Bilkent University

Department of Computer Engineering

Senior Design Project

T2533

StreamLang

Detailed Design 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

13.03.2026

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 outlines the detailed design of the StreamLang language audio streaming application. A summary of the application purpose, design goals, and the specific terminology related to StreamLang is given first. This summary is followed by a survey of the (inferred) software architecture of the competitors in the same domain trying to address the same problem. Then comes an overview of the software architecture proposed by StreamLang, the persistent data management and access control strategies adopted in our solution. Then, the detailed design of each and every subsystem is laid out, at a lower-level than previous design documents, with the help of UML diagrams and now with specific code snippets. In the next section, specific scenarios are given for software test engineers to manually verify that the software meets the requirements, and performs robustly. Finally, the social and ethical considerations in our design is described and a picture of how our teamwork played out is given.

1.1 Purpose of the System

The purpose of StreamLang is to support language learners in their grammar and vocabulary studies through the auditory medium. StreamLang does not aim to be the sole platform that a language learner develops their skills; rather, StreamLang aims to reinforce language concepts studied separately by increasing the exposure to those concepts via comprehensible input.

Studies show that comprehensible input helps with second language acquisition [1]. With StreamLang, users will be able to specify a listening stream configuration which will result in an optimally challenging comprehensible input. This way, StreamLang will enable users to most efficiently utilize the otherwise idle time they spend commuting or cooking. The intended effect is to act as a catalyst for acquisition.

1.2 Design Goals

In this section, we will discuss the high-level objectives and principles we follow while working on StreamLang, together with the implementation decisions we took to ensure we reach such objectives.

1.2.1 Usability

StreamLang, as a learning tool, should be easy to use. Users from all age groups and technology knowledge levels should be able to configure their streams and start listening without assistance. Thus, usability is one of our main concerns, and we are following different principles to ensure this. We are caching session tokens on the user's device to make logging into the app easier. We are prioritizing simplicity and accessibility in the user interface. We are trying to keep configuration options easy to learn while making sure users, when required, are able to configure the streams just like they want.

1.2.2 Performance

As we are working on a mobile app, users expect 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, all of the performance-heavy operations (i.e., keyword generation for vocabulary sets, text-to-speech, etc.) are completed on the server, providing fast response times and better error handling.

Furthermore, the server should also be responsive. Constantly/routinely operating modules should not meaningfully impact the operating speed of other modules. In this regard, the heavy burden is that some user requests may require content to be generated using AI models. Any request that does not generate content should respond to user requests within one second (excluding client-side network delays). Any content-generating request shall respond to the user within one second after the AI's work is done. We are following best practices and optimizations to achieve this. Additionally, our server architecture is fairly scalable, and if ever needed, we may make use of multiple endpoints running together via load balancers.

Also, to remove stutters from the streamed audio, we are designing the streaming system so that the audio streamed to the device is downloaded faster than it's played, and all downloaded audio shall be fully cached for one playback session to facilitate scrubbing.

1.2.3 Reliability

For the same reason as performance, we also consider reliability as an important goal. Here, the actions we take to improve performance also contribute 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, the servers should have at least 95% uptime and work without disturbance when not in maintenance. We are also planning to set up monitoring and notification systems for each subsystem to be able to take early actions. Furthermore, as each subsystem of StreamLang works completely independently of the other, we plan to guarantee that a crash/downtime in one of the subsystems (except for the core backend system, which links the frontend with other subsystems) would not cause the app to be unusable.

In addition to guaranteed uptimes, all data will be periodically backed up to ensure no data loss happens in case of a failure. For increased reliability, the data will always be in a valid state. In this regard, we will be taking extra precautions for issues like race conditions to not corrupt the data, and using PostgreSQL's capabilities to set up in-place condition checks so invalid operations can be denied and the database can be rolled back to the previous valid state instantly.

1.2.4 Security

As StreamLang stores no personal data, security and privacy are not one of our main concerns. Still, we are following best practices of all forms to ensure user data is kept safe and

private. For this, we are using industry-standard JWT tokens for request/response verifications. We are hashing passwords using Bcrypt.

We are also setting up the firewall configurations of our databases and servers carefully, while making use of Cross-Origin Resource Sharing to protect our subsystems against any adversary. Lastly, we are using HTTPS for client - backend communication.

1.2.5 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 are prioritizing scalability. The mobile client is a frontend-only application programmed in Flutter, so it requires no scaling as the user base grows.

However, our subsystems, especially the generative AI systems, require scaling as they have a high workload. For this reason, 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 take precautions against concurrency race conditions and are doing research on load-balancing structures. Modules using third-party services will be scaled horizontally so long as the service provides such an option. With this architecture, we believe that we should be able to adequately scale our application as user numbers change.

1.2.6 Maintainability

As StreamLang requires high uptime, easy maintainability that allows us to apply updates and fixes without disrupting the service is an important goal of ours. For this reason, StreamLang's server's modular architecture design allows for easy maintenance. When modules in need of maintenance can function independently of each other, application functionality that is isolated from any affected modules may keep running unaffected, while any fixes/rollbacks are carried out on problematic modules. In the case of horizontally scalable modules, when a module instance has an outage/must be taken down, another instance may take its place to serve users until the instance is back up. Server operations shall be logged to make bug-fixing easier. Client maintenance is a non-issue since the entire application is already built and on the device of a user. Any updates shall be delivered through the device's application store.

1.2.7 Flexibility

Right now, we are only working 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 need to achieve flexibility in the sense that adding new languages should not require core functionality updates. For this reason, we are making sure that we hold vocabulary and grammar themes as enumerated variables, and our scraping, generative AI, and streaming modules support the languages we are interested in providing in the near future.

1.3 Definitions, Acronyms, and Abbreviations

Spaced Repetition Algorithm (SRS): An algorithm that schedules review and learning of information at gradually increasing intervals to optimize long-term memory retention.

Configuration: The word “configuration” in this report may refer to a listening stream configuration; an object which defines what kind of listening stream that the user wants to hear (e.g. vocabulary, grammar structures, narrator voice).

Content Type: Refers to the three content types: sentence, article, or story.

Lemma: Dictionary, canonical, or base form of a word.

2. Current Software Architecture

Currently, we have identified three major competitors to our proposed application. These are Taalhammer, Lenguia, and Beelinguapp. First of these, Taalhammer, utilizes a hybrid cross platform approach, being available for Android, iOS, and web [2]. The primary frontend framework of Taalhammer is Angular, and the Angular codebase is wrapped using Ionic Framework and Capacitor to maintain a single codebase and compile to native Android and iOS applications [3]. For the backend, Taalhammer uses Python, similar to our application [4]. Because Taalhammer also heavily utilizes machine learning tasks, Python for backend is an optimal solution for both our app and theirs. Their backend consists of a high number of sentence pairs for different languages, in addition to complex state management to track the language level of each user. The Atom algorithm Taalhammer uses is their proprietary SRS method [5]. Their AI module specializes in generating context-based sentences rather than serving singular words, and their TTS engine generates the sound for the content, similar to ours.

Our second competitor is Lenguia, which is available only on the web and uses Next.js as the main frontend framework and Supabase as the backend [6]. Lenguia utilizes LLM APIs for personalized story generation and text simplification [7]. They also feature a custom SRS algorithm that tracks user comprehension at word level [8].

Our third competitor is Beelinguapp, which is available on Android and iOS, and is developed natively for both platforms [9]. Beelinguapp works on synchronizing audio with dual language text, used in its “Karaoke Reading” and “Side by Side Reading” functionality. Beelinguapp also utilizes space repetition methods to track user progress and retention.

3. Proposed Software Architecture

3.1 Overview and Subsystem Decomposition

For scaling and maintainability purposes, the backend of the project was divided into modules. Each module is only responsible for a specific job and their functionalities are implemented barring other modules, and only interact with each other if it is part of the application's flow (e.g. the core backend module may communicate with the streaming module to start a stream) . While any one member may push code changes to any part of the project, every member mostly works on their respective modules and are responsible for implementing the majority of required functionalities.

Another detail of the backend modules is that they are containerized. Each module is its own Docker [10] container, and the containers are managed as a whole using Docker Compose [11]. Each module takes important information, such as internal keys, module configuration, required endpoints etc. from environment variables. Thanks to this we have great flexibility in deploying the backend. This also constitutes horizontal scaling of modules that may want to provide support in the future.

The backend is divided into four modules:

- The Core Backend Module: The coordinator of StreamLang's backend. Responsible for managing user authentication, interaction and data. When necessary, will invoke other modules using internal routes.
- The GenAI Module: Our content, translation and narration generation facility. When no new content is available satisfying a set of requirements, will generate and validate new content using generative AI. Content will be translated by this module. Also responsible for generating audio narration for all content and their translations.
- The Content Categorization Module: Processes generated or scraped text content. Populates our database with the internal representations of content required by the other modules.
- The Streaming Module: The module responsible for streaming audio to users. Interacts with S3 Object Storage to pull audio files and streams it to users using RTSP over HTTP.

Finally, the frontend is a mobile application developed in Flutter. The API of the server is documented as much as possible before functionality is implemented to reduce friction between frontend and backend development.

3.2 Persistent Data Management

There is a massive amount of data needed for the operations of StreamLang under two categories: text of the contents that will be served to the users, and the audio files. We have opted to use PostgreSQL with the pgsearch extension for storing content text, under considerations of compatibility with our backend infrastructure. The pg_search extension implements the BM25 algorithm which is used by modern search engines and brings a full-text search query that takes 22,214 ms on native PostgreSQL down to 770 ms [12]. Using this

extension also allows us to avoid using two different databases for application information and content storage, dramatically reducing complexity and management burden. Very little cost in terms of implementation overhead and more than satisfactory performance makes this the obvious choice.

The second dimension of massive data that StreamLang needs to manage is the audio files that are the narrations of the content / generated or previously scraped. There are three considerations here that are critical:

- **Handling high and unexpected data volume:** As we give any user the ability to create their own listening streams, the corresponding audio files will scale linearly with the number of users. Given that audio files are much more data-intensive than text, and that spikes in user count will lead to the generation of more audio, the storage space should be very high.
- **Handling high egress traffic:** While many users will be served recycled content based on their listening stream configuration and therefore the ingress traffic is expected to diminish over time, the egress traffic will scale linearly with active user count.
- **Affordability:** While satisfying all of the above constraints, the solution should not be prohibitively expensive to be outside the reach of five undergraduate students.

In order to satisfy all of these constraints, we opted to use Cloudflare R2 [13] distributed object storage. The most significant attractor for this decision was the pay-as-you-go model that will work well with a low user count application, since renting and managing a file service infrastructure ourselves has a high upfront cost and no guaranteed return. Using a cloud service like S3 also provides flexibility to handle traffic spikes and presents virtually no limits on the egress traffic and data volume that can be handled.

3.3 Access Control and Security

StreamLang holds user accounts to be able to provide customized listening experiences (i.e., persistent streaming configurations and progress saving). For this reason, each user account carries a unique UUID identifier. Accounts may be created using email and password pairs, and emails are unique to accounts. Given passwords are hashed via Bcrypt, and only hashes are stored in the DB. Account logins also require email and password. After initial login, account verifications in transactions are done via JWT tokens. To achieve this, after the login operation, the core backend creates a unique token for an account, which has a set lifetime, and allows access to login-requiring routes (i.e., any route except register or login). Those tokens are stored by the mobile client.

Both our PostgreSQL DB and S3 Audio Object Storage unit implement a firewall to the extent that only our submodules are able to access them. Passwords and private keys to those storage services and to our cloud computing instances are always kept private. All transactions are encrypted and sent through HTTPS.

Those measures ensure that all user information is always kept private and secure, and that our service is protected against adversaries (as long as the third-party providers we work with keep their promises).

4. Subsystem Services

4.2.1 Core Backend

4.2.1.1 SRS Mechanism

The SRS algorithm is our method of choice for scheduling reviews for any given user. The algorithm, on a surface level, works by taking a graded user review, and calculating an interval until the next review based on some pretrained and/or user trained weights with precisely defined terms and heuristics.

Specifically, the algorithm works on three variables that are stored as a representation of the user's memory regarding any given study material:

- Retrievability (R): Probability that the learner can recall specific information at a specific time.
- Stability (S): Time required for R to decrease from 100% to 90% in days.
- Difficulty: Complexity of a given information.

Any time a user listens to a stream, i.e. "reviews" that stream, all three values are recalculated based on their grading. The next review's due date is finally calculated based on these three variables.

Our application uses FSRS6, a variant of FSRS that operates on 21 weights [14]. For our applications, we picked pretrained weights [15]. Normally, in an optimal scenario, the weights would constantly be trained taking into account the user's review history, but for the purposes of our application, we decided to not train the variables. This should still yield decent results, since it is reportedly still better than Anki's default scheduling algorithm [], which is a very popular review application.

Another choice we made is to use a predetermined rating for every user review, as we currently do not plan to provide a mechanism for a user to grade any certain stream. Our application intends to provide a more passive learning experience, and as will be discussed later, requiring the user to rate every tracked content would be too cumbersome for users. Rating is a 4 value system:

- Again: 1
- Hard: 2
- Good: 3
- Easy: 4

In FSRS6, "Again" is the only failing grade, with "Hard" being the value that increases stability the least. We chose to use a "Hard" grading as the default review grade.

The specific equations used for the algorithm [14] are as follows:

$$I(DR, S) = \frac{S}{0.9^{-\frac{1}{w_{20}}} - 1} \cdot (DR^{-\frac{1}{w_{20}}} - 1)$$

Fig. 1. Interval calculation [14]

$$S'(D, S, R, G) = S \cdot (1 + w_{15} \cdot w_{16} \cdot e^{w_8} \cdot (11 - D) \cdot S^{-w_9} \cdot (e^{w_{10} \cdot (1-R)} - 1))$$

Fig. 2. Next stability calculation [14]

$$D_0(G) = w_4 - e^{w_5 \cdot (G-1)} + 1$$

Fig. 3. Initial difficulty calculation [14]

$$\Delta D = -w_6 \cdot (G - 3)$$

Fig. 4. Next difficulty calculation based on grade [14]

$$D' = D + \Delta D \cdot \frac{10 - D}{9}$$

Fig. 5. Dampened Next difficulty calculation [14]

$$D'' = w_7 \cdot D_0(4) + (1 - w_7) \cdot D'$$

Fig. 6. Final Next difficulty calculation [14]

$$R = (1 + factor \cdot \frac{t}{S})^{-w_{20}}$$

$$factor = 0.9^{-\frac{1}{w_{20}}} - 1$$

Fig. 7. Retrievability calculation [14]

The algorithm also includes a short term stability value calculation, but we do not use it in our program.

For each user, the tracking is done per “item” per stream. Item in this case means a vocabulary -a collection of words- or a grammar point, which represents a particular grammar construct. This is because a user may partially listen to a stream, and scheduling reviews only taking into account the parts of a stream a user listened to will prevent a user from skipping reviews for a particular item they want to learn.

4.2.1.2 Content Selection

In order to describe how the Content Selection mechanism of the Core Backend has shaped up through our development, this section will utilize detailed UML diagrams to give a deep sense of the code-flow. Unlike with previous reports, low-level details about individual function calls will be given. The main purpose of the Content Selection mechanism is, given the listening stream configuration of a user, selecting the appropriate contents to be streamed. The content id's, if found, are sent to the Streaming Module which is the module that establishes a connection with the client (described under its own section). The general design goals of the Content Selection mechanism is outlined in previous reports; here, the focus is more on the details.

The following sequence diagram gives a focused tracing of the control flow of the /stream/start endpoint, which the client calls to get a stream ID that can be plugged in to a URL already known by the client to start streaming.

The sequence diagram, due to the amount of detail, is massive, and is therefore divided into four parts to better fit inside the document. Each part will be shortly explained before the next part is presented.

For more comfortable, high-resolution viewing of the sequence diagram, we recommend downloading it from [this link.]

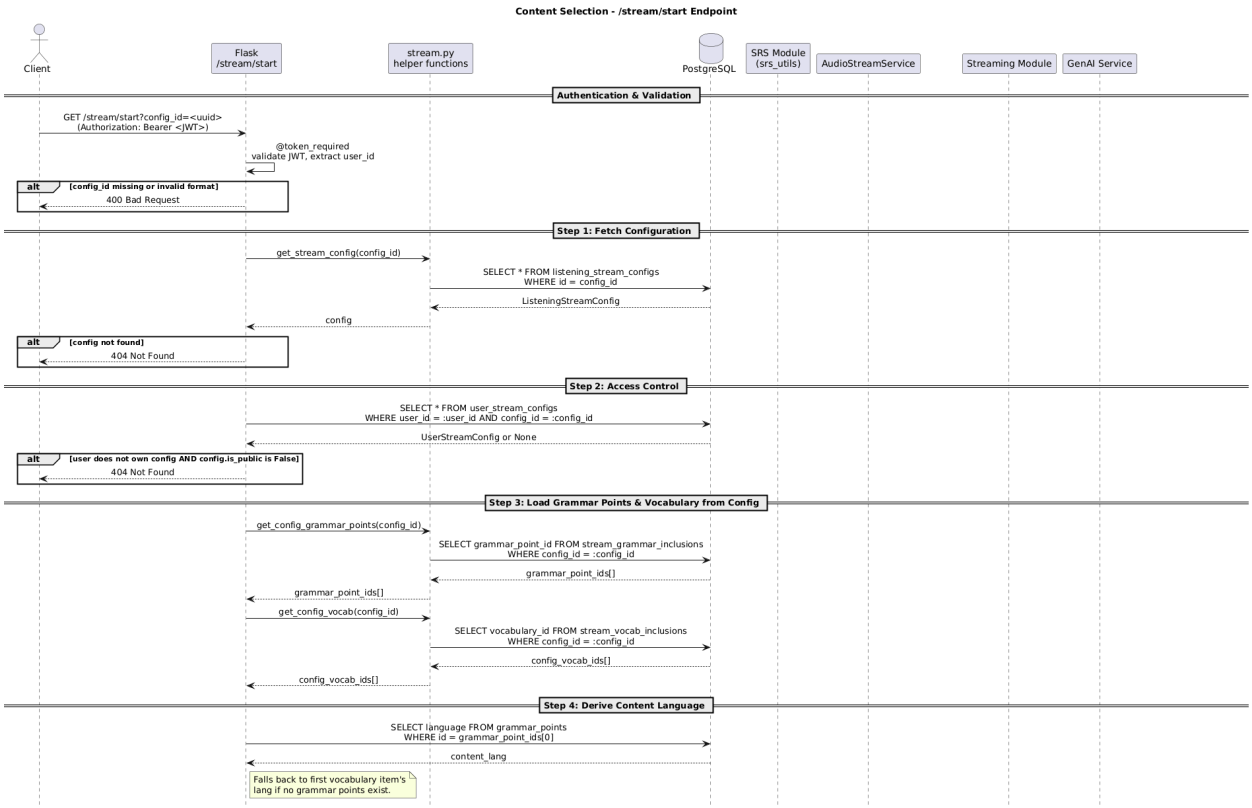


Fig. 8. First part of the Content Selection Sequence Diagram

First, the user is authenticated via a python decorator function “@token_required”. The use of a decorator function for authentication exhibits a principled coding approach where the same functionality is concentrated in a single point and repetition is avoided. Then the configuration is fetched from the database. Here, in order to prevent users from accessing listening stream configurations which are not theirs and also not public, the database table “user_stream_configs” is consulted. In the case of an unauthorized access attempt, the 401 Unauthorized response is **not** sent; instead, 404 Not Found response is sent. This design aims to prevent leaking information about the existence of a listening stream configuration with a given id. After access permission is verified, the grammar points, vocabulary and the language of the config is collected from relevant database tables.

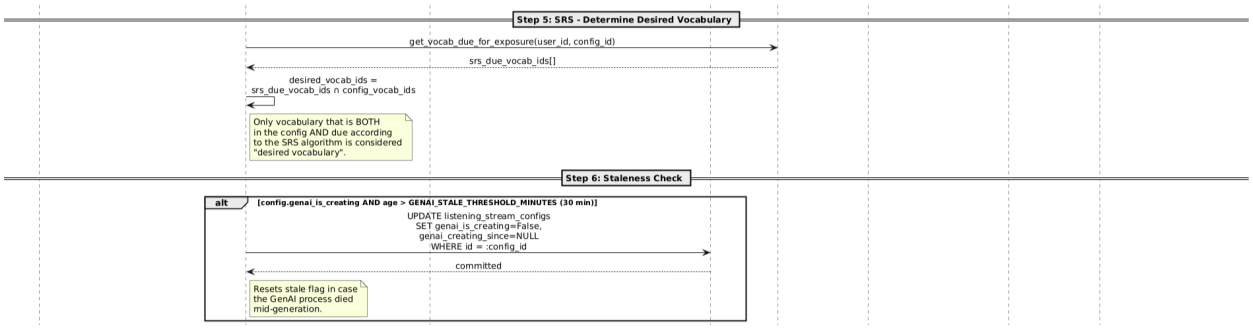


Fig. 9. Second part of the Content Selection Sequence Diagram

The SRS functionality is called upon to get the vocabulary due for exposure according to the user’s schedule (explained in more detail under Section 4.2.1.1 SRS Mechanism).

After that, there is an important **robustness** feature that is a sign of conscious, well-thought out design. Every listening stream configuration has a flag “genai_is_creating” and “genai_creating_since”. We will see, in further steps, if the *genai_is_creating* flag is true, the Core Backend does not trigger GenAI module’s content generation mechanism even if no suitable content is found - the assumption is that GenAI is already creating, and we do not want to queue the same job again. But what if something went wrong during generation in the GenAI module, like a crash? The flag would be stuck in true, and the user could never access the content they want. That’s why a staleness check of the *genai_is_creating* flag in the Core Backend resets this flag every *GENAI_STALE_THRESHOLD_MINUTES* minutes, to prevent such a disaster.

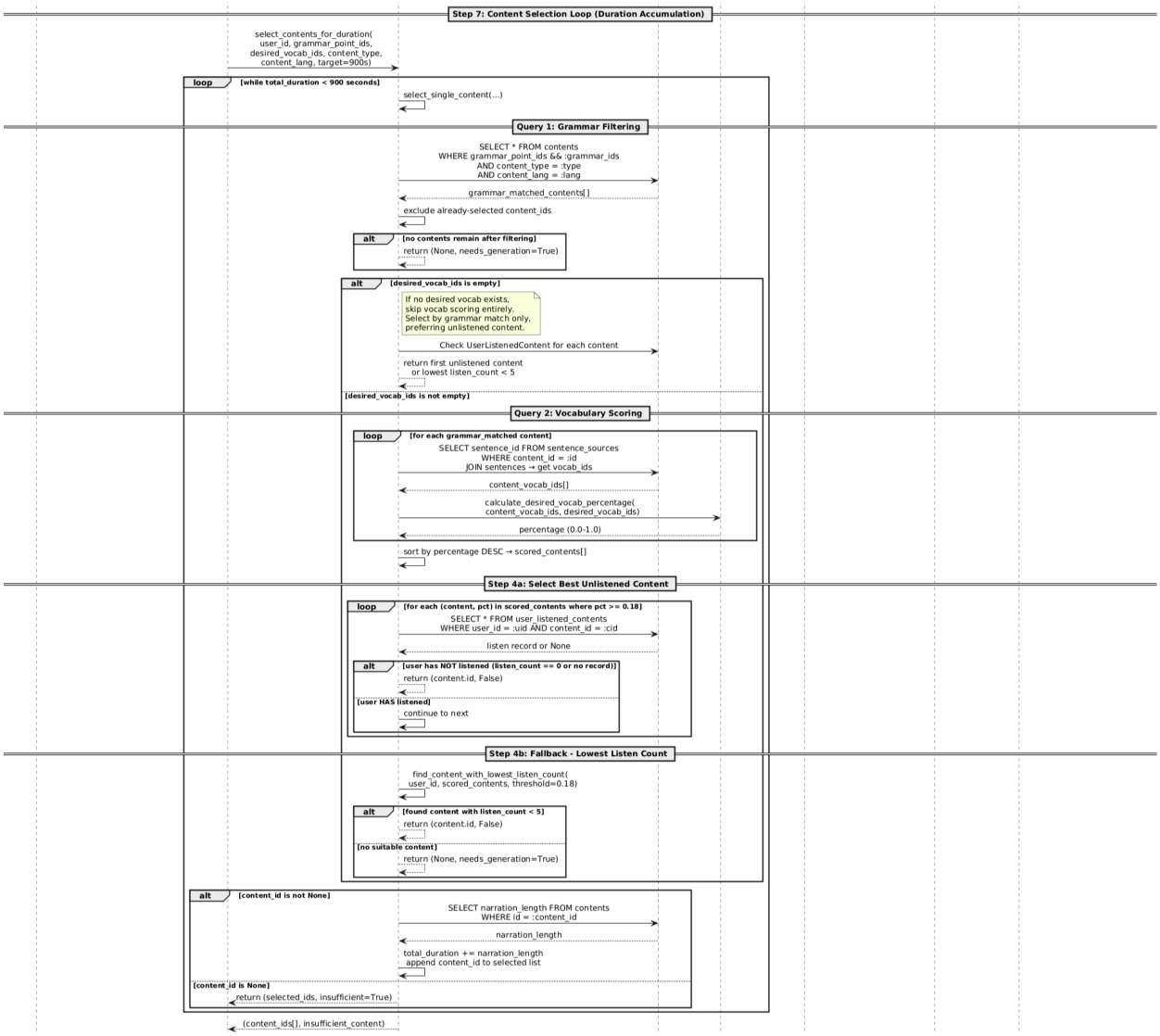


Fig. 10. Third part of the Content Selection sequence diagram

The third part of the content selection sequence diagram describes the most important functionality. A loop selects content IDs from the database until the total content length reaches 900 seconds. This means that a total narration length of 15 minutes of content is scheduled at a time. Then, the priority is to find the suitable content from the content database first and, only if this is not successful, trigger AI generation. A query is run on the content database that filters the content according to the grammar points desired. Every content must have the grammar structures that the listening stream configuration designates; this is non-negotiable. However, the vocabulary selection is a little more lenient. First, a call to the SRS system is made to retrieve for which vocabulary in this config the time for exposure has come. Then, whichever content has the highest percentage of the desired vocabulary (with a lower limit of 18%) is served first. The constraints for whether the user listened to the content before or not is taken into the equation as well; the details can be traced on the sequence diagram. If a user has listened to a content five times already, the system counts that content as non-existent for that

user. Since we trigger GenAI content generation if we cannot fill a 15 minute listening stream, there is an implicit constraint that the content has not been listened to in the past 15 minutes as well.

The explanation “if no desired vocab exists, skip vocab scoring entirely” on the sequence diagram refers to the fact that when there is no vocabulary scheduled for exposure at that time for the user, it is better for the scheduler to ignore that constraint - since FSRS relies on exposure right before someone is about to forget, and premature exposures may undermine long-term retention.

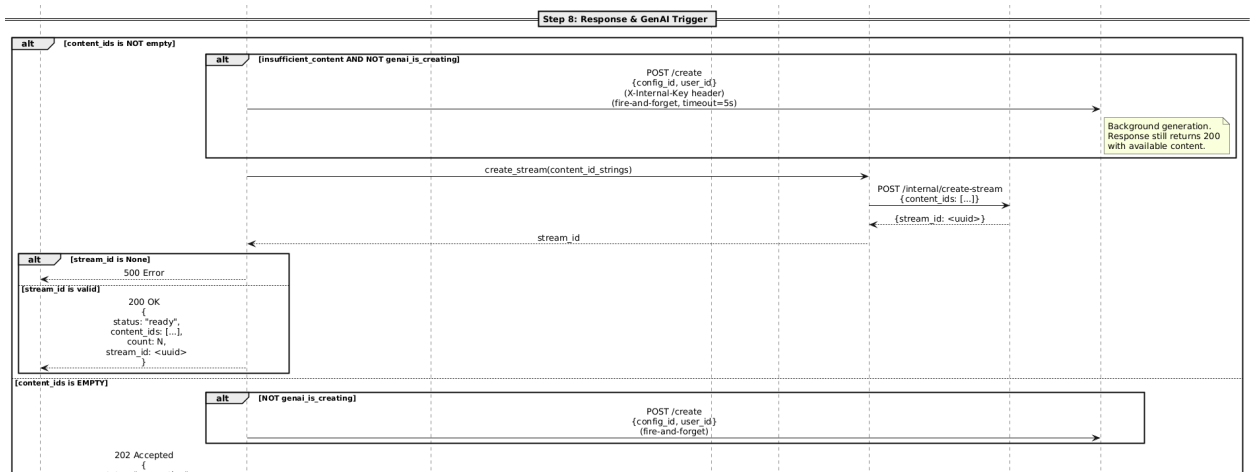


Fig. 11. Fourth part of the Content Selection Sequence Diagram

As it can be seen, if there is insufficient content in the content database, the system triggers GenAI generation. If there is already GenAI generation going on on the same listening stream configuration, the system does not trigger generation again. There is a sophisticated control system of GenAI triggering that will be best described with a state diagram, as in the following figure.

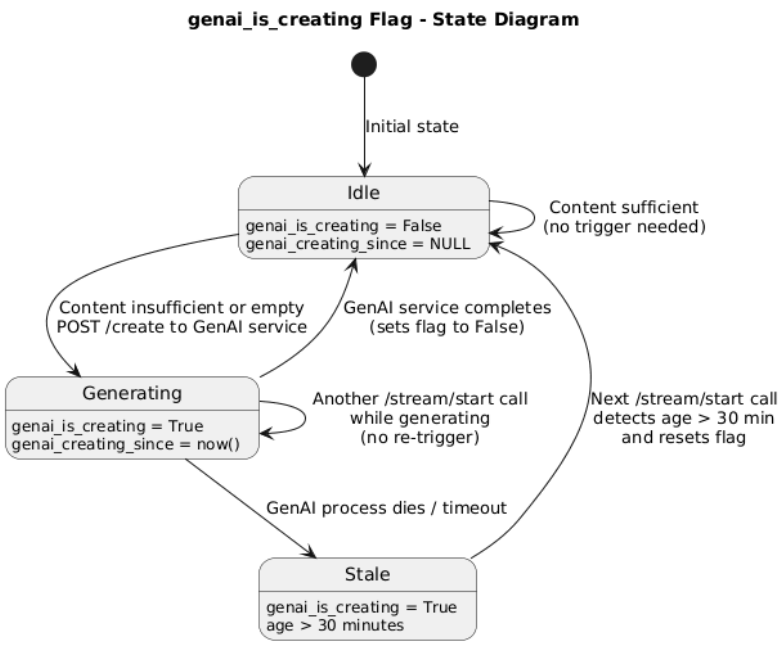


Fig. 12. State diagram of triggering GenAI based on an “already creating” flag

The GenAI internal service’s endpoint is called with a secret key generated with OpenSSL, which polls the operating system at /dev/urandom to get a true source of randomness conjured up from noise from hardware interrupts and thermal jitter.

In between a call is made to the Streaming Service with the contents that are chosen for streaming, and a proper response code is sent to the mobile user.

If the sequence diagram was hard to follow due to the inevitable complexity of the low-level details given, the following activity diagram may serve as a higher-level human readable overview of the content selection mechanism.

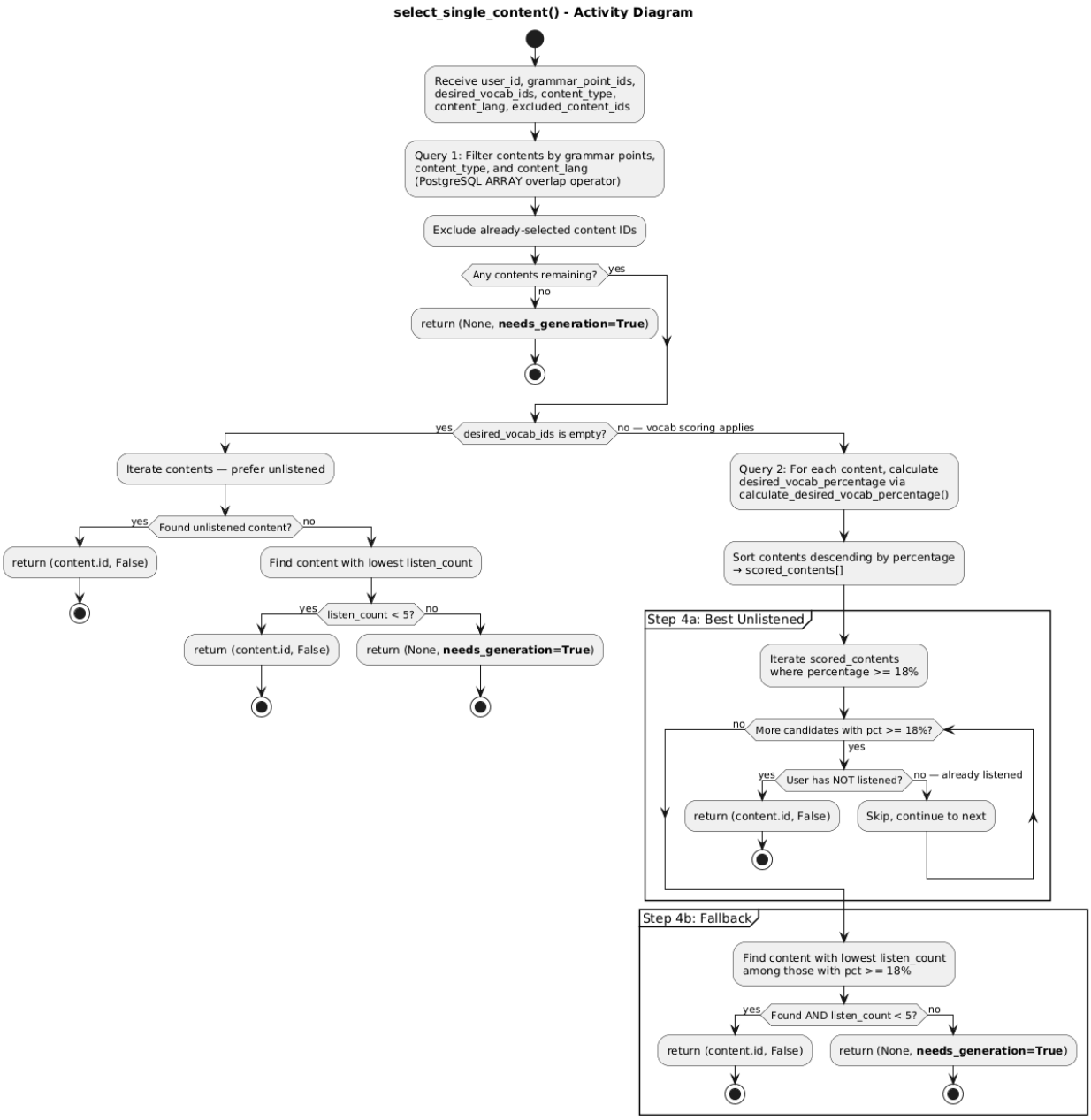


Fig. 13. Activity diagram of the content selection mechanism

4.2.2 GenAI

The purpose of the GenAI module is to take a listening stream configuration, and both generate and narrate content befitting the constraints. There are three main criteria that this module should satisfy

- *Correctness*: The content generated should use language correctly and satisfy the listening stream configuration. The narration should contain minimal artifacts and the pronunciation should be right.
- *Engagement*: The content generated should be interesting.

- *Novelty*: When generating long stretches of content for the same configuration, do not repeat yourself.
- *Reasonable speed*: We can make the user wait perhaps five minutes but not an hour.

Ensuring correctness is achieved with a mix of prompt engineering to instruct the LLM model what vocabulary / grammar structures to use, and using a strong LLM model (like Claude Opus). The GenAI module has been tested with local LLMs for now. Connection to a cloud LLM is in the near future.

Engagement and novelty is related to LLM complexity and prompt engineering. The GenAI module first instructs the LLM to generate a number of distinct subtopics, and then injects these subtopics to further generations to diversify the content.

Reasonable speed is related to hardware capabilities. This criteria may be in tension with the other ones as a stronger LLM means slowness but better correctness and the ability to create engaging content.

The entry point of the GenAI module is the /create endpoint. We now add on to the previous reports by showing the detailed code flow in a sequence diagram. The module is implemented in seven different subservices: the generation service, the LLM client, the prompt builder, the content processor, the narration service, the TTS service, and the S3 service. Since the sequence diagram is quite large, it will be presented in parts. The diagram can be downloaded here for a better viewing experience.

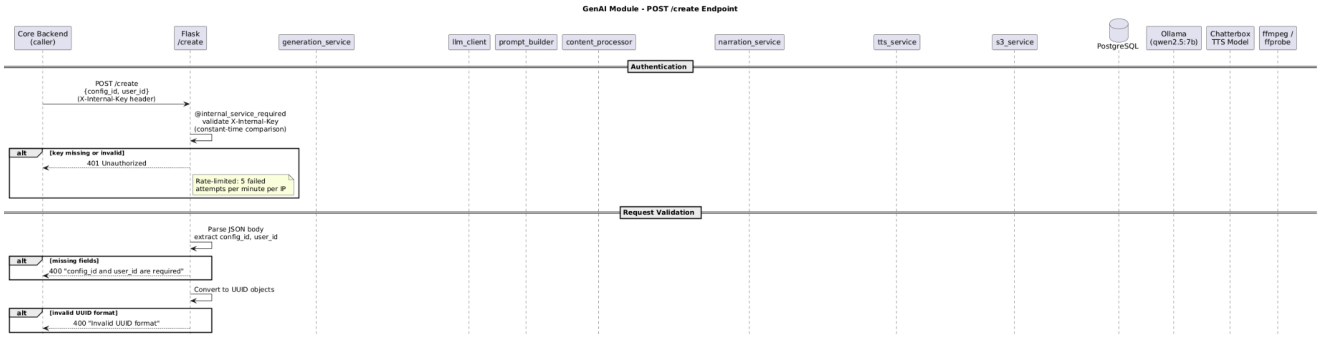


Fig. 14. First part of the GenAI sequence diagram

First, we authenticate the request. Every internal service request requires a secret key only known by StreamLang modules. Then, the presence of config_id and user_id are checked.

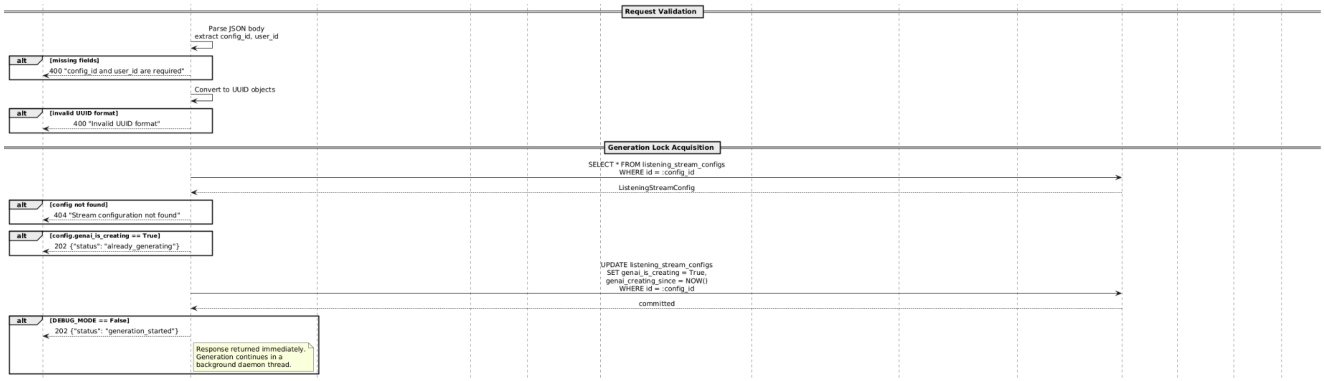


Fig. 15. Second part of the GenAI sequence diagram

The request format is validated and then genai_is_creating is set to True. The scheduling mechanism checks this flag in order to avoid repeating generation requests for the same configuration while a generation is already in progress. Also, genai_creating_since is set to now, which allows us to reset the genai_is_creating flag if it has been True for an unreasonably long time, indicating a crash / failure in the generation process.

If DEBUG_MODE == False, the request returns immediately with status 202 - indicating generation is ongoing in a background daemon thread. If DEBUG_MODE == True, in order to enable line-by-line execution with a debugger, the generation is done in a blocking manner.

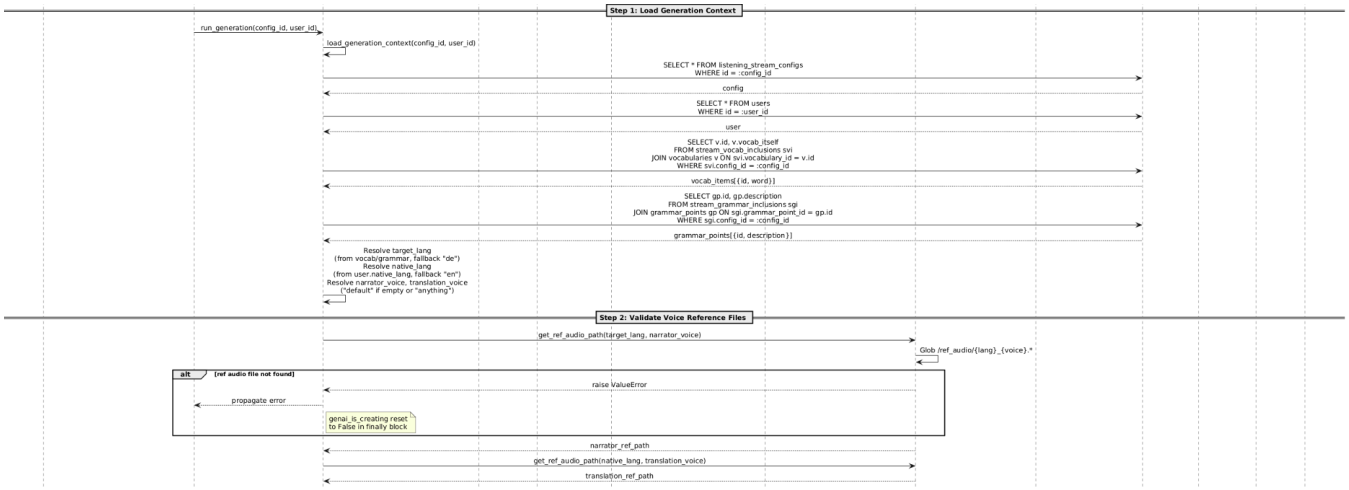


Fig. 16. Third part of the GenAI sequence diagram

The “generation context” which includes the vocabulary, grammar structures, the target language and the native language of the user, description of the content and vocabulary are loaded. Then, the reference audio files for the given languages required for Chatterbox TTS are validated.

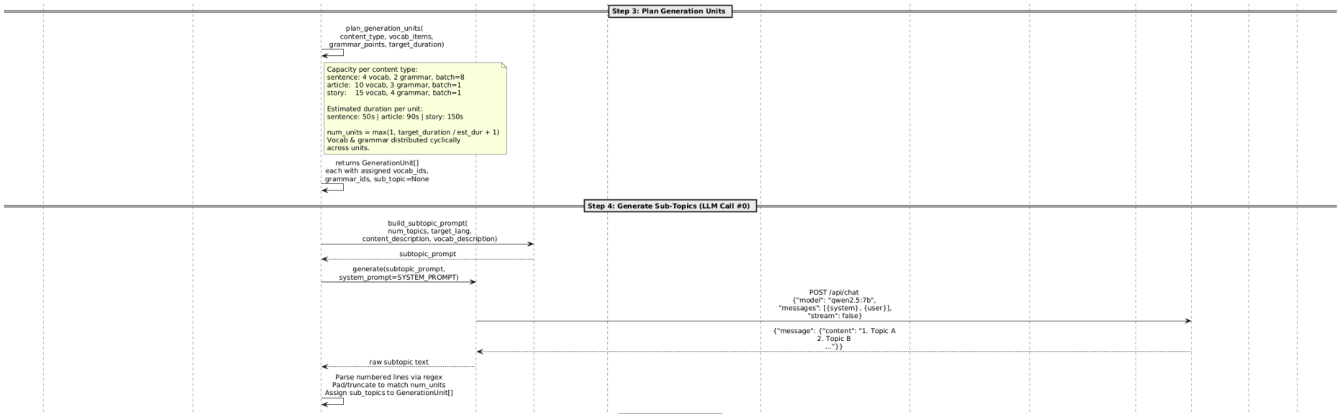


Fig. 17. Fourth part of the GenAI sequence diagram

The GenAI module plans “generation units” before diving into generating the content itself. Every /create call tries to create and narrate 15 minutes of content for a given configuration. First, a configuration may be a story, article, or individual sentences. The total number of units (of a story, article or sentence) required to fit 15 minutes is determined.

There is a certain number of grammar points and vocabulary a certain type of content can take. For example, it would be unreasonable for a single sentence to demonstrate 15 grammar structures. A 150 second story demonstrating 200 vocabulary would likewise be unreasonable. Therefore, the limits of each content type is set as a constant, to enable **correctness**. Then the grammar points and vocabulary of the configuration is fairly distributed among the generation units. Then, a different subtopic is created via an LLM call for each single unit. The prompt for subtopic generation emphasizes maximal variety, for purposes of **novelty** and **engagement**.

```
f"Generate exactly {num_topics} diverse scenario ideas for short {target_lang} language-learning
content.\n"
        f"{theme_line}\n"
        f"{vocab_line}\n"
        "Each scenario must be clearly different from all others.\n"
        f"Output ONLY the scenarios, one per line, numbered 1 through {num_topics}.\n"
        "Do not add any other text."
```

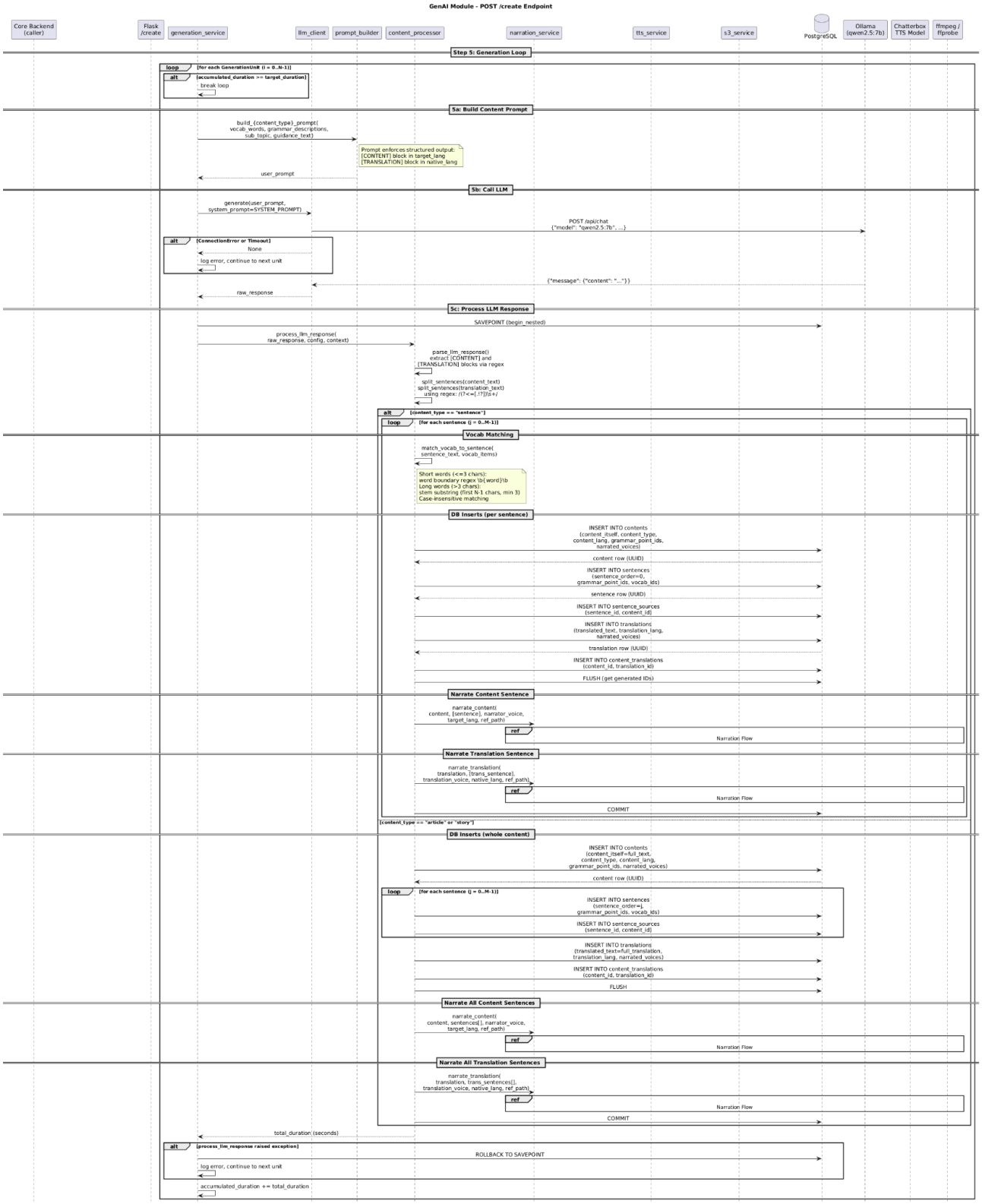


Fig. 18. Fifth part of the GenAI sequence diagram

The above part of the diagram can be viewed here.

After validation and planning generation units, the GenAI module actually creates the content. The content creation prompt, even though it changes slightly for each content type, has the following pattern:

```
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"
        "..."
```

After generating the contents, the translations are also generated. The necessary DB insertions follow. This includes inserting the contents themselves and also new rows at relation tables that connect which grammar / vocabulary is included in a certain content and also rows at relation tables that connect a content to its translation. An important design decision is that these DB manipulations are not committed to the database immediately. The reason is that when we are narrating the content, if a problem occurs (and persists after several retries), we do not want entries in our database without a narration and we want to rollback all the insertions we did. That is what the sequence diagram demonstrates.

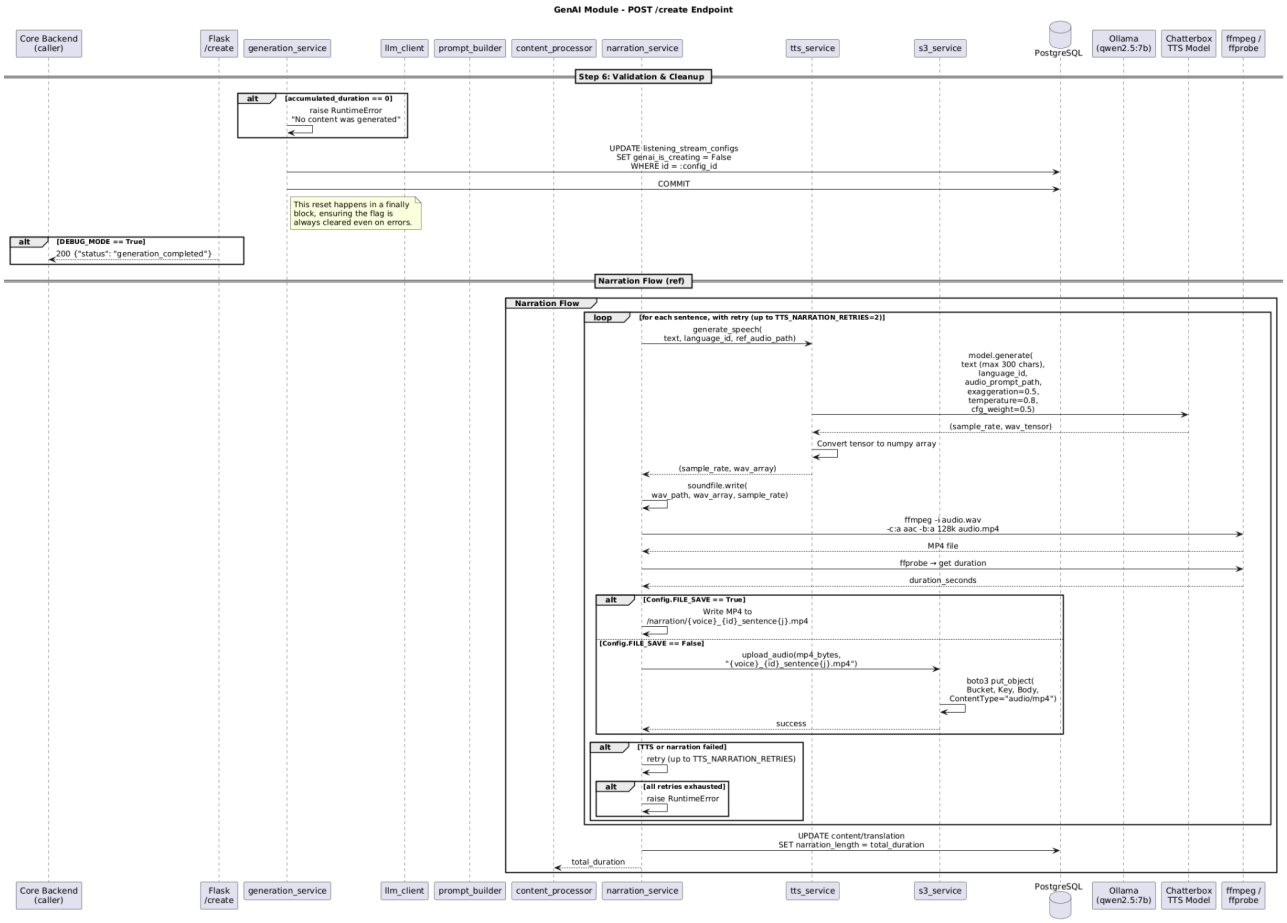


Fig. 19. Sixth part of the GenAI sequence diagram

The final part of the GenAI sequence diagram can be viewed here.

After the narration of contents and DB insertions are done, we validate the length of the narration created one last time and then set the genai_is_creating flag back to False. The GenAI module is capable of both using a local LLM and a cloud provider; it is also capable of both saving to a local file and saving to Cloudflare S2. The sequence diagram demonstrates the final step of saving the audio file to local disk (for debugging purposes) after the important step of converting the audio to AAC-LC inside an MP4 container for maximum mobile compatibility using ffmpeg.

4.2.3 Frontend

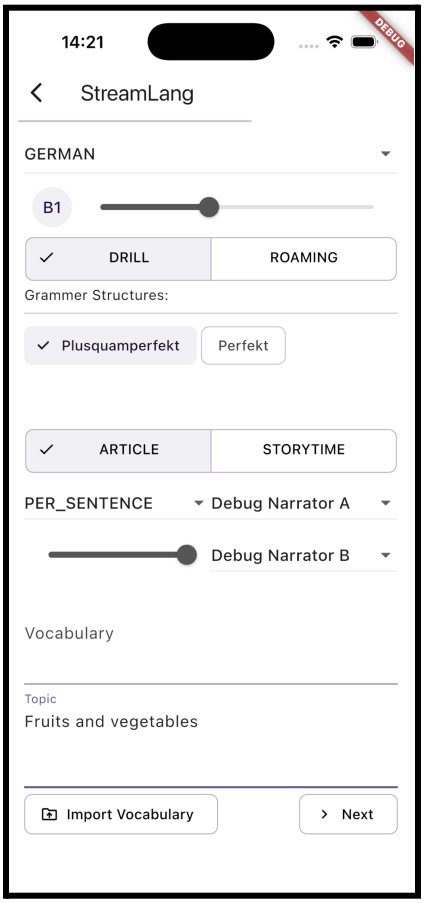
The frontend, specifically the mobile application made using a combination of the Flutter framework alongside some platform-native code. Our frontend follows the classic tried-and-tested registration and login flows with the following possible actions:

- 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.

I. Tech Stack: As previously mentioned, we are using the Flutter framework. However to accomplish our desired functionality, we are utilizing the following packages:
   A. HTTP: The official library from the Flutter team to have http communications
   B. Get: Get (or widely known as GetX) is a comprehensive library most known (and used here) for its state management and localization abstractions.
   C. 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, hence we use the video library, but disable requests for video when connecting.

II. Configuration Interface



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

1. If the user decided to change a previous setting, they would have to go back one-by-one, limiting usability,
2. Due to an 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.

Fig. 20. Stream configuration screen

4.2.4 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 content of which the amount of grammar points in that content passes a 5% threshold.
6. Tokenize sentences by vocabulary.
7. Check if vocabulary in a sentence matches the ones in the vocabulary table, if yes, add those vocabulary to the sentence.

While this is a high-level overview of the process, the architectural definition of the batch job is shown in a sequence diagram. This sequence diagram will be explored in several parts.

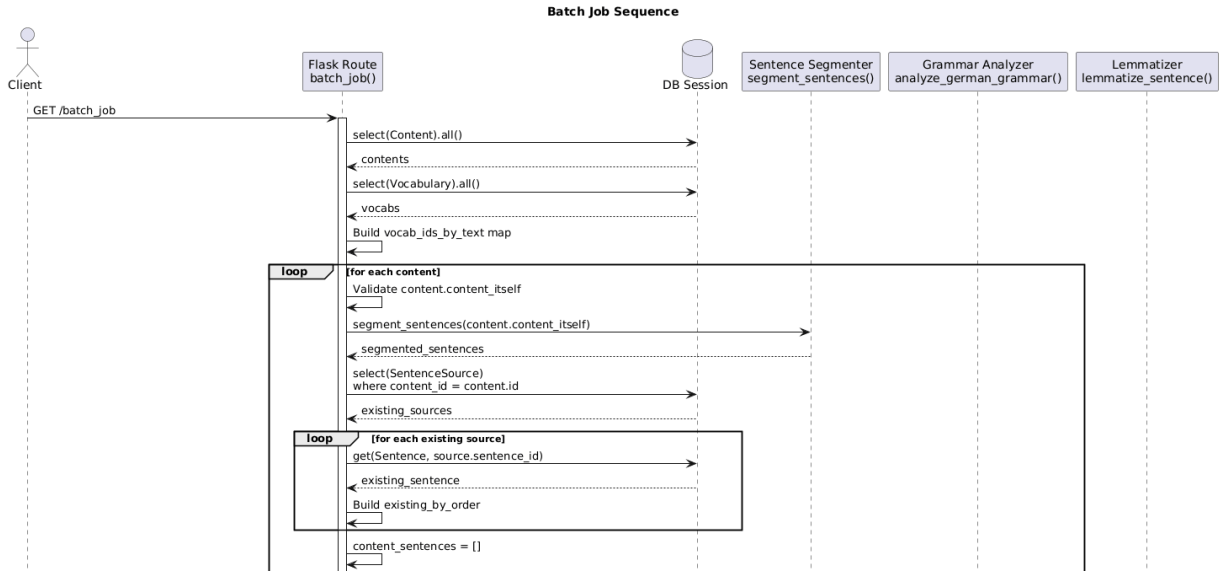


Fig. 21. First part of the Content Categorization Sequence Diagram.

We have our client, database, sentence segmenter, grammar classifier, and lemmatizer as different actors. Sentence segmenter segments a content into sentences, grammar classifier returns a list of grammar for a sentence, and lemmatizer returns a list of lemmatized words for a sentence. First of all, we fetch all entries in the content and vocabulary tables from the database. Then, we create a dictionary of vocab ids and vocab text for efficient lookup later. Next, we iterate through each content in a loop. Inside this loop, we first check if the content actually has the text field. If not, we raise a ValueError. Next, we segment the content into sentences using the sentence segmenter submodule. Next, we fetch existing sentence_sources for the content, if there are any. In another loop, we loop through existing sentences for that content, validating their fields, and building a dictionary existing_by_order, mapping positional order of sentences to their database records.

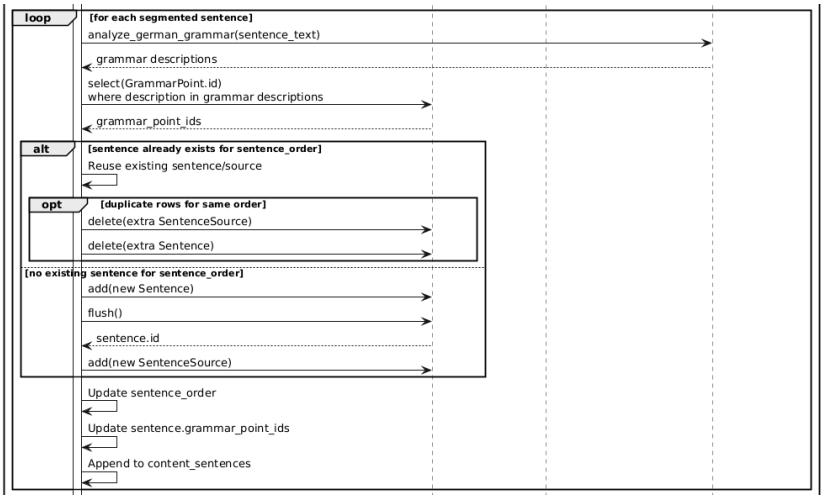


Fig. 22. Second part of the Content Categorization Sequence Diagram.

After this, in another loop, we iterate over the segmented sentences, classify them by their grammar points, and extract grammar_point_ids corresponding to obtained grammar points. Then, we check if a sentence already exists in the current order position. If yes, we reuse the first matching record and delete duplicate records. If not, we create new Sentence and SentenceSource objects, linking them to the content, and add them to the database. Next, we assign the identified grammar_point_ids to the sentence and track them in the content_sentences list.

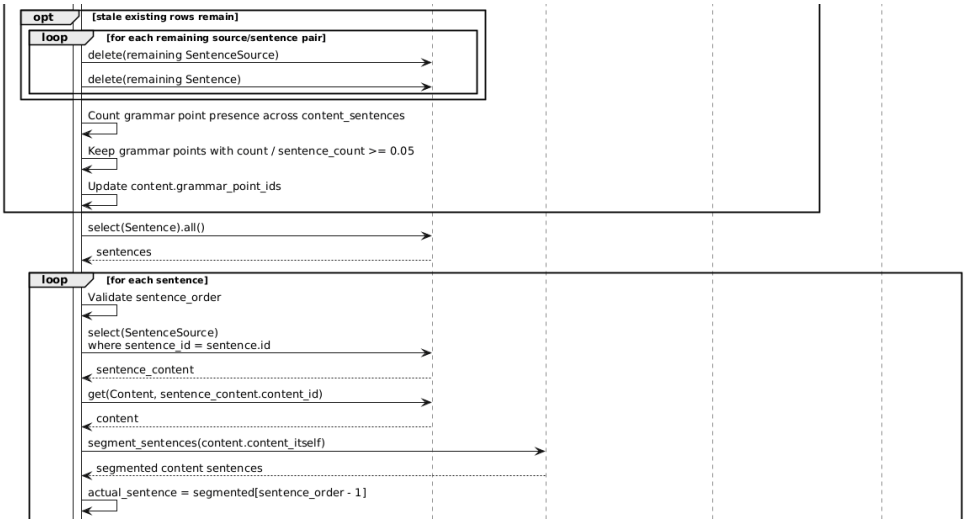


Fig. 23. Third part of the Content Categorization Sequence Diagram.

Next, we do a cleanup in the database for stale sentence records. Afterwards, we count grammar points for a content across sentences, and keep the ones that pass the 5% threshold. Finally, we update the grammar_point_ids field of the content with the grammar points that passed the threshold. Next, we move on to the vocabulary matching part. First, we fetch sentences in the database and validate them. Next, we fetch sentence_sources to find the origin

content of the sentences. Then, we segment the content to obtain text of the sentences in the content.

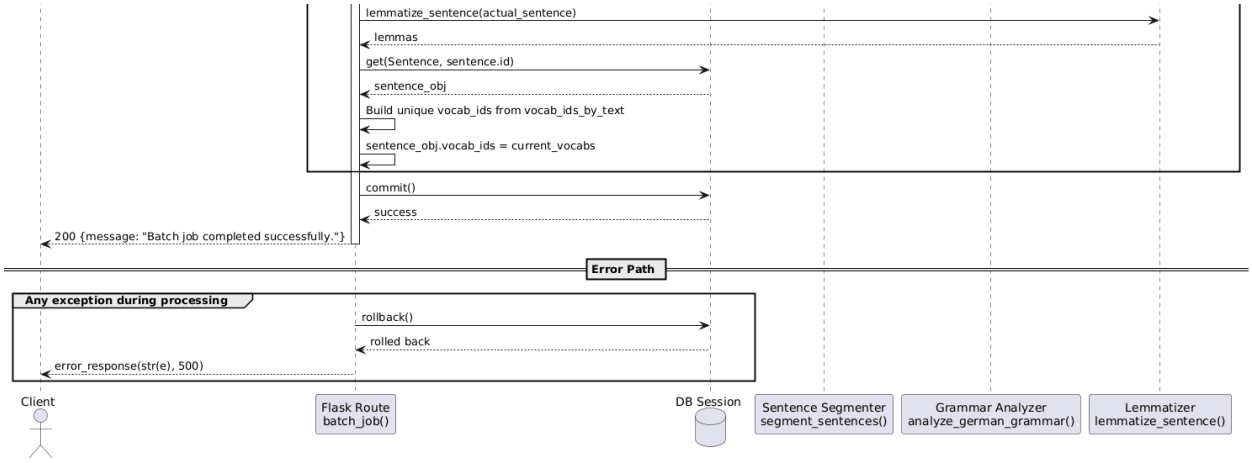


Fig. 24. Fourth part of the Content Categorization Sequence Diagram.

Finally, we lemmatize the words in the sentences to obtain lemmas. Next, we match existing vocabulary items with the lemmatized words. If there is a match, we update the vocab_ids fields of sentences with the matching vocabulary we identified. Finally, we commit to the database and return with a success message. In the case of an error, we rollback the changes and return with an error message.

4.2.5 Streaming

The streaming module is a two server setup reverse proxied into a single address. The first server is the Flask application responsible for internal and external API calls.

The Flask application currently only has one internal endpoint /internal/create-stream. After being called, it first checks if the data format is valid, which should be a list of content IDs. If not, the response is a 400 Bad Request with an error message. After validating that the required data exists, it then checks every content ID to see if there is a corresponding audio file for it using the Audio Location Service. If no audio exists for any given ID, the call fails and returns a 404 Not Found response. Finally, after iterating through every ID, if there was no error, it adds a stream to the Stream Store Service with the file content IDs for that stream, and responds with 200 OK along with the stream ID in the body.

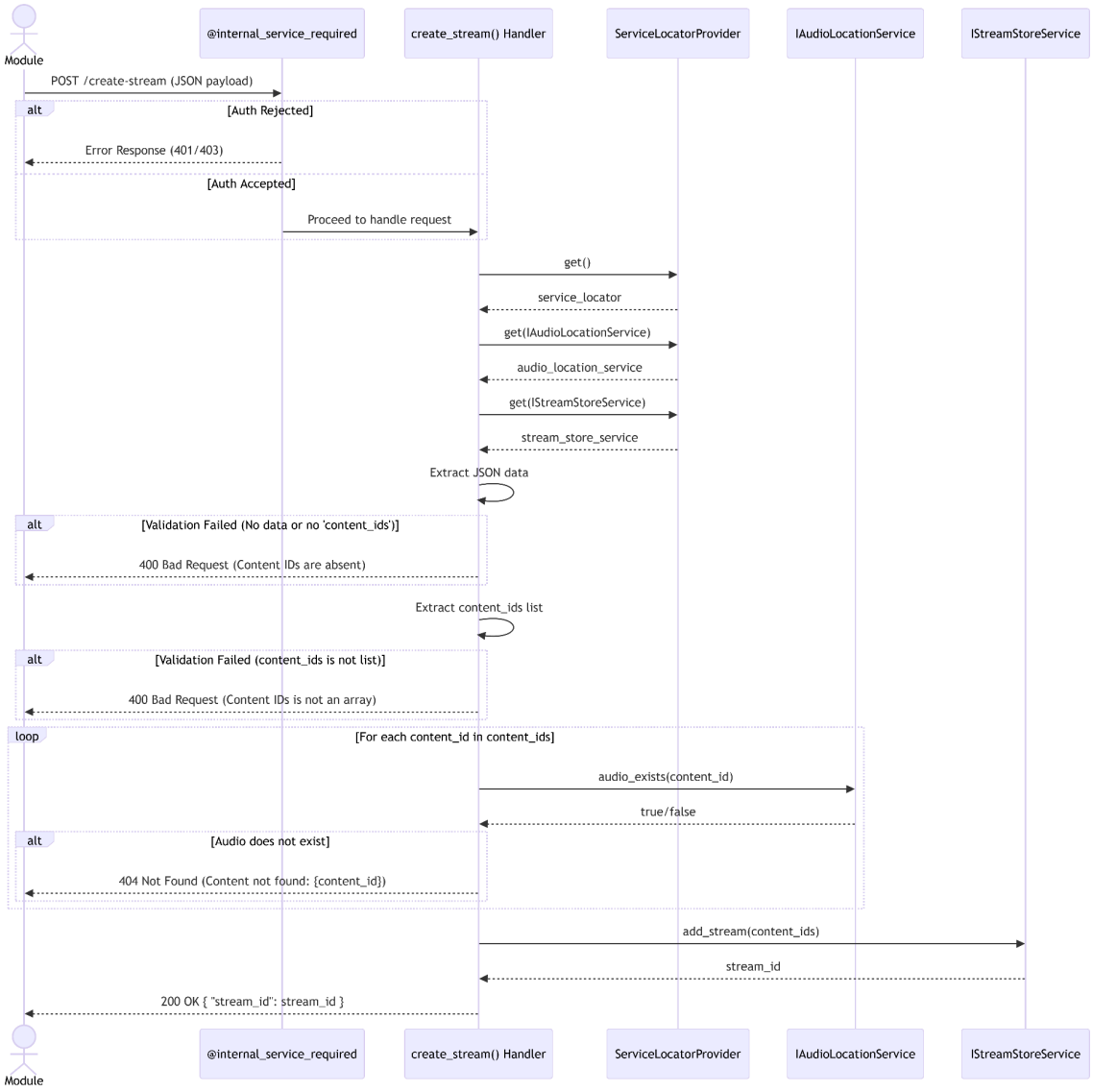


Fig. 25. Sequence Diagram for /internal/create-stream

The second server is an RTSP server using the GStreamer library. GStreamer is an established open source multimedia framework [17] that handles the actual audio streaming work, like loading the file, decoding it and sending 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 in 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 is 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.

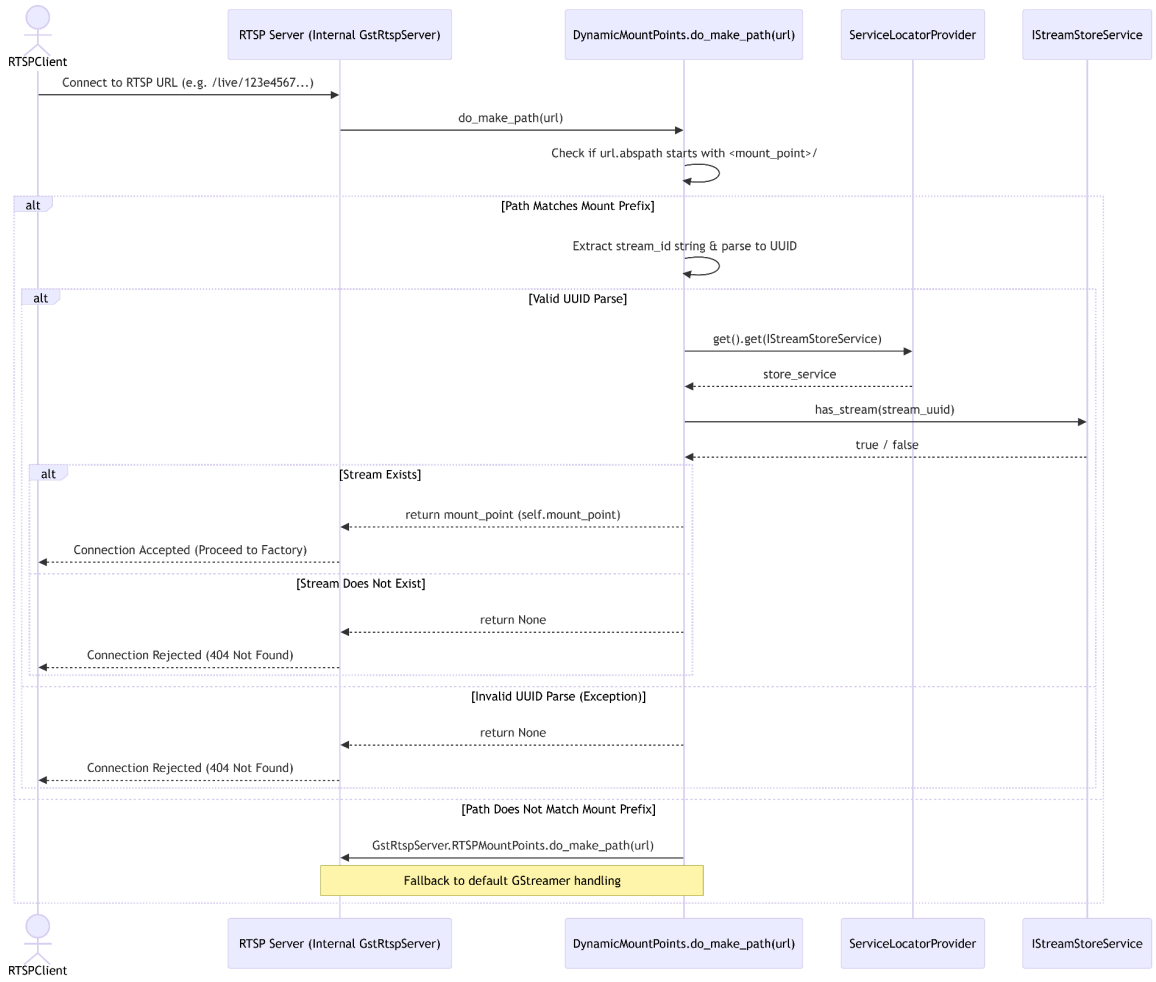


Fig. 26. Sequence Diagram for DynamicMountPoints

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.

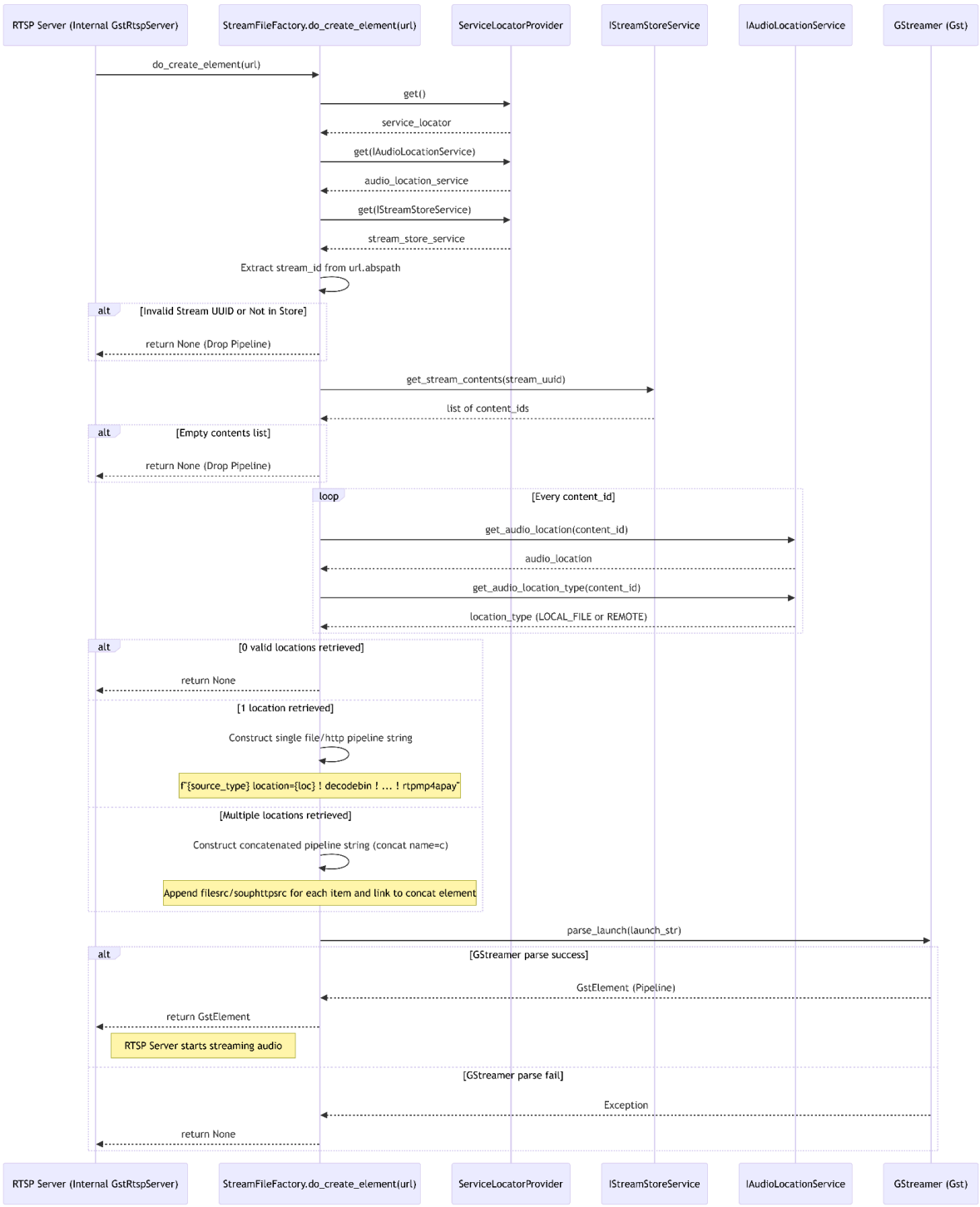


Fig. 27. Sequence Diagram for StreamFileFactory

5. Test Cases

5.1 Procedures for Software Test Engineers

You will notice that this section is short. That is because there is a single core use case of the application: creating a listening stream based on a user configuration. StreamLang is not designed around having a plethora of use-cases; it is designed around accomplishing one very difficult use case excellently. **By its nature**, StreamLang has only a few test cases.

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 valid unique name, email, and payment details.
    - 4. Verify 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

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 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 invalid email format and prevents registration.
- **Priority/Severity:** Major

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 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

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

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

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

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

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

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

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

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 stories.
- **Priority/Severity:** Major

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

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

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

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

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

5.2 Non-Functional Tests

5.2.1 Usability tests

- Session Caching: Verify that the necessary tokens are being successfully cached on the user's device to make authentication easier. Restart the application and confirm the user is automatically logged in.

5.2.2 Performance Tests

- Non-Generative Response Time: Send several requests to endpoints that do not engage the GenAI module, and ensure they respond within one second.

5.2.3 Reliability Tests

- Module Isolation: Simulate a crash or downtime in an independent subsystem. Verify this does not lead to a totally unusable application.
- Database Rollback: Do an invalid operation to evaluate the database protections in place. Confirm PostgreSQL's checks work properly, and automatic roll back is functional in case of data corruption.

5.2.4 Security Tests

- Internal Endpoint Security: Attempt to access internal backend APIs from an unauthorized origin. Verify that external sources cannot access such endpoints.

5.2.5 Maintainability Tests

- Logging Verification: Ensure that all the subsystems log accurately, and their logs are being saved properly. A successful result would have accurate timestamping and data formatting, with traceable logs.

6. Consideration of Various Factors in Engineering Design

6.1 Constraints

StreamLang is affected by some non-technical factors. Those factors, together with their level of effect (ranging from 0 to 10, 0 being non-affecting and 10 being maximum-affecting), and our descriptions of how we determine those levels are shown in the table below.

Table I
Non-technical Factors

Factor	**Level of Effect**	**Reason**
Public Health	2	StreamLang, as a mobile application, carries the same health risks as any other mobile app. Those include hand injuries, vision impairments, etc. However, unlike many other mobile apps, StreamLang carries no addictive factor. Longer usage times are not rewarded in any way, and as a learning app, it can't be described as addictive. It also doesn't induce stress through examinations or competitiveness.
Safety	2	StreamLang is a language learning app that streams content scraped from the internet. Theoretically, it is possible for the scraped content to be biased, harmful, or misinformation. However, in our subsystems, such content is audited by multiple LLMs. Thus, the risk is low. Other than this, it is not possible to misuse StreamLang to cause any harm.
Security	1	StreamLang holds user accounts and progress information in databases. Thus, the breach of this data carries a security constraint. Against this, we make use of known best practices for privacy and security. The only way a security issue may occur is if a third-party cloud provider we use causes it. As the app requires high compute power and high storage, it is not economically viable for us to handle those ourselves; we are taking a small risk by using cloud providers.
Welfare	8	StreamLang is a language learning platform. It provides a cheap, efficient, customized, and accessible way of improving listening skills. It applies to any age group with any previous knowledge, thus it affects the language knowledge levels of people very positively.
Environment	4	As previously mentioned, StreamLang requires high compute power. Especially for content scraping, content generation, and text-to-speech, we are using heavy ML models. As a result, this

		computing power requirement has some negative environmental impact.
Economic	4	Again, due to high computing power requirements and cloud computing usage, our running costs affect the economic factor. Furthermore, one may argue that the usage of StreamLang affects the need for language courses, but we see our app as a supplement to traditional language schools rather than a replacement.

We also have some technical constraints. With StreamLang, we are aiming at the average language learner. With the current market expectations, we need a responsive cross-platform app with good UX design. We also need extensive search capabilities for the backend, while also staying efficient. But alongside efficiency, we also need compatibility with AI systems and ease of development.

As described in the table, for each of the non-technical constraints, we are taking several measures. To reduce the negative health effects, we are making sure we are not promoting excessive usage in any way. Users do not get rewarded as they use the app. Furthermore, they may be prompted to wait and give a break if content generation is taking longer than usual.

To reduce the safety risk, we are employing several small LLM agents to audit streamed content. We are also choosing our scraping sources carefully, among respected news sources, etc. Furthermore, we allow users to give feedback to streams so any unsafe content can be reported and manually removed.

To reduce the security risk, we are following best encryption/hashing/firewall practices. We are also making sure that we do not store excessive data, and users are allowed to have their data deleted via special requests.

To increase StreamLang's positive effect on public welfare, we are making sure the language learning process is as effective as possible by allowing users to configure the streams to their needs. Also, again during scraping, we are prioritizing texts that would accelerate learning (i.e, texts from language learning books, news sources that use professional language, etc.).

Lastly, to reduce the negative environmental and economic effects, we are again following best practices to make optimizations on our algorithms. We prefer smaller LLMs and ML models when sufficient, so we don't waste unnecessary compute power. Unlike many AI-based solutions present in the industry, we are abstaining from solving every single problem using the largest, most expensive to run LLMs. We are only using high computing power when we are required to do so.

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, offers 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 brings a significant advantage for deployment. Postgres offers fast relational storage, meaning it allows us to store all user data in a single database with no performance limitations. CloudFlare R2 offers 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 [13].

6.2 Standards

To ensure consistency and maintainability, the following standards are applied:

- 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) use the markdown file format.

7. Teamwork Details

7.1 Contributing and Functioning Effectively on the Team

Mehmet Emin Avşar
I implemented the content selection mechanism of the core backend and also the GenAI module. In order to contribute effectively, I always tested my implementations before creating a PR. I communicated frequently with my teammates so that we were on the same page about how the implementation is going.

Ruşen Ali Yılmaz
I designed and developed the mobile application part of the StreamLang frontend. For clear and effective communication, I created the Frontend API specification, which effectively and descriptively communicates the API specification to my other teammates. During the design and development phases, I contributed to the initial architectural design of the project.

Uygar Bilgin
I designed and implemented the Content Categorization Module of StreamLang. During this process, I frequently communicated with my teammates to decide what other modules required

from the Content Categorization Module, and vice versa. This allowed seamless integration of the part I implemented and parts they implemented in the project.

Can Tücer
I worked on the Core Backend, which includes API routes for user authentication management, listening configuration management, etc. I also developed the Content Scraping Module, which scrapes our decided sources for usable listening contents, parses texts, checks for user safety, and makes the required updates so the text is ready for TTS. In parallel, I worked on the architecture of our system, set up our Debian backend server, managed PostgreSQL server, R2 Object Storage, firewalls, and public domains. Lastly, I built our project website.

Göktuğ Ozan Demirtaş
I was mainly responsible for the Streaming Module, and implemented all required functionality. The SRS tracking implementation was also done by me along with some refactoring to make code more maintainable. I also made quality of life changes in other parts of the codebase whenever my eyes caught something that can be improved upon (like streamlining internal API requests).

7.2 Helping Creating a Collaborative and Inclusive Environment

Mehmet Emin Avşar
I called meetings often to foster a spirit of collaboration. When distributing work, I tried to be fair and reasonable. I distributed the work evenly across my teammates to not exclude anyone.

Ruşen Ali Yılmaz
Even though I work solo on the mobile application side of things, I always sought out feedback and insights into design choices, and responded to criticism by improving the design and the app wherever said criticism applied.

Uygar Bilgin
During my implementation process, I frequently sought feedback from my peers, with pull request comments on Github being one example of this. We collaborated on different tasks we required from each other during the implementation, for example designing a helper function for a module one of my friends was implementing, which required using functions in the Content Categorization Module.

Can Tücer
I worked very closely with my teammates in every implementation phase. During the Core Backend development, we had meetings to decide on the needs of the frontend, agreed on a common API documentation, and worked on that. Again via meetings, I made sure the Content Scraping module will address the needs of other modules, especially Emin’s generative AI

module, since he will be using the scraped content directly. Lastly, before deciding on our Cloud service stack, we decided on our budget and required hardware all together.

Göktuğ Ozan Demirtaş

When developing the Streaming Module, I made sure to finalize as much as possible with the rest of the team on things like the streaming protocol before actually getting work done. If this was not possible, I communicated to anyone who may need to use the Streaming module on changes or left ample documentation to facilitate easy use and/or testing of functionality. I was always available for feedback when an internal API related or functionality related change was necessary.

7.3 Taking Lead Role and Sharing Leadership on the Team

Mehmet Emin Avşar

I created the majority of the issues on our Github page with detailed descriptions of what needs to be implemented, and how the different modules are going to connect with each other in our long-term vision. Instead of designing everything myself, I shared the leadership role by assigning people to design their own parts after explicating what is expected from them: for example Göktuğ single-handedly designed the SRS functionality and Uygar designed the content categorization module. Throughout these designs, we had frequent conversations over PRs and the group chat to stay in sync.

Ruşen Ali Yılmaz

I took the lead role for the mobile application. I took initiative to design and develop the frontend, and by designing it early, it enabled me to write a clean API specification that the frontend expected, well before the backend development had begun. I've also contributed heavily to the architectural decisions, acting as a lead on the matter.

Uygar Bilgin

I worked as the main developer of the Content Categorization Module, making design choices about what the module was going to provide and what function it was going to fulfill. During the planning stages, we all worked as a group and gave input about the direction and architectural design choices of our application.

Can Tücer

I took the lead in Core Backend and Content Scraping modules. I made sure they run as optimized as possible. I also followed the GitHub issues and pull requests relevant to those modules, advised my teammates to make some changes, and followed the progress in other modules to avoid any mismatch.

Göktuğ Ozan Demirtaş

I took charge in developing the Streaming Module and making all its design choices. I researched and settled on what I believe to be an adequate streaming stack and server architecture for the module. I also made what I believed to be necessary architectural refactorings when implementing features in other parts of the codebase, like when implementing SRS tracking.

8. References

[1] L. Loschky, “Comprehensible Input and Second Language Acquisition,” *Studies in Second Language Acquisition*, vol. 16, no. 03, p. 303, Sep. 1994, doi: https://doi.org/10.1017/s0272263100013103.

[2] “Master any language like a polyglot,” *Taalhammer*, Jan. 18, 2026. https://www.taalhammer.com/ (accessed Mar. 11, 2026).

[3] I. Auth0, “isAuthenticated$ return false,” *Auth0 Community*, Oct. 05, 2023. https://community.auth0.com/t/isauthenticated-return-false/117929 (accessed Mar. 11, 2026).

[4] I. Auth0, “How to authenticate two applications (Wordpress and Angular) with one login,” *Auth0 Community*, Feb. 05, 2019. https://community.auth0.com/t/how-to-authenticate-two-applications-wordpress-and-angular-with-one-login/20986 (accessed Mar. 11, 2026).

[5] Mateusz Wiącek, “Spaced Repetition System (SRS) and the forgetting curve in language learning – history, theory and our practice,” *Taalhammer*, Mar. 2024. https://www.taalhammer.com/spaced-repetition-in-language-learning/#h-atom-new-algorithm-even-better-language-learning (accessed Mar. 11, 2026).

[6] Till Building Products, “Building a Startup in One Week Using ShipFast,” *YouTube*, Jun. 28, 2024. https://www.youtube.com/watch?v=E-rQHapyL78 (accessed Mar. 11, 2026).

[7] Lenguia, “Lenguia | Learn Languages with Comprehensible Input,” *Lenguia.com*, 2026. https://www.lenguia.com/ (accessed Mar. 11, 2026).

[8] “I built an app because I fell in love with a girl,” *Indie Hackers*, 2024. https://www.indiehackers.com/product/lenguia-language-learning-through-input/i-built-an-app-because-i-fell-in-love-with-a-girl--O8s1iYq-9fA9WZQJk20 (accessed Mar. 11, 2026).

[9] “Beelinguapp | Learn languages with music & audiobooks,” *beelinguapp.com*. https://beelinguapp.com/

[10] Docker Inc., “Accelerated Container Application Development,” Docker, https://www.docker.com/ (accessed Mar. 12, 2026).

[11] Docker Inc., “Docker Compose,” Docker Documentation, https://docs.docker.com/compose/ (accessed Mar. 12, 2026).

[12] B. Hagan, “Comparing native Postgres, Elasticsearch, and pg_search for full-text search,” Neon, https://neon.com/blog/postgres-full-text-search-vs-elasticsearch (accessed Mar. 4, 2026).

[13] “Cloudflare R2 · Cloudflare R2 docs,” *Cloudflare Docs*, Apr. 05, 2024. https://developers.cloudflare.com/r2/

[14] Expertium, “A technical explanation of fsrs,” Expertium’s Blog, https://expertium.github.io/Algorithm.html (accessed Mar. 12, 2026).

[15] Open-Spaced-Repetition, J. Ye, J. Su, and Y. Cao, “The Algorithm open-spaced-repetition/fsrs4anki,” GitHub, https://github.com/open-spaced-repetition/fsrs4anki/wiki/The-Algorithm#fsrs-6 (accessed Mar. 12, 2026).

[16] Open-Spaced-Repetition, J. Ye, J. Su, and Y. Cao, “ABC of SRS open-spaced-repetition/fsrs4anki,” GitHub, https://github.com/open-spaced-repetition/fsrs4anki/wiki/ABC-of-FSRS (accessed Mar. 12, 2026).

[17] GStreamer Team, “Home,” GStreamer, https://gstreamer.freedesktop.org/ (accessed Mar. 12, 2026).