Cloud native applications, infrastructure and patterns

Part 2: Architecture and Orchestration

Renaud Lachaize & Thomas Ropars

Univ. Grenoble Alpes
M2 MoSIG

October 2025

Main references (1/2)

- B. Scholl, T. Swanson, P. Jausovec. **Cloud native: Using containers, functions, and data to build next-generation applications**. O'Reilly, 2019.

- J. Garrisson, K. Nova. **Cloud-native infrastructure: Patterns for Scalable Infrastructure and Applications in a Dynamic Environment.** O'Reilly, 2017.

- B. Burns, J. Beda, K. Hightower, L. Evenson. **Kubernetes up & running (3rd edition)**. O'Reilly, 2022.

- B. Burns. **Designing distributed systems. Patterns and paradigms for scalable, reliable services**. O'Reilly. 1st edition (2018) or 2nd edition (2025).

Main references (2/2)

- Kubernetes documentation: https://kubernetes.io/docs/home/

- Cloud Native Computing Foundation (CNCF) Web site: https://www.cncf.io

- Jordan Webb. The Container orchestrator landscape. LWN.net. August 2022. https://lwn.net/Articles/905164/

- How Kubernetes reinvented virtual machines. Ivan Velichko. August 2022. https://iximiuz.com/en/posts/kubernetes-vs-virtual-machines/

Outline

- ~~Origins and main characteristics~~
- **Microservices**
- Container orchestration
- Design patterns

Microservices (1/2)

- An architectural style for the design of applications and software services.
- This paradigm has connections and synergies with other software engineering/management trends such as DevOps and 12-factor apps.

- A microservice architecture is **a service-oriented architecture** (SOA), in which **applications are decomposed into small, loosely-coupled services**, with **each service being relatively small and focused** on a specific functionality.

- The opposite of a “software monolith”, a.k.a. a “monolithic architecture”.

Microservices (2/2)

- Each service has an **independent codebase**.
- Each service is **owned, developed/maintained, and operated by a small team**.

- Each service can be viewed as its own mini-application, with independent tests, builds, configuration, data and deployments.
- **Services are executed in separate processes and communicate through network APIs**.
- The API of a given service can be synchronous or asynchronous.

Microservice-based architectures – Examples (1/2)

Hipster Shop

- An open-source demo application created by Google
- E-commerce Web application
- Available from: https://github.com/GoogleCloudPlatform/microservices-demo

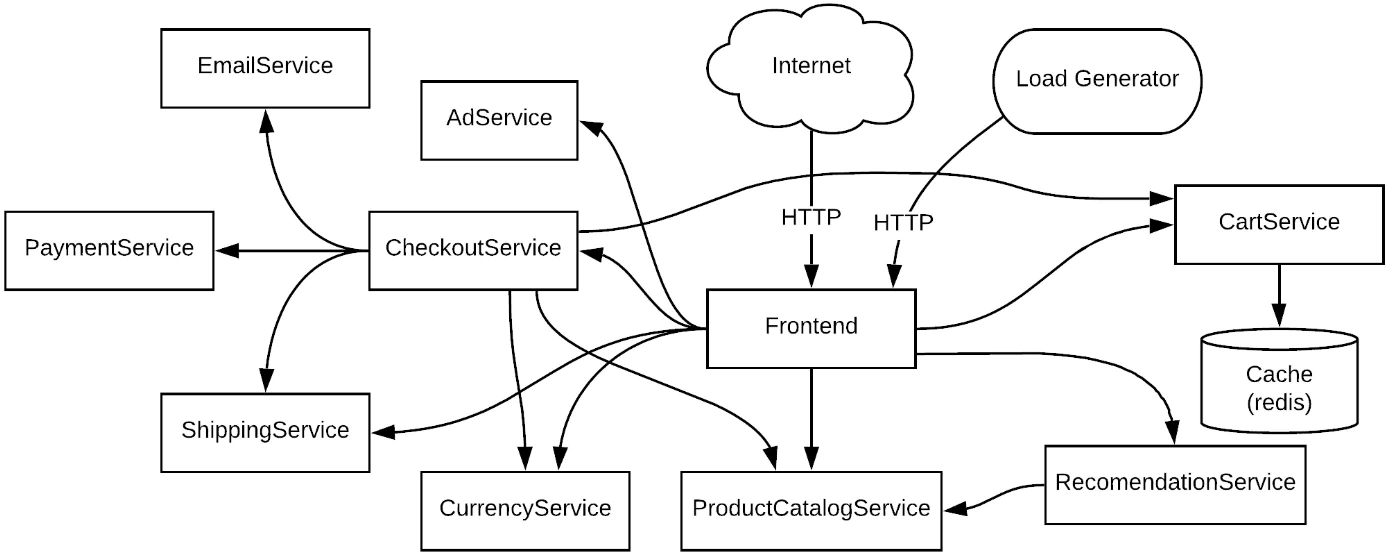


Image credit: https://github.com/GoogleCloudPlatform/microservices-demo

Microservice-based architectures – Examples (2/2)

DeathstarBench

- A set of open-source demo applications/benchmarks created by Cornell University
- Several applications: Social network (see figure below), Hotel reservation, Media service, …
- Code available from: https://github.com/delimitrou/DeathStarBench/
- Research paper: http://www.csl.cornell.edu/~delimitrou/papers/2019.asplos.microservices.pdf

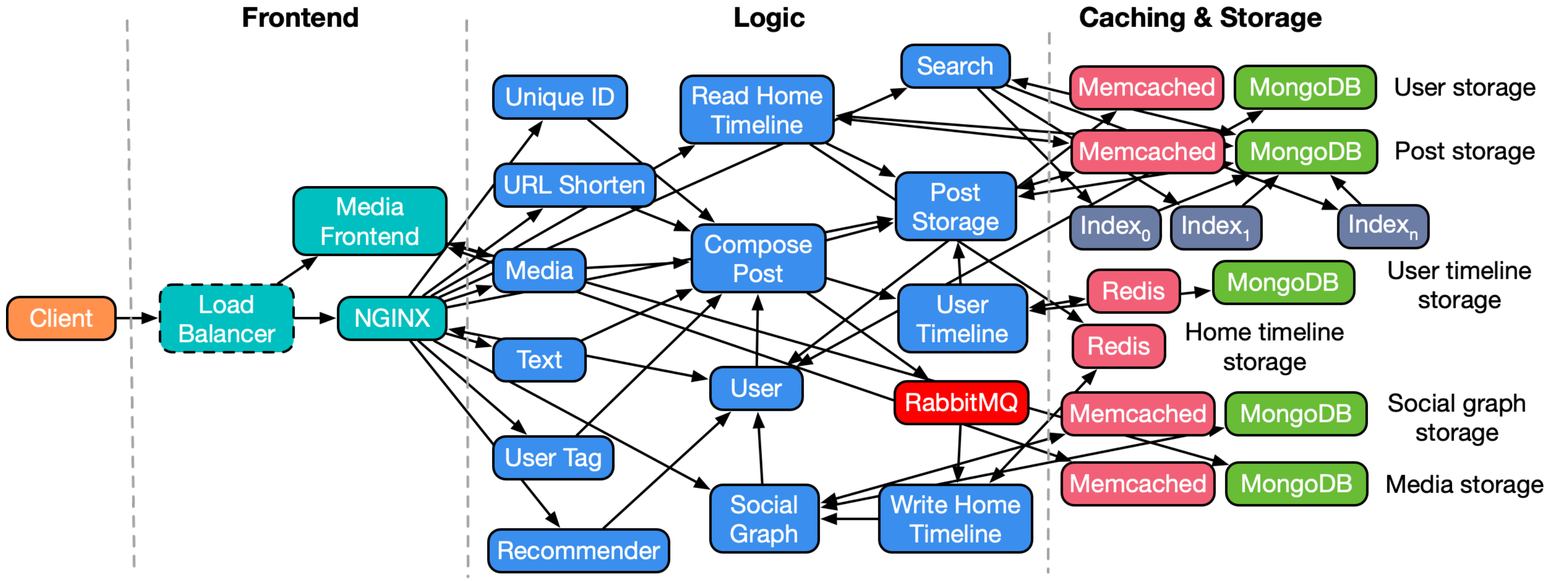


Image credit: https://github.com/delimitrou/DeathStarBench/tree/master/socialNetwork

Microservices – Potential benefits

- Small and focused teams
- Better communication between teams
- Increased agility, enabling faster evolution and more frequent deployments
  - “Continuous integration” (CI) and “Continuous delivery” (CD)
- Continuous innovation (e.g., experimentation with new features through A/B testing)
- **Modularity**
  - Simplified support for localized changes
  - **“Polyglot” design**: Simplified support for using diverse technologies (programming languages and frameworks, libraries, storage systems) in different parts of the global application.
- Improved fault isolation
- Better scaling and optimized resource usage
- Improved observability

Microservices – Challenges

- **Complexity** (e.g., due to increased distribution)
- API versioning and integration
- Data integrity and consistency
- Monitoring and logging
- Service discovery and routing
- Availability
- Performance issues (e.g., increased average and tail latencies)
- Achieving insightful and efficient end-to-end observability (distributed, large-scale debugging and profiling

Communication protocols (1/4)

All interactions between apps/services are based on network (layer-5 / application-level) protocols. We will describe the main ones.

- **Request-response protocols**
  - Can be either **synchronous** (client blocks until reply) or **asynchronous** (non-blocking)
  - The asynchronous pattern:
    - is more complex but more efficient and resilient (introduces less coupling between services).
    - requires identifiers for associating requests and responses.
  - Main protocols: HTTP 1.1 and HTTP/2

Communication protocols (2/4)

- **Request-response variants:**
  - **Message queues**
    - An asynchronous request-response pattern can also be implemented with message queues.
    - Using one (single consumer) queue for each direction.
    - Example: AMQP (point-to-point mode), NATS (request-reply mode)
  - **Websockets**
    - Supports low-latency, bidirectional communication (server can send messages without awaiting client requests).
    - Uses HTTP as bootstrap.
  - **Remote Procedure Calls**
    - Higher-level construct (procedure calls rather than messages). Supports bidirectional streams.
    - Example: gRPC (by Google - https://grpc.io) based on HTTP/2

Communication protocols (3/4)

- **Publish-subscribe protocols:**
  - Allow decoupling message producers (“publishers”) from message consumers (“subscribers”) through topics subscriptions.
  - Are **inherently asynchronous**.
  - Support unidirectional, one-to-many or many-to-many communication patterns.
  - Enable loose coupling between services, and even transparent integration.
  - Example: AMQP (pub-sub mode), NATS (pub-sub mode)

Communication protocols (4/4)

- **Remark: Idempotent requests**
  - Because cloud applications are prone to (machine/service/communication) failures, a given message may be sent and received multiple times.
  - **Regardless of the chosen communication pattern and protocol,** in order to handle failures in a correct and simple way**, it is generally advised to design idempotent application-level protocols.**
  - **A message is idempotent if processing it N times produces the same result as processing it once.**
  - This requires:
    - Using unique message identifiers.
    - Designing services and data models that keep track of such identifiers (or do not need them)

Gateways

- **Act as a stable entry point for requests sent by external clients to a cloud service**.
- **Play various roles:**
  - SSL termination (i.e., termination of the SSL channel established between a client and the entrance of the cloud platform).
  - Client/service authentication.
  - Abstracting the internal (and possibly moving) architecture of a high-level service (e.g., monolithic design vs microservice architecture).
  - Abstracting the dynamic nature of the service implementation (e.g., due to autoscaling and fault tolerance).
  - Providing access to (relatively) static resources/data (e.g., HTML/CSS files).
- **A single gateway can perform one or several roles.**
- **Several gateways may be layered.**

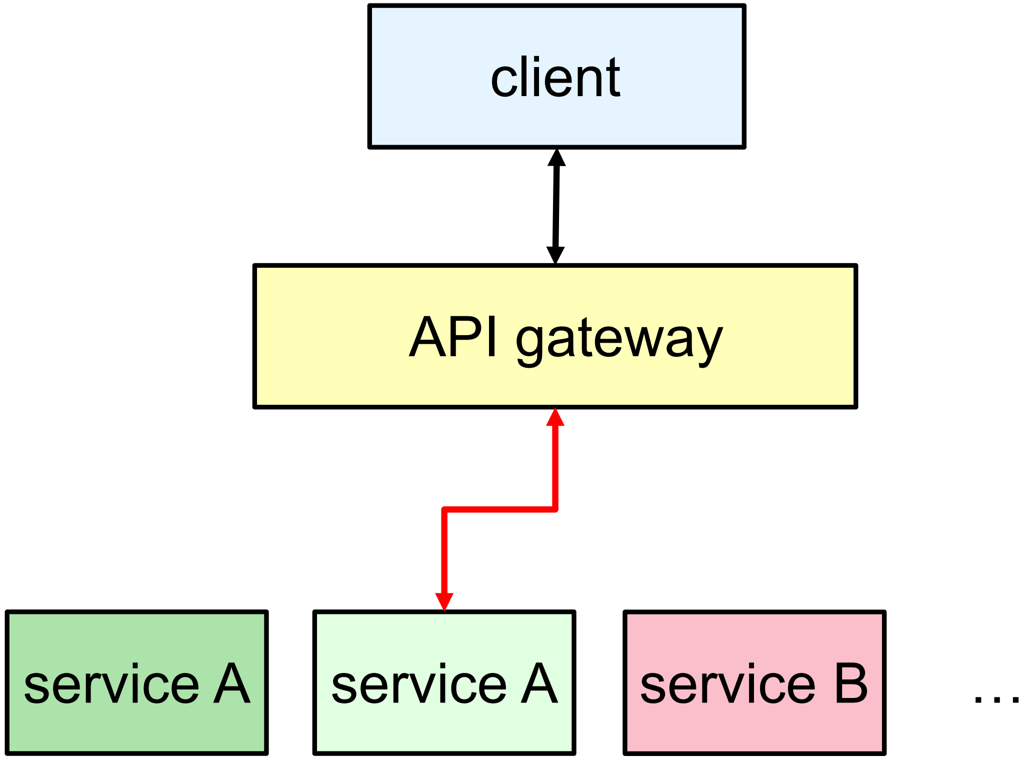
API gateway (1/6)

- **A particular type of gateway in the context of a microservice architecture**.
  - Exports a high-level service API to the external clients (typically HTTP/REST-based).
  - Forwards the client requests to the appropriate service(s) in the backend architecture.
- **Main benefits:**
  - **Decouples clients from services** (e.g., services can be versioned or refactored without introducing strong dependencies).
  - Allows using non-HTTP protocols for the services (e.g., messaging protocols).
- **Main responsibilities:**
  - **Routing**
  - **Aggregation**
  - **Feature offload**

API gateway (2/6)

Routing

- **One of the most common functions of a gateway.**
- **Forwards requests to backend services** (which are in a private network not directly accessible from clients).
- The selection of the recipient service instance(s) is typically based on routing rules defined with respect to:
  - In-request metadata such as IP, port, protocol headers, target URL
  - Gateway-level statistics regarding load balancing, availability of the different service replicas



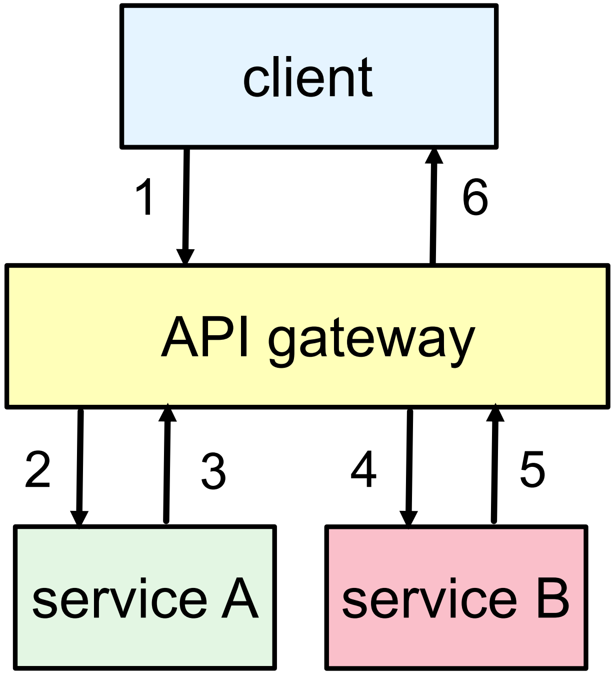
API gateway (3/6)

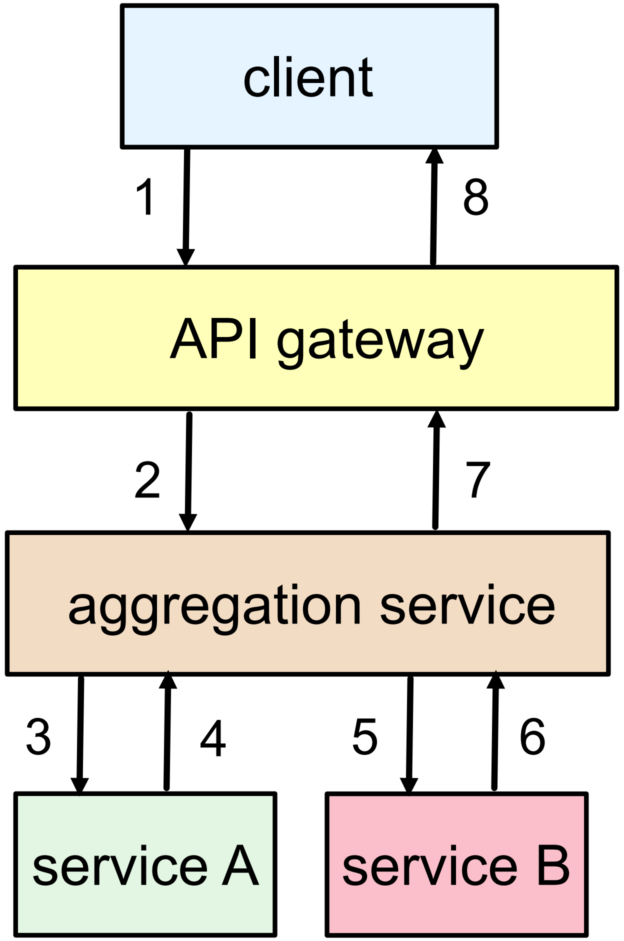
Aggregation

- A gateway may also act as an aggregator.
- In such a setup, a single external client request received by the gateway is translated (by the gateway itself) into a sequence of requests to multiple services.
- Benefits:
  - Provides higher-level operations to the clients.
  - Helps improving the client-perceived latency by reducing the number of communication round-trips between the client and the cloud service.
- Drawbacks:
  - May increase coupling between the gateway and the service. The gateway may itself become a monolith.
  - May significantly increase the processing load on the gateway.
- An alternative design consists in using a separate aggregation service.

API gateway (4/6)

Aggregation





API gateway (5/6)

Offloading

- A gateway can also offload some functionality from the individual services, if such functionality is a “cross-cutting” concern (i.e., not a part of the business logic of a service).
- Typical examples:
  - Authentication
  - Logging
  - Rate limiting, retry policies, circuit breaking
  - Compression
- Like for aggregation, putting too many features on a gateway introduces risks of reintroducing monoliths or performance bottlenecks.

API gateway (6/6)

- **An API gateway may become a performance bottleneck and/or a single point of failure.**
- It is thus necessary:
  - To **monitor the performance/load of a gateway (not just the backend services)**
  - To **replicate the gateway and use load balancing** (e.g., via DNS services)

- API gateways are typically implemented using HTTP reverse proxies (for example: Nginx, HAproxy).

- API gateways can be deployed and managed directly by cloud tenants
- … or cloud tenant can also choose to rely on a **managed gateway service** provided by the cloud platform. Examples: AWS API Gateway, Google Cloud Endpoints.

Internal vs. external service communication

There are various categories of network traffic in a (cloud) cluster:

- **Ingress traffic (a.k.a. “North-South” traffic)**
  - **Requests from external clients** into the cluster
  - Handled by **“API gateways”** (also known as **“ingress controllers”** or “ingress gateways”)
- **Internal traffic (a.k.a. “East-West” traffic)**
  - **Communications between (micro)services** running on the cluster
  - Handled by the distributed **“service mesh”** (this concept will be detailed later)
- **Egress traffic**
  - **Communications from a service** running on the cluster **to an external service** (e.g., a remote storage service)
  - Handled by **“egress gateways”** (also known as “egress controllers”)
    - Crucial for monitoring and controlling outgoing requests (e.g., for security purposes)
    - Typically implemented at the level of the service mesh

Service mesh (1/9)

- A microservice architecture introduces complex and frequent communications between various services.
- Given that failures are likely to happen, **services must be resilient and almost immune to communication issues**.

- As mentioned previously ("declarative communications"):
  - **Dealing with network issues and advanced network configuration aspects (e.g., dynamic service discovery, security) should be addressed at the infrastructure level rather within applications**.
  - This yields simpler and more robust applications and supports dynamic modifications of the networking configuration.
  - **Within a cloud native infrastructure, a service mesh is a distributed system in charge of managing these aspects.**

Service mesh (2/9)

- The main building block of a service mesh is a **proxy**.
  - **There is typically one proxy instance running next to each service instance.**
    - (In other words, there are many proxies deployed on each physical host of the system.)
    - Some designs also support an alternative setup with a single proxy per host.
    - Using one proxy per service results in higher resource consumption but yields simpler configurations and higher performance.
  - **A proxy performs the following tasks:**
    - **Intercepting all the incoming and outgoing traffic** of its associated service instance.
    - **Handling the intercepted traffic** according to specific configuration code/rules.
    - **Collecting and reporting metrics.**

Service mesh (3/9)

- **Examples** of proxy projects/implementations used in service meshes:
  - Envoy (https://www.envoyproxy.io)
  - Istio (https://istio.io)
  - Linkerd (https://linkerd.io)
  - Consul connect (https://www.consul.io)
  - (Note: some of these projects can also be used for other purposes.)

- Notes:
  - Proxies are often deployed using a “sidecar container” pattern.
  - In the Kubernetes architecture, a proxy runs in the same “Pod” as its associated service.
  - (These notions will be described in more details later.)

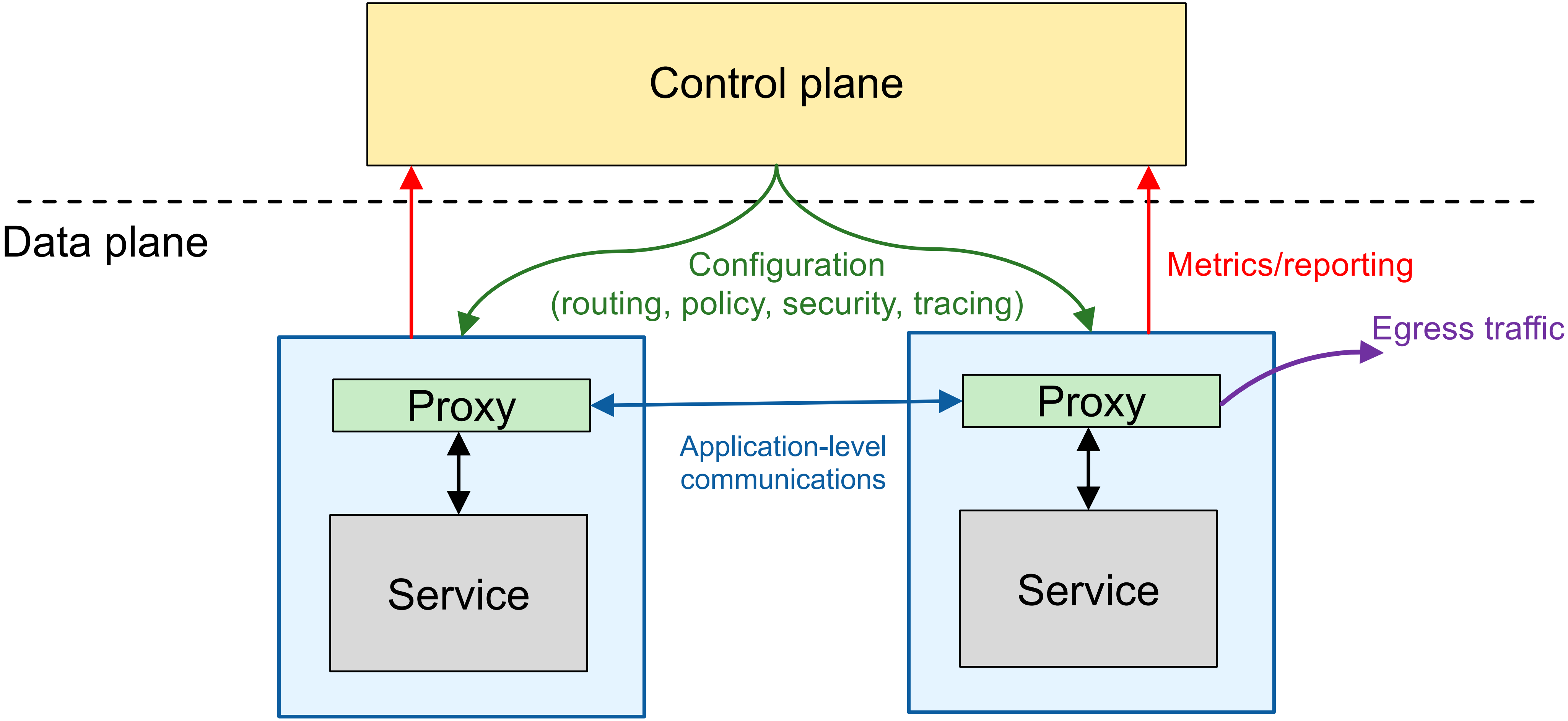
Service mesh (4/9)

Distributed architecture:

- The collection of all the proxies in the service mesh is called the **“data plane”**.

- The component in charge of managing the data plane is called the **“control plane”**.
  - In particular, the control plane is in charge of deploying a consistent, distributed configuration on all the proxies and of reconfiguring them as needed over time.
  - The control plane usually exposes an API that can be used to configure the service mesh.
  - In addition, the control plane hosts other tasks (e.g., for telemetry and policy of the hosted services).

Service mesh (5/9)



Service mesh (6/9)

The main concerns handled by a service mesh can be divided into the following categories:

- Traffic management
- Failure handling
- Security
- Tracing and monitoring

Service mesh (7/9)

Traffic management:

- General goals:
  - Managing traffic between services within the mesh
  - Managing traffic between in-mesh services and external services
- Within a mesh, each service can have multiple endpoint instances.
- **The service mesh is in charge of performing request routing to an endpoint instance, according to considerations such as:**
  - Service discovery (identification of new/removed instances)
  - Load balancing
  - Service versions
  - Environment type (e.g., testing vs production)
- Various criteria can be used for routing decisions:
  - Proxy configuration settings, proxy statistics, request contents/metadata

Service mesh (8/9)

Failure handling:

- **General goals:**
  - Maintain service availability and correctness whenever possible
  - Mitigate the performance impact of failures
- Communication failures can (and will) occur due to various network or infrastructure issues.
- **Two types of failures:**
  - Transient: generally solved by request retries
  - Non-transient: require more complex/specific handlers
- **Basic techniques/parameters:**
  - Retries
  - Timeouts
  - Circuit breakers (rejecting further accesses to a failed service, in order to avoid cascading failures)
- The above mechanisms can be configured for each service (or service version).
- The service mesh can also be used to inject failures (request aborts or delays) for testing purposes.

Service mesh (9/9)

- **Security:**
  - General goals:
    - Authentication (client/server, end user)
    - Authorization / access control (client/server, end user)
    - Confidentiality (traffic encryption)
    - Management of secrets (credentials, keys)

- **Tracing and monitoring:**
  - General goals:
    - Collecting and reporting metrics regarding traffic characteristics, performance, errors
    - End-to-end (i.e., cross-service) request tracing (via correlated request identifiers), to facilitate traffic observation, troubleshooting and performance debugging

Outline

- Origins and main characteristics

- Microservices

- **Container orchestration**

- Design patterns

Container orchestration (1/2)

- **Vocabulary warning**: Not to be confused with the expression “service orchestration”, which is often used with a somewhat different meaning.

- A **container orchestrator** is:
  - **a software system**
  - … in charge of **simplifying** (through automation) **the management of a fleet of container-based applications**
  - … **on a cluster** of (virtual or physical) machines.

Container orchestration (2/2)

The **main duties** of a container orchestrator include:

- The **provisioning** and **deployment** of containers
- The setup of the **network configuration** of the containers

- The **placement decisions** for the containers on the cluster nodes

- **Health monitoring** (for containers and nodes) and appropriate reactions when needed

- **Load balancing** between containers
- **Autoscaling** decisions (mostly for horizontal scaling but also possibly for vertical scaling)

Container orchestration – Case study: Kubernetes

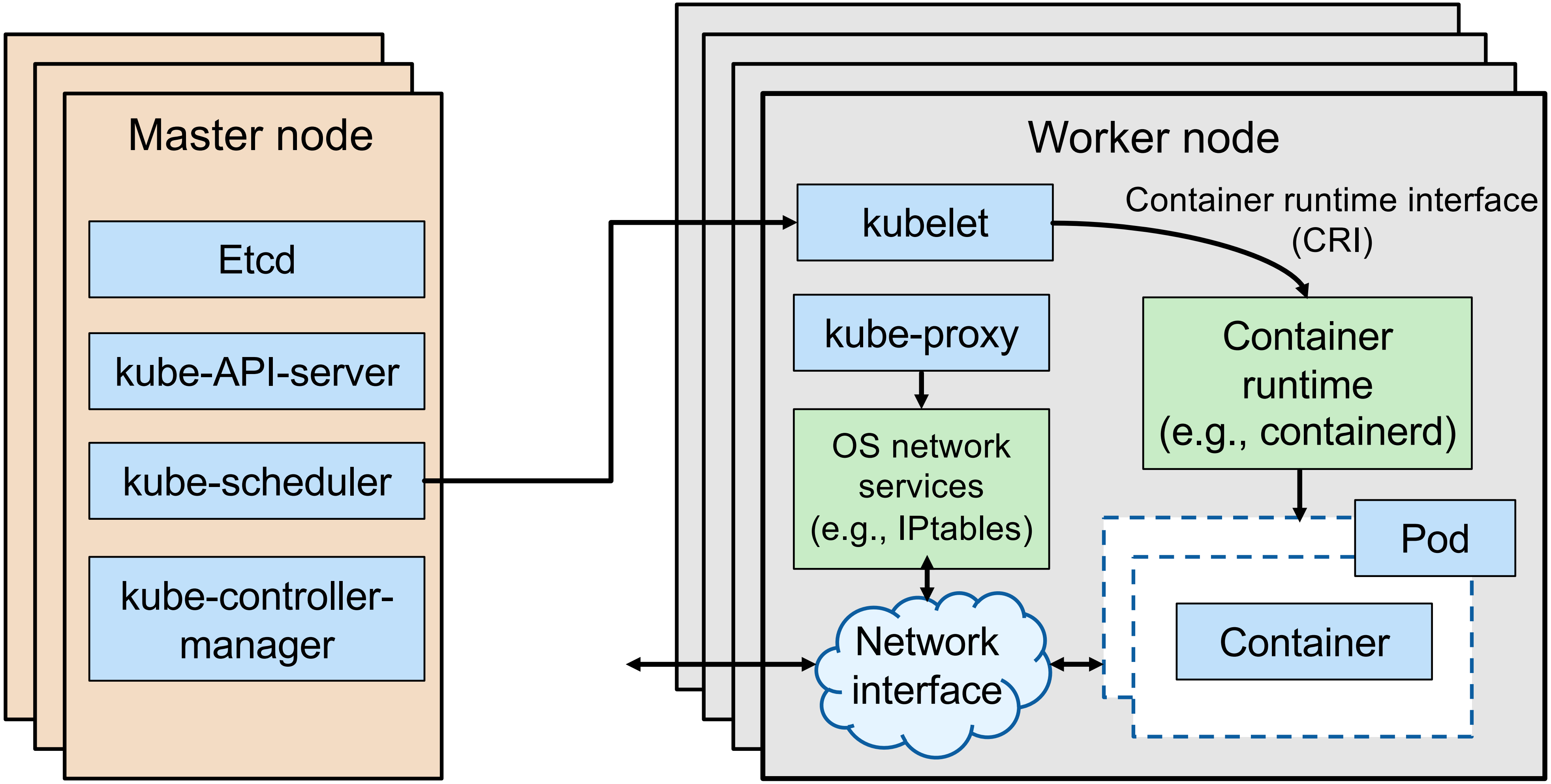
- Kubernetes: currently **the most popular container orchestrator**
- Also known under the abbreviation “k8s”
- A project initially developed by Google and influenced by the design of previous (closed, internal) cluster/container management systems used within the company.
- Released as open source in 2014. Now managed by the Cloud Native Computing Foundation (CNCF).
- Implemented in the Go language.

- A good/short introductory reference (for the description of the design principles):
  - B. Burns et al. Borg, Omega and Kubernetes: Lessons learned from three container-management systems over a decade. ACM Queue. January-February 2016. https://static.googleusercontent.com/media/research.google.com/fr//pubs/archive/44843.pdf

Kubernetes components

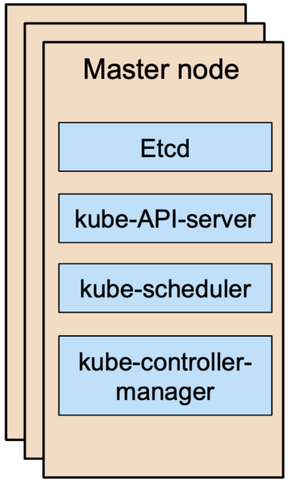
- Kubernetes is a distributed system based on 3 main categories of components.
  - **Master components:** Provide the **“control plane”** of the cluster.
    - Make global (cluster-wide) decisions. E.g., regarding the scheduling/placement of tasks, the number of replicas for a given service, the response to specific events like failures.
    - Are typically deployed on dedicated nodes for better robustness.
  - **Node components:** Provide the **“data plane”** of the system. Act as local agents on the cluster resources :
    - For health and performance monitoring
    - For handling the orders issued by the control plane (e.g., running a new container)
    - For setting up the network connectivity of the containers
  - **Addons**: Provide additional services, such as:
    - DNS services
    - High-level user interfaces (e.g., management/monitoring dashboards)

Kubernetes components overview



Kubernetes components – Details (1/5)

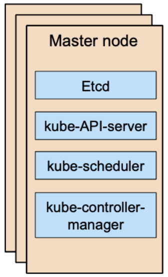
- **Master components**



  - **Kube-api-server**: the **front end for the Kubernetes control plane**
    - Exposes the API of the control plane
    - Designed for horizontal scaling (load balancing)
  - **Etcd**: a strongly consistent and highly available **key-value database**
    - Used to store all the cluster configuration/metadata, including the current state and desired state of the cluster
  - **Kube-scheduler**: monitors the creation of new “Pods” and selects nodes for them
    - Placement decisions for Pods are based on various factors, including: individual and collective resource requirements, hardware/software/policy constraints, affinity and anti-affinity specifications, data locality …

Kubernetes components – Details (2/5)

- **Master components (continued)**

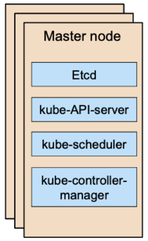


  - **Kube-controller-manager**: runs (most of) the Kubernetes controllers
    - Logically, each controller is a distinct process
    - However, to reduce complexity, all controllers are compiled into a single binary and run in a single OS process
    - Some controllers can also run externally

  - **Cloud-controller-manager**: runs controllers that interact with the underlying cloud provider
    - Allows the cloud provider's code and the Kubernetes code to evolve independently of each other

Kubernetes components – Details (3/5)

Remarks:

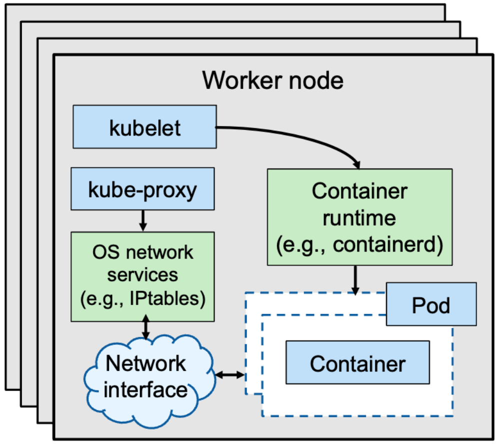


- Master components are **replicated on multiple master nodes** for failover and high availability.

- **“Managed Kubernetes” offerings** typically offload the burden of hosting/administrating these master components from the end-users (cloud tenants).
  - Examples : Google Kubernetes Engine (GKE), Azure Kubernetes Service (AKS), AWS Elastic Kubernetes Service (EKS).

Kubernetes components – Details (4/5)

- **Node components**: Run on every compute (worker) node of the cluster


Worker node
kubelet
kube-proxy
Container runtime (e.g., containerd)
OS network services (e.g., IPtables)
Pod
Network interface
Container

  - **Kubelet**: local agent in charge of running containers in "Pods"
    - Reads and applies the Pods specifications
    - Checks the health of the containers
    - Does not manage the containers that were not created by Kubernetes
  - **Kube-proxy**: local network proxy
    - Maintains the network configuration/rules on the cluster nodes (regarding communication with other cluster nodes and external nodes), in order to support the Kubernetes "service" concept.
    - Typically relies on the local OS services (e.g., packet filtering) to implement traffic forwarding.
  - **Container runtime**: in charge of executing containers
    - The Kubernetes CRI (Container Runtime Interface) provides an abstract interface that allows plugging various runtime implementations (e.g., Docker, containerd, cri-o, …)

Kubernetes components – Details (5/5)

- **Add-ons**: Some examples
  - **DNS**: Serves DNS records for Kubernetes services
    - Containers started by Kubernetes automatically include this DNS server for their DND lookups
  - **Web UI** (Dashboard)
    - General-purpose Web-based user interface for Kubernetes clusters
    - Allows managing and troubleshooting the Kubernetes cluster, and applications running on the cluster
  - **Container resource monitoring**
    - Records generic time-series of metrics about containers in a centralized database
    - Provides a user interface to browse the collected data
  - **Cluster-level logging**
    - In charge of collecting and saving container logs to a central store with a search/browse interface.

Kubernetes – Main concepts (1/4)

- **Pods**:
  - **A pod is an abstraction encapsulating a set of containerized components.**
  - A pod is the basic scheduling unit for Kubernetes.
  - There are two main setups in practice:
    - Pod with a single container
    - Pod with multiple containers that work together
  - **All the containers within a pod are guaranteed to be deployed on the same machine and can share resources.**
  - Each pod runs a single instance of an application (there is no replication/sharding within a pod).
  - Networking:
    - Each pod is assigned a unique IP address.
    - Containers within the same pod share the same network namespace (including network ports).
    - Containers within the same pod can communicate using localhost.
  - Storage:
    - A pod can specify a set of shared storage volumes, which can be accessed by all the containers within the pod.

Kubernetes – Main concepts (2/4)

- **Service**:
  - In Kubernetes, a “service” is an “endpoint” abstraction allowing to expose an application (running as a set of pods, characterized by a given **“label”**) as a network service.
  - Useful to abstract the fact that the pod instances providing a given functionality can vary over time (e.g., due to scaling or load balancing).
- **Volume**:
  - On-disk files in a container are ephemeral (they disappear upon termination of the container), which can be limiting in certain setups/situations.
  - Volumes survive container crashes/restarts and support sharing between containers in a pod.
  - The lifetime of a volume corresponds to the lifetime of the enclosing pod.
  - Many types of backing storage systems are supported.
- **Persistent volume:**
  - Unlike a standard volume, a persistent volume can exist/survive independently from the pod(s) that may consume it. The same persistent volume may be used successively by different pods.

Kubernetes – Main concepts (3/4)

- **Namespace**:
  - Warning: Same word but not the same concept as the notion of (Linux) namespaces used to implement containers.
  - Kubernetes allows using multiple virtual clusters on the same physical cluster.
  - Such a configuration is useful for environments in which there are many users spread across multiple teams and projects.
  - The notion of “namespaces” is used to define such virtual clusters.
    - Provides a scope for naming resources (a resource name must be unique within a given namespace).
    - Supports resource quotas to control the division of resources.

Kubernetes – Main concepts (4/4)

- **Controllers**:
  - A concept inspired from other fields like robotics and automation/control theory.
  - A **“control loop”:**
    - is an infinite loop whose purpose is to **regulate the behavior of the system**.
    - continually monitors the current state of the system (here, a cluster) and acts accordingly **in order to reach a desired state** (analogy: a thermostat).
  - **Kubernetes uses many different controllers**, each in charge of a specific aspect of the cluster state.
    - A set of **built-in controllers** (running inside the kube-controller-manager) supporting the important core behaviors.
    - Also possibly additional, **user-defined controllers** (that can run within Kubernetes, as a set of pods, or outside).

Kubernetes – Some examples of controllers (1/2)

- DaemonSet: **ensures that all (or some) nodes of the cluster run a copy of a Pod.**
  - For example, useful to run a storage daemon or a logging/monitoring daemon on every node.

- ReplicaSet: **maintains a stable number of replica pods running at any given time.**
  - Typically used to enforce a given replication factor in order to provide high availability guarantees.

- Deployment: **provides declarative updates for Pods and ReplicaSets.**
  - A ”deployment object” is used to describe a desired state.
  - Then the deployment controller changes the actual state at a controlled rate, in order to reach the desired state.
  - Useful for many purposes: rolling out and monitoring ReplicaSets, scaling ReplicaSets, updating Pods, rolling back to earlier deployment versions, …

Kubernetes – Some examples of controllers (2/2)

- StatefulSet: **manages applications that have one or more of the following requirements:**
  - Stable, unique network identifiers
  - Stable, persistent storage
  - Ordered, graceful deployment and scaling
  - Ordered, automated rolling updates
- **Unlike a ReplicaSet**, **a StatefulSet has the following properties**:
  - Each replica gets a persistent hostname with a unique index (e.g., db-0, db-1, etc.)
  - Each replica is created in order from lowest to highest index, and the creation will pause until the Pod at the previous index is healthy and available. (The same principle also applies when scaling up the number of replicas.)
  - When a StatefulSet is deleted, the replica pods are deleted in order from highest to lowest. (The same principle also applies when scaling down the number of replicas.)

Kubernetes – Metadata management

- In Kubernetes, **all kinds of resource objects** (e.g., instances of Pods, ReplicaSets, Nodes, etc.) **carry some metadata**.
- These metadata are **useful for different purposes**, such as:
  - Keeping track of important information
  - Categorizing resource instances, and associating them with each other
  - Defining custom information and behaviors
  - Note: In most cases, these metadata do not imply any built-in semantics by default
- There are **different types** of metadata:
  - **Labels**: key-value pairs attached to an object (*analogy: catalog of dishes*)
  - **Selectors**: used to identify/filter objects based on their labels (*analogy: filter to find gluten-free deserts*)
  - **Annotations**: key-value pairs to store additional, non-identifying, arbitrary metadata, such as build information or release notes (*analogy: list of ingredients and their origins*)

Credits and more information:

- https://yuminlee2.medium.com/kubernetes-labels-selectors-and-annotations-2e3e931282a7
- https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/
- https://kubernetes.io/docs/concepts/overview/working-with-objects/annotations/

Kubernetes metadata – some details about labels (1/2)

- “Labels are intended to be used to specify identifying attributes of objects that are meaningful and relevant to users, but do not directly imply semantics to the core system.”
- “Labels enable users to map their own organizational structures onto system objects in a loosely coupled fashion.”

- Labels can be attached to objects at creation time and subsequently added and modified at any time.
- Each object can have a set of key/value labels defined. **Each Key must be unique for a given object**.

(Source: Kubernetes documentation - https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/)

Kubernetes metadata – some details about labels (2/2)

Example Labels:

- "release" : "stable″ , "release" : "canary"
- "environment" : "dev" , "environment" : "qa" , "environment" : "production"
- "tier" : "frontend” , "tier" : "backend” , "tier" : "cache"
- "partition" : "customerA” , "partition" : "customerB"
- "track" : "daily” , "track" : "weekly"

(Source: Kubernetes documentation - https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/)

Container Orchestrators – Interfaces with the host (1/3)

Several standard interfaces have been specified:

- **Container Networking Interface (CNI)**
  - “a specification and libraries for writing plugins to configure network interfaces in Linux containers, along with a number of supported plugins.”
  - https://github.com/containernetworking/cn

- **Container Storage Interface (CSI)**
  - “A specification enabling storage vendors to develop a plugin once and have it work across a number of container orchestration systems”
  - Main scope: provisioning and attaching storage volumes, managing snapshots.
  - https://github.com/container-storage-interface/spec/blob/master/spec.md

Container Orchestrators – Interfaces with the host (2/3)

Container Runtime Interface (CRI)

- Defined in the context of **Kubernetes**
- Related to the the OCI (Open Container Interface) runtime specification … but somewhat different and complementary
  - The OCI runtime specification defines the configuration, execution environment, and lifecycle of a container. In particular, it defines how to properly run a container *"filesystem bundle"* which fully adheres to the OCI Image Format Specification.
  - **The container runtime interface (CRI) is an API for container runtimes to integrate with the kubelet component on each compute/worker node.**
  - The Kubernetes CRI leverages OCI-compliant runtimes.

- **A CRI-compliant runtime has additional concerns compared to an OCI-compliant runtime.**
  - These concerns include image management and distribution, storage, networking, snapshotting, etc.
  - A CRI-compliant runtime is aimed at providing the functionality required to manage containers in a dynamic cloud environment whereas an OCI-compliant runtime focuses on a narrower goal: creating and running containers on a local machine.

Container Orchestrators – Interfaces with the host (3/3)

Container Runtime Interface (CRI) – Continued

- **Wrap-up:**
  - **The notion of CRI allows decoupling the kubelet from the underlying container runtime** (in a flexible and extensible way).
  - **A CRI-compliant runtime** (such as cri-o) **typically delegates the management of container execution to a (lower-level) OCI runtime** (such as *runc* or *crun*).

- **Examples of CRI-compliant runtimes:**
  - containerd, cri-o, kata containers (VM-based), Firecracker (VM-based)
  - Warning: Docker is not CRI compliant and deprecated (for usage on Kubernetes worker nodes). See: https://kubernetes.io/blog/2020/12/02/dont-panic-kubernetes-and-docker/

- **For more information:**
  - https://www.capitalone.com/tech/cloud/container-runtime/
  - https://kubernetes.io/blog/2016/12/container-runtime-interface-cri-in-kubernetes/
  - https://github.com/kubernetes/cri-api/blob/master/pkg/apis/runtime/v1/api.proto

Outline

- Origins and main characteristics

- Microservices

- Container orchestration

- **Design patterns**

Cloud native design patterns – Overview (1/2)

- **Many cloud native applications rely on a set on common, generic design patterns based on containers.**
- Adhering to these patterns can help:
  - Building more efficient, adaptable and robust distributed applications
  - Improving developer productivity and fostering code reuse
  - Facilitating technical discussions between architects, developers, teams by providing a common vocabulary and conceptual framework
- **Acknowledgments**:
  - This part of the lecture is heavily inspired by the following book:
    - Designing distributed systems. Patterns and paradigms for scalable, reliable services. O'Reilly, 2018.
    - Authored by Brendan Burns (co-founder of the Kubernetes project, distinguished engineer at Microsoft Azure and previously at Google Cloud Platform).
    - Provides code examples based on Kubernetes.
  - For additional concepts and vocabulary, see also:
    - https://docs.microsoft.com/en-us/azure/architecture/patterns/
    - http://www.cloudcomputingpatterns.org

Cloud native design patterns – Overview (2/2)

- We will describe several kinds of patterns:

Single-node patterns: forming the basic elements of a distributed system	Sidecar pattern
	Ambassador pattern
	Adapter pattern
Serving patterns: for long-running, request-oriented applications	Replicated, load-balanced services
	Sharded services
	Scatter/gather pattern
Batch computational patterns: for intermittent, parallel computations on an input dataset	Work queue system
	Event-driven batch processing
	Coordinated batch processing

Cloud native design patterns – Outline Overview

Single-node patterns: forming the basic elements of a distributed system	Sidecar pattern
	Ambassador pattern
	Adapter pattern
Serving patterns: for long-running, request-oriented applications	Replicated, load-balanced services
	Sharded services
	Scatter/gather pattern
Batch computational patterns: for intermittent, parallel computations on an input dataset	Work queue system
	Event-driven batch processing
	Coordinated batch processing

The sidecar pattern (1/2)

- A **single-node** pattern **based on two containers**:
  - Application container
  - Sidecar container

- **The sidecar is co-scheduled with the application container** and they can share resources (e.g., the network).
  - For example, in the case of Kubernetes, this is achieved using a Pod.

- **The sidecar is generally transparent**.
  - The application is unaware of its presence.
  - Because the application and sidecar containers interact indirectly (e.g., through the network, file system or other OS facilities), the application can be developed/built completely independently.

The sidecar pattern (2/2)

- The purpose of the sidecar is to **augment/improve the behavior of the application**.
  - This can be useful both for legacy and cloud native applications.
- **Typical use cases:**
  - **Introspection** of the application container (e.g., for resource usage monitoring or debugging)
  - **Updating the configuration** of the application containers (e.g., through shared files)
  - Introducing **network security** features (e.g., encryption and authentication)
  - **Implementing the "data plane" in a service mesh architecture** (service discovery, load balancing, tracing, etc.)
- One of the main benefits of the sidecar pattern is code reuse. However, **achieving modularity and reusability requires care about the following aspects**:
  - Parameterized containers
  - Definition of the API surface of the containers
  - Documentation of the containers (e.g., within Dockerfiles)

The ambassador pattern

- **A particular form of sidecar pattern**.
- This expression is usually dedicated to **sidecars in charge of brokering communications between application containers and the outer world**.
- **Use cases**:
  - The features handled by the control plane of a **service mesh**, which may also include the following examples.
  - Implementing a (transparent) proxy supporting **routing to a sharded service**.
  - Implementing a (transparent) proxy for **service brokering**.
    - For example, a service may rely on a database, that is provisioned differently in various cloud environments (e.g., local vs. remote, IaaS vs. PaaS)
    - The proxy handles the lookup and connection to the database service and abstracts away the differences.
  - Implementing (transparent) **request splitting** for testing purposes.

The adapter pattern

- **A particular form of sidecar pattern.**

- The adapter container is **used to modify the interface provided by the application container, so that it conforms to a specific/homogeneous interface** expected for all the applications in a given distributed system/infrastructure.

- The modifications can be related to the data format and/or the service API.

- Typical use cases:
  - **Monitoring** (e.g., collecting performance metrics)
  - Advanced **health checks**
  - **Logging**

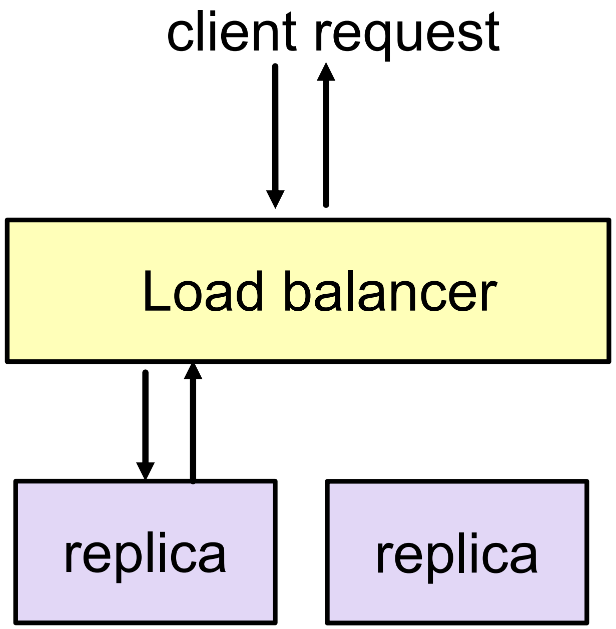
Cloud native design patterns – Outline Overview

Single-node patterns: forming the basic elements of a distributed system	Sidecar pattern
	Ambassador pattern
	Adapter pattern
Serving patterns: for long-running, request-oriented applications	Replicated, load-balanced services
	Sharded services
	Scatter/gather pattern
Batch computational patterns: for intermittent, parallel computations on an input dataset	Work queue system
	Event-driven batch processing
	Coordinated batch processing

Replicated load-balanced services (1/3)

- **Principle:**
  - **Horizontal scaling using several instances ("replicas") of the same service,** in order to improve performance and reliability.
  - **The load balancer is the service entry point and routes a given request to a given replica.**

- The load balancer can be implemented:
  - either as an ambassador (embedded within every client)
  - or as an explicit, standalone service.



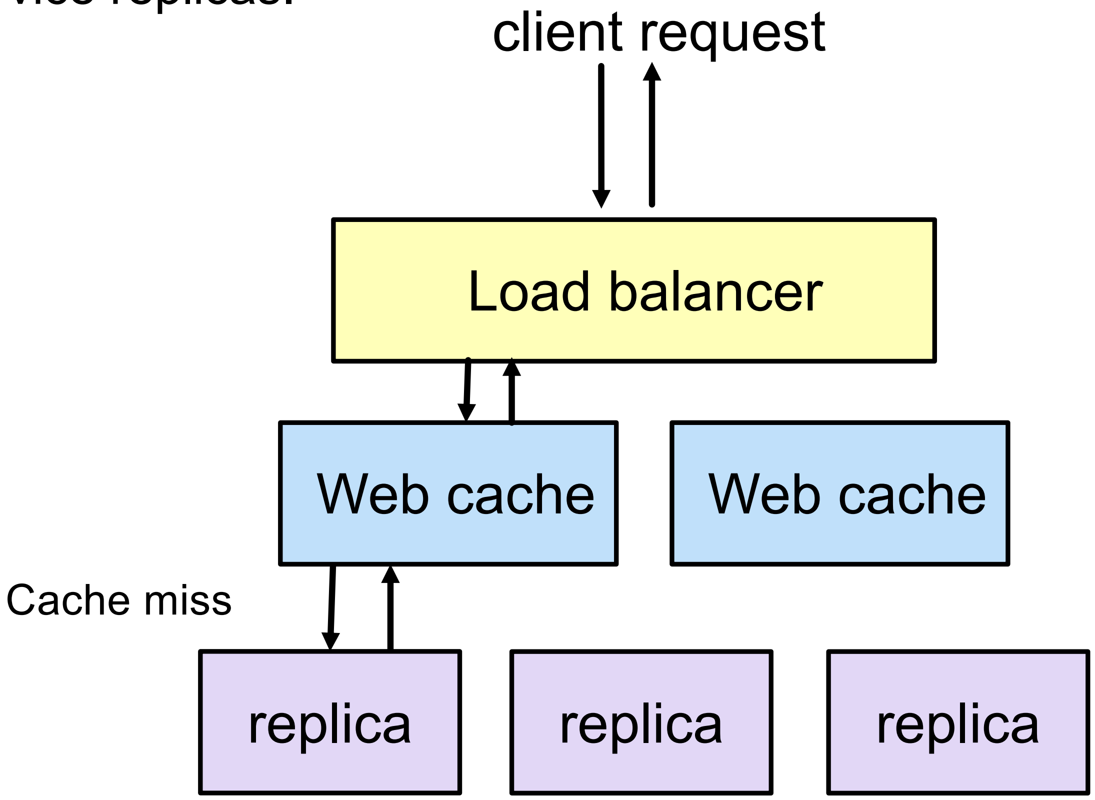
Replicated load-balanced services (2/3)

- The load balancer can use **readiness probes** to help the determine when a new replica is ready to server requests.
  - The load balancer can query a new replica through a specific HTTP URL to check if it is ready.
  - The readiness probe can be implemented as a sidecar of the service.

- **For stateful services**, the load balancer can use a routing algorithm that preserves **"session stickiness"**, so that the requests from the same client are routed to the same replica.

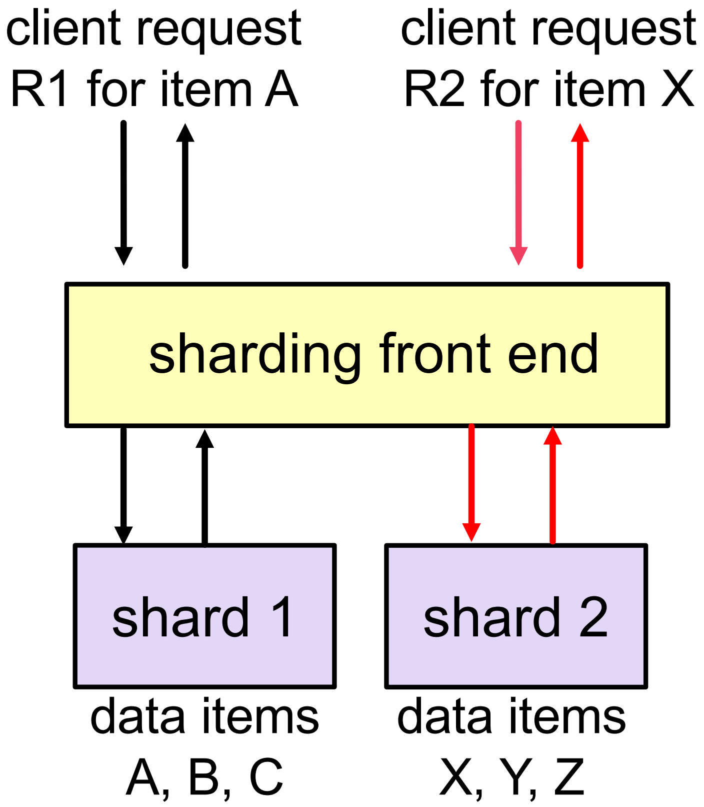
Replicated load-balanced services (3/3)

- In the case of application requests with read-only semantics and implemented with HTTP, **performance can be improved by introducing a Web caching layer** between the load balancer and the service replicas.



Sharded services (1/2)

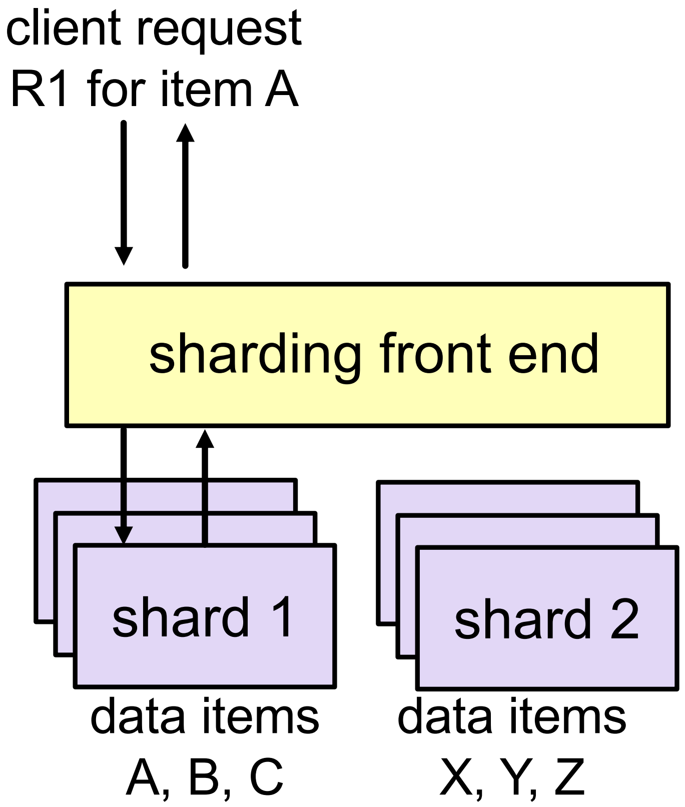
- Also known as “partitioned services”.
- **In contrast to a replicated service,** in the case of a sharded service, **each service instance (also named “shard”) is only capable of serving some of the requests.**
- The frontend examines each request and routes it towards the appropriate shard, based on a sharding function applied to a request key.
- This pattern is useful for stateful services in which the size of the overall state (data) is too big to fit on a single machine.
- **The sharding function can rely on either:**
  - On **key-range partitioning** (based on the value of the original request key)
  - Or **hashed-based partitioning**, in which each partition owns a range of hashes. Typically, a technique named “consistent hashing” is used.



Sharded services (2/2)

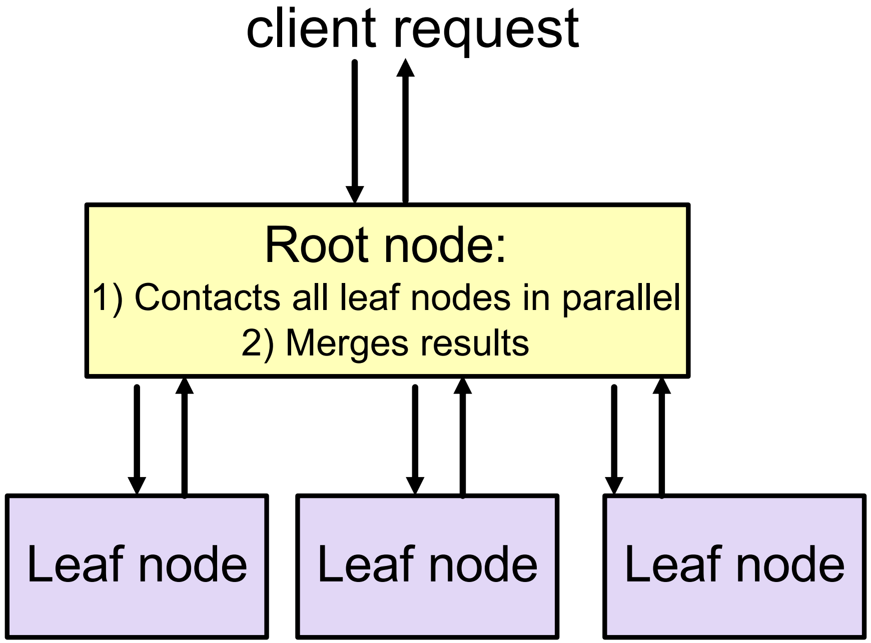
- **This sharding pattern can also be combined with the previous pattern (replication),** in order to support high availability.

- In order to address scaling (up and down) and load imbalance, a sharded service **must generally support dynamic shard rebalancing**.



The scatter/gather pattern (1/2)

- Another tree-based distributed pattern.
- **Unlike the basic replication or sharding patterns, each service instance (often called "leaf node") is involved in the processing of each request.**
  - Each instance performs a small amount of processing and returns a result to the root instance.
  - **The root instance combines all the partial results received from the instances**, in order to build a complete result, which is sent back to the client.
- This pattern:
  - Can work with shards or identical replicas
  - Is **useful for work that can be easily parallelized, i.e., when the different partitions can work mostly independently**


client request

Root node:
1) Contacts all leaf nodes in parallel
2) Merges results

Leaf node

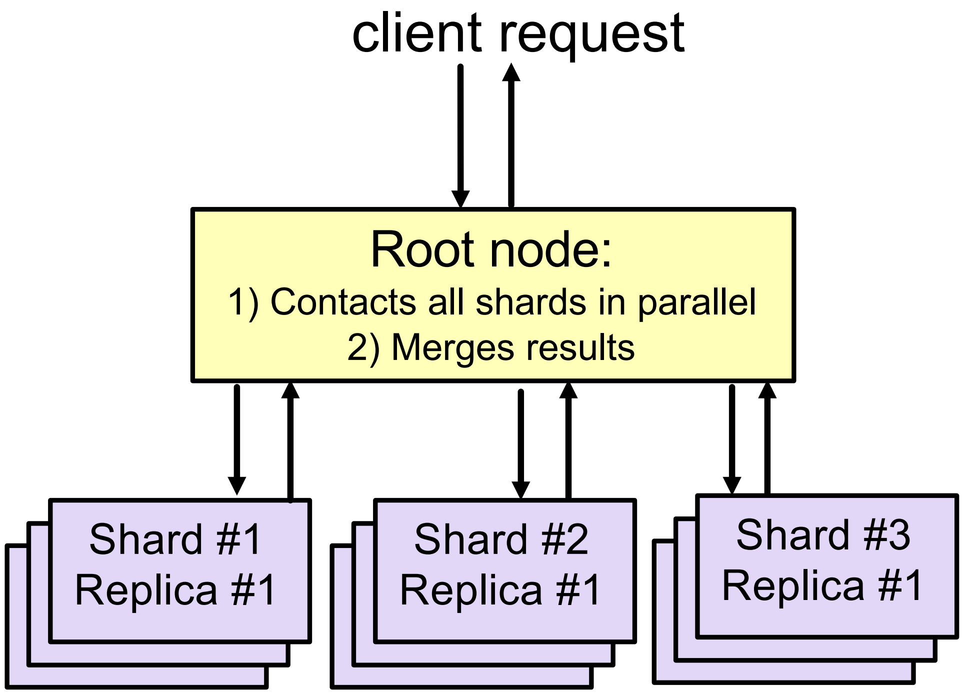
Leaf node

Leaf node

The scatter/gather pattern (2/2)

- **In order to improve performance, this pattern is usually augmented into a shared and replicated scatter/gather design.**
  - Each leaf node (shard) is replicated.
  - Each request is sent to one replica per shard.

- **Using too many leaf nodes can hurt performance**:
  - Parallelization overheads increase with the number of nodes.
  - Performance is limited by the slowest leaf node ("straggler effect").

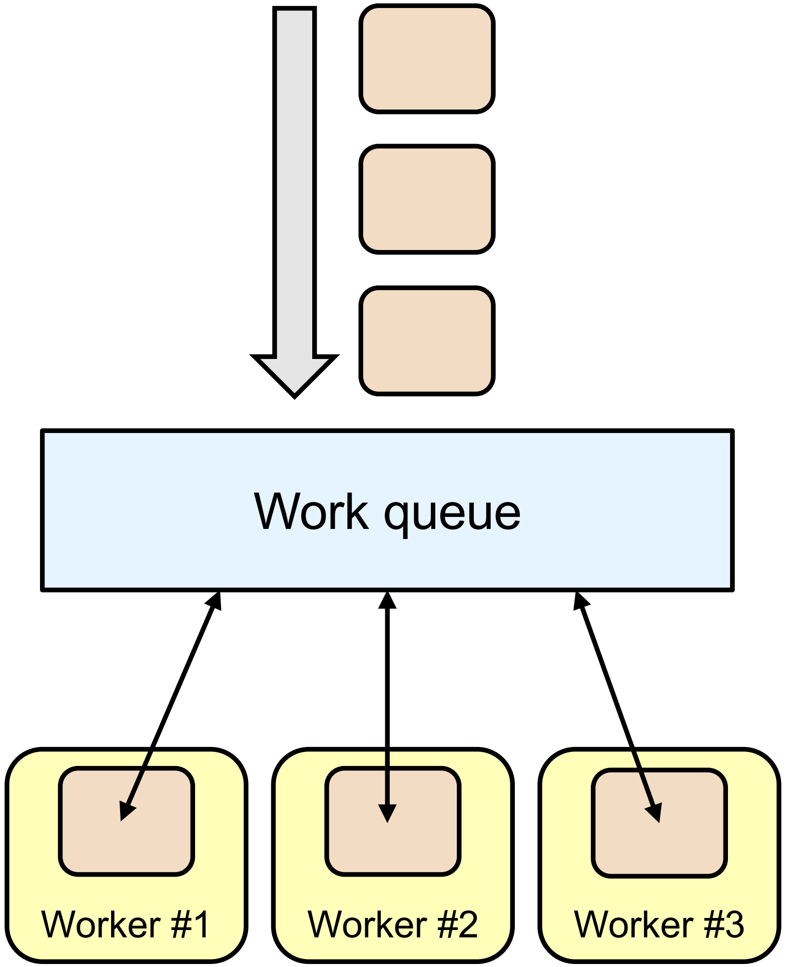


Cloud native design patterns – Outline Overview

Single-node patterns: forming the basic elements of a distributed system	Sidecar pattern
	Ambassador pattern
	Adapter pattern
Serving patterns: for long-running, request-oriented applications	Replicated, load-balanced services
	Sharded services
	Scatter/gather pattern
Batch computational patterns: for intermittent, parallel computations on an input dataset	Work queue system
	Event-driven batch processing
	Coordinated batch processing

Work queue systems (1/5)

- A simple form of batch processing system.
- **The work queue is used by clients to submit work items, to be processed by a set of workers.**
- Each work item corresponds to an independent piece of work.
- Each worker processes one item at a time.
- The set of workers can be scaled up or down according to the amount of pending work.



Work queue systems (2/5)

- Most of the queue management logic is independent from the actual type of work to be performed.
- Similarly, most of the interactions with the queue are independent from the internal implementation of the queue.
- Thus, it is easy to decouple application-specific and generic code logics, by using distinct containers. The generic logic can be put into ambassador containers.

- Main interfaces:
  - **Work submission** (between clients and the work queue manager)
  - **Work queue submission** (between the work queue manager and the actual work queue implementation)
  - **Worker interface** (between the actual work queue implementation and the workers)
    - Assigning work
    - Signaling work completion

Work queue systems (3/5)

Fault tolerance

- Worker processes or machines may fail.
- A long-running work queue system must take into account such failures.
- Requires a mechanism to identify a failed worker and “resubmit” the work item (without involving the initial client).

- This problem is simplified if the processing of the work items is idempotent.
  - If so, there is no need to “undo” the partial side effects of the failed worker. It is only necessary to reassign the work item to another worker.

Work queue systems (4/5)

Work queue implementation

Different strategies are possible for the concrete implementation of the work queue pattern. For example:

- **Leveraging the job management features of a container orchestrator**
  - The work queue is actually the job queue of the orchestrator.
  - The orchestrator selects a worker node and creates a new container for each work item.
  - The orchestrator is in charge of monitoring failed nodes/containers and creating another instance.
- **Using a “publish-subscribe system**
  - The work queue is actually a publish-subscribe topic, configured in such a way that each message is only fetched by one consumer.
  - A set of worker processes are created on a set of worker nodes.
  - Each worker fetches a work item from the topic, processes it, acknowledges it and starts over with a new item.
  - If a worker has not acknowledged a given message within a certain time interval, the worker is assumed to have failed and the message is automatically reinserted in the topic by the publish-subscribe infrastructure.
  - A container orchestrator can be used to manage the worker nodes/processes. But in this design, the orchestrator is not involved in the assignment of the work items.

Work queue systems (5/5)

Dynamic scaling of the workers

- The set of workers can be adjusted according to the amount of pending work, which may vary over time.
- **A possible autoscaling design**:
  - Monitoring two **metrics**:
    - Interarrival time: average time (measured over a long period of observation) between the arrival of consecutive work items in the queue.
    - Average processing time for the work items (not counting the time spent waiting in the queue).
  - **Upscaling**: to guarantee that (average processing time / number of parallel workers) is lower than the average interarrival time
    - Increase number of workers and/or use faster machines
  - **Downscaling**: same principle in reverse direction if (average processing time / number of parallel workers) is much lower than the average interarrival time

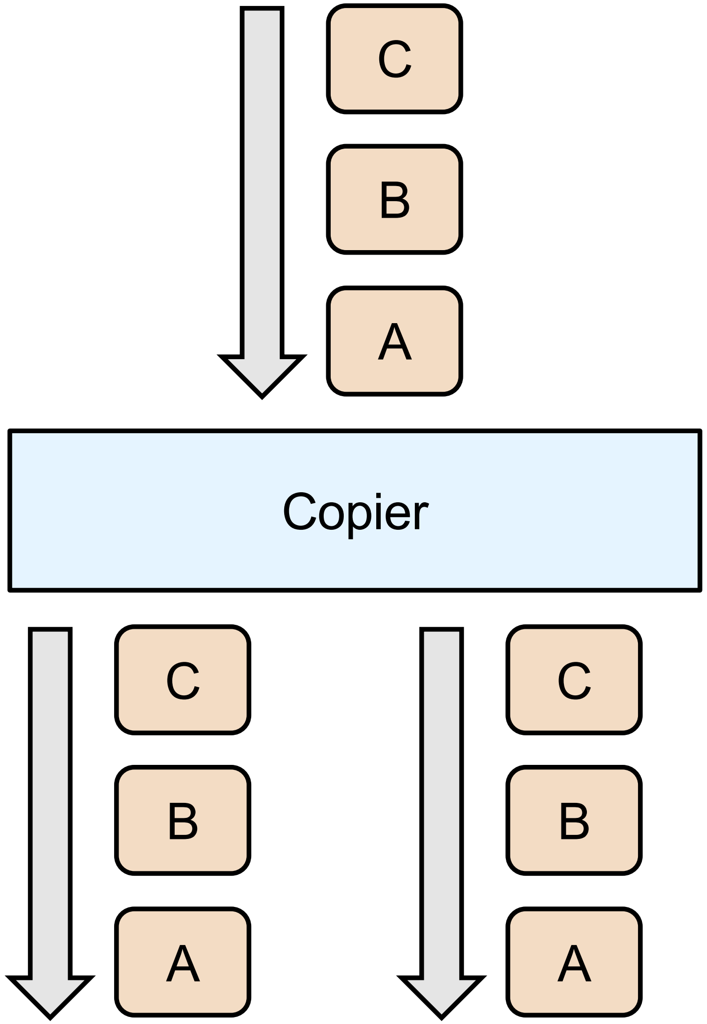
Event-driven batch processing (1/8)

- **Also known as "workflow system".**
- Main principles:
  - Scenario in which a given event/input requires more than a single action. This could be, for example:
    - The need to perform a sequence of actions
    - The need to produce multiple outputs from the same input.
  - **In such a scenario, we need to link multiple work queue together to form a directed acyclic graph (DAG) of processing stages.**
- **We will describe several architectural patterns for workflows:**
  - Basic chain/sequence of work queues: simple generalization of the previous pattern
  - Copier
  - Filter
  - Splitter
  - Sharder
  - Merger

Event-driven batch processing (2/8)

The “Copier” pattern

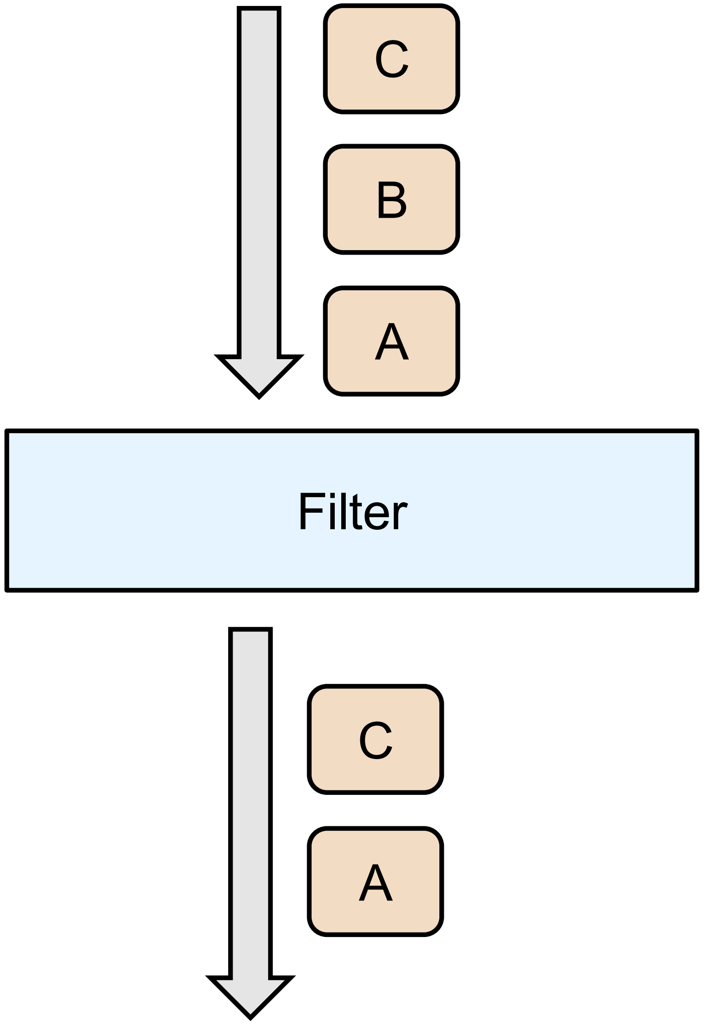
- **Input**: a single stream of work items
- **Output**: **two or more duplicated streams** of work items (each one is an identical copy of the input stream)
- Useful when there are multiple and independent processing tasks to be performed on the same items.
- **Example**: transcoding a raw video stream into several target formats.
- **Note**: work items are not necessarily costly to duplicate because they do not have embed a copy of the data items (they may contain only references to the data being stored elsewhere).



Event-driven batch processing (3/8)

The “Filter” pattern

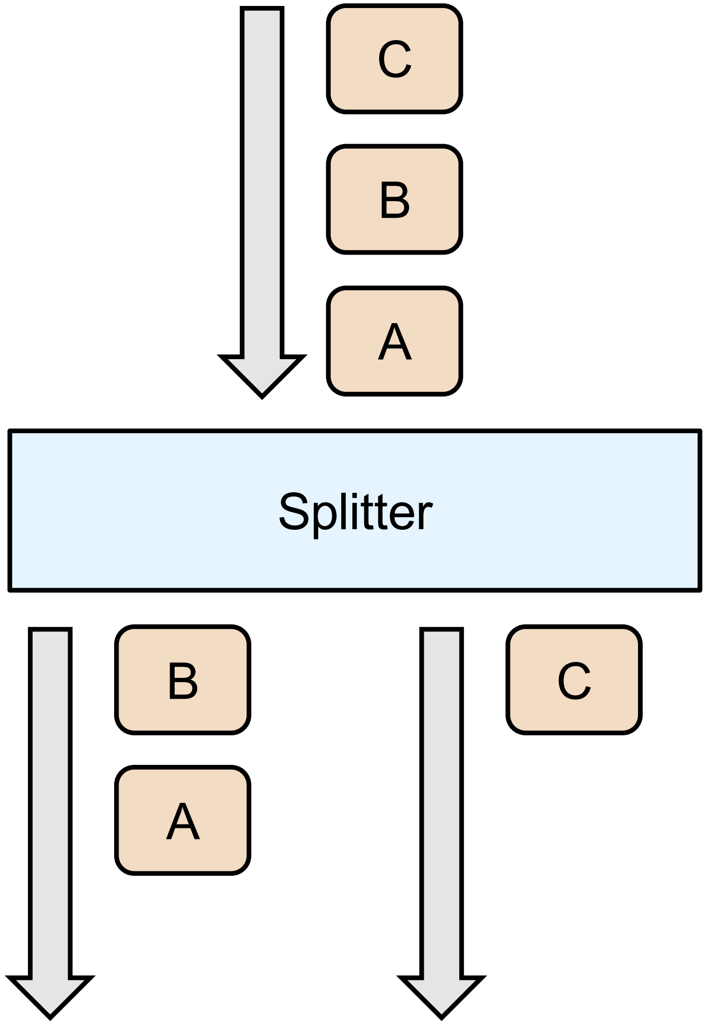
- **Input**: a single stream of work items
- **Output**: a single stream of work items, in which, some of the original items have been removed, according to a particular criterion.
- Useful to perform conditional work depending on client-defined settings
- **Example**: filter out work items according to the type of attached input file
- **Note**: can be implemented using an ambassador container in front of a regular work queue



Event-driven batch processing (4/8)

The "Splitter" pattern

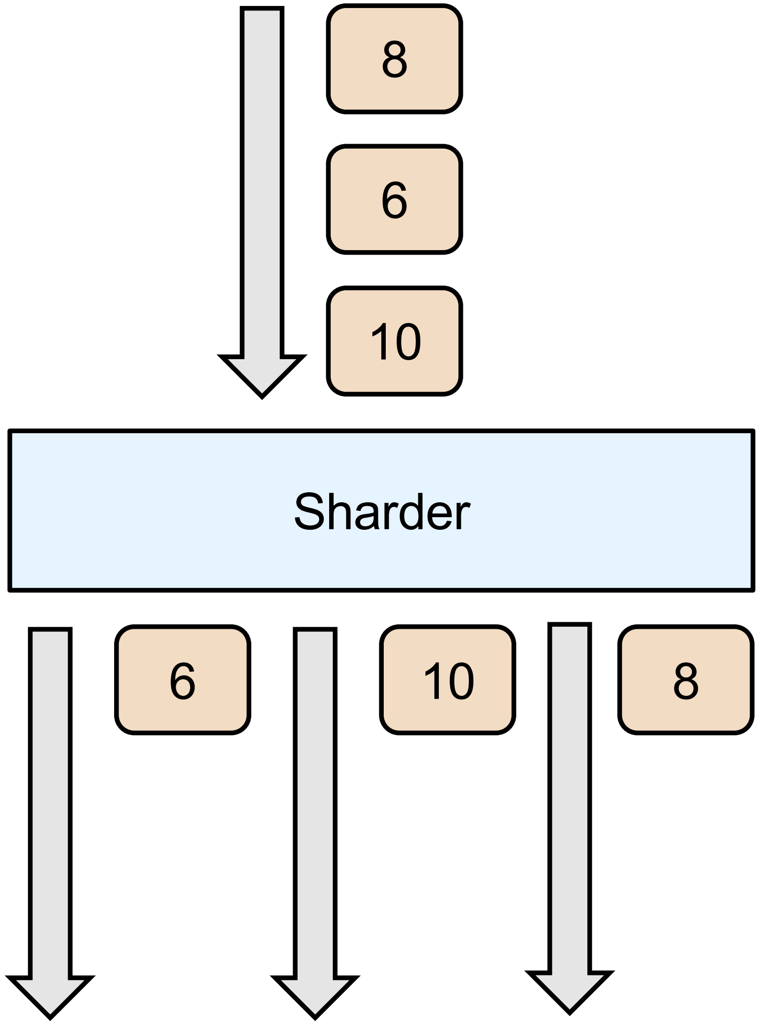
- **Input**: a single stream of work items
- **Output**: two streams of work items. **Each input item is forwarded to only** one of the output streams according to a particular criterion.
- Useful to fork a processing pipeline into two distinct parts.
- **Example**: differentiate processing steps for orders based on the type of notification requested by users (email vs. text).
- **Note**: a variant of this pattern can combine the "copier" and "splitter" behaviors (e.g., for users who choose both email and text notifications).



Event-driven batch processing (5/8)

The “Sharder” pattern

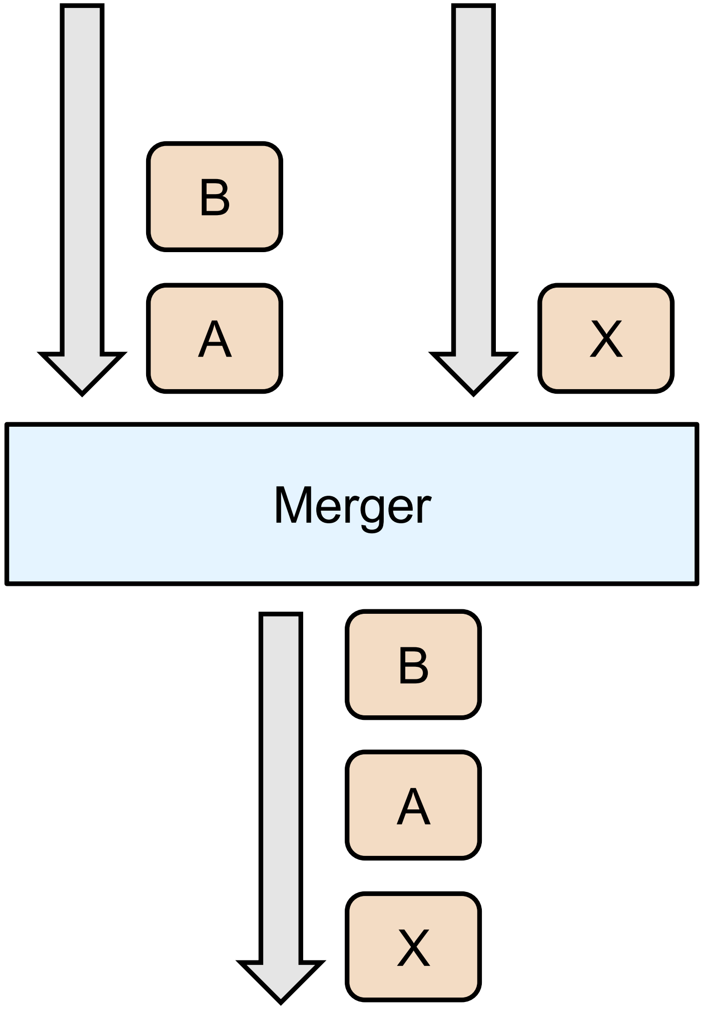
- **A variant of the splitter pattern**
- **Input**: a single stream of work items
- **Output**: multiple output streams of work items. **Each input item is forwarded to only** one of the output streams **according to a sharding function**.
- The sharding function may dynamically take into account the unavailabity of some output queues.
- Useful:
  - For large-scale, hierarchical/distributed processing & load balancing
  - For improved reliability via partial availability (e.g., if the shards are placed on different data centers / regions)



Event-driven batch processing (6/8)

The “Merger” pattern

- **The opposite of a “copier” pattern**
- **Input**: multiple input streams of work items
- **Output**: a single output stream of work items, which is the union of the input streams
- Useful for:
  - “Reconciling” different processing pipelines that share the same finishing steps (e.g., sharded pipelines)
  - Factoring server resources for workflows sharing common processing stages



Event-driven batch processing (7/8)

How to implement the previously described patterns?

- **Inadequate solutions:**
  - Using a **“traditional” distributed storage system** (e.g., file system, database, object storage) for such a purpose is often impractical for developers and can also raise efficiency and/or robustness concerns.
  - Similarly, using **raw networks streams (e.g., TCP/IP sockets)** is often cumbersome and fragile.

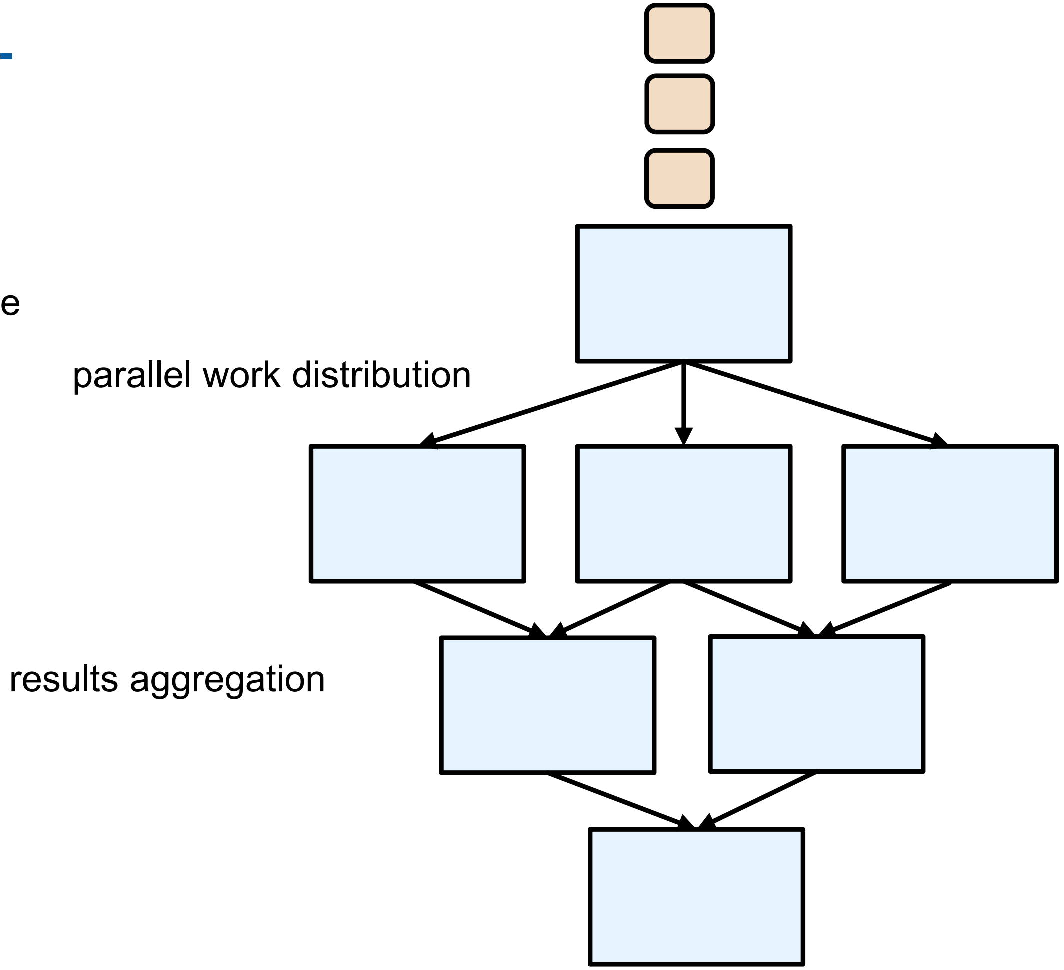
Event-driven batch processing (8/8)

How to implement the previously described patterns? (continued)

- **One of the most appropriate building blocks is a “publish-subscribe” system**
  - Also known as a “messaging bus” or “message broker”.
  - **Each input or output stream of work items and results is mapped to a distinct “queue” (a.k.a. “topic”).**
  - Each queue can receive items produced by one or more publishers.
  - Each queue can be consumed by one or more subscribers.
  - **Each pattern instance:**
    - **Acts as a subscriber for one or several queues**
    - **Acts as a publisher for one or several queues**

Coordinated batch processing (1/3)

- **A specific set of patterns for parallel, multi-step computations.**
- Based on **two main phases**:
  - **Phase 1: Parallel work distribution**
  - **Phase 2: “Aggregation”** of the results to generate a global output
- Such patterns are used (with various incarnations) in many different domains, such as:
  - Scientific computing
  - Big data processing (e.g., the MapReduce programming model)
- Phase 1 is related to the “splitter/sharder” patterns described previously.
- **We will mainly focus on phase 2**, with two major patterns: “Join” and “Reduce”



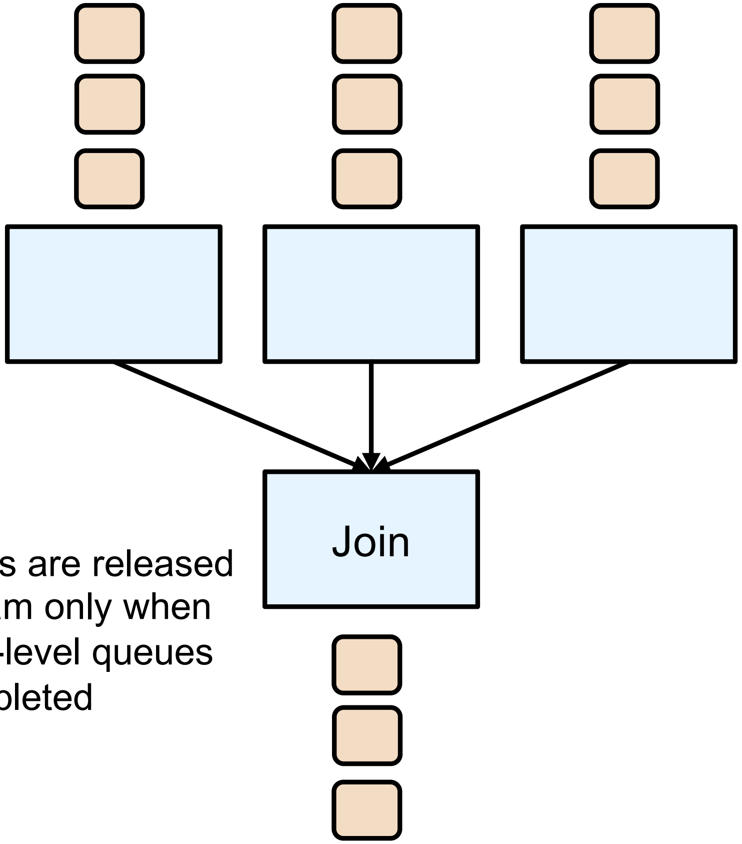
parallel work distribution

results aggregation

Coordinated batch processing (2/3)

“Join” pattern

- Also known as **“barrier synchronization”**.
- **Ensures that a given set of work items have been processed before moving on to the next stage of a workflow.**
- **Warning: not the same thing as the previous “Merge” pattern**
  - “Merge” merely combine multiple streams into one
  - Instead, “Join” ensures that all the processing streams have completed.
- Typical usage: **ensures that no data/output is missing before some sort of aggregation is performed**.
- Decreases the intrinsic parallelism within a workflow, and thus degrades the overall completion time.
- May be necessary to ensure the correctness of some parallel algorithms.



Work items are released downstream only when all the top-level queues have completed

Coordinated batch processing (3/3)

“Reduce” pattern

- **Unlike the “join” pattern:**
  - Does not require to wait until all the items have been processed. (More parallelism)
  - Progressively/optimistically merges together all the available results.
- **Each step of the reduce phase merges several items into a single one** (hence the name: it “reduces” the number of output items).
- **The reduce phase can be repeated as many (or as few) times as necessary in order to obtain a single output for the entire data set.**
- Some examples of concrete “reduce” operations:
  - Count: counting the number of occurrences of each value (e.g., word count)
  - Sum: summation of a collection of different values