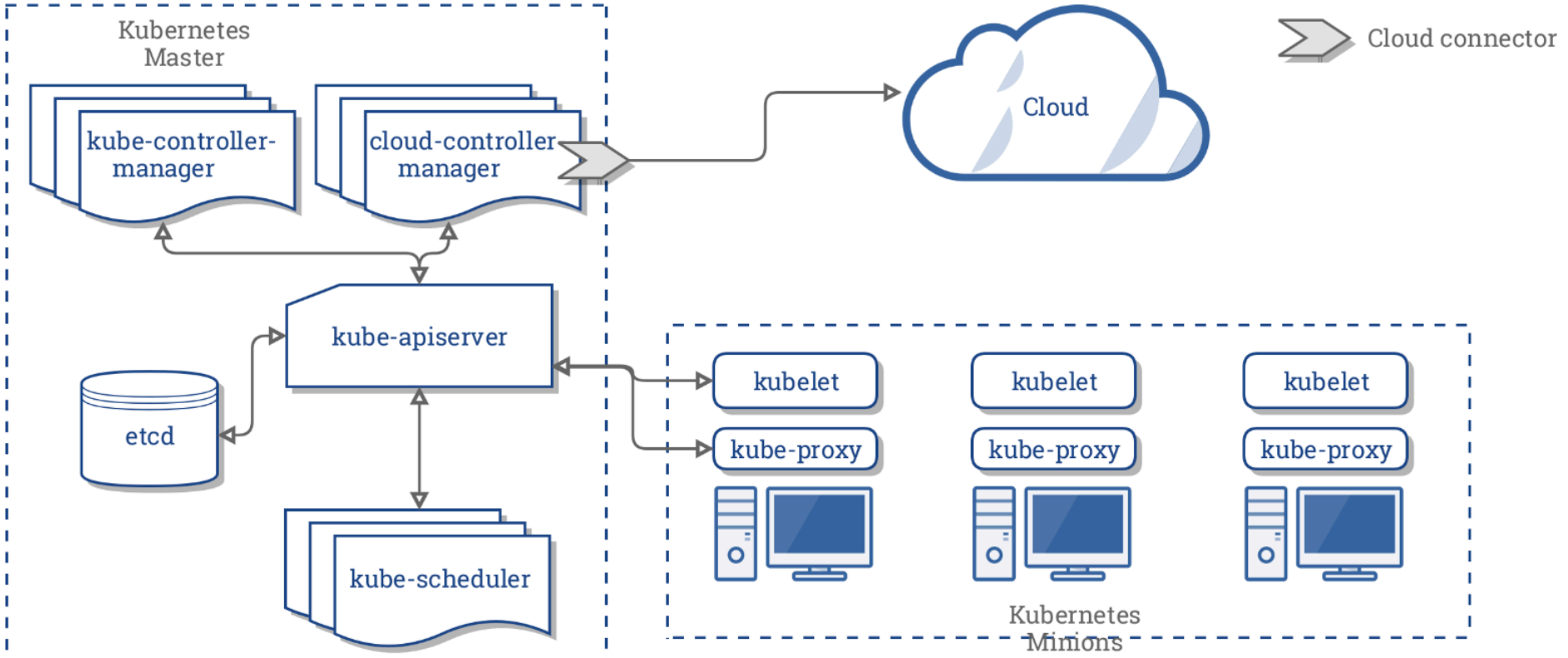
- **Pod** - A group of Containers
- **Labels** - Labels for identifying pods
- **Kubelet** - Container Agent
- **Proxy** - A load balancer for Pods
- **etcd** - A metadata service
- **cAdvisor** - Container Advisor provides resource usage/performance statistics
- **Replication Controller** - Manages replication of pods
- **Scheduler** - Schedules pods in worker nodes
- **API Server** - Kubernetes API server

Kubernetes Architecture




Internet
kubectl
Worker node
Master node
kubelet
kube-proxy
Pod
Pod
docker
API server
container
container
controller-manager
(replication, namespace,
serviceaccounts, ...)
scheduler
Worker node
kubelet
kube-proxy
Pod
Pod
docker
container
container
etcd

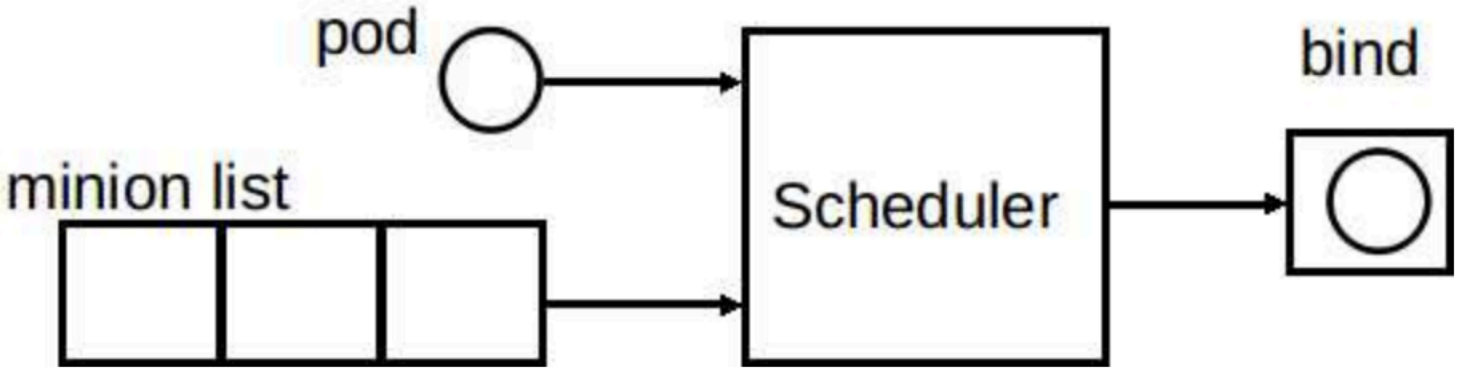

Deployment
Master
Node
Node
Node
Node
Node
Node
K8s cluster
VM Pool



Resource Allocation

Kubernetes Scheduler的基本功能

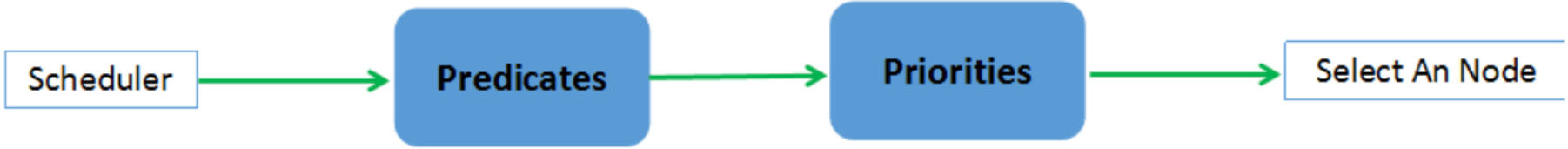
根据特定的调度算法将pod调度到指定的工作节点（Node）上，这一过程也叫绑定（bind）
Scheduler 的输入为需要调度的 Pod 和可以被调度的节点(Node)的信息，输出为调度算法选择的 Node，并将该 pod bind 到这个 Node



Kubernetes Scheduler中调度算法分为两个阶段：

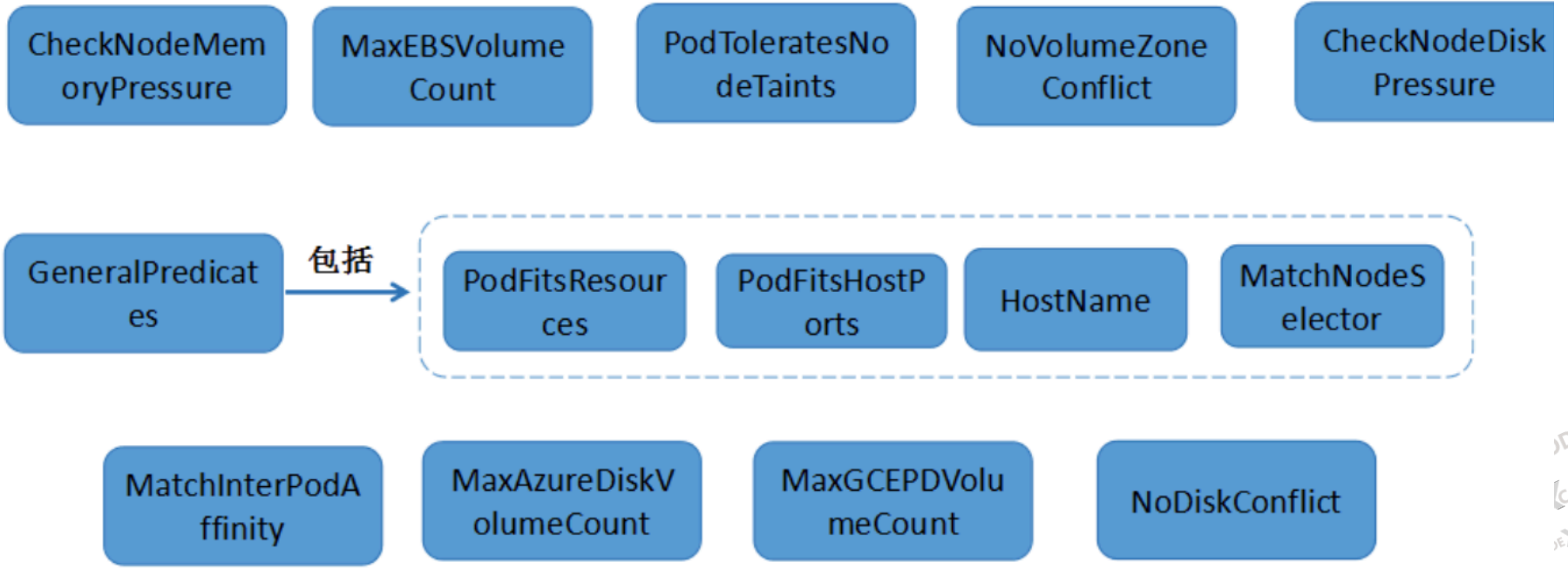
预选：根据配置的 Predicates Policies（默认为 DefaultProvider 中定义的 default predicates policies 集合）过滤掉那些不满足Policies的的Nodes，剩下的Nodes作为优选的输入。

优选：根据配置的 Priorities Policies（默认为 DefaultProvider 中定义的 default priorities policies 集合）给预选后的Nodes进行打分排名，得分最高的Node即作为最适合的Node，该Pod就Bind到这个Node。



预选规则详细说明

预先规则主要用于过滤出不符合规则的Node节点，剩下的节点作为优选的输入。在1.6.1版本中预选规则包括：

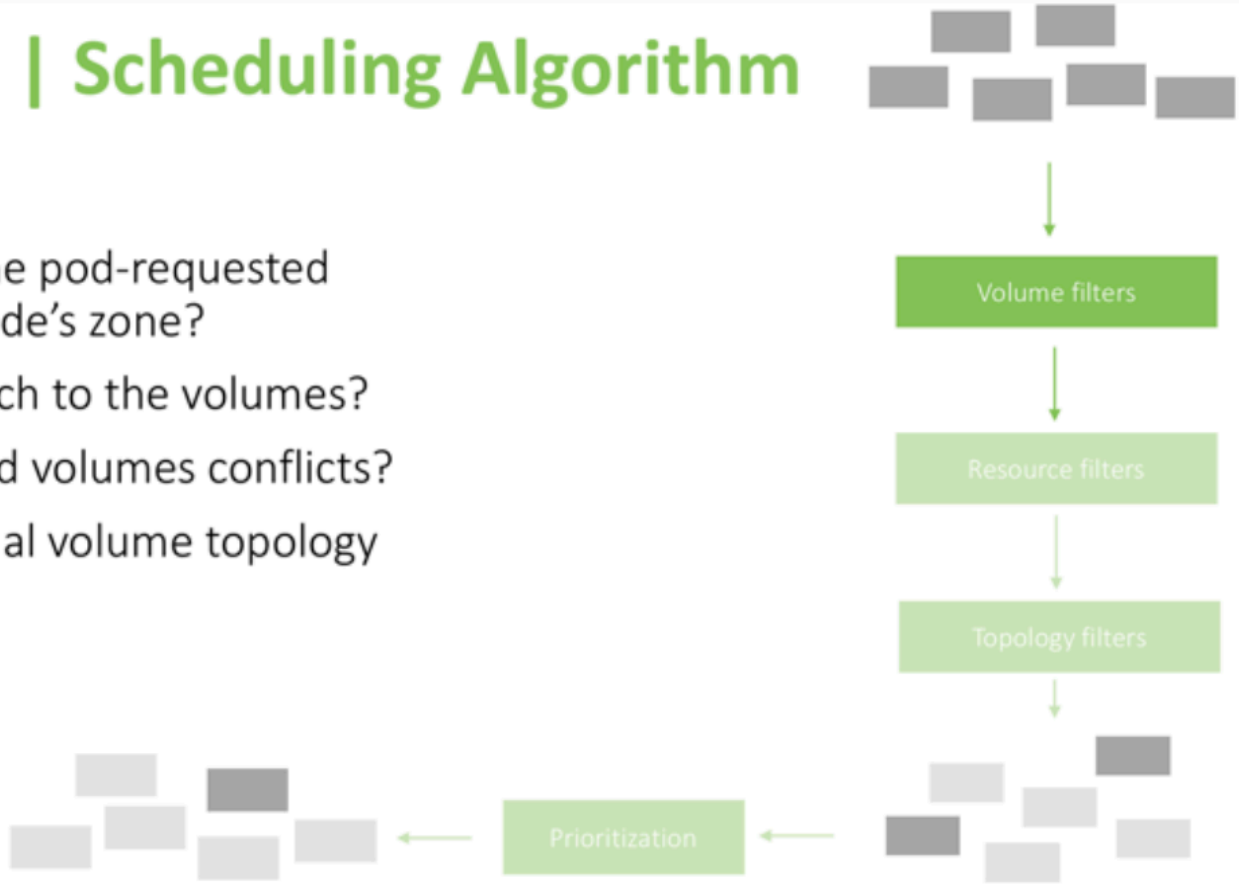


Volume Filters

Kubernetes | Scheduling Algorithm

Volume filters

- Do the zones of the pod-requested volumes fit the node's zone?
- Can the node attach to the volumes?
- Are there mounted volumes conflicts?
- Are there additional volume topology constraints?

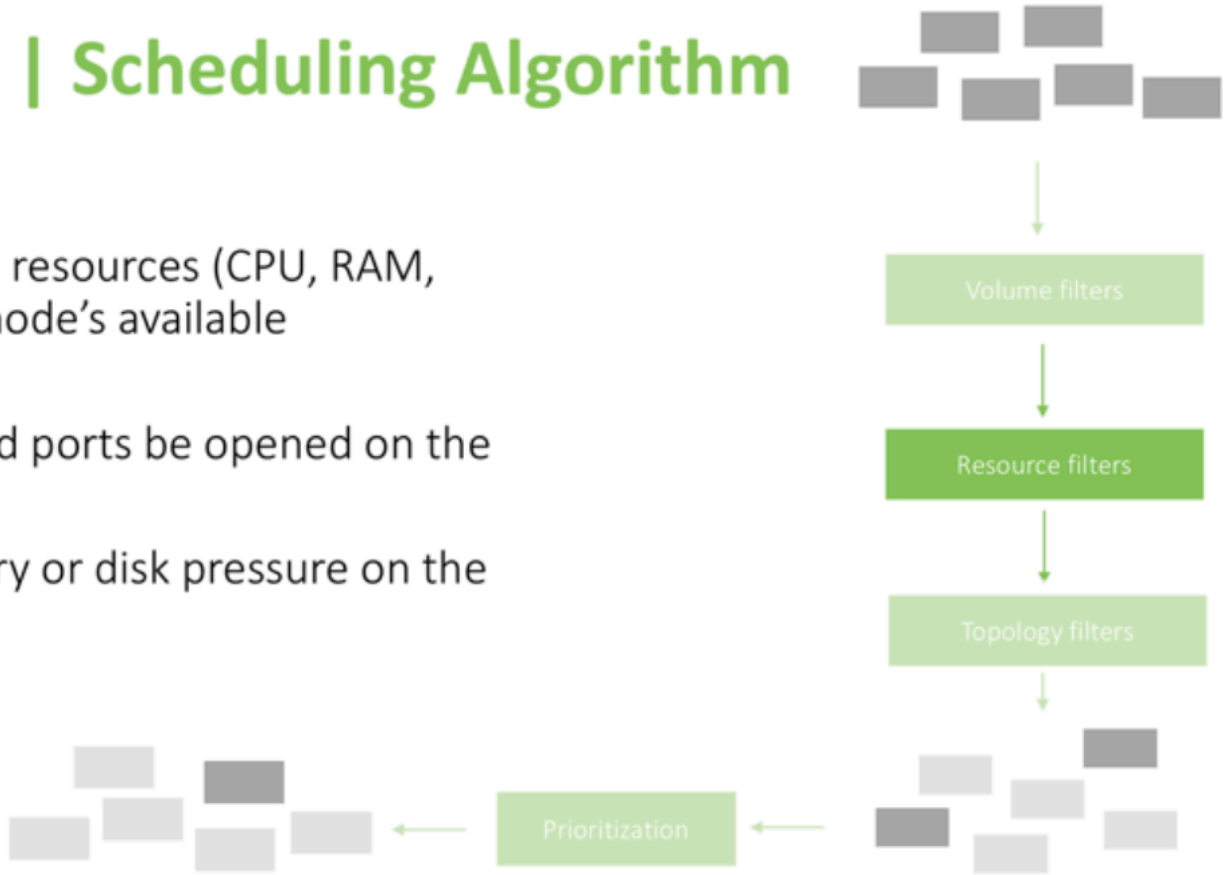


Next Are Resource Filters:

Kubernetes | Scheduling Algorithm

Resource filters

- Do pod-requested resources (CPU, RAM, GPU, etc.) fit the node's available resources?
- Can pod-requested ports be opened on the node?
- Is there no memory or disk pressure on the node?

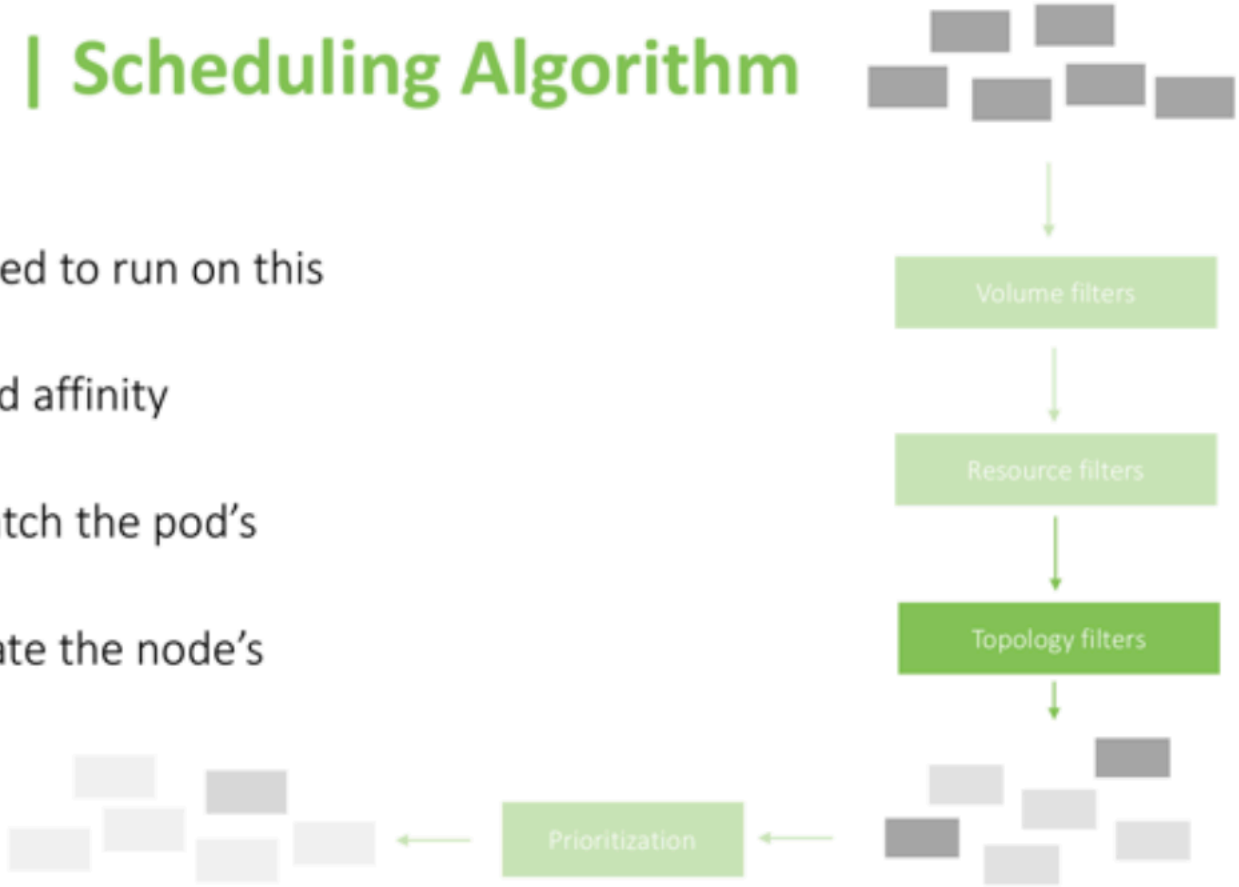


The Last Set of Filters Is Affinity Selectors:

Kubernetes | Scheduling Algorithm

Topology filters

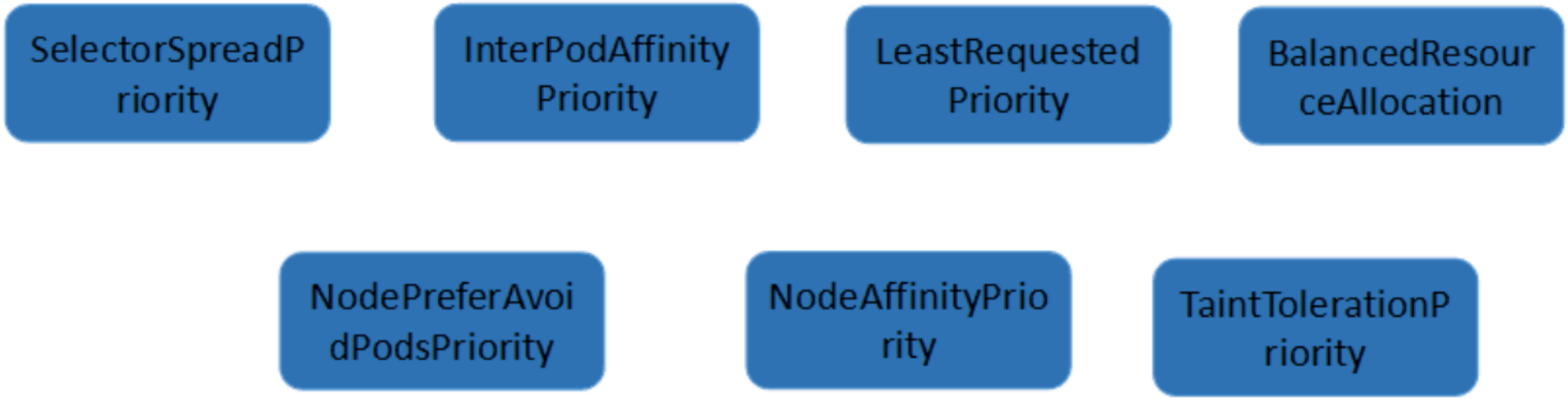
- Is the pod requested to run on this node?
- Are there inter-pod affinity constraints?
- Does the node match the pod's node selector?
- Can the pod tolerate the node's taints?



优选规则详细说明

优选规则对符合需求的主机列表进行打分，最终选择一个分值最高的主机部署 Pod。kubernetes 用一组优先级函数处理每一个待选的主机。每一个优先级函数会返回一个0-10的分数，分数越高表示主机越“好”，同时每一个函数也会对应一个表示权重的值。最终主机的得分用以下公式计算得出：

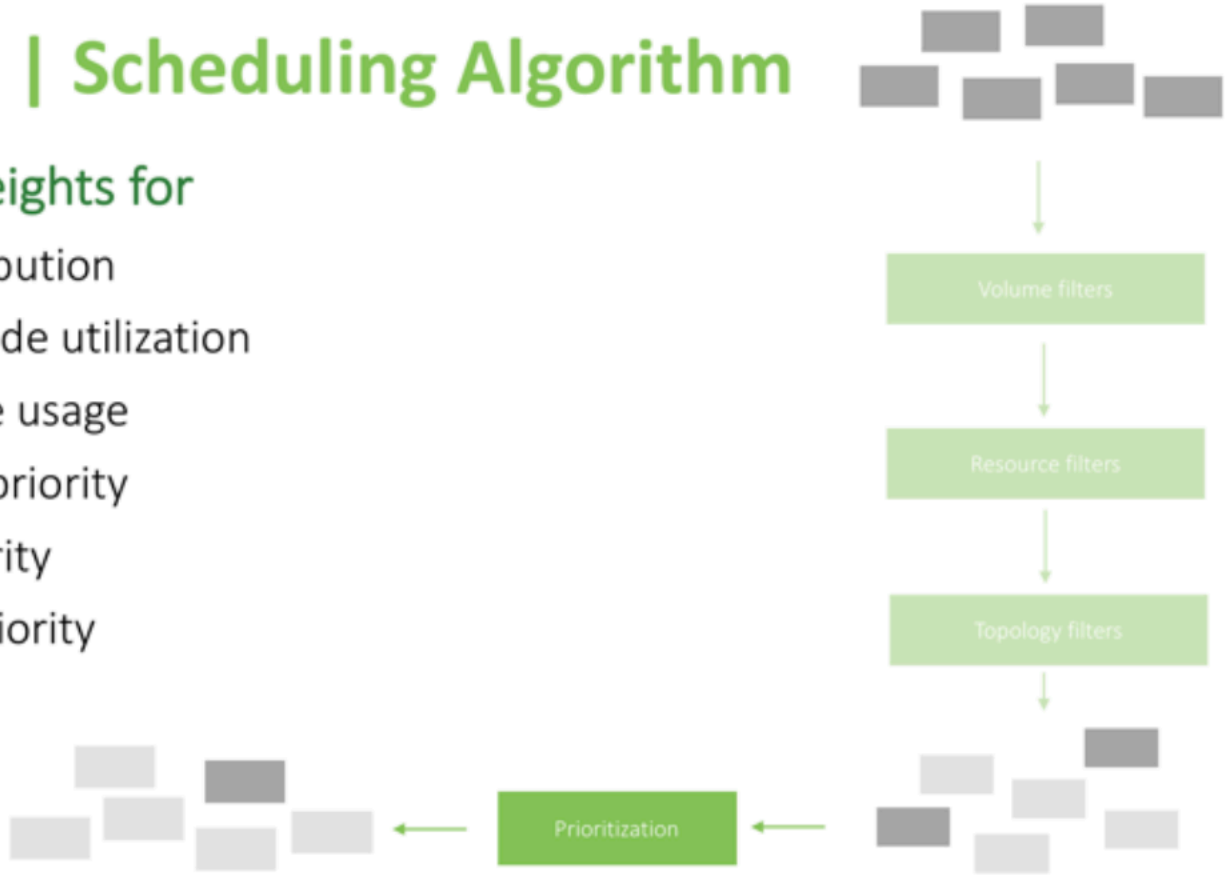
finalScoreNode = (weight1 *priorityFunc1)* + *(weight2* priorityFunc2) + … + (weightn * priorityFuncn)



Kubernetes | Scheduling Algorithm

Prioritize with weights for

- Pod replicas distribution
- Least (or most) node utilization
- Balanced resource usage
- Inter-pod affinity priority
- Node affinity priority
- Taint toleration priority



Job states and state transitions

- Admitted: Initially admitted.
- Pending: Waiting for dependencies to be satisfied.
- Ready: All dependencies are satisfied. The job is ready to run but waiting to be scheduled. The state exists for throttling pod creation purpose.
- Scheduled: Pods are created or re-created (in case of retry).
- Running: Some pods for the job is in the state of PodRunning.
- Failed: The job has failed.
- Succeeded: The job has succeeded.

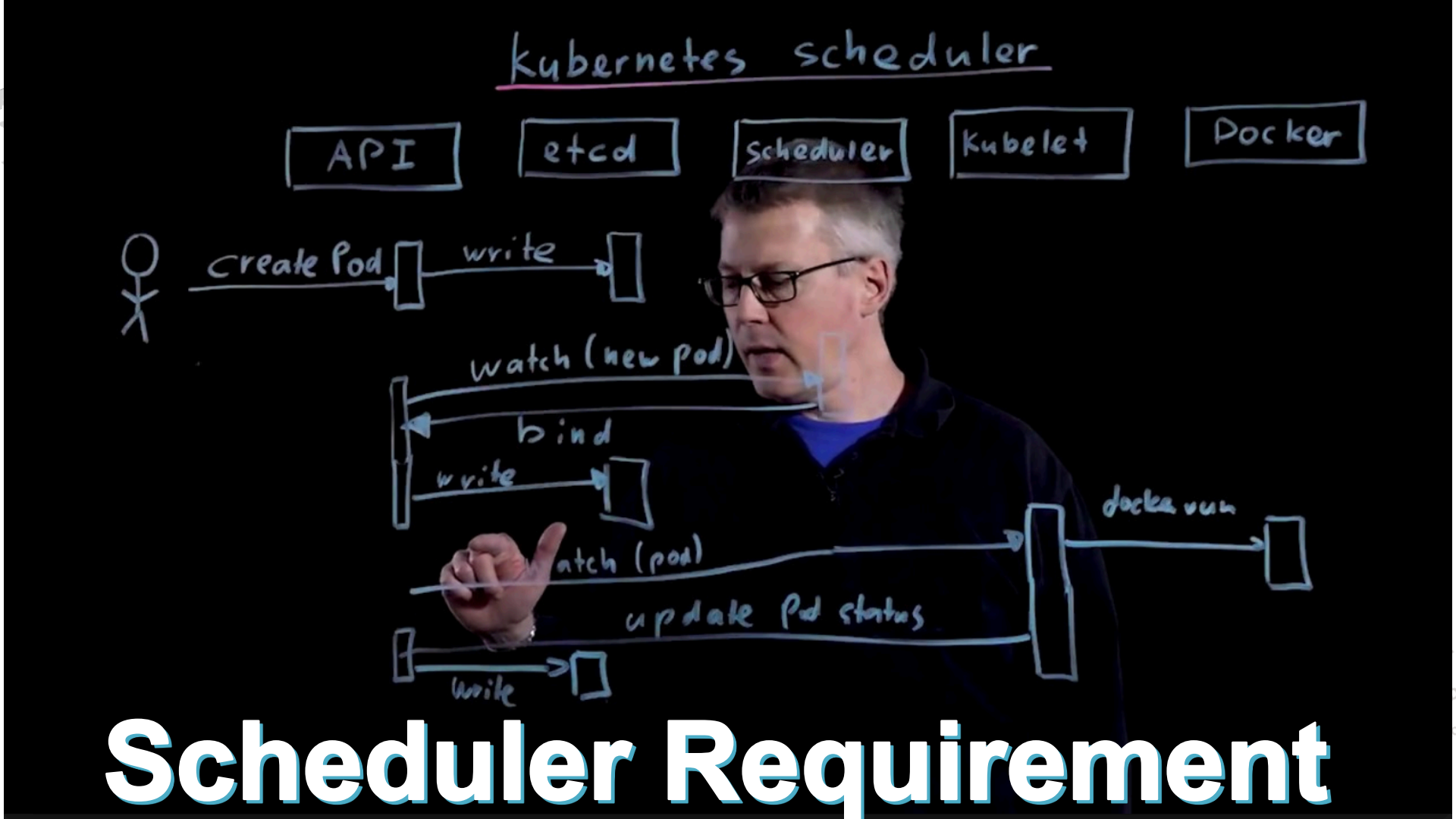
```
job foo = {
  ...
  dependencies = {
    parent=bar
  }
}
```

three types of dependencies between jobs: parent, after and cleanup.

Use after to specify that you want this job's tasks to run only after the specified job or jobs have successfully completed. If you want all the jobs in this dependency relationship to use the same set of local data, you can run them in the same allocation.

Use parent to make this job the child job of another job. A child job‘s tasks can only run as long as its specified parent is running, and are killed when the parent job is terminated. A job can have multiple child jobs but only one parent.

Use cleanup if your job spawns other jobs that you want to be killed when you stop the main job.



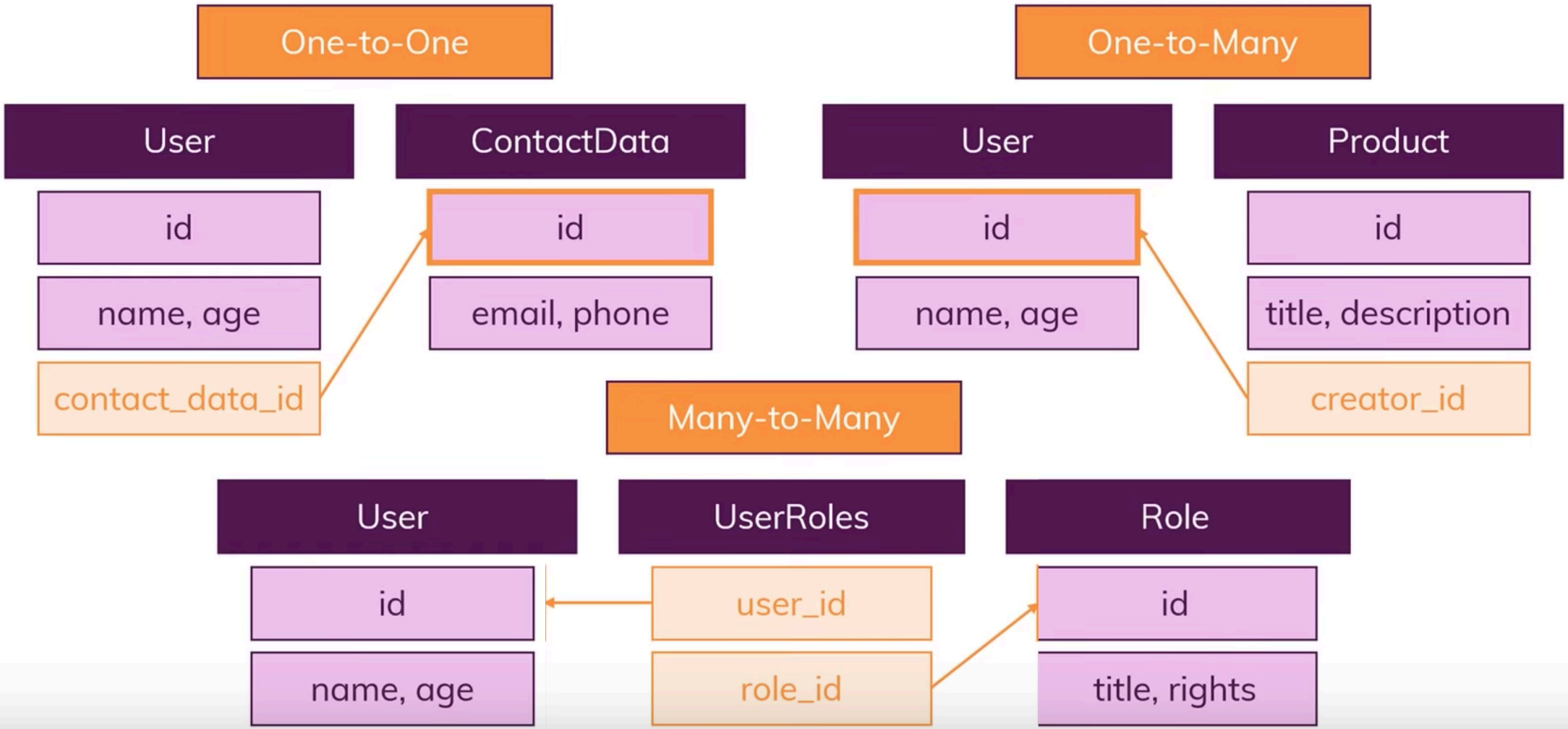
Scheduler Requirement

- Cancellation - you often want to kill a long running job, or prevent one from running.
- Priority - you often want high priority jobs to run in preference to low priority jobs. But implementing this in a way that low priority jobs don't wait forever in system where lots of jobs are generated is "non-trivial"
- Resources - some jobs may only be schedulable on systems which have certain resources. E.g. some will require large amounts of memory, or fast local disk, or fast network access. Allocating these efficiently is tricky.
- Dependencies - some jobs may only be runnable once other jobs have completed, and thus can not be scheduled before a given time.

- Deadlines - some jobs need to be completed by a given time. (or at least be started by a given time.)
- Permissions - some users may only be able to submit jobs to certain resource groups, or with certain properties, or a certain number of jobs, etc.
- Quotas - some systems give users a specified amount of system time, and running a job subtracts from that. This could make a significant difference to the numbers in your example.
- Suspension - some systems allow jobs to be check-pointed and suspended and the resumed later.

Database Schema

Types of Relations



Step 1

Task

Step 2

One to Many

Step 3

```
<!-- tasks begin -->
<createTable tableName="task">
  <column name="id" type="varchar(36)">
    <constraints primaryKey="true" nullable="false"/>
  </column>
  <column name="project_id" type="varchar(36)">
    <constraints nullable="true"/>
  </column>
  <column name="state" type="varchar(255)"/>
  <column name="started_time" type="timestamp"/>
  <column name="queued_time" type="timestamp"/>
  <column name="end_time" type="timestamp"/>
  <column name="resource_properties" type="varchar(10240)">
    <constraints nullable="true"/>
  </column>
  <column name="entity_kind" type="varchar(20)"/>
  <column name="entity_id" type="varchar(36)"/>
  <column name="operation" type="varchar(255)"/>
</createTable>
```

```java
public class TaskEntity extends BaseEntity {

  public static final String KIND = "task";
  @Transient
  AtomicInteger nextStepSequence = new AtomicInteger();
  private String entityId;
  private String entityKind;
  @Enumerated(EnumType.STRING)
  private State state;
  @Enumerated(EnumType.STRING)
  private Operation operation;
  private Date startedTime;
  private Date queuedTime;
  private Date endTime;
```

```java
/**
 * Task state.
 */
public enum State {
  QUEUED,
  STARTED,
  ERROR,
  COMPLETED
}
```

```
<!-- steps begin -->
<createTable tableName="step">
  <column name="id" type="varchar(36)">
    <constraints primaryKey="true" nullable="false"/>
  </column>
  <column name="task" type="varchar(36)">
    <constraints nullable="true"/>
  </column>
  <column name="sequence" type="integer"/>
  <column name="state" type="varchar(255)"/>
  <column name="started_time" type="timestamp"/>
  <column name="queued_time" type="timestamp"/>
  <column name="end_time" type="timestamp"/>
  <column name="operation" type="varchar(255)"/>
  <column name="options" type="varchar(8000)">
    <constraints nullable="true"/>
  </column>
</createTable>
```

```java
public class StepEntity extends BaseEntity {

  public static final String KIND = "step";
  private int sequence;

  @Enumerated(EnumType.STRING)
  private State state;
  @Enumerated(EnumType.STRING)
  private Operation operation;
  private String options;
  private Date startedTime;
  private Date queuedTime;
  private Date endTime;
  @ManyToOne(fetch = FetchType.LAZY)
  private TaskEntity task;
  @OneToMany(mappedBy = "step")
  @Cascade({CascadeType.ALL})
  private List<StepErrorEntity> errors = new ArrayList<>();
  @OneToMany(mappedBy = "step")
  @Cascade({CascadeType.ALL})
  private List<StepWarningEntity> warnings = new ArrayList<>();
  @OneToMany(mappedBy = "step", fetch = FetchType.LAZY)
  @Cascade({CascadeType.ALL})
  private List<StepResourceEntity> resources = new ArrayList<>();
  @Transient
  private List<BaseEntity> transientResourceEntities = new ArrayList<>();
  @Transient
  private Map<String, Object> transientResources = new HashMap<>();
  @Transient
  private boolean disabled = false;
```

```java
/**
 * Step state.
 */
public enum State {
  QUEUED,
  STARTED,
  ERROR,
  COMPLETED
}
```

```java
public class StepDao extends AbstractDAO<StepEntity> {

  @Inject
  public StepDao(SessionFactory sessionFactory) {
    super(sessionFactory);
  }

  public List<StepEntity> findByState(String state) {
    return list(namedQuery("Step.findByState")
        .setString("state", state.toUpperCase()));
  }

  public List<StepEntity> findInTask(String taskId, Optional<String> state) {
    if (state.isPresent()) {
      return list(namedQuery("Step.findByStateInTask")
          .setString("state", state.get().toUpperCase())
          .setString("task", taskId));
    }

    return list(namedQuery("Step.findAllInTask").setString("task", taskId));
  }

  public List<StepEntity> findByTaskIdAndOperation(String taskId, Operation operation) {
    return list(namedQuery("Step.findByTaskIdAndOperation")
        .setString("task", taskId)
        .setString("operation", operation.toString()));
  }
}
```

Tombstone
Clean stale data task, step and etc.

```java
/**
 * TombstoneBackend is responsible for writing the tombstone entries for an object.
 */
@Singleton
public class TombstoneSqlBackend implements TombstoneBackend {

  private static final Logger logger = LoggerFactory.getLogger(TombstoneSqlBackend.class);

  private final TombstoneDao tombstoneDao;

  private final MaintenanceConfig maintenanceConfig;

  @Inject
  public TombstoneSqlBackend(TombstoneDao tombstoneDao, MaintenanceConfig maintenanceConfig) {
    this.tombstoneDao = tombstoneDao;
    this.maintenanceConfig = maintenanceConfig;
  }

  @Transactional
  public TombstoneEntity create(String entityKind, String entityId) {
    TombstoneEntity tombstone = new TombstoneEntity();
    tombstone.setEntityKind(entityKind);
    tombstone.setEntityId(entityId);
    tombstone.setTombstoneTime(System.currentTimeMillis());
    tombstoneDao.create(tombstone);
    return tombstone;
  }
```

```java
@Transactional
public List<TombstoneEntity> getStaleTombstones() {
  List<TombstoneEntity> tombstones = tombstoneDao.listByTimeOlderThan(
      System.currentTimeMillis() - this.maintenanceConfig.getTaskExpirationThreshold().toMilliseconds());

  logger.info("Found {} stale tombstones", tombstones.size());
  logger.debug("Stale tombstones are {}", tombstones);
  return tombstones;
}

@Transactional
public void delete(TombstoneEntity tombstone) {
  logger.debug("Deleting tombstone {}", tombstone);
  tombstoneDao.delete(tombstone);
  logger.debug("Deleted tombstone {}", tombstone);
}
```

Who cleans tombstones

Background Daemon

Task

Suspension

Cancellation

Failure

Resume? Tombstone?

Task

Suspension - Resume

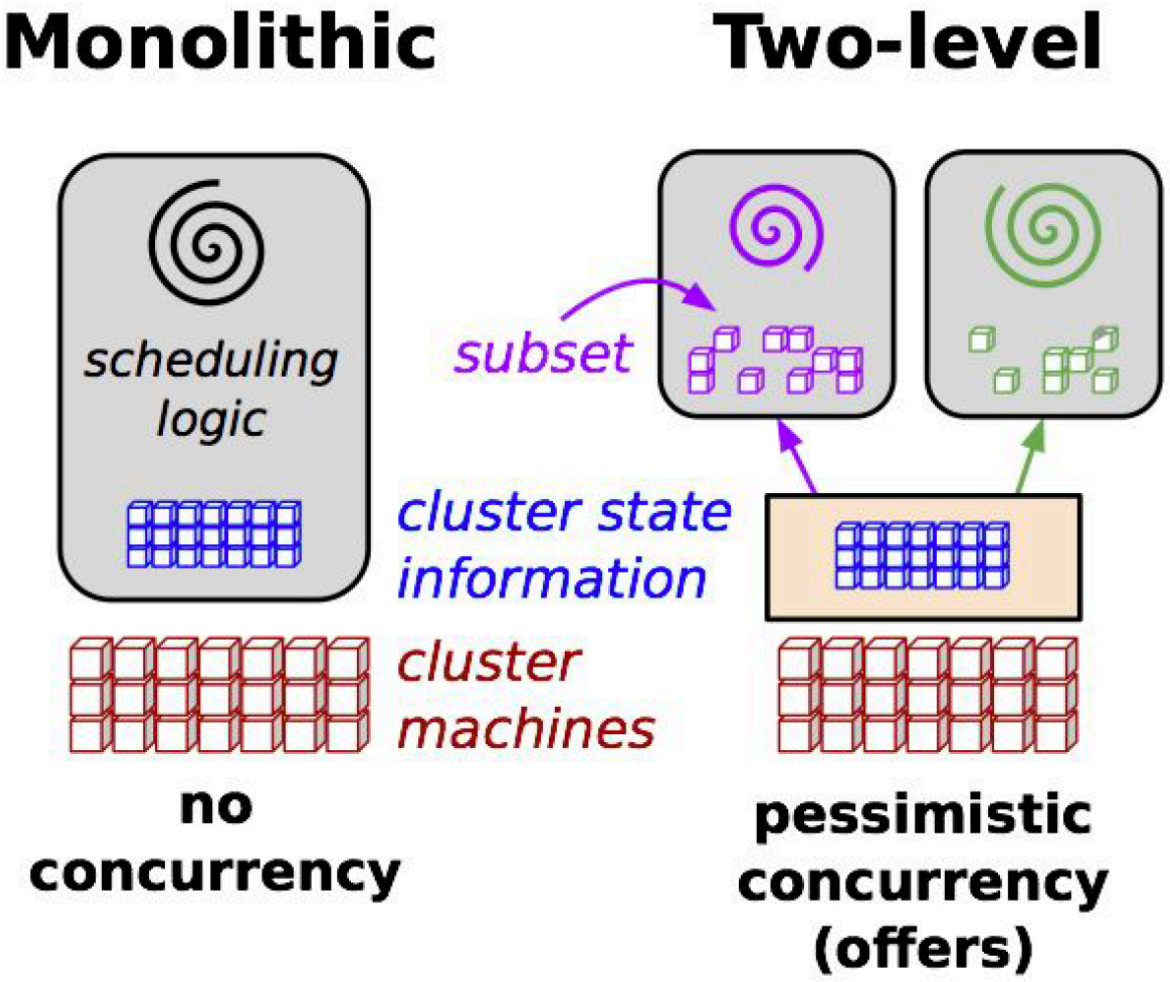
Cancellation - Tombstone

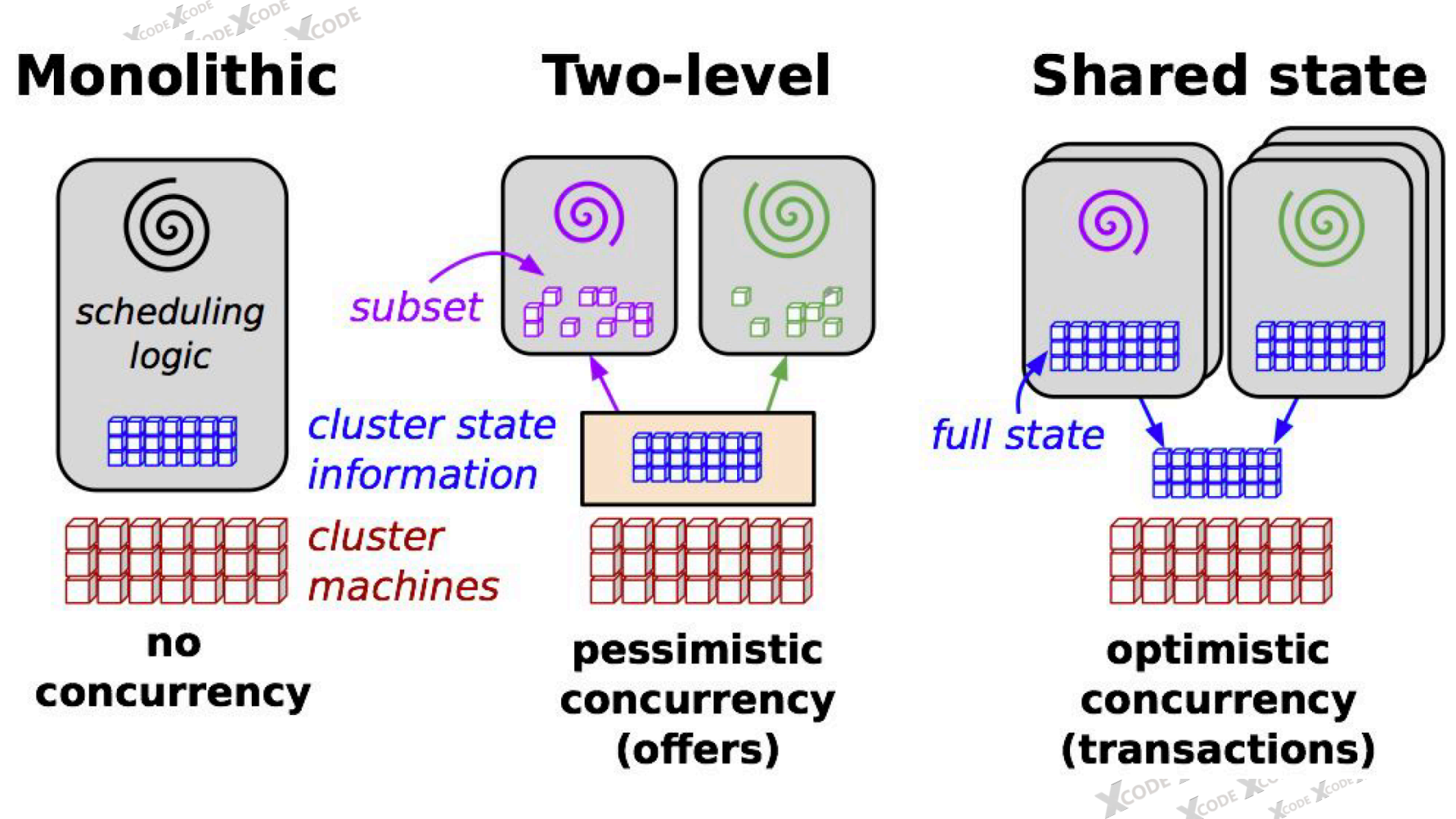
Failure– Retry or Tombstone

Scale Scheduler

Monolithic scheduler: a single process responsible for accepting workloads, ordering them, and sending them to appropriate machines for processing, all according to internal and user-defined policies. Single resource manager and single scheduler.

Two-level scheduler: Single resource manager serves multiple, parallel, independent schedulers using conservative resource allocation (pessimistic concurrency) and locking algorithms.


Monolithic
Two-level
scheduling logic
subset
cluster state information
cluster machines
no concurrency
pessimistic concurrency (offers)


Monolithic
Two-level
Shared state
scheduling logic
subset
cluster state information
full state
cluster machines
no concurrency
pessimistic concurrency (offers)
optimistic concurrency (transactions)

Customer Obsession

Leaders start with the customer and work backwards. They work vigorously to earn and keep customer trust. Although leaders pay attention to competitors, they obsess over customers.

Ownership

Leaders are owners. They think long term and don't sacrifice long-term value for short-term results. They act on behalf of the entire company, beyond just their own team. They never say "that's not my job".

Invent and Simplify

Leaders expect and require innovation and invention from their teams and always find ways to simplify. They are externally aware, look for new ideas from everywhere, and are not limited by "not invented here". As we do new things, we accept that we may be misunderstood for long periods of time.

Are Right, A Lot

Leaders are right a lot. They have strong judgment and good instincts. They seek diverse perspectives and work to disconfirm their beliefs.

Learn and Be Curious

Leaders are never done learning and always seek to improve themselves. They are curious about new possibilities and act to explore them.

Hire and Develop the Best

Leaders raise the performance bar with every hire and promotion. They recognize exceptional talent, and willingly move them throughout the organization. Leaders develop leaders and take seriously their role in coaching others. We work on behalf of our people to invent mechanisms for development like Career Choice.

Insist on the Highest Standards

Leaders have relentlessly high standards - many people may think these standards are unreasonably high. Leaders are continually raising the bar and drive their teams to deliver high quality products, services and processes. Leaders ensure that defects do not get sent down the line and that problems are fixed so they stay fixed.

Think Big

Thinking small is a self-fulfilling prophecy. Leaders create and communicate a bold direction that inspires results. They think differently and look around corners for ways to serve customers.

Frugality

Accomplish more with less. Constraints breed resourcefulness, self-sufficiency and invention. There are no extra points for growing headcount, budget size or fixed expense.

Earn Trust

Leaders listen attentively, speak candidly, and treat others respectfully. They are vocally self-critical, even when doing so is awkward or embarrassing. Leaders do not believe their or their team's body odor smells of perfume. They benchmark themselves and their teams against the best.

Dive Deep

Leaders operate at all levels, stay connected to the details, audit frequently, and are skeptical when metrics and anecdote differ. No task is beneath them.

Have Backbone; Disagree and Commit

Leaders are obligated to respectfully challenge decisions when they disagree, even when doing so is uncomfortable or exhausting. Leaders have conviction and are tenacious. They do not compromise for the sake of social cohesion. Once a decision is determined, they commit wholly.

Deliver Results

Leaders focus on the key inputs for their business and deliver them with the right quality and in a timely fashion. Despite setbacks, they rise to the occasion and never settle.