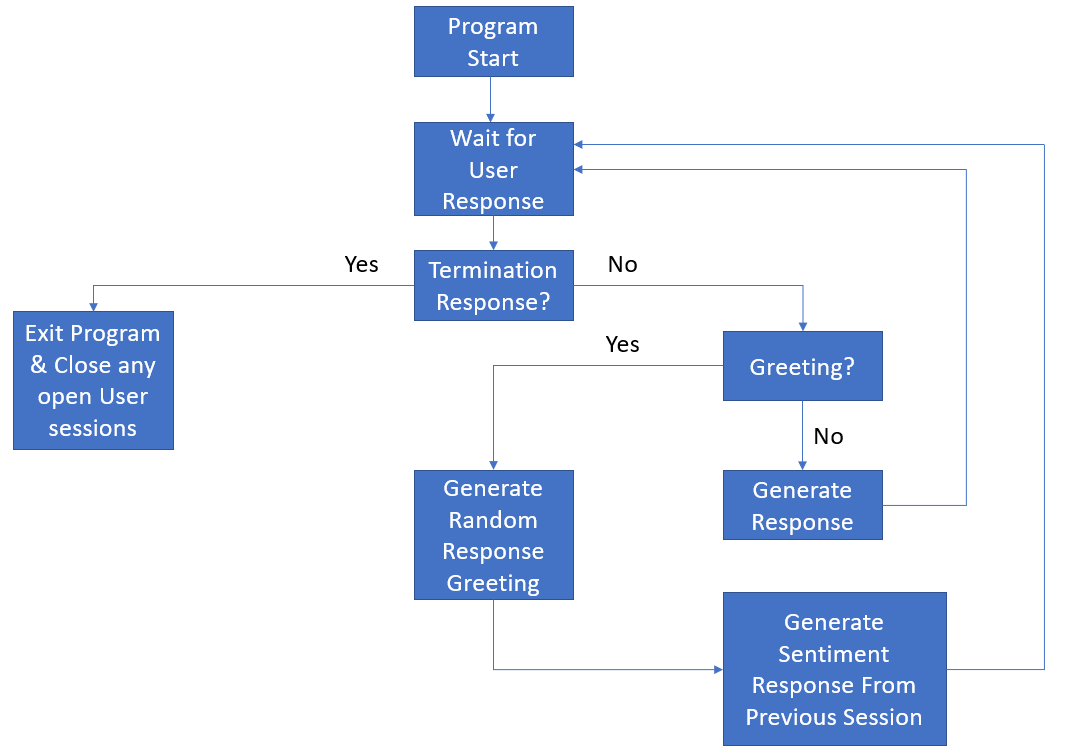
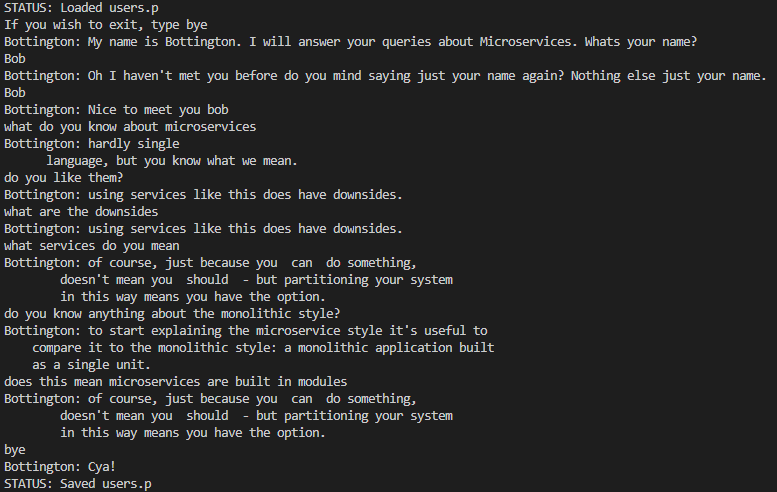
Microservices Chatbot Report & Evaluation

# By: Anthony Chen

My Microservices Chatbot named Bottington uses several NLP techniques to generate the User’s Input and attempt to respond in a convincing manner. The system parses user responses and uses term frequency and sentiment analysis to determine how it should respond.

**Logic Diagram:**

**Sample Dialog:**

[**Knowledge Base Appendix:**](https://github.com/Lrogan/CS4395.001-Repository/blob/main/Portfolio_Assignment6/chatbot.txt)

Consists of articles scrapped from the web, lightly scanned and cleaned for processing

Here are some Samples:

Why a robust CI/CD pipeline matters

In a traditional monolithic application, there is a single build pipeline whose output is the application executable. All development work feeds into this pipeline. If a high-priority bug is found, a fix must be integrated, tested, and published, which can delay the release of new features. You can mitigate these problems by having well-factored modules and using feature branches to minimize the impact of code changes. But as the application grows more complex, and more features are added, the release process for a monolith tends to become more brittle and likely to break.

Following the microservices philosophy, there should never be a long release train where every team has to get in line. The team that builds service "A" can release an update at any time, without waiting for changes in service "B" to be merged, tested, and deployed.

To achieve a high release velocity, your release pipeline must be automated and highly reliable to minimize risk. If you release to production one or more times daily, regressions or service disruptions must be rare. At the same time, if a bad update does get deployed, you must have a reliable way to quickly roll back or roll forward to a previous version of a service.

Challenges

Many small independent code bases. Each team is responsible for building its own service, with its own build pipeline. In some organizations, teams may use separate code repositories. Separate repositories can lead to a situation where the knowledge of how to build the system is spread across teams, and nobody in the organization knows how to deploy the entire application. For example, what happens in a disaster recovery scenario, if you need to quickly deploy to a new cluster?

Mitigation: Have a unified and automated pipeline to build and deploy services, so that this knowledge is not "hidden" within each team.

Multiple languages and frameworks. With each team using its own mix of technologies, it can be difficult to create a single build process that works across the organization. The build process must be flexible enough that every team can adapt it to their choice of language or framework.

Mitigation: Containerize the build process for each service. That way, the build system just needs to be able to run the containers.

Integration and load testing. With teams releasing updates at their own pace, it can be challenging to design robust end-to-end testing, especially when services have dependencies on other services. Moreover, running a full production cluster can be expensive, so it's unlikely that every team will run its own full cluster at production scales, just for testing.

Release management. Every team should be able to deploy an update to production. That doesn't mean that every team member has permission to do so. But having a centralized Release Manager role can reduce the velocity of deployments.

Mitigation: The more that your CI/CD process is automated and reliable, the less there should be a need for a central authority. That said, you might have different policies for releasing major feature updates versus minor bug fixes. Being decentralized doesn't mean zero governance.

Service updates. When you update a service to a new version, it shouldn't break other services that depend on it.

Mitigation: Use deployment techniques such as blue-green or canary release for non-breaking changes. For breaking API changes, deploy the new version side by side with the previous version. That way, services that consume the previous API can be updated and tested for the new API. See Updating services, below.

**Appendix - User Models:**

Users are saved by their name, which is then the key for a dict that contains a User Object that saves their likes, dislikes, neutral statements, and the last time they chatted with Bottington.

User : anthony

Likes: ['i love microservices', 'microservices are pretty cool', 'do you enjoy microservices']

Dislikes: ['i hate monolithic', "monolithic isn't that great"]

Neutrals: ['what do you know about architecture', 'computer science is cool', "how do i know you're real", 'yes they probably are', 'what is dropwizard?', 'do you know anything about the monolithic styel?', 'is there anything you want to say?', 'what do you mean by module']

Days Since Last Chatted: 0

User : bob

Likes: []

Dislikes: []

Neutrals: ['what do you know about microservices', 'do you like them?', 'what are the downsides', 'what services do you mean', 'do you know anything about the monolithic style?', 'does this mean microservices are built in modules']

Days Since Last Chatted: 0

Evaluations:

The chatbot overall functions exactly how you would expect a chatbot to behave however there are several improvements that could be made. The main improvement to be made is to expand and modify the knowledge base so that more coherent sentences are formed. Another Improvement to make is to expand upon the sentiment analysis structure of the bot so that it can more accurately and confidently interweave emotional responses into the conversation. Some strengths it has is that it is quite lightweight and doesn’t require huge amounts of computing power to be able to process the input and generate an output in a reasonable amount of time. The other strength is that the knowledge base is easily expandable and modified by merely changing the .txt file.