CSCI370 Final Report

Whippomorphs

Nathan George
Grant Lemons
Byron Sharman

Revised 2025-06-16

CSCI370 Summer 2025
Advisor: Tree Lindemann-Michael

Revision	Date	Contents
1	2025-05-16	Initial revision. Added the following sections: • ***Introduction*** • ***Functional Requirements*** • ***Non-Functional Requirements*** • ***Risks*** • ***Definition of Done***
2	2025-05-23	Added the following sections: • ***System Architecture*** • ***Software Test and Quality*** • ***Ethical Considerations***
3	2025-06-01	Revised the following sections: • ***Software Test and Quality*** • ***Ethical Considerations***
4	2025-06-08	Added the ***Results*** section.
5	2025-06-09	Altered the document structure to match the report requirements. Added the following sections: • ***Future Work*** • ***Acknowledgements*** • ***Technical Design*** Revised the following sections: • ***Introduction*** • ***Functional Requirements*** • ***System Architecture***
6	2025-06-14	Editing for final submission. • Updated diagrams ▸ Figure 1 ▸ Figure 2 • Added diagrams ▸ Figure 3 ▸ Figure 4 • Improved flow • Expanded ***Lessons Learned*** • Updated & Expanded ***Future Work*** • Rearranged ***Technical Design*** • Expanded Table of Contents depth

Table 1: Revision History

Contents

I. Introduction

Redacted per client request.

II. Requirements

IIa. Functional Requirements

In our first meeting with SwimTech, we were provided with the following set of functional requirements for the audio processing pipeline:

The final service must:

- be deployable/installable for end-user testing
- use an API interface
- process a list of names for matching students with feedback
- have a simple UI for testing as a web app

We have broken those larger overarching requirements down into specific items related to the API and demo we have built for this project.

The voice categorization API must:

- be hosted on a cloud service
- process audio uploaded to that service
- return JSON associating verbal feedback from an instructor with a student

The demo site must:

- record audio from the host computer's microphone
- access the voice categorization API to process the recorded audio
- access the voice categorization API to retrieve results related to an individual swimmer or instructor
- display the feedback results after they have been processed
- be written in Vue (a JavaScript front-end framework)

The demo application need not be deployed, but the API should be hosted on a cloud platform.

IIb. Non-Functional Requirements

In addition to the functional requirements relating to the core features that the two systems we built must have, we worked with SwimTech to establish a set of requirements on the performance, cost, and maintainability of the software we would deliver to them.

- Feedback should be processed within 2 hours of the recording being uploaded.
- SwimTech should review and assent to the costs of the services used for speech to text and text categorization.
- Project code should be well-documented and maintainable.
- The software should be accompanied with step-by-step documentation for how to set up and run the service.
- Python code should use modern type hints.
- Code should be clean and easy to read.

III. Definition of Done

As with all projects, there are more features which we could have implemented given more time, like attempting to filter out the echos and background noise of an indoor pool. However, adding this feature would have come at the cost of our ability to meet the requirements of this project in some other area. Thus, we established the following definition of done to set clear guidelines on what the state of this project should be at the end of the five weeks.

This project is considered done when the following requirements are true:

- There is sufficient documentation to enable SwimTech to integrate the audio processing system with their existing codebase.
- We can demonstrate the user experience of a swim instructor through a demo which enables an instructor to record audio and see the feedback associated with a swimmer in their class.
- All of the functional and non-functional requirements have been fulfilled.

IV. Risks

Risk	**Likelihood**	**Impact**	**Mitigation**
Not feasible to distinguish which voice lines go with which person	Unlikely	Severe	A new project direction would be necessary
Voice recognition is too challenging	Unlikely	Severe	Rapid prototyping to assess the feasibility of voice recognition
AWS goes down	Unlikely	Severe	Accepted risk – AWS outages across multiple availability zones are very rare
Cloud provider bill is unacceptably high	Moderate	Moderate	Estimate the cost of cloud bills with simulation data
Echoing from an indoor pool makes it challenging to run speech to text	Moderate	Moderate	Explore filtering options to remove echoing from audio

Table 2: Risks

V. System Architecture

For this project, we have used Amazon Web Services (AWS) as the cloud provider for hosting the components of the voice categorization API. After the second week of the project we learned that SwimTech's development team is unfamiliar with AWS, preferring to use DigitalOcean. After reviewing the services that DigitalOcean provides, we compiled a report for SwimTech detailing the tradeoffs between DigitalOcean and AWS (See Appendix 2). In short, the reasons we have used AWS for this project are that the transcription on AWS is cheaper and more accurate, and our team was able to get more done in AWS because we were familiar with it.

Figure 1 shows a diagram of the AWS infrastructure we have provisioned for the voice categorization API. The four main services in this diagram are AWS Transcribe, the Serverless Function, the Key-Value DB, and API Gateway.

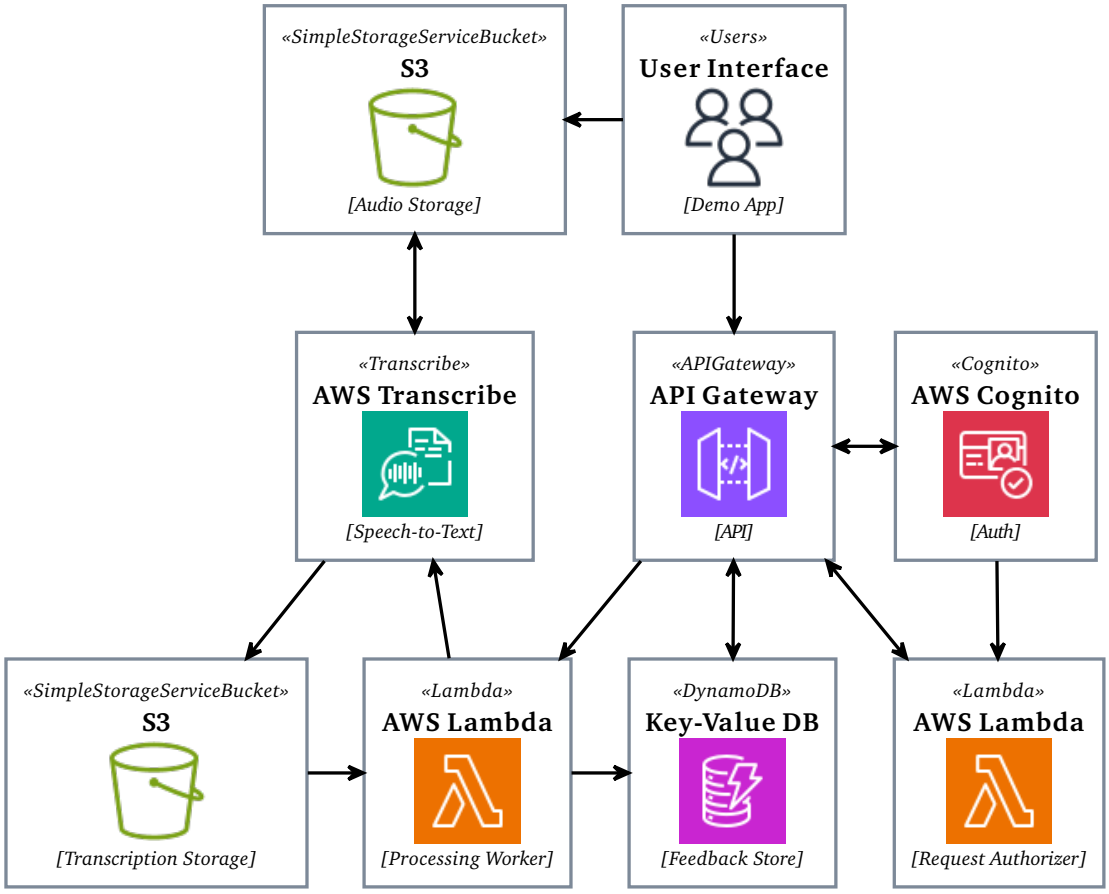

Figure 1: An overview of the project's infrastructure

Va. AWS Lambda

AWS Lambda is an AWS service which spins up compute only when it is needed. It is essentially a function that can be called from a webpage to do some more complicated logic on the backend, like updating a database or running a transcription service. The Serverless Function is the control hub for the main processing logic of the API. It is started by an event from the API containing context: a timestamp, an instructor ID, and a list of students in the class. On a request to the `/process` endpoint an instance of the Lambda is triggered, which does the following steps:

- Begin a transcription job and await completion.
- Retrieve and parse transcription output.
- Run categorization processing steps on transcription.
- Place categorized feedback in a database entry.

Vb. AWS Transcribe

AWS Transcribe is a managed speech-to-text transcription service provided by AWS. Transcribe operates on objects in cloud file systems called AWS S3 buckets. Specifically, Transcribe must

be provided a file URI in the format `s3://bucket-name/file-path`. When Transcribe has finished processing, the resulting transcription is saved to an S3 bucket along with additional metadata about the models confidence of each word in the transcription. This is why the Serverless Function is linked to both AWS Transcribe and its two S3 buckets. The request for processing recieved by the Lambda provides context on the location of the audio file to be processed, and uses this to start a job in AWS Transcribe. The transcription is placed in another S3 bucket, and is retrieved and parsed by the Lambda to extract and process the text.

Vc. Key-Value DB

The results of each processing job are stored in a key-value database table. Database entries are composed of the following fields:

- User ID of the student for whom the feedback applies.
- Timestamp at which the feedback was uploaded.
- User ID of the instructor who gave the feedback.
- The feedback itself.

As a NoSQL database, DynamoDB does not operate exactly like a traditional relational database. Entries in a table have a primary key composed of a "partition key" and "sort key". In our case, the student's User ID serves as the partition key, and the timestamp as a sort key. In order to retrieve feedback from the database, the table is indexed in two ways: on the partition key, and on the Instructor ID. This allows quick access to records for the two ways our API allows: by Student ID or Instructor ID.

Vd. API Gateway

The API Gateway coordinates communication to services on the backend through a REST API. The gateway specifies which endpoints are available, what the format the requests and responses should have (defined through JSON schemas), integrations with other services, and authorization requirements for each endpoint among other things. The gateway serves to trigger different parts of our infrastructure through **integrations**. These allow the gateway to either trigger a service like a Lambda, or directly make an API call to a service, like DynamoDB. Because API Gateway integrates some logic for mapping inputs and outputs to and from the integration, we were able to directly integrate with the database table for feedback retrieval.

VI. Authentication & Authorization

One of the most critically important components of a secure system is the authentication and authorization service. Simultaneously, it can be one of the most complex portions to create from scratch. For this reason, we used AWS Cognito, which offers seamless SSO integration (logging in w/ Google, Microsoft, &c.) and JSON Web Tokens, in order to secure student feedback from outsiders and selectively provide access as appropriate for users, both students and instructors.

This system is disabled at the moment for two reasons. The first is that authentication got in the way of the demo, and the second that the client is not yet sure how they want to integrate our project into their ecosystem. This impacts the User IDs associated with students, which are deeply intertwined with endpoint authorization for student feedback.

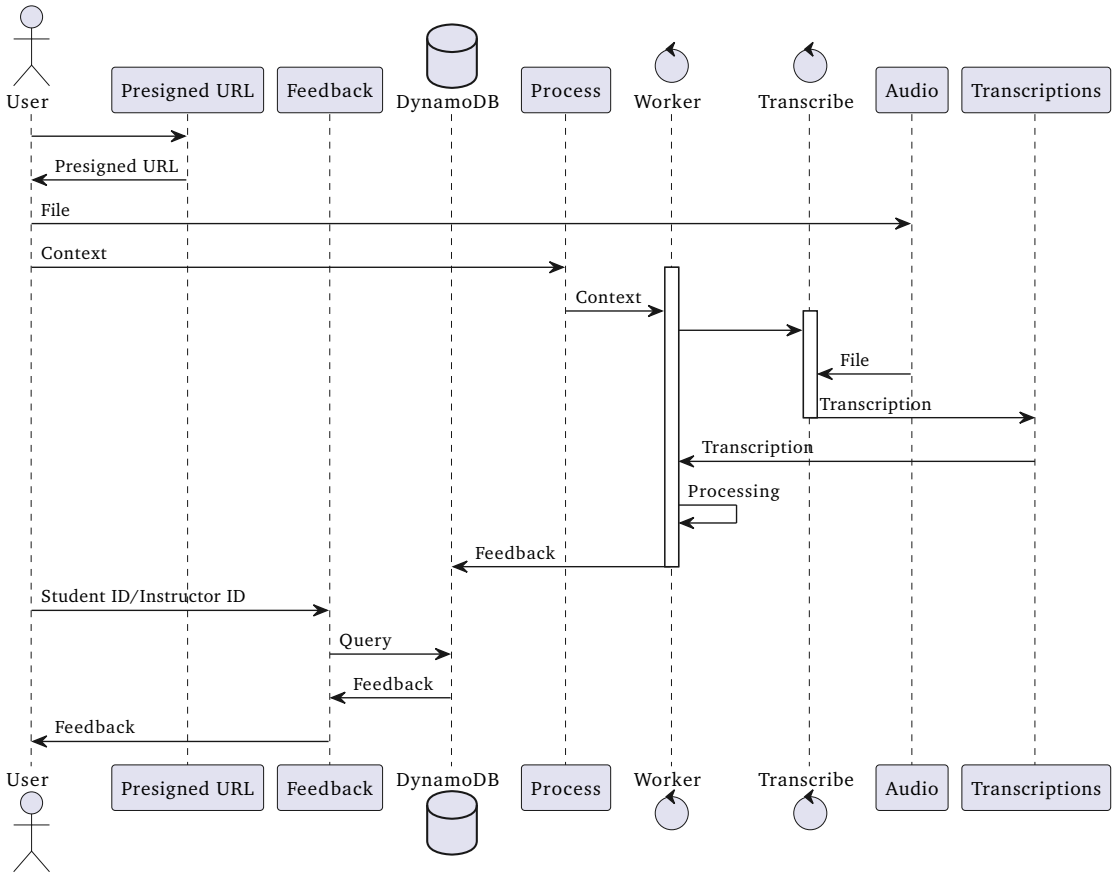

Figure 2: The data flow through our infrastructure

VIa. Appropriateness

Our client's workload on our system is not constant; swim lessons occur at scheduled times of the week, and since it is a local business, demand peaks at select times of day. As such, our architecture was chosen for scalability. Deployment cost is linked to workload, which means we don't waste compute resources at times the system is unused, such as at night.

This architecture is also extremely resilient, as the serverless components distribute work across multiple AWS Availability Zones. This mitigates the risk of infrastructure downtime, which, though unlikely, would have a severe impact.

VIb. IaC and CI/CD

Infrastructure as Code (IaC) is the concept of defining infrastructure resources declaratively through code, rather than provisioning them manually. This allows benefits such as making infrastructure provisioning automated, well-defined, and reproducible, as well as tracking infrastructure changes in source code. Additionally, since it makes applying infrastructure as trivial as running a shell command, a continuous deployment pipeline can easily make infrastructure changes on push. Automating this process makes it easier to hand off the project to our client's developer team.

To reap all these benefits, we defined our infrastructure in Terraform HCL, an industry standard IaC language, and provisioned it using OpenTofu, an open-source fork of Terraform.

Our CI/CD pipeline integrates the client's source control solution (a GitHub repository) with their AWS account, in order to consistently deploy components of our infrastructure automatically on commits. With this system in place, the learning curve for development is much simpler, which helped us hand-off our product to the client. There are several steps in our pipeline, which the client no-longer needs to concern themselves with:

- Building the environment needed for our processing in AWS Lambda (a Lambda layer.)
- Provisioning infrastructure.
- Updating the demo site and invaliding CDN cache.
- Updating hosted API Documentation. (SwaggerUI)

VII. Technical Design

The core logic of the Voice Categorization API matches feedback to students. To do this, we identify proper nouns in the feedback as potential student names. These potential names are matched against the students in the class. Sentences with a student's name identified in them are associated with said student.

Within this process, however, two main problems emerge:

1. Two swimmers may have the same first name making the name categorization ambiguous. (see ***Match Ambiguity***)
2. AWS Transcribe may use a spelling of the student's name that does not match the proper spelling character-for-character. (see ***Phonetic Matching***)

To address these two problems, we developed the categorization pipeline shown in Figure 5.

VIIa. Match Ambiguity

To address the first problem where categorizing the feedback may be ambiguous with swimmers with the same first name, we allow for varying levels of specificity to identify the swimmers names. To match names, the following formats are checked in order:

1. `[First] [Last]`
2. `[First]`
3. `[Last]`

Figure 3 illustrates the control flow of an unambiguous match by last name.

If when checking a format, it can be matched ambiguously, then the matching fails. This can be improved upon in the future by placing the unmatched feedback in a database to be manually categorized later. Once the feedback is matched, we look up the name from the list of swimmer names and match it with that piece of feedback.

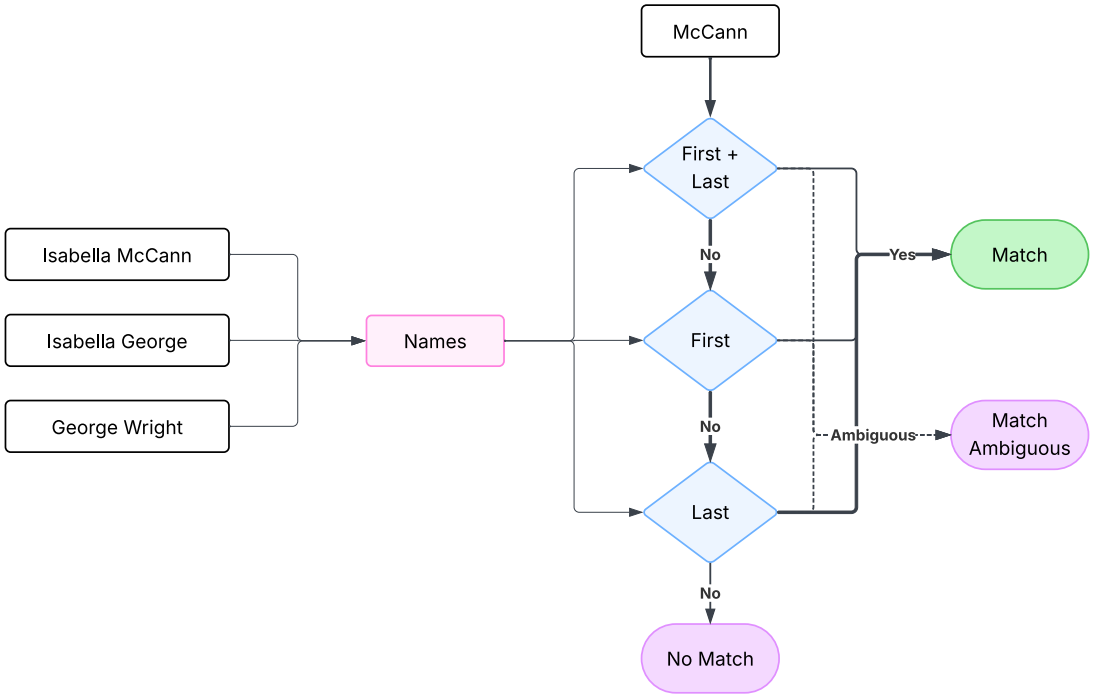

Figure 3: Flow chart representing tiered name matching specificity. Input "McCann" unambiguously matches the last name of student "Isabella McCann"

VIIb. Phonetic Matching

Because Transcribe may use a different spelling of name, we attempt to compare names phonetically instead of by character. To accomplish this, we use an algorithm developed by Lawrence Philips called **metaphone**. The metaphone algorithm takes in a word and outputs a string of characters representing the sounds in that word. It does this by applying a set of deterministic rules to merge character sequences which sound similar. For instance, both `KS` and `X` sound the same, so they are both represented by the sequence `KS` in metaphone's phonetic representation. After transforming the names that we would like to compare into their string phonetic representation we compare the strings to determine if two names match. Figure 4 illustrates how transcribed names and canonical spellings both map to the same metaphone representation.

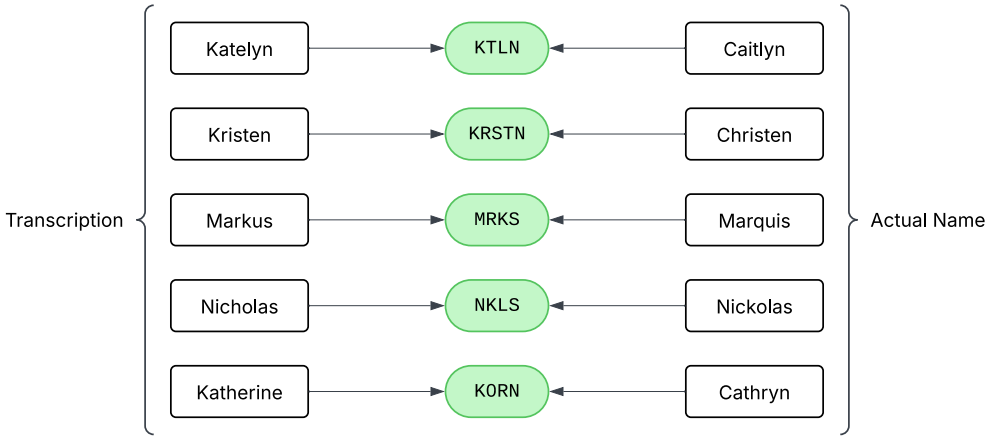

Figure 4: Identical phonetic representations of common and uncommon name spellings

VIIc. Part of Speech Detection

To perform the name matching procedure, we need to extract the names from the feedback; otherwise, matching words like `Paul` and `pool` with the same phonetic representation could be confounded. To extract names, we use part of speech detection with the Python library **Natural Language Toolkit** (NLTK). In Figure 5, the first yellow processing box after transcription shows the part-of-speech tagging of the feedback "Nathan should keep his fingers together."

To identify names, we look for words tagged by NLTK as proper nouns (`NNP`) and extract those as names. Though this often works, it may mis-categorize the part of speech of a name leading to errors in the categorization. A more resilient approach to name extraction might involve checking each word against a dataset of first and last names to determine if it is a name. After extracting the names from the transcription, we compute the phonetic representation of the names in the feedback as well as the list of swimmer names.

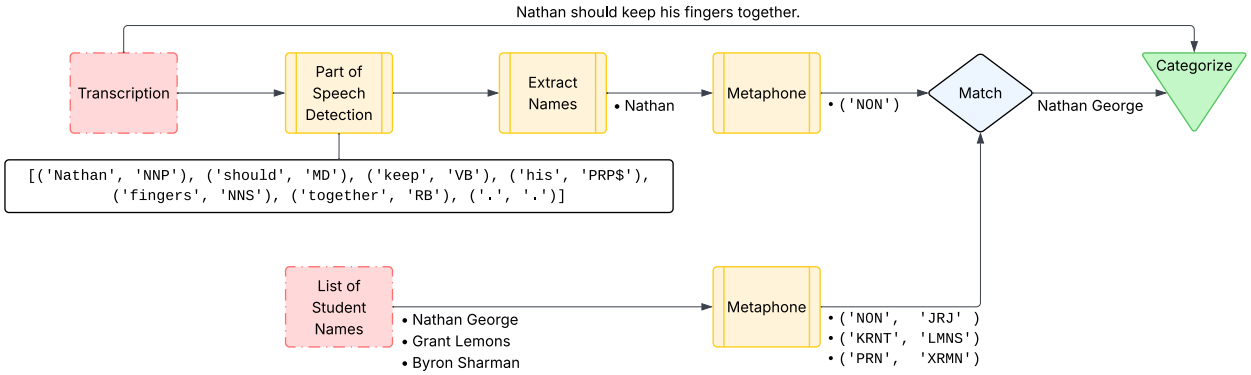

Figure 5: The categorization pipeline

VIII. Software Test and Quality

To maximize the value of our project to SwimTech, we wanted to write bug-free code adhering to a high standard of quality. These two characteristics make our code easier to integrate into another software system, providing a solid foundation for our client to start with.

VIIIa. Testing

Testing is key to ruling out common faults in programs. We wrote automated unit tests for the core functionality of the serverless function worker to verify that it handles normal cases as well as edge cases, like alternate spellings of names and multiple students with the same name.

We attempted to implement automated end-to-end testing, but the asynchronous nature of our product made doing so challenging. Due to the accelerated timeline of our project, we had to de-prioritize automated end-to-end testing, and did not complete it. However, we do regularly manually test the functionality of our component by using the demo to record audio and seeing if feedback shows up.

Automated Tests

Test	Description
`test_phonetic_eq()`	Alternative spellings of names are matched correctly
`test_extract_names()`	Names are correctly extracted from the transcription
`test_extract_feedback()`	Feedback is split up correctly if a transcription includes feedback for two students
`test_match_name()`	Names are matched from most specific to least specific
`test_categorize_transcription()`	Transcription matched to the correct name
`test_transcription_to_feedback.py`	Collection of tests running through the entire categorization pipeline including testing edge cases where students have the same name

Non-Automated Tests

- Record audio from the demo site and ensure that feedback is visible on the feedback page.
- View network requests and verify status codes match the 2xx range.

  These tests validate the ability of the entire system to work as one cohesive unit from uploading audio to retrieving feedback. There were key to validating that we met the requirements for both the voice categorization API and the demo application.

VIIIb. Software Quality

Our code quality evaluation process was augmented with formatting and linting tools which check for style, obtuse formatting, and some syntax-related bugs. We integrated two such tools, `pylint` and `biome`, into the development of our categorization worker (Python 3) and demo (TypeScript) codebases, respectively. Both run when we push code to GitHub as part of the continuous integration process. We do not merge pull requests that fail these tools' quality requirements.

In addition to automated quality checks, our code review process included manual approval from another team member. Beyond filtering common errors, code reviews helped our team stay on the same page about how each component of our system works, maintaining a healthy knowledge of the codebase. This knowledge improved quality by facilitating the development of a cohesive system rather than a hodgepodge of clumsily connected components.

Our testing tools and quality assurance tools are part of our continuous integration pipeline on GitHub. This reduced the developer friction in running these tools manually and ensured that code merged into the main branch adheres to our testing and quality standards.

IX. Ethical Considerations

The ACM and IEEE publish a set of ethical standards for engineers. [1] Within our work for SwimTech, a few of these principles stand out as especially significant to our work in this field session project.

2.01 Provide service in their areas of competence, being honest and forthright about any limitations of their experience and education.

2.05 Keep private any confidential information gained in their professional work, where such confidentiality is consistent with the public interest and consistent with the law.

3.12 Work to develop software and related documents that respect the privacy of those who will be affected by that software.

8.01 Further their knowledge of developments in the analysis, specification, design, development, maintenance and testing of software and related documents, together with the management of the development process.

With regards to the first principle, our team lacked extensive industry experience and had not completed our degrees. Thus, it was important for us to not only learn quickly but also communicate gaps in our knowledge so SwimTech has a solid understanding of our skills and an accurate expectation of our results. Confidentiality is pertinent to this project; we signed NDAs about specific details of the project. The voice categorization service interacts with sensitive user data, including audio recordings as well as transcribed feedback of swimmers. Thus, our exclusive use of cloud services is motivated by a desire to choose a platform known to be secure for data at rest and in transit, greatly lessening the likelihood that data is accessible to unauthorized users. Lastly, as this experience was designed to grow our skills as software engineers, it was key that we utilized this opportunity to grow as more ethical programmers.

There are a few additional ethical principles relevant to our work:

3.10 Ensure adequate testing, debugging, and review of software and related documents on which they work.

3.11 Ensure adequate documentation, including significant problems discovered and solutions adopted, for any project on which they work.

These are both related to the hand-off, when we deliver the voice categorization service to SwimTech. If we run out of time adding testing or sufficient documentation, the value of this project to SwimTech may be severely degraded.

In order to address these ethical considerations, we implemented a few ethical tests which we can evaluate throughout the project. The first test is the legality test, where we need to make decisions within the context of the NDAs we signed, which includes hosting our code in a private repository. The second test is the reversibility test. The team which we are delivering this software to are developers like ourselves, so we must consider how our decisions might burden our successors and how they impact the cloud bill this software incurs.

As our project revolves around machine learning processing of audio files by swim instructors about swim students, most of whom are children, data privacy is an ethical concern. To mitigate this, we are using a transcription service from a reputable source that does not train on inputs and does not retain copies of the input.

To mitigate the impact of potential security vulnerabilities, we implement a two-day deletion policy on all audio recordings and their transcripts. We also have extremely tight access control policies that ensure least access to all our services, including but not limited to our database and the buckets containing audio recordings and transcripts.

Our API has strict authentication and authorization requirements, which ensures customer feedback is private to said student and authorized employees and is not visible to other students or unrelated parties.

X. Results

See the ***Definition of Done*** section for a summary of what we expected to accomplish at the end of the project.

Xa. Project Completion Status

We have successfully created an API that performs all the requirements. We learned that the client does not yet have any existing database to track students and instructors, so we made a placeholder one designed to be easily replaceable in case the client decides to make their own. We also made a demo that records audio, uploads it to the API, and displays the resulting feedback. Our main deliverable is the API, and other goals, such as allowing instructors and students to log in to the demo and displaying feedback specific to them, are out of scope.

Our API has three main endpoints, `/get_presigned_url`, `/process`, and `/feedback`. These are quite performant, returning responses within a few hundred milliseconds, though the speed can vary based on if any Lambdas involved need to cold-start.

`/get_presigned_url` returns a URL the frontend can use to upload directly to an S3 bucket, including the authorization information required.

`/process` asynchronously runs the speech-to-text and voice categorization processes, putting the results in a database.

`/feedback` has some sub-endpoints which query feedback by instructor ID or student ID from the database.

We have fully documented all the endpoints and their schemas using OpenAPI, autogenerated from the AWS API Gateway based on the documentation we've attached to each endpoint and the schemas we've defined. This OpenAPI spec is then used to generate a static SwaggerUI site, which presents the API spec in a clear manner at the path `/api/v2/spec` from our demo site.

Additional extensive documentation for all the components of the project has been written and is available through GitHub.

Deploying the API, SwaggerUI site, and demo are fully automated thanks to our comprehensive CI/CD pipeline.

Xb. Future Work

The largest batch of work to be done is deciding on how the application will retrieve the list of names of the swimmers in the class. Currently, the voice categorization service needs a list of names and User ID's as input. If this stays the same then, the app that the swim coaches use will need to pull a list of names from a database managed by SwimTech. Alternatively, a class id could be sent to the categorization service, and the service could directly lookup a list of names.

Additionally, within our testing suite, we intended to write some end-to-end tests. Unfortunately, due to the infrastructure and the time constraints of the project this became more challenging than we initially anticipated. Specifically, due to the fact that no event is sent out when a request to the service is finished, it is difficult to get the results in a deterministic manner.

Instead of triggering the `/process` endpoint with context, a good way to re-architect our infrastructure would be transitioning to an event-based pipeline from the current hub-and-spoke structure currently implemented with our worker Lambda. In this model, context would be associated with a job name via the `/get_presigned_url` endpoint (renamed to something like `/create_job`) and placed in a DynamoDB table. Then, the AWS Transcribe job and categorization Lambda invocations would be triggered by events in the audio file and transcription output S3 buckets respectfully. If the job name is maintained within the pipeline, the categorization Lambda can retrieve job context from the DynamoDB table.

Because the surrounding context for our project within our client's systems is unclear, we were unable to fully implement Authentication and Authorization within the project (see ***Authentication & Authorization***). For authorization,

Xc. Lessons Learned

Our team learned a great deal while working on this project. For instance, most of us had minimal DevOps experience, and during our first attempts at writing the CI/CD pipeline, we found ourselves constantly dealing with corrupted Terraform state and AWS resources strewn everywhere. Through these experiences we have learned the limitations of per-branch infrastructure and also why larger projects define infrastructure in separate Terraform configurations, using tools like Terragrunt to batch-process them.

We also learned to avoid long-running feature branches. Our migration from our `v1` prototype backend to the more robust `v2` backend was on a single branch, and it took much longer than anticipated. This became a major blocker for other features, and stalled the work most developers on our team could do.

A final lesson was that creating and enforcing timelines matters. Our `v1` prototype was done by the end of the first sprint, so we felt we would surely be able to iron out the rest of the project in the remaining four weeks. We found that tasks take the time they're given, however, and our velocity slowed such that those last few tasks took the entirety of the remaining four weeks. Had we enforced per-ticket deadlines, and kept ourselves to a stricter timeline, we would have had higher velocity for longer and been able to reach for client stretch goals.

XI. Acknowledgements

We would like to thank the following parties:

- Our advisor, **Tree Lindemann-Michael**, for guiding us through the whole process and assisting our team dynamic
- Our client liaison, **Sloane Smith**, for her work in maintaining healthy communications
- **Evan Lim**, another SwimTech employee who provided us with credentials to SwimTech's AWS account
- The course coordinators, advisors, and guest speakers, especially **Kathleen Kelly**, for making the whole experience possible

XII. Team Profile

Nathan George

Major: Computer Science
Class: Junior
Hometown: Colorado Springs, CO
Experience: Programming lead for FRC Robotics, Camp Counselor at Camp Como
Clubs: ACM, Competitive Programming

Grant Lemons

Major: Computer Science
Class: Junior
Hometown: Dallas, TX
Experience: 2x Software Engineering Intern at Headstorm, 2x Blasterhacks Silver Medalist, TA for CSCI220 & CSCI410
Certifications: AWS Solutions Architect – Associate
Clubs: ACM
Hobbies: Cooking

Byron Sharman

Major: Computer Science
Class: Junior
Hometown: Colorado Springs, CO
Experience: TA for CSCI220 (Data Structures and Algorithms), 2x Blasterhacks Silver Medalist
Clubs: ACM

References

[1] IEEE-CS/ACM Joint Task Force on Software Engineering Ethics and Professional Practices, “Code of Ethics for Software Engineers.” [Online]. Available: https://www.computer.org/education/code-of-ethics

Appendix 1: Key Terms

API: Application Programming Interface
Cloud Provider: A platform like AWS, GCP, or Azure that provides cloud services
UI: User Interface
AWS S3: Simple Storage Service, a cloud file-system
JSON: JavaScript Object Notation, a serialization format
CDN: Content Delivery Network
AWS: Amazon Web Services
GCP: Google Cloud Platform
ACM: Association for Computing Machinery, a national organization with a branch at the Colorado School of Mines
IaC: Infrastructure as Code, a programmatic expression of infrastructure resources
CDN: Content Delivery Network
CI/CD Pipeline: Continuous Integration / Continuous Deployment pipeline

Appendix 2: AWS–DigitalOcean Comparison Report

Intro

To develop the voice categorization service for Swim Tech's field session project, we decided to use Amazon Web Services (AWS) to host the cloud infrastructure which we would need to provision to build this service. We understand that Swim Tech's development team currently uses DigitalOcean to provision their cloud infrastructure, so we would like to explain our reasoning for choosing AWS over DigitalOcean for this project.

Infrastructure Requirements

Voice transcription is the most expensive component of the infrastructure on both DigitalOcean and AWS. Conventionally, voice transcription is done by running audio through a language model which has been trained to transcribe text for a specific language. Similar to large language models, these models run more efficiently on dedicated hardware like GPUs but can run slower and with less precision on a CPU.

AWS provides a fully managed transcription service called AWS Transcribe. For our use case, AWS Transcribe is expected to cost $0.02400 per minute of audio.

DigitalOcean does not provide a transcription service like AWS, so transcription must run on one of their other services. To select the right service on DigitalOcean, we must look at the specific hardware requirements of a standard transcription model. Here we look at OpenAI Whisper. The smallest model of Whisper, `tiny`, requires 1 GB of RAM, which prevents it from being run on the smallest Droplet (DigitalOcean's serverless function – see Serverless Architecture), which has only 512 MB of RAM. A Droplet with 1 GB of RAM costs $6.00 per month. A more reasonable estimate of running the `small` model requiring roughly 2 GB of memory would cost $24.00 per month. Even with the `small` model, transcription on DigitalOcean will likely be slower and less accurate than AWS Transcribe because the DigitalOcean hardware is not specifically tuned for audio transcription.

Price Comparison

We estimate with 4 hours of lessons per week that roughly 30 minutes of transcription would be needed. Over a month, this would be 120 minutes. With these estimates, the cost of voice transcription for each platform is given below.

Service	**Monthly Usage**	**Price Per Minute**	**Monthly Bill**
Digital Ocean	120 minutes	N/A	$24.00
AWS	120 minutes	$0.02400	$2.88

Additionally, both platforms require additional infrastructure beyond just transcription to serve the website, perform additional processing, and store processed feedback. These costs are much smaller than the transcription costs for both cloud providers, and we would expect them both to be less than $1.00 per month.

Pivot Cost and Project Completion Risk

The second reason we used AWS for this project is that we have experience with it. When we learned that DigitalOcean was preferred over AWS, we had already finished a prototype and had designed our architecture on AWS. Since then, we have invested several hours in exploring how to make DigitalOcean as cost-effective as AWS.

We believe that we made the right decision with going with what we know to provide a better product for Swim Tech, and have made strides to document how our system in AWS works so that other developers can effectively use this tool. At this time, switching to a new cloud provider would be a significant change in direction requiring us to redo work started in the first week of the project and jeopardizing our ability to finish on schedule.

Serverless Architecture

For services with limited or irregular demand, provisioning a server 24/7 doesn't make sense from a cost perspective. Instead, quickly spooling up a new process on a managed machine allows us to only pay for the compute time we use.

For services that may need to scale, using a serverless architecture is also most effective (to a point) because it load-balances by default. This reduces the need for additional load-balancing infrastructure that would also need to be running nonstop.

This pay-for-what-you-use model is the main benefit of using AWS in our project. Our architecture requires several services, such as a database, API server, authentication service, and transcription service. Some of these could run on the same compute instance 24/7, but it wouldn't be scalable, and not all could run on the same instance—so it would be required to pay for multiple services to run 24/7. Although some compute solutions on DigitalOcean are cheap, the transcription service would require a high tier of compute, which is expensive.

By using AWS instead, we would only be billed for usage, which is much cheaper. AWS API Gateway, for instance, charges $3.50 per million requests, which the application is unlikely to reach over its entire lifetime. The cheapest DigitalOcean droplet, by comparison, is $4 per month. Similarly, AWS Transcribe costs $0.02400 per minute of audio processed, or about $1.44 per hour of audio. We're not sure which level of DigitalOcean droplet would be required to run it, but assuming it works without using a GPU instance (which is absurdly costly for a long-running instance), a conservative estimate using the cheapest general-purpose droplet costs $63.00 per month to run.

Note: Long-running Processes

Provisioning compute instances, as we would have to on DigitalOcean, comes with additional complexity inherent with long-running processes. Running a program for months or years on end without error is unrealistic, risking service downtime and sometimes requiring manual intervention, which is **very** costly.

Note: Resilience

Serverless architectures on AWS are inherently resilient, as processes can be spawned in any of many Availability Zones. This greatly reduces the likelihood of a provider outage impacting the service. In contrast, for the same level of resilience on DigitalOcean, we would need to provision three times the baseline infrastructure needed to run the application.

Authentication & Authorization

One of the most critically important components of a secure system is the authentication and authorization (jointly, “auth”) service. Simultaneously, it can be one of the most complex portions to create from scratch. For this reason, on AWS we will use AWS Cognito, which offers seamless SSO integration (logging in w/ Google, Microsoft, &c.) and JSON Web Token technology, in order to secure student feedback from outsiders, and selectively provide access as appropriate for users, both students and instructors.

Were we to use DigitalOcean, we’d need to construct this ourselves, which is yet another server to pay for, and likely a database to store user login information as well.