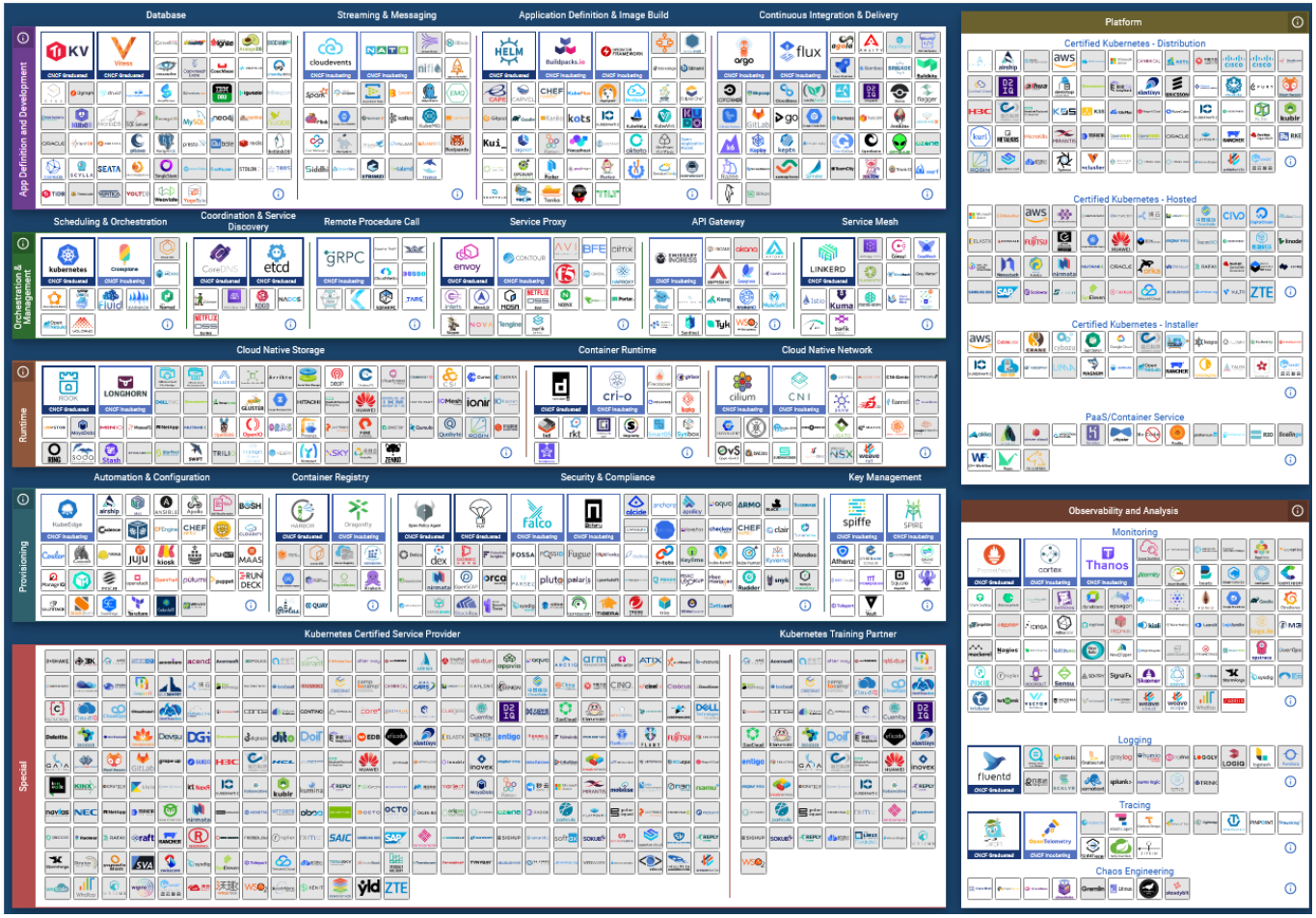
Lecture 13: Containerization and Serverless Computing

CSCI4180

Patrick P. C. Lee

Cloud-Native Community



Cloud native community: https://landscape.cncf.io/

Cloud-Native

- Simple definition:
  - *“Cloud-native architecture and technologies are an approach to designing, constructing, and operating workloads that are built in the cloud and take full advantage of the cloud computing model.”*
- Official definition from CNCF:
  - *“Cloud native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach.”*
- Cloud native is about *speed* and *agility*

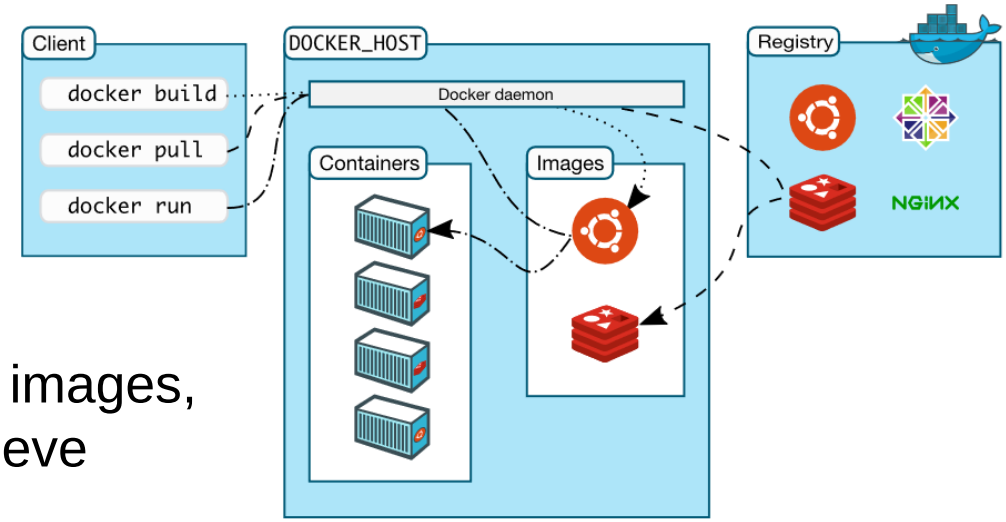
Reference: https://docs.microsoft.com/en-us/dotnet/architecture/cloud-native/definition

Containers

- Containers provide more lightweight resource isolation than VMs → **micro-service** compute model
  - Google reportedly runs all applications in containers
- Containers package an application and all its dependencies in **images** that can be stored and shared
- Container management frameworks:
  - Dockers, CoreOS, Kubernetes

Docker Architecture

- Client-server architecture



- Docker daemon:
  - Start/stop containers, create images, store images in registry, retrieve images from registry
- Docket client:
  - Communicate with the daemon
  - Client and daemon often run on the same host
- Docket registry
  - Container image repository

Reference: https://docs.docker.com/engine/docker-overview/

Docker: Images and Layers

- A container **image** contains all necessary files to run the container and has multiple read-only **layers**; each layer is a subset of files under the container file-system tree

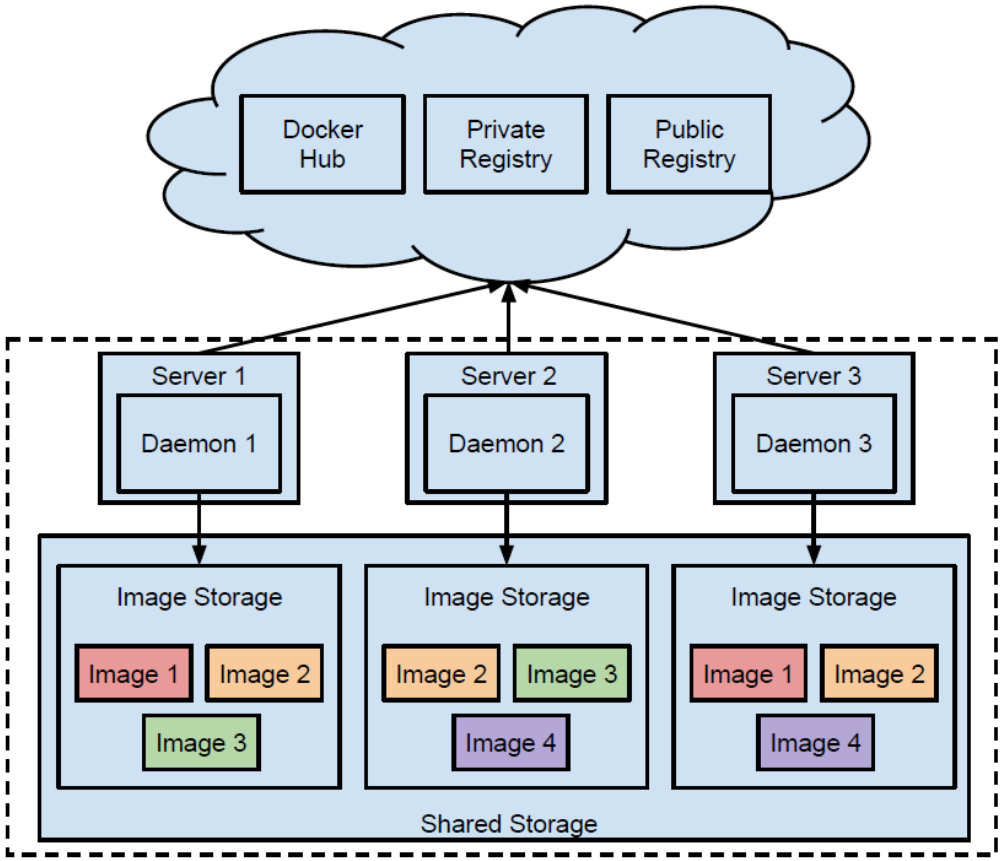
- Once the image is retrieved, the daemon creates the container and stores changes in a **writable layer** via the copy-on-write (COW) mechanism
  - Before modifying a file from a read-only layer, it is copied to the writable layer and all changes are made to the copy
  - Once the container exits, the writable layer is typically discarded

- **Graph drivers**: provide mappings to storage
  - Overlay drivers and specialized drivers

Docker's Limitations

- Each daemon uses its exclusive local storage for data and metadata

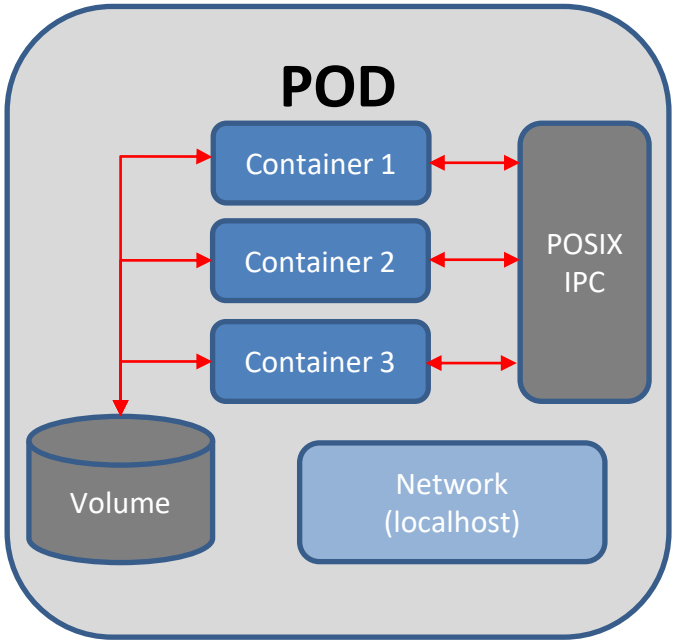
- Limitations:
  - Redundant pulls (high network overhead)
  - Over-used storage space
  - High startup latency over distributed storage



Kubernetes

- Originally designed by Google
- Maintained by the Cloud Native Computing Foundation (https://www.cncf.io/)
- OpenSource Written in Go (https://github.com/kubernetes/kubernetes)

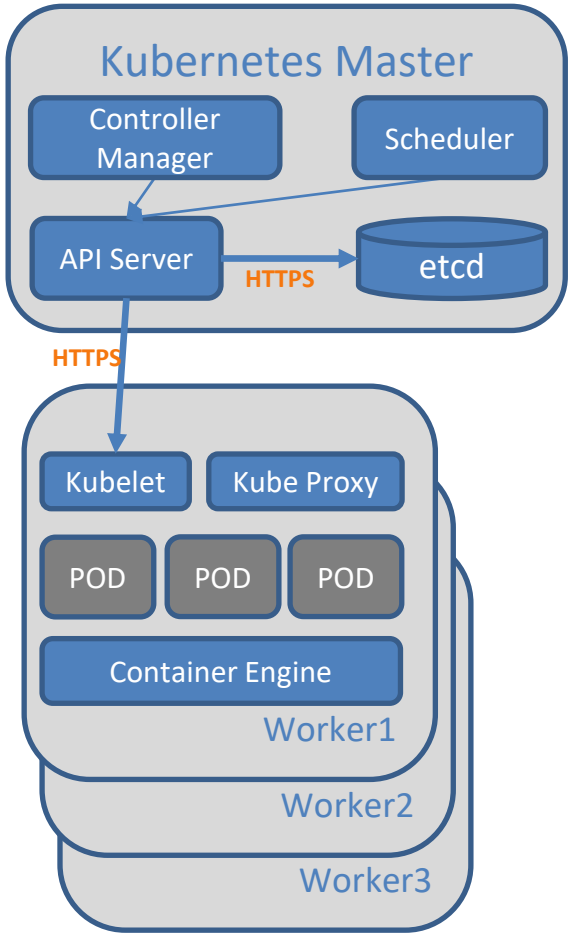
Kubernetes: PODs



- Grouping of containers with common purpose
- All containers in a POD need to be tightly dependent on each other
- Smallest unit that Kubernetes can deploy
- Set of metadata (name, labels) for the POD
- Shared Volume (persistent for container only, not for POD)
- Inter-Process communication (POSIX queues, shared memories)
- Inter-container network communication

https://gonorthforge.com/wp-content/uploads/2019/10/NII_MKT-Intro-to-Kubernetes-v1.0.pptx

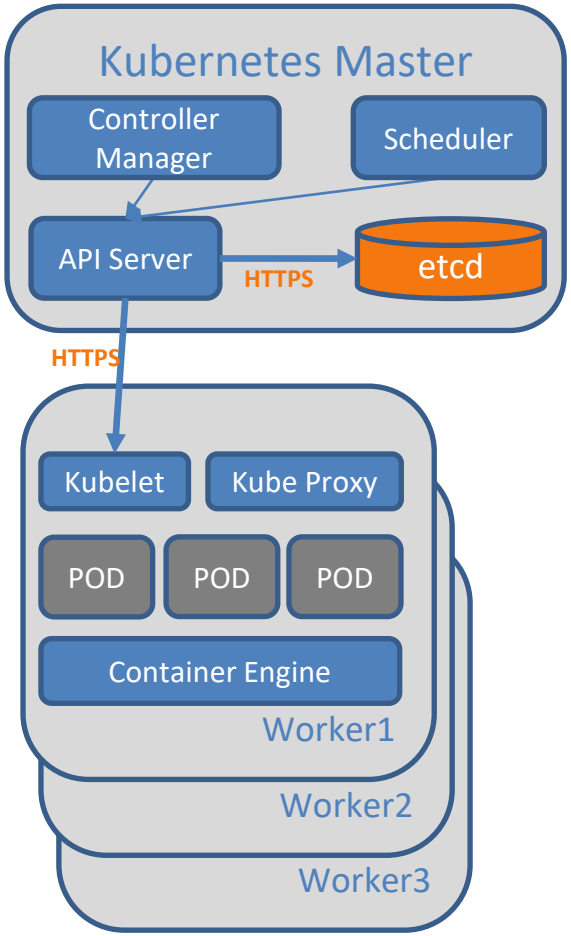
Kubernetes Cluster



- Master Components
  - Cluster's control plane
  - These components make global decisions about the cluster, detecting and responding to cluster events
- Worker Components
  - Run on every node
  - Maintaining running pods and providing the runtime environment

https://gonorthforge.com/wp-content/uploads/2019/10/NII_MKT-Intro-to-Kubernetes-v1.0.pptx

Kubernetes Cluster



- etcd is a distributed and consistent key-value store
- The only storage backend currently supported by Kubernetes
- Primary store for all Kubernetes API objects and their configuration
- The etcd database also stores the actual state of the system and the desired state of the system.
- Etcd has a watch functionality to monitor any changes. It monitors if actual and desired states diverge, Kubernetes will make the appropriate changes to the system.
- For a demo of etcd: http://play.etcd.io/

https://gonorthforge.com/wp-content/uploads/2019/10/NII_MKT-Intro-to-Kubernetes-v1.0.pptx

Kubernetes Cluster

More on master components:

- API server:
  - Entry point to the system
- Controller manager:
  - Watches the state of the cluster
- Scheduler:
  - Schedules pods to nodes

Serverless Computing

- Serverless computing is a computing model that enables users to simply write the code (called functions) and leaves all the server provisioning and administration tasks to the cloud provider

- Serverless computing = FaaS + BaaS
  - FaaS (Function as a Service): offerings of cloud functions
  - BaaS (Backend as a Service): offerings of serverless frameworks for specific application requirements

- Examples of deployment: AWS Lambda, Google Cloud Functions, and Azure Functions

Reference: “Cloud Programming Simplified: A Berkeley View on Serverless Computing”, Technical Report, UCB, 2019

Serverless Computing

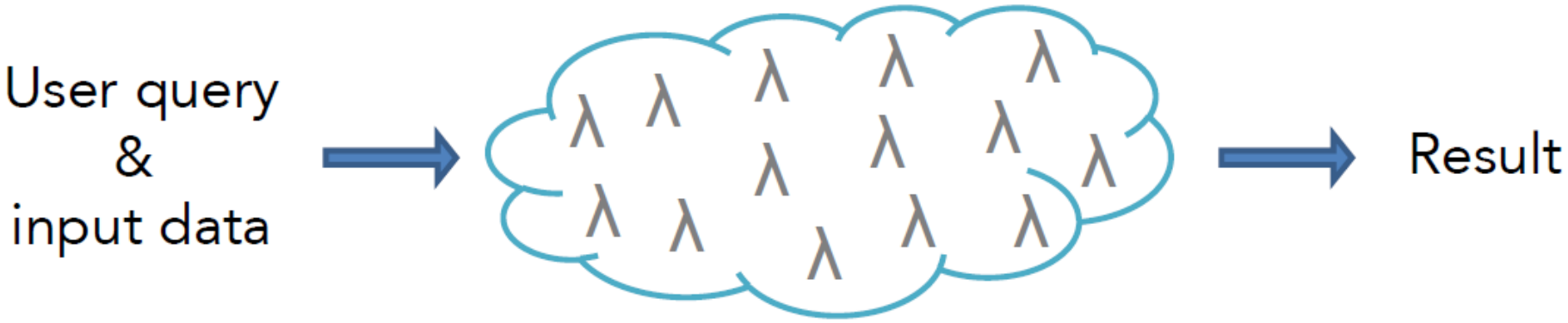
	Characteristic	*AWS Serverless Cloud*	*AWS Serverful Cloud*
PROGRAMMER	When the program is run	On event selected by Cloud user	Continuously until explicitly stopped
	Programming Language	JavaScript, Python, Java, Go, C#, etc.[4]	Any
	Program State	Kept in storage (stateless)	Anywhere (stateful or stateless)
	Maximum Memory Size	0.125 - 3 GiB (Cloud user selects)	0.5 - 1952 GiB (Cloud user selects)
	Maximum Local Storage	0.5 GiB	0 - 3600 GiB (Cloud user selects)
	Maximum Run Time	900 seconds	None
	Minimum Accounting Unit	0.1 seconds	60 seconds
	Price per Accounting Unit	$0.0000002 (assuming 0.125 GiB)	$0.0000867 - $0.4080000
	Operating System & Libraries	Cloud provider selects[5]	Cloud user selects
SYSADMIN	Server Instance	Cloud provider selects	Cloud user selects
	Scaling[6]	Cloud provider responsible	Cloud user responsible
	Deployment	Cloud provider responsible	Cloud user responsible
	Fault Tolerance	Cloud provider responsible	Cloud user responsible
	Monitoring	Cloud provider responsible	Cloud user responsible
	Logging	Cloud provider responsible	Cloud user responsible

Reference: “Cloud Programming Simplified: A Berkeley View on Serverless Computing”, Technical Report, UCB, 2019

Serverless Computing

- Enables users to launch short-lived tasks with **high elasticity** and **fine-grained resource billing**

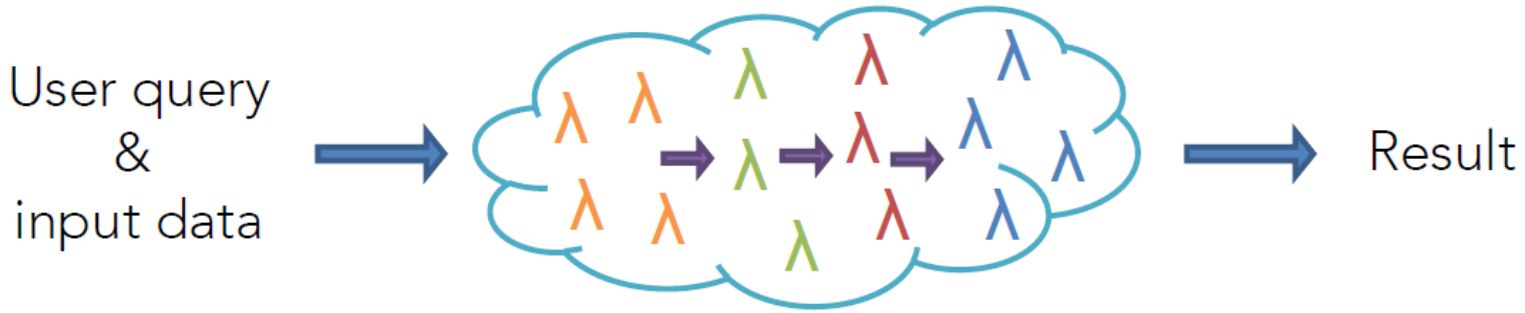
- Increasingly used for interactive analytics
  - Exploit massive parallelism with large number of serverless tasks
  - Use cases: video encoding, distributed software compilation, MapReduce (Corral)



Reference: “Pocket: Elastic Ephemeral Storage for Serverless Analytics”, OSDI’18

Challenge: Data Sharing

- Analytics jobs involve multiple stages of execution
- Serverless tasks need an efficient way to communicate intermediate data (**ephemeral data**) between different stages of execution



- Direct communication among serverless tasks is difficult
  - Object sizes vary a lot
  - Tasks are short-lived and stateless

Data Sharing

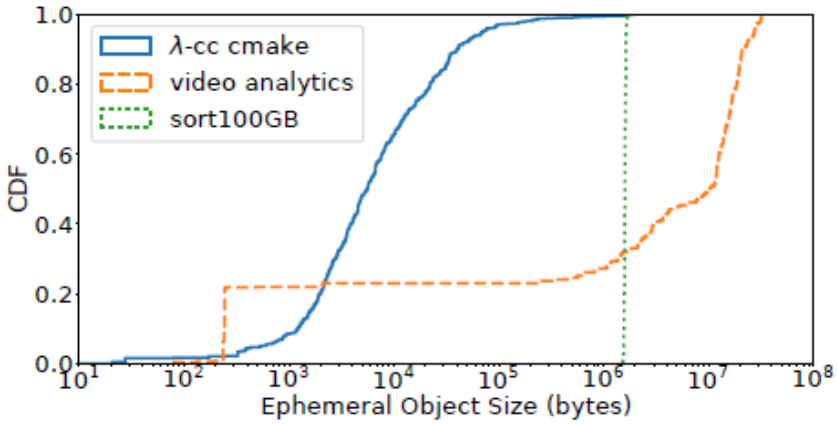


Figure 2: Objects are 100s of bytes to 100s of MBs.

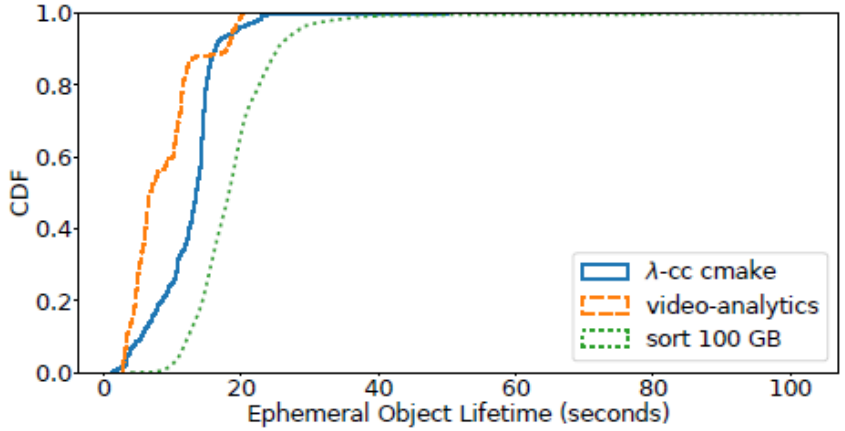


Figure 3: Objects have short lifetime.

Data Sharing

- The natural approach for sharing ephemeral data is through a **common data store**
- Examples:
  - Object stores (e.g., Amazon S3)
  - Databases (e.g., CouchDB)
  - Distributed caches (e.g., Redis)
- Performance-cost trade-off:

	Elastic scaling	Latency	Throughput	Max object size	Cost
S3	Auto, coarse-grain	High	Medium	5 TB	$
DynamoDB	Auto, fine-grain, pay per hour	Medium	Low	400 KB	$$
Elasticache Redis	Manual	Low	High	512 MB	$$$
Aerospike	Manual	Low	High	1 MB	$$
Apache Crail	Manual	Low	High	any size	$$
Desired for λs	*Auto, fine-grain, pay per second*	*Low*	*High*	*any size*	*$*

Requirements of Ephemeral Storage

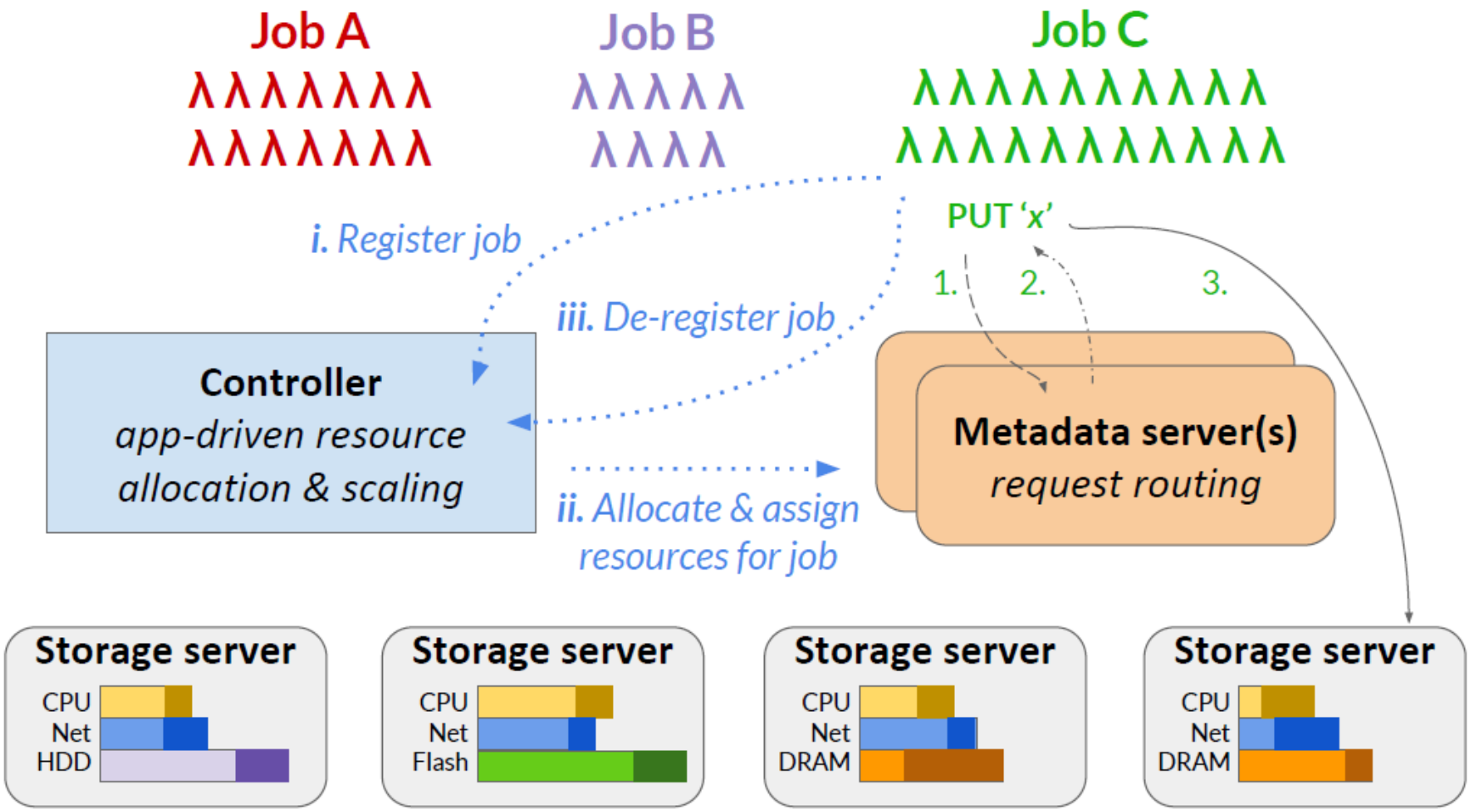
- Existing cloud storage systems are not designed for ephemeral storage
  - e.g., S3 is designed for long-term durable storage
- Requirements:
  - High-performance for a wide range of object sizes
  - Cost efficiency, i.e., fine-grained, pay-what-you-use resource billing
  - Fault tolerance is **not** critical
    - Why?
- Important to meet elasticity, performance, and cost demands of serverless analytics jobs

Pocket

- An elastic, distributed data store for ephemeral data sharing in serverless analytics
- Pocket achieves high performance and cost efficiency by:
  - Leveraging multiple storage technologies
  - Rightsizing resource allocations for applications
  - Autoscaling storage resources in the cluster based on usage
- Pocket achieves similar performance to Redis, an in-memory key value store, while saving ~60% in cost for various serverless analytics jobs

Reference: “Pocket: Elastic Ephemeral Storage for Serverless Analytics”, OSDI’18

Pocket Architecture



Implementation

- Pocket's metadata and storage server implementation is based on the **Apache Crail** distributed storage system
- Use **ReFlex** for the Flash storage tier
- Pocket runs the storage and metadata servers in containers, orchestrated using **Kubernetes**

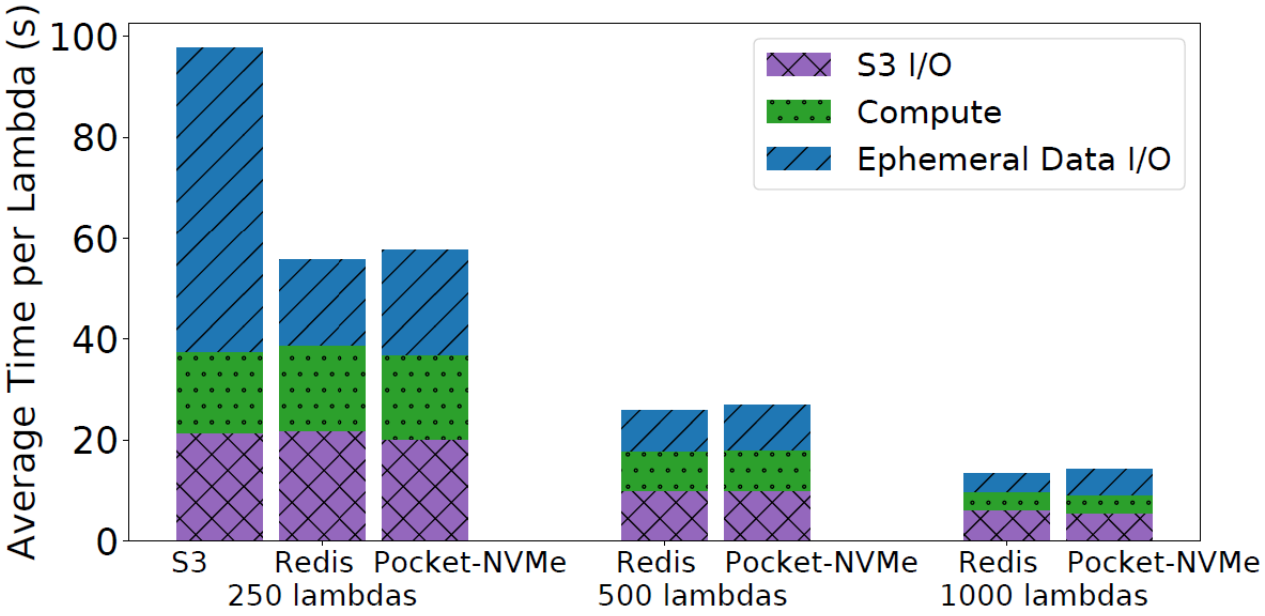
Evaluation

- Deploy Pocket on Amazon EC2

Pocket server	EC2 server	DRAM (GB)	Storage (TB)	Network (Gb/s)	$ / hr
Controller	m5.xl	16	0	~8	0.192
Metadata	m5.xl	16	0	~8	0.192
DRAM	r4.2xl	61	0	~8	0.532
NVMe	i3.2xl	61	1.9	~8	0.624
SSD	i2.2xl	61	1.6	$\lesssim 2$	1.705[1]
HDD	h1.2xl	32	2	~8	0.468

- We use AWS Lambda as our serverless platform
- Applications: MapReduce sort, video analytics, distributed compilation

Results



Execution time breakdown of 100GB sort

- S3 does not provide sufficient throughput
- Pocket provides similar throughput to Redis, yet MapReduce sort is insensitive to latency, so Pocket can use NVMe Flash instead of DRAM to reduce cost