Design Report

Medical ASR for Radiology

Group 14

Reinder Grondsma - s3161218

Art de Heer - s3201058

Tim Hendriks - s3206068

Jibbe Konniger - s3176568

Thomas Scholtens - s3016668

Supervisor

Dr. ir R. Ordelman

UNIVERSITY
OF TWENTE.

Abstract

This report details the design and development of an open-source, on-premises Automatic Speech Recognition (ASR) prototype, tailored towards radiologic dictations at Hospital Group Twente (ZGT). The project was initiated to address the high license costs of the current proprietary system and to demonstrate that open-source models can achieve clinical quality while ensuring local data privacy and GDPR compliance.

Through systematic benchmarking of state-of-the-art models, including OpenAI's Whisper and NVIDIA's Canary, NVIDIA's Parakeet-TDT-0.6b was identified as the optimal foundational architecture for its superior baseline performance on Dutch medical jargon and exceptional inference speed. The system was developed using a centralized, modular pipeline that incorporates specialized data normalization, Voice Activity Detection (VAD)-based dynamic chunking for real-time transcription, and a dedicated post-processing layer to handle Dutch punctuation commands and clinical abbreviations.

Fine-tuning the model on 13 hours of specialized radiology dictations resulted in a Word Error Rate of 15.5% and a Character Error Rate of 7.0%, which represents a significant improvement over zero-shot baselines. The final prototype features a web-based interface that supports real-time transcription, manual correction modes, and a transparent feedback loop for continuous model improvement. This work establishes a scalable framework for implementing similar ASR systems across other medical departments while eliminating dependence on third-party cloud services.

Table of Contents

Table of Contents 2
Chapter 1 - Introduction 5
1.1 Background 5
1.2 Source Code and Glossary 5
Chapter 2 - Requirements specification 6
2.1 Introduction 6
2.2 Stakeholder analysis 8
2.3 User stories 8
2.4 Use cases 11
2.5 Functional Requirements 15
2.6 Non-functional requirements 17
2.7 Constraints 19
2.8 Assumptions 19
2.9 Diagrams 20
Chapter 3 - Project Approach 23
3.1 Architecture and model choice 23
3.2 Data & Privacy 26
Chapter 4 - General Design Choices 29
4.1 Finding the base model 29
4.2 Web application decisions 31
Chapter 5 - Implementation 34
5.1 File Structure 34
5.2 Data ingestion and normalization workflow 36
5.3 Training strategy 39
5.4 Evaluation loop 40
5.5 Real-time Dictation Prototype 41
5.5.4 Further training 43
Chapter 6 - Testing 44
6.1 Testing plan 44
6.2 Test results 45
6.3 System performance and infrastructure 47
Chapter 7 - Discussion 49
7.1 Domain ZGT audio vs. synthetic generation 49
7.2 Scalability across departments 49
7.3 The impact of delayed data acquisition 50
7.4 Data Quality vs. Quantity 50
7.4 Latency vs. accuracy trade-off 51
7.5 Re-evaluating containerized deployment 51
7.6 Acceptability of the resulting WER 52

Chapter 8 - Conclusion **53**
8.1 Reflection on functional requirements 53
8.2 Reflection on non-functional requirements 55
Chapter 9 - Future Improvements **57**
9.1 Data 57
9.2 Second layer after ASR model 58
Appendices **59**
Appendix A - Source Code and Glossary 59
Appendix B - Traceability Matrix 62
Appendix C - Diagrams 64
Appendix D - File Structure 65
Appendix E - Test results **66**
References **68**

Chapter 1 - Introduction

1.1 Background

ZGT is starting development of an open-source, on-premises speech recognition software for radiologic dictation. The current solution (G2 Speech / ATLAS, with integration with HiX) works, but the license costs are so high that it is infeasible to make speech recognition widely available within ZGT. Therefore, a prototype was built to show that this is possible with open-source models that are safe, local, and of acceptable quality. This prototype is the first step toward more widespread implementation of generative AI (Artificial Intelligence) in the hospital.

This team was challenged to design and build a working prototype of a Speech-to-Text (STT) or Automated Speech Recognition (ASR) system that radiologists can use to safely and locally convert their reports into text. First off, the current dictation process had to be analyzed and mapped. After that, training data was obtained to fine-tune and train an open-source model. The system is required to compute and transcribe a dictate of at least one minute for further processing. The end product needed to be simple, be scalable, and comply with the relevant privacy laws. Also, the end product needed to be extendable to other medical fields.

1.2 Source Code and Glossary

Appendix A contains a glossary and instructions for requesting the source code for this project.

Chapter 2 - Requirements specification

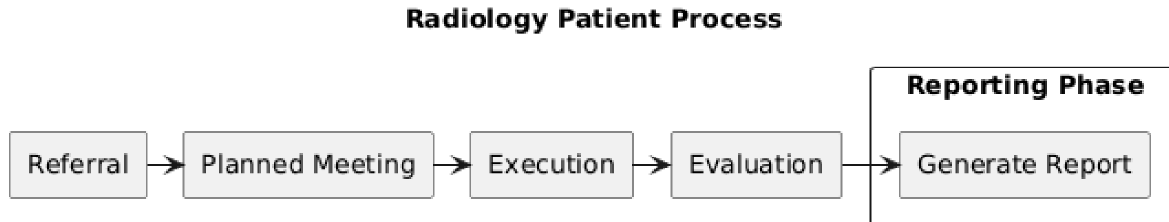
2.1 Introduction

2.1.1 Purpose

This chapter presents the requirements analysis for the Medical ASR for Radiology project, in collaboration with Hospital Group Twente (ZGT). First, a stakeholder analysis was conducted for everyone involved in the system. Then, based on the stakeholder analysis, user stories, and use cases are created. Based on those user stories and use cases, functional and non-functional requirements were made. Then, some constraints were noted, and assumptions were made to ensure the project would succeed. Finally, some diagrams are used to visualize how the system operates, both software- and hardware-wise. At the bottom of this chapter, the traceability matrix is presented, which connects all the different components of this report.

2.1.2 Scope

The following diagram illustrates the clinical workflow of a patient within the ZGT radiology department, tracking the progression from the initial medical request to final documentation.



Figuere 1: A visual representation of the process that radiology patients go through.

The scope of this project is specifically centered on the Reporting Phase. During this stage, the radiologist evaluates the previously captured imaging data and generates a formal medical report. The primary objective is to replace the current Automatic Speech Recognition (ASR) system that assists in this documentation process by converting the radiologist's spoken speech into written text in real time.

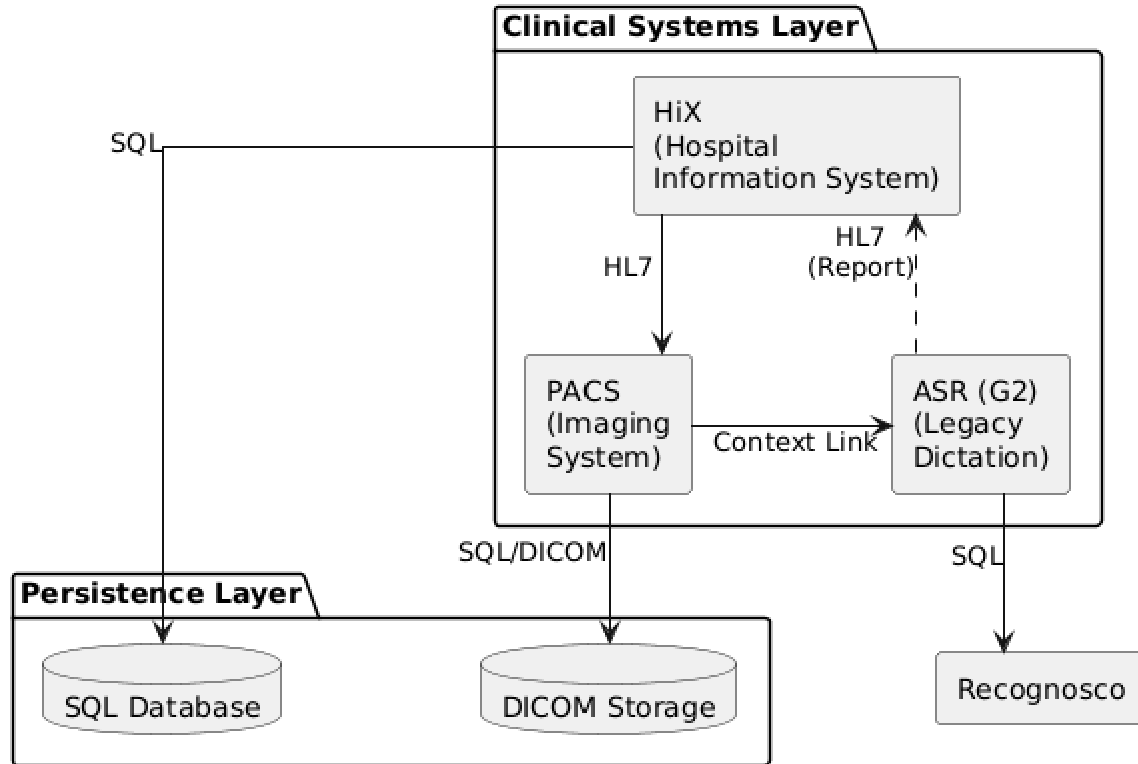


Figure 1: A visual representation of the software infrastructure used by ZGT.

The current radiology infrastructure at ZGT follows a modular architecture where HiX (Hospital Information System) initiates the workflow by sending examination orders to the PACS (Imaging System) via HL7 (the protocol). The PACS serves as the primary interface for radiologists to view images retrieved from DICOM Storage. During the reporting stage, a Context Link synchronizes the ASR (G2) system with the active patient study. Currently, this legacy ASR relies on a proprietary "black box" engine called Recognosco to process speech-to-text via SQL.

The objective is to transition from the proprietary G2 and Recognosco infrastructure toward a modular, open-source solution hosted locally. This shift aims to eliminate dependency on third-party licensing while ensuring full data sovereignty within ZGT's own environment. The prototype sits directly between the PACS and the final report output, converting spoken Dutch medical dictations into formatted text in real time using a speech-to-text model. By processing audio locally on ZGT infrastructure, the system eliminates the need for third-party SQL dependencies and high licensing costs. While the system is designed to eventually feed data back into HiX via HL7, this project focuses on the internal transcription pipeline, specifically audio capture, Voice Activity Detection (VAD), and model inference, rather than full hospital-wide integration.

2.2 Stakeholder analysis

To provide a comprehensive overview of the project's impact, a stakeholder analysis was conducted to identify the key parties involved and their specific requirements. The success of the ASR prototype depends on balancing the clinical needs of medical staff with the technical and ethical constraints of the hospital environment.

The following three stakeholders have been identified:

- Radiologists (Primary Users): As the main users, they are interested in the system's performance. The prototype must deliver high-accuracy, real-time dictation with low latency to ensure that the reporting phase is efficient and does not disrupt the diagnostic workflow.
- Maintainers (Technical Operators): They primarily focus on the ease of deployment in ZGT's local environment, the system's scalability, and the long-term stability of the open-source pipeline.
- Patients (Data Subjects): The data patients are not direct users of the software; however, their data is being processed in the system. Their interests are centered on the uncompromising protection of their sensitive medical data (GDPR compliance) and on the clinical accuracy of the final report, which directly impacts their diagnosis and safety.

Stakeholder	**Role**	**Interest**
Radiologist	Primary user	High-accuracy transcription; real-time performance; low-latency workflow integration.
Maintainer	Technical operator	On-premise deployability; system scalability; long-term open-source stability.
Patient	Data Subject and Safety Beneficiary	GDPR compliance (privacy); data sovereignty; high accuracy for patient safety.

Table 1: Stakeholder analysis summary.

2.3 User stories

Based on the stakeholder analysis, user stories were formulated to capture the system's functional goals from the perspective of its main actors.

2.3.1 Actors

The final system will have the following actors:

- Maintainer from the ZGT IT department.
- Radiologist from ZGT. This is the user of the system.

2.3.2 Example workflow

This section contains an example of each actor's workflow to provide context for the user stories that follow.

Maintainer Workflow

The maintainer deploys the ASR system as a containerized application on the local ZGT infrastructure, ensuring a reproducible and portable installation without manual configuration. After deployment, the environment is configured by setting GPU access, storage locations, logging, and runtime parameters, followed by validation checks to confirm correct system operation. The system is designed to integrate with the existing HiX-based workflow and replace the current G2 solution, while operating entirely on-premises to maintain data privacy and compliance. During operation, the maintainer monitors performance metrics such as Word Error Rate (WER), latency, Real Time Factor (RTF), and medical terminology accuracy to evaluate system performance. When improvements are required, new training data is used to retrain or fine-tune the model, after which updated model versions are deployed independently within the modular architecture while maintaining compatibility with hardware constraints such as a single NVIDIA H100 GPU (in the future).

Radiologist Workflow

The radiologist receives a referred patient, performs the required examinations and imaging procedures, and then opens the ASR dictation interface to begin reporting. Relevant patient information and imaging data are presented within the interface to support the reporting process. During dictation, spoken input is transcribed in real time, allowing continuous monitoring of the generated transcript. After completing the dictation, the radiologist reviews the transcript and corrects any recognition errors to ensure clinical accuracy. These corrections are stored as feedback to support future model improvement. They may also wish to insert additional content manually. For this purpose, the interface provides a toggle that allows the user to switch between a mode for editing existing text (correcting) and a mode for entering new content. Once the report is finalized, the corrected dictation is saved to the HiX system for further processing and distribution within the clinical workflow.

2.3.3 Stories

This section contains the user stories that capture the goals and expectations of the system's actors. The user stories describe desired functionality from the perspective of different actors (radiologists and system maintainers). They focus on the value the system

should provide, rather than implementation details. Each story follows the standard structure: “As a *[role]*, I want *[goal]*, so that *[benefit]*”, so that both intent and motivation are explicit. User stories from radiologists are named “US-R*x*”, where R stands for **R**adiologist, and *x* is the use case identifier number. Use cases from the maintainer are called “US-M*x*”, following similar logic.

Radiologist

- **US-R1:** “As a radiologist, I want to be able to dictate using the system in real time, so that I don't have to type everything.” (M)
- **US-R2:** “As a radiologist, I want to be able to see possible errors in the dictation so that I can correct mistakes with more ease and the system can learn from that.” (C)
- **US-R3:** “As a radiologist, I want to have the system transcribe twice as fast as I normally speak, so that it can even keep up with me when I am speaking very fast.” (M)
- **US-R4:** “As a radiologist, I want the system to accurately recognize radiology-specific terminology and abbreviations during dictation, so that I do not need to correct common medical terms manually.” (M)
- **US-R5:** “As a radiologist, I want to review and correct errors made by the system, so that the model can learn from my feedback and improve further dictations” (M)
- **US-R6:** “As a radiologist, I want to be able to speak in abbreviations while the system writes them out, so that long medical terms do not slow down the speed of my dictation.” (C/S)
- **US-R7:** “As a radiologist, I want to be able to dictate for more than 60 seconds at once, so that I can productively write reports.” (M)
- **US-R8:** “As a radiologist, I want to say specific voice commands (e.g., 'Punt', ' Nieuwe regel') during live dictation such that the system inserts correct punctuation automatically, so that I do not need to insert those by hand and the system does not need to infer them.” (M)
- **US-R9:** “As a radiologist, I want the transcript to remain visible during dictation, so that I can monitor mistakes immediately.” (S)
- **US-R10** “As a radiologist, I want to be able to use the system simultaneously with other users, so that I do not have to wait my turn to dictate.”
- **US-R11** “As a radiologist, I want to be able to manually add text in a mode specifically for manual addition of text, so that the system does not wrongly perceive my addition as a correction.”

Maintainer

- **US-M1:** “As a maintainer, I want to be able to deploy the system from a container, so that deployment is reproducible, portable, and independent of manual configuration.” (C)
- **US-M2:** “As a maintainer, I want to be able to expand the system to other medical fields, so that new fields can be supported without redesigning or retraining the entire system from the beginning.” (M)
- **US-M3:** “As a maintainer, I want the system to have a modular architecture, so that individual components can be updated independently.” (S)
- **US-M4:** “As a maintainer, I want to integrate the system into HiX, so that the system can replace the currently existing solution.” (W)

- **US-M5:** “As a maintainer, I want the system to run on ZGT’s infrastructure, so that no cloud services are required, and privacy is kept.” (M)
- **US-M6:** “As a maintainer, I want clear documentation, so that it is easily reproduced and implementable.” (S)
- **US-M7:** “As a maintainer, I want GPU resource usage to fit within a single H100, so that the system is employable on ZGT’s infrastructure.” (M)
- **US-M8:** “As a maintainer, I want Word Error Rate, latency, Real Time Factor, and medical terminology accuracy measured automatically so that the model performance can be evaluated objectively.” (C)

2.4 Use cases

This section describes the system's main use cases, detailing how different actors interact with it to achieve specific goals. Use cases translate intentions from user stories to interaction scenarios. Each use case specifies the primary actor, preconditions, the flow of events, and postconditions. Use cases where the radiologist is the primary actor are named “UC-R*x*”, where R stands for Radiologist, and *x* is the use case identifier number. For use cases where the maintainer is the primary actor, they are called “UC-M*x*”, following the same logic.

2.4.1 Radiologist Use Cases

UC-R1 - Start Dictation Session

Preconditions: The system is running, the radiologist has access to the web interface, and the microphone is functional.
Main Flow:

1. The radiologist opens the dictation interface.
2. The radiologist clicks "Start Dictation".
3. The system initializes audio capture and establishes a WebSocket connection.

Postconditions: An active dictation session is running and an audio streaming channel is established.

UC-R2 – Perform Live Dictation

Preconditions: A dictation session has started (UC-R1) and the ASR model is operational.
Main Flow:

1. The radiologist speaks into the microphone.
2. The frontend streams audio chunks to the backend.
3. The backend processes audio through the ASR model and returns a transcript to the UI in real time.

Postconditions: The transcript reflects spoken dictation and remains visible to the user.

UC-R3 - Insert Formatting via Voice Commands

Preconditions: The dictation session is active.
Main Flow:

1. The radiologist speaks a formatting command (e.g., "Punt" or "Nieuwe regel").
2. The system detects the command and inserts the corresponding punctuation or formatting.

Postconditions: The transcript contains correct punctuation and structural formatting.

UC-R4 - Handle Terminology and Abbreviations

Preconditions: The dictation session is active and the terminology database is available.
Main Flow:

1. The radiologist dictates radiology-specific terminology or abbreviations.
2. The system processes the input and recognizes domain-specific terms.

Postconditions: The transcript is updated with correctly recognized medical terminology.

UC-R5 - Review, Correct, and Add To Transcript

Preconditions: A transcript is available and the toggle for correction/addition mode is accessible.
Main Flow:

1. The radiologist reviews the transcript and edits incorrect text.
2. The radiologist toggles to "Addition" mode and manually inserts information.

Alternative Flow: The system highlights low-confidence words as an optional enhancement.
Postconditions: The corrected and extended transcript is stored locally in the session.

UC-R6 - Submit Corrections for Model Improvement

Preconditions: The transcript has been reviewed and corrected (UC-R5).
Main Flow:

1. The radiologist confirms or submits the final transcript.
2. The system stores the audio and corrected transcript as paired data with relevant metadata.

Postconditions: Feedback data is stored for potential future model retraining.

2.4.2 Maintainer Use Cases

UC-M1 - Deploy Model

Preconditions: GPU infrastructure (NVIDIA H100) and the Docker/NVIDIA Container Toolkit are available.
Main Flow:

1. The maintainer pulls or builds the ASR container image.
2. The maintainer configures GPU access and storage volumes.
3. The maintainer starts the service and performs a health check.

Postconditions: The ASR system is running with the model loaded in VRAM.

UC-M2 - Configure Environment

Preconditions: The system is deployed and configuration files are accessible.
Main Flow:

1. The maintainer defines environment variables such as model and logging paths.
2. The maintainer configures GPU memory allocation and restarts the service.

Postconditions: The system operates under the updated and documented configuration.

UC-M3 - Update Model

Preconditions: A new model version and validation datasets are available.
Main Flow:

1. The maintainer registers the new model artifact and restarts the service.
2. The maintainer runs validation benchmarks to confirm performance meets criteria.

Postconditions: The system runs the updated version, with the previous version available for rollback.

UC-M4 - Review System Performance Metrics

Preconditions: Logging is enabled and the system has processed inference data.
Main Flow:

1. The maintainer retrieves evaluation metrics (WER, latency, RTF).
2. The maintainer reviews accuracy reports and identifies performance drift.

Postconditions: System performance status is documented to decide if retraining is required.

UC-M5 - Retrain Model

Preconditions: Sufficient labeled data and GPU resources are available.
Main Flow:

1. The maintainer prepares the training dataset and triggers the fine-tuning job.
2. The maintainer evaluates the new version on a validation dataset.

Postconditions: A new model version with updated weights is produced and ready for promotion.

2.5 Functional Requirements

This section details the functional requirements derived from the Use Cases and User Stories. Requirements are divided into User Requirements (actions performed by the user) and System Requirements (functionalities provided by the system). Prioritization follows the MoSCoW method.

2.5.1 User Requirements

The User Requirements specify the actions the Radiologist and Maintainer must perform to achieve their goals. These requirements focus on user-facing functionality and are prioritized using the MoSCoW method to ensure critical clinical and technical needs are addressed.

Must (FR-UM-Rx / FR-UM-Mx)

The radiologist ***must*** be able to

1. **FR-UM-R1**: Access the dictation interface via a standard web browser. *(US-R1)*
2. **FR-UM-R2**: Start a session using a dedicated "Start Dictation" control. *(US-R1)*
3. **FR-UM-R3**: See the transcript appear in real time. *(US-R2, US-R5, US-R9)*
4. **FR-UM-R4**: Review and correct the text generated by the ASR. *(US-R2, US-R5)*
5. **FR-UM-R5**: Insert punctuation and formatting using voice commands. *(US-R8)*
6. **FR-UM-R6**: Toggle between a correction mode and an addition mode. *(US-R11)*

The maintainer ***must*** be able to:

7. **FR-UM-M1**: Start and verify the ASR service. *(US-M1)*
8. **FR-UM-M2**: Perform a basic validation test after deployment. *(US-M1)*
9. **FR-UM-M3**: Train the ASR to be expandable to other fields. *(US-M2)*
10. **FR-UM-M4**: Run the system on the ZGT infrastructure. *(US-M5)*

Should (FR-US-Rx / FR-US-Mx)

The radiologist ***should*** be able to:

1. **FR-US-R1**: Receive a visual confirmation that the dictation process has started. *(US-R1)*

The maintainer ***should*** be able to:

2. **FR-US-M1**: Add medical fields to the model selection. *(US-M2)*
3. **FR-US-M2**: Restart the system with updated configuration. *(US-M2)*
4. **FR-US-M3**: Register a new ASR model version. *(US-M3)*
5. **FR-US-M4**: Replace the currently deployed model. *(US-M3)*
6. **FR-US-M5**: Run validation benchmarks. *(US-M3)*
7. **FR-US-M6**: Have access to complete technical documentation. *(US-M6)*

Could (FR-UC-Rx / FR-UC-Mx)

The radiologist ***could*** be able to

8. **FR-UC-R1**: See words flagged by the model as potential errors. *(US-R2)*
9. **FR-UC-R2**: Use abbreviations when they please. *(US-R6)*

The maintainer ***could*** be able to

10. **FR-UC-M1:** Deploy the ASR system using containerized infrastructure. *(US-M1)*
11. **FR-UC-M2:** Make necessary adjustments such that implementation with other systems is possible. *(US-M4*

Won't-Have (FR-UW-Rx / FR-UW-Mx)

The radiologist ***will not*** be able to

1. **FR-UW-R1**: Automatically retrieve patient info from the HiX system when starting the system. *(US-M4)*

The maintainer ***will not*** be able to

2. **FR-UW-M1**: Integrate this system into HiX in its current form. *(US-M4)*

2.5.2 System-level requirements

The System Requirements define the internal functionalities, logic, and background processes the software must provide to support the user-level requirements. These focus on how the platform manages data processing, session states, and infrastructure constraints to ensure a reliable and accurate clinical transcription pipeline. Prioritization remains based on the MoSCoW method.

Must (FR-SM-x)

The system ***must*** be able to:

1. **FR-SM-1**: Initialize audio capture. *(US-R1)*
2. **FR-SM-2**: Maintain the session state (idle/dictating). *(US-R1)*
3. **FR-SM-3**: Accurately recognize radiology-specific terminology and abbreviations. *(US-R4)*
4. **FR-SM-4**: Accurately recognize specific voice commands for punctuation. *(US-R8)*
5. **FR-SM-5**: Continuously show the dictated text in the web interface. *(US-R5, US-R9)*
6. **FR-SM-6**: Be extended to work for other medical fields without retraining the whole model. *(US-M2)*
7. **FR-SM-7**: Run on the ZGT infrastructure. *(US-M5)*
8. **FR-SM-8**: Distinguish between text generated by the system and text manually added by the user. *(US-R11)*

Should (FR-SS-x)

The system ***should*** be able to:

1. **FR-SS-1**: Have a modular architecture such that individual components can be updated, replaced, or restarted independently. *(US-M3)*

Could (FR-SC-x)

The system ***could*** be able to

2. **FR-SC-1**: Show words flagged by the model as potential errors. *(US-R2)*
3. **FR-SC-2**: Write out any abbreviations as the complete medical term. *(US-R6)*
4. **FR-SC-3**: Show a timer to show how long a dictation is in progress. *(US-R7)*
5. **FR-SC-4**: Be deployed in a containerized format. *(US-M1)*
6. **FR-SC-5**: Calculate Word Error Rate automatically. *(US-M8)*

7. **FR-SC-6**: Calculate latency automatically. *(US-M8)*
8. **FR-SC-7**: Calculate real time factor automatically. *(US-M8)*
9. **FR-SC-8**: Calculate medical terminology accuracy automatically. *(US-M8)*

Won't (FR-SW-x)

The system ***will not*** be able to

1. **FR-SW-1**: Integrate with the HiX system in its current form. *(US-M4)*
2. **FR-SW-2**: Restrict access to unauthorized users.

2.6 Non-functional requirements

This section presents the non-functional requirements (NFRs) for the ASR system, defining the minimal performance and quality targets necessary for clinical implementation.

Performance and Efficiency

- The system must transcribe words at least twice as fast as the input is generated (RTF < 0.5). *(US-R3)*
- The system must support continuous dictation sessions of at least 60 seconds. *(US-R7)*
- The system must display partial transcripts with an end-to-end latency lower than 2 seconds for streaming dictation. *(US-R1, US-R3, US-R9)*
- The system shall sustain performance targets for dictation sessions of up to 15 minutes without degradation. *(US-R7)*
- Dictation must start within 1 second of the user pressing the "Start" button.

Data Privacy and Security

- The system shall run exclusively on local GPU infrastructure. *(US-M7)*
- The system shall operate without external cloud APIs and ensure no data leaves the ZGT infrastructure.
- If audio/transcripts are stored, retention shall be configurable and deletions must be logged. *(US-R5)*

Reliability and Availability

- The system shall support at least 1 concurrent session and be designed to scale to multiple sessions. *(US-R10)*
- If the WebSocket connection drops, the system shall allow reconnection within 10 seconds with clear user feedback. *(US-R1, US-R9)*
- If the backend fails, the frontend shall enter an error state and offer a restart without losing previously received text. *(US-R1)*

Maintainability and Usability

- The system shall be deployable via a documented, repeatable procedure (e.g., containerization). *(US-M1)*
- All model versions and configurations shall be versioned and capable of being rolled back. *(US-M3)*
- The system shall expose basic health checks and log key events like session starts/stops and model versions.
- Starting a dictation from the web interface home shall require no more than 2 user actions.
- The UI shall clearly indicate the current system state (Listening, Transcribing, or Paused).

2.7 Constraints

This section outlines the specific limitations and boundary conditions within which the ASR system must operate. These constraints are primarily dictated by the hospital's security policies, hardware availability, and the specific scope of this prototype.

- **Local Infrastructure Only**: The system must run completely locally on ZGT's internal infrastructure.
- **No External Processing**: The system shall have no external or cloud-based data processing, and no data may be shared outside the ZGT network.
- **Hardware Resource Limits**: The system must operate within the resources of a single NVIDIA H100 GPU.
- **Model Size Restrictions**: The system has limits to the model size, as implied by the specific hardware constraints.
- **Regulatory Compliance**: The system must comply with GDPR and other relevant medical data protection regulations.
- **No Direct Integration**: The current prototype has no direct HiX integration for automated data retrieval.
- **Project Timeline**: The development of the system is constrained by the limited duration of the design project.

2.8 Assumptions

This section outlines the core assumptions made during the design and development phases. These factors are considered true for the duration of the project and are essential for its successful execution and smooth operation.

- **Data Provision**: Training and testing datasets are primarily provided by ZGT.
- **Language Consistency**: Spoken input is assumed to be exclusively in Dutch.
- **Hardware Availability**: The prototype runs on the NVIDIA Spark GPU, which is assumed to be consistently available for development and inference.
- **User Feedback Integrity**: Radiologists are assumed to provide fair, accurate, and correct feedback when interacting with the model.
- **Ground Truth Definition**: Manual corrections made by users are treated as the absolute ground truth for subsequent training and fine-tuning.
- **Architectural Flexibility**: The system is designed such that switching between different ASR models does not require major code refactoring.
- **Equipment Reliability**: Radiologists are assumed to use a functional, high-quality microphone for all dictations.
- **Standardized Protocol**: Users are assumed to adhere to the specific voice commands and dictation conventions agreed upon at the start of the project.
- **Stakeholder Engagement**: Project supervisors and the client are assumed to provide timely and relevant feedback regarding the project's progress.

2.9 Diagrams

Component diagram

This diagram illustrates the modular decomposition of the ASR system. It defines the boundaries between the Frontend (responsible for audio capture and UI rendering), the ASR Backend (handling session management and post-processing), and the Model Runtime (where the actual inference occurs). This visualization confirms that individual components, such as the VAD or the post-processing layer, are decoupled to allow for independent updates or replacements.

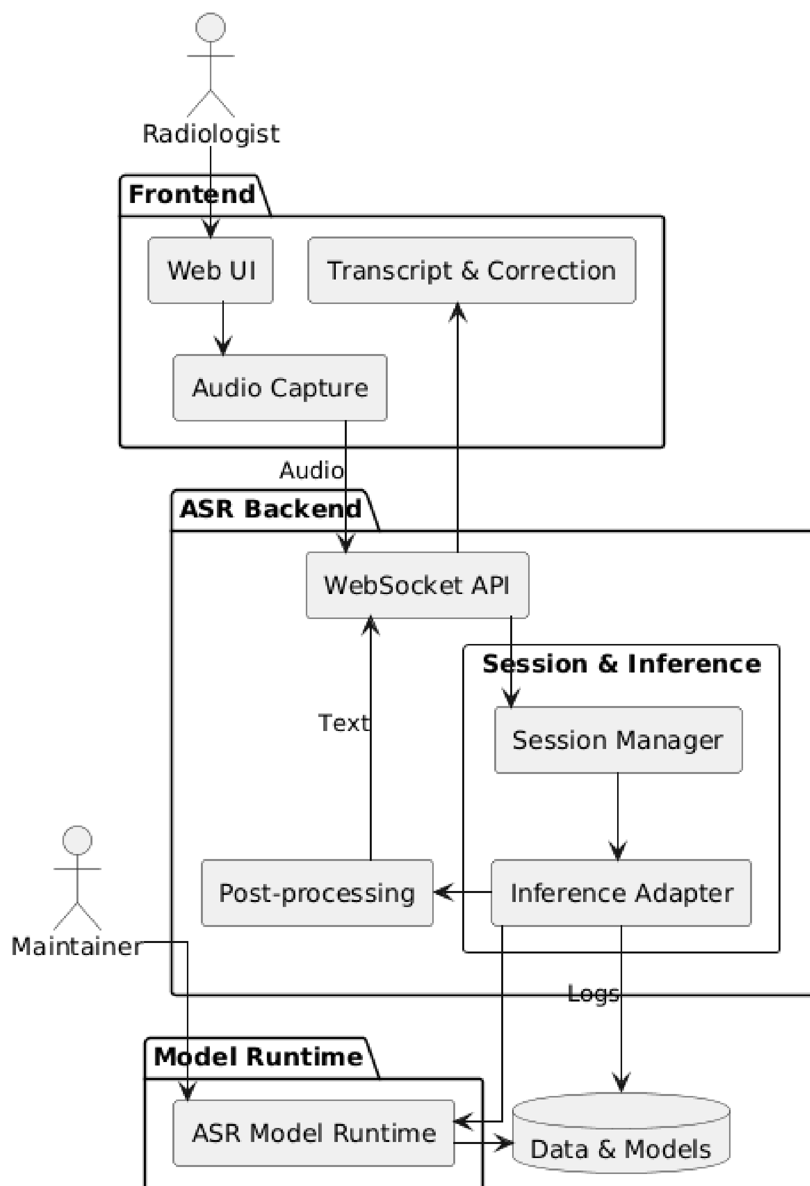


Figure 2: Component diagram of the ASR system

Deployment diagram

The deployment diagram maps the software stack to the physical hardware provided by ZGT. It shows the containerized ASR Stack running on the NVIDIA DGX Spark cluster, highlighting how the backend container communicates with the NVIDIA H100 GPU via CUDA for high-speed inference. This diagram is critical for demonstrating that the system operates entirely within the hospital's on-premise infrastructure, fulfilling local data privacy requirements.

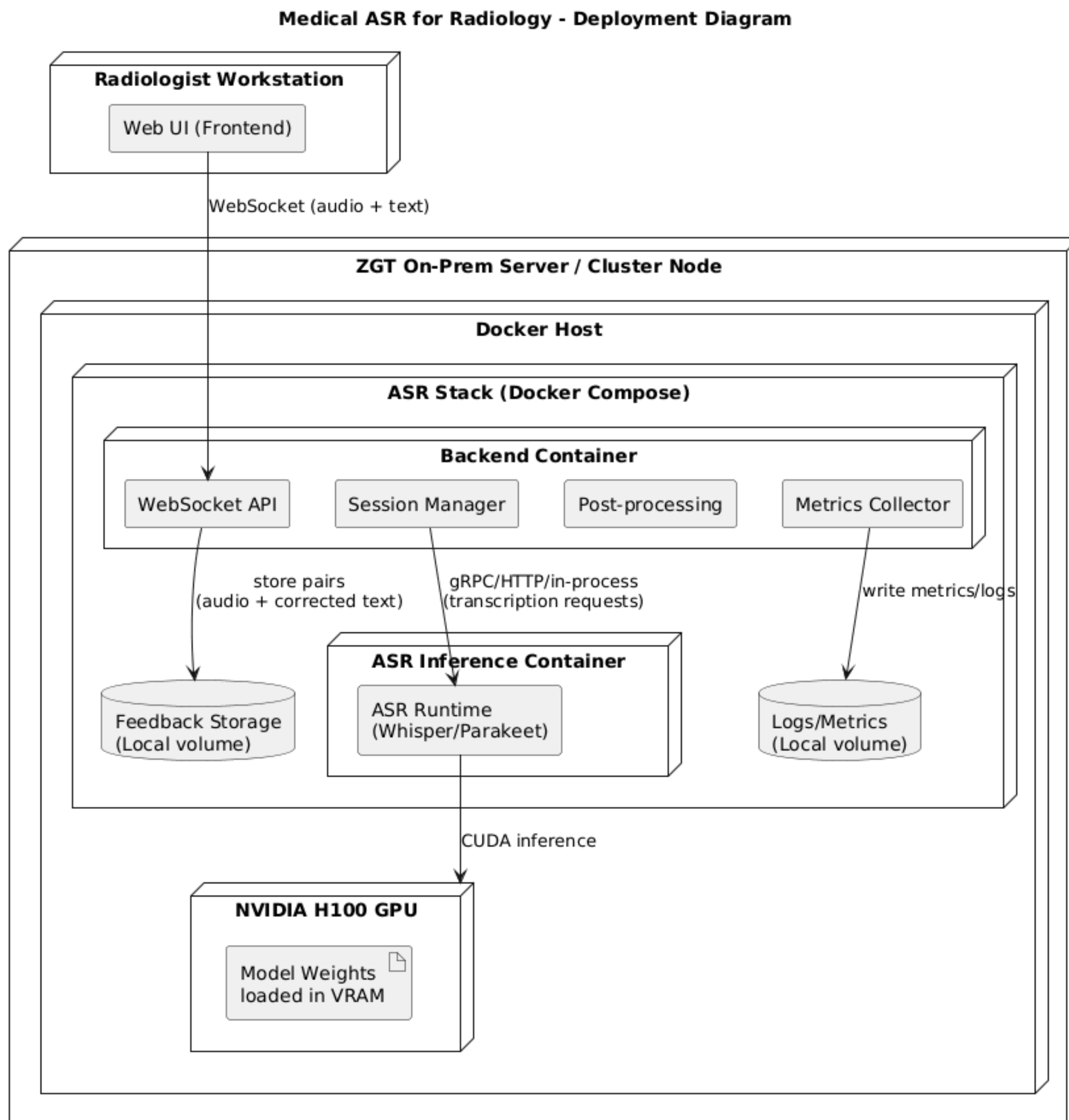


Figure 3: Deployment diagram

Sequence Diagram

This diagram, found in Appendix C, tracks the chronological flow of data and control during a standard reporting session. It visualizes the lifecycle of a dictation, starting from the WebSocket handshake and moving through the real-time processing loop where audio chunks are transcribed, cleaned by the post-processor, and streamed back to the user. The

diagram also explicitly shows the "Review & Edit" phase, where user corrections are captured and stored in the Feedback Storage to enable the continuous learning pipeline.

Chapter 3 - Project Approach

In this chapter, the choices made regarding the base ASR model used are discussed, as well as the source of the training data.

3.1 Architecture and model choice

3.1.1 Which open-source ASR Models are available for the Dutch language, and how are they competing against each other?

There are many open-source ASR models available online. As of now, HuggingFace is the de facto platform for exploring and comparing open-source AI models. This platform provides an entry point for finding various ASR models and further comparing them through more rigorous analyses. The most prominent ASR models currently available include Nvidia Canary, Nvidia Parakeet (NVIDIA NeMo Team, 2025), IBM Granite (IBM Research, 2023), or OpenAI's Whisper (Radford et al., 2022). These are very modern multi-language ASR models. These models are very versatile; they are trained on a wide variety of speech contexts and languages. The model required for this product will only need to transcribe Dutch speech. There are models specifically trained for Dutch, like *Kaldi_NL* (Povey et al., 2011) or Nvidia's *stt_nl_fastconformer_hybrid_large_pc*. These models are quite old; they were last updated in 2023 and 2024, respectively. This may not seem like a long time ago, but a study by Bălan et al. (2024) has shown that modern multilingual models outperform older Dutch-specific models out of the box. For example, on the Common Voice 17 Dutch test set, Whisper Large-v3 achieved a Word Error Rate (WER) of roughly 4.3%, whereas the traditional Kaldi_NL baseline lagged significantly behind at 20.7% (Bălan et al., 2024).

There are community-made, finetuned versions of these large, multilingual models for Dutch, such as yuriyvnv/whisper-large-v3-high-mixed-nl, which perform better in Dutch than the general model that was not finetuned. This specific model reports a 9.5% relative improvement in Dutch speech to the base whisper model (Perezhohin, 2024). To be viable for this product, these models would still require fine-tuning on a medical Dutch dataset, much like their base versions. There are also pre-existing models fine-tuned on medical speech, though the open-source models in this area are mostly community-made and therefore lack sufficient validation. They also do not work with the Dutch language at all, so this last type of model will be disregarded.

The following three types of ASR models will be considered:

- General multilanguage ASR models, like *Whisper*, *Canary*, *Parakeet,* etc.
- Community finetuned versions of the previous type models.
- A bit older models specifically developed for Dutch, like *Kaldi_NL* or *stt_nl_fastconformer_hybrid_large_pc* by Nvidia

3.1.2 How are objective comparisons made? Which metrics will be used?

There are four key performance metrics used to compare models. The first one is the Word Error Rate (WER). This is the primary metric for linguistic accuracy. It calculates the minimum number of alterations, such as substitutions, insertions, and deletions, required to transform the model's output into the correct transcript. Alongside WER, the Character Error Rate (CER) will also be used. While similar to WER, CER calculates the minimum number of alterations at the character level rather than the word level. This is exceptionally critical in a medical context, where a single incorrect letter can drastically alter the meaning of a diagnosis or medication name (e.g., *hyperthyroidism* versus *hypothyroidism*). The third metric is the Real Time Factor (RTF). This metric measures how fast a model can transcribe speech. This indicates how quickly the model works relative to the audio's length. If a 10-second clip takes 5 seconds to process, the RTF is 0.5. The last important metric is the latency, which measures the "lag" or delay. It is the total time it takes for the system to hear a sound and respond.

Apart from these performance metrics, the Dutch language must also be taken into account in support of each model.

3.1.3 What is the trade-off between model size and real-time performance on the available hardware?

There are multiple Nvidia DGX Sparks available to us. Each unit is powered by the NVIDIA GB10 Grace Blackwell Superchip (NVIDIA 2026), providing a massive 128GB of unified memory. The primary bottleneck for modern ASR models like Whisper Large-v3 (approx. 1.55B parameters) or NVIDIA Canary (1.1B parameters) is typically Video RAM (VRAM). On standard consumer cards (e.g., RTX 4090 with 24GB), running multiple large models or performing full fine-tuning can quickly lead to "Out of Memory" (OOM) errors.

Large models like Whisper Large-v3 typically require only 12-16GB of VRAM for inference. With 128GB available, several instances of these models can run simultaneously to handle multiple Dutch audio streams without a performance hit.

Fine-tuning is significantly more resource-intensive, often requiring 3–4x the memory of inference to store gradients and optimizer states. While a 1B-parameter model might struggle on a 24GB card during training, the DGX Spark can handle fine-tuning for models with up to 70B parameters. The target ASR models are well within this limit, allowing for full-parameter fine-tuning on medical Dutch datasets

In summary, the NVIDIA DGX Spark provides ample resources to meet the requirements. The hardware will not be a limiting factor

3.1.4 Model selection and strategy

Given the current landscape and hardware capabilities, older models like *Kaldi_NL* were discarded. While historically significant for the Dutch language, their architectures are

bound by their training data and are significantly outperformed by modern alternatives in both general and medical contexts.

Instead, the strategy focused on utilizing large, modern multi-language models (such as *Whisper Large-v3* or *NVIDIA Canary*). Because the NVIDIA DGX Spark hardware, with its 128GB of unified memory, completely removes the VRAM bottleneck typically associated with fine-tuning billion-parameter models, it provides the computational freedom to perform full-parameter fine-tuning. Therefore, state-of-the-art base models and high-performing community-finetuned Dutch variants were selected, and fine-tuned strictly on a specialized Dutch medical dataset to optimize for both low Word Error Rate (WER) in clinical terminology. It was unclear how well each of these models would perform after fine-tuning. Therefore, multiple models were considered, fine-tuned, and compared to determine which would perform best. The models were compared on the metrics discussed previously. Unfortunately, these metrics are not based on tests with Dutch test sets.

Model name	WER	RTFx	Note
nvidia/canary-1b-v2	4.89	630.22	
openai/whisper-large-v3	4.91	126.16	
nvidia/parakeet-tdt-0.6b-v3	5.05	2154.22	
openai/whisper-large-v3-turbo	5.44	188.6	
yuriyvnv/whisper-large-v3-high-mixed-nl	4.43*	-	
yuriyvnv/whisper-large-v3-cv-fully-synthetic-nl	4.44*	-	
jonatasgrosman/wav2vec2-large-xlsr-53-dutch			Older model type, just to test

Table 2: A comparison of different STT models for Dutch speech based on WER and RTFx.

The last two models in the table are fine-tuned models for Dutch. The reported WER is based on a self-reported WER by the creator on the Dutch Common Voice dataset. This differs from the dataset used in the other models, which are averages across many datasets. Given that these numbers cannot be directly compared, and given the performance of the non-finetuned models in Dutch, tests were conducted to assess how well these models perform natively in the target use case. For this, a subset of the data provided by ZGT was used, specifically radiology dictations that match the target use case. All models will be tested in the exact same way, and metrics such as Word Error Rate (WER) and Character Error Rate (CER) will be computed. The model that performs best on this domain-specific dataset is expected to also perform best after fine-tuning, making it the most suitable base model for further development.

3.2 Data & Privacy

Dataset selection & Criteria

The goal of this project was to create an ASR system that is somewhat optimized for Dutch radiological dictation. To achieve clinical-grade accuracy, the training data had to reflect the domain-specific terminology and linguistic patterns unique to radiology. (Ko et al., 2017)

In selecting the dataset, three paths were considered to obtain the data.

- Public Datasets
- Domain-Specific Clinical Data
- Synthetic Data Generation

Public Datasets

Most Dutch datasets are either too small or highly restricted. The following three, however, are not:

- Mozilla Common Voice: This is the largest “crowdsourced” dataset. It is great at capturing the real-world variety. It contains various accents, background noise, and microphone qualities, which are essential to making a model robust. (*Mozilla Common Voice*, n.d.)
- CGN (Corpus Gesproken Nederlands): the largest Dutch speech dataset. It contains nearly 800 hours of data, ranging from formal news broadcasts to spontaneous conversations.
- Google Fleurs: This dataset has been used to train many state-of-the-art models, such as Whisper. (Conneau et al., 2022)

Below is a short summary of the three datasets, including their licenses, GDPR status, and use constraints.

Dataset	**License**	**GDPR Status**	**Use Constraints**
Mozilla Common Voice	CC-0 (public domain dedication)	Public dataset (voice data collected with consent; still biometric data under GDPR)	Must check version & attribution requirements
CGN (Corpus Gesproken Nederlands)	Research license	Contains personal speech data; processed under a research framework	Usage typically restricted to research
Google FLEURS	Creative Commons Attribution 4.0 (CC BY 4.0)	Public dataset (voice recordings; biometric personal data released under	Allowed for research and commercial use with mandatory attribution.

		informed consent)	

Table 3: An overview of three datasets for the Dutch language (Mozilla Common Voice, n.d.; Conneau et al., 2022).

CGN is primarily distributed under licenses minted for academic research. While commercial licenses exist, they involve high costs and strict usage terms. Because this project aimed for a clinical application in a hospital setting, a standard research license would likely be insufficient. Furthermore, the CGN license generally prohibits the distribution of derivative works. Since the speech-to-text model's weights would be directly trained on this data, the model itself could be legally classified as a derivative work. This would prevent the distribution or scaling of the product to other departments or institutions without explicit, and likely costly, additional permissions from the Nederlandse Taalunie (Dutch Language Union), which owns the CGN.

Besides CGN, there is also Mozilla Open Voice and Google Fleurs. Open Voice is particularly advantageous as it is released under a CC0 Public Domain license. This allows for the development and distribution of an ASR system without legal encumbrance or royalty obligation. Google FLEURS is smaller in volume and uses the CC-BY 4.0 license, which permits commercial application and serves as a good baseline for evaluating model performance in Dutch. Many other models are also evaluated on this metric. However, since the final ASR system was specialized for radiology speech, it would most likely perform worse on regular speech than it did before fine-tuning. Because of this, evaluating on regular Dutch speech data would not show any contextual performance metrics and is therefore obsolete. That is because ASR systems trained or adapted to a specific domain perform best in that domain, and performance degrades when evaluated on mismatched speech data. (*Mozilla Common Voice*, n.d.; Conneau et al., 2022; Ko et al., 2017)

A major issue, however, was that the datasets mentioned contain little medical terminology. This is crucial to creating a good ASR system for this project. Hence, the other options were expected to be more viable.

Domain-Specific Clinical Data

Domain-Specific Clinical data would be accessible through the client (ZGT). A script would record radiologists' dictation, which would then be uploaded to a shared drive. An issue that could come up, however, is that the dataset size might be limited. The upside of this data, however, is that it perfectly fits the hospital's request. The ASR system will be designed for the people who also provide the data, whose purpose is to train the model. This would give better results than using training data that is very distant from the radiologist's speech. (Ko et al., 2017)

Synthetic data

If synthetic data were to be used, complete freedom would be given in what terminology and accent would be used, on top of other things. The data would also be easy to generate, hence this form of data gathering would not lead to limited data access for model training. A downside, however, is that it is still difficult to reproduce the radiology dictations. The intonation and sentence breaks, among other things, might be hard to replicate. (Huddy, 2025)

Final choice

In the end, it was decided not to use the publicly available datasets. But first, try to use the data provided by the ZGT effectively. If this would not work out for any reason, e.g., the dataset size would be too small, then creating the data synthetically would be attempted and either used on its own or combined with the provided data to create an optimal ASR system within the limited timespan.

Datasourcing & acquisition

The ZGT-provided data was collected using a script on their end and uploaded to a shared drive as a zip file with the date of the upload. The unzipped folder would contain subfolders named after a specific speaker. The specific speakers would then each have their own recordings in their own folder. The base name of the recording matches their transcript. Which is how they could be paired. The recordings (.wav files) would contain the speech data of the relevant text. The text would contain the output of what their current system (G2 speech) heard.

There was no need to anonymize the data, as this had already been done by ZGT.

Chapter 4 - General Design Choices

During the design of the ASR system, some choices had to be made that were critical to the project's course. In this chapter, deliberate inclusions and exclusions are mentioned with a short explanation as to why a certain choice has been made

4.1 Finding the base model

During the research phase, the following possible base models to use for the system were identified:

- nvidia/canary-1b-v2
- openai/whisper-large-v3
- openai/whisper-large-v2
- nvidia/parakeet-tdt-0.6b-v3
- openai/whisper-large-v3-turbo
- yuriyvnv/whisper-large-v3-high-mixed-nl
- jonatasgrosman/wav2vec2-large-xlsr-53-dutch

To make a definitive decision, an evaluation was conducted to determine which model performed best on the target use case without any fine-tuning. The assumption was made that the model performing best before fine-tuning would also yield the best results after fine-tuning.

In week 5 of the project, access to the first batch of data was granted, comprising approximately one hour of audio with corresponding transcripts. Because this was the only available data at the time and one hour of audio is generally sufficient for evaluation, this batch was designated as the test set.

4.1.1 Verifying and Normalizing

The transcripts of the received data were obtained directly from ZGT's previous ASR system. These transcripts had already undergone some post-processing and contained inherent inaccuracies. Specifically, the dataset contained the following issues:

- Mistakes made by the previous ASR
- Punctuation
    - Spoken punctuation commands had already been converted into symbols. For example, if the radiologist said 'Punt' (English: Period), it appeared as a period (".") in the transcript. The same applies to commands for new lines, new paragraphs, colons, parentheses, etc.
    - Whenever a new line or new paragraph was said at the beginning of the audio, it was dropped in the transcripts for no apparent reason.
- Abbreviations were used in the transcripts, while it was spoken in full in the audio.

The mistakes were corrected manually, and a script was used to normalize the transcripts. This script effectively reversed the post-processing by removing punctuation and replacing it with the actual spoken commands.

4.1.2 Results

After normalizing the test set, the base models were used to evaluate their zero-shot performance. The results are detailed in the table below:

Model	**WER**	**CER**
parakeet-tdt-0.6b-v3	.4417	.2081
whisper-large-v3-turbo	.4697	.2681
whisper-large-v3	.4750	.2199
canary-1b-v2	.5055	.2995
wav2vec2-xls-r-1b-dutch	.5746	.3176
whisper-large-v3-nl	.6153	.3456
whisper-large-v2	.7228	.3569

Table 4: An overview of the zero-shot performance of various STT models that were considered for this project.

4.1.3 Conclusion

Based on the zero-shot performance results, *nvidia/parakeet-tdt-0.6b-v3* emerged as the most accurate base model for the specific radiology use case. Achieving the lowest Word Error Rate (WER) of 0.4417 and Character Error Rate (CER) of 0.2081, it noticeably outperformed the other candidates.

It is important to note that although these baseline error rates may initially appear high, this is expected given the dataset's challenging nature. The audio recordings consist of highly dense medical jargon specific to radiology, specialized vocabulary that general-purpose ASR models are not extensively trained to recognize. Furthermore, the radiologists in the recordings tend to dictate at a very fast pace, which significantly compounds the difficulty of zero-shot transcription. These factors combined perfectly illustrate why fine-tuning was an absolute necessity for this project.

Beyond its superior accuracy on the baseline test, another significant advantage of the *parakeet-tdt-0.6b-v3* model is its exceptional inference speed, measured by its high Real-Time Factor (RTFx). Because the model uses a Token-and-Duration Transducer (TDT) architecture, it can process and transcribe audio incredibly quickly, a crucial feature for the end-user experience in a clinical setting. It could also support future improvements, like

adding an auditing LLM to the ASR output, since the model is fast enough to allow ample time for a secondary layer to review it.

Furthermore, this model choice provides major logistical benefits for the development cycle. Because the Parakeet model is relatively compact (0.6 billion parameters) compared to its closest competitor in the tests, whisper-large-v3 (1.55 billion parameters), the computational overhead required for fine-tuning is drastically reduced. This smaller footprint meant training runs would be considerably faster and require less GPU VRAM, enabling more efficient experimentation and iteration.

While the *whisper-large-v3* and its turbo variant also showed competitive results, the Parakeet model demonstrated a much stronger baseline understanding of the complex dataset right out of the box. Following the initial hypothesis that the strongest pre-trained model would ultimately yield the best results, the *parakeet-tdt-0.6b-v3* was deliberately selected as the foundational model for the fine-tuning phase of the ASR system. However, to avoid relying entirely on a single architecture, it was also decided upon to concurrently train a LoRA (Hu et al., 2021) for Whisper-large-v3, just to keep the options open.

4.2 Web application decisions

During the project's implementation phase, especially for the web application, certain design choices had to be made that shaped the prototype as it is now. From the UI to everything that produces the ASR's output, all options were carefully studied and selected for performance. Below, the most important decisions made during the web application's implementation phase will be outlined.

4.2.1 General Frontend Design

The User Interface can make or break a product. This meant ensuring that the UI is simple, clear, and intuitive.

We decided to make the interface as simple as possible, only containing relevant information for the user. It consists of a start/stop button, a train button, an edit button, and a submit button. This button layout is below the main text box, where the ASR will display its output. No specific research has been conducted on this exact layout, but this choice was made because it is still a prototype that could be improved. When the audio recording starts, an indicator (a title in the main text box) appears to notify the user.

The main output of the ASR is put into a form on the web application. This makes it easy to edit the ASR output and submit changes through an API. For now, since there is no integration with other systems, the submission is not completely relevant, but ensuring this is a possibility for implementation when integrations are in place is important. Also, when the train mode is selected, submitting the (edited) text for retraining is possible using the same kind of API, but, again, implementing this was outside the scope of this project.

4.2.2 Overall backend design

The backend design is a pipeline with multiple parts, as shown in Figure 4. When audio comes in from the websocket, it is first stored in a buffer, then split into fragments according to the audio chunking strategy. When the splitting logic returns a fragment, it will optionally be saved and then passed to the ASR model. When overlap is used, timestamps of the start and end of the currently relevant part will be included. The ASR model then transcribes it and, if relevant, filters out any text spoken during the overlapping part. This original text is sent to the postprocessing pipeline, which applies a series of configurable postprocessing steps. When finished, both the original and the processed text will be sent to the frontend via a websocket and stored in the database.

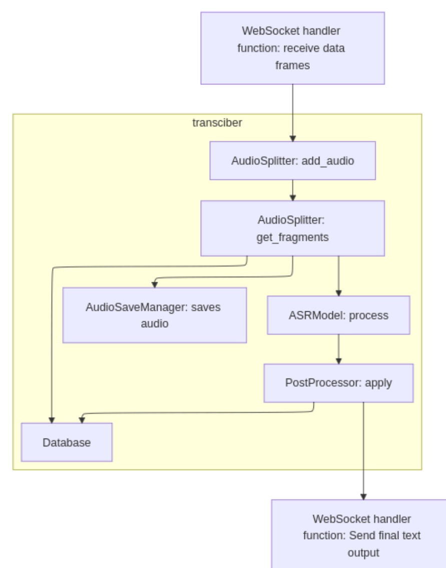


Figure 4: Diagram of backend design.

4.2.3 Chunking strategy

The ASR model cannot handle a continuous stream of audio, so a strategy was needed to fragment it into smaller blocks (or chunks). The NeMo Parakeet models recommended a maximal input of 30 seconds, so the strategy was to create blocks of at most that length.

Fragmenting at exactly 30 seconds every time could cause words to be cut off in the middle. To solve this, a voice activity detection (VAD) system, Silero (Silero Team 2025), was chosen; however, it is easily interchangeable with another VAD system that can provide the probability that a fragment contains speech. The probability data from this VAD is parallel to the audio stream. This way, there are 2 buffers in which the audio can be compared against its voice-activity level. The VAD detects if the currently streamed audio is actually a voice. The value of the VAD is high when it detects speech, low when it does not. Fragmentation into chunks can occur when it is certain there is at least 3 seconds of silence. This way the

problem where words are cut off in the middle is mostly solved.

In order to allow passing overlapping audio for models that support timestamps, the system can also store some audio from a previous fragment, and return a larger audio fragment including both audio from before and after the main fragment, together with timestamps indicating where the part of this fragment that should be transcribed begins or ends.

Another issue was when no silence was detected, or when the silence was not long enough. In that case, a split must occur after the maximum fragment length (30 seconds). To ensure that, in this split, there is no cutoff in the middle of a word, the decision was made to cut at the point where the VAD yields the lowest voice-activity probability. There are always silences in someone's speech, even if it is only for a couple of milliseconds. Using the minimum value of the VAD, it is possible to be somewhat confident that no split occurs in the middle of a word.

4.2.4 Database

The database stores transcripts corresponding to audio fragments and metadata about full transcripts. Storing original and corrected transcriptions is necessary to use them for training a new ASR model, and storing the edited and processed text in the backend also prevents data loss on the frontend. A database has been chosen over separate files for performance and to simplify the codebase. Training data can be exported using a script that reads it from the database or from an API endpoint.

Chapter 5 - Implementation

With the base model selected and the system architecture defined, the next challenge was building a robust environment to handle all this data. This chapter explains how the ASR system was implemented from the ground up. It covers the automated infrastructure for managing files, the data normalization workflow for processing raw audio and transcripts, and the training strategies.

5.1 File Structure

The ASR Radiology project is organized into two primary environments: a local Git repository containing the source code and application logic, and a shared data drive hosted at '/mnt/data/asr_ut/' which stores large-scale datasets, model checkpoints, and evaluation assets. This separation ensures that the codebase remains lightweight while providing the training pipeline with high-speed access to heavy binary data.

A comprehensive visualization of the project's repository and shared data drive structure, including detailed descriptions of key directories and configuration files, is provided in Appendix D.

5.1.1 Local Repository Organization

The local repository is structured as a modular Python project managed by *uv*. It is divided into the core ASR pipeline (training/), a web-based transcription application (backend/), and configuration assets.

In the training code, pyproject.toml exposes CLI commands such as create-dataset and train, allowing researchers to trigger complex workflows directly from the terminal, while src/config.py serves as the single source of truth for directory paths, ensuring the code dynamically resolves locations on the shared drive. Additionally, the config/abbreviations.txt file is critical for medical ASR accuracy, as it defines the expansion rules used during the preprocessing stage to convert shorthand radiology notes into full spoken Dutch.

5.1.2 Shared Data Drive Structure

The shared drive serves as the project's data storage and model registry. It is organized to support reproducibility by maintaining fixed speaker splits and versioned datasets.

The config/speaker_split.py file ensures that data from a specific doctor or radiologist never leaks from the training set into the test set, guaranteeing the model's ability to generalize to new voices. At the same time, the data_newline_detected folder contains transcripts that have undergone an initial Whisper pass to insert structural markers like 'nieuwe regel' (new line), essential for generating formatted medical reports.

5.1.3 Global workflow integration

The interplay between these two structures follows a linear progression. The code in src/ pulls raw data from the shared drive, processes it into structured CSVs within the datasets/ directory, and saves the resulting fine-tuned models back to the models/ folder. Finally, the backend/ application loads these models to provide a real-time transcription interface for end-users.

5.2 Data ingestion and normalization workflow

5.2.1 Receiving the raw data

The received data from the ZGT would be in a zip file of the corresponding day. Each day would contain a folder for each speaker, with their corresponding audio and text files within that folder. The text file and audio file would share a common base name, in the form of a number, allowing for programmatically pairing each audio file with its corresponding text file.

The audio files would contain the spoken audio. The text files would contain the script of what was said in the audio file, translated with the current installed system: G2 speech. The radiologists have to add special characters by saying them in their complete form. Below, a list can be found of all the special commands:

‘komma’	→ “,”	(comma)
‘slash’	→ “/”	(slash)
‘punt’	→ “.”	(dot)
‘dubbele punt’	→ “:”	(colon)
‘punt komma’	→ “;”	(semicolon)
‘haakje openen‘	→ “(“	(opening bracket)
‘haakje sluiten’	→ “)”	(closing bracket)
‘nieuwe regel’	→ “\n”	(new line – *one line break*)
‘nieuwe alinea’	→ “\n\n”	(new paragraph – *two line breaks*)
‘graden’	→ “°”	(degrees)

G2 speech would then apply postprocessing and convert these characters back to their original form. And this would be the version in the obtained dataset.

5.2.2 Preprocessing

Among other issues, there was an issue in the data, because if a data pair started or ended with a new line, it would be cropped from the text file. So if a transcript started or ended with the speaker saying “nieuwe regel” or “nieuwe alinea”, this would later confuse the model during finetuning, since these commands would be replaced with a newline symbol, which would then be cropped. Since the predicted text would not correspond with the expected token. A solution to this was to run a plain whisper model on all the obtained data, checking for the start and end of a file and whether one of the two mentioned commands was heard; if it was, the heard command would be added back to the text transcript. An issue with this, however, was that it would not find all the mistakes, and many would still slip through. Although many still slipped through, improvements were found in the final model.

Another similar issue that arose was that when “nieuwe aliena” was mentioned, only one new line was added instead of two. The solution for this was similar: in the same script as the start and end, it would now also check whether any “nieuwe alinea” in the middle of the script had been said. If it had, it would insert two new lines in the transcript. This also did not catch all the mistakes, but it did catch some, which showed improvements in the model results.

This entire process would take a while to complete, so it was decided not to include it in the preprocessing pipeline and to make it its own step. The script would essentially create a copy of the initial dataset; however, now with the newly inserted newlines.

Pipeline

Since the transcripts still contained post-processing artifacts from the G2 system, which would not match the spoken audio, preprocessing steps still had to be applied. The preprocessing pipeline happened in the following steps:

The manifest would be loaded from the raw files. It would look for all pairs of audio and their corresponding transcript. The speaker label would then be derived from the folder name of the pair. This would result in a dataframe containing the file paths and the transcripts.

Afterward, the text would be normalized. This would include converting all text to lowercase. Furthermore, all known abbreviations would be written out. For example, G2 speech would put down ‘cm’ when the speaker had said “centimeter”. The normalization step reverted this. Lastly, the above-listed commands would all be converted to their original spoken form.

In the next step, speakers would be split up using a fixed list defined in the config file on the shared drive. This file contains the names of the speakers and should therefore remain on ZGT's remote infrastructure. It is essential that everyone can access the current version of the file. Hence, the decision was made to keep this on the shared remote drive. In this step, the data is separated into three buckets. It separates speakers for the *train*, *dev*, and *test* sets. This ensures the model is trained and evaluated on a different set of speakers. This prevents inflation of results. Then it is also calculated how many hours are needed for the dev and test set, based on the given amount of training hours, which can be passed as a flag.

After splitting up the speakers, the script tries to allocate equal audio length to each speaker in each split. This uses a fixed random state, ensuring that if the script were run again, it would give the same results. First, the audio would be resampled to 16kHz then the audio length can be calculated using the absolute time of audio files, or the active voice can be used by passing it as a flag. When this is applied, the script uses signal processing to estimate a speaker's time and applies a decibel threshold. The advantage of this is that when one speaker is quiet for long periods in an audio file, it ensures an equal split for all the speakers.

Now all the pre-processing steps have been applied, and the final dataset can be assembled. A *train*, *dev*, and *test* CSV file would be saved to the split directory. Which can then be used separately for their own specific purposes. When a new dataset is created, it does not overwrite the old one; it creates a new folder. This was done to ensure that results could be properly compared. On top of that, one dataset might not outperform the old one when the model is fine-tuned; hence, it is generally a good idea to keep the old dataset.

Below, a diagram of the overview of the pipeline can be found.

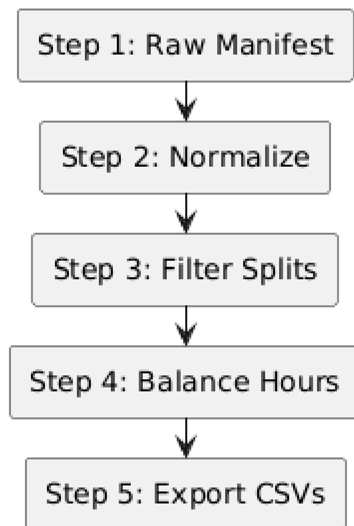


Figure 5: A diagram of the overview of the pipeline.

5.2.4 Post Processing

After the data has been preprocessed and the ASR has made suggestions, these suggestions need to be processed back into their original state. This process, called 'post-processing', would basically reverse the steps done in the pre-processing step. It will transform the 'punt', 'komma', and all the other punctuation commands back to their punctuation form and revert the 'nieuwe regel' and 'nieuwe alinea' back to 1 newline and 2 newlines respectively. The basic implementation flow is the following.

Once the data has been processed by the ASR and it produces a stream of unprocessed fragments, it is important to ensure some overlap among the fragments. This ensures that commands like 'nieuwe regel', which could be chopped up into 2 different fragments, are actually handled like a 'nieuwe regel' command, instead of being processed as 'nieuwe' and 'regel' separately.

After that had been implemented, work started on a substitutor. This substitutor works by replacing the (complete) punctuation command with its represented punctuation. This substitutor will not substitute punctuation commands when it is part of another word, for instance, 'standpunt' contains 'punt', but it will not be substituted with 'stand.'

But before data can actually be processed, accidental punctuation inferred by the ASR system must be removed. This is done by the punctuation stripper. It works by taking a string like “hello, world!” and replacing all the punctuation and returns the following: “hello world”. By now, data can be processed with the substitutor and give the sentences some structure with punctuation

Finally, to be complete, the sentence also requires some capitalization. This function is designed to capitalize the first letter of a sentence. It works by first checking which letter comes first, then checking for the end of a fragment and its punctuation. If there is no punctuation in the current fragment, it will also check in the next fragment. If it finally finds a “punt,” it will capitalize the first letter of the next word again.

To finish off the post-processing, the incorrectly generated whitespace needs to be restored to its original state. Sometimes it can happen that when processing something like "end punt begin" → "end . Begin". Then the whitespace in this sentence needs to be fixed to the following: "end. Begin". For all the different punctuation that could require some additional check on the spacing, for instance, with brackets having "(some text here)" and normally having "(some text here)" (no whitespaces before the first and after the last word), the spacing is checked, fixed if incorrect, and the corrected sentence is returned.

5.3 Training strategy

5.3.1 Variable dataset sizes

Instead of training the model on all the cleaned data at once, this was done step by step. A script was used to create several datasets with varying audio lengths, such as 1-hour, 5-hour, and 10-hour batches.

We created these different sizes for two main reasons. First, it helped in understanding how the amount of training data affected the Parakeet model's ability to learn. Second, training on smaller batches of data was much faster. This made it possible to quickly run tests and check the results by hand. If the model kept getting certain phrases wrong, a smaller dataset made it much faster to track down the cause. Often, this helped spot and fix mistakes made earlier when cleaning the text, such as forgetting to spell out a medical abbreviation, or improve “nieuwe regel” detection. With increasing amounts of data, 1-hour, 5-hour, 13-hour, and 22-hour datasets could be created.

5.3.2 Training variations

Along with testing different dataset sizes, various training settings (often called hyperparameters) were also experimented with for each run. Specific details were tweaked like the learning rate (how quickly the model updates its knowledge), the batch size (how much audio it processes at one time), the optimization algorithms (the specific rules the

model uses to improve itself), and the number of epochs (how many times the model reviews the entire dataset during training).

Because medical dictations are full of complex jargon and radiologists speak very quickly, the model's default settings were not always the best fit. By testing these combinations in a structured manner, it was possible to determine which settings produced the most stable and efficient learning process for the specific radiology system.

While the primary focus was on full-parameter fine-tuning of the Parakeet base model, an alternative training strategy was also explored. A Low-Rank Adaptation (LoRA) was attempted for the whisper-large-v3 model. Because Whisper is a significantly larger model, using LoRA would theoretically enable efficient fine-tuning for medical Dutch without the massive computational overhead of updating all its parameters. However, the experiments with Whisper LoRA did not yield much success. It struggled to match the efficiency and accuracy improvements demonstrated by Parakeet, ultimately reaffirming the decision to center optimization efforts entirely on the Parakeet architecture.

5.4 Evaluation loop

5.4.1 The evaluation set

The resulting model after each run was automatically evaluated. For this, an evaluation set was required. It was crucial that this evaluation data was kept completely separate from the training data. To ensure the test was fair, the data were also separated by radiologist. This meant the model was never evaluated on a speaker it had already heard during training. This guaranteed that the model was actually learning to understand medical Dutch, rather than just memorizing a specific doctor's voice.

5.4.2 Logging and tracking

After every single training run finished, the system automatically saved a record of what happened. It logged the exact training settings used, such as the learning rate, batch size, and number of epochs, along with the final results on the evaluation set. These results included performance metrics such as the Word Error Rate (WER) and Character Error Rate (CER). It also saved the predictions made by the model on the evaluation set, along with the correct transcript, which made it possible to inspect the kind of mistakes the model was making. By keeping a strict log of every experiment, a clear history was created that showed exactly which parameter combinations worked best and which ones caused the model to struggle.

5.4.3 Iterative improvement

The logged results were continuously reviewed to determine the next steps. If a training run showed high error rates, the logs could be inspected to see whether the training settings needed to be tweaked again or if there was still an issue lurking in the text normalization

rules. To make this iterative process as efficient as possible, these cycles were frequently automated to run overnight. For example, using the exact same base model and dataset, the system could be programmed to systematically iterate through different settings: the first run might use a specific learning rate for a set number of epochs. As soon as that finished and logged its results, the system would automatically start the next iteration with the same learning rate but a different number of epochs, followed by another iteration with a completely new learning rate. The following day, the results would be analyzed and a new sweep conducted to narrow the parameters around the top-performing values.

This automated testing made it possible to wake up to a detailed comparison of multiple iterations, creating a highly effective loop: train, evaluate, check the logs, and improve.

5.5 Real-time Dictation Prototype

We were tasked to implement a real-time dictation prototype. This prototype should make it possible to use the trained ASR model and speak to it, transcribing what is said in real time and displaying it on the interface. The prototype's implementation can be split into the usual front-end and back-end components. These implementations make it possible to have a working prototype that can be used and optimized for future improvements (see also the future improvement section)

5.5.1 Front-end Implementation

We tried to keep the interface as simple and as intuitive as possible. The interface would include a start-stop button to start and stop the voice recording, a simple way to submit the final (edited) transcript to their system, and a way to train the model on the given text.

The interface includes a form section where the transcribed text will appear. Once the user clicks the start dictation button, an indicator in the form will appear to show that recording has started and the microphone is connected. Then, while the user is speaking, the form will be filled with fragments of ASR output that have already been processed by the backend. Finally, once the user has stopped the recording, the recording form indicator will disappear. The user is now able to edit the transcript to their liking, and once the implementation with the system is complete, they can save the transcript to their system.

5.5.2 Back-end Implementation

The backend implementation is slightly more complex than the frontend implementation. The basic idea of the backend is to first handle the audio received from the front end. This is done via a websocket, which, once the socket is active, streams audio through the backend's pipeline.

The first process the audio then takes is the conversion to a usable sample rate. The selected model (currently using the NVIDIA NeMo framework, but different implementations are possible) takes audio with a sample rate of 16kHz. The microphone from the web

application sends 48 kHz, which has to be resampled. This is done with the *soxr* resample function.

Once this is done, the audio is split into usable fragments. For splitting the audio into fragments, *silero* voice activity detection (VAD) is used, but this can easily be replaced with another VAD. Two different synchronous buffers are created, one for the audio stream and one for the VAD on that audio stream. Then there are two different cases for the splitting of the audio. One case is where the VAD detects a silence of more than 3 seconds. Then it fragments the audio at that position and adjusts the buffers for the next fragment. The other case that can happen is when the talking is longer than the permitted (maximal) fragment size of the defined 30 seconds. Then the VAD estimates the best point to fragment the audio section. Finally, the leading silences in each of the fragments must be accounted for, such that it does not continuously split on these leading silences and only splits after speech has occurred. Additionally, long silences should not be sent to the ASR, to prevent waste of resources and random transcriptions based on the noise.

These audio fragments are then processed by the ASR. This is done by simply sending the resampled audio data, which is raw PCM, to the ASR model, which is loaded once the web server has started. Additionally, if the option to save the audio for training is enabled, a wave file with the audio is saved. Each time a fragment is passed onto the ASR, it returns a JSON-formatted text fragment, which now makes it possible to continue to the post-processing step. This post-processing step and the post-processor itself are explained in section *5.2.4* of this chapter. Once the text fragments have been processed by the post-processor, they will finally be sent to the web client over the websocket once again, revealing the transcribed text to the user.

For now, a database that saves the text with the corresponding audio fragments is also used. Text is stored in four different categories: the original text, the original text after manual correction, the text after processing, and the final edited text. This is stored in a database to limit file IO during normal operation and transcription.

5.5.3 Connection

The application runs on the *FastAPI* framework. This is a framework that allows for quick api creation. The server runs on the local host, which allows the use of the microphone without having to establish an HTTPS certificate. In production, this can be replaced by adding an HTTPS proxy in front of the backend. Several routers have been implemented that can handle the connection between the frontend and the backend.

There is a websocket endpoint that handles the audio stream from the web client. For handling the incoming audio, it creates a transcriber object responsible for handling the transcription process. This transcriber is also responsible for sending the transcriptions back over the websocket. The websocket endpoint stores this transcriber in a connection manager in order to be able to reconnect the frontend to the relevant transcriber in case the

connection is lost. Then, once the connection is established and open, for instance by clicking the start dictation button, the web socket is open, and a live audio stream is sent through for processing. The fragments sent back over the websocket are then put into the form in the frontend.

There is also an endpoint for updating a fragment through a PATCH request. The frontend calls this request when a fragment is edited, and the backend updates the stored fragment in the database.

Lastly, there are endpoints for retrieving fragments from the backend. The main relevant endpoint during operation is the endpoint that retrieves all fragments for a specific transcript in order, which allows for fetching the existing fragments when reopening the same transcript without depending on the browser storing these

5.5.4 Further training

The corrected data stored in the database can be used for further training, optionally with a fallback to the original data. A basic script to retrieve this from the database is located alongside the backend code. This script exports a CSV of audio file paths and corresponding transcripts in the same format needed by the training code.

Chapter 6 - Testing

While the previous chapter detailed the iterative development and continuous internal evaluation of the model, this chapter focuses on the formal verification of the final prototype. To prove that the system meets the rigorous demands of a clinical environment, it must be validated against the functional and non-functional requirements established in Chapter 2.

6.1 Testing plan

The testing plan consists of two parts. The first part was already briefly mentioned in the previous chapter and is about testing the performance of the ASR model. The second part is about the performance of the real-time prototype.

6.1.1 Plan to test the ASR model iterations

To objectively determine the final accuracy of the trained ASR model, the completely untouched "test" dataset that was isolated during the initial data ingestion phase is relied upon. This test set contains approximately one hour of spoken audio. Crucially, this data features radiologists whose voices the model has never encountered during training, providing a reliable metric for real-world generalization.

To ensure a fair evaluation, the transcripts for this test set were passed through the exact same automated normalization pipeline used for the training data. However, because these original transcripts were generated by the legacy G2 system, they inherently contained historical recognition errors. To calculate an accurate Word Error Rate (WER) and Character Error Rate (CER), the evaluation data could not just be normalized; it had to be perfect. Therefore, the transcripts for this entire hour of data were manually reviewed and corrected, fixing any obvious legacy mistakes to establish a flawless "ground truth." By comparing the model's transcription of the test audio against this manually verified ground truth, the value added by the domain-specific fine-tuning process can be definitively quantified. However, because this team lacks domain expertise in radiology, it should be noted that the manually reviewed transcripts may still contain subtle errors regarding complex medical jargon.

6.1.2 Plan to test the prototype

Testing the prototype in general is something that cannot really be done programmatically. Hence, manual testing is conducted in this phase. This testing consists of running the web application, starting the recording, and starting to talk into the microphone. Some medical jargon was created and spoken to the prototype. Then, the reliability of the output of the prototype was compared with what was really said. This tests if the prototype is working and if the trained ASR is actually transcribing medical jargon correctly.

The chunking strategy also requires a lot of manual testing. For determining what the best values are for when to split up silence, the ASR is used while trying to speak as uninterrupted as possible. This is repeated multiple times, each time with different values for the minimum length of a silence before a cut is made, and how low the probability of speech must be before it is seen as silence. The parameters range from ≤ 0.05 to ≤0.2 for the probability of the silence and 2 to 10 seconds for the minimal length of the silence. Whenever the fallback case of the automatic chunking is hit after thirty seconds, the test is continued by speaking uninterrupted for a shorter amount of time, giving it the chance to split on the silence. There is also a baseline to which the results can be compared, a baseline in which splitting is not based on silence thresholds, but solely on the 30-second required split. Finally, once the manual test has concluded, the output of the ASR is compared with the text that was read and the baseline.

Additionally, unit tests have been written for parts of the codebase, with a specific focus on the audio splitter and the postprocessing, since these were the components with the most complex algorithms. The ASR system is not covered under these unit tests, since it is hard to deterministically test it.

Testing that fragments were properly stored in the database was done in a manual way by running a transcription, and then refreshing the page so the fragments are retrieved from the database again. It can then be checked that the contents of the page did not change.

6.2 Test results

6.2.1 Results over different datasets

The iterative process focused on two parallel tracks: continuously improving the quality of the training data and optimizing the model's training parameters. After creating the text normalization rules and finalizing the data pipeline, systematic hyperparameter sweeps across the different dataset sizes were conducted.

To find the optimal training configuration without introducing too many variables, most parameters were kept constant, exclusively varying the learning rate and the number of training epochs. The goal of these sweeps was to identify the exact combination that yielded the highest accuracy on the evaluation data without causing the model to overfit. The tables below detail the runs executed on each finalized dataset, tracking the performance on the isolated evaluation set. Note that many runs were done for each dataset; only the three most successful runs are included. Results were based on a single run on the evaluation set. You can find the full results for each dataset in appendix E.

Dataset (hours)	Base model	Number of epochs	Learning rate	WER	CER

1	Parakeet	10	1.3e-5	0.333	0.16
5	Parakeet	20	1e-4	0.191	0.079
10	Parakeet	20	1e-4	0.164	0.075
13	Parakeet	25	1e-4	0.155	0.070
22	Parakeet	20	1e-4	0.146	0.073

Table 5: A summary of the best run for each dataset.

As shown in the summary table, the model trained on the 13-hour dataset achieved a WER of **15.5%** and a CER of **7.0%**. While this slightly underperforms the results obtained from the larger 22-hour dataset in terms of WER, it represents the most balanced performance across the metrics. A detailed analysis of why the 13-hour configuration yielded these superior results compared to the larger dataset can be found in the discussion.

6.2.2 Comparison with basemodels

The initial choice of a base model was based on an older evaluation set. This was a set with only one speaker and was also a little smaller, but it was the only available data at the time. Given that a higher quality evaluation set is now available, the models will be compared against base models evaluated against the same data.

Model	**WER**	**CER**
parakeet-tdt-0.6b-v3	.427	.168
whisper-large-v3	.465	.252
whisper-large-v2	.495	.234
canary-1b-v2	.517	.278
wav2vec2-xls-r-1b-dutch	.573	.220
The parakeet model	.155	.070

Table 6: A comparison of the finetuned and several out of the box models.

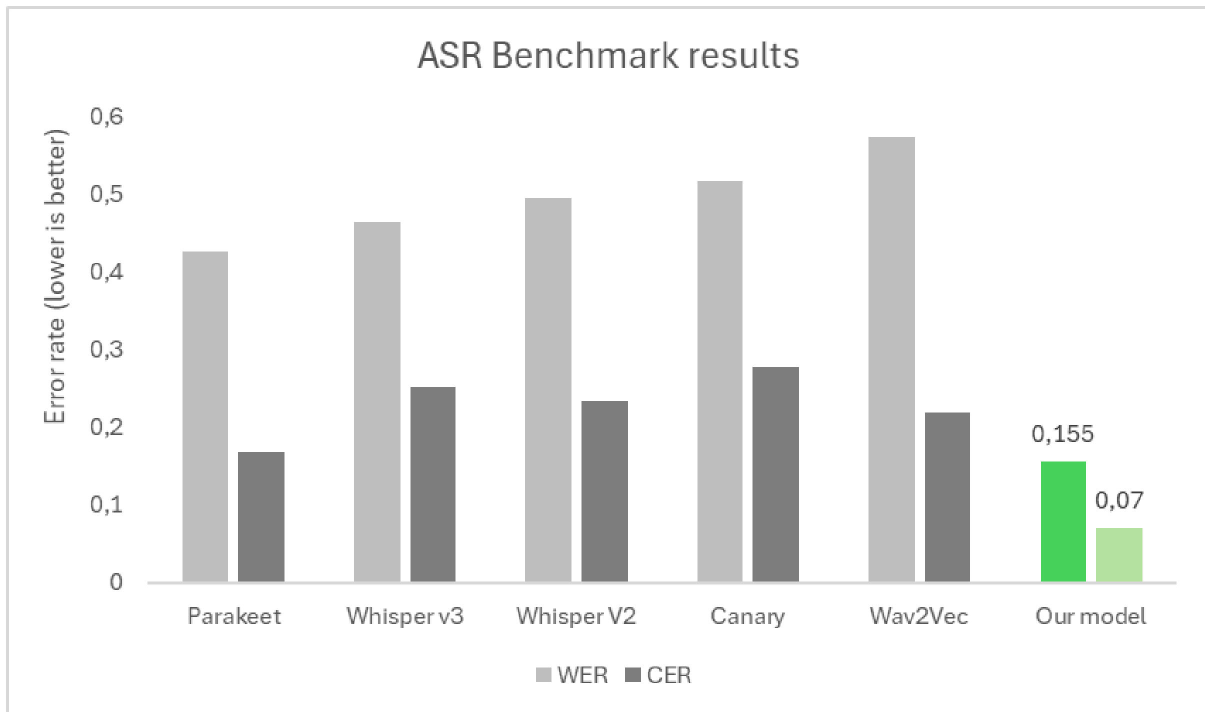


Figure 6: A visual representation of the comparison between the finetuned model and several out of the box models.

6.3 System performance and infrastructure

This section evaluates the technical efficiency of the prototype, ensuring it operates within the hardware constraints of the ZGT infrastructure and meets the speed requirements necessary for clinical use.

6.3.1 Real-Time Efficiency (RTF)

To ensure the system can keep pace with fast-paced radiology dictations, the Real Time Factor (RTF) of the inference pipeline was measured. The non-functional requirements dictate that the system must achieve an RTF below 0.5, meaning it must transcribe audio at least twice as fast as the speech is generated. During the testing phase using the parakeet-tdt-0.6b-v3 model, which utilizes an efficient Token-and-Duration Transducer architecture (Xu et al., 2023), a remarkable RTFx of 2154.22 was recorded. This exceptionally high inverse real-time factor translates to an RTF significantly lower than the defined limit of 0.5 (a little over a thousand times lower). This proves that the system processes audio almost instantaneously, providing a massive performance buffer that easily handles continuous, real-time dictation.

6.3.2 Audio chunking strategy and latency

A critical factor in real-time dictation is the strategy used to segment audio for processing, as it directly governs both transcription accuracy and end-to-end latency. Initially, the

system was constrained by a strict two-second latency requirement. However, forcing the model to rapidly process very short audio fragments deprived it of the necessary linguistic context, which negatively impacted the Word Error Rate. Consequently, the engineering approach was formally revised to prioritize context through a dynamic chunking strategy. By utilizing Silero Voice Activity Detection (VAD), the system now intelligently accumulates audio until a natural pause in speech is detected, or until a maximum fragment limit of thirty seconds is reached. To validate this updated strategy and ensure the system remained responsive under this new maximum latency threshold, extensive manual testing was conducted. By continuously dictating long reports into the web interface and timing the delay between a natural speech pause and the corresponding text rendering, the prototype's real-world performance was verified. These manual observations confirmed that the backend efficiently processes and returns even the largest thirty-second audio chunks well within the revised time limit, proving that the dynamic chunking strategy successfully balances crucial linguistic context with acceptable responsiveness.

Chapter 7 - Discussion

The design process of an ASR system for the radiology sector was a challenge. In this chapter, the 10 weeks of the Design Project for Technical Computer Science will be evaluated and discussed. The initial planning of the project will be discussed, as well as some complications that occurred in the timeline and how the team managed to put a successful ASR system (demo) on the shelf.

7.1 Domain ZGT audio vs. synthetic generation

A fundamental decision in the pipeline was the exclusive reliance on real clinical dictation data provided by ZGT. The primary advantage of this approach was the avoidance of acoustic domain shift; the model learned the specific cadence, background noise profiles, and lexical distribution unique to the target demographic. However, this created a heavy dependency on the legacy G2 system. Because the ground-truth text was derived from G2's post-processed output, it contained deterministic artifacts, such as punctuation substitutions and truncated newlines, that did not strictly align with the raw acoustic data.

Reversing these artifacts through the preprocessing scripts proved to be a lossy process. In retrospect, integrating synthetic data generation (e.g., using Text-to-Speech on verified medical corpora or the transcript text provided by ZGT) could have given interesting results. While synthetic audio struggles to capture the acoustic nuances of a specific radiologist, it provides perfectly aligned, artifact-free text-audio pairs. This would have eliminated the alignment discrepancies and unflagged omissions that eventually propagated into the training batches.

Despite these theoretical advantages, the idea of using synthetic data was rejected. The decisive factor was the nature of real radiological dictation. Radiologists dictate at an exceptionally rapid pace and employ an unnatural speech flow to continuously interject explicit formatting commands (e.g., "punt," "nieuwe alinea"). It became clear that text-to-speech engines did not replicate this nature of speech very well. Furthermore, utilizing purely synthetic audio for specialized clinical ASR fine-tuning remains a largely unproven methodology. The risk of the model overfitting to an artificial acoustic artificial sound was deemed too high. Therefore, accepting the engineering burden of lossy preprocessing was a trade-off to ensure the model accurately mapped the true behavior of its end-users.

7.2 Scalability across departments

System-level requirement FR-SM-6 mandated that the system be extensible to other medical fields without requiring a complete redesign. Architecturally and procedurally, this requirement was fulfilled. The modular backend and centralized state management successfully decoupled the inference engine from the specific model weights. Furthermore,

the developed preprocessing and training pipeline is fully reusable. As long as a new department (such as cardiology or neurology) provides paired audio and transcript data containing similar structural nuances, specifically the formatting quirks of the legacy G2 system, the existing methodology can be applied directly to train a new model.

The core limitation, however, lies in the model's adaptability. A generalized medical ASR capable of adapting to new clinical departments out of the box could not be realized. The current Parakeet model is highly specialized for radiology; exposing it to distinct pathologies zero-shot would result in a very high WER. Therefore, extending the system is not a simple "one-click" deployment but rather a data-acquisition hurdle. It strictly requires the collection of a new domain-specific corpus and the compilation of fresh abbreviation dictionaries.

Once that new data is secured, it can seamlessly be processed through the established pipeline to yield a department-specific model. However, it should be noted that naive sequential fine-tuning of the existing model on a new dataset risks "catastrophic forgetting" of the original radiology domain (Luo et al., 2023). Therefore, the base parakeet model would be needed as the model for training. Thus, while the software infrastructure and training methods are scalable, there will not be one model for multiple departments.

7.3 The impact of delayed data acquisition

A significant logistical constraint during this project was the delayed arrival of the primary training and evaluation data. The first batch of clinical dictations was not delivered until week 6 of the 10-week development cycle. This drastically compressed the timeline available for empirical evaluation and model fine-tuning. Prior to receiving this data, the preprocessing and normalization pipelines had to be designed "blind", relying on assumptions about how the G2 system formatted its output. When the actual data arrived, unforeseen formatting quirks, such as leading and trailing newline deletions, required rapid, reactive refactoring of the data ingestion scripts. This delay ultimately limited the number of hyperparameter sweeps that could be executed, meaning the 13-hour model, while highly accurate, might still be sub-optimal compared to what could have been achieved with a longer experimental runway.

7.4 Data Quality vs. Quantity

An unexpected finding in the results was that the 13-hour dataset yielded a model with a lower Character Error Rate (7.0%) than the larger 22-hour dataset (7.3%). In machine learning, it is generally assumed that more data leads to better generalization; however, this assumption did not extend to the results that we were seeing.

Upon closer inspection, it appears that the additional 9 hours of data introduced a significant volume of samples where these commands were inconsistently labeled. Due to the legacy G2 system's quirks, many instances of these spoken commands remained as formatting artifacts (like actual line breaks) rather than the literal text strings the model was

being trained to predict. Consequently, the 22-hour model began to "learn" to drop these words altogether, a known consequence of training on datasets with high label noise (Frénay & Verleysen, 2014).

Because these commands are high-frequency tokens in this use case, these omissions penalized the CER. Due to the delayed data acquisition (discussed in Section 7.3), a critical "time-to-compute" bottleneck was encountered; the iterative process of refining the detection scripts and re-running the extensive training pipeline for the 22-hour set was not feasible within the remaining project window. Therefore, the 13-hour model was selected as the optimal deliverable, since a model that consistently drops or gets punctuation commands wrong is not acceptable for professional use.

7.4 Latency vs. accuracy trade-off

A strict non-functional requirement (NFR-4) initially dictated that the system must display partial transcripts with an end-to-end latency of less than two seconds. During the early development of the real-time dictation prototype, an attempt to enforce this was made by using fixed, aggressively short audio chunking. However, this approach severely degraded the system's Word Error Rate (WER). Modern end-to-end models like Parakeet rely heavily on bidirectional linguistic context to accurately decode homophones, resolve sentence boundaries, and infer complex medical jargon.

By forcing the model to evaluate isolated, 1-to-2-second acoustic fragments, it was effectively starved of this necessary context. Consequently, the strict 2-second latency constraint was consciously abandoned. Instead, as detailed in the system performance evaluation, the system pivoted to a dynamic chunking strategy driven by Voice Activity Detection (VAD). Waiting for a natural pause in speech or utilizing a larger buffer up to 30 seconds inherently increased the end-to-end latency. However, this trade-off was deemed perfectly acceptable to the client.

7.5 Re-evaluating containerized deployment

From an infrastructure perspective, several maintainer user stories (US-M1) and functional requirements (FR-SC-4) aimed to have the final ASR system deployed as a fully containerized application (e.g., using Docker). This requirement was ultimately scoped out of the final deliverable. The primary motivation for containerization was to ensure a reproducible, portable installation.

However, as the project progressed, it became evident that forcing containerization introduced unnecessary architectural overhead for a prototype. Because the target deployment environment was a homogenous, highly specific hardware setup, ZGT's on-premise NVIDIA DGX Spark cluster, the broad portability benefits of Docker were marginalized. Furthermore, configuring Docker images for seamless GPU pass-through via

the NVIDIA Container Toolkit added an abstraction layer that complicated debugging. Ultimately, it was decided with ZGT that delivering a modular, well-documented codebase with explicit environment configurations satisfied the core maintainability goals.

7.6 Acceptability of the resulting WER

Non-Functional Requirement 13 (NFR-13) stipulated that the system must achieve a clinically acceptable Word Error Rate (WER). While reducing the WER to 15.5% (and the Character Error Rate to 7.0%) is a significant machine learning accomplishment compared to the zero-shot baselines, the gap between "algorithmic success" and "clinical readiness" warrants critical discussion. In practical terms, a 15.5% WER implies that roughly 3 out of every 20 words are transcribed incorrectly. This error rate inherently means the system still leans heavily on the radiologist to act as an active proofreader to catch dangerous clinical errors, though this is also the case in the legacy G2 system.

Despite this, ZGT evaluated the 15.5% WER as "acceptable" for a prototype, although acknowledging it does not yet surpass the performance of their legacy G2 system. However, the true value of deploying this prototype lies not in immediately beating G2, but in breaking the hospital's dependency on it. As discussed regarding the data pipeline, relying on G2's black-box post-processing severely hampered the training. Deploying this prototype, even at its current accuracy, introduces a native, transparent feedback loop. By capturing raw audio paired with the radiologist's direct manual corrections, the system will immediately begin generating perfectly aligned, artifact-free training data. Therefore, this 15.5% WER should not be viewed as the system's ceiling, but rather as the necessary threshold to activate a continuous learning pipeline that will rapidly drive future improvements.

Chapter 8 - Conclusion

Over the past 10 weeks, this design project set out to develop a functional prototype of an open-source, on-premise Automatic Speech Recognition (ASR) system tailored for radiologic dictations at Hospital Group Twente (ZGT). The objective was to prove that an open-source model could be fine-tuned to transcribe dense medical Dutch safely, locally, and with acceptable clinical accuracy, thereby paving the way to break dependency on expensive proprietary solutions.

8.1 Reflection on functional requirements

8.1.1 Must-level requirements

Must (FR-UM-Rx / FR-UM-Mx) User level

The radiologist ***must*** be able to

- ? access the dictation interface via a web browser (US-R1)
- ? start a session using a “Start Dictation” control (US-R1)
- ? see the transcript appear in real time (US-R2/US-R5/US-R9)
- ? review and correct the text generated by the ASR (US-R2/US-R5)
- ? insert punctuation and formatting using voice commands (US-R8)
- ? toggle between a correction mode and an addition mode (US-R11)

The maintainer ***must*** be able to:

- ? start and verify the ASR service (US-M1)
- ? perform a basic validation test after deployment (US-M1)
- ~ train the ASR to be expandable to other fields (US-M2)
- ? run the system on the ZGT infrastructure (US-M5)

Must (FR-SM-x) System level

The system ***must*** be able to:

- ? initialize audio capture (US-R1)
- ? maintain the session state (idle/dictating) (US-R1)
- ? accurately recognise radiology specific terminology and abbreviations (US-R4)
- ? accurately recognise specific voice commands for punctuation (US-R8)
- ? continuously show the dictated text in the web interface (US-R5/US-R9)
- × be extended to work for other medical fields without retraining the whole model (US-M2)
- ? run on the ZGT infrastructure (US-M5)
- ? distinguish between text that was generated by it and potentially corrected by the user, and text that was manually added by the user (US-R11)

The prototype successfully fulfills nearly all critical "Must-level" requirements. It delivers a fully functional, real-time dictation interface for radiologists, complete with medical terminology recognition, voice-command formatting, and manual correction toggles. Furthermore, the system operates entirely locally on ZGT's hardware, ensuring strict data privacy.

The only unmet system requirement is extending the model to new medical fields *without* retraining. While the software architecture itself is highly adaptable, the Parakeet AI model is strictly specialized for radiology. Expanding to other departments will require gathering new domain-specific data and retraining the base model to maintain clinical accuracy and prevent catastrophic forgetting.

8.1.1 Should-level requirements

Should (FR-US-Rx / FR-US-Mx) User level

The radiologist ***should*** be able to:
- ⍰ receive a visual confirmation that the dictation process has started (US-R1)

The maintainer ***should*** be able to:
- × add medical fields to the model selection (US-M2)
- ⍰ restart the system with updated configuration (US-M2)
- ⍰ register a new ASR model version (US-M3)
- ⍰ replace the currently deployed model (US-M3)
- ⍰ run validation benchmarks (US-M3)
- ⍰ have access to complete technical documentation (US-M6)

Should (FR-SS-x) System level

The system ***should*** be able to:
- ⍰ have a modular architecture such that individual components can be updated, replaced or restarted independently. (US-M3)

The prototype successfully meets nearly all "Should-level" requirements. The user interface provides clear visual confirmation during active dictations , and the system's modular architecture allows maintainers to seamlessly update configurations, swap model versions, run benchmarks, and access full documentation.

The only omitted feature is the ability to select different medical fields. Because the model is currently specialized solely for radiology , expanding to other departments requires training entirely new models rather than simply toggling a setting in the interface.

8.1.2 Could-level requirements

Could (FR-UC-Rx / FR-UC-Mx) User level

The radiologist ***could*** be able to
- × see words flagged by the model as potential errors (US-R2)
- ✓ use abbreviations when they please (US-R6)

The maintainer ***could*** be able to
- × deploy the ASR system using containerized infrastructure (US-M1)
- ✓ make necessary adjustments to the system such that implementation with other systems is possible (US-M4)

Could (FR-SC-x) System level

The system ***could*** be able to

- × show words flagged by the model as potential errors (US-R2)
- × write out any abbreviations as the complete medical term (US-R6)
- × show a timer to show how long a dictation is in progress (US-R7)
- × be deployed in a containerised format (US-M1)
- ✓ calculate Word Error Rate automatically (US-M8)
- × calculate latency automatically (US-M8)
- × calculate real time factor automatically (US-M8)

Most "Could-level" features were omitted to prioritize core ASR performance. Minor UI additions, such as error-flagging and dictation timers, were excluded. On the technical side, containerized deployment (e.g., Docker) and external system integration frameworks were scoped out to avoid unnecessary architectural overhead on ZGT's specific hardware. Finally, automated calculation of latency and RTF was left out of this iteration, because it turned out not to be a factor that was necessary to measure.

8.2 Reflection on non-functional requirements

- □ The system must be able to transcribe words twice as fast as the input is generated, meaning the RTF has to be below 0.5 (US-R3)
- □ The system must be able to dictate in sessions of at least 60 seconds at once (US-R7)
- □ The system shall run on local GPU infrastructure. (US-M7)
- × The system must display partial transcripts with an end-to-end latency lower than 2 seconds for streaming dictation. (US-R1, US-R3, US-R9)
- □ The system shall sustain the performance targets for dictation sessions of at least 15 minutes without degradation. (US-R7)
- □ The system shall support at least 1 concurrent session, and shall be designed to scale to multiple sessions. (US-R10)
- □ The system shall operate without external cloud APIs and without sending any data outside ZGT infrastructure.
- □ If the WebSocket connection drops, the system shall allow reconnection within 10 seconds and continue dictation with clear user feedback. (US-R1, US-R9)
- □ If the ASR backend fails, the frontend shall show an error state and offer restart without losing already received text. (US-R1)
- × The system shall be deployable via container with a documented, repeatable procedure. (US-M1)
- □ All model versions and configs shall be versioned and able to be rolled back. (US-M3)
- □ The system shall expose basic health checks and log key events (session start/stop, model version, errors).
- □ From opening the web interface, starting dictation shall require no more than 2 actions.
- □ Dictation should be able to start no more than 1 second after pressing the button.
- □ The UI shall always indicate whether the system is listening, transcribing, or paused.

The prototype successfully meets almost all non-functional performance, stability, and security constraints. The system transcribes exceptionally fast, easily exceeding the Real-Time Factor (RTF) requirement. It operates reliably for extended sessions on local GPU infrastructure without relying on external cloud APIs, strictly preserving data privacy. Furthermore, it achieved a functional Word Error Rate (WER) and features error handling for connection drops.

Three specific requirements were intentionally omitted from this iteration. First, the strict two-second latency limit was abandoned in favor of a dynamic chunking strategy. While this slightly increases response times, it provides the model with much better linguistic context and improves overall accuracy. Second, as previously mentioned, containerization via Docker deployment was scoped out to avoid unnecessary architectural overhead. Finally, data retention features, specifically configurable auto-deletion logs for stored training data, were not implemented in the prototype.

Chapter 9 - Future Improvements

Of course, the time limit of this project was 10 weeks. Therefore, this chapter refers to some possible future improvements of the system. Improvements are possible in all parts of this project, so they will be discussed per part.

9.1 Data

Data is one of the most important aspects of this project, and yet, one of the biggest areas of improvement for the future. The data received for training was flawed in two major ways: the audio files occasionally missed a few seconds of speech, even though it was in the transcript, and speech in the audio did not always perfectly match the transcript due to various smaller issues. This can be fixed in part by preprocessing the transcripts, but this is by no means perfect (yet).

Currently, the preprocessing step handles text normalization only; it replaces punctuation by words (“.” becomes “punt”) and replaces some commonly used abbreviations by the full form, but it does not touch the audio side of the training pairs at all. This means that any mismatch originating in the audio, such as missing seconds of speech or a speaker deviating from the written transcript, passes through into training without any correction or flagging.

This can be improved in a few ways. Introducing a forced alignment step would allow the pipeline to detect segments where the audio and transcript fall out of sync, either trimming or flagging them rather than passing corrupted pairs silently into training. Furthermore, it could be beneficial to manually check data fragments that are flagged by a baseline model due to very low agreement with the original.

Beyond refining the existing dataset, the inclusion of data augmentation techniques offers a promising avenue for increasing model robustness. During the final stages of the project, experiments with augmentation were conducted, specifically targeting audio perturbations such as pitch shifting, speed variation, and the addition of background noise. While these initial tests did not yield a significant improvement in the WER, the working theory is that this was primarily due to the limited time available to perform an exhaustive search for the optimal parameter settings, and there might be room for future improvement.

Another approach to improve data quality would be to produce a completely new set of data. Currently, the data is acquired by relying on an external STT system’s output for a given audio fragment. The text fragments generated by this external system are not always correct, and there is no access to intermediate forms of the generated text (before postprocessing). This is the main driver of the current problems. Using the finetuned STT model, however, data from each step within the pipeline could be recorded, and it would also be easier to implement ways to make sure the audio and the text match more exactly.

9.2 Second layer after ASR model

The model output could possibly be improved in various ways to lower the WER even further. The postprocessor can always be improved further to catch more tricky cases to replace, remove, or add punctuation. Furthermore, some form of spellchecker or LLM verification could probably improve the WER again. An attempt at using a simple spellchecker using a specialized dictionary was made, but there was not enough time left to achieve a significant improvement.

The spell checker's improvements were largely canceled out by three problems. First, its edit distance algorithm couldn't reliably distinguish malformed medical terms from similar common words, causing "thora" to match "there" instead of "thorax". Second, since the checker operated on individual tokens, split words like "pneumo thorax" were corrected independently rather than merged, producing errors like "pseudo thorax". Third, the radiological dictionary was incomplete, missing inflected and plural Dutch forms like "longarteriën", leaving common variants vulnerable to wrong substitutions. The net result was only a 0.2% WER reduction, which didn't justify further development in the short time that was left.

The framework that was utilized, Nvidia NeMo, provides native support for integrating external language models during decoding to enhance the base ASR architecture. An n-gram language model trained via KenLM (Heafield, 2011) using the entire available transcript corpus was also experimented with. However, this approach yielded only very tiny WER improvements. This is likely because the text dataset was too small to provide meaningful linguistic context beyond what the end-to-end ASR model had already learned. Though this approach did not lead to significant improvements, an n-gram language model is still expected to be of value for future system enhancements. While domain-specific word boosting was also considered, it was not explored further. Relying on manually curated lists of terms to boost is a somewhat rigid and naive approach to error correction, particularly when lacking explicit prior knowledge of the target vocabulary.

Instead, in this team's opinion, the biggest improvement would come from using a smart, context-aware Large Language Model (LLM) to review and fix the text. This LLM could be trained on past, accurate reports, so it learns the specific vocabulary and common transcription mistakes. To make it even better, the LLM could be fed the top few "guesses" the speech system came up with, rather than just its final answer. By looking at the context of the whole document alongside those alternative guesses, the LLM could intelligently choose the right words to create a much more accurate final transcript.

Appendices

Appendix A - Source Code and Glossary

Source Code

A copy of the source code is available on request. Please contact Reinder Grondsma by email via *r.grondsma@student.utwente.nl*

Glossary

Throughout the reading of this report, the reader may notice the use of various terms specific to the problem domain, which might be ambiguous. Below are brief introductions to the terms in question.

AI (Artificial Intelligence)
Systems designed to perform tasks that typically require human intelligence, such as speech recognition, pattern detection, and decision-making.

ASR (Automatic Speech Recognition)
Software that converts spoken language into written text automatically.

Benchmarking
The process of measuring system performance (e.g., WER, latency, RTF) against predefined standards or previous versions.

Containerization
A deployment method where software runs inside isolated environments (e.g., Docker containers) to ensure reproducibility and portability.

Docker
A platform used to build, deploy, and manage containerized applications.

End-to-End Latency
The time between a spoken word and the moment it appears in the transcript on the user interface.

Fine-Tuning
The process of retraining a pre-trained machine learning model on domain-specific data (e.g., radiology reports) to improve accuracy.

GDPR (General Data Protection Regulation)
European regulation governing data protection and privacy of personal data.

H100 (NVIDIA H100 GPU)
High-performance GPU used for AI workloads. In this project, it represents the hardware resource constraint.

Health Check
An automated system test that verifies whether the ASR service is operational and responsive.

HiX
Hospital information system used by ZGT for logging and managing patient data.

Hotword Boosting
A speech recognition technique that increases the probability of correctly recognizing specific domain-related words or phrases.

Inference
The process where a trained model generates predictions (e.g., converting speech into text).

Maintainer
Technical operator responsible for deployment, configuration, monitoring, and retraining of the ASR system.

Model Artifact
Packaged trained model file ready for deployment.

NVIDIA Container Toolkit
Software that enables GPU access within Docker containers.

Prototype
A first or preliminary version of a device or system from which other forms are developed

RTF (Real Time Factor) and RTFx
Ratio of processing time to audio duration.
RTF = 1 → 60 seconds audio takes 60 seconds to process.
RTF < 1 → Faster than real-time.
RTFx is the inverse of RTF.
RTF = 0.1 ⇒ RTFx = 1 / 0.1 = 10.

Stakeholder
A person with an interest or concern in something, especially a business.

Validation Dataset

A separate dataset is used to evaluate model performance after training.

WebSocket
Communication protocol enabling real-time bidirectional data exchange between frontend and backend.

WER (Word Error Rate)
A metric that measures transcription accuracy by comparing recognized words with the correct reference text

ZGT
Hospital organization where the system will be deployed and operated locally.

Appendix B - Traceability Matrix

In this matrix, every user story is listed exactly once. For each user story, every use case, every functional requirement, and every non-functional requirement that is associated with it has been listed.

User Story	Use Case	FR	NFR
US-R1	UC-R1, UC-R2	FR-UM-R1, FR-UM-R2, FR-US-R1, FR-SM-1, FR-SM-2	NFR4, NFR9, NFR10
US-R2	UC-R5	FR-UC-R1, FR-UM-R3, FR-UM-R4, FR-S-C1	-
US-R3	UC-R2	-	NFR1, NFR4
US-R4	UC-R4	FR-SS-4	NFR13
US-R5	UC-R6	FR-UM-R3, FR-UM-R4, FR-SM-5	NFR7
US-R6	UC-R4	FR-UC-R2, FR-SC-2	-
US-R7	UC-R1	FR-SC-3	NFR2, NFR5
US-R8	UC-R3	FR-UM-R5, FR-SM-4	-
US-R9	UC-R6	FR-UM-R3, FR-SM-5	NFR4, NFR9
US-R10	UC-R1	-	NFR6
US-R11	UC-R5	FR-UM-M6, FR-SM-8	-
US-M1	UC-M1	FR-UM-M1, FR-UM-M2, FR-UC-M1, FR-SC-4	NFR11
US-M2	-	FR-UM-M3,	-

		FR-US-M1, FR-US-M2, FR-SM-6	
US-M3	UC-M4, UC-M2	FR-US-M3, FR-US-M4, FR-US-M5, FR-SS-1	NFR12
US-M4	UC-M2	FR-UC-M2, FR-UW-R1, FR-UW-M1, FR-SW-1	-
US-M5	-	FR-UM-M4, FR-SM-7	NFR8
US-M6	-	FR-US-M6	-
US-M7	-	-	NFR3
US-M8	UC-M3	FR-SC-5, FR-SC-6, FR-SC-7, FR-SC-8	-

Table 7: A traceability matrix of the user stories, use cases, and requirements.

Appendix C - Diagrams

Sequence Diagram

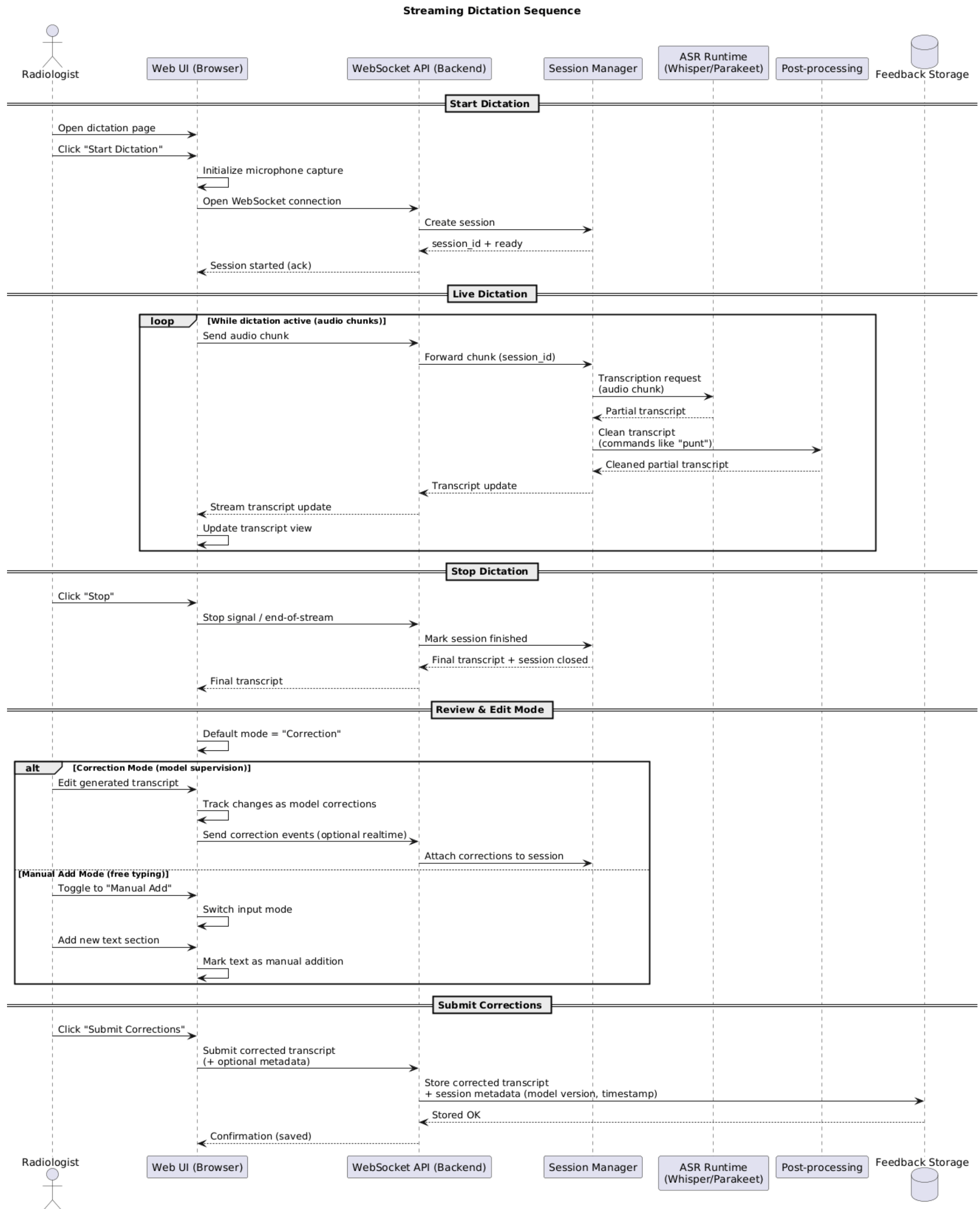


Figure 7: A sequence diagram of the complete system.

Appendix D - File Structure

Local Repository

```
root/
├── backend                # FastAPI server & Frontend
│   ├── app                # Backend: API, ASR adapters, Chuking/VAD logic
│   ├── scripts            # Backend scripts, primarily export for training
│   ├── tests              # Backend-specific unit and integration tests
│   ├── web                # HTML/JS/CSS frontend served at /app
│   ├── pyproject.toml     # Backend project metadata and dependencies
│   └── uv.lock            # Deterministic dependency lockfile for backend
└── training               # ASR training pipeline
    ├── config             # Domain-specific rules
    ├── src                # Core ASR Pipeline
    │   ├── preprocessing # Dataset cleaning, normalization, and splitting
    │   ├── test           # NeMo/Lightning training orchestration
    │   ├── train          # Model evaluation and benchmarking
    │   ├── utils          # Shared logging and manifest helpers
    │   └── config.py      # Centralized path and environment configuration
    ├── pyproject.toml     # Frontend project metadata, dependencies and CLI
entry points
    └── uv.lock            # Deterministic dependency lockfile for frontend
```

Shared Data Drive

```
/mnt/data/asr_ut/
├── all_raw_data/       # Raw .wav/.txt pairs
├── data_newline_detected/  # Raw .wav/.txt pairs with newline detection
├── datasets/               # Generated datasets (train, dev, test)
│   └── <dataset_name>/
├── evaluation_set/         # Manual 1h evaluation set
├── models/                 # Fine-tuned .nemo and .pt model checkpoints
├── outputs/                # Logs, predictions, and error metric TSVs
└── config/
    └── speaker_split.py    # Fixed train/dev/test speaker assignments
```

Appendix E - Test results

Training runs on the 1-hour dataset:

Base model	Epochs	Learning rate	WER	CER
Whisper	10	1.3e-5	.325	0.16
Whisper	10	1e-5	.341	0.18
Whisper	10	7e-6	.371	0.18

Training runs on the 5-hour dataset:

Base model	Epochs	Learning rate	WER	CER
Parakeet	20	1e-4	.191	.079
Parakeet	25	1e-4	.192	.081
Parakeet	25	7e-5	.199	.086

Training runs on the 10-hour dataset:

Base model	Epochs	Learning rate	WER	CER
Parakeet	30	3e-5	.177	.080
Parakeet	20	1e-4	.164	.075
Parakeet	50	3e-5	.164	.075

Training runs on the 13-hour dataset:

Base model	Epochs	Learning rate	WER	CER
Parakeet	30	1e-4	.158	.071
Parakeet	25	1e-4	.155	.070
Parakeet	20	1e-4	.158	.072

Training runs on the 22-hour dataset:

Base model	Epochs	Learning rate	WER	CER
Parakeet	20	1e-4	.146	.073

Parakeet	20	7e-5	.152	.073
Parakeet	20	5e-5	.156	.075

References

Bălan, D. A., Ordelman, R. J. F., & van den Heuvel, H. (2024). *A systematic benchmark for Dutch automatic speech recognition* [Poster]. University of Twente. https://research.utwente.nl/en/publications/a-systematic-benchmark-for-dutch-automatic-speech-recognition

European Parliament and Council of the European Union. (2016). *Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation)* (Publication L 119/1). Official Journal of the European Union. https://eur-lex.europa.eu/eli/reg/2016/679/oj

Frénay, B., & Verleysen, M. (2014). Classification in the presence of label noise: a survey. *IEEE Transactions on Neural Networks and Learning Systems*, *25*(5), 845–869. https://doi.org/10.1109/TNNLS.2013.2292894

Heafield, K. (2011). KenLM: Faster and smaller language model queries. In *Proceedings of the Sixth Workshop on Statistical Machine Translation* (pp. 187–197). Association for Computational Linguistics. https://aclanthology.org/W11-2123

Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., & Chen, W. (2021). *LoRA: Low-rank adaptation of large language models*. arXiv. https://doi.org/10.48550/arXiv.2106.09685

Hugging Face. (n.d.). *Preprocessing an audio dataset*. Hugging Face Audio Course. Retrieved April 14, 2026, from https://huggingface.co/learn/audio-course/en/chapter1/preprocessing

IBM Research. (2023). *Granite foundation models*. arXiv. https://arxiv.org/abs/2310.02426

Kuchaiev, O., Li, J., Nguyen, H., Hrinchuk, O., Leary, R., Ginsburg, B., Kriman, S., Beliaev, S., Lavrukhin, V., Cook, J., Castonguay, P., Popova, M., Huang, J., & Cohen, J. M. (2019). *NeMo: A toolkit for building AI applications using Neural Modules*. arXiv. https://doi.org/10.48550/arXiv.1909.09577

Luo, Y., Yang, Z., Meng, F., Li, Y., Zhou, J., & Zhang, J. (2023). *An empirical study of catastrophic forgetting in large language models during continual fine-tuning*. arXiv. https://arxiv.org/abs/2308.08747

NVIDIA. (2026). *DGX Spark hardware overview*. https://docs.nvidia.com/dgx/dgx-spark/hardware.html

NVIDIA NeMo Team. (2025). *Canary-1B-v2 & Parakeet-TDT-0.6B-v3: Efficient and high-performance models for multilingual ASR and AST*. arXiv. https://arxiv.org/abs/2509.14128

Perezhohin, Y. (2024). *whisper-large-v3-cv-high-mixed-nl* (Version from 2024) [Machine learning model]. Hugging Face. https://huggingface.co/yuriyvnv/whisper-large-v3-high-mixed-nl

Picovoice. (2026, January 23). *Voice activity detection (VAD): The complete 2026 guide to speech detection*. https://picovoice.ai/blog/complete-guide-voice-activity-detection-vad/

Povey, D., Ghoshal, A., Boulianne, G., Burget, L., Glembek, O., Goel, N., Hannemann, M., Motlicek, P., Qian, Y., Schwarz, P., Silovsky, J., Stemmer, G., & Veselý, K. (2011). The Kaldi speech recognition toolkit. In *2011 IEEE Workshop on Automatic Speech Recognition & Understanding (ASRU)*. IEEE. https://www.danielpovey.com/files/2011_asru_kaldi.pdf

Radford, A., Kim, J. W., Xu, T., Brockman, G., McLeavey, C., & Sutskever, I. (2022). *Robust speech recognition via large-scale weak supervision*. arXiv. https://doi.org/10.48550/arXiv.2212.04356

Ramírez, S. (2024). *FastAPI* (Version 0.115.4) [Computer software]. https://github.com/tiangolo/fastapi

Silero Team. (2025). *Silero VAD: Pre-trained enterprise-grade voice activity detector (VAD)* (Version 6.2) [Computer software]. https://github.com/snakers4/silero-vad

XL8. (n.d.). *Filter out inaccurate media data for better ASR*. https://www.xl8.ai/blog/filter-out-inaccurate-media-data-for-better-asr

Xu, H., Jia, F., Majumdar, S., Huang, H., Watanabe, S., & Ginsburg, B. (2023). *Efficient sequence transduction by jointly predicting tokens and durations*. arXiv. https://doi.org/10.48550/arXiv.2304.06795

Conneau, A., Ma, M., Khanuja, S., Zhang, Y., Axelrod, V., Dalmia, S., Riesa, J., Rivera, C., & Bapna, A. (2022, May 25). FLEURS: Few-shot Learning Evaluation of Universal Representations of Speech. arXiv.org. https://arxiv.org/abs/2205.12446?

Huddy, S. (2025, April 15). Choosing Between Synthetic and Real Speech Data: Advantages vs Disadvantages. Way With Words. https://waywithwords.net/resource/choosing-synthetic-and-real-speech-data/

Ko, T., Peddinti, V., Povey, D., Seltzer, M. L., & Khudanpur, S. (2017). A study on data augmentation of reverberant speech for robust speech recognition. IEEE Conference Publication. https://ieeexplore.ieee.org/abstract/document/7953152

Mozilla Common Voice. (n.d.). https://commonvoice.mozilla.org/en