Bilkent University

Department of Computer Engineering

Senior Design Project

T2533

StreamLang

Design Project Final Report

22203632 - Uygar Bilgin - uygarblgn13@outlook.com
22202995 - Mehmet Emin Avşar - emin.avsar@ug.bilkent.edu.tr
22202913 - Göktuğ Ozan Demirtaş - goktugozandemirtas@gmail.com
22203805 - Ruşen Ali Yılmaz - rusenaliyilmaz@gmail.com
22203239 - Can Tücer - can.tucer@ug.bilkent.edu.tr

Uğur Doğrusöz
Mert Bıçakçı
İlker Burak Kurt

30.04.2026

This report is submitted to the Department of Computer Engineering of Bilkent University in partial fulfillment of the requirements of the Senior Design Project course CS491/2.

1. Introduction

This document showcases the final design of our project. We begin by sharing our finalized set of requirements, which were updated constantly as the project grew. The latest system architecture is then described in detail. In the next section, we provide further details on our implementation and design. In both of those sections, the details for each subsystem are given separately, to ensure that enough information is provided for the whole system of StreamLang to be understood clearly. Then, we present the outcomes of the test cases that were given in the previously written Detailed Design Report. Later, are given our latest and finalized iterations of engineering factor considerations, ethical responsibilities, work distribution, and what we've learned so far. The document is finalized as we discuss our outcomes and provide recommendations for any future work that may be done on StreamLang.

We believe that the provided explanations with such extended detail will be enough for those concerned to fully understand how our system works, what requirements, factors, and responsibilities we've followed, and how we functioned together as a team.

2. Requirements Details

This section includes our refined and finalized functional and non-functional requirements, which were previously listed in the Detailed Design Report and Analysis and Requirements Report, together with how we satisfied them.

2.1. Functional Requirements

StreamLang's main focus is creating structured and customized listening streams for language learners. We believe the advanced customization and personal curations we provide will help users attain fluency faster compared to other language learning methods. The users will be able to learn the language they want to learn at the level they desire, through the listening streams with a theme they are interested in, right through their mobile phones.

StreamLang has a strong grammar-centric architecture where not only the vocabulary, but also the grammar structures that the user is exposed to are tracked and then reintroduced at optimal times for memory retention according to SRS (Spaced Repetition System) principles, a method that has been shown to be effective in retaining information [1]. At any point during their training, users are able to reconfigure the grammar and vocabulary levels they would like to be exposed to.

The additional Anki integration allows users to pair their Anki decks with a stream configuration. This way, users are able to create configurations using their already prepared learning cards, speeding up the process.

Listening stream configuration templates catering to users with different skill sets and levels in the language will be available, again speeding up the configuration process.

The specific scenarios users should be able to perform in StreamLang, in other words, our finalized functional requirements, are listed as follows:

- Selecting the language.
    - Only German is selectable at the moment, but the addition of new languages will be trivial due to the reasons explained in the following sections.
- Selecting the level of vocabulary.
- Selecting the type of content they are going to listen to.
    - They may choose to listen to an article, a story, or a sentence.
- Selecting the theme of the stream.
    - They may select a theme by prompting keywords or categories.
- Selecting whether they want to hear the translations.
    - They can select to hear the translations after each sentence, or after the whole stream ends.
- Selecting the grammar structures they want to study.
    - They may select past tense, present tense, perfect tense, etc., depending on the language they've selected.
- Selecting which voice they want to hear.
    - They may select from options like male and female voices, depending on the language they've selected.
- Selecting the speed of the voice.
    - This was the only functional requirement we failed to fulfill. See Section 7.3.4. for details.
- Selecting which Anki account to integrate (optional).
    - Choosing vocabulary from an Anki deck overrides the "level of vocabulary" selection.
    - Select whether new cards should be created in their Anki deck for words that they frequently come across during the listening stream.

2.2. Nonfunctional Requirements

2.2.1. Usability

StreamLang, as a learning tool, is supposed to be accessible and usable by users from all age groups and technology knowledge levels. Without the necessity of a manual or external assistance, users should be able to configure their streams and start listening.

To satisfy this requirement, we prioritized simplicity and accessibility in the design of the user interface. We kept the configuration options simple to understand and manage while making sure users, when requested, are able to fine-tune the configurations just like they want. We are also storing access tokens in client devices to remember the account logins.

2.2.2. Performance

As this is a mobile app, we were required to provide high interactivity with smooth visuals and no delay. This means the mobile app should not show any stutters, should not crash, and waiting times should be minimal.

For this purpose, we delegated all of the performance-heavy operations (i.e., keyword generation for vocabulary sets, text-to-speech, etc.) to the servers running on the cloud, providing fast response times and better error handling. Then, we had to ensure the servers were responsive and that the slowdown in a submodule did not affect the whole system. In this regard, the heavy burden was that some of the user requests may require content to be generated using large language models.

We decided that any request that does not require such generation should be responded to within one second (excluding client-side network delays). Similarly, any content-generating request should be responded to within one second after the generation is done. To guarantee this, we built upon a fairly scalable server architecture, allowing us to distribute computing loads to different endpoints. We also made sure to use the optimal hardware while trying to use sufficiently sized smaller models to shorten generation times.

2.2.3. Reliability

For the same reason as performance, we considered reliability as an important requirement. The decisions we took to improve performance also contributed to the reliability by moving error-prone functionalities out of user devices, improving error detection, and increasing service uptime. As a result, assuming continuous third-party service availability, we expect the servers to have at least 95% uptime.

2.2.4. Security

StreamLang stores no personally sensitive data. Thus, security and privacy were not a main concern. Yet, we made sure to follow the best practices of all forms to ensure user data is kept safe and private. We used industry-standard JWT tokens for request/response verifications. We hashed passwords using Bcrypt. We set up the firewall configurations of our databases and servers carefully, while making use of Cross-Origin Resource Sharing to protect our subsystems against adversaries. Lastly, we used HTTPS for client - backend communication.

2.2.5. Scalability

We believe that StreamLang provides a solution to a problem the majority have, and we are preparing to launch the app publicly on mobile app stores. Thus, we prioritized scalability from the beginning.

The mobile client is a frontend-only Flutter application, so it requires no scaling. However, our subsystems, especially the generative AI systems, may require scaling as the user base grows. For this reason, as previously discussed, the backend is separated into multiple modules to facilitate horizontal scaling. Any module that is directly facing the user and is running strictly locally on our own servers has the ability to run in multiple instances and can be scaled infinitely. That is, we already took the precautions against race conditions. Modules using third-party services can be scaled horizontally when required. With this architecture, we believe that we are able to adequately scale our application as user numbers change.

2.2.6. Maintainability

As StreamLang is an online service, it requires high uptime. This made maintainability that allows us to apply updates and fixes without disrupting the service an important requirement. Here, once again, the modular architecture of our subsystem allows for easy maintenance. As modules in need of maintenance can function independently of each other, while any fixes/rollbacks are carried out on problematic modules, others can keep running uninterrupted. In the case of horizontally scalable modules, when a module instance is down due to maintenance or a bug, another instance can take its place to serve users until the instance is back up. Lastly, in the case of frontend maintenance, there will be no downtime since the entire application is already built and runnable on the client device. Any updates will be delivered through the device's application store. More details about our maintenance plan are provided in Section 6 of this report.

2.2.7. Flexibility

Right now, StreamLang only works with German. However, we are aware that for StreamLang to achieve its main goal, we must provide a streaming service for multiple languages concurrently. Thus, we were required to achieve flexibility, meaning that adding new languages should not require updates to core functionality.

For this reason, we made sure that the vocabulary and grammar themes are stored dynamically as enumerated variables, and our scraping, generative AI, and streaming modules support the languages we are interested in providing in the near future. More information about the future work that may be done on StreamLang is provided in Section 8 of this report.

3. Final Architecture and Design Details

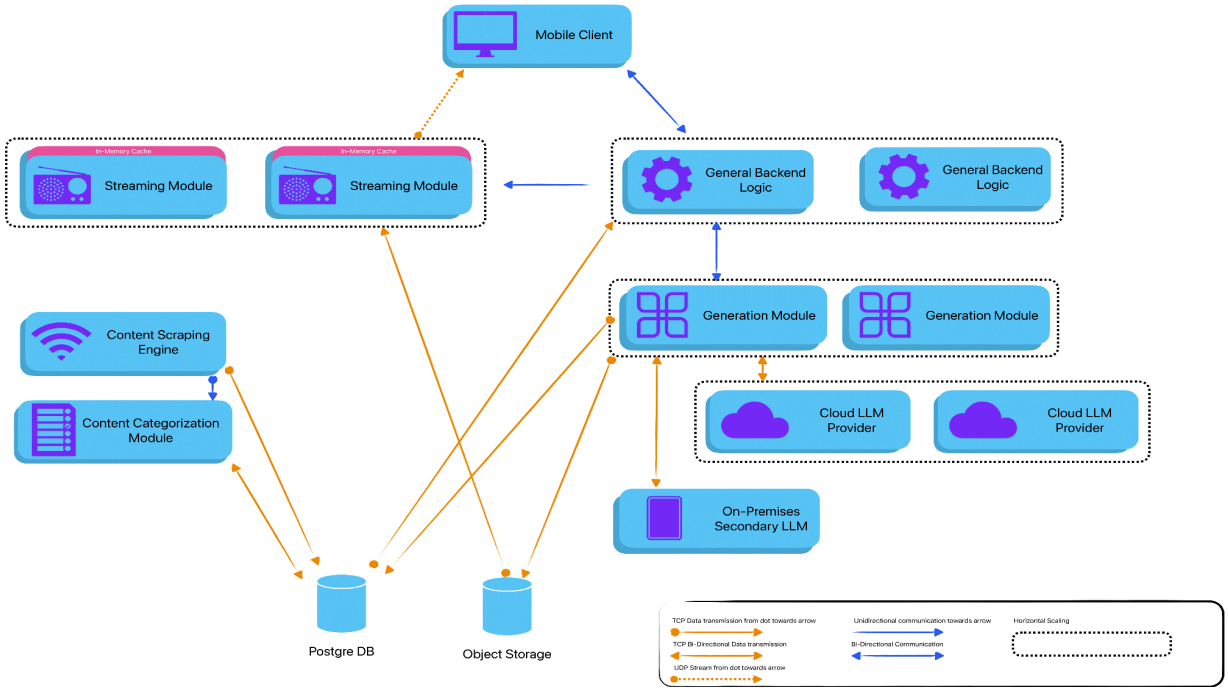

Figure 1. The Architecture of StreamLang

StreamLang demarcates its functionality into the following submodules: Streaming Module, Core Backend Module, GenAI Module, Content Categorization Module, and the Frontend. The helper components that interact with the submodule network are our on-premises LLM, PostgreSQL DB, and object storage in Cloudflare R2.

The high-level design details of each of the modules are given under this section. The following Section 4. Development / Implementation Details provides details about the development process.

3.1. Core Backend Module

Our Core Backend Module handles the main functionality use case of our application, which is serving audio content to users. This is facilitated by its content selection and stream channel creation mechanism, which is triggered by a call to the */stream/start* endpoint in the backend.

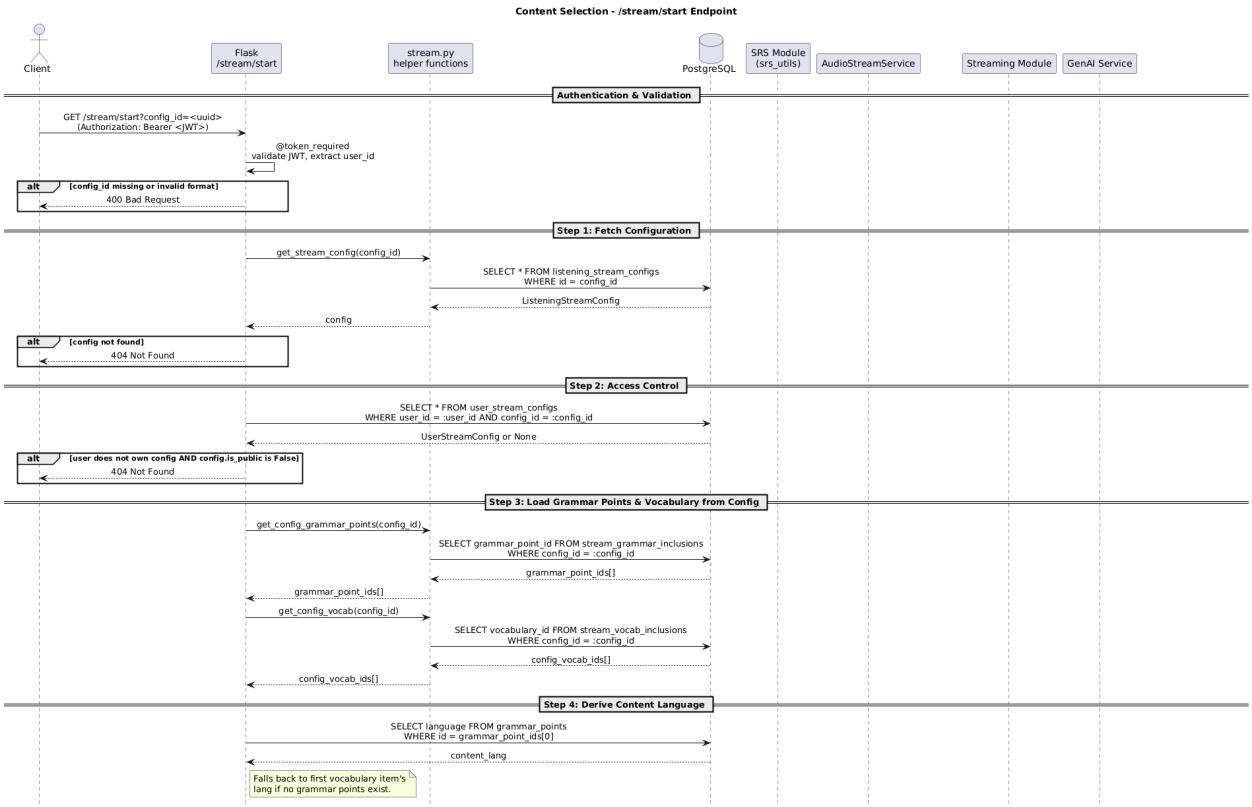

Figure 2. First part of the Content Selection Sequence Diagram

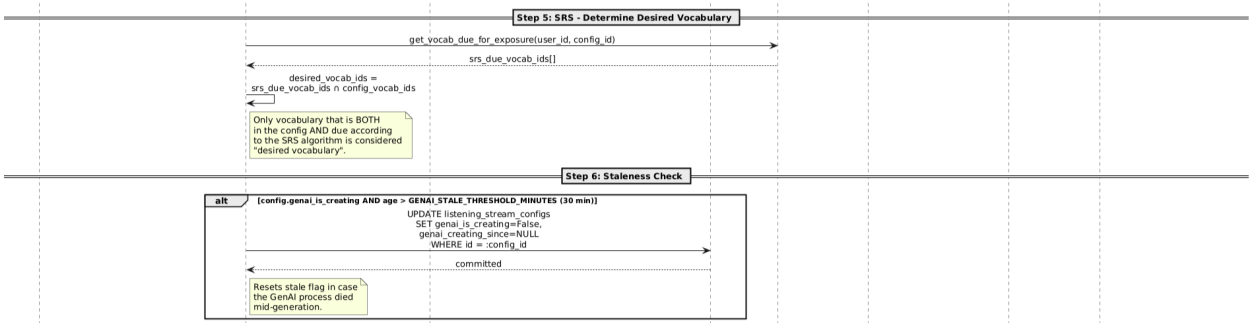

Figure 3. Second part of the Content Selection Sequence Diagram

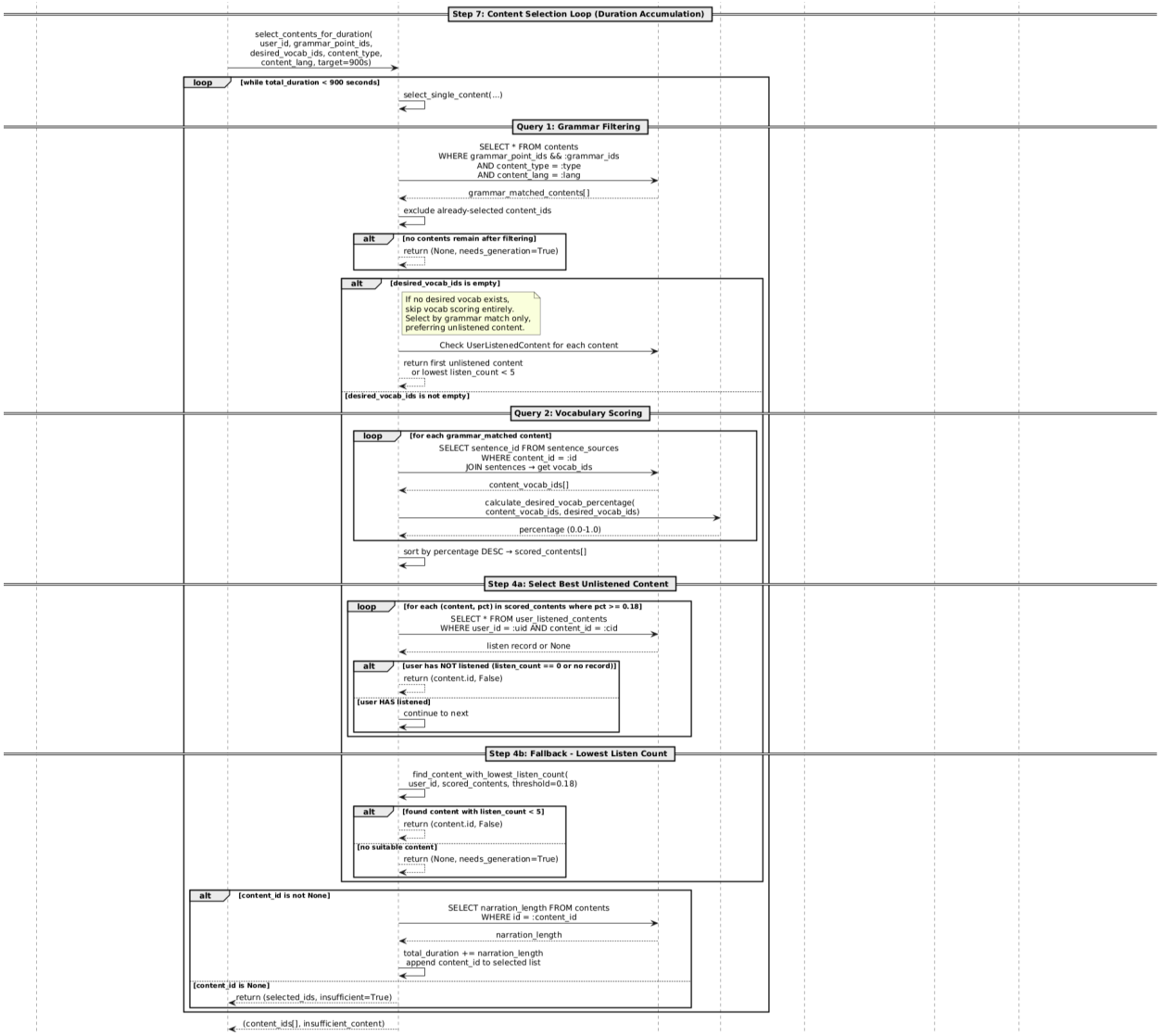

Figure 4. Third part of the Content Selection Sequence Diagram

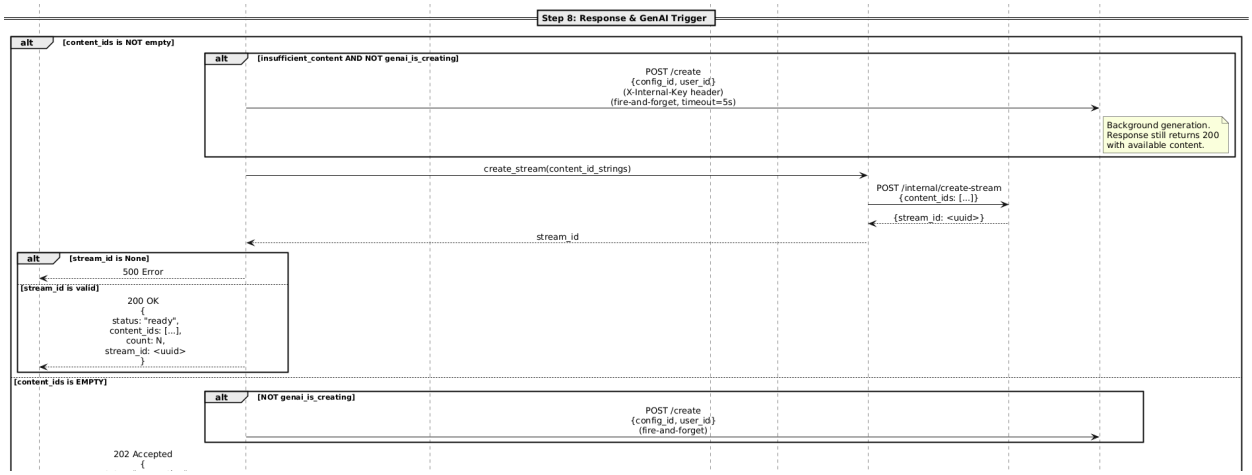

Figure 5. Fourth part of the Content Selection Sequence Diagram

Initially, the content selection mechanism interacts with our SRS implementation to fetch vocabulary due for exposure according to the user's schedule. Then, some suitable content is fetched from the database according to the user configuration. If no such content could be found, the GenAI module is triggered to generate content according to the user's needs. Below, there is also an activity diagram for the *select_single_content* function, which handles the content selection functionality for the */stream/start* endpoint.

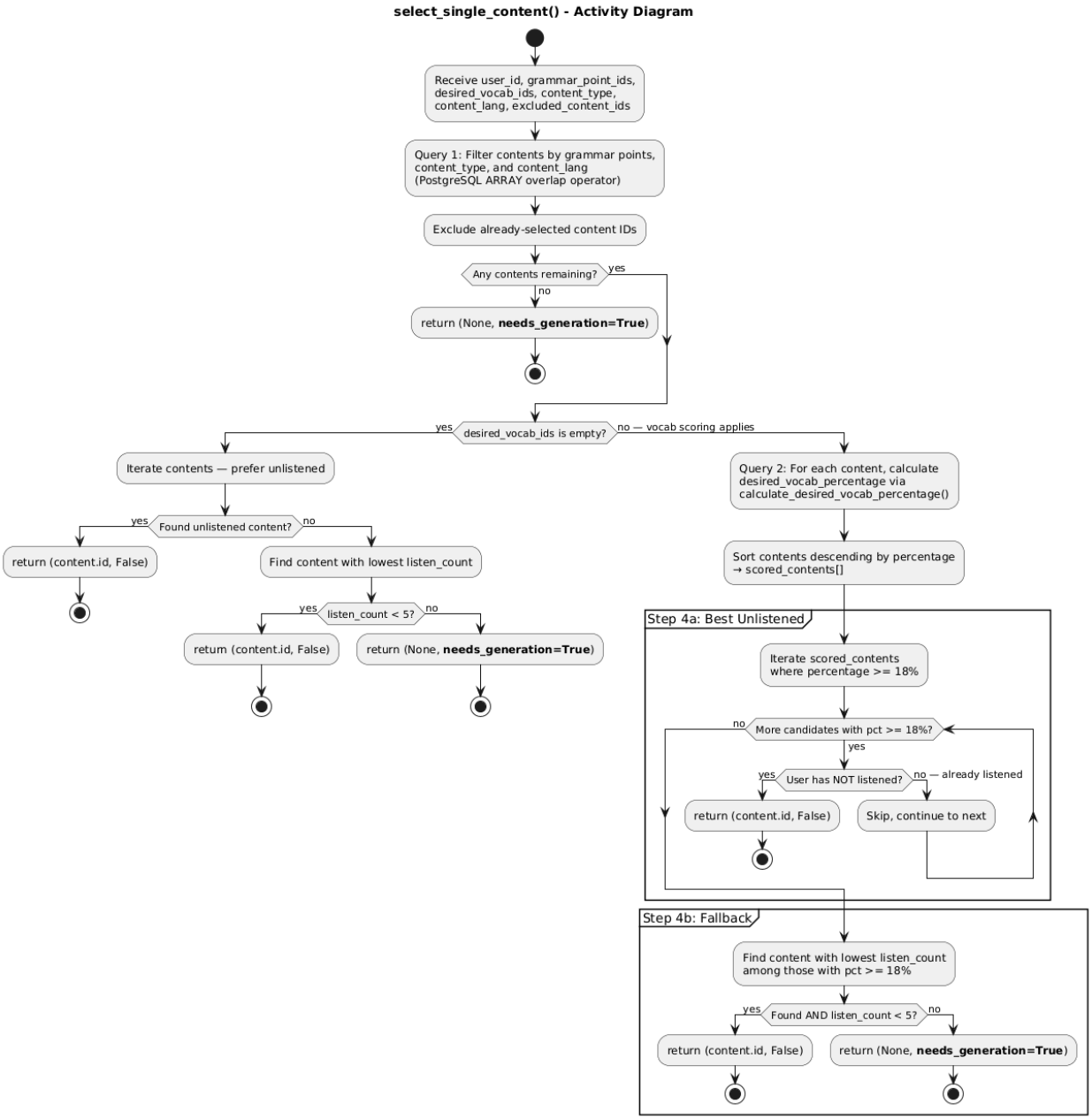

Figure 6. Activity Diagram of the Content Selection Mechanism

In addition to the Content Selection Mechanism, Core Backend also handles authentication and user state management. Those operations are facilitated by several endpoints defined as */login*, */logout*, and */register*, etc. These endpoints use JWT to store login tokens for users in the database.

Lastly, the Core Backend manages the state of listening configurations. Configurations created by users are saved, loaded, and edited via the endpoint routes running through Core Backend. In other words, Core Backend is the main communication endpoint of the frontend, and handles/updates the states of all dynamic objects we have in the system.

3.2. Streaming Module

The Streaming Module is solely responsible for taking a series of audio files and serving them to a user. This also requires the module to track a given playlist of audio files and serve them opaquely to the user from a single endpoint. Since the files are stored using an S3 service, we also need to hide the audio file details from the user. The module is implemented using a two-server setup with two main endpoints:

- **HTTP Post Endpoint:** <*web server*>*/internal/create-stream*: An internal endpoint only usable by other modules to tell the streaming module to prepare for streaming some list of contents to a user using a generated stream ID.
- **RTSP Mount Point:** <*rtsp server*>*/live/*<*stream_id*>: The endpoint a user needs to stream content to their device using the RTSP protocol. The stream creation is done using endpoints on the core backend.

The endpoint */internal/create-stream* is run on a Flask server. After being called, it first checks if the data format is valid, which should be a list of audio entries. If not, the response is a 400 Bad Request with an error message. After validating that the required data exists, it then generates the file name for every audio entry, and checks in batches to see if there is a corresponding audio file for them using the Audio Location Service. If no audio exists for any given file name, the call fails and returns a 404 Not Found response. After finally iterating through all names, if there was no error, it adds a stream to the Stream Store Service with the file names for that stream, and responds with 200 OK along with the stream ID in the body.

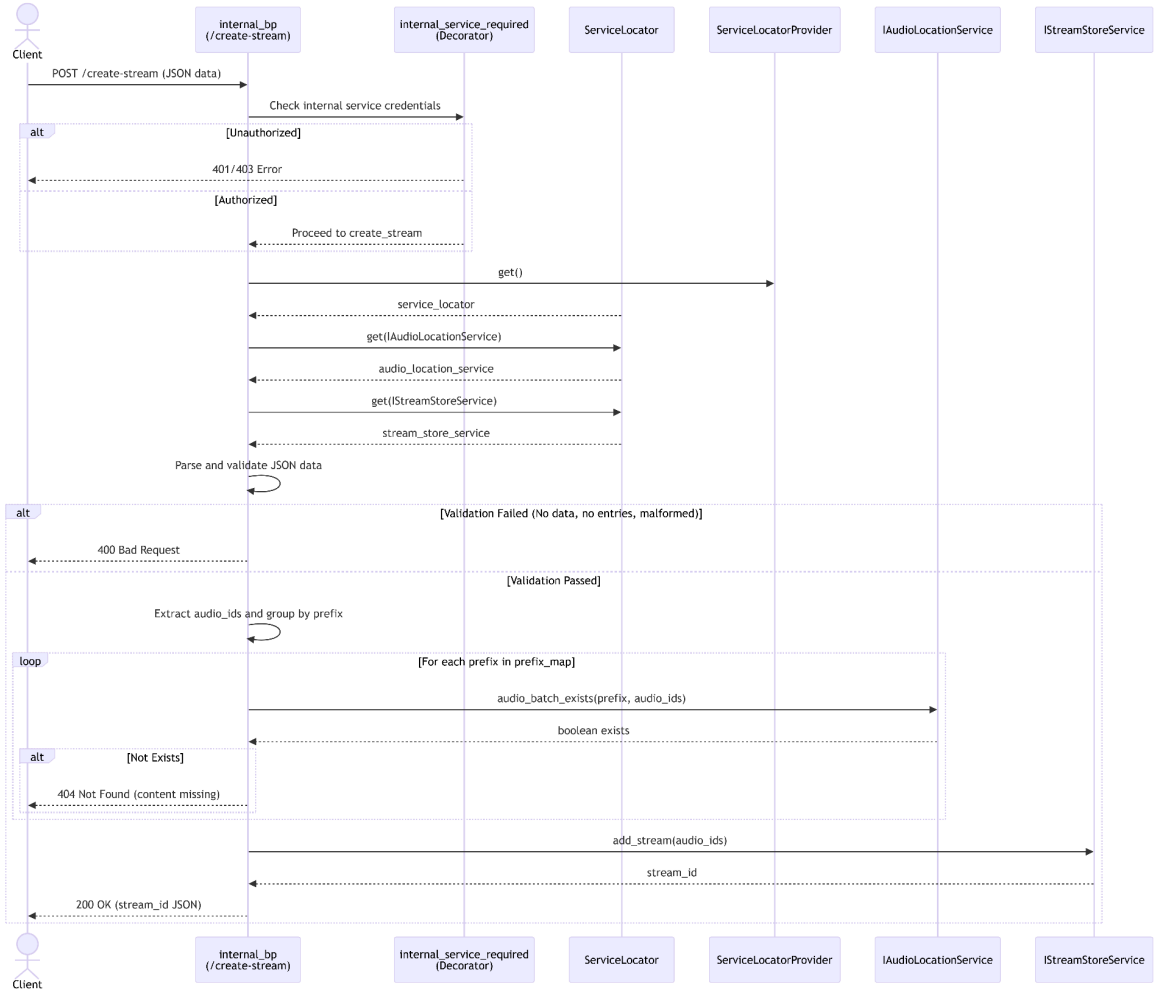

Figure 7. Sequence Diagram for /internal/create-stream

The second server is an RTSP server using the GStreamer library, responsible for the */live/<stream_id>* endpoint. GStreamer is an established open source multimedia framework [2] that handles the actual audio streaming work, like loading the files, decoding them and sending the data to the user. GStreamer is then wrapped with Python code to read stream data from the Stream Store and stream their associated audio files. The Python code used for GStreamer is made up of two classes: DynamicMountPoints and StreamFileFactory:

The DynamicMountPoints is used to check if an RTSP request URL and the provided stream ID are valid and forward, and proceed to creating the pipeline using a file factory. If not, it falls back to GStreamer's default handling, which in this case returns a 404 response to the client.

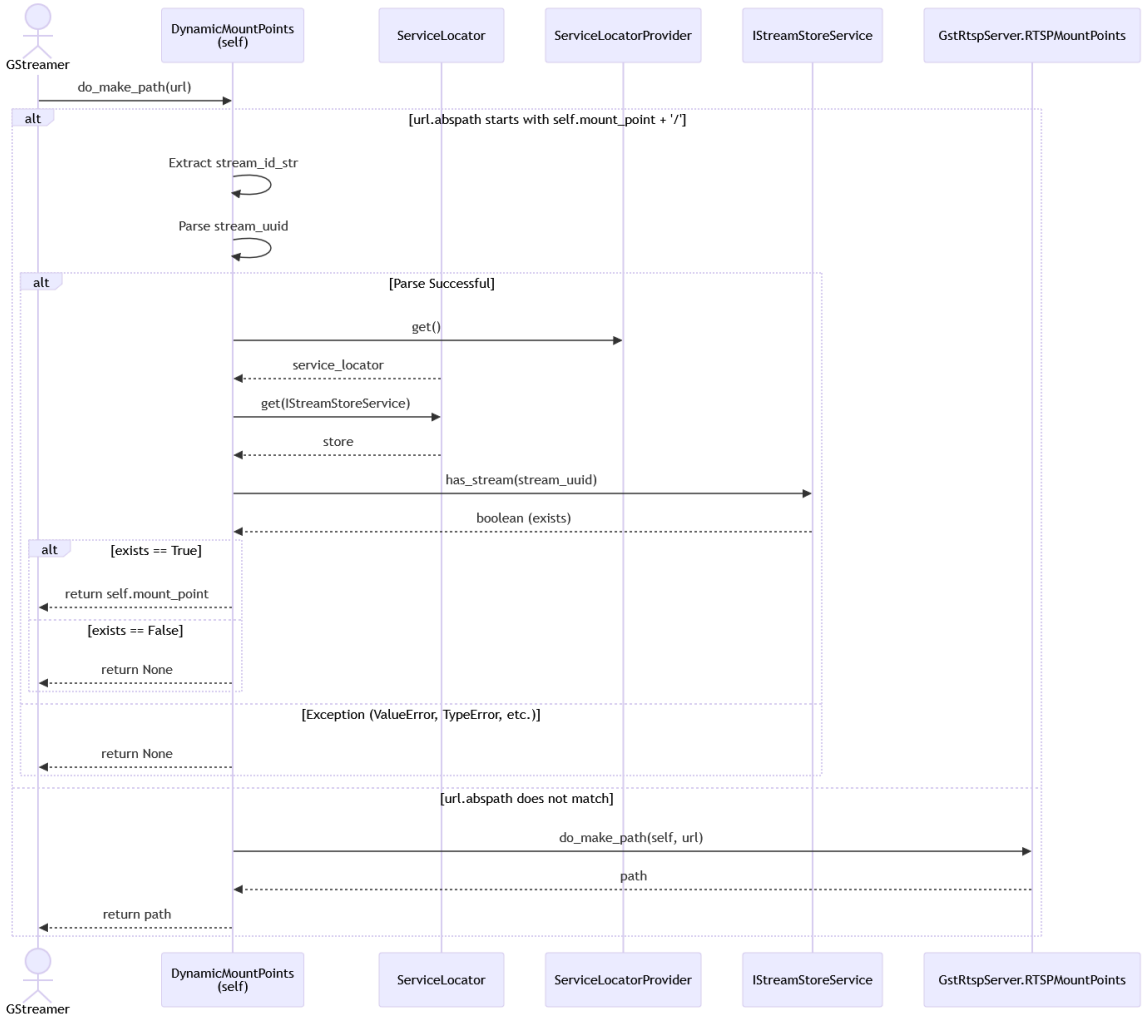

Figure 8. Sequence Diagram for DynamicMountPoints

StreamFileFactory is responsible for constructing the launch parameters for the specific stream URL for GStreamer to create a streaming pipeline. It first loads the stream contents from the Stream Store Service and checks if the audio files for them exist using the Audio Location Service. If they do, it constructs the launch string and creates a GStreamer pipeline with it. If not, it does nothing.

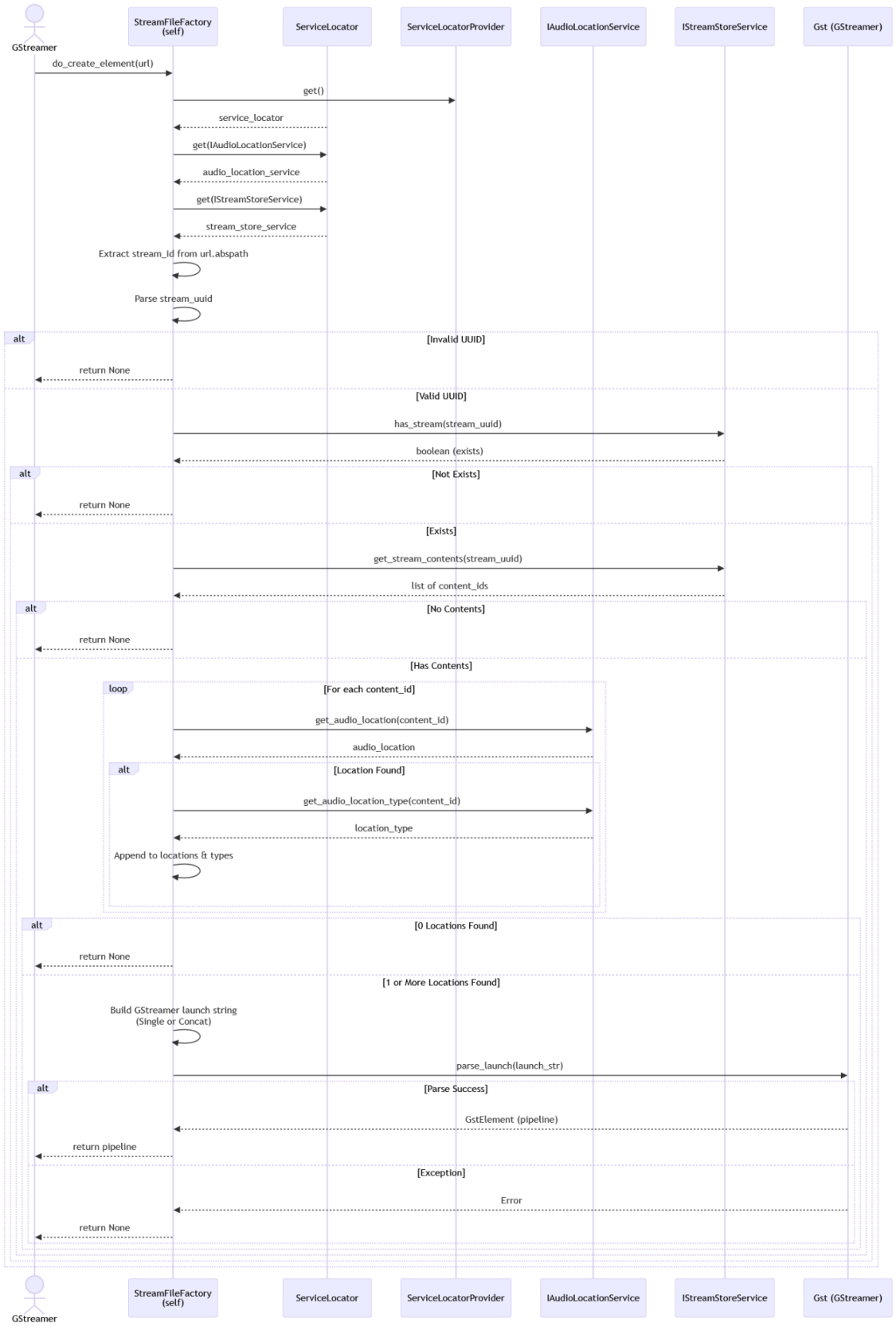

Figure 9. Sequence Diagram for StreamFileFactory

The implementations of the other services are either trivial or contain unnecessary detail (e.g. our internal format for naming audio files) and do not need to be fully explained like the previous subroutines. Other details will be given in Section 4.2.

3.3. GenAI Module

The GenAI module is the collection of features that require a call to an LLM or the narrator model Chatterbox. The principles that we followed in the design of this module is threefold:

1. **Modularity:** A simple modular structure allows for easier understanding of the module, and thereby eases debugging and thereby reduces development time. It also allows for easier upgrades and changes to individual submodules due to loose coupling. For example, changing the prompts to increase accuracy requires a change in the prompt builder, whose interface with all the other modules is precisely defined; thereby, there are no or easily containable spillover effects when you want to make a change to a module. Following the design principle of modularity, the GenAI module makes use of the following loosely coupled services: the content processor, embedding service, generation service, LLM client, narration service, prompt builder, and S3 service.
2. **Simplicity Without Compromising Accuracy:** During the development of the prompts, for example, we started with the most direct instructions to the LLM, and incrementally increased the complexity of the prompts as concerns for accuracy necessitated it. We have also carefully documented the GenAI module in a human-readable way that will make understanding the data flow as simple as possible.
3. **Efficiency:** Since GenAI is costly, we have optimized our choice of models wherever we can. For example, when new content is generated, its embedding should be pushed to the database. Instead of using a giant multilingual model to carry out this embedding, we coordinate smaller bilingual models. This results in decreased compute use and also faster embeddings.

The modular structure and the high-level interactions between the services can be summarized with the following diagram.

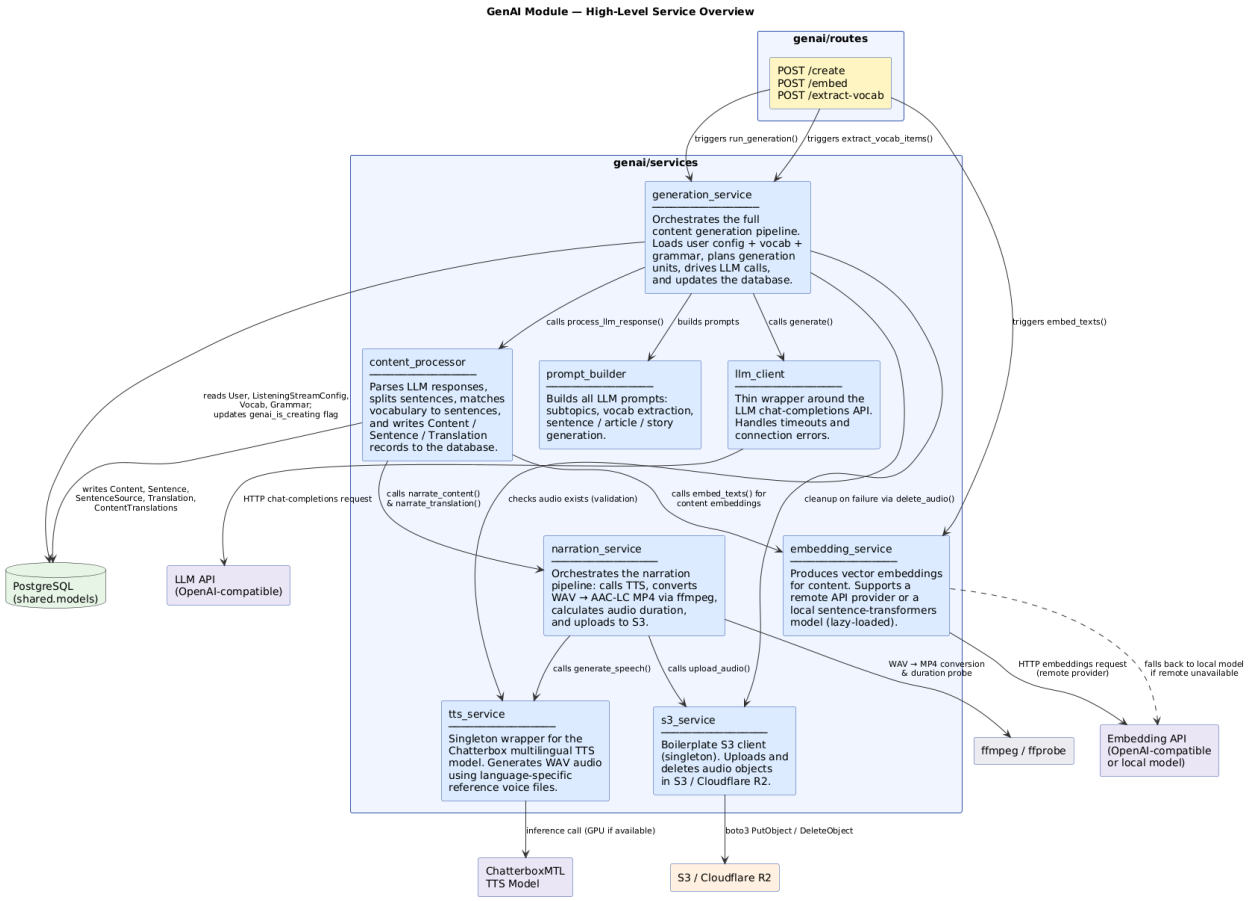

Figure 10. The services of the GenAI module and their interactions

The GenAI module is designed to create content that, even when the general topic name, the vocabulary, and the grammar structures remain the same, there will be enough variety to keep the listener engaged for hours. In addition, whenever there are excessive grammar points and vocabulary selections that might strain the LLM if it tries to pack it all into a single story or article, there must be a solution that will preserve cohesion. This is achieved with multi-step LLM orchestration (and is one of the key reasons why StreamLang is better than the user just making ChatGPT generate all that content). First, subtopics are generated, and then the grammar points and vocabulary in the listening stream configuration are shuffled such that fresh pairings of subtopics and grammar point/vocabulary matchings remain engaging. Every time the listening stream configuration runs out of content, the GenAI module creates different subtopics and repeats this process.

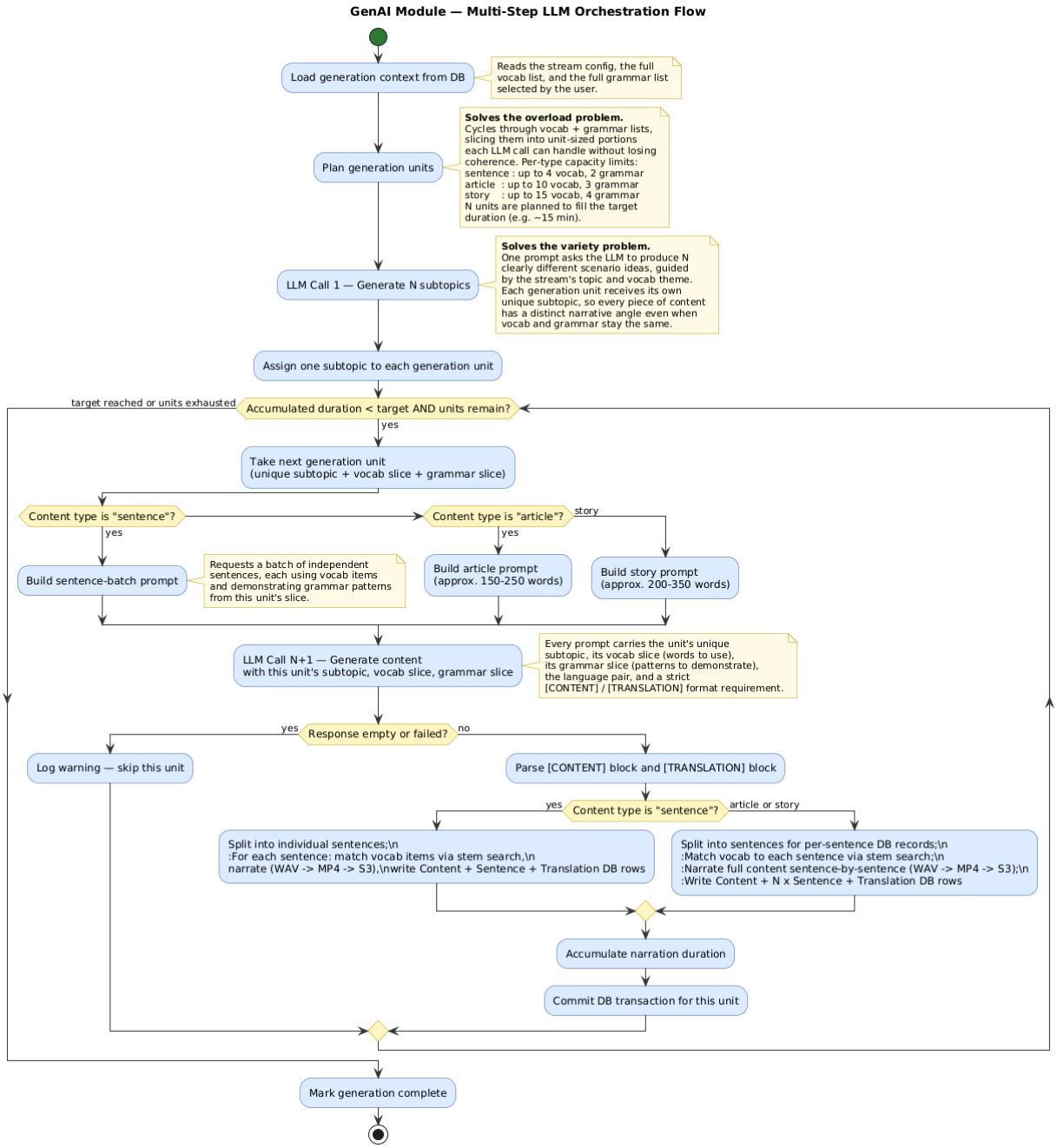

Figure 11. Multi-step LLM Orchestration Flow of the GenAI Module

This was the high-level overview of the GenAI module, emphasizing design principles and what they achieve. The development and implementation details are given in Section 4.

3.4. Content Categorization Module

The main logic and the use case of the content categorization module is done through a batch job that is run periodically. This batch job does the following:

1. Separate content into sentences.
2. Populate sentences and sentence_sources tables.
3. Classify grammar points in sentences.
4. Add grammatical point relations to sentences.
5. Add grammar point relations to the content, for which the amount of grammar points in that content passes a 5% threshold.
6. Tokenize sentences by vocabulary.
7. Check if the vocabulary in a sentence matches the ones in the vocabulary table; if yes, add those vocabulary to the sentence.

This batch job is implemented as an endpoint, which, when called, carries out all these functions sequentially, while also being robust so that no duplication of database entries will occur, and the entire process will happen atomically so that the database won't be in an incomplete state. During this process, we require interacting with separately implemented helpers that carry out the natural language processing (NLP) tasks required. These helpers use the SpaCy library to achieve this. Their internals will be discussed in Section 4.4. Below is the sequence diagram for the Content Categorization Module.

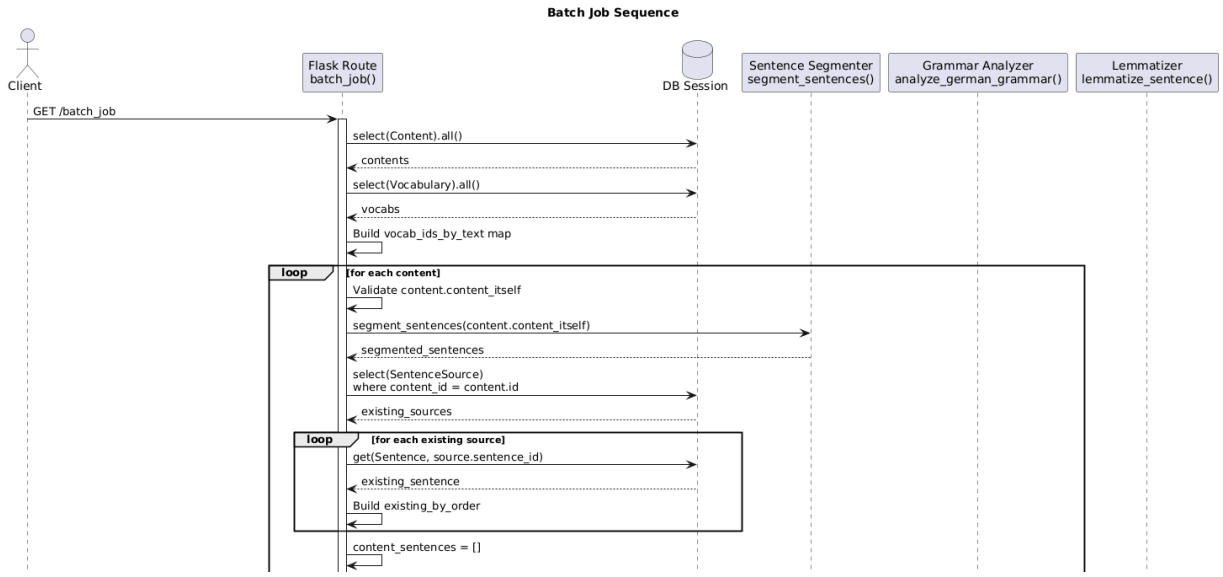

Figure 12. First part of the Content Categorization Sequence Diagram.

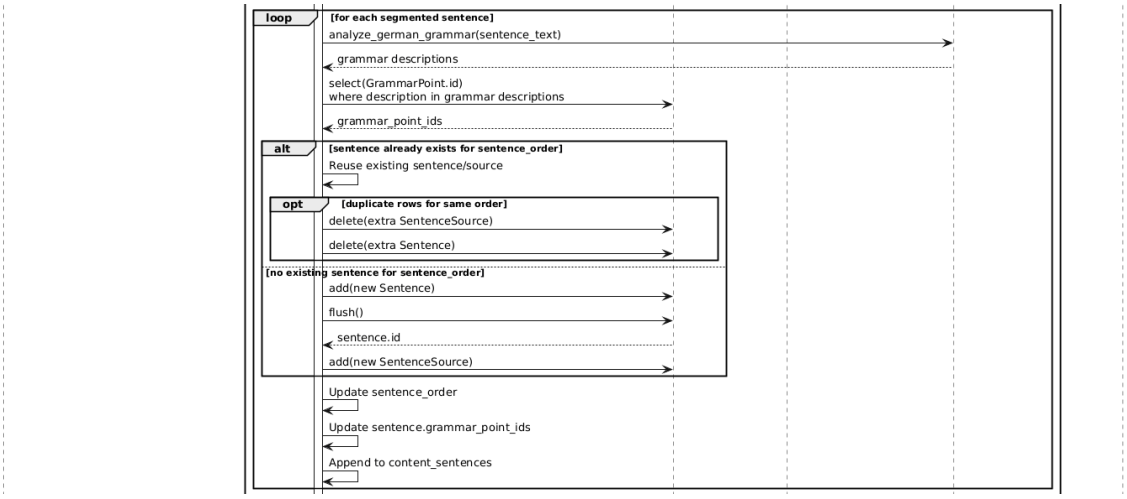

Figure 13. Second part of the Content Categorization Sequence Diagram.

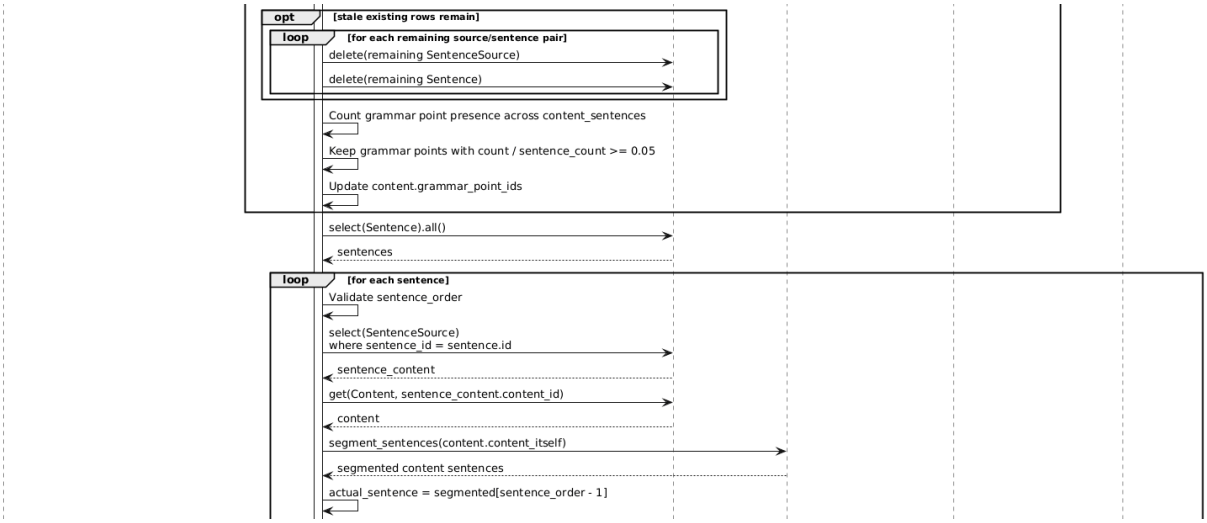

Figure 14. Third part of the Content Categorization Sequence Diagram.

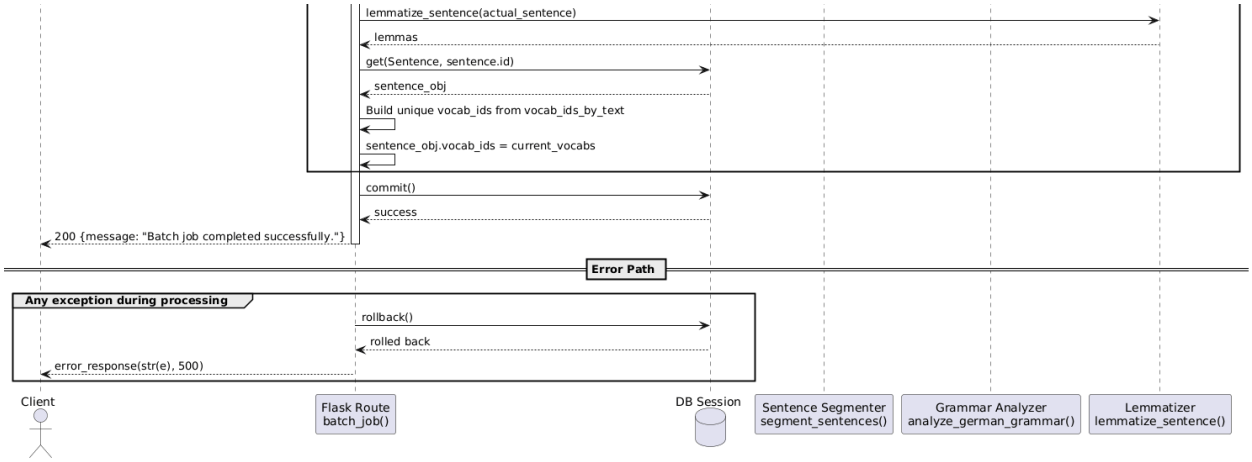

Figure 15. Fourth part of the Content Categorization Sequence Diagram.

In this design, we used a modular approach to keep the NLP helpers and the batch job endpoint separate so that we could reuse those NLP helpers if needed again. The batch job had to be implemented as a single large endpoint because we needed to ensure the atomicity of the transaction, so that we could not separate it into several different endpoints, as that would leave the database in an incomplete state.

3.5. Frontend

The mobile frontend of StreamLang is structured into four distinct layers that separate the UI from the business logic and API communication. Even though our structure is a common structure, it still guarantees a solid separation of concerns, keeping the code in every layer clean, making it easy to understand and easy to maintain.

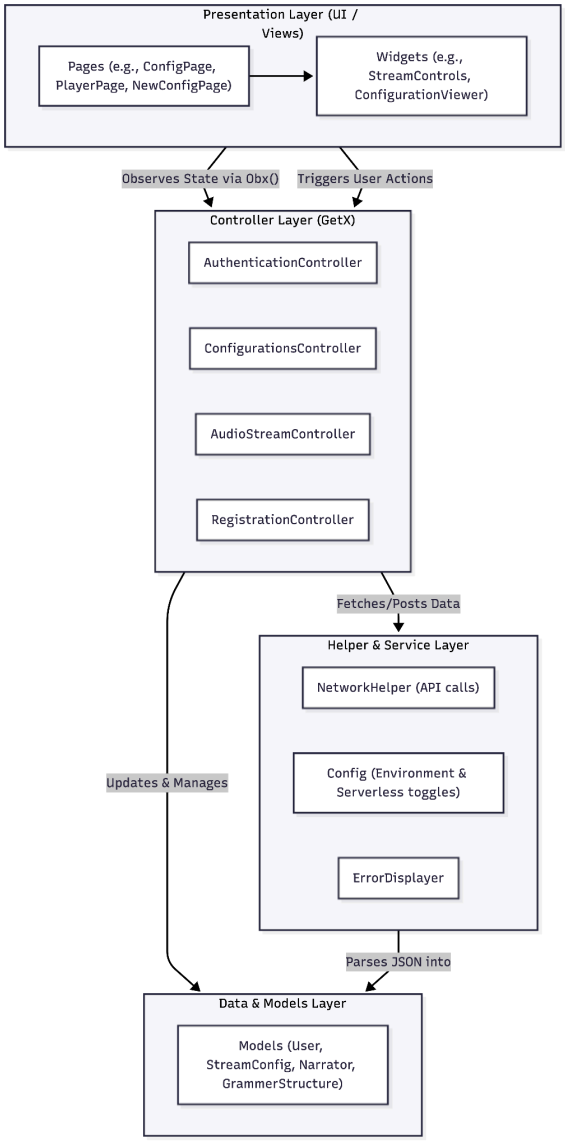

Figure 16. Diagram showing the relation among layers.

- **Presentation Layer (UI)**

Strictly responsible for rendering the interface and capturing user input. It uses an observer pattern to react to changes rather than self-managing states.

Components in the presentation layer can be split into two: the pages and the widgets. Pages, such as PlayerPage and ConfigPage, are the main composers of the application. Widgets, like StreamControl, are the small but repeatedly used functional components.

- **Controller Layer**

For our controller layer, we utilized GetX state management. It is the core business logic of our backend. For most controller classes, we use the singleton pattern. This is to ensure consistent global state access across the entire app. When it comes to data management, we use the built-in features of GetX, which manage observable states of data. We also have an adapter interface to handle media streaming controls, as the pure library we use requires initialization and utility methods to write clean and maintainable code, and to also create a layer of abstraction.

- **Helper/Service Layer**

This is a basic, but critical layer that acts as a communication bridge between the controllers and the backend server API. There are 2 significant components in this layer: “NetworkHelper” and “Config”. The prior is responsible for the centralization of non-RTSP HTTP communication and managing standard HTTP status code responses. The latter is simply an environment variable manager, but it allows for network bypass and enables debugging logs.

- **Data/Model Layer**

This layer is mostly core, plain Dart classes for our data. Classes we use have helpers that handle JSON encoding and decoding. We also have a few custom runtime exceptions defined in this layer to have proper error handling in our app.

4. Development/Implementation Details

This section of the report is going to draw close parallels with the Detailed Design Report due to the minimal number of changes accrued since the writing of that report, a benefit of careful construction of our software design.

4.1. Core Backend Module

The detailed architecture of the Core Backend Module was given in Section 3.1. This section describes the implementation decisions taken during the development of this module, and our reasoning behind each decision.

We started our code development with the core backend module. When the implementation has started, the REST API documentation was already written, and our object diagram was already prepared. Thus, the implementation phase was pretty short and straightforward. Step by step, the following tasks were done:

1. We started by setting up our PostgreSQL DB. We chose PostgreSQL as we definitely needed a relational DB for our user and configuration management requirements, and we could make use of vector storage for sentence retrieval via embeddings. We set up Postgres on Azure, as they were offering sufficient storage and compute limits with their Student plans. We set up the firewall and network rules of DB so that it will be only reachable by the backend client and our development environments, to ensure maximum user data security.
2. We applied the discussed and planned DB schema and ran dummy data insertion tests to avoid any problems later on.
3. Then we started the development of the backend app. We used Flask for easier and faster development. Python also made the integration to LLM generation and ML categorization services much easier, as both had to use Python to be able to run over best-in-class frameworks for AI-related tasks.
4. We developed the authentication routes (*/register*, */login*, */logout*), together with the enhancements like password hashing and matching mechanism, duplicate email prevention, etc. Those were required to increase account security. Carried out the unit tests for those.
5. We developed the JWT token generation and storage mechanism. Carried out the unit tests for those. Tokens were used to allow persistent sessions. This was required especially since StreamLang is a mobile app.
6. Then we developed the configuration management endpoints (*/getConfigurations*, */addConfiguration*, */deleteConfiguration*, */getNarrators*, */getGrammerStructures*). Created the required enumeration variables to be used as configuration options. Carried out the unit tests for those.
7. Finally, we've implemented the */stream/start* endpoint, which is used to connect the frontend to the streaming service. The tests for this endpoint were done together with the

tests of the streaming module, as both modules had to function together for the tests to be viable.

After all of those steps were done, we didn't have to make any changes or improvements to the core backend module, ignoring a few small bug fixes.

4.2. Streaming Module

This section is responsible for explaining the motivations behind the decisions outlined in Section 3.2. and giving any additional details that are worth mentioning.

The Streaming Module is solely responsible for taking a series of audio files and serving them to a user. This also requires for the module to track a given playlist of audio files and serve them opaquely to the user from a single endpoint. Since the files are stored using an S3 service, we also need to hide the audio file details from the user.

One important design choice in the streaming module is the modular code structure. Following a core SOLID principle, SRP, the streaming module's functionality has been separated into services. This means that different facilities, like the stream store facility or a facility for fetching audio locations (URLs) for a given content ID, are completely separate from each other. This allows for easy replacement of business logic for some specific functionality.

Another decision made was to not reinvent the wheel. Although most problems in software are never fully solved, content streaming is one that has been tackled numerous times by others. While it may be technically possible to make an optimized implementation strictly serving our purposes, such optimizations may take a long time while being equally unneeded for the current scope of our application. That is why the streaming module uses the GStreamer [2] library to do the actual streaming. It supports dynamic mount points calculated through code, seamless streaming of multiple files, and more. Thanks to this, a big part of the module was implemented painlessly, and if customized code is ever needed, the code is structured in such a way that an in-house solution can be developed with relatively small friction.

Consequently, the streaming module actually runs two servers, one web server for API calls when needed (currently internal only) and one streaming server. Although this may sound complicated, the module is relatively easy to use both for consumers of the API (the core backend in our case) and users of the application.

Since the Flask and GStreamer servers run as separate processes, they share no memory. To solve the problem of sharing data between these two processes, the final Stream Store Service implementation creates a local SQLite3 database in the module and stores all the data there, which can be accessed at all times by anyone.

Currently, the two servers are assumed to be accessed from different addresses. However, the module is set up in such a way that nginx [3] is running as a reverse proxy. If it is

eventually needed to consolidate the addresses of the modules into a single address, the module should be internally ready for any changes in the configuration of the entire network.

S3 storage is queried using the boto3 library [4]. File existence checks are done using the list_objects_v2 function in the API using our structured names as prefixes (one for translated audio and one for the original audio).

With everything mentioned, the application has a reasonably efficient and easily extendable streaming module that is simple for consumers. It does not do anything more than it needs to do, hides all details from the user that are not needed, and gives a good streaming experience backed by a well-developed and industry-standard streaming library.

4.3. GenAI Module

The most important working principles and the service decomposition of the GenAI module have been given in Section 3.3. This section is about the development journey and the methods used during development, alongside specific implementation choices that may shed light on our principles. Those who want to see a very specific trace of the GenAI module's control flow can access and download a sequence diagram from this link.

A three-tiered verification strategy has been followed during the development of the GenAI module.

- **Tier 1 - Manual "Try-Out"**

When trying out prompts, models for narration, LLMs, or embedding models, we first try them out in isolation by finding an online provider and giving them incrementally more complex tasks. For example, when we were first considering Qwen3-32B as an LLM, we used the HuggingFace space for that model to feed it our prompts and evaluate the results intuitively. This verification tier serves as a vibe check before we commit valuable time to integrating new methods into our system. A documented example of a vibe check can be found in this link.

- **Tier 2 - Smoke Test**

In this tier, we connect the new technology or method to the GenAI module, run generation, and then see if it meets our quality requirements (cohesion, accuracy, and simplicity).

- **Tier 3 - Extended Usage Test**

After confirming that the new technology works once, we run it against varied configurations several times to see if there is a region of the input or time domain where the results start to degrade. If there is any, we assess the level of degradation against the time cost of switching to a new technology and the risk that we will face similar issues after the switch. We act strategically with regard to time without letting perfectionism paralyze us, while also holding up to our principles.

The debugging strategy that we use is stepping through the code with the VS Code debugger to see where the problem occurs. We have a purpose-written launch.json file that allows us to conveniently launch every module of the backend simultaneously in the debugger. This allows us to step through one module to another to diagnose wide-spanning issues.

There are several interesting implementation details with the GenAI module. As a start, let's take a look at how we design our prompts. The following is the prompt we use to generate a batch of independent sentences for a given listening stream configuration:

```
f"LANGUAGE REQUIREMENTS (STRICTLY ENFORCED):\n"
    f"- The CONTENT language is: {target_lang}\n"
    f"- The TRANSLATION language is: {native_lang}\n"
    f"- {target_lang} and {native_lang} are TWO DIFFERENT languages. The content and translation MUST NOT be in the same
language.\n"
    f"- Everything under [CONTENT] MUST be written entirely in {target_lang}.\n"
    f"- Everything under [TRANSLATION] MUST be written entirely in {native_lang}.\n"
    "\n"
    f"Generate exactly {batch_size} independent sentences in {target_lang} (NOT in {native_lang}).\n"
    f"{topic_line}\n"
    f"{desc_line}\n"
    f"{vdesc_line}\n"
    "\n"
    f"Each sentence MUST use at least one of these vocabulary words (use the base/dictionary form where possible):\n"
    f"  {_format_vocab_list(vocab_words)}\n"
    "\n"
    f"Each sentence MUST demonstrate at least one of these grammar patterns:\n"
    f"{_format_grammar_list(grammar_descriptions)}\n"
    "\n"
    "Rules:\n"
    "- Write natural, meaningful sentences a real person might say or read.\n"
    "- Each sentence must be self-contained.\n"
    "- Vary sentence structure between sentences.\n"
    "- Try to use as many of the given vocabulary words and grammar patterns as you naturally can.\n"
    "\n"
    f"After the {target_lang} sentences, translate each sentence into {native_lang}, in the same order.\n"
    "\n"
    "Format your response EXACTLY like this (no extra text):\n"
    "[CONTENT]\n"
    "Sentence 1.\n"
    "Sentence 2.\n"
    "...\n"
    "[TRANSLATION]\n"
    "Translation 1.\n"
    "Translation 2.\n"
    "..."
```

Several choices pop up. First, we make use of examples ("format your response EXACTLY like this") to stabilize the LLM's response. Second, we repeat our requirements in separate places to reinforce their effect - "write natural, meaningful sentences [...]" and "Try to use as many of the given vocabulary words and grammar patterns as you naturally can". We also include "guardrails" to warn against the LLM's idiosyncratic mistakes: "TWO different languages" because we used to get the translation and the content itself in the same language

for some models we tried. Another method used in prompt construction is making explicit some of our assumptions about the generated content that the LLM might overlook: “each sentence must be self-contained” / “vary sentence structure between sentences”. These instructions generate content that flows more naturally.

Another implementation detail in the GenAI module is the wrapper design pattern that we followed. For example, in order to upload audio to S3, we do not call S3’s specific functions. We have a wrapper file uploader service, and we call the functions of that service. In this way, if we decide to change the object storage solution we use in the future, there will be minimal migration burden created. The same pattern is used for LLM calls, embedding calls, and narration calls as well.

The GenAI module, as with all other backend modules, is written in Python.

4.4. Content Categorization Module

Content Categorization Module is mainly made up of two parts: a categorization backend that uses SpaCy to implement natural language processing (NLP) tasks, and a batch job endpoint that interacts with the database accordingly. The main architectural design of the Content Categorization Module is explained in Section 3.4, which goes into detail about how the batch job is implemented. This section will mainly focus on the backend NLP part of the module.

For our NLP backend, we have three helpers:

- **Grammar Point Classification (german_grammar_point_classification.py):** This helper file implements the analyze_german_grammar function, which takes a sentence string as the argument and returns a list of German grammar points. To achieve this, it uses some useful features of the SpaCy library. Mainly, it uses part-of-speech tagging and morphological features to analyze words in a sentence to determine their grammatical features. These can be their word type (noun, verb, etc.), dependencies with other words in the sentence, and their morphological features such as verb forms, mood, and tense. All these are used to determine if a sentence contains one of the fourteen grammar points we defined.
- **Sentence Lemmatizer (german_lemmatize_sentence.py):** This small helper file implements the lemmatize_sentence function, which takes a sentence string as an argument and returns the list of lemmatized words in that sentence. The lemma form of a word is its simple form, without any extra morphological features. Some examples of a word and its lemmatized form could be looking vs. look, or is vs. be. This function is used to determine if words in a sentence match words in the vocabulary table of the database, as we need to compare their lemmatized forms to do the comparison accurately.
- **Sentence Segmentation (german_sentence_segmentation.py):** This helper file implements the *segment_sentences* function, which takes a text (content) as the argument and returns a list of sentence strings. Its main goal is to intelligently split a text into its individual sentences to be inserted into the database.

4.5. Frontend

The frontend, specifically the mobile application made using a combination of the Flutter framework alongside some platform-native code. Our frontend follows the classic tried-and-tested registration and login flows with the following possible actions:

- **Registration:** Registering a new account,
- **Login:** Logging into an existing account,
- **Password Reset:** Sending a password reset request for a user.

Henceforth, here are the actual details worth describing.

Tech Stack: As previously mentioned, we are using the Flutter framework. However, to accomplish our desired functionality, we are utilizing the following packages:

- **HTTP:** The official library from the Flutter team to have HTTP communications
- **Get:** Get (or widely known as GetX) is a comprehensive library most known (and used here) for its state management and localization abstractions.
- **Media Kit + Media Kit Video Libraries:** These libraries are used for RTSP playback. Although we only stream audio, the media kit audio libraries lack the necessary HTTP tunneling feature we require for NGINX compatibility with the streaming service. We use the video library, but disable requests for video when connecting.

Configuration Interface:

The configuration interface has a lot of hours of work put in. StreamLang had to use a single-page configuration interface as:

- If the user decided to change a previous setting, they would have to go back one by one, limiting usability.
- Due to the advanced level of customizability of StreamLang, there are a lot of options. Separating them into different pages would mean the user would have had to do the consecutive “next” presses, which would lead to an increase in frustration.
- This design choice came with a crucial requirement: no clutter. To reduce clutter, StreamLang uses a simple high-contrast theme and keeps its widgets standardized within the app.

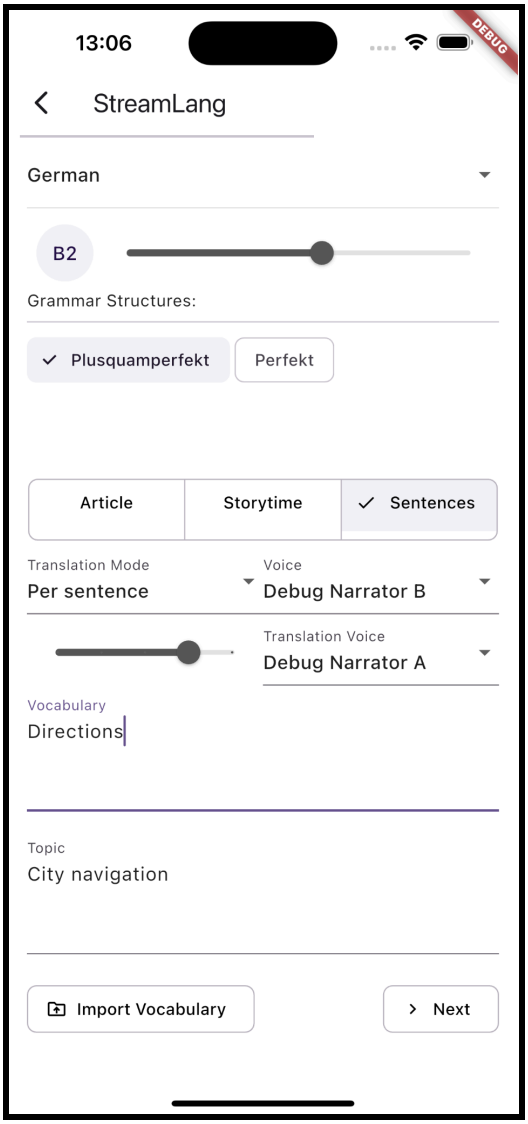

Figure 17. Screenshot of configuration interface.

5. Test Cases and Results

Test Case 1: Account Registration - Success

- **Test Type/Category:** Functional
- **Summary/Title/Objective:** Verify successful creation of a new user account and completion of the onboarding flow.
- **Procedure of testing steps:**
    - **1.** Open the mobile application.
    - **2.** Tap "Create Account".
    - **3.** Enter a valid, unique name, email, and payment details.
    - **4.** Verify that a one-time magic link is sent to the email address.
    - **5.** Click the magic link to confirm the email.
    - **6.** Select a native language and a target language on the Onboarding page.
    - **7.** Confirm the user is directed to the Templates page.
- **Expected results/Outcome:** The user account is created, they are logged in, and successfully complete the language selection, landing on the Templates page.
- **Priority/Severity:** Critical
- **Actual Results:** Account registration was successful.
- **Discussion**: Everything went as expected.

Test Case 2: Account Registration - Invalid Email Format

- **Test Type/Category:** Functional
- **Summary/Title/Objective:** Verify the system rejects account creation with an improperly formatted email address.
- **Procedure of testing steps:**
    - **1.** Open the mobile application.
    - **2.** Tap "Create Account."
    - **3.** Enter a valid name, payment details, but use an invalid email format (e.g., "test@.com," "test@com").
    - **4.** Attempt to proceed with registration.
- **Expected results/Outcome:** The system displays an error message indicating an invalid email format and prevents registration.
- **Priority/Severity:** Major
- **Actual Results:** The invalid email format was rejected.
- **Discussion:** Everything went as expected.

Test Case 3: Account Registration - Existing Email Conflict

- **Test Type/Category:** Functional
- **Summary/Title/Objective:** Verify the system prevents account creation using an email address already registered.
- **Procedure of testing steps:**
    - **1.** Open the mobile application.
    - **2.** Tap "Create Account."
    - **3.** Enter a valid name, payment details, and an email address known to be already registered.
    - **4.** Attempt to proceed with registration.
- **Expected results/Outcome:** The system displays an error message indicating the email is already in use and prevents registration.
- **Priority/Severity:** Major
- **Actual Results:** Account registration was prevented.
- **Discussion:** Everything went as expected.

Test Case 4: Account Registration - Weak Password

- **Test Type/Category:** Security
- **Summary/Title/Objective:** Verify the system enforces a minimum password strength during registration.
- **Procedure of testing steps:**
    - **1.** Open the mobile application.
    - **2.** Tap "Create Account."
    - **3.** Enter valid details, but use a password known to be weak (e.g., "123456" or "password").
    - **4.** Attempt to proceed with registration.
- **Expected results/Outcome:** The system displays an error message requiring a stronger password.
- **Priority/Severity:** Major
- **Actual Results:** Weak password was rejected.
- **Discussion:** Everything went as expected.

Test Case 5: User Login - Correct Credentials

- **Test Type/Category:** Functional
- **Summary/Title/Objective:** Verify a registered user can successfully log into their account.
- **Procedure of testing steps:**
    - **1.** Open the mobile application.
    - **2.** Enter a valid registered email and the correct password.
    - **3.** Tap "Login."
    - **4.** Verify the user is redirected to the main application interface (Templates page).
- **Expected results/Outcome:** Login is successful, and the user is granted access to the application.
- **Priority/Severity:** Critical
- **Actual Results:** User was able to log in.
- **Discussion:** Everything went as expected.

Test Case 6: User Login - Incorrect Password

- **Test Type/Category:** Functional/Security
- **Summary/Title/Objective:** Verify login fails when using the correct email but an incorrect password.
- **Procedure of testing steps:**
    - **1.** Open the mobile application.
    - **2.** Enter a valid registered email and an incorrect password.
    - **3.** Tap "Login."
- **Expected results/Outcome:** The system displays an authentication error (e.g., "Invalid credentials") and access is denied.
- **Priority/Severity:** Major
- **Actual Results:** Login attempt was unsuccessful.
- **Discussion:** Everything went as expected.

Test Case 7: Password Reset - Request Flow

- **Test Type/Category:** Functional
- **Summary/Title/Objective:** Verify a user can successfully request a password reset for a registered account.
- **Procedure of testing steps:**
    - **1.** On the login page, initiate the Password Reset flow.
    - **2.** Enter a registered email address.
    - **3.** Submit the request.
    - **4.** Verify the application UI shows a confirmation message (e.g., "Check your email").
- **Expected results/Outcome:** The password reset request is acknowledged, and the system initiates the process of sending a reset link/code.
- **Priority/Severity:** Major
- **Actual Results:** This was not implemented.
- **Discussion:** Focusing on the core functionality pushed peripheral functions like this to the background. Still, it is an important piece and should exist before StreamLang goes in front of a customer.

Test Case 8: Stream Configuration - Vocabulary Selection

- **Test Type/Category:** Functional
- **Summary/Title/Objective:** Verify the user can select the desired vocabulary for their stream configuration.
- **Procedure of testing steps:**
    - **1.** Navigate to the stream configuration interface.
    - **2.** Locate the vocabulary selection option.
    - **3.** Insert the vocabulary you want in the listening stream.
    - **4.** Save the configuration and start streaming content.
- **Expected results/Outcome:** The generated/selected content is tailored to include the vocabulary included in the configuration.
- **Priority/Severity:** Critical
- **Actual Results:** The generated audio stream contains the vocabulary selected in the configuration.
- **Discussion:** The core functionality of StreamLang was satisfied.

Test Case 9: Stream Configuration - Grammar Structure Selection

- **Test Type/Category:** Functional
- **Summary/Title/Objective:** Verify the user can select specific grammar structures for their stream configuration.
- **Procedure of testing steps:**
    - **1.** Navigate to the stream configuration interface.
    - **2.** Locate the grammar structures selection option.
    - **3.** Select multiple specific grammar structures (e.g., "Past tense," "Accusative case").
    - **4.** Save the configuration and start streaming content.
- **Expected results/Outcome:** The generated content frequently includes sentences demonstrating the selected grammar structures.
- **Priority/Severity:** Critical
- **Actual Results:** The generated audio stream contains the grammar structures selected.
- **Discussion:** The GenAI module's unit planning functionality allows it to naturally place grammar points to where they belong.

Test Case 10: Stream Configuration - Drill Mode Activation

- **Test Type/Category:** Functional
- **Summary/Title/Objective:** Verify "Drill" mode functions to cycle sentences using selected grammar structures.
- **Procedure of testing steps:**
    - **1.** Select several grammar structures in the configuration.
    - **2.** Select "Drill" mode.
    - **3.** Start the audio stream.
- **Expected results/Outcome:** The audio stream consists of sentences where every sentence cycles through the selected grammar structures repeatedly.
- **Priority/Severity:** Major
- **Actual Results:** This functionality was canceled.
- **Discussion:** Using "Drill mode" and trying to have the same grammar structure in every single sentence of a story or article led to drastic degradations in quality and cohesion. Consistent with our priorities, we canceled this feature.

Test Case 11: Stream Configuration - Content Type: Article

- **Test Type/Category:** Functional
- **Summary/Title/Objective:** Verify content generation/selection when "Article" content type is chosen.
- **Procedure of testing steps:**
    - **1.** Select "Article" as the content type.
    - **2.** Start the audio stream.
- **Expected results/Outcome:** The content is streamed as a series of coherent articles.
- **Priority/Severity:** Major
- **Actual Results:** The content streamed is actually a series of articles.
- **Discussion:** StreamLang's core functionality was satisfied.

Test Case 12: Stream Configuration - Content Type: Story

- **Test Type/Category:** Functional
- **Summary/Title/Objective:** Verify content generation/selection when "Story" content type is chosen.
- **Procedure of testing steps:**
    - **1.** Select "Story" as the content type.
    - **2.** Start the audio stream.
- **Expected results/Outcome:** The content is streamed as a series of coherent stories.
- **Priority/Severity:** Critical
- **Actual Results:** The content streamed is actually a series of stories.
- **Discussion:** StreamLang's core functionality was satisfied.

Test Case 13: Stream Configuration - Translation After Each Sentence

- **Test Type/Category:** Functional
- **Summary/Title/Objective:** Verify translation is provided immediately after each sentence when configured.
- **Procedure of testing steps:**
    - **1.** Configure the stream to select "Individual sentences" content type.
    - **2.** Set the option to hear translations "after each sentence".
    - **3.** Start the audio stream.
- **Expected results/Outcome:** For every language learning sentence heard, the translation in the native language follows immediately.
- **Priority/Severity:** Major
- **Actual Results:** The ordering of the translations is correct.
- **Discussion:** The core functionality of StreamLang was satisfied.

Test Case 14: Listening State Persistence

- **Test Type/Category:** Functional
- **Summary/Title/Objective:** Verify that the user's listening state is recorded and resumed upon re-opening a stream.
- **Procedure of testing steps:**
    - **1.** Start listening to a stream template and stop at a specific time mark (e.g., 03:00).
    - **2.** Close the stream/application completely.
    - **3.** Re-open the application and click on the same template.
    - **4.** Start listening again.
- **Expected results/Outcome:** The stream loads and begins playing from the exact position where the user stopped (e.g., 03:00).
- **Priority/Severity:** Critical
- **Actual Results:** The streaming module has logical chunks of a few minutes, and restarting the stream throws the user back to the beginning of a chunk.
- **Discussion:** This might lead to a jarring user experience and may be improved.

Test Case 15: Streaming Scrubbing/Seeking

- **Test Type/Category:** Usability/Performance
- **Summary/Title/Objective:** Verify smooth and responsive seeking functionality within the streamed audio content.
- **Procedure of testing steps:**
    - **1.** Start a listening stream.
    - **2.** Use the progress bar/scrubber to quickly jump to several different points in the audio.
- **Expected results/Outcome:** Seeking is fast, accurate, and seamless, with no noticeable delay or stuttering in audio playback.
- **Priority/Severity:** Major
- **Actual Results:** Trying to move back sometimes leads to a stutter.
- **Discussion:** This might lead to a jarring user experience and may be improved.

Test Case 16: Streaming Performance - No Stutters

- **Test Type/Category:** Performance
- **Summary/Title/Objective:** Verify the streamed audio plays smoothly without stutters or interruptions.
- **Procedure of testing steps:**
    - **1.** Start a listening stream over a stable network connection**.**
    - **2.** Let the stream play continuously for an extended period (e.g., 5 minutes).
- **Expected results/Outcome:** The audio plays without any noticeable stuttering, crashing, or delay, confirming a smooth user experience.
- **Priority/Severity: Critical**
- **Actual Results:** There are occasional, very short artifacts (less than one second per two minutes).
- **Discussion:** A postprocessing batch job may detect speech boundaries and clip these artifacts out.

6. Maintenance Plan and Details

If the application is to be deployed in production, it will need to be maintained to make quality of life improvements, catch and fix any data or logic errors, and add application features.

As of right now, there are no automated maintenance tools. Since the application makes use of AI models, advancements in available products will be followed. The models we use will be updated to newer ones when available if they offer substantial improvements in capability without requiring disproportionately more advanced hardware.

The libraries we use right now meet the requirements for our implemented functionality, so major updates will not be done until the depreciation of their current versions. We will routinely do minor updates (if/when available) on the libraries. If we become aware of any security-related update, something GitHub sometimes automatically alerts its users with, we will install patches immediately.

No maintenance is planned on the stored data, barring any errors detected by the application. For an application of this scale, at the start, not keeping any data would be redundant at the expense of the convenience of users. This includes object storage, as once the content is in the database, it may be served to other users.

Tracking any bugs that are detected by a user will be done using the application stores from which the application can be downloaded from. They will be checked regularly. Any complaints will be forwarded to the team, with bugs always taking higher priority. Feature suggestions will also be received the same way. They will be forwarded to the team and have the lowest priority among the feedback received from users.

The team will always keep an internal list of possible improvements separate from user-contributed ones. This list will always have more priority. Any team member may, at their own discretion, add an item to the list after examining the codebase or the application itself. After internal discussion, the team may move any item in the user suggestion list to the higher priority list.

Maintenance is an evolving procedure and can only be planned so far ahead for an application in any given state. The maintenance plan outlined here may be modified as the application requirements change in the future.

7. Other Project Elements

7.1. Consideration of Various Factors in Engineering Design

7.1.1. Constraints

StreamLang is affected by some non-technical and technical factors. Those factors, with their detailed descriptions, were previously listed in our Detailed Design Report. In this section, we further describe why we consider those factors while also discussing the actions we took during the consideration of those factors.

The first non-technical factor we considered was public health. We considered this factor due to the mobile app nature of StreamLang. Our only concern was the effect StreamLang could have on the screen addiction, eye health, and orthopedic health of users. Even though this was an insignificant factor, we still made sure to make StreamLang carry absolutely no addictive factor. In the final product, longer usage times are not rewarded anyway. Also, any type of achievement stress is avoided by introducing absolutely no competitiveness.

Similarly, safety was another factor we considered. In our app, we had to use external sources and LLM-generated content for streams. We were aware that it is theoretically possible for the streamed content to be biased, harmful, or misinformative. Building on this, we designed the LLM Orchestration with a built-in review system to filter this kind of content and make the streams appropriate for everyone.

We also considered security. This factor was considered because we store user account information. We acknowledged that the breach of user data would be a significant privacy and security threat, and took various decisions during development to prevent such a scenario. That is, we made use of known best practices, used privacy-enhancing technologies like encrypting and hashing, and configured our network firewalls to enhance security.

Due to the computing power we had to use, we also considered the environmental impact factor. As the reason for the high compute power requirement was the usage of LLM generation and ML classification models, we decided to make efficiency an important target. We used the smallest, easiest to run models as long as they provided enough quality in their response. We utilized aggressive caching strategies to depend on LLM generation as little as possible.

The high computational power usage was not only affected by the environmental impact factor, but also by the economic limitations. Acquiring such compute instances in the cloud was not cheap, and, without our efficiency improvements, keeping StreamLang running would’ve been costly. Here, smaller models and caching allowed us to use our own hardware for our heavy subsystems.

The last constraint we considered was a positively affected one, welfare. As StreamLang is a language learning platform, we acknowledged that it is able to improve the listening skills of people in a very cheap and accessible way. To build upon this, we've tried to make StreamLang easy to use by all age and income groups.

We also had some technical constraints. With the current market expectations, we needed a responsive cross-platform app with good UX design. Alongside design, we also needed compatibility with AI systems and ease of development to be able to improve StreamLang with the new technologies.

To be able to solve the complex technical constraints, we chose a combination of Python, PostgreSQL, and CloudFlare R2 Object Storage for the backend. Python, although often not chosen for enterprise backend development, offered ease of development with industry-standard packages available for AI integration. Also, even though Python can be more computationally expensive than alternatives, being able to run on almost any environment brought a significant advantage for deployment. Postgres offered fast relational storage, meaning it allowed us to store all user data in a single database with no performance limitations. CloudFlare R2 offered fast, easy-to-set-up, and pay-as-you-go object storage for our generated audio files, allowing us to distribute the audio files through a fast and reliable content delivery network while keeping costs under control [5].

7.1.2. Standards

To ensure consistency and maintainability, we followed a set of standards. Those standards were already specified in the Detailed Design Report, and we did not have to make any changes. Thus, our final version of standards is as follows:

- **REST API:** The "REST API" standard is used for communication with and structuring the backend services.
- **System Modeling:** All architectural modeling (including system design modeling) is constructed using the UML 2.5.1 standard.
- **Effective Dart:** The mobile application codebase adheres strictly to the "Effective Dart" style guide to ensure code clarity.
- **Google Python Style Guide**: The backend server modules strictly follow Google's Python style guide.
- **IEEE 830-1998:** Official project documentation employs the IEEE 830-1009 standard for referencing styles.
- **Markdown:** Our internal documentation (including the API specifications) uses the markdown file format.

7.2. Ethics and Professional Responsibilities

Throughout the project, we evaluated ethical responsibilities from both the perspective of inner-team relations and from the perspective of the effect we have on the outer world. To carry out the former, we acknowledged that it is the responsibility of everyone in our team to ensure that a healthy work environment is maintained. We always prioritized mutual respect during our communications and made sure to fulfill our duties as they were given to us. Every one of our peers in the team was treated equally, and responsibilities were distributed fairly.

When looking at the effect of StreamLang on the outer world, we listed the ethical considerations we have to follow during the early stages of development. A list of such considerations was presented in the Analysis and Requirements Report. By paying attention to the effect StreamLang has on that problem, we as developers tried our best to minimize StreamLang's negative effects and tried to maximize its positive effects. In this section, we discuss our approach and decisions to achieve this.

Our first and most important ethical consideration was respecting the ownership of content. We acknowledged that StreamLang had to make heavy use of already existing content and AI generation. Thus, from the beginning, we recognized the importance of ownership and copyright of any published work. To populate our dataset using already existing work, or to use code readily available on the internet, we checked licenses and only make use of what we are allowed to according to their clauses. Accordingly, we prioritized using publicly available content for our listening streams. For the fully AI-generated content we create, we claim no ownership and consider any generated text available to the public.

Our second ethical consideration was the effect of StreamLang on public health. As can be seen in Section 7.1.1, we deemed that StreamLang did not have a valid negative effect on health as it was not designed to be addictive in any way, etc. Thus, we did not find it necessary to take any action in this regard.

Another ethical consideration was privacy. StreamLang's impact on privacy was considered low, as we do not store or process any personally identifying or sensitive information. Yet, we acknowledged that users may prefer to keep their configuration usage private. To honor this, we made sure to protect user data with row-level security and trusted only access to the database.

We also considered misrepresenting and disturbing languages as an important ethical consideration. As we used previously generated content and, more dangerously, LLM-generated content, we were aware that some of the streamed content may have been inaccurate, offensive, discriminatory, etc. To eliminate this risk and to fulfill this ethical responsibility, we designed the LLM Orchestration so that it has a built-in review system to filter this kind of content and make the streams appropriate to everyone.

Our last consideration was expensiveness. As we partly depend on LLM generation, and the compute costs for running those models are expensive, we preferred to use smaller models,

allowing us to partly use our own compute power rather than subscription-based cloud solutions. Additionally, we reduced our dependency by making the generation a fallback mechanism and utilizing the already generated and cached contents.

7.3. Teamwork Details

7.3.1. Contributing and functioning effectively on the team to establish goals, plan tasks, and meet objectives

Mehmet Emin Avşar

I have created a vast majority of the issues on GitHub. I also took the initiative in calling for meetings. I have followed up with people frequently to see how their work is going on. Above all, I tried to work hard myself to set an example for others.

Göktuğ Ozan Demirtaş

My main responsibility was implementing the Streaming Module. I also, on the request of teammates, implemented SRS tracking. Other than these, I made sure to make quality-of-life changes in other parts of the codebase when I saw fit (e.g., I streamlined internal API requests using simple wrappers) and did refactoring when necessary for maintainability.

Ruşen Ali Yılmaz

I was responsible for the design and development of the mobile application. I designed the UI and the UX with active feedback from the team. As the project requirements changed, I refactored our design to fit within our changes. I communicated closely with my team for the API definition and change requests. Aside from my frontend work, I also host a few of our AI models, such as the generation module's LLM and the text-to-speech engine.

Can Tücer

I actively participated in every team discussion we had. I tried to take a role in the decision-making by sharing my opinions whenever possible. I also participated in code and pull request reviews, report preparations, etc. During development, I worked on the Core Backend module. I carried out the tech stack research and was responsible for selecting and setting up the optimal Cloud services for our needs.

Uygar Bilgin

I designed and implemented the Content Categorization Module of StreamLang. During this process, I have worked on the necessary tasks that are required by my teammates and delivered them on time. I implemented GitHub issues related to my task, edited and closed them accordingly. I communicated with my teammates to integrate our work seamlessly.

7.3.2. Helping creating a collaborative and inclusive environment

Mehmet Emin Avşar

I have been respectful in my communications with team members. If they could not attend a meeting or had other priorities in life that led to them falling behind schedule, I was understanding and practical instead of judgmental. We met in EA about every three weeks to discuss the project and do some implementations together. I tested the application, and if I needed to work with the Streaming Module, for example, I promptly set up physical meetings with the supervisor of that module, without bothering the supervisors of other modules. In this way, I have achieved both collaboration and efficiency by letting everyone focus their energies where it matters most.

Göktuğ Ozan Demirtaş

When developing my parts of the codebase, I always followed the process of deciding on details with my teammates before actually implementing functionality. Whenever this was not possible, I made sure to document and/or communicate any design choices I made to any cohort who may need to know these decisions. I was always available for feedback through the entire project, and I never backed down from assisting my teammates when help was requested, in debugging, in taking on work, etc.

Ruşen Ali Yılmaz

At each and every step of development, I communicated clearly with the team. As I developed a significant portion of the application side of our project before the development of the backend started, I defined the API specification that the frontend requires to function properly. I was always open to the team's requested changes and implemented the requested changes promptly. When the backend tasks took longer than desired, I stepped in and started hosting the compute-heavy AI tasks on my local home cluster.

Can Tücer

I tried to help as much as I can when one of my teammates had an issue I've faced before. I tried to take tasks from my teammates whenever I was done with mine. I've tried to create a positive and friendly environment in all of our meetings. I always greeted any feedback I got from my teammates with great interest.

Uygar Bilgin

I frequently sought feedback from my peers and brainstormed with them at our meetings. We held regular physical meetings, where we sat down and worked on our project together, talking about issues we encountered in the project so that we could effectively solve them.

7.3.3. Taking lead role and sharing leadership on the team

Mehmet Emin Avşar

I've written the vast majority of the issues on GitHub. I have communicated clearly with my teammates over the phone to let them know their responsibilities. I have left space for autonomy in making the task division: for example, the supervisors of the content categorization module designed it entirely by themselves.

Göktuğ Ozan Demirtaş

I was solely responsible for developing the Streaming Module and made all the internal design choices. I did my own research and settled on what I believe to be an adequate streaming stack and server architecture for the module, according to the constraints given by the project. Other than the work I was personally leading, I made sure to voice my opinions and make wider refactorings/changes in other parts of the project for the sake of a better codebase, like I did when implementing SRS tracking.

Ruşen Ali Yılmaz

As the solo and lead developer of the frontend, I believe I took a significant lead. Although it was not assigned to me, I took the initiative to design the frontend and host AI models. However, whenever a change was required, I listened and communicated with the team on the desired changes.

Can Tücer

I tried to give constructive feedback on every decision we've taken and implementation we've made. I've reviewed pull requests, and I've asked for revisions whenever necessary. I've taken a very active role in the selection of our tech stack and in the design of cloud systems. I've chosen our cloud service providers and made the necessary arrangements to get everything up and going.

Uygar Bilgin

I worked mostly on the Content Categorization Module, making design choices about what the module was going to provide and what function it was going to fulfill. During our later stages, I worked on acquiring a sample content dataset and ensuring it worked correctly with my content categorization implementation.

7.3.4. Meeting objectives

Right now, StreamLang works as a mobile application that allows anyone to listen to a stream using the grammar points and vocabulary they configured in German. Even though we initially designed the application to work in several different languages, we switched to German only afterward. Our supervisor advised us to focus on the technical aspects and not lose too much time trying to implement many different languages, and we found the advice helpful.

Another objective that we could not meet fully was being able to select the voice speed in a wide range. We could not find a free and accurate multilingual narration model that allowed for a speed parameter to be specified. Other methods we tried, such as postprocessing, led to unacceptable distortions in audio.

Overall, we could meet the timelines promised in the Project Plan Section of the Analysis and Requirements Report with no issues. The rest of our development went as planned, and we achieved the majority of our objectives.

7.4. New Knowledge Acquired and Applied

Mehmet Emin Avşar

I have gained valuable experience using Claude Code in charting the development of a medium-sized project from start to finish. I deeply believe that I've developed a closer affinity with a tool that every software developer should be able to use professionally. I have also learned new tools for testing APIs such as Bruno; open-source alternatives to paid tools. I have gained good experience with Docker - my abstract knowledge about it has been enhanced with experience. In overview, I now have increased confidence in my ability to bring a good idea to fruition.

Göktuğ Ozan Demirtaş

My knowledge base expanded considerably throughout the project. I finally learned Docker, a very widely used tool to streamline software deployment, and a rite of passage for most software developers. I also learned of other tools like nginx and GStreamer. While they are not exactly general-purpose, learning itself is an ability that can be improved, and it is best improved by learning. Perhaps most importantly though, for this project I decided to use more AI tooling as someone who doesn't dabble in them too much. After using AI in this project, I see the value in using AI as a tool for quick learning and quick coding. However, as per my expectations, it is not a replacement for doing the work or learning the tools yourself, and I believe noticing the weaknesses of tools like these is just as valuable as recognizing their capabilities.

Ruşen Ali Yılmaz

As a prior Flutter developer, my technical knowledge did not expand much, aside from the RTSP audio streaming, for which I used the common “Media_Kit” library for Dart. Due to the need to test before the backend was up and running, I also learned GStreamer to test audio streaming functionality. I learned how to properly and effectively use Figma for design purposes, as I had only used Adobe XD beforehand.

Can Tücer

I’ve probably gained most experience in presenting and reporting product design. Throughout the project, I’ve learned a lot about how to prepare design reports with the required information and readability. I’ve also learned a lot about Flask, LLM orchestration, networking, and system design, as I had a chance to review and work on those parts of our project. I’ve had a chance to see the latest developments in LLM and Text-to-Speech technology, and discovered that, even now, they are advanced enough to shift the traditional ways of learning a language. Overall, I think this was a good product design from scratch experience, and I can apply the knowledge I’ve gained to any other development work I may work on in the future.

Uygar Bilgin

Because I mostly worked on the Content Categorization Module, I have gained valuable experience on the framework I mainly used, SpaCy. It helped me better understand a natural language processing (NLP) model to segment sentences and categorize them based on their German grammar attributes. In addition, I have gained experience in all the technologies I used, such as Docker and Flask. Lastly, this project helped me gain experience in working on a project larger than what I had worked on before, while showcasing the importance of teamwork.

8. Conclusion and Future Work

It was a journey. To crown our four years in the Bilkent CS department with a substantive project that we can entirely call “our own” with the idea, execution, and product is a satisfaction like no other. StreamLang now stands as an application where anyone can define the grammar structures and vocabulary they want to listen to, and then indulge in immersive content precisely tuned to their level. As previously stated, we only have German right now. To add French and Japanese is our future goal. Beta-testing the product with real users and running incremental improvements for one-two months would be the highest return next step.

9. References

[1] D. W. Price et al., “The effect of spaced repetition on learning and knowledge transfer in a large cohort of practicing physicians,” Academic Medicine, vol. 100, no. 1, pp. 94–102, Jan. 2025, doi: 10.1097/ACM.0000000000005856.

[2] GStreamer Team, “GStreamer: open source multimedia framework,” GStreamer. [Online]. Available: https://gstreamer.freedesktop.org/. [Accessed: Mar. 12, 2026].

[3] nginx, “nginx,” nginx.org. [Online]. Available: https://nginx.org/. [Accessed: Mar. 12, 2026].

[4] Amazon Web Services, “Boto3 documentation,” AWS Documentation. [Online]. Available: https://docs.aws.amazon.com/boto3/latest/. [Accessed: Mar. 12, 2026].

[5] Cloudflare, “Cloudflare R2,” Cloudflare Docs. [Online]. Available: https://developers.cloudflare.com/r2/. [Accessed: Mar. 12, 2026].