Part 2:

1. We ran the monolithic implementation of the HotelMap app to get some baseline metrics so we could evaluate and compare with the microservices implementation of the app. We also modified the supplied LUA script such that it would only make requests for the search functionality of the app and ignore any other unimplemented functionalities. Furthermore, we used a newer version of the wrk which allowed us to get more accurate results for the 99.999th latency percentile and keep the throughput constant. We chose a fixed number of threads for the workload generator @ 10 threads since it was a good balance between the available cores of the machine we were using and the handling of the connections. Each test was left to run for 30 seconds. 10 tests were made, each with increasing number of connections (100-1000, increments of 100).

In the microservices implementation of the app, we broke down each major component into a microservice running in a different container.

As we can see, the implementation of the app with microservices results in a significant increase in the latency. Even at a low number of connections, the latency is almost 10 times greater. At 500 connections, we observed a peek at around 160ms @ 99.999th%. In general, we observed very high latency figures on the 99.999th percentile except in the cases of 700 and 900 connections. A possible explanation of such behavior is that the communication latency between each microservice adds up to the overall latency of the app.

Graphical user interface, application

Description automatically generatedThis is a screenshot taken from Jaeger which we used to monitor the gRPC. We can see that the microservice “frontend” takes the most time to complete. In this example we can see the invocation of the microservice “frontend” which in turn called the “geo” microservice and then the “profile” to get the profile for the requested “geo” location. The two circled regions show the time spent in the RPC while doing nothing.