SynthLang

Technological Feasibility

April 02, 2026

Project Sponsor Dr. Benjamin V. Tucker

Faculty Mentor Isaac Shaffer

Members

Paul: Lead, Coder

Tyler: Recorder, Coder

Logan: Release Manager, Coder

Jalen: Architect, Coder

Table of Contents

1 Introduction

A person's voice is fundamental to their identity. For individuals who lose the ability to speak due to disabilities, like the late Stephen Hawking, Augmentative and Alternative Communication (AAC) technology provides a vital bridge to the world. For languages like English or Mandarin, finding and training suitable TTS models is no issue; however, for under-resourced languages, such models can be difficult to come by–if they exist in the first place. Our sponsor, Benjamin Tucker, has been researching techniques for creating TTS models for low-resource languages such as South African English (SAE), Afrikaans, and isiXhosa, but he has a problem: his current TTS system uses a legacy implementation of tacotron2. As of 2020, this implementation is no longer maintained by Nvidia and can no longer be run on modern hardware. This is where our project comes in.

Our solution to this problem has two main components: a TTS model training pipeline and a web application. The former will provide a simple framework that Dr. Tucker can leverage to train his own TTS models without having to worry about the intricacies of the TTS system. The latter will host the models trained and allow users to input text and generate speech using said models.

2 Technological Challenges

Designing a flexible TTS model training pipeline is no small undertaking. The team has identified several key technological challenges that must be addressed in order to realize the desired system.

2.1 TTS Framework

Designing a flexible TTS model training pipeline is a significant undertaking because it must accommodate the complex requirements of both acoustic modeling and vocoder synthesis. The primary challenge arises from the need for a robust and modular API that supports the training and fine-tuning of custom models, which is particularly vital for the low-resource languages our sponsor studies. Furthermore, the framework must facilitate transfer learning to ensure high-quality synthesis even when labeled data is scarce. Modularity is essential for the customization of each stage of the pipeline, from preprocessing to modeling. This will ensure that the system remains adaptable as new speech synthesis technologies emerge.

2.2 Computational Resources

Building on the need for a flexible framework, the project faces a major hurdle in securing the computational power required for intensive deep learning tasks. While the sponsor has access to high-performance GPU workstations, university IT policies often prohibit these machines from being used as public-facing web servers. This restriction, combined with the lack of a dedicated budget for persistent cloud hosting, creates a dilemma: the system must provide the accessibility of a web application without incurring the high costs of specialized GPU-enabled hosting. Consequently, the challenge lies in architecting a solution that balances performance and accessibility while strictly adhering to institutional security.

2.3 Web Technology Stack

The complexity of managing these computational resources necessitates a carefully integrated web technology stack to deliver a functional user experience. Developing a web interface for model management and audio generation is challenging because the process is often bottlenecked by the compute-intensive nature of TTS inference. The stack must ensure seamless communication between a responsive frontend and a high-performance machine learning backend, while also providing essential features like real-time audio playback and download capabilities. Moreover, selecting technologies

that prioritize longevity and minimal library overhead is crucial to ensure the application remains stable as underlying standards and browser capabilities evolve.

2.4 Machine Learning Framework

At the heart of this web stack is the machine learning framework, which serves as the engine for all speech synthesis and processing tasks. Choosing the right framework is a critical challenge because it must balance raw computational performance with ease of development and research flexibility. The framework needs to support a wide range of modern neural architectures, such as RNNs, Transformers, and diffusion-based models, while maintaining compatibility with the various TTS libraries utilized in the project. Furthermore, a strong ecosystem and active community support are necessary to troubleshoot complex issues and leverage the latest research breakthroughs, ensuring the pipeline remains at the cutting edge of technological feasibility.

2.5 Python Package Manager

Finally, the stability of the entire system depends on a reliable Python package manager to handle the intricate web of dependencies required by modern deep learning frameworks. The primary challenge is to ensure consistent environments across development and production, which requires the ability to bundle specific Python versions and resolve complex dependency conflicts automatically. Given the rapid pace of updates in the machine learning ecosystem, the team must select a tool that minimizes the overhead of environment setup and deployment. A modern, efficient package manager is therefore essential to prevent dependency conflicts and ensure the long-term maintainability and reproducibility of the project's software stack.

3 Technological Analysis

3.1 TTS Framework

Our project requires a flexible API for training acoustic models and vocoders. While we are still shopping around for specific models, it is important that we have an API that is flexible and allows us to customize each part of the training/synthesis pipeline. This flexibility will also allow our client to more easily switch acoustic model generation technologies as new ones are developed.

3.1.1 Desired Characteristics

To evaluate candidate technologies, the following criteria were identified:

- **Flexibility and Modularity:**
  The framework must allow customization of each stage of the TTS pipeline. This is particularly important for under-resourced languages, which may require non-standard preprocessing and modeling techniques.
- **Support for Transfer Learning:**
  Given the limited availability of labeled data, the system must support fine-tuning of pretrained models. Transfer learning is essential for achieving acceptable performance in low-resource settings.
- **Ease of Use and Development Speed:**
  The framework should provide high-level abstractions and prebuilt pipelines to reduce development time. This is important given the team's limited prior experience with deep learning-based speech synthesis.
- **Documentation and Community Support:**
  Comprehensive documentation and an active user community are necessary for troubleshooting and ensuring continued usability.
- **Maintenance and Longevity:**
  Active development and long-term support reduce the risk of dependency issues and obsolescence.
- **Open Source Availability:**
  The solution must be open source to allow full customization and ensure accessibility for academic use.

3.1.2 Alternatives

a) Coqui TTS

Coqui TTS is an open-source framework designed to provide a unified interface for training and deploying multiple TTS model architectures. It supports a wide range of acoustic models and neural vocoders, along with built-in tools for dataset preprocessing and training.

Coqui has been widely used for rapid prototyping, multilingual synthesis, and voice cloning. However, the original repository is no longer actively maintained by its founding organization, and ongoing support is provided through community-maintained forks.

b) SpeechBrain

SpeechBrain is a PyTorch-based open-source speech processing toolkit developed by the academic research community. It supports a wide variety of speech tasks, including TTS, automatic speech recognition (ASR), and speaker recognition.

SpeechBrain provides a highly modular architecture that allows developers to construct custom pipelines from low-level components. It is actively maintained and widely used in research environments, making it a strong candidate for flexible and extensible system design.

c) ESPnet

ESPnet is an open-source end-to-end speech processing toolkit developed primarily for academic research. It supports a variety of tasks, including TTS, ASR (automatic speech recognition), and speech translation, and provides reproducible training recipes for multiple model architectures such as Tacotron2 and FastSpeech2. ESPnet is widely used in research contexts due to its emphasis on reproducibility, standardized training pipelines, and support for multilingual and low-resource scenarios.

3.1.3 Analysis

a) Coqui TTS

Coqui TTS provides strong support for rapid prototyping due to its high-level API and extensive built-in functionality. It includes pretrained models, dataset preprocessing tools, and simplified training workflows, enabling the development of a working system with minimal overhead. The framework also supports transfer learning and allows for swapping between different acoustic models and vocoders. Its primary drawback is the

lack of official maintenance, which introduces potential risks related to long-term stability and compatibility.

b) SpeechBrain

SpeechBrain offers a high degree of flexibility and is well-suited for building custom pipelines. Its modular architecture allows developers to define and modify every component of the training process, making it a strong candidate for research-oriented applications. In addition, SpeechBrain is actively maintained and supported by a robust community. However, it lacks the out-of-the-box usability of Coqui, requiring significantly more development effort and domain knowledge to implement a complete TTS pipeline.

c) ESPnet

ESPnet provides a highly structured and reproducible framework for training TTS models, with particular strengths in low-resource and multilingual scenarios. Its training recipes make it especially useful for transfer learning and dataset augmentation. However, ESPnet has a steep learning curve and requires a deeper understanding of machine learning and speech processing concepts. It is less user-friendly than Coqui and does not provide the same level of high-level abstraction. This could result in slower initial development and possible project failure.

3.1.4 Selected Approach

Each of the evaluated frameworks offers distinct advantages. Coqui TTS excels in ease of use and rapid prototyping, while SpeechBrain and ESPnet provide greater flexibility and long-term maintainability.

Criterion	**Coqui TTS**	**SpeechBrain**	**ESPnet**
Flexibility / Modularity	3	5	5
Transfer Learning Support*	4	5	5
Ease of Use*	5	2	2
Documentation / Community	3	4	4
Maintenance / Longevity	2	5	5
Open Source*	Yes	Yes	Yes

*** Indicates priority criterion**

Based on this analysis, Coqui TTS was selected as the primary framework for initial development. This decision is primarily driven by its ability to support rapid prototyping and its extensive built-in functionality, which reduces development overhead.

However, due to concerns regarding long-term maintenance, the system will be designed with a modular architecture that allows for future integration of alternative frameworks such as ESPnet. This approach balances short-term feasibility with long-term flexibility.

3.1.5 Feasibility Validation Plan

To validate the feasibility of the selected approach, the team developed a test script using the Coqui TTS framework to demonstrate its suitability for rapid prototyping and performance evaluation. The script utilizes a pretrained model provided by the framework and executes inference across multiple text inputs of varying lengths.

The test script was designed to benchmark inference performance on both CPU and GPU configurations. It measures execution time for each test case while ensuring accurate timing through device synchronization and warm-up runs. Additionally, the script generates and saves audio outputs for each test case, verifying the end-to-end functionality of the synthesis pipeline.

By comparing inference times and confirming successful audio generation, this test demonstrates that Coqui can be quickly integrated, produces usable output, and supports both CPU and GPU execution environments. These results provide evidence that the framework meets the project’s requirements for rapid prototyping and functional speech synthesis.

Further validation will involve extending this prototype to include training workflows using custom datasets provided by the client, ensuring that the framework can support the full pipeline from training to deployment.

3.2 Computational Resources

The core challenge is to web-enable a powerful Text-to-Speech (TTS) application, based on the client's existing `tacotron2` program. The client, Dr. Tucker, has a high-performance faculty workstation with NVIDIA Titan GPUs that is ideal for running the TTS model. However, university IT policy prohibits this machine from being used as a web server. Critically, **there is no budget for dedicated, persistent server hosting, especially one that offers GPU resources**. Therefore, the primary issue is architecting a solution that provides the accessibility of a web application, without incurring any hosting costs.

3.2.1 Desired Characteristics

An ideal solution must balance performance, accessibility, and resource constraints. Based on the project goals, we can establish the following key characteristics as metrics for evaluation:

- **Minimal Hosting Cost:** The solution aims to avoid reliance on paid, persistent cloud services, prioritizing cost-effective hosting strategies of the server components.
- **Performance:** The system must efficiently use a server-side CPU to synthesize speech with minimal delay.
- **Accessibility & User Experience:** The final product must be a web application accessible from a standard browser. The primary user interface should be simple: a user enters text, and the system returns playable audio (there is no training the model here).
- **Compliance with IT Policy:** The architecture must strictly adhere to the rule that no web services are hosted on the client's faculty-owned computer.
- **Maintainability & Scalability:** The chosen technology stack should be mature, well-documented, and suitable for long-term maintenance. The architecture should be scalable to handle multiple users or larger processing jobs in the future.
- **Feasibility:** The proposed solution must be achievable. The project's main goal is to tackle the technical risks associated with the chosen path.

3.2.2 Alternatives

Based on our research and the key characteristics, we identified two primary architectural approaches.

a) Fully server-side processing

This model uses a lightweight Jinja2 templated with HTMX frontend that calls a backend server for all heavy processing. The backend, Fastapi, would run on a server, perform the TTS synthesis on its CPU, and return the audio. This architecture requires a continuously running, persistent server process.

b) Hybrid client-side processing with WebGPU

This approach has the frontend download the TTS model to the user's browser. The application then uses the WebGPU API or the server's GPU to run the model directly on the user's own GPU (in Dr. Tucker's case, his Titan cards). This offloads all major computational costs from the server to the client machine, requiring a server only for serving the initial static files.

3.2.3 Analysis

We evaluated both alternatives against our desired characteristics:

- **Performance:** Alternative A's performance is dependent on the server's CPU, but it is consistent and reliable. Alternative B could theoretically be faster for Dr. Tucker by using his powerful local GPUs, but it is subject to the overhead of the browser, JavaScript, and WebGPU, and performance would vary drastically for other users with less powerful machines.
- **Accessibility:** Alternative A is highly accessible; any user with a browser can use it. Alternative B is far less accessible, as it would only function for users with powerful GPUs and browsers that support WebGPU. However, it should be noted that this application is being made for 1 person and their hardware specifically, so while it would still be less accessible, for the moment, it still fits within Dr. Tucker's own parameters.
- **Compliance:** Both alternatives are fully compliant with IT policy, as the web server component for both resides on a web service not on Dr. Tucker's machine.
- **Feasibility:** Alternative A is highly feasible. The technology stack (Jinja2, FastAPI, PyTorch) is mature and well-suited for this task. The workflow is well-understood. Alternative B carries significant risk; converting complex PyTorch models to a web-compatible format is a difficult, error-prone process, and WebGPU is still a new and evolving technology.

3.2.4 Selected Approach

Our investigation into the two primary architectures revealed a clear trade-off. The fully server-side approach (Alternative A) offers stability and reliability using a proven technology stack, while the hybrid client-side approach (Alternative B) presents an opportunity to leverage the user's local GPU power at the cost of significant complexity and risk. To make a final decision, we evaluated both alternatives against our predefined characteristics in the table below, using a simple 1-5 rating (where 5 is best).

Desired Characteristic	Server-Side	Client-Side (WebGPU)
Minimal-hosting cost*	5	3
Performance	3	5
Accessibility	5	3
Compliance (IT	5	3

Policy)		
Maintainability	5	2
Feasibility/Risk	5	1

*** Indicates priority criterion**

Ultimately, the stability and broad accessibility of a server-side architecture outweigh the performance benefits of local GPU acceleration. Alternative A aligns perfectly without "Minimal Hosting Cost" and "Compliance" requirements by utilizing standard CPU resources, whereas Alternative B's reliance on WebGPU introduces excessive complexity and compatibility risks. Based on this analysis, we have chosen **Alternative A (Server-side)** to ensure the project remains technically feasible and maintainable within our constraints.

3.2.5 Feasibility Validation Plan

In order to demonstrate the feasibility of fully server-side inference, the team created a prototype script that runs inference both on the CPU and the GPU of a local machine. This is to show that inference can be accomplished in reasonable time without the usage of a dedicated GPU. Below are the results from the script:

```
Running on CPU
text_1: 3.1103s → saved to ./test_audio/cpu_text_1.wav
text_2: 2.3375s → saved to ./test_audio/cpu_text_2.wav
text_3: 4.7872s → saved to ./test_audio/cpu_text_3.wav
text_1_2: 5.0929s → saved to ./test_audio/cpu_text_1_2.wav
text_1_2_3: 10.0415s → saved to ./test_audio/cpu_text_1_2_3.wav

Running on CUDA
text_1: 0.2072s → saved to ./test_audio/cuda_text_1.wav
text_2: 0.1621s → saved to ./test_audio/cuda_text_2.wav
text_3: 0.2605s → saved to ./test_audio/cuda_text_3.wav
text_1_2: 0.3336s → saved to ./test_audio/cuda_text_1_2.wav
text_1_2_3: 0.5829s → saved to ./test_audio/cuda_text_1_2_3.wav

===== SUMMARY =====
Test Case       Chars      CPU (s)    GPU (s)    Speedup
text_1          94         3.1103     0.2072     15.01
text_2          74         2.3375     0.1621     14.42
text_3          149        4.7872     0.2605     18.38
text_1_2        169        5.0929     0.3336     15.27
text_1_2_3      319        10.0415    0.5829     17.23
```

As can be seen on the previous page, while inference time is significantly shorter on the GPU (~16x speedup), CPU inference times for medium sized passages are still reasonable.

3.3 Web Technology Stack

Our final product requires a functional website where a user will be able to input text in a variety of languages and receive generated speech. The website will have to have a functional front end and back end, and be able to interface with the acoustic model that we have trained to generate the desired languages.

3.3.1 Desired Characteristics

Here are some parameters the website must fit, and some characteristics that would result in a better experience for the client.

- **Fully functional front and back end:** This website will be a single page with a text window to input text and a dropdown menu to select a language/voice. The website needs to be hosted somewhere and have a functioning back end to communicate with the acoustic model and generate audio.
- **Speedy Audio Generation:** The most important thing, beyond basic functionality as stated by our client, is speed. Their current design takes minutes to generate audio because it is bottlenecked by the CPU. Where we host will not have access to GPU power, so we will need to find a way to quickly generate audio using just CPU power. Another way to speed up the user experience is to chunk the audio generation. By breaking a paragraph into sentences, we can play audio for the user while the rest of the input text is still being converted into audio.
- **Audio Settings:** Clicking a button and having audio play after waiting for generation isn't super useful. Some additional functionality with the audio clips would be helpful
    - Allow for pausing, restarting, rewinding, fast forward.
    - Buttons to regenerate the audio with different voices
    - Storage of previously generated audio, or storage of previously used text
    - Downloadable audio in multiple formats
    - A loading bar to indicate to the user how long the generation will take
- **Longevity:** The client wants a website that will work for a long time with minimal input/changes. So, using minimal libraries, or libraries that are well maintained and stable would be best.

3.3.2 Alternatives

We will now evaluate some potential technology stacks against our criteria, mainly speed of generation for the client.

Frontend Technologies

a) React

React is a javascript framework made for front end development. It was released in 2013, by Jordan Walke. It allows for modern, responsive websites, and it is what the team has the most experience in.

b) HTMX

HTMX allows for embedding functionality directly into the code HTML code, and then calling those HTML pages from the front end; it is not a framework. It allows for close integration with the backend, preventing a problem of having the frontend and backend be two separate applications.

Backend Technologies

a) FastAPI + WebSocket

FastAPI was developed in 2018 by Sebastián Ramíez to be a scalable, and most importantly, fast Python backend framework. It has been used by numerous huge companies to power microservices, and it is also the most used backend when dealing with machine learning and AI, because it handles large, complex data sets so efficiently.

This is included with FastAPI, because FastAPI provides native support for asynchronous code, and WebSocket. WebSocket is a python library that opens a single connection and communicates both ways. Different from a standard HTML page, which opens a new connection every time there is a change, and closes after updating. It is used in any real time website, live chat, video games, etc.. It would allow us to break up the text input into chunks and play them back to the user in real time as they are generated.

b) Django

Django is a Python backend developed in 2005 by Adrian Holovaty and Simon Willison. It is a framework that comes with a lot of features out of the box, and is best suited for large scale projects and projects that need a lot of functionality.

c) Flask

Flask is a Python backend developed by Armin Ronacher and released in 2010. It is much lighter than Django and has an easier learning curve. It doesn’t come with as much built-in functionality, and this better suits it for smaller projects.

d) ONNX

ONNX (Open Neural Network Exchange) is a format developed by Facebook and Microsoft to speed up models. It allows for specific hardware optimizations and therefore would allow us to optimize, allowing us to run just on CPU power if necessary, or optimize for speed with a GPU if we have access to one.

e) Pytorch

Not using ONNX, and just PyTorch. In the last few years, PyTorch has added torch.compile(), which can increase the speed in a similar way to ONNX. PyTorch would allow us to use fewer libraries and dependencies.

3.3.3 Analysis

React vs HTMX: These two technologies have very little impact on the client's experience using the app, as we can design a quality frontend with either of these technologies, so we will discuss ease of use for the developers and hosting. React is the industry standard, and it is what we are most familiar with, so speed of development would be faster. HTMX is new to us, but it does reduce the overhead because HTMX allows us to host everything server-side. If we use React, the backend and frontend would need to be hosted and developed separately, and with our project potentially being hosted by NAU ITS or elsewhere, this added complexity could cause a headache for our client, now and in the future.

ONNX vs PyTorch: While it is true that ONNX might be faster with some configuration, it will need to be configured again if anything about the server changes. Because we don't know what the client will do with our website after we are done working on it, the server might not always stay the same. Another downside of ONNX is that it is not maintained anymore, and people have reported an overall buggy experience. The extra complexity and lack of maintainability make ONNX hard to choose.

WebSockets can be used with any of the backends, but only FastAPI has native support with no external libraries. With Django, the best choice is to use the Django Channels library, and with Flask, something like Flask-Sock or Flask-SocketIO. While WebSocket would allow for a better user experience, they do not actually speed up the generation of audio. Having extra libraries and extensions of our backend bring in more place for future bugs, and version changes to break our website. So WebSockets might not be worth using unless we are using FastAPI.

Flask vs Django vs FastAPI: Django and Flask have no significant speed differences, but Django adds a lot of added complexity, and is more well-suited for a full functional website, with users, and security, and admins. Flask is lightweight and easily

customizable, and would be easy to add additional features too if desired. FastAPI could be multiple times faster for audio generation than the other two frameworks, and natively supports asynchronous communication, and therefore WebSocket. It is also the most used backend for machine learning purposes. All three of these frameworks are well-maintained and unlikely to break anything in the near future.

Criterion	Django	Flask	FastAPI	WebSocket	ONNX	PyTorch
Speed of Generation*	2	2	5	5	5	3
Maintenance / Longevity	5	5	4	4	2	5
Complexity	4	2	2	4	4	2

*** Indicates priority criterion**

3.3.4 Selected Approach

After conducting analysis on our chosen approaches, we are choosing to proceed with **FastAPI, No WebSocket, and PyTorch**.

FastAPI is an easy choice. It isn't more complicated than the other frameworks, while also allowing for much greater speeds and async communications.

We have decided to not use WebSocket, at least initially. It is harder to set up and adds another dependency that could break our website down the line. If we have time at the end of the project, testing WebSocket to improve user experience is possible.

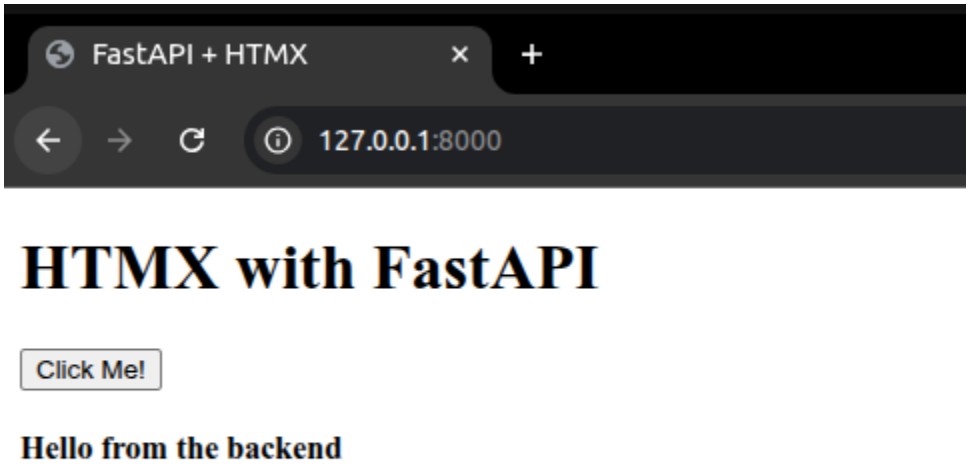
While ONNX can be very fast, the lack of maintenance is a red flag. More importantly, ONNX needs a lot of configuration to be more efficient than PyTorch, if the client wants to change hosting platforms or something on the server changes, the ONNX code may need to be changed as well which is something that is outside of the client's technical expertise. A few years ago, PyTorch added torch.compile(). This does something similar to ONNX conversion, but is much simpler and cross-platform. It is also less likely to be deprecated in the future. Again, because PyTorch is so much simpler, if we later want to try ONNX, it will be easy to rewrite backend sections to try it.

For the frontend, we are choosing to use **HTMX**. While we could use React and create a quality product. HTMX decreases the complexity of development and hosting of our website quite a bit. With our hosting platform yet unknown, having something that can be

fully hosted server-side with no separate client-side app hosting required increases the portability and maintainability of our product, and this is very important for our client.

3.3.5 Feasibility Validation Plan

We have created a simple FastAPI backend with an HTMX hello world. The button gets information from the backend using a HTMX button and a template for the HTML



In order to prove the functionality of our web stack, the client floated using an old model of his and constructed a test website with it. In this way, we can have a functioning website for our tech demo, even if the acoustic model is not yet done. The website will be developed in this order: Basic Frontend, Basic Backend, Audio Generation, Polling and chunking of input text, then PyTorch optimizations last.

3.4 Machine Learning Framework

A core technical challenge in our project is selecting an appropriate deep learning framework for speech synthesis and processing. This component is central to the system, as it must support tasks such as audio feature extraction, model training for speech generation, and real-time inference. These functions require efficient handling of sequential data, GPU acceleration, and flexibility for experimenting with different neural architectures (e.g., RNNs, Transformers, or diffusion-based models). The framework we choose will directly impact development speed, model performance, and long-term maintainability of the system.

3.4.1 Desired Characteristics

An ideal solution must satisfy several key characteristics. First, performance and speed are critical because speech models are computationally intensive and may require near real-time inference. Second, ease of development and debugging is important since rapid iteration will be necessary during model experimentation. Third, community support and ecosystem maturity matter because speech processing often relies on prebuilt libraries and shared research implementations. Fourth, flexibility is essential to allow customization of model architectures. Finally, maintenance and scalability are important to ensure the system can evolve and be deployed reliably over time. These characteristics serve as the evaluation criteria for comparing alternatives.

3.4.2 Alternatives

We considered two primary frameworks: PyTorch and TensorFlow. PyTorch, developed by Meta AI, is a widely used deep learning framework known for its dynamic computation graph and Pythonic interface. It has become the dominant framework in academic research, particularly in speech and natural language processing, due to its flexibility and ease of experimentation. TensorFlow, developed by Google, is another mature and widely adopted framework with strong support for production deployment, especially through TensorFlow Serving and TensorFlow Lite. It has historically been favored in industry environments requiring scalability and optimization.

3.4.3 Analysis

Our analysis evaluated both frameworks against the desired characteristics. In terms of performance, both frameworks offer strong GPU acceleration, though TensorFlow has a slight edge in production optimization tools. However, PyTorch performs comparably in most research and development scenarios. For ease of development, PyTorch clearly excels due to its intuitive syntax and dynamic graph execution, which simplifies debugging and experimentation. TensorFlow, while powerful, has a steeper learning curve and can be more complex to debug. Regarding community support, both frameworks have large user bases, but PyTorch has stronger momentum in cutting-edge speech research, with many state-of-the-art models released in PyTorch first. In terms of flexibility, PyTorch again stands out, allowing easier implementation of custom architectures. For maintenance and scalability, TensorFlow offers more built-in tools, though PyTorch has improved significantly with tools like TorchServe.

Framework	Performance	Ease of	Community	Flexability	Scalability

			Use		
PyTorch	4	5	5	5	4
TensorFlow	5	3	4	4	5

3.4.4 Selected Approach

Overall, while TensorFlow offers strong production capabilities, PyTorch provides superior ease of use, flexibility, and alignment with current speech research. Based on this evaluation, we have chosen PyTorch as the primary framework for our project.

3.4.5 Feasibility Validation Plan

To validate the feasibility of PyTorch, we implemented a small prototype that learns word associations using a neural network model. The model was trained on a dataset of word pairs and evaluated using both training and validation accuracy metrics.

Prototype Results and Observations

```
==========================================
Epoch 0, Loss: 527.8559, Accuracy: 1.04%
Epoch 50, Loss: 3.3524, Accuracy: 100.00%
Epoch 100, Loss: 1.3379, Accuracy: 100.00%
Epoch 150, Loss: 0.8116, Accuracy: 100.00%

Predictions:
cat -> dog
dog -> bone
bird -> sky
fish -> water
sun -> moon
day -> night
hot -> cold
light -> dark
happy -> sad
fast -> slow
big -> small
up -> down
left -> right
open -> close
```

Figure 1

Figure 1 shows the training output over multiple epochs. The results demonstrate that the model successfully learns associations between words, as indicated by increasing training accuracy over time.

The prototype works by converting words into numerical representations (embeddings) and training a neural network to predict associated words. For example, given an input such as “cat,” the model learns to predict “dog.” This demonstrates how PyTorch can be used to build and train models for sequence-based or language-related tasks.

3.5 Python Package Manager

In order to ensure long-term project functionality, we require a package manager that can bundle all project dependencies alongside the version of python.

3.5.1 Desired Characteristics

An ideal solution must have the following characteristics:

- **Bundle Python Version:**
  Dependencies often require specific Python versions. The package manager must reliably install and isolate the correct Python version without affecting other projects on the system.
- **Manage Dependency Versions:**
  The solution should automatically resolve, install, and isolate dependency versions with minimal manual intervention. This includes handling conflicts and ensuring reproducibility across environments.
- **Ease of Use:**
  The package manager should simplify the creation and management of virtual environments while maintaining an accurate record of dependencies and their versions. It should require minimal setup and reduce the likelihood of user error.

3.5.2 Alternatives

a) Conda

Conda is a widely used package and environment management system, particularly popular in data science and machine learning workflows. It provides robust environment isolation and supports installing both Python and non-Python dependencies. Conda has been in use for many years and has a large user base, but can be slower and more complex to configure.

b) uv

uv is a modern Python package manager designed for speed and simplicity. It provides fast dependency resolution, automatic virtual environment management, and tight integration with Python version management. uv has gained popularity as a lightweight alternative to traditional tools such as pip and virtualenv.

3.5.3 Analysis

a) Conda

Conda provides strong support for environment isolation and dependency management, including the ability to install specific Python versions. It is particularly useful for complex machine learning environments that require compiled dependencies. However, Conda environments can be slow to create and manage, and resolving dependency conflicts can require manual intervention. Additionally, its complexity may introduce unnecessary overhead for a relatively focused project.

b) uv

uv offers a streamlined and efficient approach to package management. It automatically manages virtual environments and resolves dependencies quickly with minimal user input. Its lightweight design makes it easier to use and reduces setup time. While uv is newer and has a smaller ecosystem compared to Conda, it is sufficient for managing Python-based dependencies required by this project.

Criterion	**Conda**	**uv**
Bundle Python Version	5	4
Manage Dependency Versions	4	5
Ease of Use	3	5
Performance (Speed)	2	5
Ecosystem Maturity	5	3
Complexity / Overhead	2	5

3.5.4 Chosen Approach

Our team chose **uv** as the package management solution. This decision was based on its ease of use, fast dependency resolution, and minimal configuration requirements. These characteristics allow the team to focus on development rather than environment setup.

While Conda provides more extensive support for complex environments, its additional complexity and slower performance were not justified given the scope of this project. uv provides sufficient functionality to manage dependencies and Python versions while maintaining a simpler and more efficient workflow.

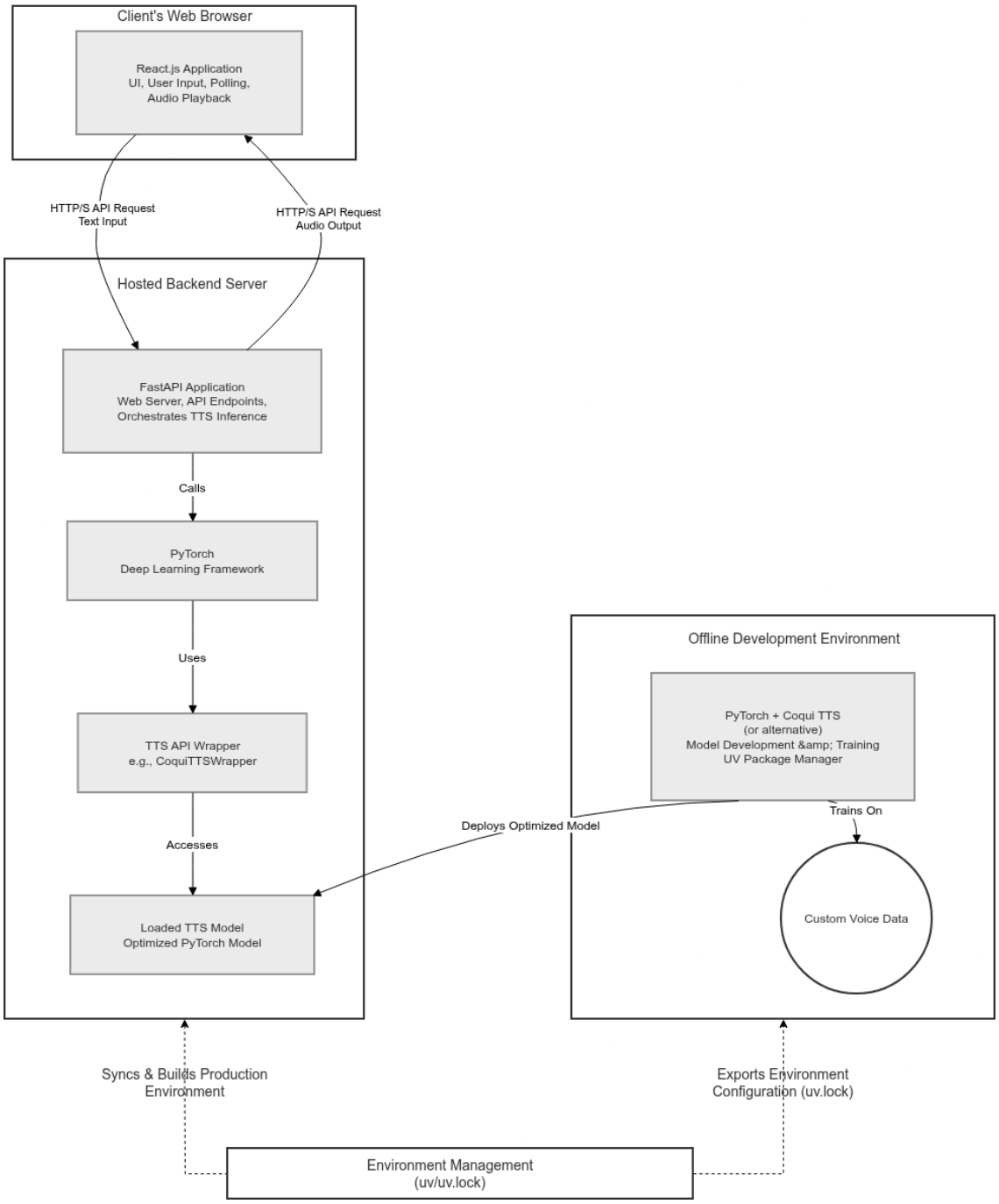
3.5.5 Feasibility Validation Plan

To validate the feasibility of uv, the team created a development environment that includes all required dependencies for the TTS pipeline. Then the team tested the uv virtual environment on different machines to verify correct containerization.

4 Technological Integration

To deliver a powerful, maintainable, and flexible Text-to-Speech solution, we will integrate our chosen technologies into a coherent server-side architecture. This section outlines how the various components—from the web stack to the deep learning inference engine—will interoperate to satisfy all product requirements.

4.1 SYSTEM DIAGRAM OVERVIEW



Our envisioned system employs a robust client-server architecture. The user interacts with a dynamic frontend, which communicates with a dedicated backend server responsible for the TTS inference.

1. **Offline Model Development:** The process begins in an Offline Development Environment. Here, developers utilize PyTorch as the deep learning framework, integrating a TTS API (initially Coqui TTS, housed within a modular TTS API Wrapper). Custom voice datasets are used to train and fine-tune acoustic models. Once a model is satisfactorily trained and optimized (e.g., using torch.compile() for performance), it is packaged and deployed.

2. **Backend Deployment & Service:** The FastAPI application, along with the optimized PyTorch TTS model and its associated API wrapper, is deployed to a Hosted Backend Server. This server will be a designated environment (e.g., a university server or a Platform-as-a-Service like Render) capable of running persistent services. The FastAPI Application serves as the web server, exposing API endpoints (e.g., /synthesize) that the frontend will call. It internally orchestrates the loading and execution of the PyTorch deep learning framework, which in turn utilizes the Loaded TTS Model via the modular TTS API wrapper.

3. **Frontend Interaction & Data Request:** The user interacts with the application through their Client's Web Browser, which loads the React.js Application. This application provides the user interface for text input and audio playback control. When the user enters text for synthesis, the React app sends an asynchronous HTTP/S API Request (containing the text) to the FastAPI backend server. For efficient updates, especially if segmenting audio, the React app can employ Polling to check the status of the synthesis or retrieve generated audio chunks.

4. **Server-Side TTS Inference:** Upon receiving a request, the FastAPI. Application on the backend server passes the input text to the PyTorch framework. PyTorch, using the Loaded TTS Model (via the TTS API Wrapper), performs the computationally intensive TTS inference. This process leverages the server's CPU or GPU (if available) to generate the audio waveform from the text.

5. **Audio Delivery and User Experience:** Once the audio is generated, the FastAPI application sends the resulting audio data back to React.js. Application in the user's browser as an HTTP/S Response. The React app then uses the browser's native audio capabilities for immediate playback. This architecture ensures that the demanding computational work is handled by a dedicated server, providing a consistent and high-quality experience for all users, regardless of their local hardware, while maintaining the flexibility to swap TTS engines due to the modular design.

6. **Environment & Dependency Management**:
    - **The "Single Source of Truth" (Environment Parity)**: To ensure environment parity, we utilize uv to generate a cryptographic uv.lock file during the offline Development Environment stage. This file acts as our single source of truth, guaranteeing that the exact versions of PyTorch and the TTS API Wrapper used during training are the ones deployed to the Hosted Backend Server.
    - **Streamlining the Deployment Pipeline**: The integration of uv significantly streamlines our deployment process. Because uv is written in Rust and utilizes a global dependency cache, it allows for near-instantaneous environment builds on our Hosted Backend Server. This reduces the overhead of our CI/CD pipeline and ensures that scaling or redeploying the FastAPI application is fast and reliable.
    - **Isolation of Complex Dependencies**: Given the complexity and size of deep learning frameworks like PyTorch, we use uv to manage isolated virtual environments. This integration prevents dependency conflicts between our web server (FastAPI) and our inference engine (PyTorch), ensuring that the high-performance requirements of real-time TTS generation are met without environment-level instability.

5 Conclusion

Speech and language are extremely important parts of cultures around the world. For under-resourced languages such as Diné Bizaad, continued use and learning are essential for the language's health. TTS can serve not only as a means of teaching these languages but also as an aid for those who are unable to speak to be able to take part in their culture. However, for many of these languages, no such TTS models exist. To address this issue, our team will create a speech synthesis pipeline that will enable our client to more effectively produce TTS models even where little transcribed audio exists.

Additionally, we will create a simple web interface where our client can upload his acoustic models and generate audio. This document outlined various technical challenges involved with the creation of this pipeline and weighed various technologies to aid its development. We covered what neural network libraries are available, possible TTS frameworks, what web technologies will be required, and where we will get the compute for running the web interface. Through this evaluation, the team identified trade-offs between ease of use, flexibility, and long-term maintainability. A primary framework was selected to support rapid prototyping, while a modular system design was proposed to allow future integration of more research-oriented tools.

Overall, this analysis demonstrates that the proposed system is technically feasible within the project constraints and that appropriate technologies exist to support its development. Moving forward, the team will begin implementing a prototype of the training pipeline, focusing first on validating data preprocessing and transfer learning capabilities using a small dataset. This will happen alongside the development of the web interface and further refinement of the system architecture to ensure flexibility and extensibility.