# CS45, Lecture 15
# VMs & Containers

**Winter 2023**

Akshay Srivatsan, Ayelet Drazen, Jonathan Kula

CS45 Lectures Course Info Software Installation Ed Discussion Gradescope

# CS 45: Software Tools Every Programmer Should Know

## Course Description

Classes teach you all about advanced topics within CS, from operating systems to machine learning, but there's one critical subject that's rarely covered, and is instead left to students to figure out on their own: proficiency with their tools. This course will teach you how to master the key tools necessary for being a successful computer scientist, such as the command line, version control systems, debuggers and linters, and many more. In addition, we will cover other key topics that are left out of standard CS classes, but that are essential to being a proficient computer scientist, including: security and cryptography, containers and virtual machines, and cloud computing.

## General Information

This course meets in-person twice a week, Mondays and Wednesdays from 4:30 to 5:50 at 300-300. The course is offered for 2 units on a S/NC basis. For more information about the course structure, visit the Course Info page.

## Computer Setup & Software Installation

This course will have about a 50/50 mix of conceptual background and hands-on practice with the tools we'll teach– this means you'll need to be able to download and install software onto your computer (either your personal computer, or a computer you have access to). Click here for more information about setting up your computer and the kinds of software we'll be using. (Let us know if this will present a challenge, e.g. if you're using a chromebook or a very old computer, or don't have access to a personal computer– we may be able to help 😄)

## Course Staff


CS 45: Software Tools Every Programmer Should Know

Course Description

Classes teach you all about advanced topics within CS, from operating systems to machine learning, but thereâ€™s one critical subject thatâ€™s rarely covered, and is instead left to students to figure out on their own: proficiency with their tools. This course will teach you how to master the key tools necessary for being a successful computer scientist, such as the command line, version control systems, debuggers and linters, and many more. In addition, we will cover other key topics that are left out of standard CS classes, but that are essential to being a proficient computer scientist, including: security and cryptography, containers and virtual machines, and cloud computing.

CS 45: Software Tools Every Programmer Should Know

Course Description

Classes teach you all about advanced topics within CS, from operating systems to machine learning, but there's one critical subject that's rarely covered, and is instead left to students to figure out on their own: proficiency with their tools. This course will teach you how to master the key tools necessary for being a successful computer scientist, such as the command line, version control systems, debuggers and linters, and many more. In addition, we will cover other key topics that are left out of standard CS classes, but that are essential to being a proficient computer scientist, including: security and cryptography, containers and virtual machines, and cloud computing.

General Information

This course meets in-person twice a week, Mondays and Wednesdays from 4:30 to 5:50 at 300-300. The course is offered for 2 units on a S/NC basis. For more information about the course structure, visit the Course Info page.

Computer Setup & Software Installation

# Learning Goals

- Have a basic conceptual understanding of VMs and their use cases
- Have a basic conceptual understanding of containerized applications and their use cases
- Know the difference between VMs and containers
- **Have concrete experience using a VM locally (UTM/VirtualBox)**
  - Spin up an Ubuntu (arm64/amd64) image locally
- **Have concrete experience using docker locally**
  - Deploy a pre-built container locally and expose a port

# Computers Within Computers

## Isolation & Testing (without extra hardware)

# Computers Within Computers

## Isolation & Testing (without extra hardware)

**Virtual Machines** allow you to emulate an entirely separate system within an existing system.

# Computers Within Computers

## Isolation & Testing (without extra hardware)

**Virtual Machines** allow you to emulate an entirely separate system within an existing system.

The application that oversees running VMs is called a **hypervisor** (or an **emulator**, depending on the mechanism it uses).

# Computers Within Computers

## Isolation & Testing (without extra hardware)

**Virtual Machines** allow you to emulate an entirely separate system within an existing system.

The application that oversees running VMs is called a **hypervisor.**

The computer running the hypervisor is called the **host**. The computer inside the VM is called the **guest**.

# Computers Within Computers

## Isolation & Testing (without extra hardware)

**Virtual Machines** allow you to emulate an entirely separate system within an existing system.

The application that oversees running VMs is called a **hypervisor.**

The computer running the hypervisor is called the **host**. The computer inside the VM is called the **guest**.

But… why?

# Use case for VMs

- **Isolation & Sandboxing**
  - Running untrusted (or dangerous) code
  - Allowing untrusted users access to an isolated environment
- **Resource Allocation**
  - Limit the maximum amount of resources an application or user may use
  - You might rent only part of the compute power available to you
- **System Management**
  - It is *very easy* to back up and restore an entire VM; shut it down, restart it, etc, without losing access to the host computer.
- **Cross-platform testing**
  - Test your apps on other operating systems without dedicating entire computers to them, or needing to reinstall/nuke a computer you're already using.
- **Prototyping**
  - Virtualize an entire network with multiple VMs and emulated routers/switches/etc to prototype an entire network! (GNS3)
- **Emulation**
  - Emulate entirely different CPU *architectures*– slower, but allows for an incredibly diverse range of applications (including legacy applications) to run on your computer. (For example, video games on very old consoles!)
- **Just for fun**
  - Ever wanted to try out a Linux distribution without changing your whole computer to Linux? Now you can!

# Emulation vs Virtualization?

- **Emulation** involves *recreating and modelling fully different computer architectures*– basically **mimicking hardware**.
  - This is necessary when **the operating system or software you're trying to run are written for a different class of hardware than the host machine.**

NINTENDO DS™

# Emulation vs Virtualization?

- **Emulation** involves *recreating and modelling fully different computer architectures*– basically **mimicking hardware**.
  - This is necessary when **the operating system or software you're trying to run are written for a different class of hardware than the host machine.**

- Meanwhile, **virtualization** works when the OS or software you're trying to run are written for the **same** class of hardware as the host machine.
  - **Virtualization** means "making one thing look or act like multiple things."
  - In this case, it means your computer hardware!
  - Virtualization **makes use of your computer hardware directly**, and is therefore **faster** than emulation. Usually by *a lot*.

# Security concerns w/ virtualization

When we're running a virtual machine, one of the **key guarantees** is that the guest **cannot** (without permission) access or affect the host.

- This allows us to try things (like `rm -rf /`) without worrying about destroying our host computer.
- Used for security research, etc.

# Virtualization is supported by hardware

In order to **provide virtual machines with complete CPU access** but also **ensure security of the host system**, CPU manufacturers have **extensions** (special instructions!) to make it faster/easier to use Virtual Machines

- Intel VT-x
- AMD-V

# The role of hypervisors

Okay, so CPUs have extensions for virtualization; do hypervisors need to do anything else interesting? **Yes!**

- There’s more to provide to the guest than just cpu & memory:
  - network access? → Virtual Networks
  - hard drive space? → Virtual Hard Drives
  - CD-ROM access (yes still!) → Disk Images

# Bare Metal vs Hosted Hypervisors

Two types of hypervisors: **bare metal** (type 1) and **hosted** (type 2).

- Bare Metal hypervisors is when **the hypervisor is the host.**
  - This means the hypervisor is the “operating system.”
- Hosted hypervisors are **programs that run on a host operating system.**
  - We’re using type 2 hypervisors when we use UTM and VirtualBox

# Hypervisors

**Bare Metal Hypervisors**

- VMWare ESXi
- Hyper-V
- Proxmox
- Xen

**Hosted Hypervisors**

- VirtualBox
- VMWare Workstation
- UTM (uses qemu under the hood)
- qemu (also supports **emulation**)
- Hyper-V
- Parallels

# Demo: Run Ubuntu Desktop

We’re going to run a version of Ubuntu Linux on your computer!

# Containers

- Light(-er) weight
  - Allows them to be easily distributed
  - Rather than virtualizing the entire OS, it continues to use the host's **kernel**/operating system as a "base" to service whatever is running within the container.
- Faster than VMs (usually)
  - Can also use an emulated system if necessary → runs within a VM
- Designed for ephemerality
  - Containers are "disposable" – any long-term data should be stored in separate persistent "volumes"

# Containerization

You can take an application and wrap it in a **container** to ensure a consistent running environment.

- You can define the **operating system it expects to use** (but not the kernel)
- You can define the **CPU architecture the program expects** (and if the CPU architecture differs from the host, it will have to run within a virtual machine)
- You can define **dependencies and other programs that the application expects to be installed**
- You can define **the "hard drive" layout the program expects**
- All this, and the application gets a level of **isolation** from other applications on the system.

# docker

`docker` is one of the most popular tools to create and manage containers. Here's some useful-to-know terms:

- **Containers** are an ephemeral object representing a copy of a program, based on an **Image**
- **Images** are built from **Dockerfiles**, and represent a frozen copy of an application and everything needed to run it
- **Dockerfiles** are special scripts that are used to build images, including:
  - Instructions on adding files (e.g. program files) from your local system
  - Instructions on adding dependencies
- **Volumes** store persistent data even past the lifetime of a container.

# docker

`docker` is one of the most popular tools to create and manage containers. Here's some useful-to-know terms:

- **Containers are like the downloaded application.**
- **Images are like the .zip, .msi, or .dmg that you download from the website**
- **Dockerfiles are like scripts that create the .zip/.msi/.dmg**
- **Volumes are like the places on your computer where your applications store data**

# Demo: run Ubuntu using docker

Let’s run a basic ubuntu system using Docker!

# Demo: run Ubuntu using docker

Let’s run a basic ubuntu system using Docker!

```
docker run -it ubuntu:latest bash
```

# Demo: run Ubuntu using docker

Let's run a basic ubuntu system using Docker!

```
docker run -it ubuntu:latest bash
```

run a new container based on an image

# Demo: run Ubuntu using docker

Let’s run a basic ubuntu system using Docker!

```
docker run -it ubuntu:latest bash
```

Keep **STDIN** open (allows us to write things into the container)

# Demo: run Ubuntu using docker

Let's run a basic ubuntu system using Docker!

```
docker run -it ubuntu:latest bash
```

Allocate a TTY to the container (allows the container to receive things we write)

# Demo: run Ubuntu using docker

Let’s run a basic ubuntu system using Docker!

```
docker run -it ubuntu:latest bash
```

All together: “Run interactively”

# Demo: run Ubuntu using docker

Let’s run a basic ubuntu system using Docker!

```
docker run -it ubuntu:latest bash
```

Specifies the image to run (the latest version of ubuntu)

# Demo: run Ubuntu using docker

Let's run a basic ubuntu system using Docker!

```
docker run -it ubuntu:latest bash
```

Specifies which command to run within the image (bash)

# Dockerfiles

Dockerfiles, which are basically always called `Dockerfile`, look like this:

```
FROM ubuntu:latest

RUN apt-get -y update
RUN apt-get -y upgrade
RUN apt-get install -y build-essential
WORKDIR /calculator_app
ADD ./calculator/ ./
RUN make clean; make
ENTRYPOINT ["./calculator"]
```

# Dockerfiles

**FROM** specifies the base of the image you're building.

```
FROM ubuntu:latest

RUN apt-get -y update
RUN apt-get -y upgrade
RUN apt-get install -y build-essential
WORKDIR /calculator_app
ADD ./calculator/ ./
RUN make clean; make
ENTRYPOINT ["./calculator"]
```

# Dockerfiles

In this case, we're basing our image off the latest version of **ubuntu**.

```
FROM ubuntu:latest

RUN apt-get -y update
RUN apt-get -y upgrade
RUN apt-get install -y build-essential
WORKDIR /calculator_app
ADD ./calculator/ ./
RUN make clean; make
ENTRYPOINT ["./calculator"]
```

# Dockerfiles

**ubuntu** is the image name.

```
FROM ubuntu:latest

RUN apt-get -y update
RUN apt-get -y upgrade
RUN apt-get install -y build-essential
WORKDIR /calculator_app
ADD ./calculator/ ./
RUN make clean; make
ENTRYPOINT ["./calculator"]
```

# Dockerfiles

**ubuntu** is the image name. **latest** is the tag.

```
FROM ubuntu:latest

RUN apt-get -y update
RUN apt-get -y upgrade
RUN apt-get install -y build-essential
WORKDIR /calculator_app
ADD ./calculator/ ./
RUN make clean; make
ENTRYPOINT ["./calculator"]
```

# Dockerfiles

**RUN** commands will run the given command inside a shell.

```
FROM ubuntu:latest

RUN apt-get -y update
RUN apt-get -y upgrade
RUN apt-get install -y build-essential
WORKDIR /calculator_app
ADD ./calculator/ ./
RUN make clean; make
ENTRYPOINT ["./calculator"]
```

# Dockerfiles

These commands form **intermediate containers**; each command builds off of the previous.

```
FROM ubuntu:latest

RUN apt-get -y update
RUN apt-get -y upgrade
RUN apt-get install -y build-essential
WORKDIR /calculator_app
ADD ./calculator/ ./
RUN make clean; make
ENTRYPOINT ["./calculator"]
```

# Dockerfiles

These commands form **intermediate containers**; each command builds off of the previous.

```
FROM ubuntu:latest

RUN apt-get -y update
RUN apt-get -y upgrade
RUN apt-get install -y build-essential
WORKDIR /calculator_app
ADD ./calculator/ ./
RUN make clean; make
ENTRYPOINT ["./calculator"]
```

# Dockerfiles

The commands below install build tools like **make**.

```
FROM ubuntu:latest

RUN apt-get -y update
RUN apt-get -y upgrade
RUN apt-get install -y build-essential
WORKDIR /calculator_app
ADD ./calculator/ ./
RUN make clean; make
ENTRYPOINT ["./calculator"]
```

# Dockerfiles

**WORKDIR** specifies a new **working directory** from that point onwards.
(it's like a **cd** that sticks)

```
FROM ubuntu:latest

RUN apt-get -y update
RUN apt-get -y upgrade
RUN apt-get install -y build-essential
WORKDIR /calculator_app
ADD ./calculator/ ./
RUN make clean; make
ENTRYPOINT ["./calculator"]
```

# Dockerfiles

**ADD** will copy files from your local directory (relative to where the **Dockerfile** is located) into the image.

```
FROM ubuntu:latest

RUN apt-get -y update
RUN apt-get -y upgrade
RUN apt-get install -y build-essential
WORKDIR /calculator_app
ADD ./calculator/ ./
RUN make clean; make
ENTRYPOINT ["./calculator"]
```

# Dockerfiles

We can run **make** at this point because we installed it earlier.

```
FROM ubuntu:latest

RUN apt-get -y update
RUN apt-get -y upgrade
RUN apt-get install -y build-essential
WORKDIR /calculator_app
ADD ./calculator/ ./
RUN make clean; make
ENTRYPOINT ["./calculator"]
```

# Dockerfiles

**ENTRYPOINT** describes what what should happen when we **run** the image.

```
FROM ubuntu:latest

RUN apt-get -y update
RUN apt-get -y upgrade
RUN apt-get install -y build-essential
WORKDIR /calculator_app
ADD ./calculator/ ./
RUN make clean; make
ENTRYPOINT ["./calculator"]
```

# Dockerfiles

**ENTRYPOINT** describes what what should happen when we **run** the image. In this case, it says we should run the **./calculator** program, which we just built

```
FROM ubuntu:latest

RUN apt-get -y update
RUN apt-get -y upgrade
RUN apt-get install -y build-essential
WORKDIR /calculator_app
ADD ./calculator/ ./
RUN make clean; make
ENTRYPOINT ["./calculator"]
```

# Dockerfiles

At the end, Docker will “freeze” all the intermediate containers all together into an **image**. Each command forms a **layer** of that image.

```
FROM ubuntu:latest

RUN apt-get -y update
RUN apt-get -y upgrade
RUN apt-get install -y build-essential
WORKDIR /calculator_app
ADD ./calculator/ ./
RUN make clean; make
ENTRYPOINT ["./calculator"]
```

# Demo: Create a docker container

Let's create a Calculator Dockerfile together!

# Orchestration

When you have a lot of containers in a system, you may want to put another meta-system to reason about and manage your individual containers for you.

This is called **container orchestration**

# Kubernetes (k8s)

**Kubernetes** (also known as **k8s**) One of the most popular container orchestration tools.

- Highly integrated with specific cloud environments
  - Can manage certain AWS resources, for instance
- High-level complicated management tasks
  - Load balancing
  - Dynamically allocating containers to available computers (nodes)
- **High-overhead system**
  - Useful if the advanced management tools are valuable and/or if you have a **lot** of compute power you're managing.
  - Minimum: 3 to 5 separate computers all dedicated to Kubernetes

# Demo: Kubernetes

Let’s show off running a simple web server in Docker, then Kubernetes

# Deploying Your Apps

We'll talk about this more on Wednesday, but:

- Most cloud providers will offer you a "VPS" ("virtual private server") – this is basically just a VM they give you access to! (Our server is a VPS).
- Some will offer you an "app runner" or "Apps" or similar– usually this is them offering to run your container on their servers!
- Some will offer Kubernetes services: Kubernetes deployments are usually **cloud-specific**; so there will be special integration with a cloud provider's offering of Kubernetes.

# Questions?