10 November 2020

SDE Project



A technological guide for the underlying distributed design of modern systems.

Note: This guide is at the conceptual level. It requires generalized knowledge of all information technology above the hardware level to be fully utilized. It does not contain step by step guides to how to set up any of the technologies mentioned.

# Systems Design in a nutshell

## Introduction

System design choices should first be influenced by the goals of the system. Different systems have different tolerances and strengths. A bank system is designed for reliability, security and consistency while a pizza chain’s system is designed for speed, efficiency and cost reduction. Consistency and security are less of a factor in a pizza chain, a wrong order is not going to cause potentially catastrophic results and a delivery man or pizza shop being robbed is not going to cause the entire chain to go bankrupt (if managed properly).

There is more than likely a software equivalent of any logistical paradigm in existence. A good way of thinking about it is for any function a human does in an organization, there will be a software equivalent doing the same function in a system.

## Conceptual & Technological design

Designing a system on a conceptual level and on a technological level are completely separate activities, although they are related, they are not equivalent.

### Conceptual

Conceptual design choices are relatively straight forward, we decide what the system is doing and what it should and should not be good at while doing it.

Conceptual choices are limited by and influenced by the technology available.

### Technological

Technological design choices are influenced mainly by 3 factors.

1. Scale

How many people are going to use the system?

How much data will be moving around the system?

What is the geographical scope of the system?

1. Medium

How does the system interact with humans? Or does it even need to?

1. Conceptual requirements

What are the features the system is required to have?

Just to add another level of difficulty. The technological choices made should ideally be able to support expansions to the systems that have not yet been thought of or are not yet possible as well as shifts in conceptual features and requirements

# Distributed Software Architecture

Conceptually most system designs are a copy of logistic supply chains we have already figured out in the physical world. We will take a look into one of these analogical examples of a distributed architecture and also show their software equivalents.

This example describes what is known as micro service architecture. Do note that while micro services are great, people tend to go overboard and fragment too many things into too many services. There is no set guide or method on how to avoid this, it is pretty much up to the development team to be intelligent enough to design it well.

The microservice philosophy is essentially capitalism in software form. We want each citizen to do one thing extremely well.

The idea of microservices actually originated from object oriented programming, within a program we usually also have services that handle specific tasks, such as a database helper which abstracts away a layer away from the database connection mechanisms, so a change to the database will only need to be dealt with at the database helper, no other code will (should) be affected by the changes. We just took this idea and abstracted it into components of a system instead of components of a program.

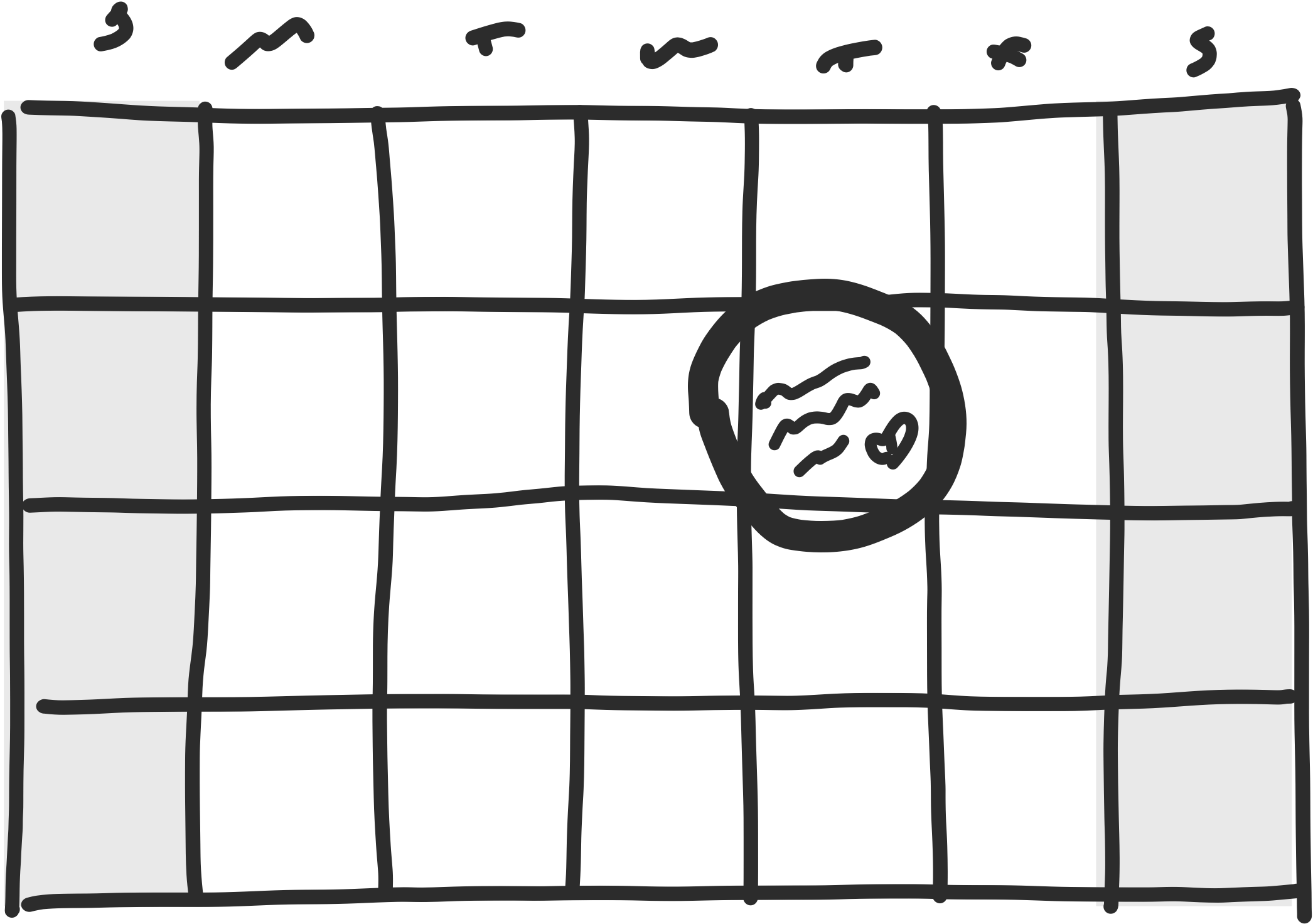
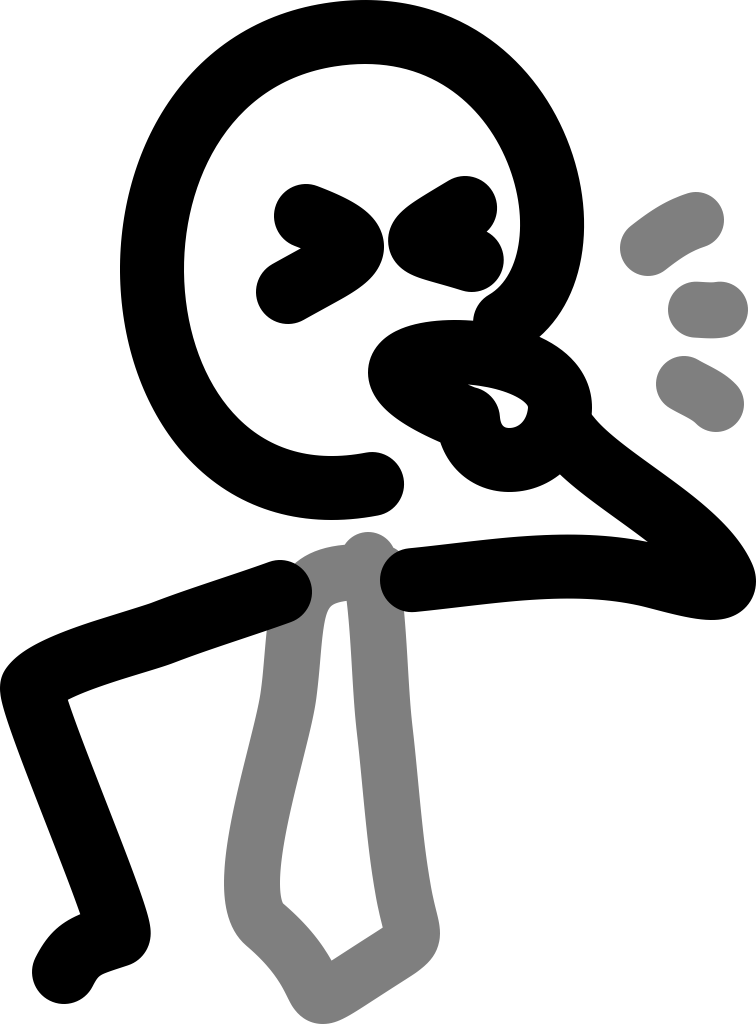
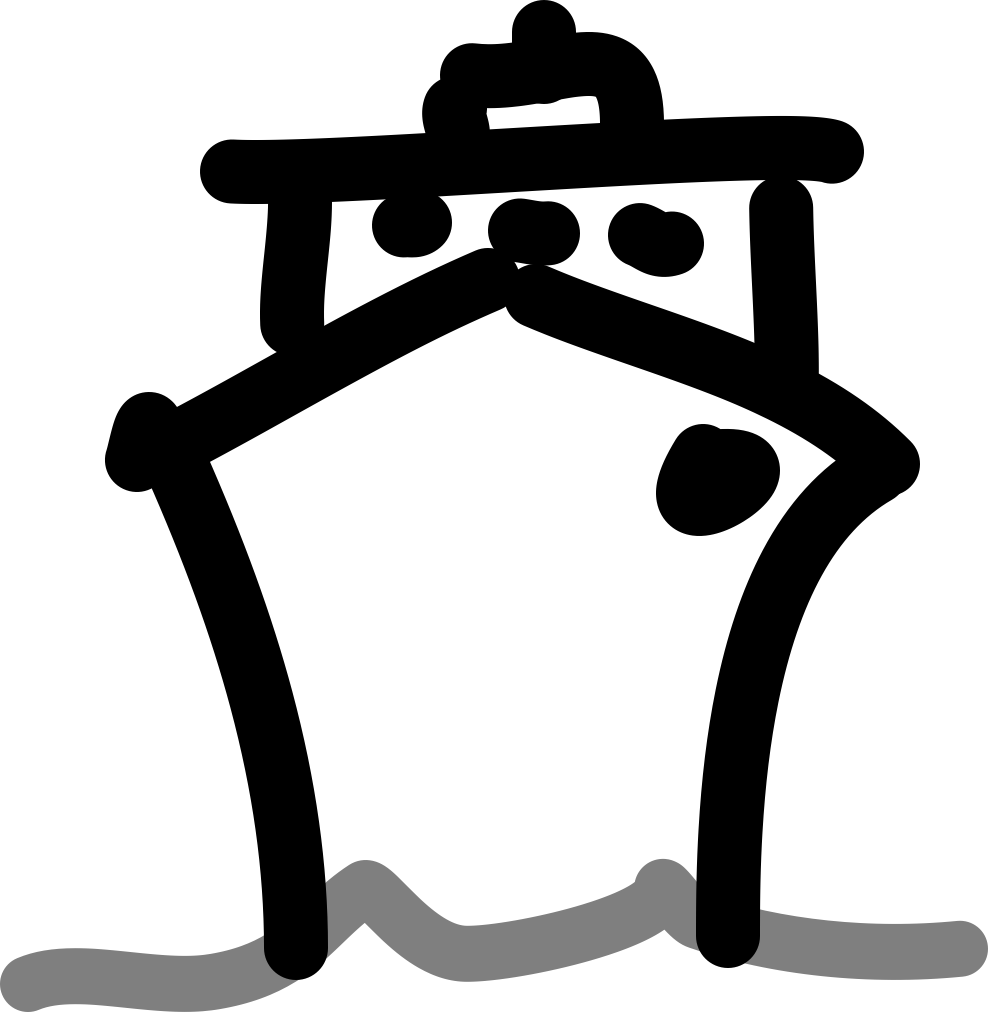
### Pizza Restaurant Chain

There can be many of each component in the following diagrams.

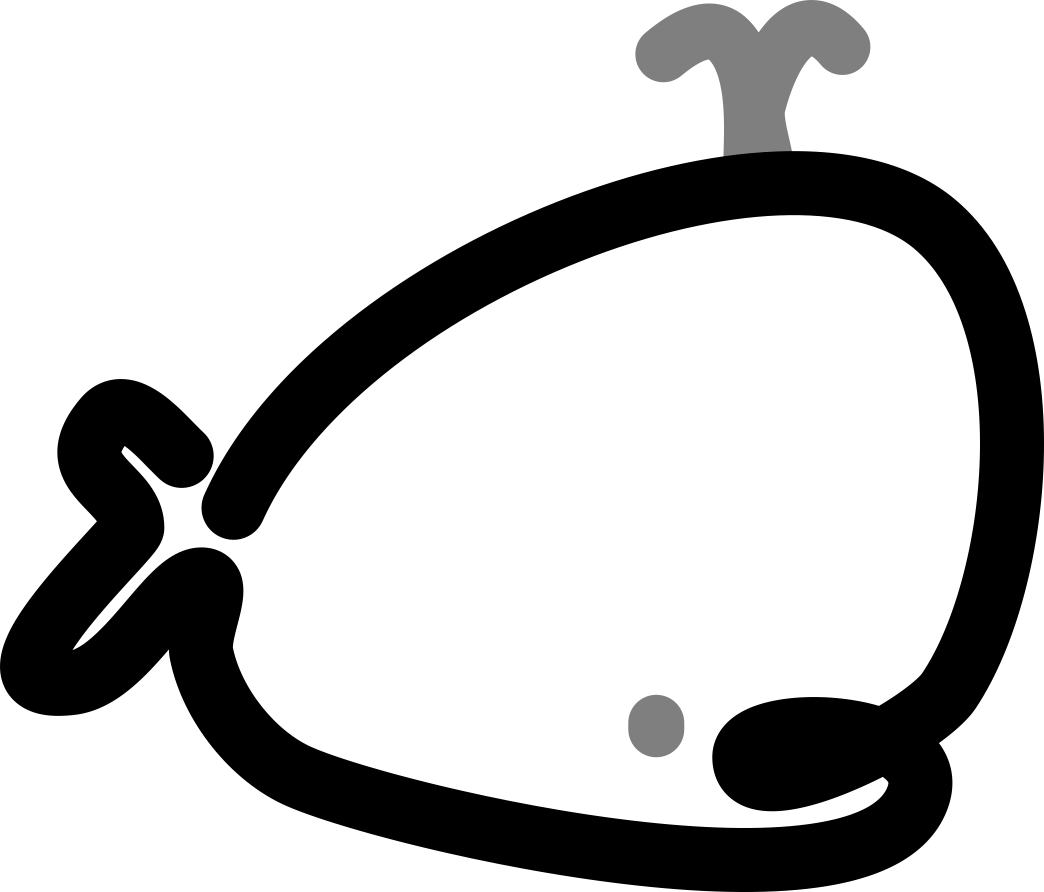
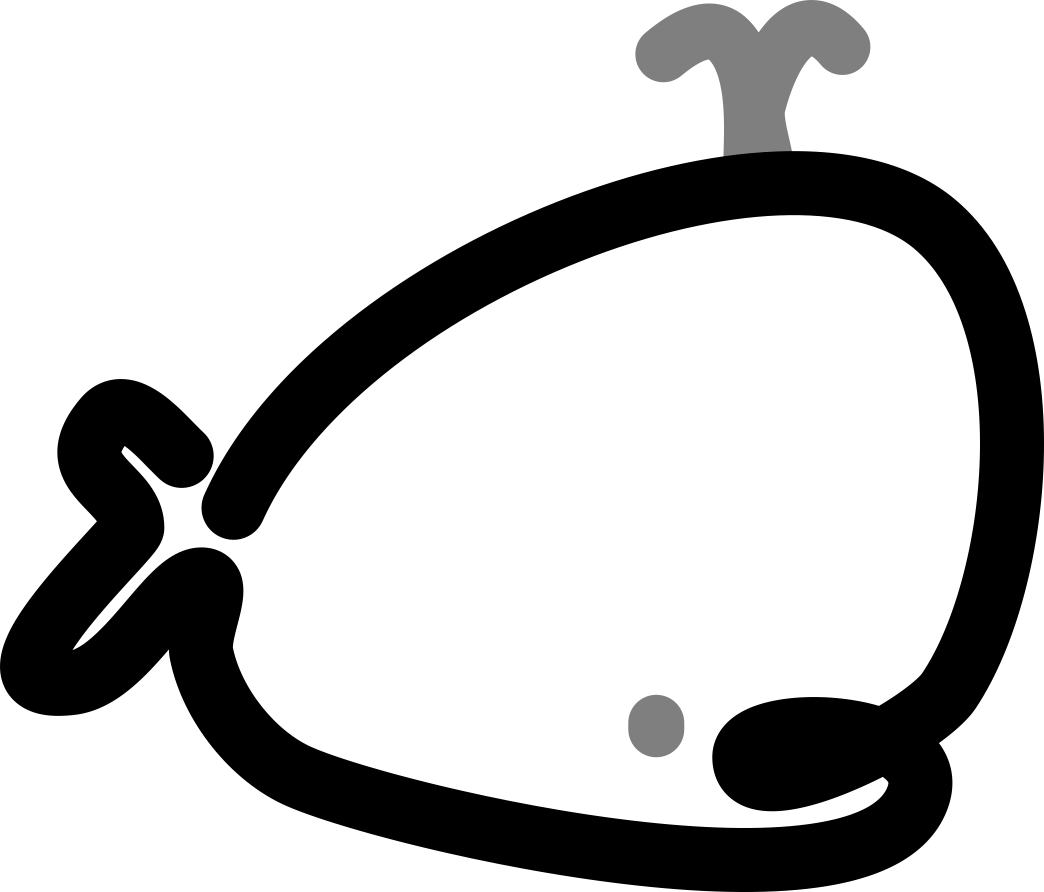
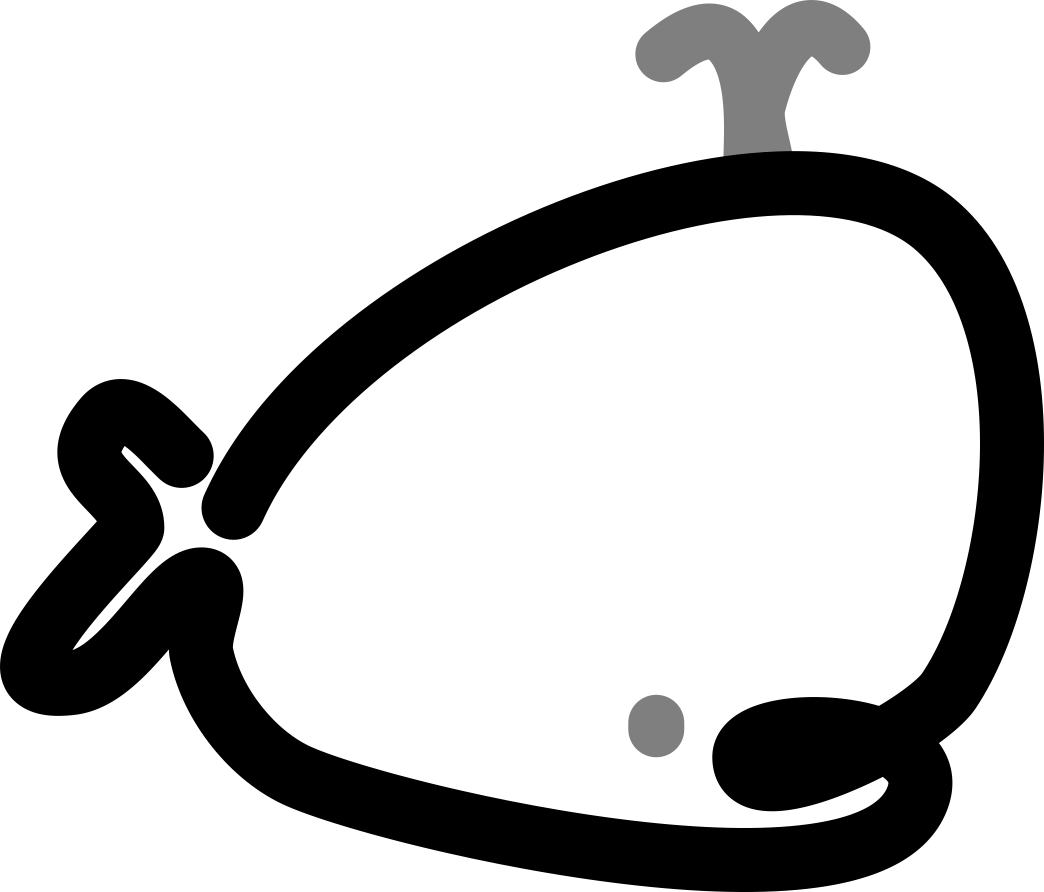
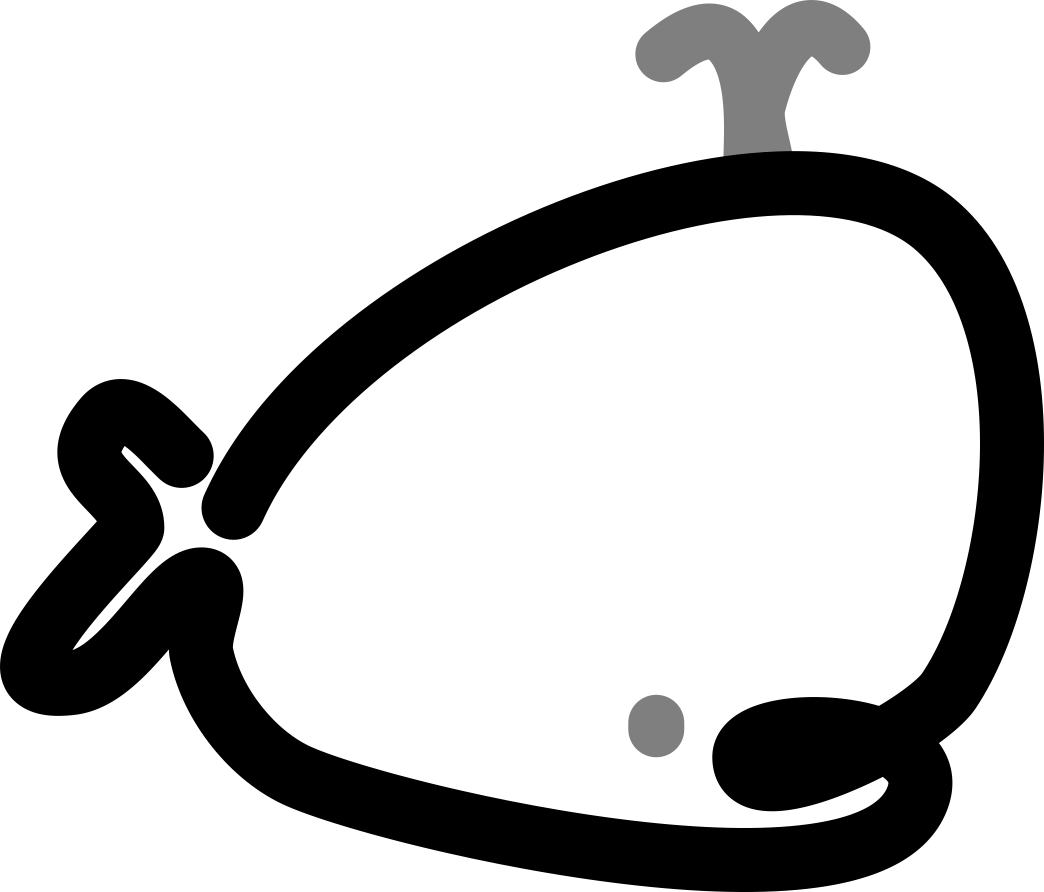
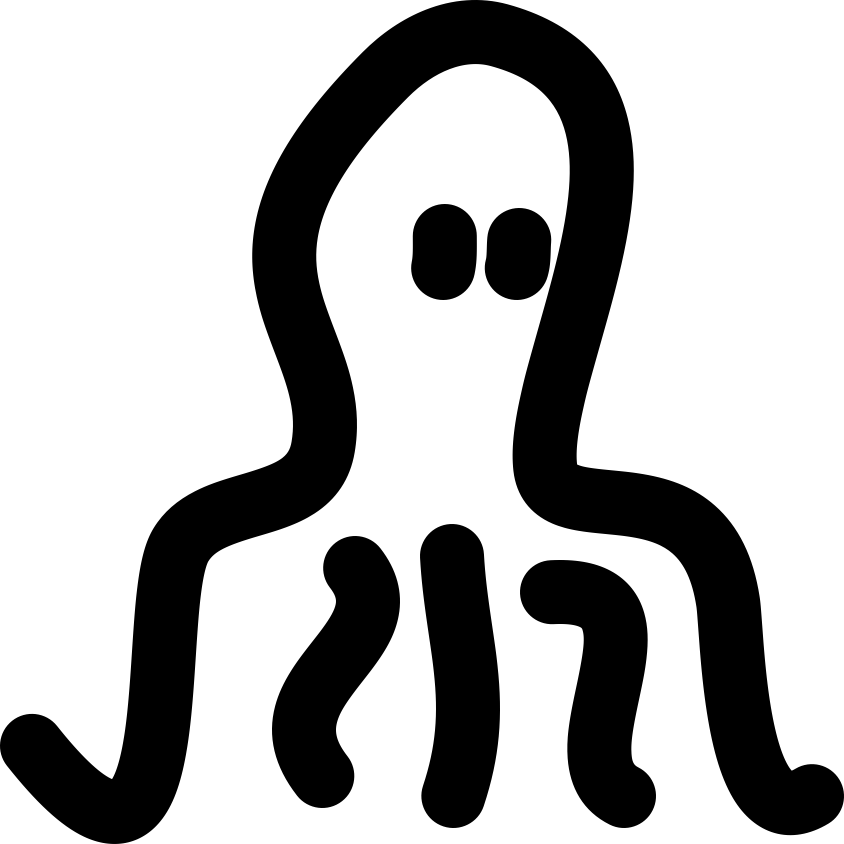
We will first look inside a single chain restaurant.

Note: We will look into each label technology further into the guide.





Kubernetes



Argo

Docker

Docker

Docker

Docker

A picture containing shape

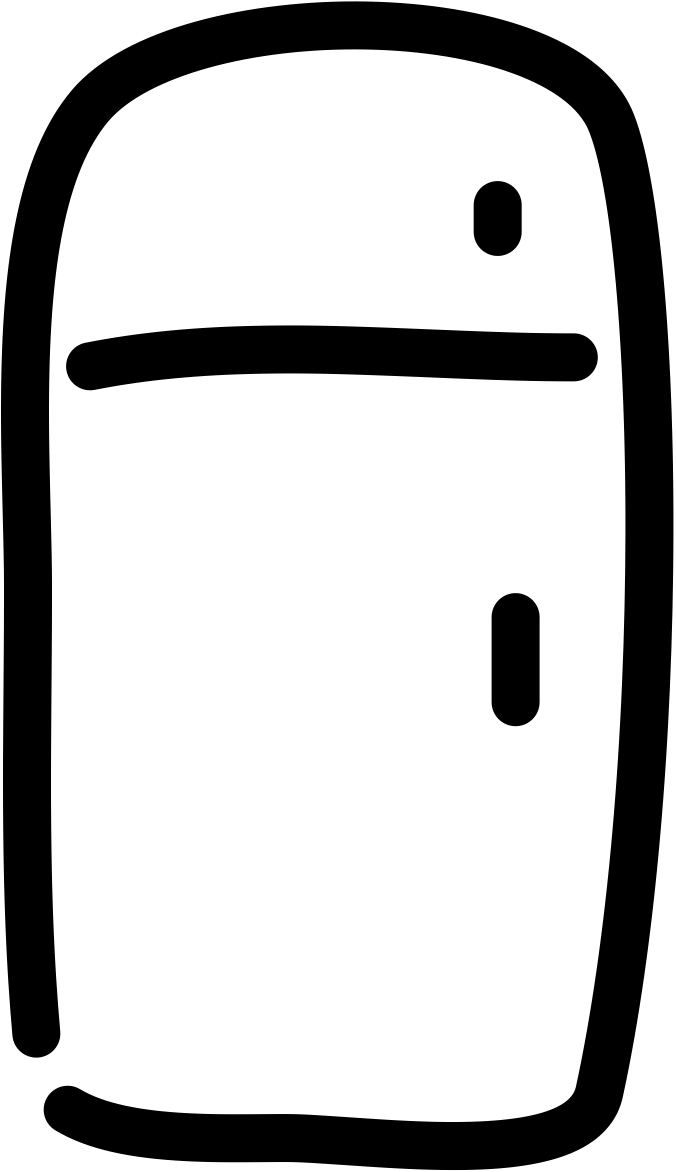
Description automatically generatedA picture containing shape

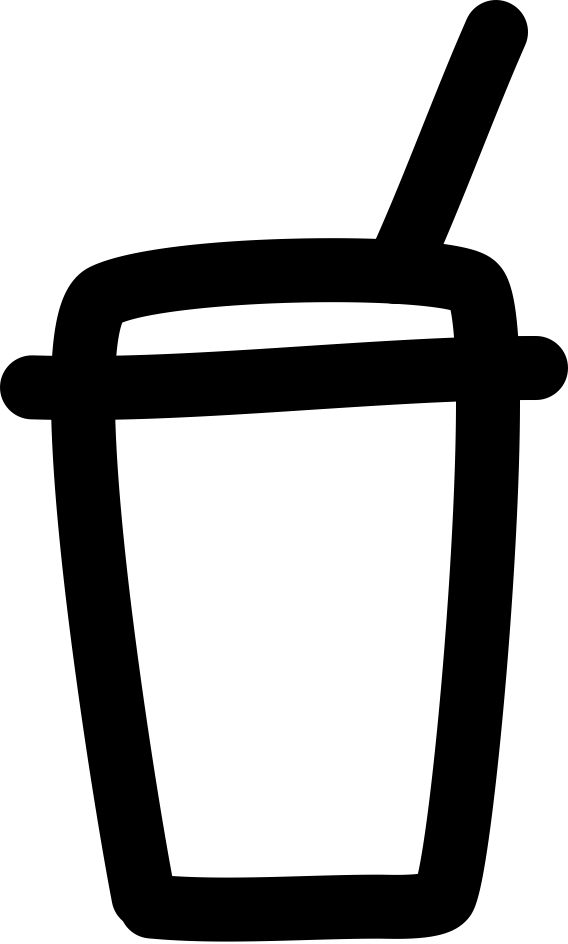
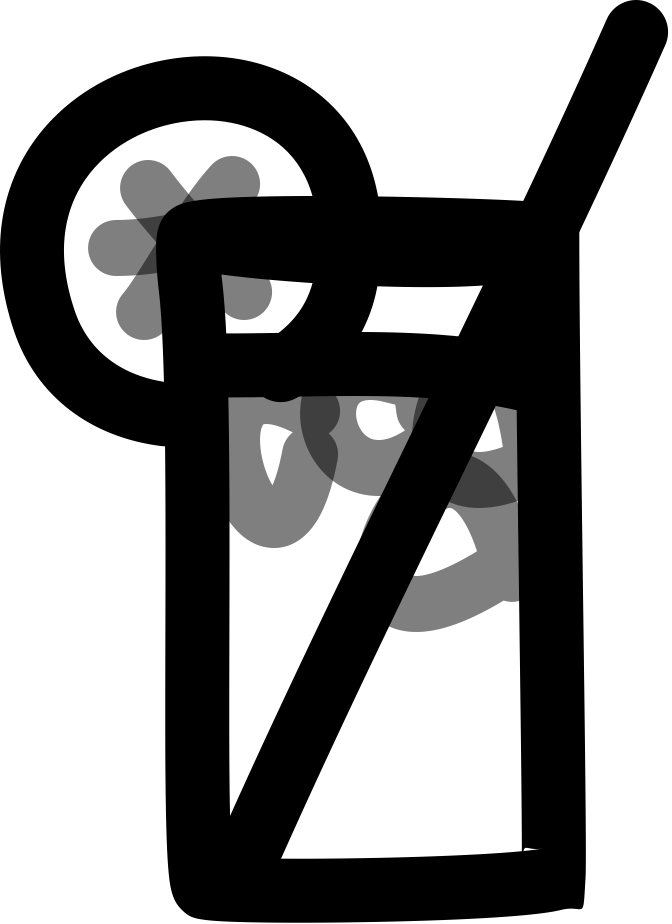
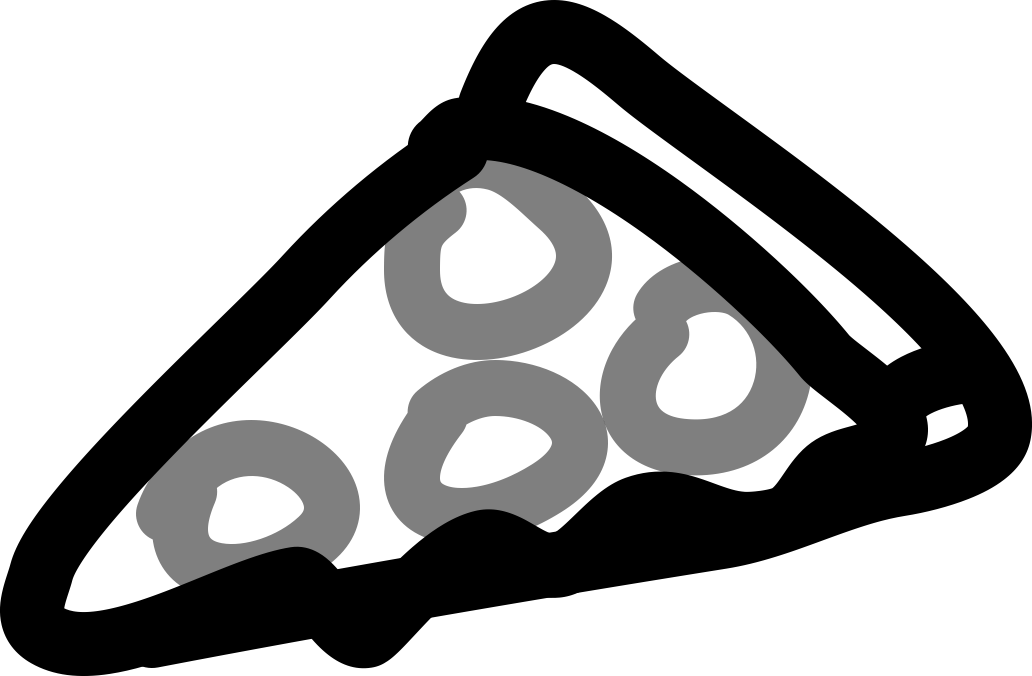
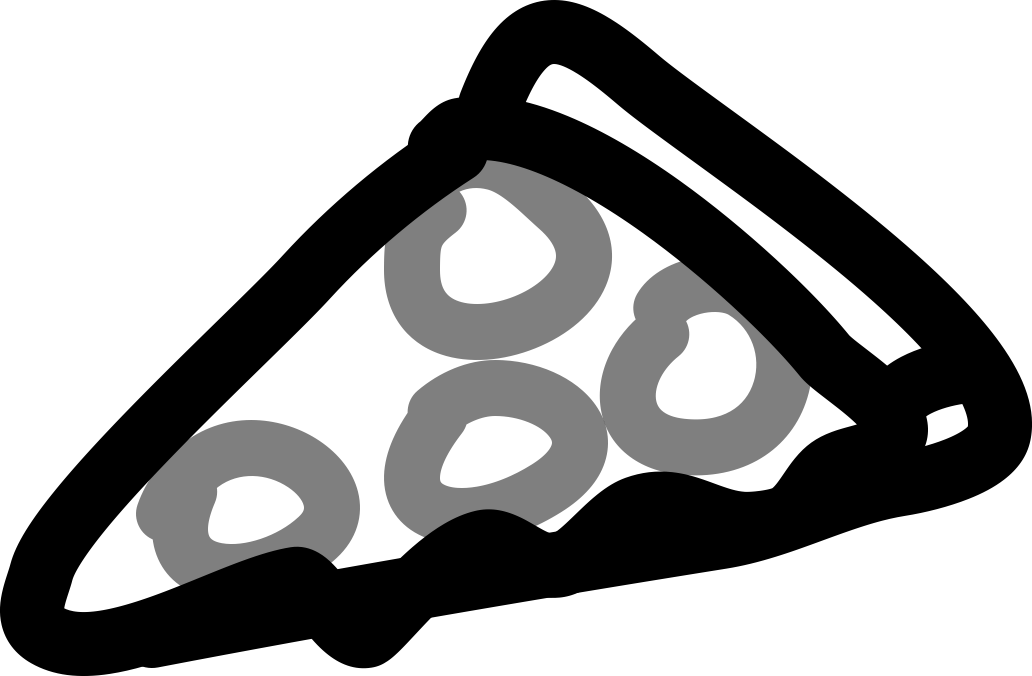
Description automatically generatedA picture containing shape

Description automatically generatedA picture containing shape

Description automatically generatedA picture containing shape

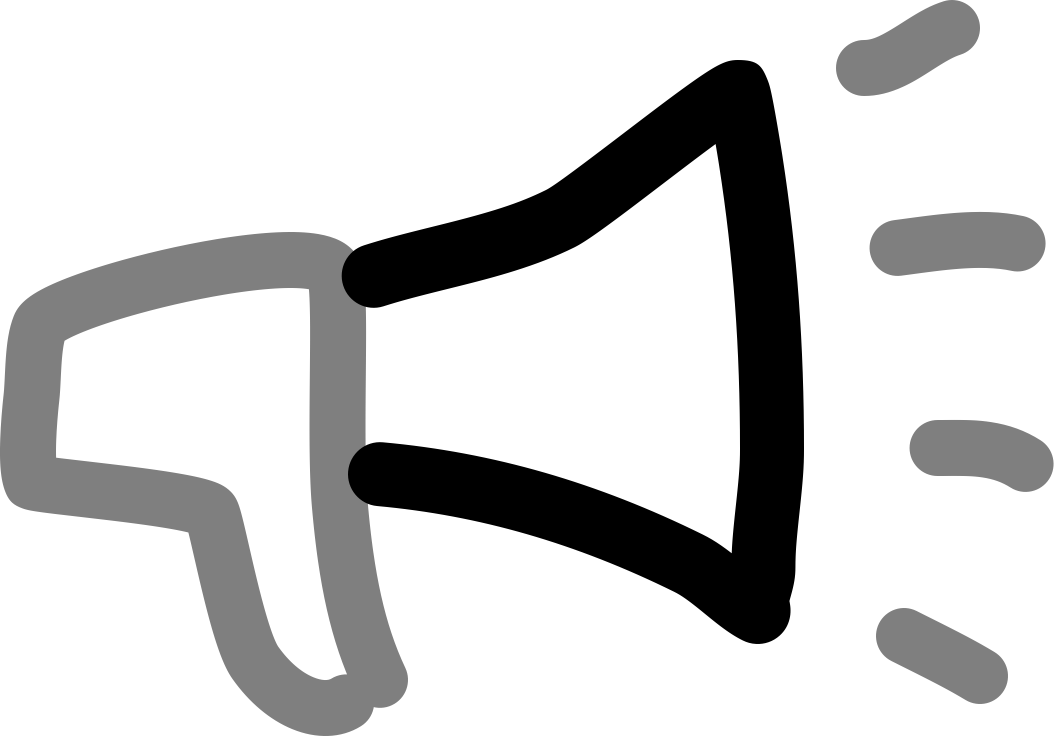
Description automatically generated





Order

Communication/interactions between containers are handled by message queues, RPCs, APIs, sockets, etc.



* Services, Specialized Programs

Each worker specializes in a specific role in the restaurant.

* Container Orchestration, Kubernetes

The manager of the restaurant keeps track of everyone working in the restaurant and schedules work hours according to busy hours and employee availability.

* Workflow Orchestration, Argo

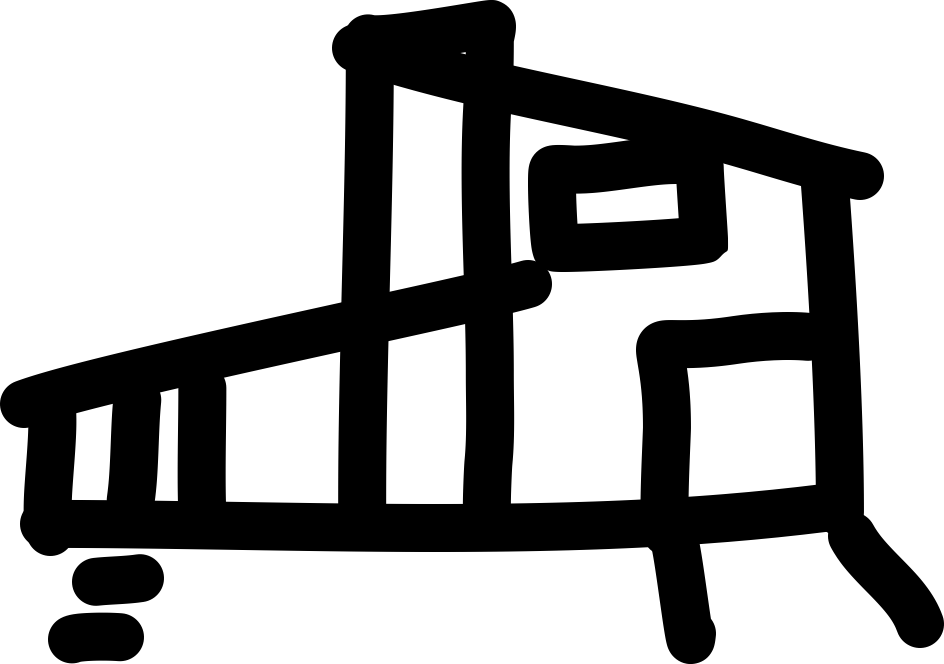
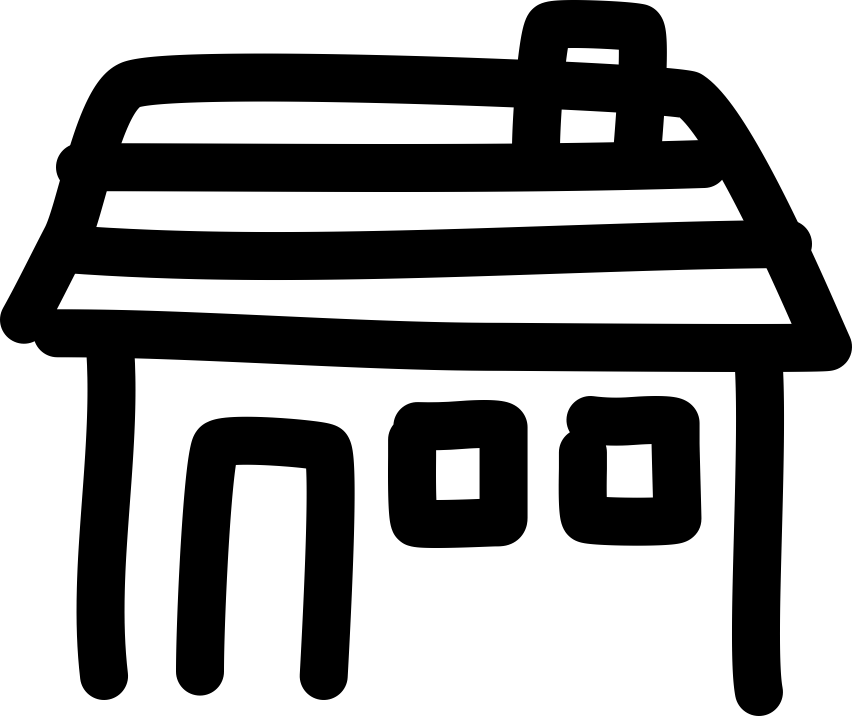
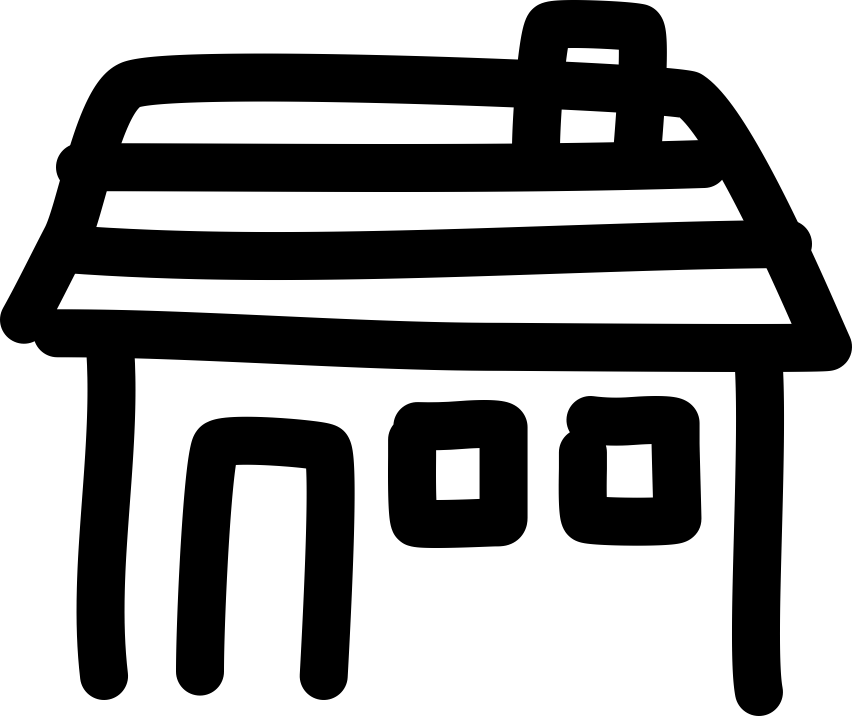
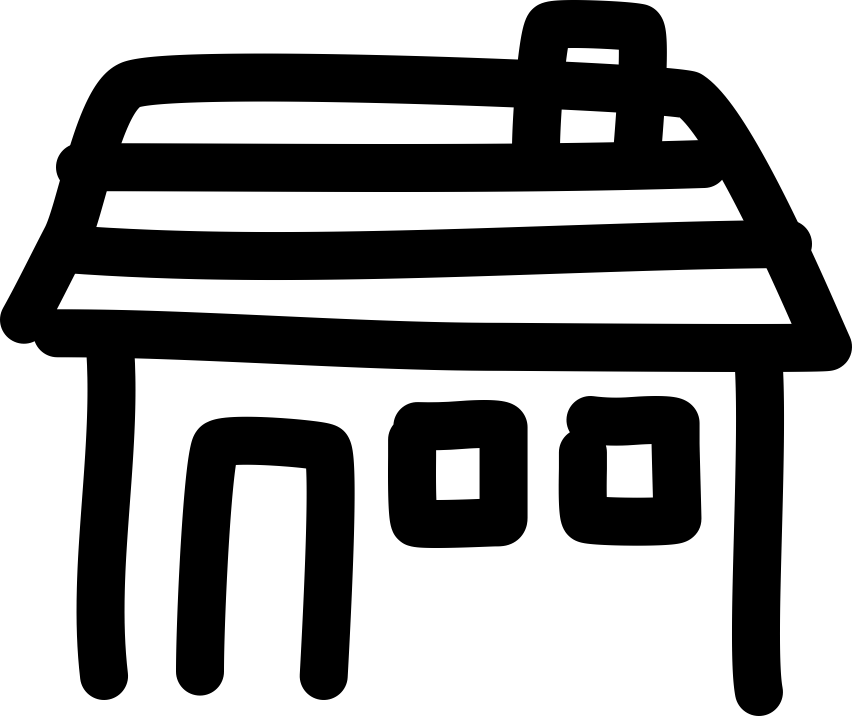
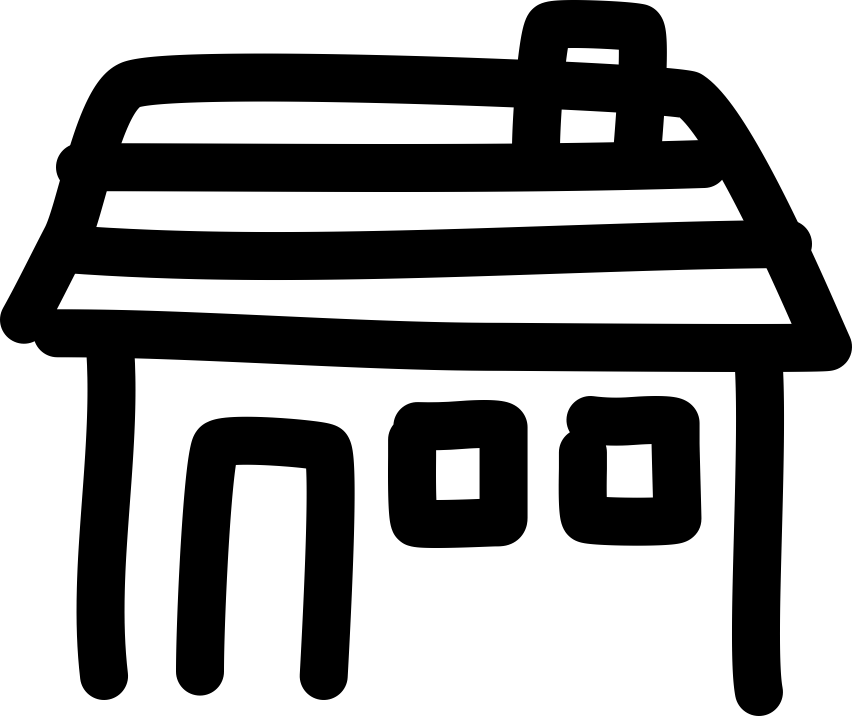
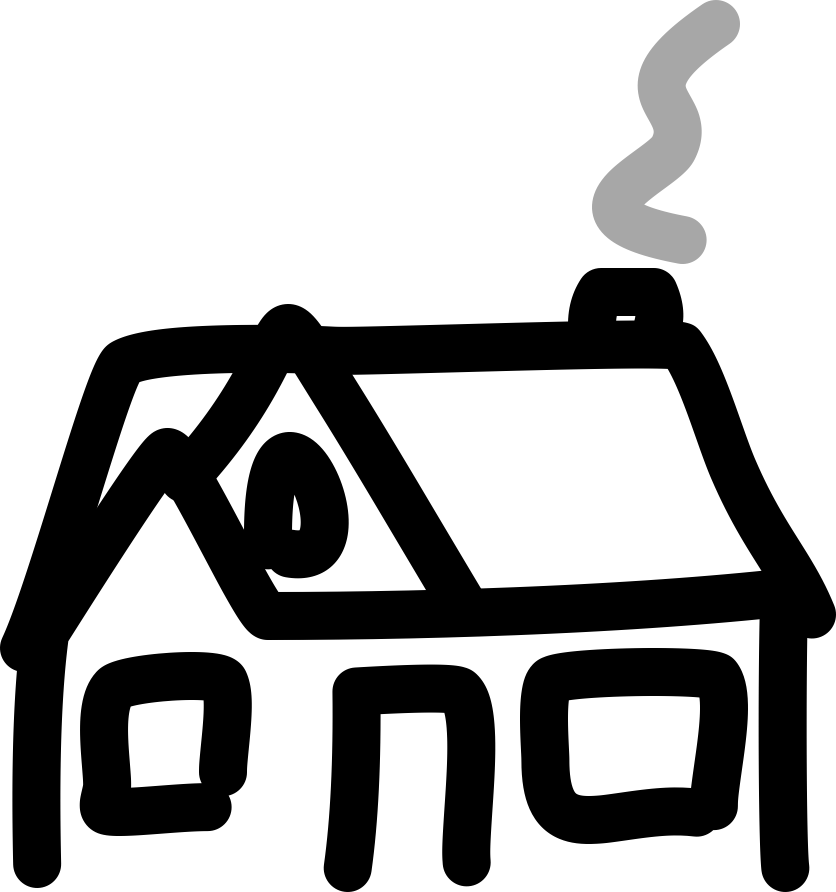
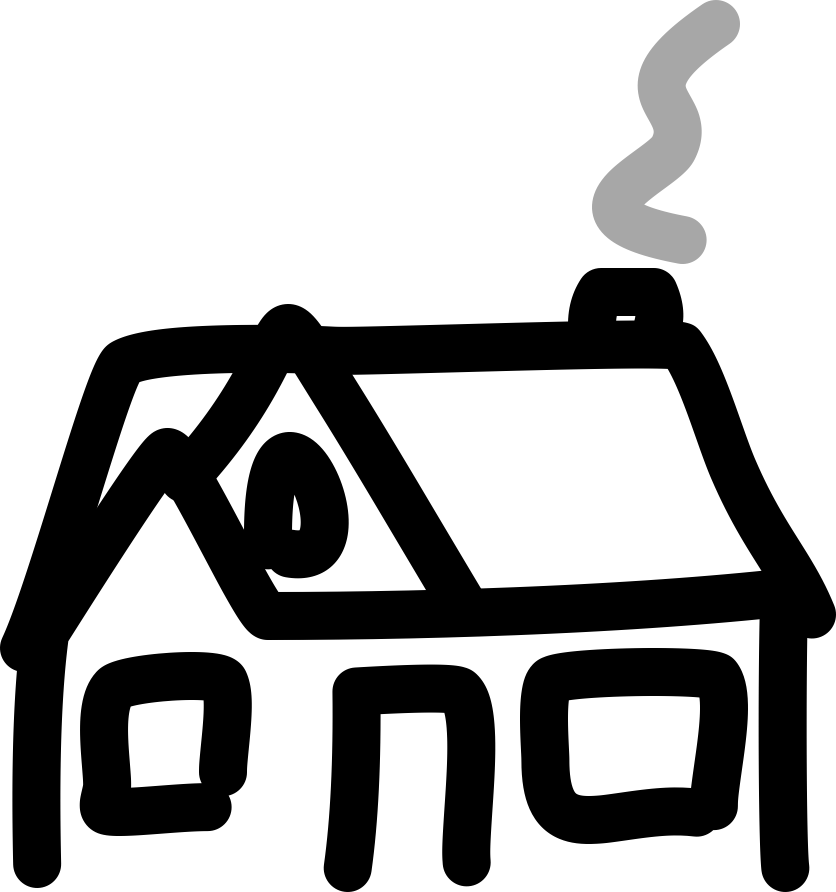
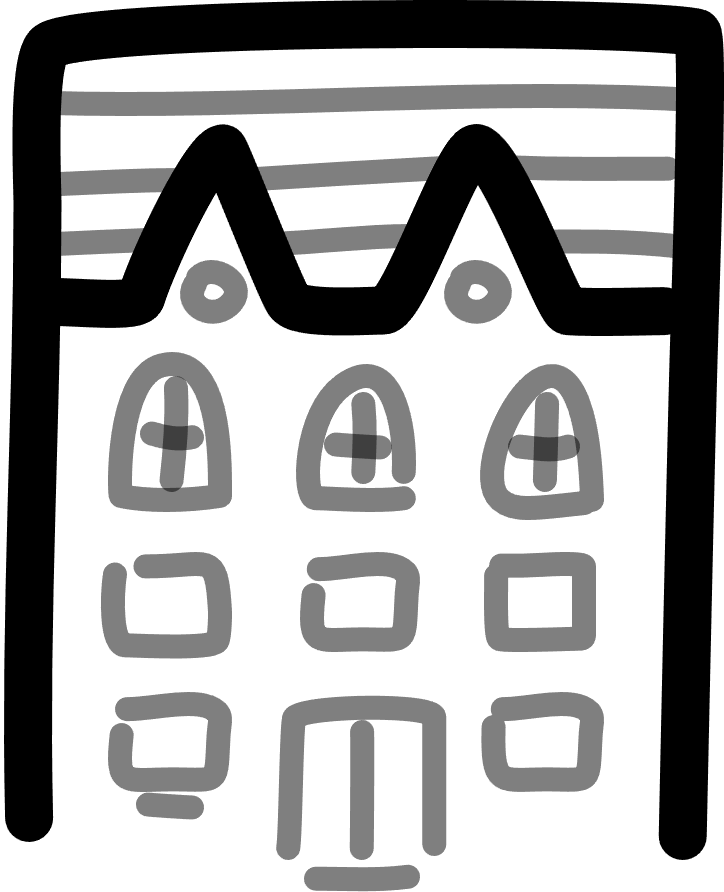
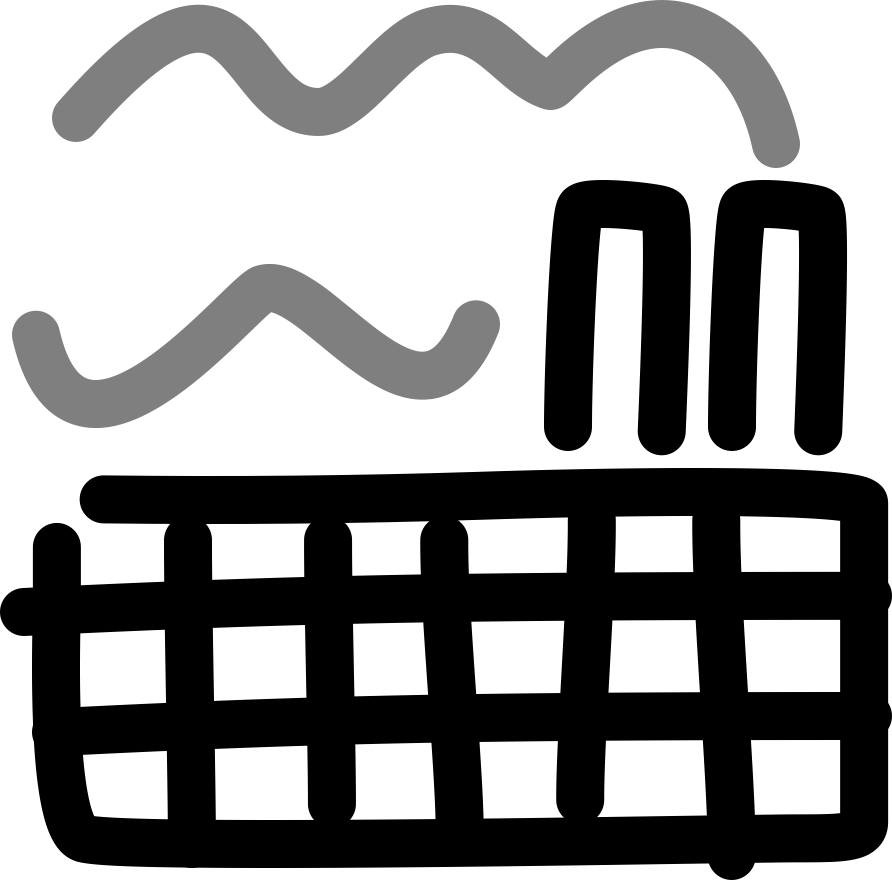
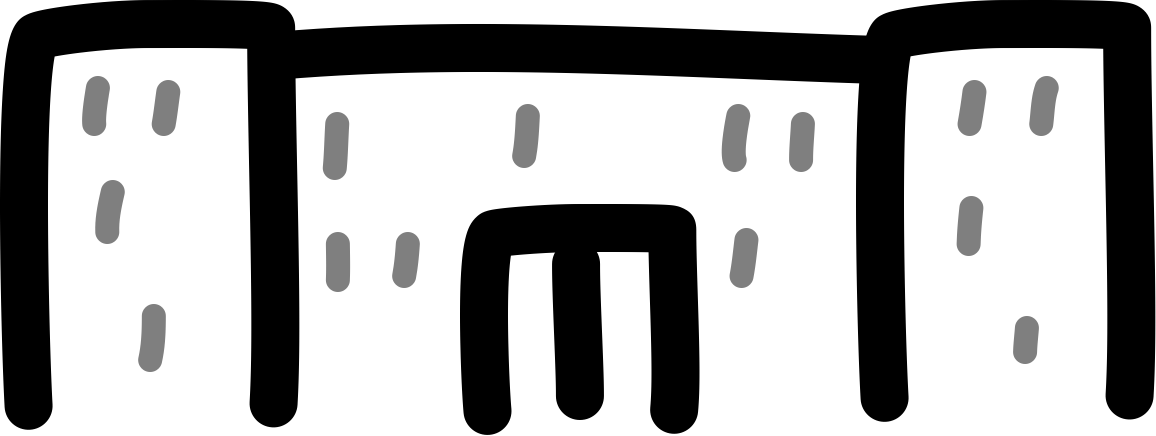
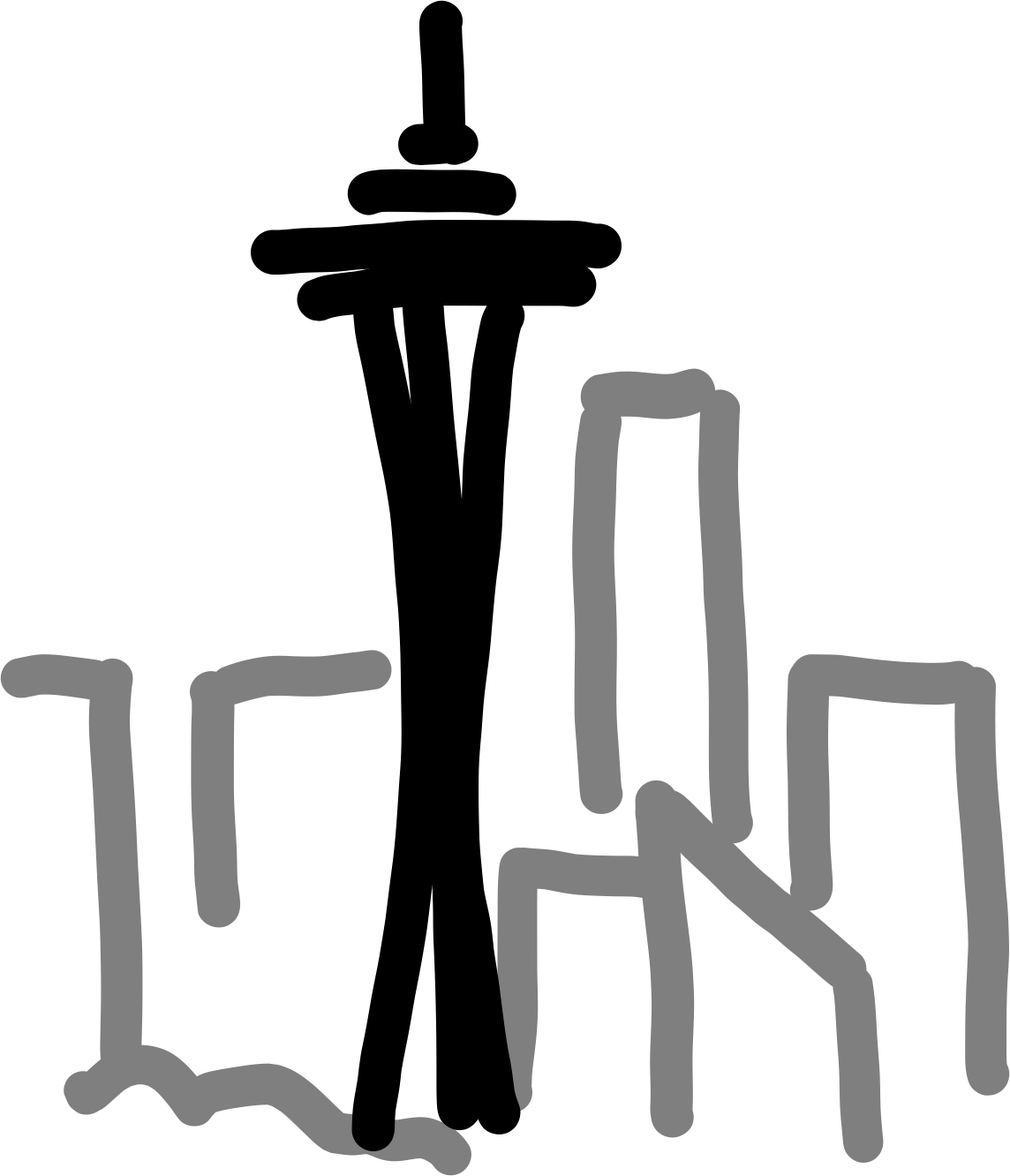
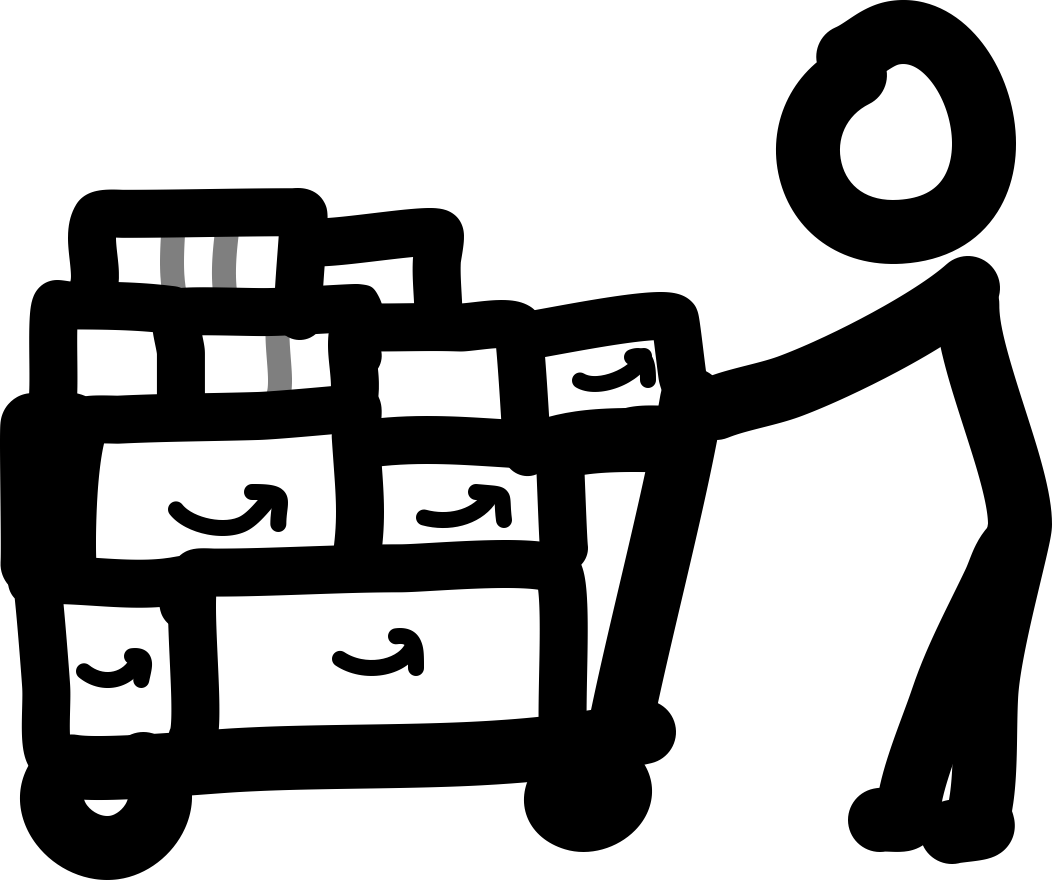
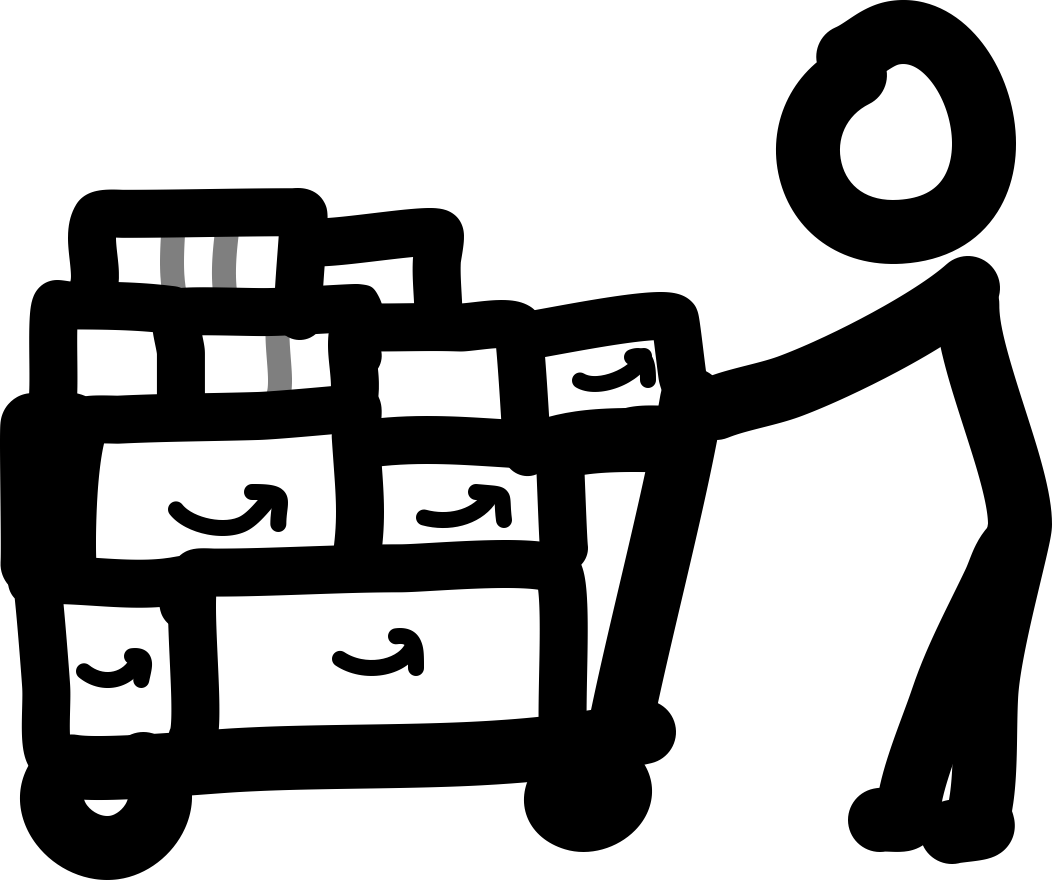
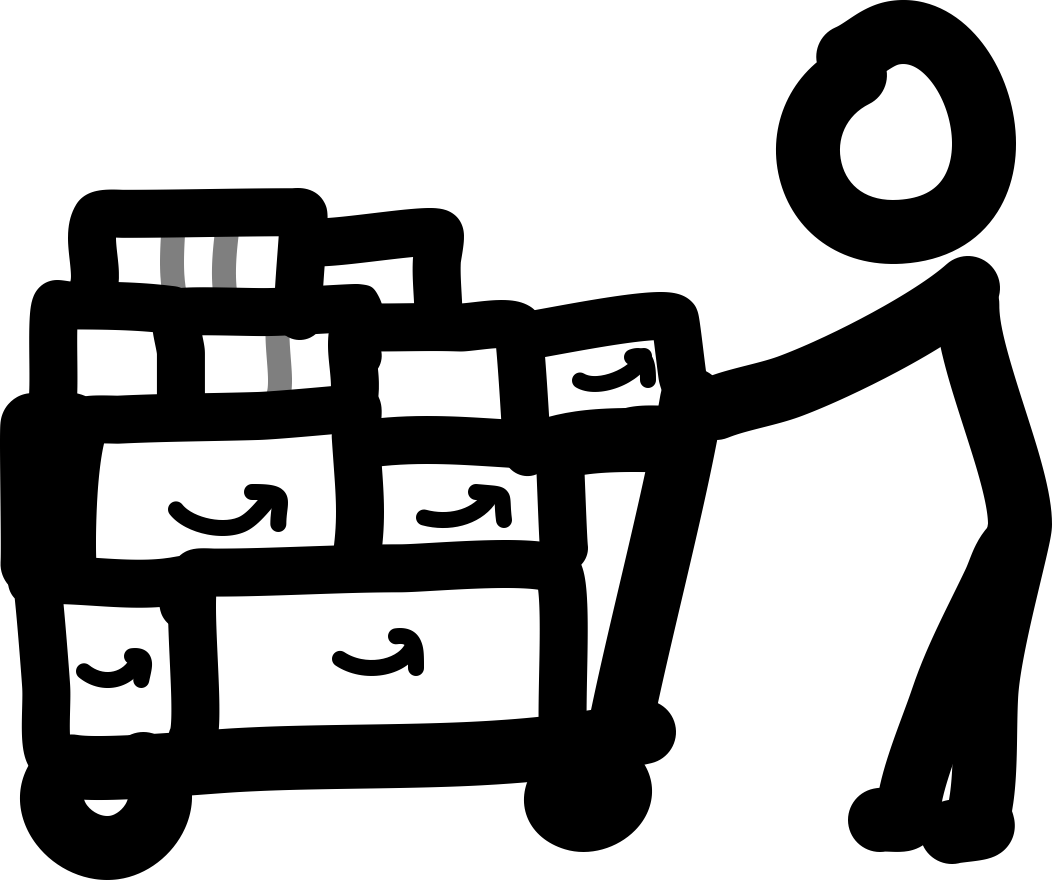
The cashier takes orders and relays them to the chefs, including different requirements for each order. Eg. no cheese, ham and tomatoes.

At a macro level, a pizza chain also represents a distributed system.

Logo

Description automatically generatedLogo

Description automatically generatedLogo

Description automatically generated

Pizza Guy

Pizza Guy

Pizza Guy

Pizza

Pizza

Pizza

Pizza

Should a restaurant in the chain burn down, the next nearest chains relative to the customer will take over service for the area.

The analogy can essentially be stretched forever due to the supply chain for pizza ingredients and productive systems in general also working in this way.

# Containers

## Introduction

Containers are an instance of an image, much like virtual machines (VM), which simulate the required environment with the use of a Linux kernel packaged in it. It is helpful to think of containers as extremely lightweight versions of VM technology.

So how does it do this? Let’s imagine that you have written a program in Python 3.7.7, your app only needs Python 3.7.7 and dependant libraries you may have used to run, everything else like a GUI is pointless overhead. So, Container Software provides a template built on the Linux Kernel with the needed dependencies of the program only and nothing else. This template is called an image.

So, if you were to fetch a Python 3.5 image from a repository and run an instance of it, you can do anything you were able to do in the host machine using a command line interface. Any mistakes or changes made in that instance is completely isolated, to start anew, just delete the instance and create a new one.

This way, your main environment remains intact in the form of an image and you can experiment and play around with the dependencies packaged in the image using containers.

There are many open source adopters of Containerd (The project that spawned containers) such as Docker, Eliot, Katacontainers, IBM Cloud Kubernetes Service, etc.

Willy-todo: maybe delve into other adopters, probably not.

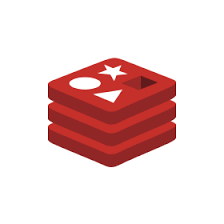
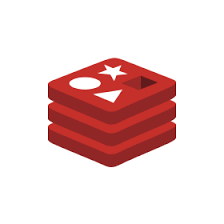
## Docker

### Introduction

Docker manages containers and images using a client-server architecture. We will take a look into its eco-system below:

* **Docker Registry** : Docker maintains all the images in the registry, they can all be pulled at any time, these are often base images of popular OS’s and software.
* **Docker Hub** : This is the repository for all your custom-built images, images can be pushed and accessed from the hub, it is essentially GitHub but for containers.
* **Docker Client** : The CLI tool used to interact with the Docker server.
* **Docker Daemon** : The Docker server process responsible for pulling, pushing, and building images. It is also used for running the container.

A picture containing shape

Description automatically generated

Images

Containers

Docker build

Docker run

Docker pull

Docker daemon

Registry

Client

DOCKER\_HOST

In the diagram above, you can see that the docker client asks the daemon to pull a Redis image from the registry. Also it asks the daemon to build and run the ubuntu image.

The docker client can be used to communicate with the Docker daemon present on the host machine or on any remote machine as well.

Text

Description automatically generatedDocker provides a hello-world image that illustrates these steps, as shown in the picture below.

There are only three things the user has to keep in mind while working with Docker:

1. Images
2. Containers
3. Networks – relevant with Docker Compose

### Layers

Layers, the building blocks of Docker, on the previous page, we saw that Docker pulled a single image “hello-world” using only a single layer. This could be done because the “hello-world” image was very small.

A picture containing text

Description automatically generatedShould we pull bigger images, we would get something like this.

Notice how it pulled three layers? The reason is that whenever Docker pulls an image, it takes up a lot of space on the device. This can be seen from running “docker image ls nginx”.

The Nginx image took up 126MB, if images were built as a standalone image with no Component Sharing, every image would take a .iso size of memory like a VM, which is extremely inefficient.

This leads us nicely into Component Sharing, an extension of object oriented software design. It takes the object oriented concept of reusable pieces of generalized code and applies it to building images. Because each image is built on top of a Linux Kernel, there are common dependencies that can be reused by other images. Docker bundles these dependencies in one stack, these stacks are called layers.

Docker caches these intermediate layers to speed up the image building process, so no redundant downloads occur. We only download what is needed.

### Sharing Data

So far, all data is stored within the container, should the container be destroyed, all data in the container will also be gone. Therefore it is good practice to separate the data’s file system from the container’s file system.

Docker provides 2 methods to achieve this, Bind Mount and Volumes.

Bind Mount

Volume

Tmpfs mount

Docker Area

Filesystem

Memory

Container

Host

#### Bind Mount

With bind mount, the host’s file system is used and mounted on the container using the -v flag with the run command.

$docker run -it -v <absolute path>:<folder path in container> project:1.0

So if you replace <absolute path> with “/Users/YourName/Desktop/” and <folder path in container> with “/desktop” your host desktop folder will be mounted in the container as /desktop and can be accessed from within it.

#### Volume

Docker volume creates volumes in the docker space, which provides more direct control via the Docker CLI.

Text

Description automatically generatedVolumes can be created and managed via the docker volume cli tool as shown

Text

Description automatically generatedAfter creating a volume, we are unable to access the mount point in the picture. To access the volume, we need to mount it with docker run and the -v flag like before.

$docker run -it -v <volume name>:<folder path in container> project:1.0

#### Bind Mount vs Volume

Volumes are more reliable than bind mounts due to the fact that the host system is unable to directly access the volumes in the docker space. A user accidentally deleting a file on a host file system that is mounted and used in a container will be permanent.

### Dockerfile

This section will require some knowledge of DevOps, I will try to give a brief overview of the normal process of development pre docker below.

In the past, when a new developer would join, they would need a huge amount of time configuring their development environment, this involves pulling the code from a repository, installing the projects dependencies locally, and then dealing with any incompatibilities with their systems. The process for deployment was also the same.

Docker removes and automates these processes via a config file called Dockerfile

Text

Description automatically generatedThe example below is based on a simple javascript todolist project, the details for each command will change depending on languages, dependencies and frameworks used.

Docker isolates the code environment specified in the Dockerfile from the host machine using containers and the containers are the instances of the image that is built.

A Dockerfile is the blueprint Docker uses to build an image of your app.

#### Dockerfile Commands

Notice how every line starts with a red word in all caps? These are the commands docker have defined, we will go through the main ones, an up to date explanation for these commands can be found in their documentation.

* FROM - This tells Docker which base image to use or what should be the main platform for this image.
* WORKDIR - This sets a working directory for other Docker instructions such as RUN and CMD, if we do not specify a working directory, we would need to provide a full path for running our app while using the RUN instruction.
* ENV - This is an optional instruction used to specify various environment variables required for your app, such as when using flask, you may want to set the host to 0.0.0.0, which allows us to access the app using any IP within the container.
* COPY - The copy instruction literally copies the file from one location to another, COPY SOURCE DESTINATION is the syntax.
* RUN - This will execute any commands in a new layer on top of the current image and commit the results. The resulting committed image will be used for the next step in the Dockerfile. This command creates a new layer and runs the commands provided as arguments in that layer.
* CMD - This runs a command inside the container once a container is forked or created from an image. You can only have **one** CMD instruction in a Dockerfile. If more CMD instructions are used, only the last one will be executed.
* ENTRYPOINT - This can be used if to you want to configure your container as an executable. If you want to override CMD while running a container, use ENTRYPOINT

Eg. ENTRYPOINT[“node”,”src/index.js”]

### Building An App Image