# CSCI 370 Final Report

Jinja Ninjas

Ian McDaniel
Gavin Gilmore
Brady Ancell

Revised June 16, 2024


CSCI 370 Summer 2024

Mr. Bartel

Table 1: Revision history

| Revision | Date | Comments |
|---|---|---|
| New | 5/14/24 | Created requirements document as well as filling in information to the best of our ability. |
| Rev – 2 | 5/19/24 | Updated requirements/description after meeting with the client |
| Rev – 3 | 6/09/24 | Added results, future work, and lessons learned sections. Updated earlier sections. |
| Rev – 4 | 6/11/24 | Added Team Profile |
| Rev – 5 | 6/12/24 | Fixed typos, updated database and infrastructure diagrams, and added Acknowledgements |
| Rev – 6 | 6/16/24 | Final revisions |

# Table of Contents

## I. Introduction

Dish, being a very large company, currently utilizes a variety of internal microservices to send and receive important information to employees. However, Dish currently communicates the results of these microservice requests via email inboxes and/or google drive files. This has proved to be slow and inefficient for Dish. To solve this, our team created a functional base frontend in order to automate the usage of microservices as well as services that could potentially be added in the future. In order to accomplish this we created various components of the system with dynamic templated panels using: Python, Flask, Jinja, Bootstrap CSS and Javascript. These templates are used for both a user panel and administrator panel across microservices. Our team's product is flexible and focuses on primarily templatizing for future backend integration.

## II. Functional Requirements

Our team is tasked with creating the following panels:

I. Admin Panel
   A. Can manage user permissions to microservices
   B. Can convert Swagger documentation for a microservice into a user friendly HTML form
II. User Panel
   A. Can generate a list of past service calls for each microservice for the user
   B. Can allow the user to make a request to an API, and then display result
   C. Has a home page that allows users to select among microservice they have access to
   D. Has a home page that displays a descriptive card for new microservices

## III. Non-Functional Requirements

Our system should utilize the following programs/libraries as per our client's requirements:

I. Jinja
II. Flask
   A. Flask-Admin
   B. Flask-SQLAlchemy
   C. Blueprints
III. Bootstrap CSS
IV. Javascript

## IV. Risks

I. We have found communication with the client slightly confusing and difficult, so a risk for us was not fully understanding deliverables and not seeing eye-to-eye with the client. However, as time progressed, this concern dissipated and is no longer a risk.
II. We don't know what the template is being used for, it may be used with sensitive data stored on Dish's servers; thus there may be some security risks. The client assured us that authentication is to be handled by the company, so we were not tasked with implementing it. However, if the application is not secure, third parties may be able to get their hands on sensitive data.

## V. Definition of Done

While there have been many slight adjustments and stretch goals added to our initial scope, the definition of done remains the same: a functional frontend that allows admin to define a microservice and its APIs and allows users to make requests to microservices and view the results. Though the list of requested features has

grown, the most important part of the project is functionality, not the style and user interface.

Features that have been added to the initial requirements are: displaying example payloads to users when making requests, allowing admin to test services before making them public, implementing metadata to microservices, displaying request results in an HTML table, allowing request results to be downloadable as JSON or csv.

There are also more features that were mentioned in the initial scope, but were much larger tasks than anticipated due to a lack of understanding (fixed from review). For example, the use of Swagger Docs for uploading new microservices, which is done by administrators, has been in our initial requirements from the start. However, we were unaware that this meant making our system compatible with multiple versions of Swagger and OpenAPI which meant more detailed implementation in our system.

These new features and newly discovered details have changed what we consider a fully functional product, but the general definition of done has stayed consistent.

## VI. System Architecture

### Technical Design Issues:

I. Lack of access to client resources and information
   A. No Access to test data
   B. “Hidden” features for testing
      1. Not understanding the types of microservices applicable
      2. Not understanding user authentication and full administrator roles

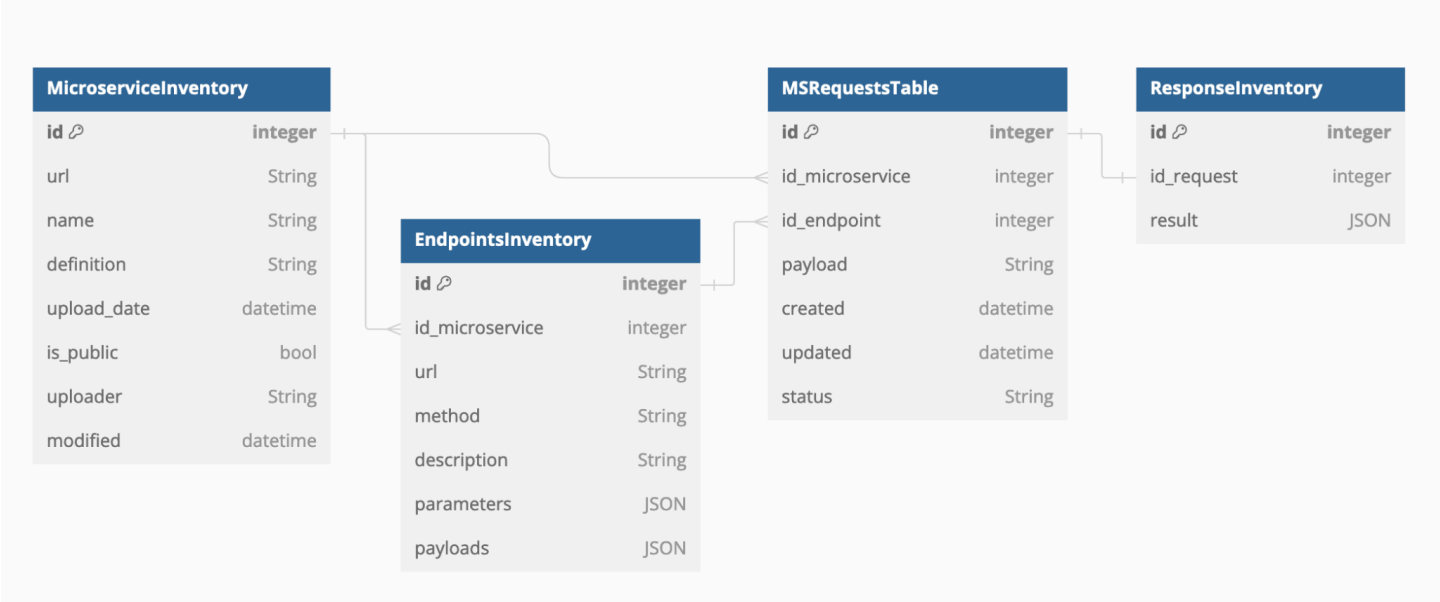
II. No standardized practices in Dish

Initially, most of our problems stemmed from a lack of clear understanding and communication between us and the client. For example, the details of the project were vague and we were not given clear examples of how our system should work. Though many of these concerns have been cleared, our testing seems less than satisfactory because we are unable to test our system with microservices that Dish actually uses.

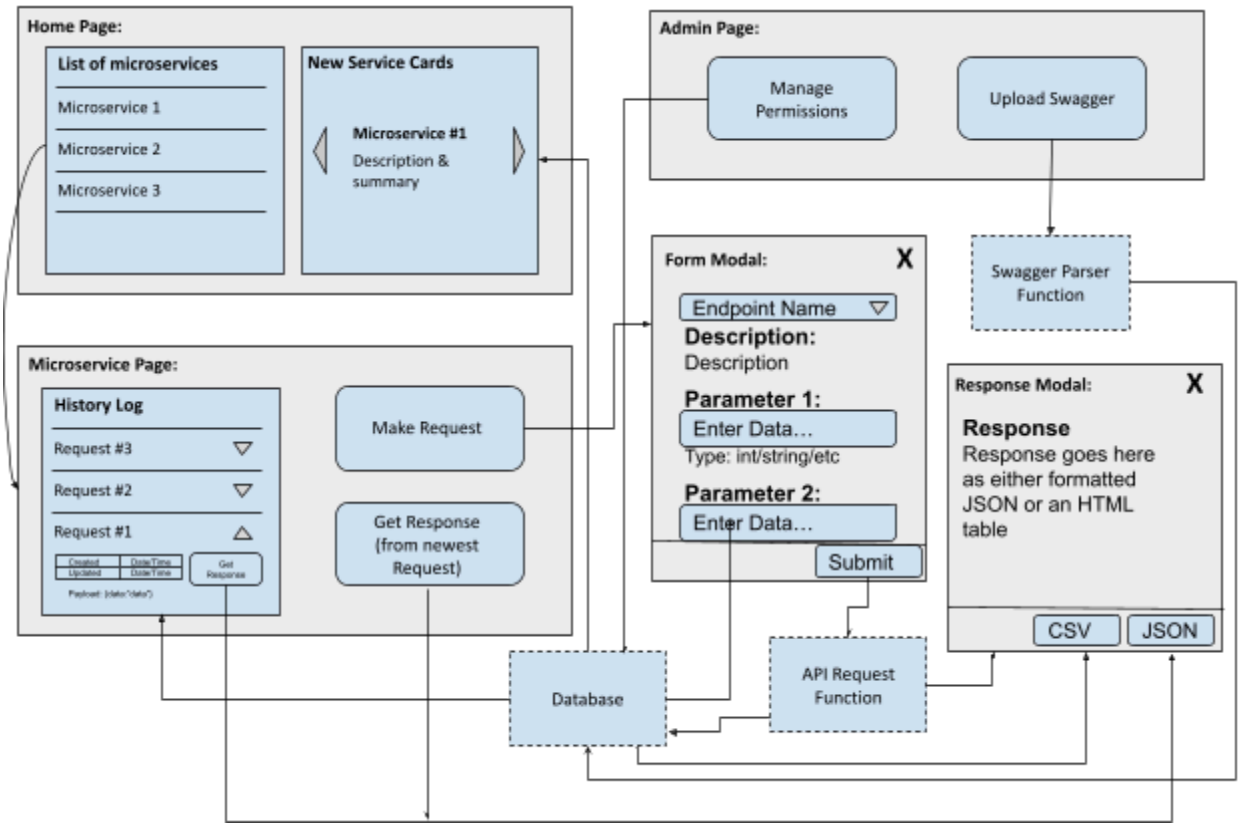
Moreover, we have had some technical issues with things not being standardized among Dish employees. This is mainly in regards to our implementation of Swagger documentation. Upon asking our client, we were told that some microservices use Swagger version 2, while others use version 3. This forced us to double the amount of work in order to make our system compatible for both.

# System Design Diagrams:

## Database Structure:



Infrastructure Diagram:




## VII. Software Test and Quality

Whilst the project revolves heavily around frontend development with a simple UI that is straightforward to test, there are a number of helper functions involved in site functionality that have been tested in order to ensure quality of software. Testing was also done based upon expected user and administrator sequence behavior. The following was tested (fixed from review):

I. User Operations:
   A. Proper available microservice(s) display on home page
      1. Test that when a service is marked as private/public visibility changes and is consistent with the database.
      2. Test that all information that is filled is accurate and up to date.
   B. Selecting available microservice directs to correct page
      1. Test that redirects to microservice pages properly load all information
      2. Test that microservice page is only accessible when a service is public
   C. Requests are properly fulfilled to microservice(s)
      1. Test to verify that requests are actually being made to microservice, API requests are not getting lost after form submittal.

2. Test that the output is properly retrieved from making a request as well as that this output is properly displayed.
3. Test to verify database contents after request completion.

D. History log properly displays requests.
1. Test that history log is updated live when a request is made.

E. General control flows testing.
1. Test that user flow is 'intuitive' and easy to follow for clients. This is done via user acceptance testing regarding multiple tasks (i.e. adding a service, using a service) (fixed from review).

II. Administrator Operations:

A. Microservice database interaction tests.
1. Test that adding a microservice properly initialized all necessary data.
a) Optional: Test for handling multiple different Open API versions
2. Test that changing microservice scope (public or private) properly updates user-end displays without hindering functionality.

All of the testing and test processes were done with a combination of 'dummy data', an applicable example microservice, and with a few real connections during our last phase of testing. The structure of our project is set up such that all helper functions were separated and tested with unit tests before all of the tests above were written. The following were all of the test processes that needed to pass in order for our software to meet the requirements as well as our definition of done.

## Test Processes:

**Defining New Microservice:**

An admin should be able to easily upload a given swagger documentation file. It should both correctly produce a form for the file and send a call to add it to a defined service/database. We tested this with dummy documentation and ensured the formatting and conversion is correct.

**Updating Microservice Permissions:**

Given an administrator with access to this service we ought to be able to modify permissions to change the scope of a microservice to/from public to/from private.

**Listing Available Microservices on Home Page:**

Given a database or service of existing microservices (and permissions), the app/website should be able to load all of them that a certain user has access to into that user's home page. To test this we made sure that only microservices that the user has access to are listed.

**Selecting a Microservice:**

The program should be able to retrieve a list of all microservices available to a user from a database or service and display it to the user as a menu. Additionally, upon selecting a microservice from the list menu, the user should be redirected to the panel/page for that microservice.

**Making a Microservice Request:**

Given a certain microservice that is selected by a user, the user should be able to access a form that allows them to make a relevant request to that microservice (GET, PUT, POST). The form should only show calls that are possible for that microservice. The user can submit this form and will receive the output of the request (whether just an acknowledgement, or the output of a GET).

**Showing Microservice Request Results:**

Given a microservice request form submission, the microservice page should show the results of said request. For many requests (PUT, POST, DELETE), this is simply an integer response, but for GET requests, it depends on the specific request.

**Generating History Log for Microservice**

Upon loading the panel for the microservice, the program should access the log of all requests made and display them to the user. It should display the microservice, the time of the request submission, all arguments, and the result. The result may be a simple "Request received" depending on the request.

*With thorough testing as well as consideration for all of the functionality mentioned above we believe our team has an adequate plan to maintain the quality of our software prior to deployment.*

## VIII. Project Ethical Considerations

The project, being an internal tool for Dish that does not use any personal data, does not have any direct ethical considerations, but there are of course considerations in how we develop the software and communicate with the client. For example, as developers we must be truthful and transparent in the way the product is made and functions.

In addition to this, once our product is created, we do not have control over which services are connected to our system. Furthermore, we have no way to know how these services are being used via our system. It is possible that our system may be used in an incorrect or even exploitative manner. To try to avoid this possibility, we have heavily documented the system in hopes that any future users will be able to understand it and use it properly.

The main code of ethics standards we are concerned with are

- *ACM 1.3 - Be honest and trustworthy.*

We need to be fully transparent with our client about what we have and have not completed. We also need to make our client aware of any known issues with the software when we deliver it.

- *ACM 2.6 - Perform work only in areas of competence.*

  As students, we are not experts in the industry and have had to teach ourselves much of the technology associated with the project. Though we have done a good job and have adhered to any known guidelines, we may not be following some industry standards we are unaware of.

- *IEEE 6.08 - Take responsibility for detecting, correcting, and reporting errors in the software.*

  Though we were doing this throughout the development process, we can never be fully confident the software is entirely error free. We were given very limited time to develop the product, thus this is a concern.

## IX. Project Completion Status

**Accomplishments** (fixed from review)

We have created a frontend website that makes user and admin interaction extremely easy and efficient by satisfying all of the requirements stated in Section II, Section III, and Definition of Done. The final product also includes some stretch goals discussed during the development process. Stretch goals that were not implemented are listed below.

**Unimplemented Features**

There are a couple features and stretch goals that arose after finalizing the scope of the project. Many we were able to implement, but others, we were unable to. As for unimplemented ones, our client requested a way for our frontend to populate every users' history log with an initial 'default GET request' for every new microservice. This would consist of the admin declaring which GET method would be the default, and having our system do an initial request upon creation of the microservice. We felt that given our scope and the amount of time we had left that this was not possible and not a priority.

Furthermore, part of our initial scope was creating 'new card panels' on the home page which provided a useful description of new microservices to users. However, we discovered that there was no reliable way to define a microservice as 'new', as we were not able to store user specific data. We experimented with defining a microservice as 'new' purely based on creation date, however this seemed like more of a nuisance than a solution as users would repeatedly be shown this "new microservice" description even if it is not new to them specifically. New microservice card panels were unimplemented, but instead, we implemented a system that shows a description of every microservice on hover. This felt like less of a nuisance and less wasted space as it is up to each user to read and utilize the description.

In addition, one of the features that was not a part of our initial scope was displaying example payloads to users for POST and PUT requests. For example, if a user was using a POST request to create a new 'pet' in a database, it would be helpful to know the structure of a pet object.

```
{
      "name" : "Fluffs"
      "type" : "dog"
```

}

Though we were able to implement this, there are some known issues with it. For example, if an item in a payload object is another type of object, our system is unable to display both. This issue, not being a part of our initial scope, was overlooked and the project timeline prevented us from fixing the feature.

### Summary of Testing

We have been continuously performing unofficial testing, such as UI and user acceptance testing with our client. These initial tests were necessary in order for us to get the system running and working. Specifically, when creating and testing our Swagger Docs JSON parser, though we got it working with our initial test JSON, we quickly found out that there were some inconsistencies in these JSON files by testing with other examples. These kinds of tests have been immensely useful throughout the development process.

The Pytest library was used to create unit tests for multiple features of the product, mainly relating to the database. Specifically, we made unit tests for the following: creating the database, connecting the database, adding rows to each table, reading data from each table, accessing endpoints of our site, accessing files in the static folder, and parsing swagger documents. We delivered the product with every Pytest passing.

### Usability Tests

The current version of our system was shared with our main client contact as well as others in the company. These tests were conducted to provide us feedback about overall flow and functionality of the site.

## X. Future Work

While the product is feature complete as per the defined scope, a few currently unimplemented features (mentioned in section IX) could be added to the product after the field session. This work would include allowing objects to be displayed as part of a payload, a currently unimplemented feature, or the ability to choose whether to upload a .JSON file instead of typing one in a form box.

However, the primary work to be done in the future is user implementation. This is not something that was given to us in the project description, as it is expected the client finishes this part in the future. Currently, the product relies on a single database with relational tables, and the available microservices are either public or private. For the product to be fully working, a system of users must be added so that services are available to a managed group of people, and it must have the ability to authenticate users to both sign into the application and log into the administrator page if applicable.

## XI. Lessons Learned

- Bootstrap is an extremely powerful library for HTML elements and associated JavaScript, but can be difficult to work with at times. However, the documentation is extraordinary and very easy to read. Just make sure you use the latest version of the documentation, as google likes to put earlier versions first in search results.
- When working with Jinja templates, the language being used inside of the HTML file is not exactly the same as python, and has many restrictions. When working with the language, you cannot use any libraries such as SQLAlchemy, flask or others. Also, conditional statements and loops must have defined endings, unlike python.

- Documentation is extremely important. We learned this not only throughout the development of our own project, but also when using the test services we were provided with. When attempting to use the test code we were given, we were unable to run the code and had to spend valuable time troubleshooting. These are the situations where great documentation is necessary.
- Viewing the project from the user's perspective is a necessity, especially when working on a project meant to abstract a complicated technology into a user-friendly frontend. Our system would not be as high quality if it didn't take into account our user base.

## XII. Acknowledgments

Thank you to Dish for the opportunity. This project would have not been possible if it were not for Dish and our client who we worked closely with. Meeting 2 days a week for review was instrumental in making sure we were always on the right track.

Additionally, we would like to thank our advisor Caleb.

## XIII. Team Profile

### Gavin Gilmore

Major/discipline: Computer Science + Business

Hometown: The Woodlands, TX

Work Experience:

Sports/Activities/Interest: Varsity Swim Team

### Brady Ancell

Major/discipline: Computer Science

Hometown: Grand Junction, CO

Work Experience:

Sports/Activities/Interest: ACM, casual mountain biking

### Ian McDaniel

Major/discipline: Computer Science + Robotics

Hometown: Houston, CO

Work Experience: None

Sports/Activities/Interest:

## References

## Appendix A – Key Terms

Include descriptions of technical terms, abbreviations and acronyms

| Term | Definition |
|---|---|

| | |
|---|---|
| *Microservice* | *A single service connected to a system of services within a 'microservice architecture': this includes a variety of different 'microservices' that are loosely coupled and independently deployable.* |
| *Request* | *A message that is sent from a person [user] to a specific web server/API/microservice that typically returns a response to the user.* |