Practical Lab

Cloud Systems Engineering

(cloud-lab)

Chair of Decentralized Systems Engineering
https://dse.in.tum.de/

Welcome to the cloud lab!

Cloud computing

- Cloud computing is powering the Internet
  - Large-scale computing resources
  - On-demand and cost effective
  - Geo-distributed data centers

Motivation: Cloud systems engineering

- Cloud systems
  - Modern cloud systems handle millions of users and TBs of data
  - Cloud software systems employ large geo-distributed data centers
- How can we build cloud systems that …
  - … scale seamlessly?
  - … are highly available?
  - … are fault tolerant?
  - … are easily configurable?
  - … are easily maintained?
- Cloud systems aim to achieve all the above in a cost-effective manner

Our focus: Learning goals

- Part I: Cloud systems workflow
  - Container: How to build applications using containers?
  - Cluster orchestrators: How to deploy jobs?
- Part II: Distributed systems system architecture
  - Sharding / re-configuration of servers
  - Fault tolerance / replication
  - Consistent hashing
  - Consistency
  - Transactions / data management
  - Distributed locking / synchronization
  - Concurrency and high-performance architectures
  - Fault detection
  - Configuration management

Learn by building an end-to-end system!

Technologies

kubernetes

APACHE ZooKeeper™

RocksDB

Format

- A set of four programming tasks:
  - Each related to a different aspect of distributed systems
  - Built on top of each other, like a stack
- For each task, we will provide
  - Necessary background via a lecture
  - Q&As: after lecture and online via Slack
- Submitted tasks will be evaluated by
  - Automated grading system (only the last commit before the deadline counts)
  - Instructors

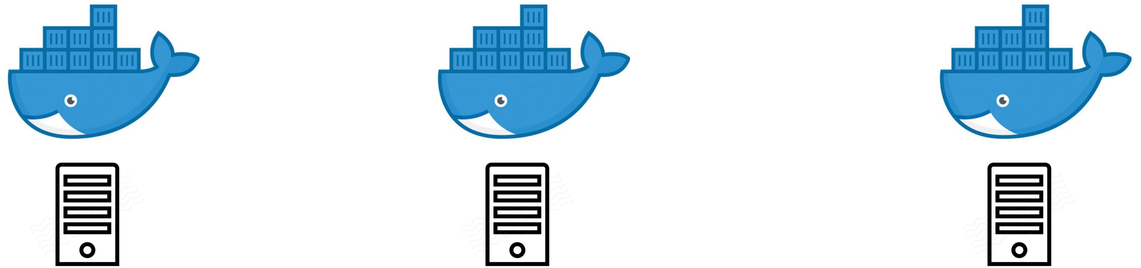
Layered architecture

#4: Distributed TXs: w/ and w/o replication

#3: Replicated distributed KVS

#2: Distributed KVS

#1: Single-node KVS	#1: Single-node KVS	…	#1: Single-node KVS



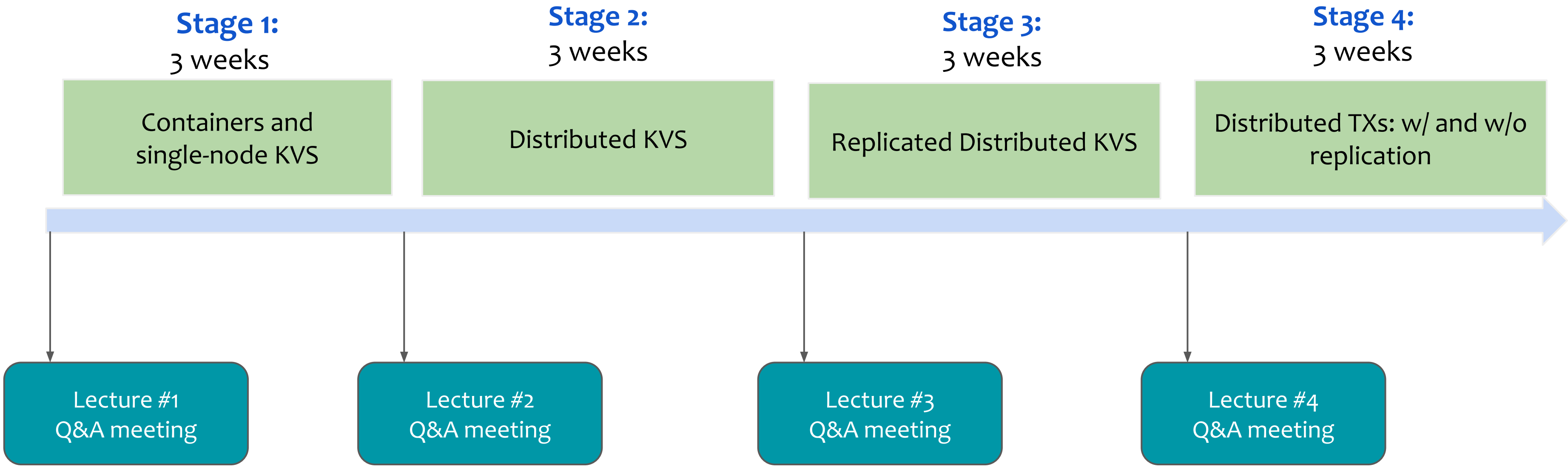
kubernetes

APACHE ZooKeeper™

Grading

Lecture	Category	Details	Grade
#1	Single-node KVS setup	Build and deploy a single node KVS	25%
#2	Distributed KVS	Shard the keys across multiple nodes: fault detection and server reconfiguration	25%
#3	Replicated distributed KVS	Replicate the KVS instances across these nodes for fault tolerance	25%
#4	Distributed Transactions	Support distributed transactions across keys and nodes: w/ and w/o replication	25%

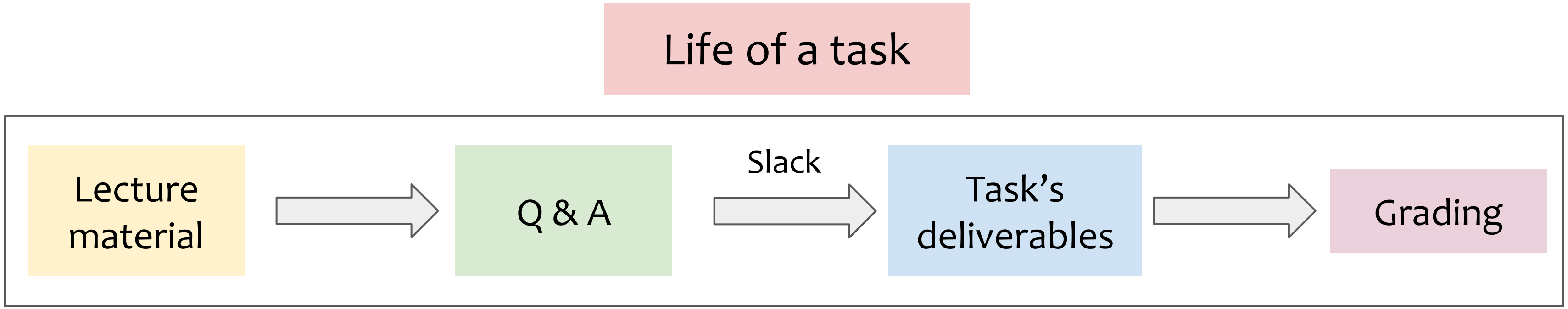
Timeline


Stage 1:
3 weeks
Containers and single-node KVS
Stage 2:
3 weeks
Distributed KVS
Stage 3:
3 weeks
Replicated Distributed KVS
Stage 4:
3 weeks
Distributed TXs: w/ and w/o replication
Lecture #1
Q&A meeting
Lecture #2
Q&A meeting
Lecture #3
Q&A meeting
Lecture #4
Q&A meeting

Organization

- Four lectures
  - One for each task – Recorded videos
    - Necessary material and deliverables will be explained
  - Q&A – Online on Thursdays via Zoom
- Online help
  - Slack channel will be monitored by the instructors/tutors

- Format:



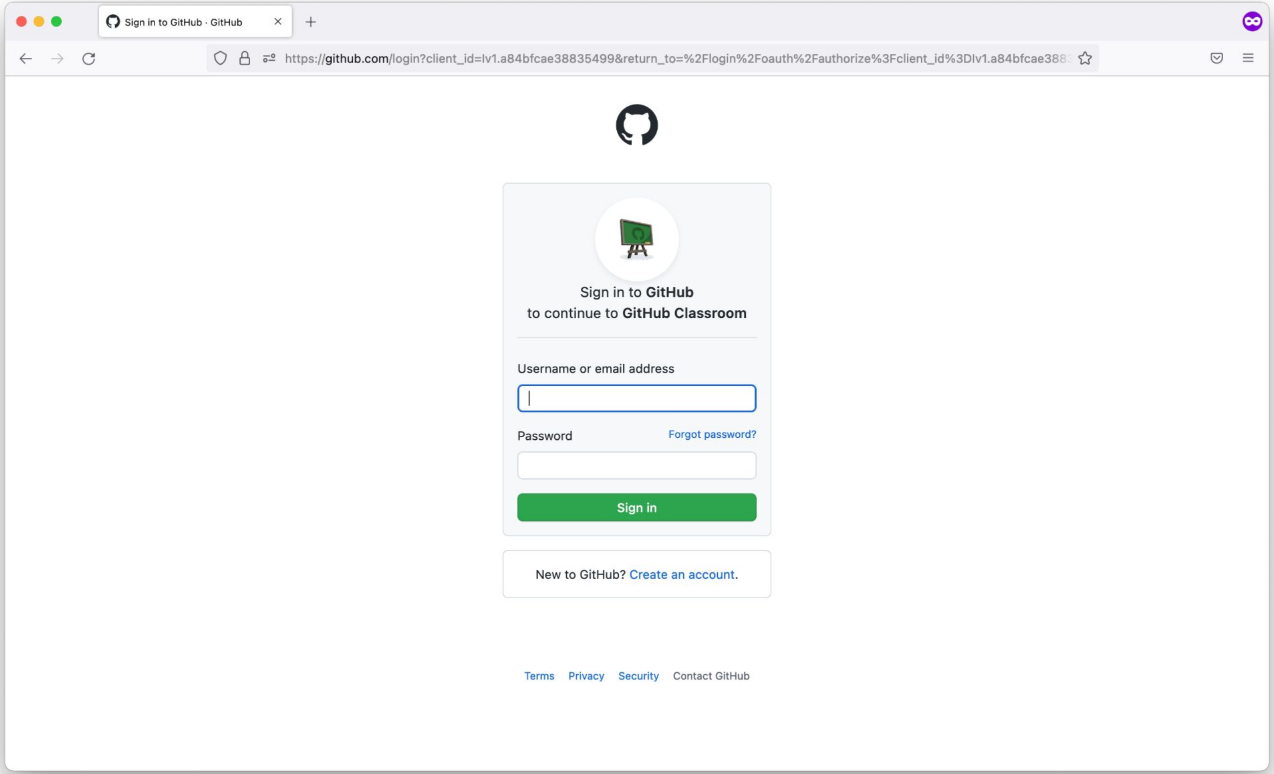
The assignments

- Code should be preferably written in C++
  - Other system programming languages like Rust may be used as well but there is no support by us
- We provide a detailed task description in the README.md of the task
- Code is hosted on GitHub and will be auto-graded via GitHub classroom
  - You will need to create a GitHub account
  - We will send you invitation links for each assignment via Slack
- Use Linux for the tasks

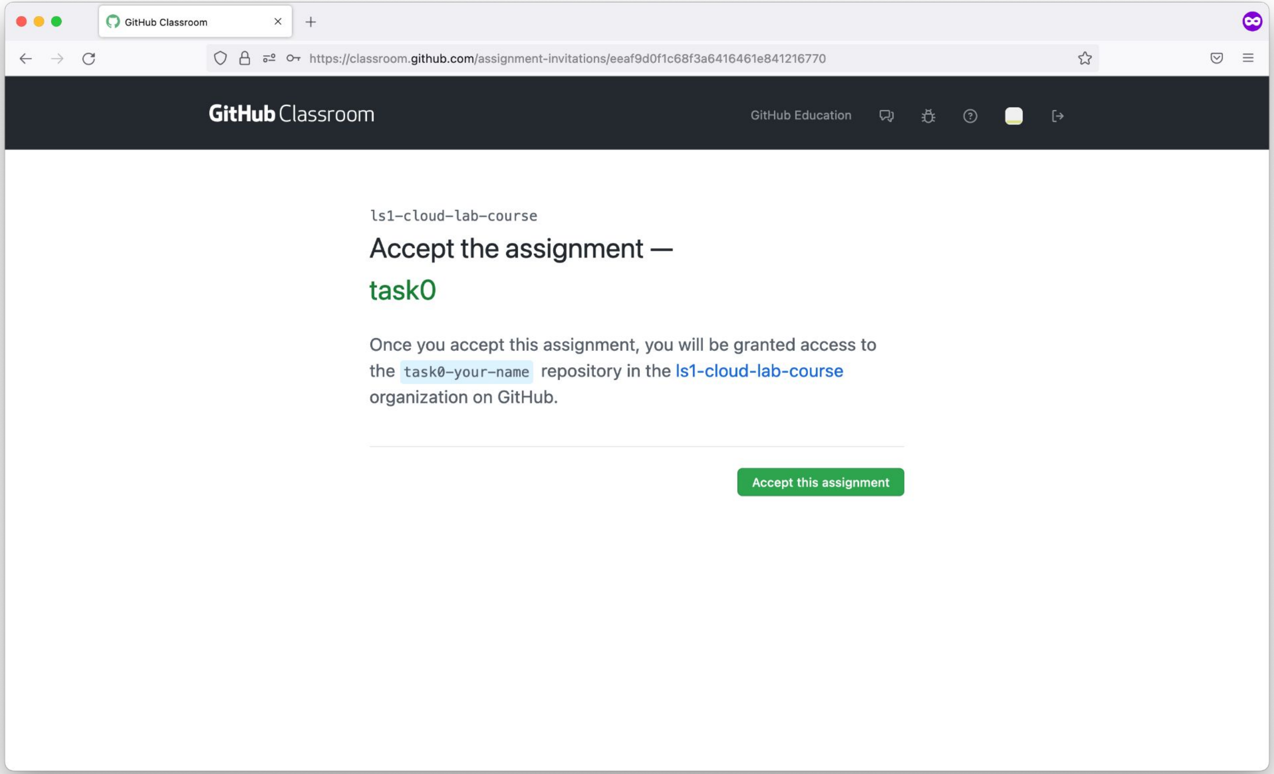
Our CI runners

- Fancy kubernetes cluster
- Three servers, each equipped with
  - 2x Intel Xeon 5215
    (10 cores / 20 threads each)
  - 128 GiB RAM
  - 10G NICs

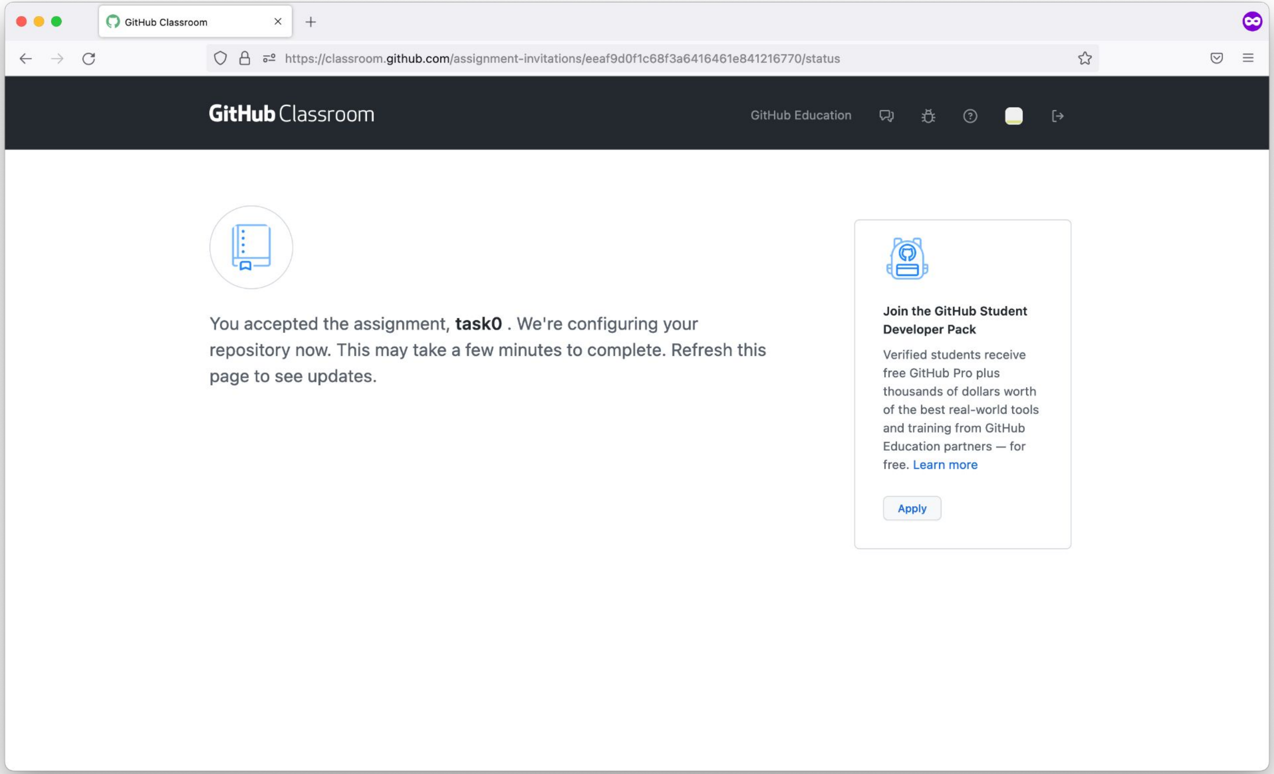
Getting the assignment code



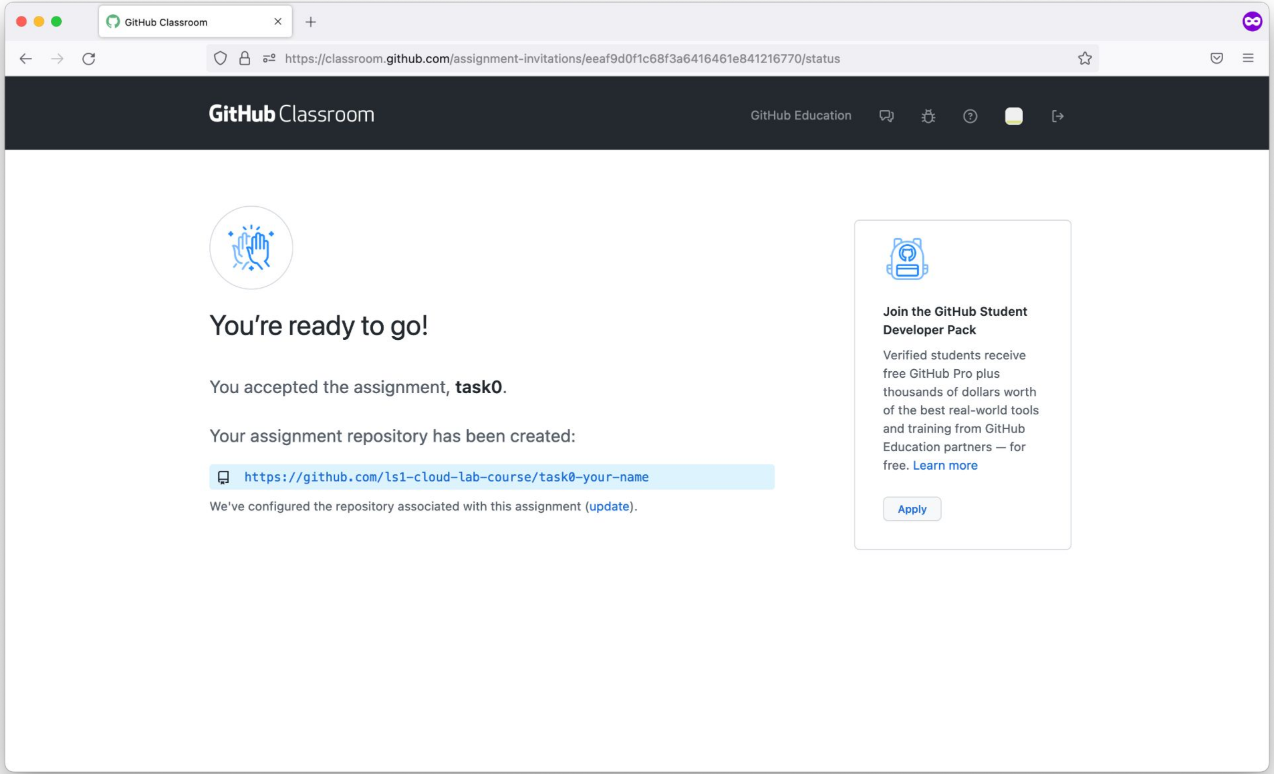
Getting the assignment code



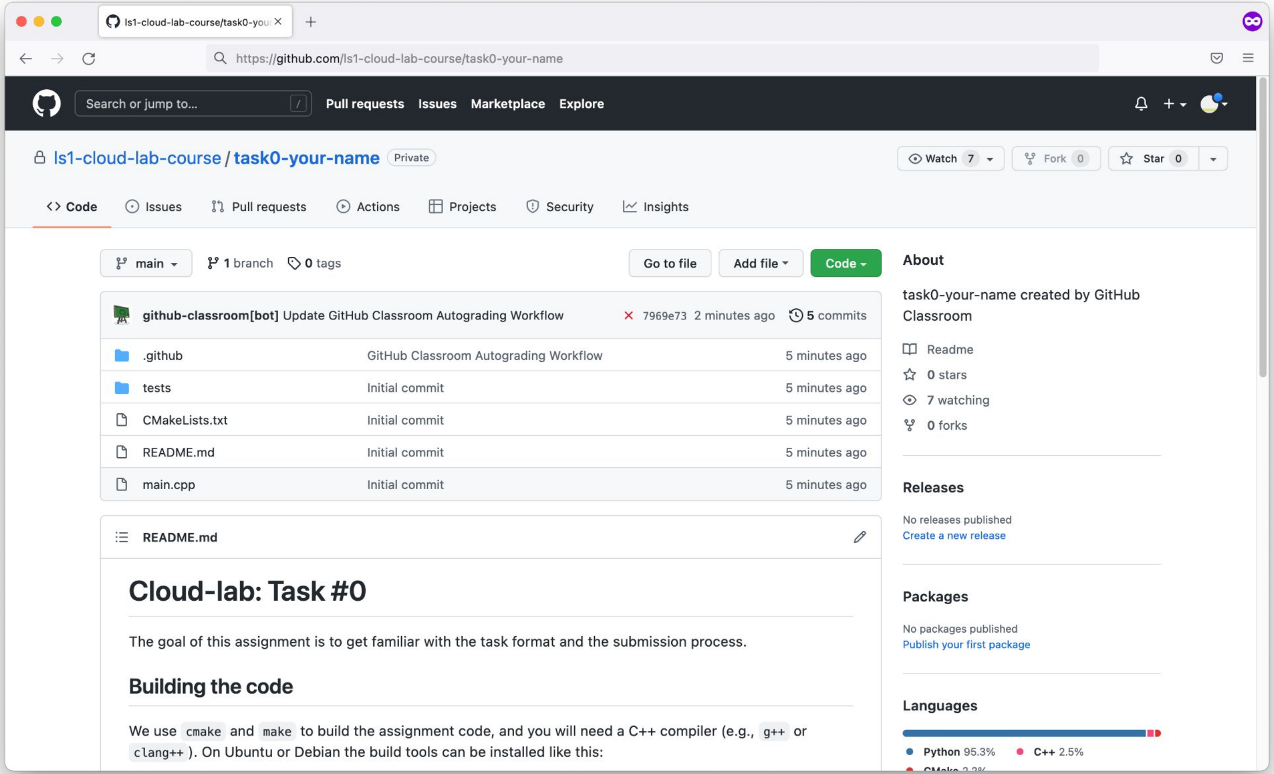
Getting the assignment code



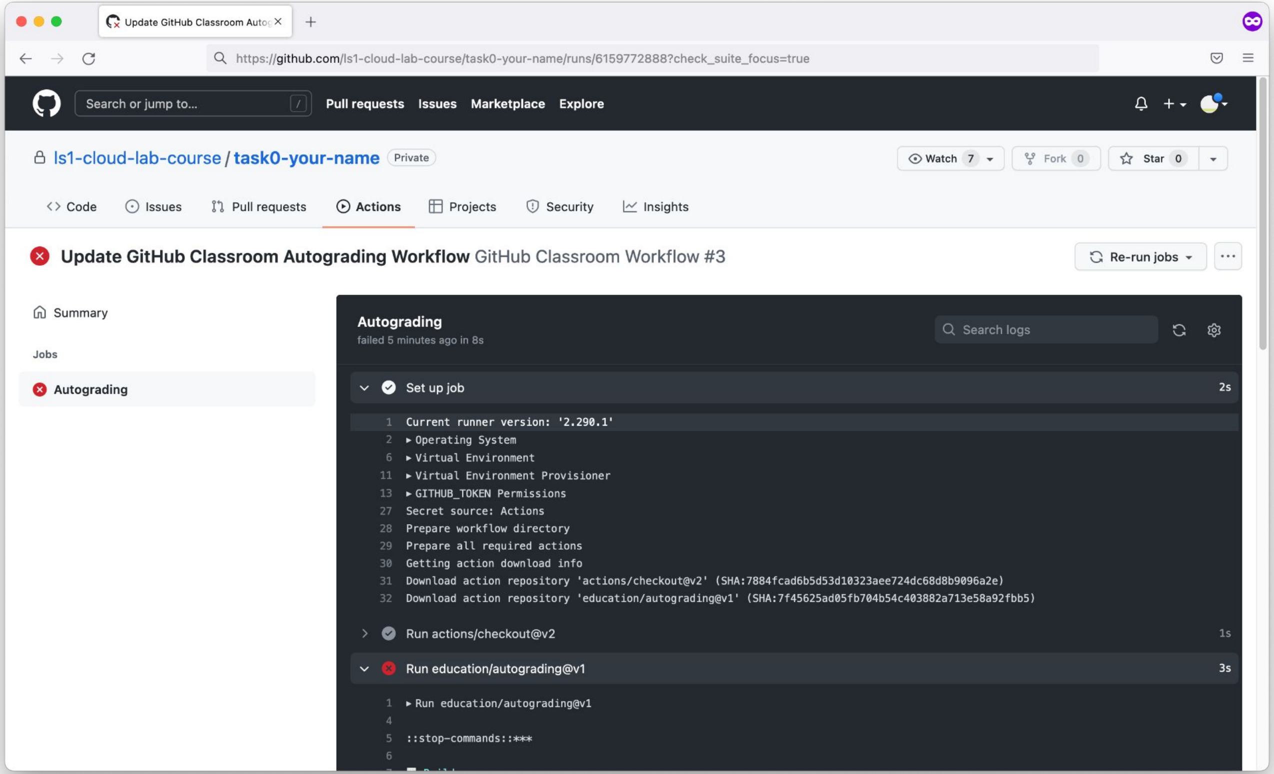
Getting the assignment code



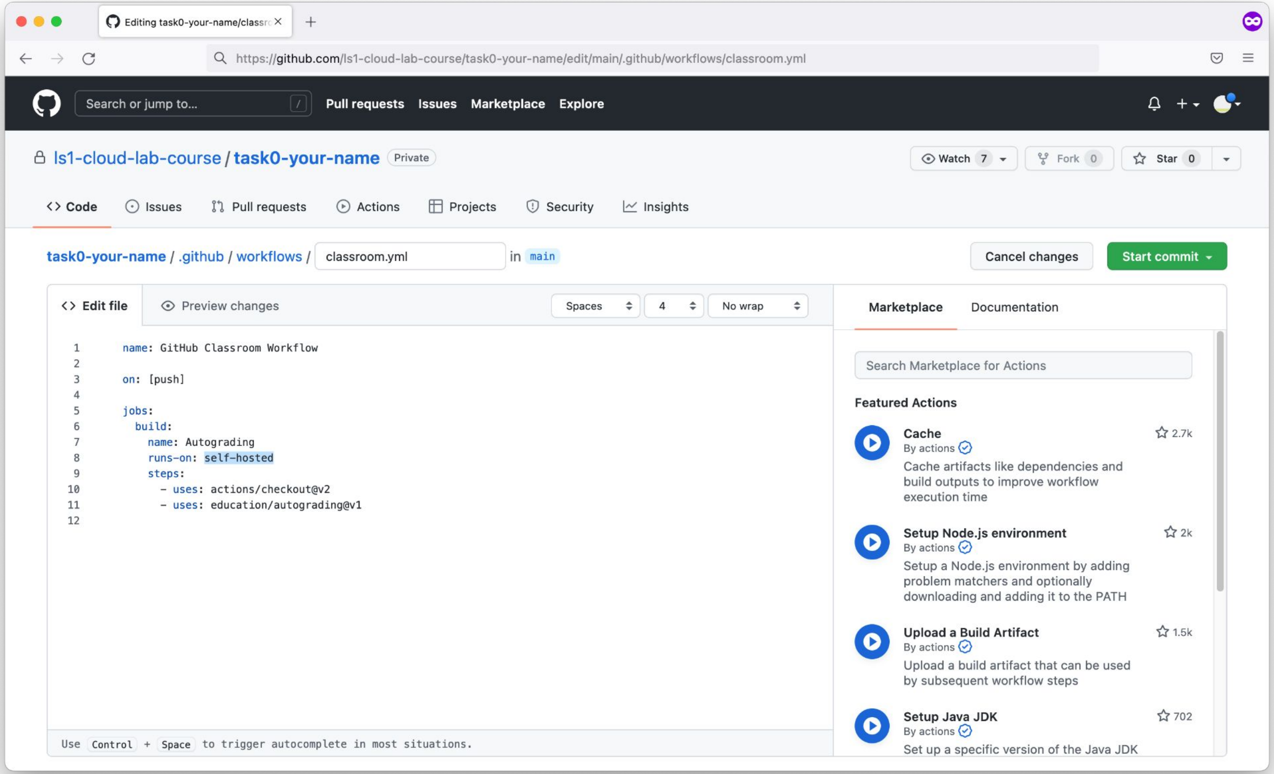
Getting the assignment code



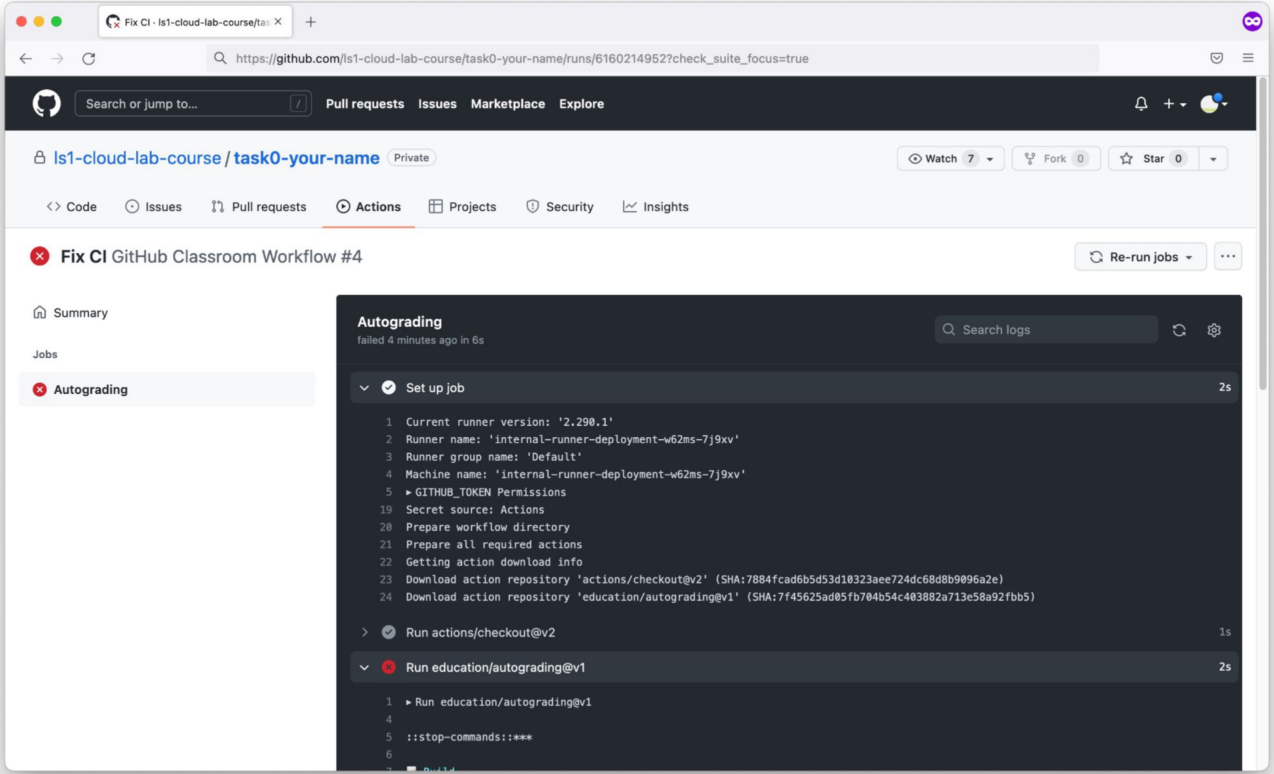
Submitting your code



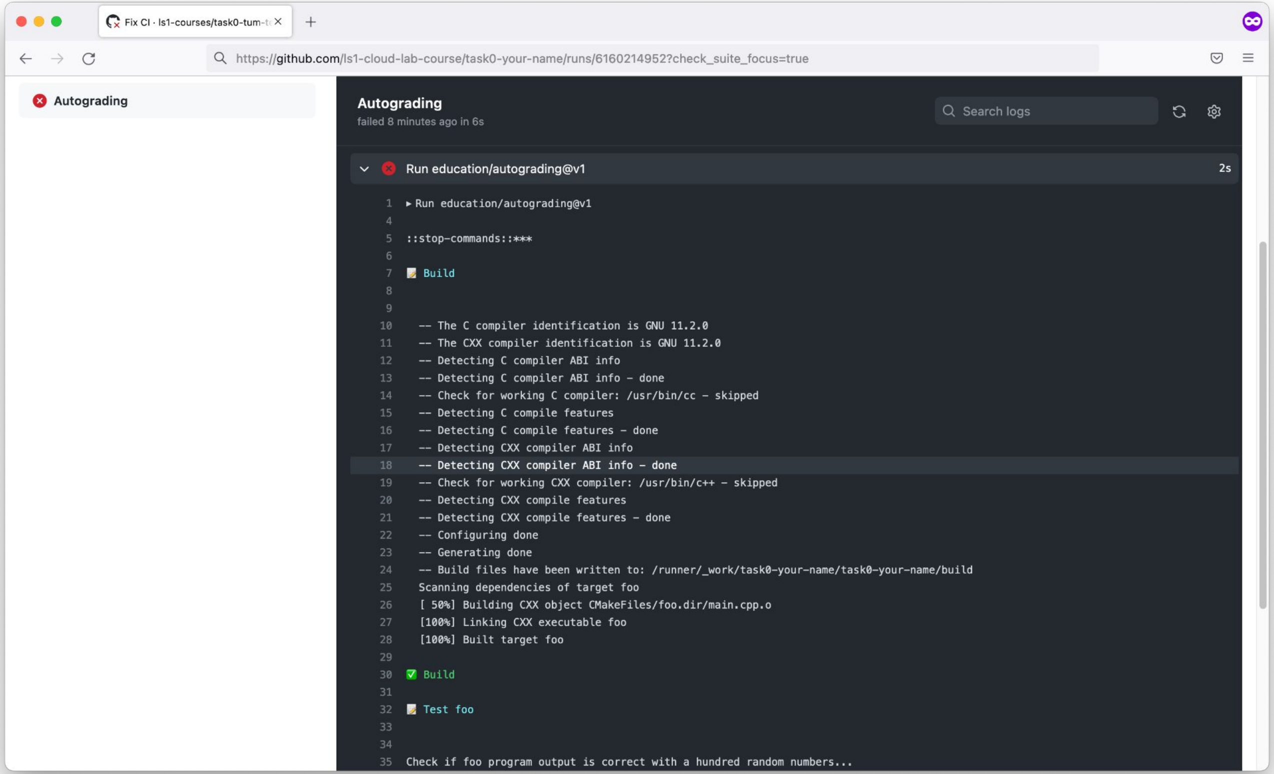
Submitting your code



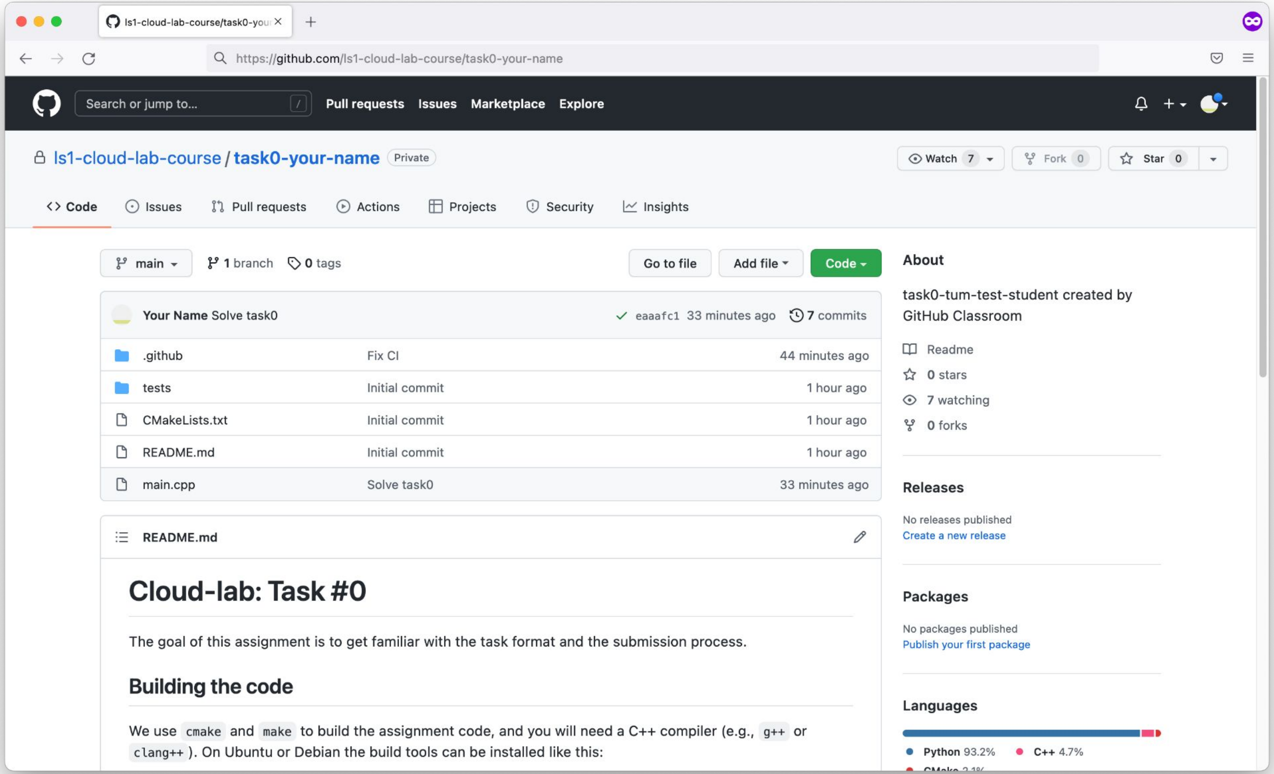
Submitting your code



Submitting your code



Submitting your code



Task #1:
Single-node KVS

Learning goals

In this task you will learn about:

- Client-server architecture(s)
- Single node key-value stores (KVSs), e.g., RocksDB

Background

Containers

Motivation

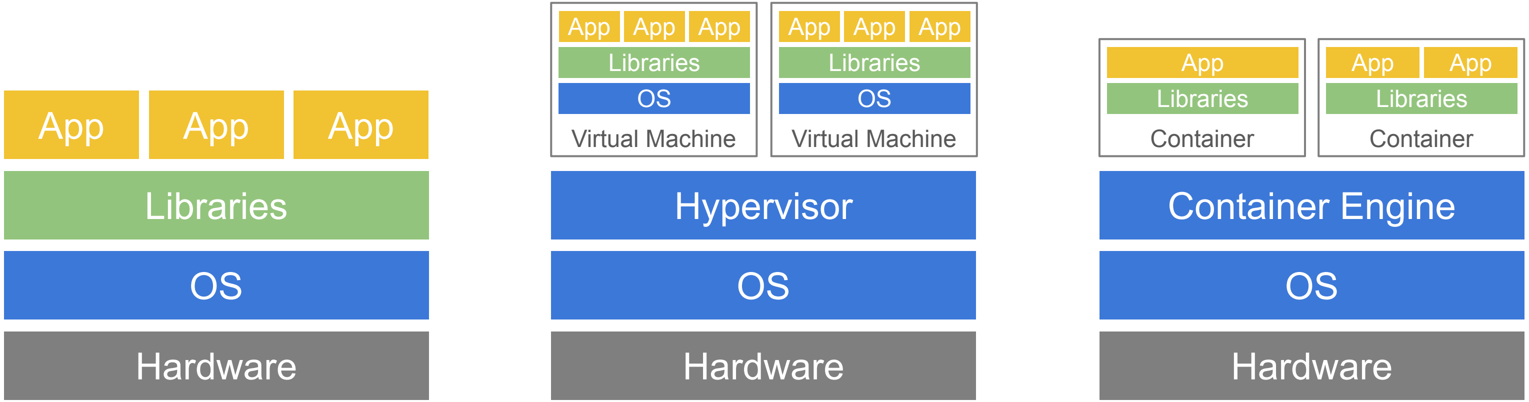
Share resources (IaaS, PaaS, …) in a flexible and cost-effective way

→ split applications into microservices

Advantages:

- Greater hardware resource-utilization
- Simply maintainable
- Scalable

Containers



Traditional system | Virtual machines | Containers

Containers

Unlike virtual machines …

- containers often contain only one application
- containers do not include an operating system (OS)
- containers limit access to resources through host OS mechanisms

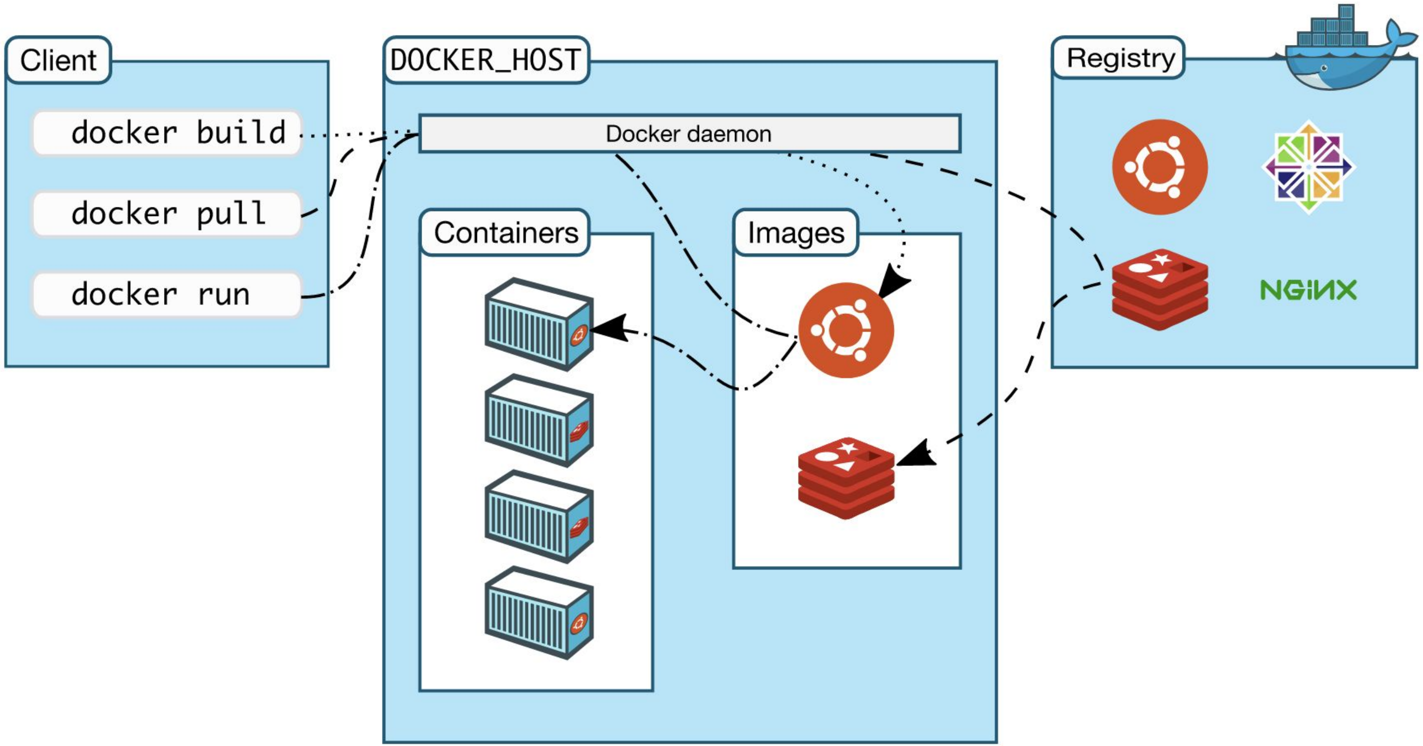
Container solutions

LXC

Docker

- Application container engine
  - Unlike LXC which is a system container engine
- Uses the resource isolation mechanisms of the Linux kernel
  - Namespaces
  - Cgroups
  - Layered filesystems
- Consists of three components: Software, objects and registry

Docker


Client
docker build
docker pull
docker run
DOCKER_HOST
Docker daemon
Containers
Images
Registry
NGINX

Getting started

1. Install Docker on your system, e.g.: `# apt install docker-ce`
2. Run some container: `$ docker run -it ubuntu /bin/bash`

Hints:

- List all containers: `$ docker ps -a`
- Get a shell for a running container: `$ docker exec -it <container name> /bin/sh`

Getting started

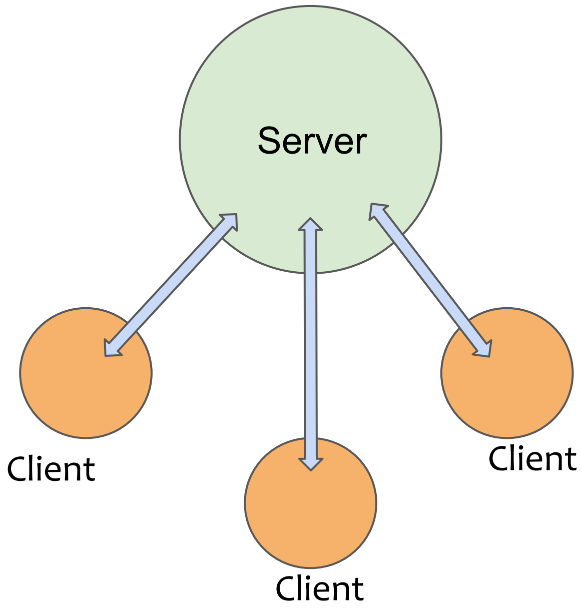
1. Create a simple Dockerfile (see Docker's docs)
2. Build an image based on the Dockerfile: `$ docker build -t <image name> .`
3. Run the container: `$ docker run -d <image name>`
4. Create and run multiple containers using Docker compose

Hints:

- Force clean rebuild of image: `$ docker build --no-cache -t <image name> .`
- To prevent a container from exiting early when debugging, add `CMD tail -f /dev/null` to the end of the Dockerfile

Client-server architecture

- Server
  - Usually a long running process (*daemon process*)
  - Manages some resources
  - Receives and process requests
- Client(s)
  - Sends one or more requests to the server
  - Wait for the server's reply
- Transport layer
  - Network medium
  - Transfers the data

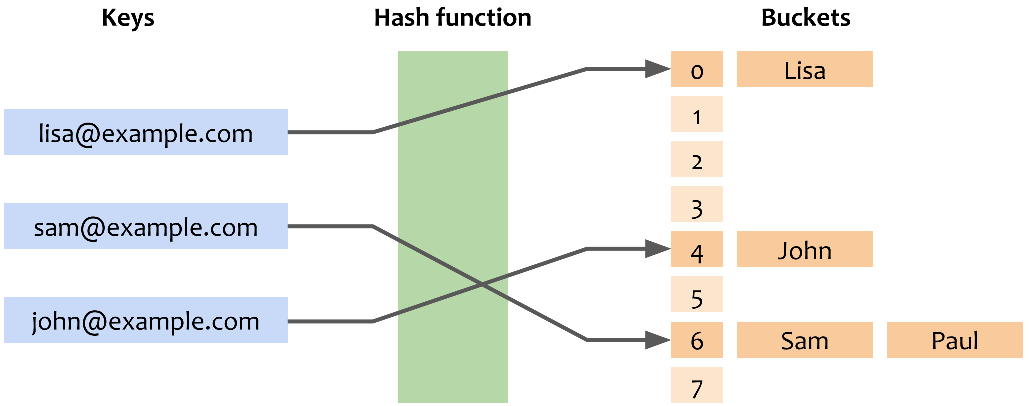


Key-Value store (KVs)

- Data structure
  - stores, retrieves and manages data
  - e.g., dictionaries, hash-tables

Key	Value
K1	AAA,BBB,CCC
K2	AAA,BBB
K3	AAA,DDD
K4	AAA,2,01/01/2022
K5	3,ZZZ,5623

dictionary



hash map

Motivation

Key-value stores play an important role at tech giants:

memcached	Redis	Voldemort	Dynamo
Facebook Twitter Zynga	GitHub Digg Blizzard Interactive	LinkedIn	Amazon

Motivation

Key-value stores play an important role in the scientific community:

- FASTER: A Concurrent Key-Value Store with In-Place Updates
  [SIGMOD ‘18]
- KVell: Design and Implementation of a Fast Persistent Key-value Store
  [SOSP ‘19]
- Nova-LSM: A Distributed, Component-based LSM-tree Key-value Store
  [SIGMOD ‘21]

KVs operations

Key-value stores typically implement a set of operations:

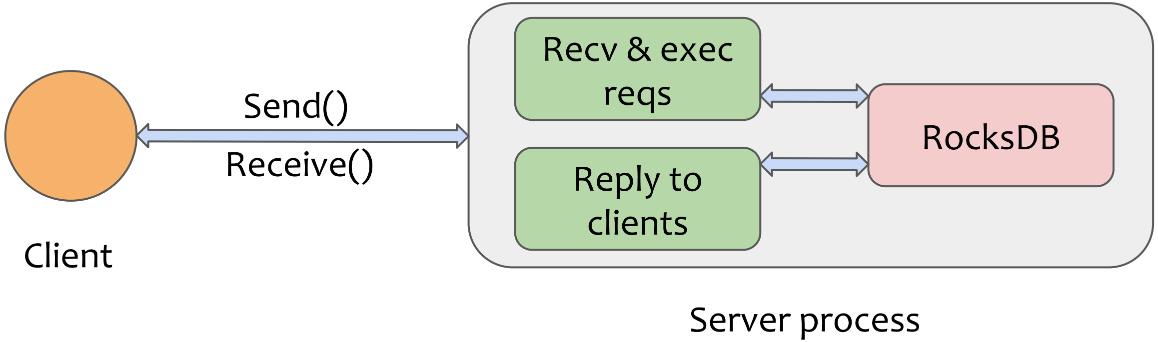
- GET
  - Retrieve a value by key
- PUT
  - Insert or update a key-value pair
- DELETE
  - Deletes a key-value pair (if it exists)
- Range Queries
  - Queries applying to a range of KV pairs

Challenges - design goals

- Performance
  - lock contention, significant write-traffic, complex memory management
  - low-latency operations and high throughput (I/O, batching)
  - parallelism (e.g., keys hashing)

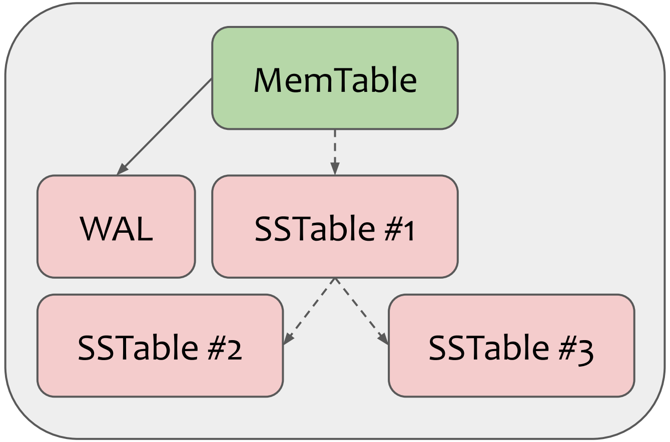
- Data properties
  - Persistency, e.g., persistent KVs or in-memory KVs
  - Consistency, e.g., linearizability or sequential consistency
  - Durability or crash consistency (for persistent KVs)

Task #1: Client/server arch + RocksDB



RocksDB architecture

- LSM-data structure
  - In-memory skiplist (MemTable)
  - SSTable files (persistent) organized on levels with (sorted) KV pairs
  - Compaction (background, multithreaded)
- Data properties
  - Linearizable reads, a read always "sees" the latest write
  - Durability, SSTables are persistent
  - Crash-consistency through Write-Ahead-Log (WAL)
- API
  - supports PUT, GET, DELETE queries



RocksDB

References

Talks & documentation:

- Rocksdb: https://github.com/facebook/rocksdb
- Rocksdb 101: https://www.youtube.com/watch?v=V_C-T5S-w8g
- Rocksdb talk: https://www.youtube.com/watch?v=tgzkgZVXKB4
- Protobufs: https://developers.google.com/protocol-buffers

Useful tutorials:

- Rocksdb tutorial: https://rocksdb.org/docs/getting-started.html
- Protobufs tutorial: https://developers.google.com/protocol-buffers/docs/cpptutorial
- Socket tutorial: https://www.linuxhowtos.org/C_C++/socket.htm
- Libevent tutorial: http://www.wangafu.net/~nickm/libevent-book/

Task #1

Your task for the next three weeks:

- Implement the network connection between client-server in a KVS architecture
- Implement the KVS operations
- Use
  - (1) RocksDB as the KV store
  - (2) kernel sockets for the networking
  - (3) google protobufs as the serialization protocol

Contacts

- Prof. Pramod Bhatotia
  - pramod.bhatotia@in.tum.de
- Dimitrios Stavrakakis
  - dimitrios.stavrakakis@tum.de

Workspace: http://ls1-courses-tum.slack.com/

Website: https://dse.in.tum.de/

GitHub: https://github.com/TUM-DSE/cloud-lab/

Channel: #ws-22-cloud-lab

Join us with TUM email address (@tum.de)

Thank you for listening!

See you in the Q&A session

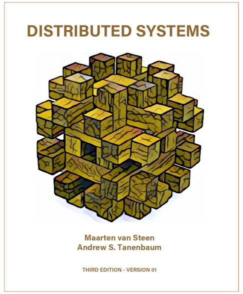
Code of conduct

- University plagiarism policy
  - https://www.in.tum.de/en/current-students/administrative-matters/student-code-of-conduct/

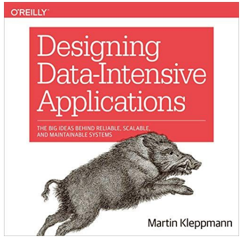
- Decorum
  - Promote freedom of thoughts and open exchange of ideas
  - Cultivate dignity, understanding and mutual respect, and embrace diversity
  - Racism and bullying will not be tolerated

Recommended readings

“Distributed Systems”
— Maarten van Steen and Andrew s. Tanenbaum



“Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems”
— Martin Kleppmann



“Designing Distributed Systems: Patterns and Paradigms for Scalable, Reliable Services Book”
— Brendan Burns