Pitchback: Improving Speaking Patterns and Communication Style Through Natural Language Processing and Data Visualization

A Thesis

Submitted to the Faculty

in partial fulfillment of the requirements for the

degree of

Bachelor of Arts
in
Computer Science

by

Sidney Wijngaarde

Department of Computer Science

DARTMOUTH COLLEGE

Hanover, New Hampshire

May 31, 2017

Advisor:

_____

Tim Tregubov

ABSTRACT

Speech disfluencies are common in human discourse. It has been shown that speech disfluencies negatively affect comprehension [Corley and Stewart, 2008].“Pitchback”, is a web application that employs speech recognition, natural language processing, and audio analysis to programmatically provide feedback for improved speech patterns and communication style. Our microservice based architecture runs feedback computations in parallel providing the user with near real time results. We have shown that this method for providing automated feedback is helpful to users preparing for public speaking events.

Acknowledgements

I first and foremost would like to thank my family for supporting my academic pursuits. I would also like to thank Tim Tregubov for advising this project as well as being a guiding hand throughout my college career. Finally, I would like to acknowledge my close friends who have seen and supported Pitchback since its inception.

Contents

List of Figures

INTRODUCTION

PROBLEM

Speech disfluencies are common in human discourse. It has been shown that speech disfluencies negatively affect comprehension [Corley and Stewart, 2008]. Public speaking skills are necessary to be successful in every field. Currently, the most effective way to gain the critical feedback necessary to improve as a speaker is to hire a speech coach. This solution is well outside of the price range for the average person. Other methods such as presenting to friends, practicing in front of a mirror, or recording oneself may not provide the insights necessary to make progress as a speaker.

Presentation software applications such as PowerPoint, Prezi, slides.com, etc. focus on displaying information rather than the speaker's effectiveness in communicating information. Despite the recent advancements in machine learning and artificial intelligence, there has been no work at the intersection of effective presentation skills, speech recognition, and natural language processing with the goal of improving presentation comprehension. Applications such as Grammarly [gra] and Hemingwayapp [hem] offer criticisms on sentence comprehension, sentence complexity, tone of voice, etc. however there are no applications allowing users to practice an oral presentation and receive similar feedback.

Solution

The proposed solution, "Pitchback", is a web application that employs speech recognition, natural language processing, and audio analysis to programmatically provide feedback for improved speech patterns and communication style. Users on Pitchback upload a video of themselves delivering their presentation as well as any materials such as a script or presentation slides. The web application uses multiple cloud services to label speech disfluencies from the audio component of the uploaded presentation.

The transcript of the presentation is analyzed for grammatical mistakes, concision (pas-

sive voice), and words per minute. Using metrics collected from the audio and transcript the platform will highlight problematic portions. Users create a project for each presentation on the platform. Each project is a collection of practice sessions, allowing users to revisit older iterations and visualize their feedback over time. The users can also post their presentations on the platform to receive feedback from other users.

PREVIOUS WORK

The technical report "Exploring Filler Words and Their Impact" [Emily Duvall and Divett] concludes with recommendations for listeners at public speaking events. A major takeaway from the report is that filler words negatively affect listener comprehension, speaker credibility and distract users from the presentation content. Corley and Stewart in the Journal of Psycolinguistic Research examine the relationship between speech disfluencies and listener comprehension and arrive at similar conclusions [Corley and Stewart, 2008]. Therefore, identifying filler words in a public speaking transcript was our main focus in providing automated feedback. Work in the field of speech recognition has lead to models for identifying filler words, edit words, and interruptions [Matthew Lease and Charniak, 2006]. Although these techniques exist there is little work bridging the gap between disfluency detection and improving speaking patterns.

Rhapsodize [Tark and Yoon] is a mobile application for improving public speaking skills. Users speak to the app and gain live negative feedback when they use common filler words. The app is positioned to evaluate everyday speech as opposed to presentation quality. Finally, Rhapsodize provides insights on filler words alone, there is no consideration for the greater sentence structure.

Criticisms on the state of presentation software come from exploring applications such as Microsoft PowerPoint, Prezi, Slides.com, three very popular applications for preparing a presentation. These applications provide users with an interface to structure visual pre-

sentations. The presenter mode in PowerPoint merely adds user notes and a timer. There is no way for users to get concrete feedback on the quality of their presentation and the ability for their audience to receive the information.

Grammarly [gra] and Hemingway [hem] app both offer feedback on sentence clarity, concision, and grammar. Using one of these applications to analyze a transcript from a presentation can give concrete feedback on the content of the presentation but offers no insights into how the user can improve his/her delivery. In other words, in analyzing just text these applications provide no recommendations on tone, pacing, and other voice related components of effective speaking.

Most existing solutions seem to address a single element of effective public speaking. Pitchback attempts to address these elements in a cohesive application that can present the user with critical feedback in real time.

DESIGN

As a user centered app the design of the usability, flow, and organization of user interface elements was critical during the building of the user interface for Pitchback. The goal for the frontend of the platform is to provide users with data visualizations that present our insights in a consumable manner. We have chosen to follow many of the concepts laid out in the material design [mat, a] specification. The views are built with the React [rea] rendering framework. The material-ui [mat, b] react component library was used to assist in implementing material design. In this chapter we will review the core views of the frontend.

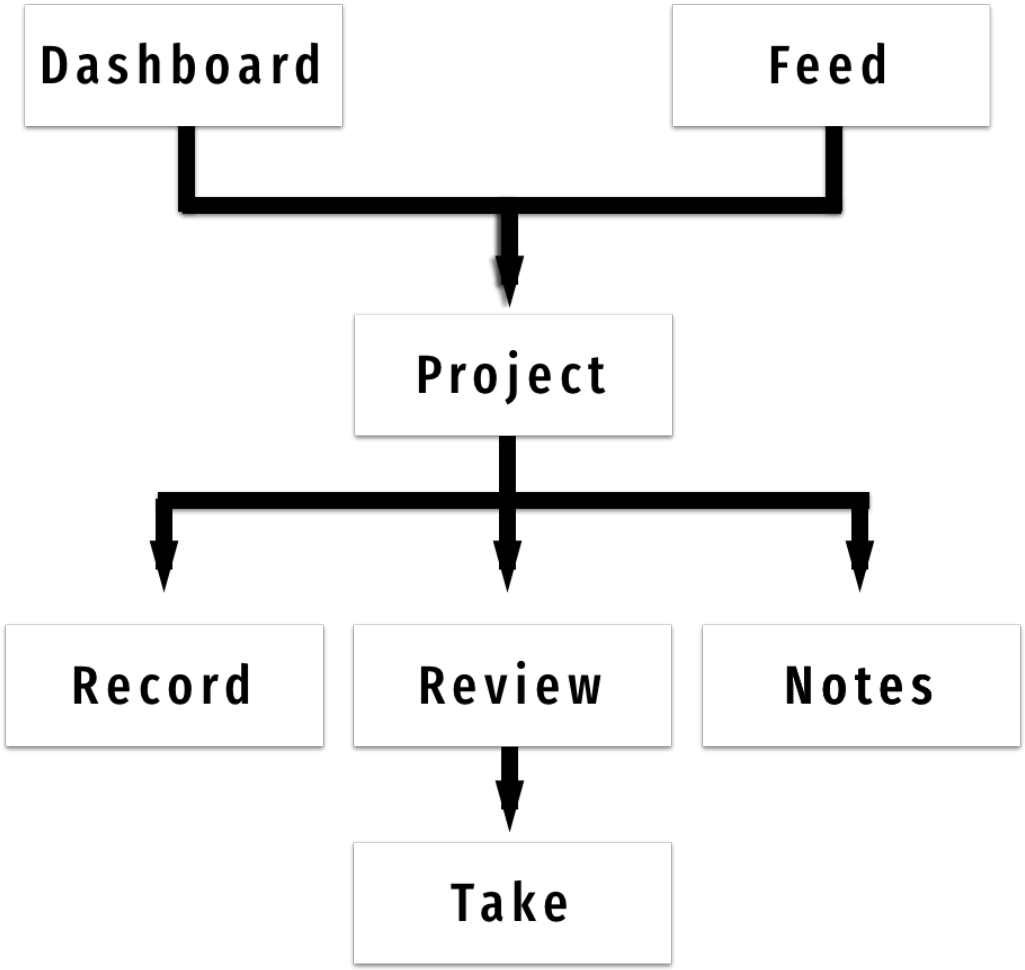

Figure 1: Views and User Flow

Dashboard

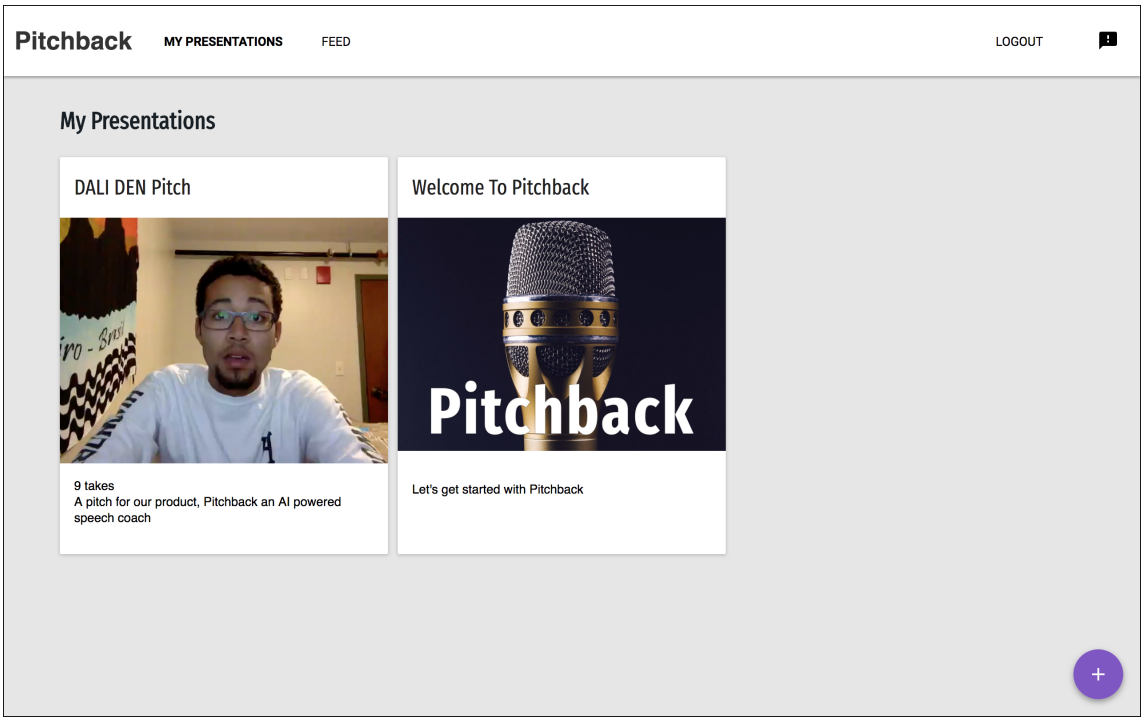

Figure 2: The Dashboard View

Each individual presentation is called a *project*. A *project* is comprised of multiple takes, or practice sessions uploaded by the user. Each *take* is a video recording of the user delivering their presentation. We built a dashboard (see figure 2) because we wanted users to create and view multiple projects. We use a card layout to give a brief overview of the projects that users have created. Cards imply that there is more information that is not currently seen. Each of these cards lead to the expanded *project view* for the specific project. The preview of the card is the first frame of the last *take* the user uploaded for that project to visually remind users of the presentation content. If there is no video in the project (0 *takes*) we display the Pitchback cover photo as shown on the right. In the lower right corner we have a floating action button indicating that a "create" action is tied to it. The button leads to the modal view in figure 3.

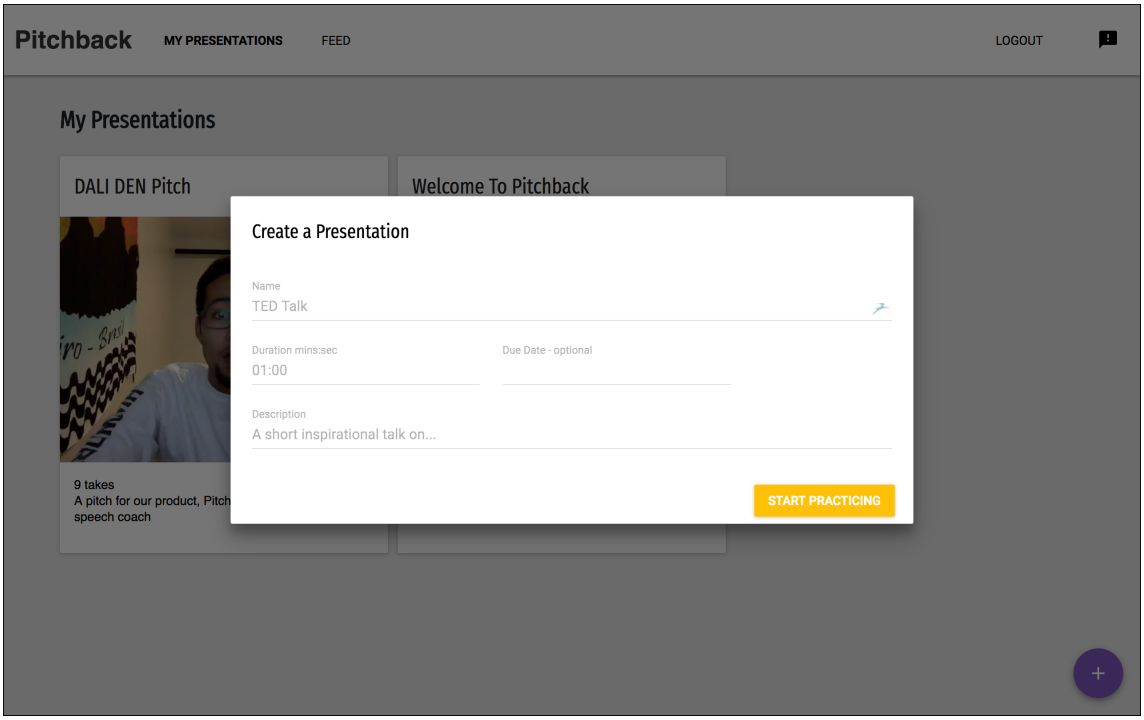

Figure 3: Create a Project Modal

The dialogue in figure 3 presents users with a form to provide metadata about their presentation. The title and description fields are used to display the cards here on the dashboard and in the later discussed *feed* view.

Project

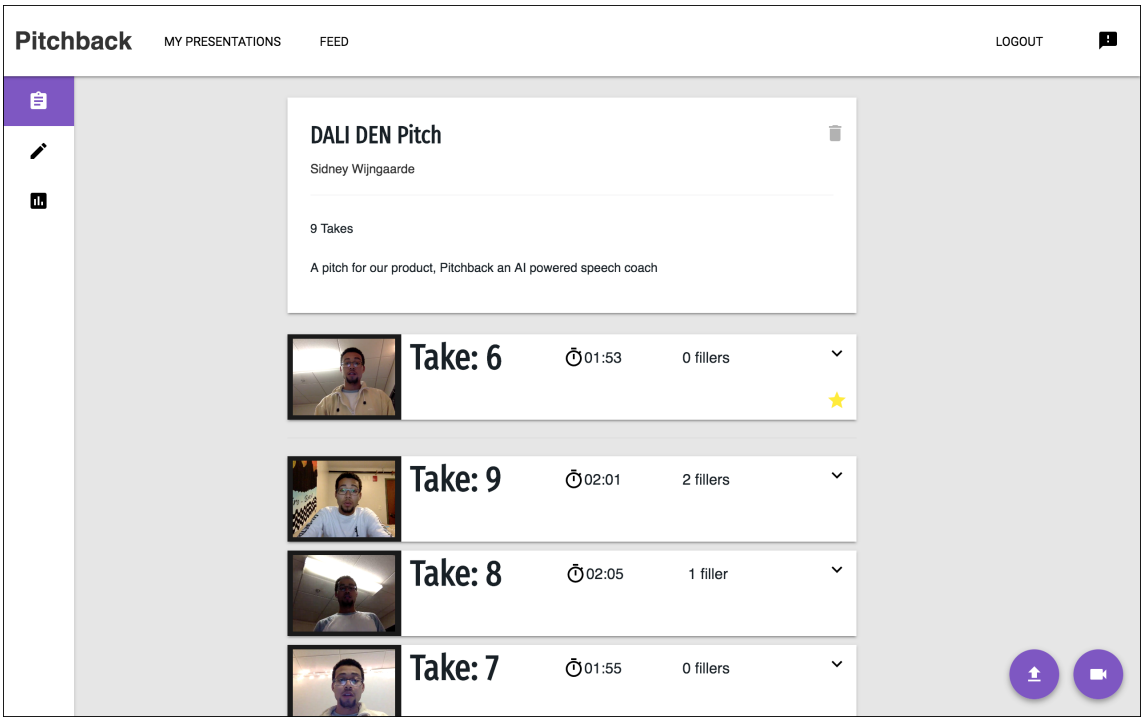

Figure 4: The Project View

We built a *project view* (see figure 4) to allow users to gain access to all of their project related data in one view. After clicking a card from the *dashboard* or *feed*, users are presented with the *project view* for that *project*. This view contains two three subviews. The first (shown in figure 4) is the *review* tab. Here users have insights into all of the data and insights regarding the current project. At the top we have the project data card which presents the metadata about the project. Takes are displayed as cards as well. The takes are organized by date from most recent to oldest. Users can favorite takes which move them to the top of this list. The preview image is the first frame of the video to provide a visual differentiation between the *takes*. Clicking the *take* expands the card to reveal data and comments from other users as in figures 5 and 6.

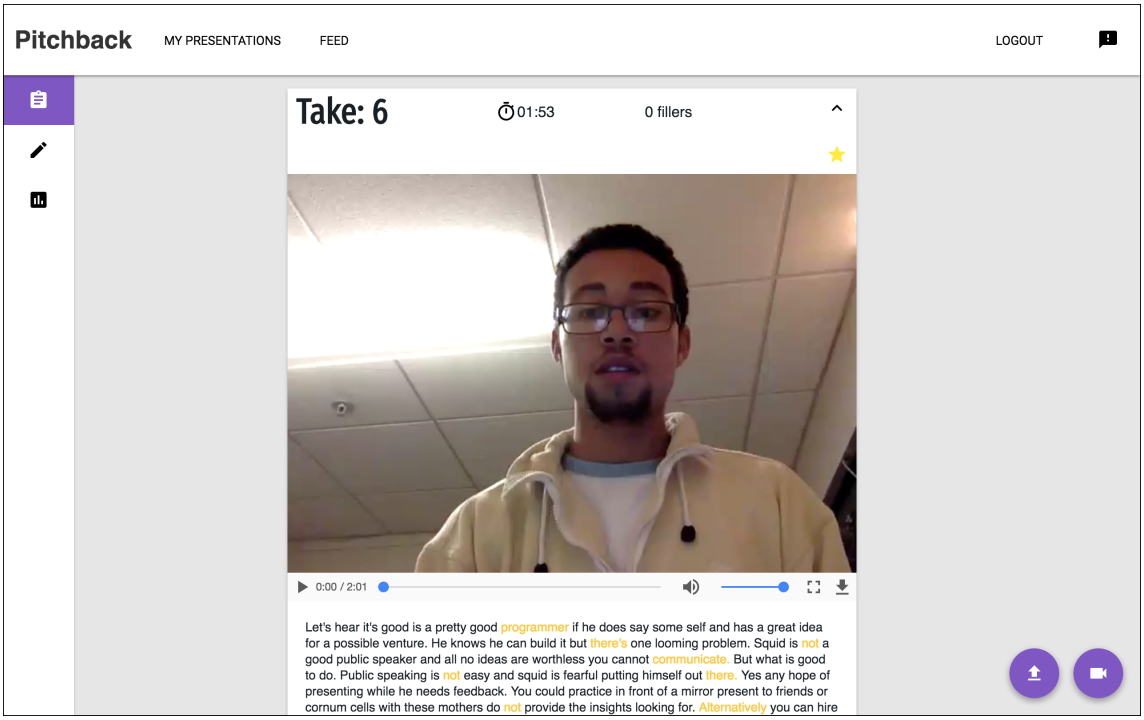

Figure 5: An expanded take card

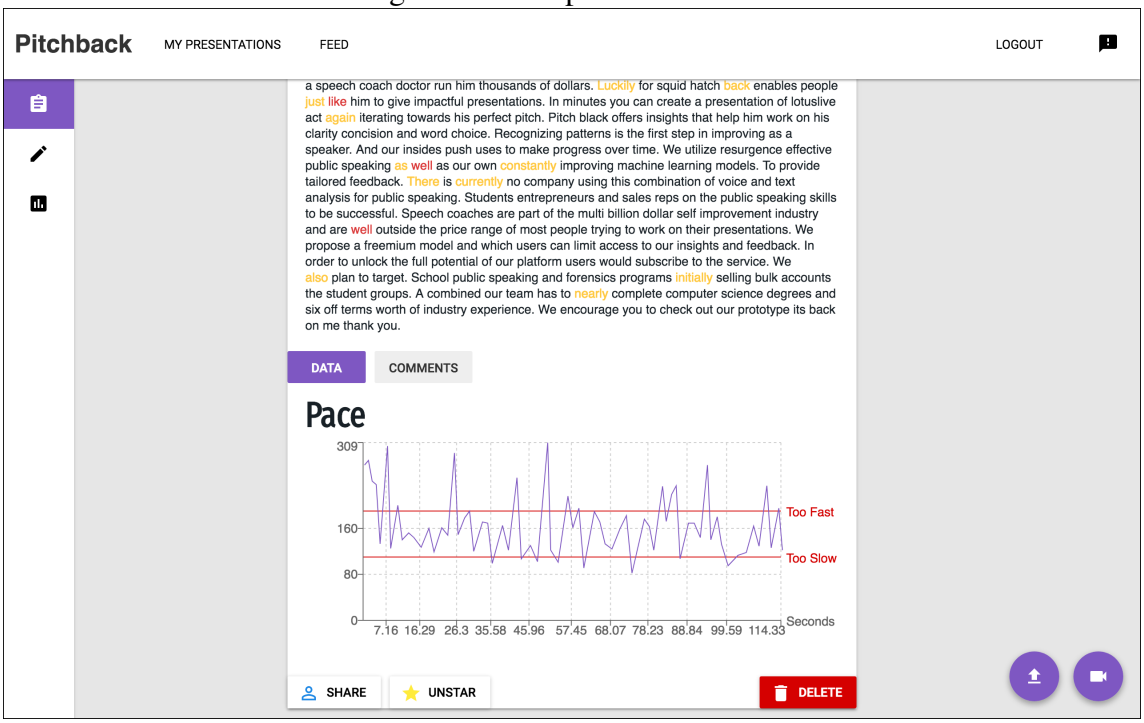

Figure 6: An expanded take card

The users video is followed by a highlighted transcript. The goal of the transcript is

to provide visual cues for Pitchback's insights as well as provide a tight feedback loop for users. Users can see the differentiation between warnings and significant errors at a glance from the color of the highlights. Hovering over highlighted words reveals a popup with a description of the error and suggestions for how to improve. Clicking the transcript navigates the video to where the mistake was made. Filler words are tallied at the end of the presentation as well. At the bottom of the card there are two options for favoriting or sharing the *take*.

Sharing, publishes the take to the general feed (see figure 11) and gives other users on the platform access to the full *take* card. These other users can leave comments under each public take. In sharing a *take* users can gain more critical feedback from others on the platform that may address aspects of public speaking that are not covered by the Pitchback feature set such as facial expressions, emotion, humor, and overall cohesion of the presentation content.

There are two floating action buttons at the bottom right of the view. The first launches an upload dialogue. Users can drag and drop in a previously recorded video to be analyzed by Pitchback. This is useful for analyzing real presentations from the web or given and recorded by the user elsewhere. The second action button launches the in-site recording view.

Record

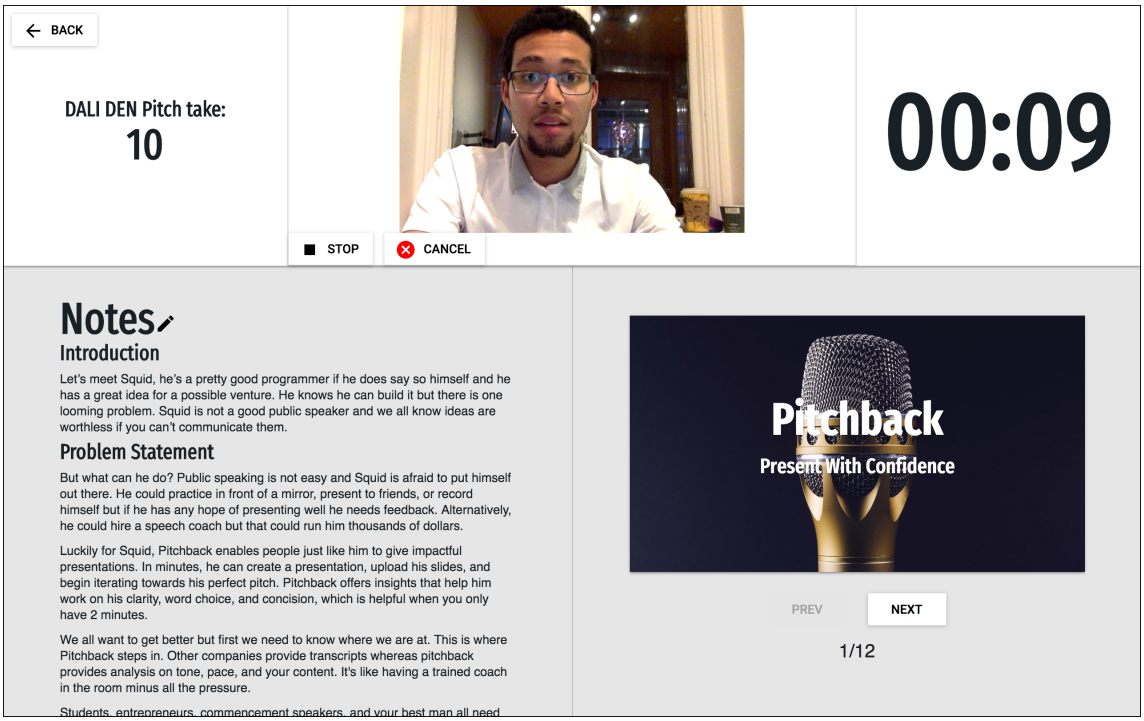

Figure 7: The Record View

Recording in-site gives users access to their presentation materials during a practice session. Users can watch themselves record while keeping an eye on the time, their slides, and notes. The view allows users to practice clicking through their slide show. Users can review the *take* before they choose to upload it to the platform.

Once users upload a *take* they are pushed back to the *review tab* of the *project view*. The takes transcript begins to fill in as the video is being processed in order to make the user interface feel faster and more responsive. This method of displaying intermediate results greatly increased the usability of the platform.

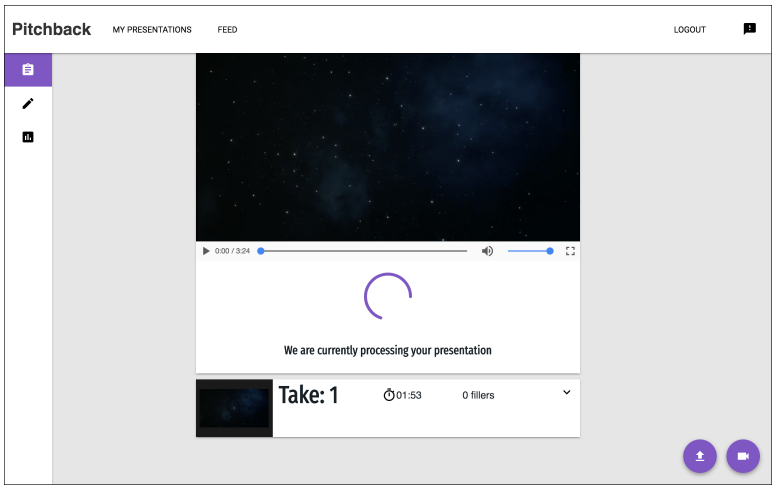

Figure 8: Initial Loader Before Intermediate Results are Received

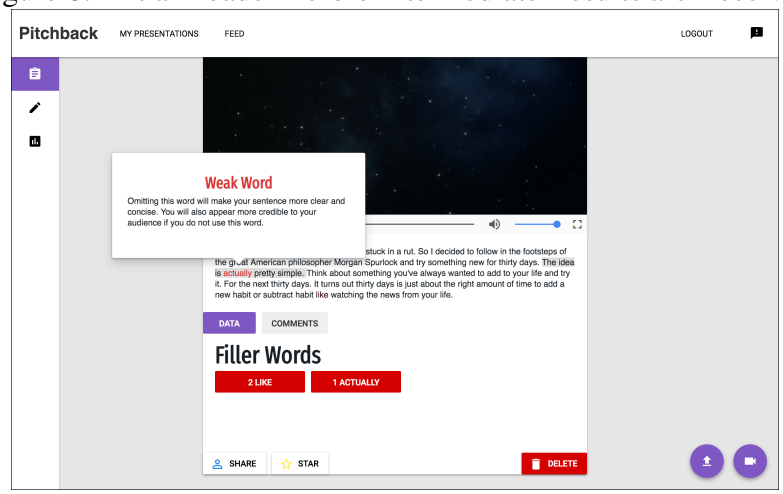

Figure 9: A Partially Populated Transcript with Insights

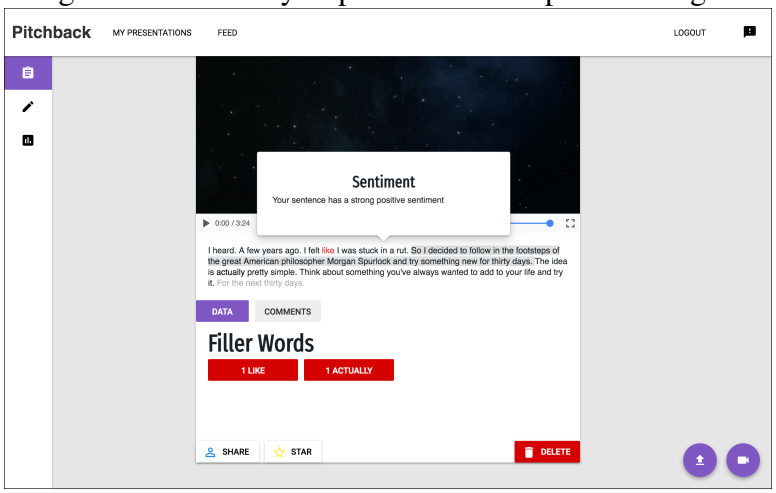

Figure 10: A Partially Populated Transcript

Feed

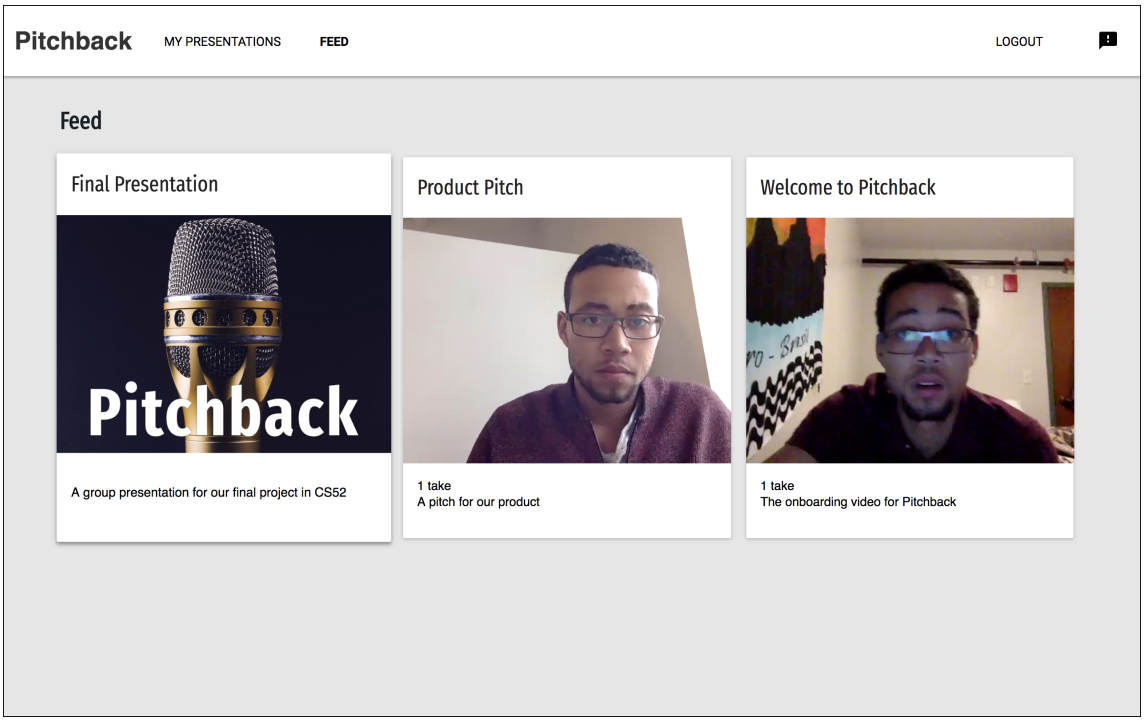

Figure 11: The Feed View

The *feed view* facilitates the peer review process by collecting public presentations. Presentations that show up in the *feed view* have at least one *take* that has been published for review. This view allows users to discover new content on the platform.

IMPLEMENTATION

In building Pitchback as a modern web platform we considered many design patterns for the architecture of our data processing system. This chapter highlights tradeoffs and design choices as well as the technologies in use.

Data Structures

To support the processing and display of presentation data we implemented three key data structures: *project*, *revision*, and *transcript*. The *transcript tree* contains nodes in a hierarchical order of types: *transcript*, *sentence*, and *word*.

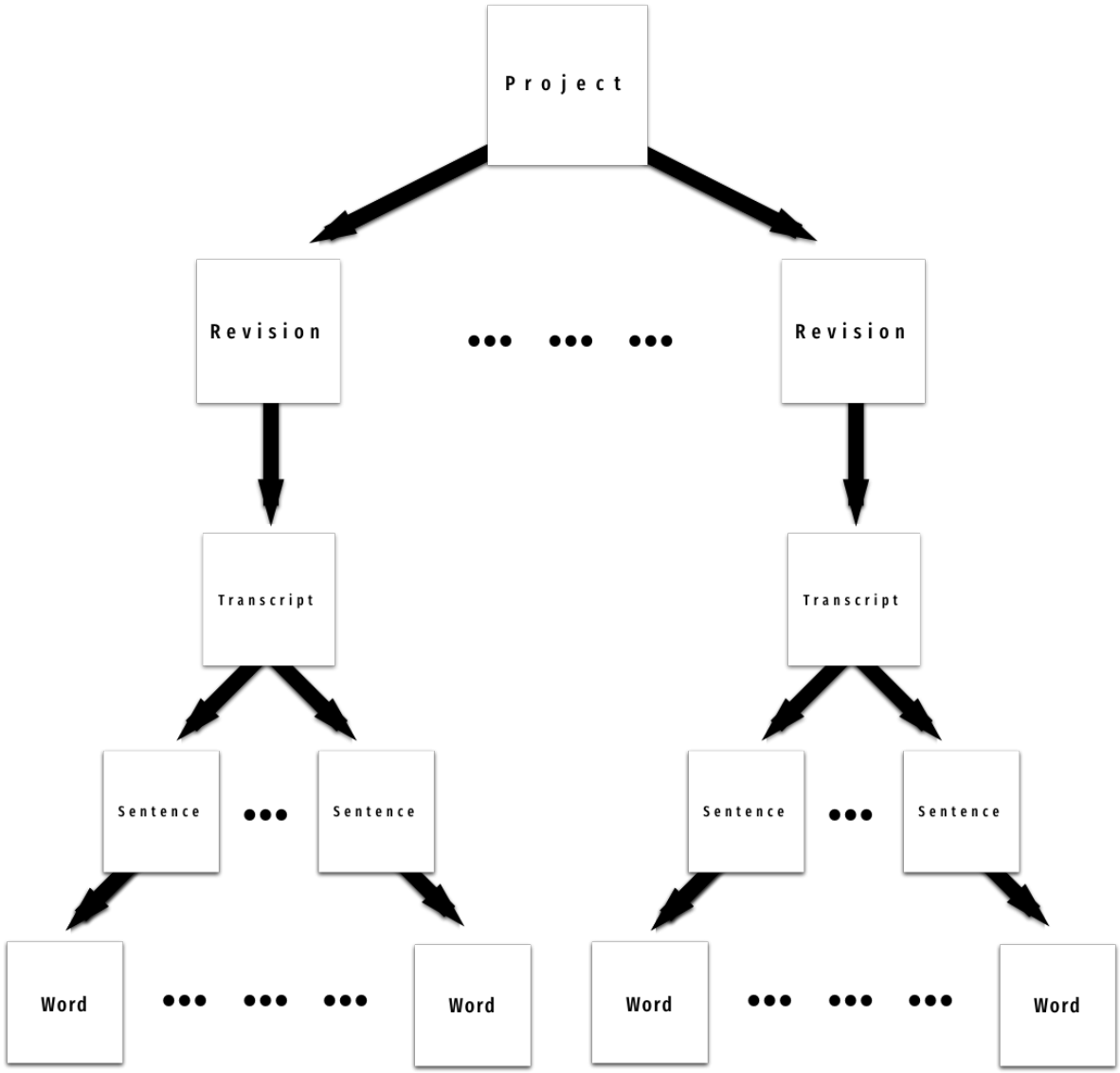

Figure 12: Data Structures

Project

The *project* data structure is the top level node of a multiple *revision* tree. The *project* schema holds references to each *revision* in the *project* as well as *project* metadata. Finally the *project node* stores markdown notes as well as a link to the users slides stored on **Amazon S3**. The schema is as follows:

name: String representing the title of the project.

description: String providing a description of the project.

duration: Integer target time for the presentation.

slides: String URL for user slides stored on **Amazon S3**

revisions: Array of *revisions* (see schema below)

Revision

The *revision* data structure is the next node in the *project tree* hierarchy. The data for each *take* uploaded by the user is stored in the *revision* database structure. In each *revision* metadata such as whether or not it is starred or published is stored. This structure also stores the *transcript tree* - the core data structure behind Pitchbacks presentation insights. Finally, the processing field is used to signal the client as to whether or not there is more data to render. The schema is as follows:

name: String respresenting the title of the project.

published: Boolean indicating if the *revision* is public

starred: Boolean indicating if the *revision* is favorited

words: Array of Strings containing the words transcribed for this *revision*

transcript: *Transcript* schema discussed below,

videoURL: String URL for the video stored on **Amazon S3**

comments: Array of comments

Transcript

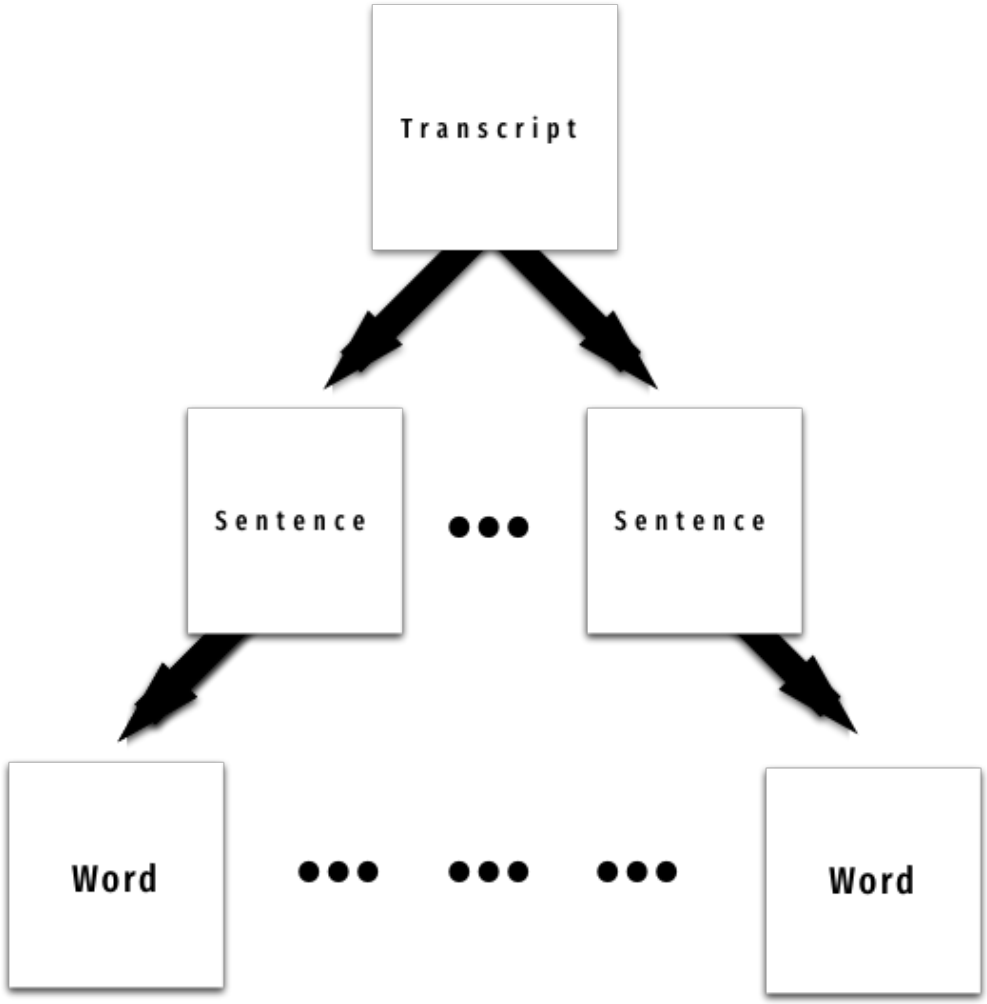

Figure 13: Transcript Tree Data Structure

We implemented the *transcript rooted tree* data structure to allow for the processing and storage of presentation transcripts in a way that preserves its hierarchical nature. There are currently three types of nodes in this tree structure: *transcript*, *sentence*, and, *word*. The root node is the only *transcript* node in the tree. This node maintains an ordered list of

sentence nodes as children. Similarly, *sentence* nodes contain an ordered list of *word* nodes as children. Finally, *word* nodes are leaf nodes in the tree. Each node in the tree contains the same data fields. The schema is as follows:

key: String unique id for the node,

type: String one of *transcript*, *sentence*, or *word*,

index: Array of two elements: start index and end index

children: Array of child *transcript* nodes,

features: Hash Table of features to counts

timestamps: Array of two elements: start time and end time

The *transcript* class is implemented as a functor, or simply a class that implements map (see figure 14). The map function accepts a function as an argument and applies the function to each node of the tree during a depth first traversal.

```
function map(func, node) {
  node.children;
  if (node.children) {
    node.children = node.children.map(child => map(func, child));
  }

  return func(node);
};
```

Figure 14: Transcript Data Structure Map Function

The function passed to map must return a valid *transcript* tree node. Since each function returns a the same type (tree node), we can compose multiple functions for processing. This is important for separation of concerns and efficiency. Functional composition allows us to write separate functions for tallying filler words, calculating words per minute, formatting data, etc. Since the functions are composed (see figure 15) we only traverse the transcript tree once during our processing.

```
// Function Composition using compose
function processTranscript = compose(
  formatPopup,
  countFilerWords,
  wordsPerMinute,
);

// The above is equivalent to
function processTranscript = wordsPerMinute(countFilerWords(formatPopup))
```

Figure 15: Transcript Data Structure Compose

We use this pattern throughout the code base for processing transcript tree nodes.

Microservices

To address separation of concerns, allow for multiple languages, and lend the system to easy scaling we chose a microservice based architecture. Our backend consists of four microservices.

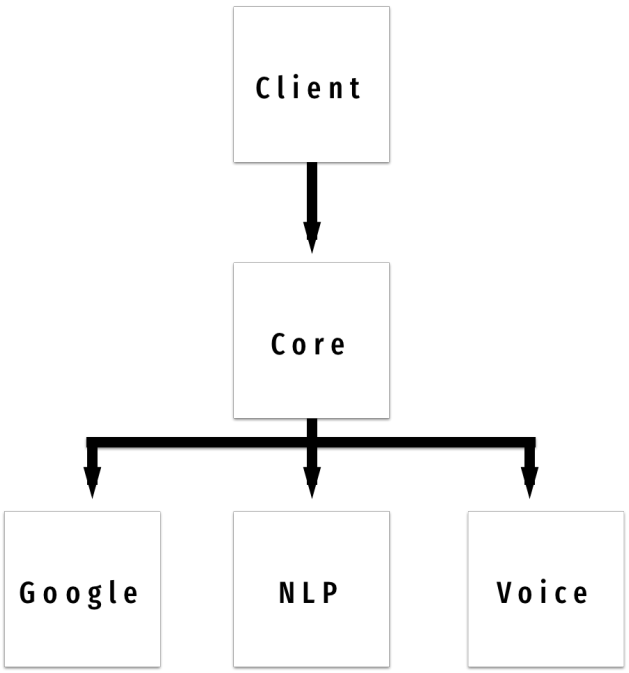

Figure 16: Microservice Architecture High Level Overview

Most of the application logic lies in the node.js service entitled **core** in figure 16. The arrows represent the direction of requests and data in the application. The **core** service is the main interface between the client and the backend. We are using a session cookie based authentication scheme. Authenticated requests pass through the core service. We use asynchronous processing upon video uploads to allow for near real time feedback to begin populating the view. This necessitated a module that is capable of coordinating requests and combining results. Jason Feng implemented the three backend microservices that provide additional data processing API's in python using flask. A high level overview of the function of each service is as follows:

Core	Natural Language	Google APIs	Voice
• Serving up the client • Authentication • Database queries • Streaming to Watson services • Coordinating asynchronous operations from other microservices • Combining results from other microservices	• Processing sentences received from Watson services • Processing entire transcripts received from Watson services	• Interfacing with Google Cloud API's • Returns sentiment scores from the Google Sentiment API	• Returns timestamps for tonal analysis features

Figure 17: Microservice Responsibilities High Level Overview

Client

Single Page Application

The client is a single page application built using React [rea], a JavaScript rendering framework. React uses functional components to split rendering logic. We have coupled React with the popular state management framework Redux [red] as well as React Router for

frontend routing. The routes we have implemented correspond to the views discussed in the Design chapter.

Module Bundling and Loading

It was critical for our app to be fast and maintainable so we chose Webpack [web] as our industry standard bundler. Webpack supports code splitting, a technique used to deliver JavaScript to the browser as needed in single page applications. This decreases the initial load time for each page by delivering smaller payloads, thus combating page weight problems experienced by large applications.

Transcript Rendering

A crucial user interface element is the transcript. In order to display the hierarchical structure discussed in the Data Structures section we use a depth first traversal to render elements. This is necessary because we display features as popups when the user hovers over a word or sentence. Sentence features must be displayed when the user hovers over any portion of the tagged sentence. Similarly, filler words should trigger a popup independent of the surrounding sentence. The depth first traversal places each word in a span tag. Each sentence is also a span tag with nested words.

Server

Node.js is a JavaScript environment that allows us to run JavaScript outside of the browser. JavaScript is an asynchronous, non-blocking, event driven, language. This model allows JavaScript to manage multiple requests/users in a performant manner and hence makes it an invaluable language for client and server-side web development.

The core microservice is built with node.js and express.js. Express is a popular routing framework for node.js web applications that allows requests to be serviced by middleware functions. It is critical that we use a framework that lends itself to maintainability and

scalability. The middleware reduces the need for repeating logic and thus allows for DRY code on the server. Middleware functions are registered to REST endpoints. The functions can be used to respond to the request, pass it along, or handle error cases.

MongoDB: is our NoSQL datastore of choice for storing documents.

FFmpeg: is a cross-platform solution for video and audio recording and conversion. FFmpeg can output the converted file on a stream which was invaluable for our data processing pipeline performance.

Cloud Services

Pitchback's backend is dependant upon multiple cloud services.

Amazon S3: is our file storage platform. Video and presentation slide uploads are stored and served from S3.

Amazon Elastic Transcoder: converts user uploaded videos into an mp4 format that is accessible by all browsers. Files are converted from Amazon S3 and thumbnails are generated in periodic intervals.

Watson Speech To Text: transcribes audio streams and returns intermediate results with metadata such as timestamps and possible alternative transcripts.

Data Flow

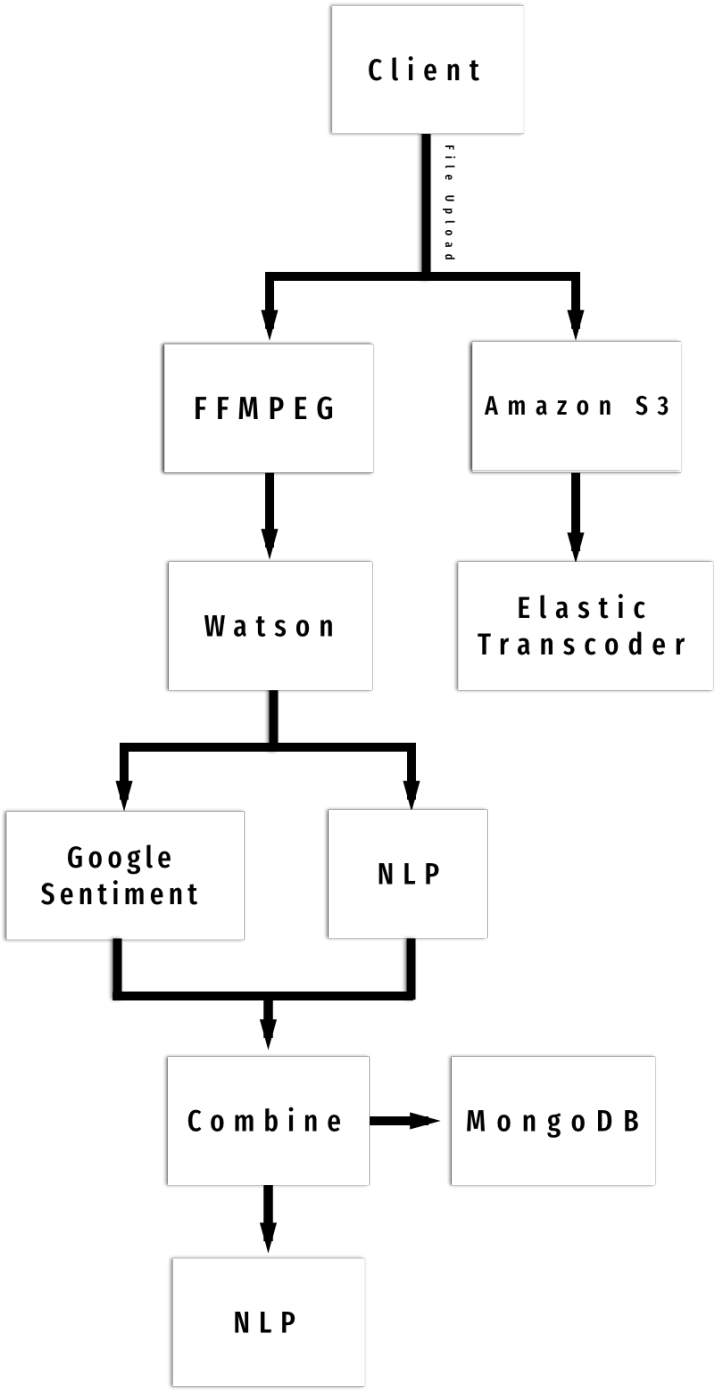

Figure 18: Data Processing Pipeline

As previously mentioned, usability of the platform from a user perspective entails performant data processing. To achieve the fastest possible results we have designed a data flow model that runs computations asynchronously on two independent streams: the *file upload stream* and the *transcription analysis stream*. Both streams are initiated by a video upload. Upon video upload a *revision* object is created in the database and both streams store intermediate results in the *revision* object. The frontend receives these intermediate results by polling an endpoint on the **core** node.js service to receive the most recent copy of the *revision* object.

File Upload Stream

The video file stream is split and a piped to **Amazon S3** via their multipart file upload api. Upon completion the video is converted by the **Amazon Elastic Transcoder** service. The transcoded video resource URL is stored under the videoURL field in the *revision* database object.

Transcription and Analysis Stream

Conversion The video file stream is also piped to a child process running **FFmpeg**. This child process converts the video stream into the wav audio format.

Transcription The **Watson Speech to Text** service exposes a websocket for transporting data and receiving results. When the stream is created we initialize the *transcript* of the current *revision* object. The output from the **FFmpeg** conversion is piped to Watson. In Node.js streams are event emitters. The Watson stream exposes two events of interest: *results* and *close*. The *results* event fires with intermediate results from Watson. These results are the transcript of a sentence unit. A sentence unit is a portion of the audio for which there is no sufficient pause in speech. We heuristically create *sentence nodes* with their child *word nodes* from these sentence units.

Sentence Tagging Each results event from the Watson stream triggers two concurrent post requests for sentence feature tagging. The first is a request to the **natural language processing** microservice at its **/grammar/sentence** route. The child *word nodes* of the *sentence node* may contain filler, weak-word, adverb, or, adjective. The second request is to the **google sentiment** microservice at its **/language/sentiment** route. The *sentence node* may contain the tag sentiment with a value ranging from -1 to 1. This value is the score given by the **google service** on a scale of negative to positive sentiment.

Combine To coordinate these requests we are using the *Promise API*, a popular JavaScript pattern for managing asynchronous code. The Promise API wraps asynchronous code and provides two functions: then and catch. The user provides a function when the promise is created to deal with success and failure cases respectively called resolve and reject. The then function is called on success with an optional single argument determined by the user. The error function is similarly called on failure. Promises are in one of three states at any given time: pending, resolved, or rejected. Both sentence tagging requests are wrapped in Promises. We use the*Promise.all* function which takes a list of promises as an argument. Promise.all waits for each promise to resolve before calling then. Therefore we pass our two sentence tagging requests to *Promise.all* as a list and wait for both responses.

Upon receiving both responses we now have two separate transcripts both representing the same *revision* but with different features tagged on its *sentence* and *word nodes*. Therefore we need to combine both transcripts into one before persisting to the *revision* database object. To combine the transcripts we take advantage of the unique keys on each node of the tree. Arbitrarily we traverse the sentiment sentence tree first and create an inverted index of key to node for a sentence if the absolute value of the sentiment score is greater than 0.4. The filter's purpose is to tag sentences that have either a strong positive or negative sentiment and ignore those that are more or less neutral. After we have created this index we traverse the sentence tree returned from the **natural language processing** service. For

each node, if the current key is in the inverted index combine the feature dictionaries of both nodes. Note that we traverse two shallow trees of depth 2 to generalize the algorithm. The generalized algorithm allows for a more complex data flow in the future with the possibility of sending entire paragraphs for more context in sentiment and grammar analysis. It is easy to see that both trees are traversed once and therefore the combination algorithm runs in O(n) where n is the number of nodes in the tree time. While traversing the *natural language processing sentence node* we maintain counters for each feature and its corresponding word. This count will make insertion into the overall transcript more efficient as we will soon see.

The combined *sentence tree* must be persisted to the *revision* database object for intermediate results to begin rendering on the frontend. There are two cases: we currently have the first sentence or there is an existing transcript in the database from previous cycles of this process. The first case is trivial in that we simply create a *transcript node*, insert the current sentence as the first child node of the transcript, and use all of its metadata as the metadata for the entire transcript. In the second case, we are adding a *sentence node* to a *transcript tree*. Thus, we must shift the indices of all nodes to be relative to the transcript rather than to the sentence itself. We must also add the feature count of the sentence to the total feature count of the transcript. Finally, the end timestamp of the transcript must be revised to the end of the final word of the current sentence. Since we update each child node of the current sentence to shift the metadata relative to the transcript, this step also runs in O(n) time.

Transcript Tagging The Watson *close* event signals the end of the file stream and the completion of transcription. Similar to the logic for combining the API calls, we use *Promise.all* to wait for the final requests and combine steps to be completed. Once complete we make a request with the entire transcript to the **/grammar/groups** route of the **natural language** service for higher level feature tagging for which the entire transcript

may be necessary. This service returns a transcript with *sentence nodes* possibly tagged as passive and/or weak-phrase. Using our tagged transcript, we do a final pass over the tree to calculate the words per minute for the presentation.

The final transcript is persisted to the database and the revision processing field is marked as finished to signal the frontend to stop polling.

Discussion and Results

Data Processing Performance

As previously discussed, the processing of our feedback in a performant manner was of the utmost importance in order to create a usable platform. In implementing our multiple stream data processing pipeline we saw significant performance improvements over prior attempts to analyze videos serially. A major contribution to this performance increase was the ability to display intermediate results on the client. This feature made our application feel much faster.

User Feedback

In implementing the client we made sure to follow human centered design iterative processes. We periodically received feedback from a small set of users to validate our design decisions. In informal discussions with 12 of these users we found that our presentation insights were helpful in putting together a presentation. Choosing material design created a look and feel that users are already familiar with due to its popularity. Therefore, users were generally able to navigate our platform without an onboarding tutorial. Observation based user studies verified these claims as users were able to use the platform under observation with little to no guidance. Our insights are presented inline with the transcript. Users found this presentation of the data to be helpful in navigating and consuming our

feedback. The layout of takes as shown in figure 4 was mentioned by many users to be useful in tracking their presentation trajectory and understanding gaps in their delivery. An obvious extension to this work would be to launch a larger beta test.

FUTURE WORK

Transcript

The effectiveness of our platform is clearly limited by the quality of the video transcript. To mitigate this issue we propose three extensions: incorporating watson alternative results, allowing users to edit and reprocess the transcript, and a paragraph transcript tree node.

Watson Alternative Results

The **Watson Speech to Text** service returns a list of words during the transcribing process in order of confidence. We simply use the highest confidence word to assemble a sentence. It is rare that the correct transcript for a given presentation is comprised of solely top confidence words. An obvious extension would be to consider all of the possible words to create our transcript. A system for doing so could include exposing these options to the user in a drop down menu, or using grammar and context to select a word from the list.

Edit Transcript and Reprocess

Integrating a feature into the current user interface for editing and reprocessing the transcript is another option for increasing the quality of the transcript. If users may edit their transcripts, the transcript tree data structure must implement insert, update, and delete.

Paragraph Tree Node

Creating another transcript tree node type paragraph, may be beneficial for data processing. Paragraph nodes would allow for specific contextualization of the sentence child nodes for

feature detection. Creating this node would require advanced natural language processing techniques to identify topic changes in a presentation transcript.

Voice Insights

Effective communication and engaging presentation style encompasses more than word choice. Jason Feng has implemented a microservice that generates timestamps for tonal features such as monotone and rising tone. An obvious next step would be to integrate this service into the data processing pipeline. A challenge in doing so is matching timestamps with word and sentence nodes in the transcript tree in a performant manner. Tonal processing is independent of the file upload and transcription analysis streams. Tonal processing would therefore run on its own stream adding to the asynchronous processing complexity.

Trends

Insights are limited to the scope of an individual take of a presentation. Pattern recognition across takes requires a new tree structure that is a collection of transcript trees. This new tree facilitates processing feature trends over time and a greater context for understanding the presentation.

Group Permissions

Our sharing feature publishes a take to the entire user base. It would be convenient for groups of users to publish takes to each other. This feature would facilitate Pitchback's use with classes, student groups, companies, teams, etc.

Conclusion

Human discourse is riddled with speech disfluencies that negatively affect comprehension [Corley and Stewart, 2008]. Inspired by recent advancements in natural language process-

ing and speech recognition we designed “Pitchback”, a web platform to programmatically provide public speaking feedback for improving speech patterns and communication style. In speaking with a small set of users we have shown that pairing an intuitive user interface with a performant data processing pipeline helps users prepare for public speaking events.

Bibliography

Martin Corley and Oliver W. Stewart. Hesitation dis-fluencies in spontaneous speech: The meaning of um. *Language and Linguistics*, 2(4):589–602, 2008.

Grammarly. https://www.grammarly.com/.

Hemingway App. http://www.hemingwayapp.com/.

Thomas Graham Emily Duvall, Aimee Robbins and Scott Divett. *Exploring Filler Words and Their Impact*. http://schwa.byu.edu/files/2014/12/F2014-Robbins.pdf.

Mark Johnson Matthew Lease and Eugene Charniak. Recognizing disfluencies in conversational speech. *IEEE TRANSACTIONS ON AUDIO, SPEECH, AND LANGUAGE PROCESSING*, 14(5):1566–1573, 2006.

Charles Tark and Mark Yoon. *Rhapsodize: Mobile Application for Improving Public Speaking Skills Through Training of Speech Disfluencies*. `http://mark-yoon.com/assets/Final_Paper.pdf`.

Material Design, a. https://material.io/.

React react. `https://facebook.github.io/react/`. Accessed: 2016-12-01.

Material UI material ui. `https://material.io/`, b. Accessed: 2016-12-01.

Redux redux. `http://redux.js.org/`. Accessed: 2016-12-01.

Webpack webpack. `https://webpack.github.io/`. Accessed: 2016-12-01.