# Group work report

During the group work session, I was assigned to group 1. Right off the bat it was obvious we had people with varying characters, methods of working and areas of expertise. Everything didn’t immediately work well together but I got some valuable insight from the experience.

We started the day by installing some Linux virtual machines. It was decided that Ubuntu would be the distribution for us, mainly due to it being easy to set up and time was of the essence.

Our group cohesion was not exactly on point. A couple of our members immediately began working on a version of the project. Some others wanted to make a plan of what we wanted to accomplish, decide on technologies and ponder how we wanted to do it all. I was on the latter group.

We had a good discussion and came up with a basic UML graph of the product. Our front end would send an image and receive a result from a microservice. Microservice would return a result that would be shown to the user. User could then send the next steps to different microservices and in the end, receive an audio file that could be played.

A diagram of a diagram

Description automatically generated

This is mostly what we got done. The rest of the class I spent helping the more active group members with compiling their dockerfiles and so on. At the end of the class we had a closing meeting and discussed how the class went. All in all the planning and closing were good.

For the next class I would maybe make the group sizes smaller. It was confusing and difficult when we essentially had 2-3 different groups in one.

I provided a github link in Moodle. This is the version I made before the course. During the session I learned several things that I did wrong with the project.

1. My initial take was a distributed monolith. I was struggling to think what benefit my approach had. The biggest problem was that one microservice called the next one. This caused a host of issues like sharing a resource, causing change in one microservice caused issues with the next one and overall difficulties deploying the product. A microservice should be usable on it’s own, a perfect small application doing just one thing well.
2. A resource intensive CNN could eat a lot of processing power, so it is critical that those resources are not wasted. Circuit breaker pattern could be implemented to help save resources in case it starts failing. CBP could for example have an incrementing number of failures in a microservice. If the service has enough failures, a timer is started and any calls to the service immediately returns an error message. After the timer runs out, a portion of the requests are let through. If these requests are successful, the service can be allowed to accept more requests. If not, the service begins to respond with mere error messages once again until the problem is resolved.
3. Flask API’s quick and dirty rest API works well for applications that request and receive information from the server. Flask might not be able to do everything that is needed, or the API might be too slow for production. In this situation sockets might the correct alternative. Sockets allow for more control over your applications communication. If you wanted to implement the CNN camera application from moodle over the internet, websockets could be considered.