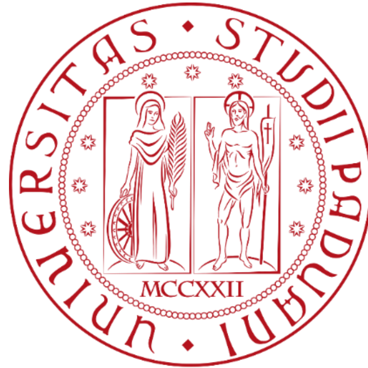
UNIVERSITÀ DEGLI STUDI DI PADOVA



DIPARTIMENTO DI INGEGNERIA DELL’INFORMAZIONE

CORSO DI LAUREA IN
INGEGNERIA INFORMATICA

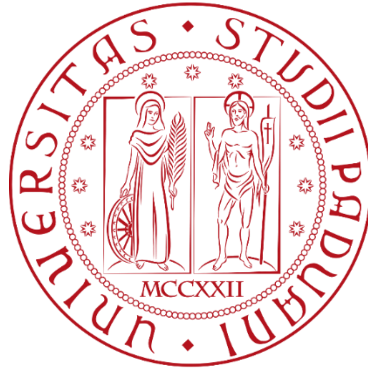
CONVERSARE CON I ROBOT: COME CREARE ASSISTENTI LLM PER CAPIRE E USARE SISTEMI AUTONOMI

RELATORE:
Prof. Damiano Varagnolo

LAUREANDO:
Francesco Pizzato

ANNO ACCADEMICO 2024-2025
Data Di Laurea: 22-09-2025

UNIVERSITY OF PADUA



DEPARTMENT OF INFORMATION ENGINEERING

BACHELOR’S DEGREE IN
COMPUTER ENGINEERING

CONVERSING WITH ROBOTS: BUILDING LLM ASSISTANTS TO UNDERSTAND AND UTILIZE AUTONOMOUS SYSTEMS

SUPERVISOR:
Prof. Damiano Varagnolo

CANDIDATE:
Francesco Pizzato

ACADEMIC YEAR 2024-2025
Graduation Date: 22-09-2025

TABLE OF CONTENTS

ABSTRACT 1
CHAPTER I: OVERVIEW OF THE NANOGPT PROJECT 2
1.1 Objectives of the Work 2
1.2 Problem Statement and Specific Objectives 3
1.3 Research Contributions 4
1.4 Structure of the Thesis 4
CHAPTER II: BACKGROUND CONCEPTS 6
2.1 Large Language Models LLMs 6
2.2 Software Tools Employed 7
2.3 Speech Recognition and Synthesis Technologies 8
2.4 Hardware and Computational Constraints 8
CHAPTER III: REQUIREMENTS AND USE CASES 10
3.1 Stakeholders and Use Scenarios 10
3.2 Functional Requirements 10
3.3 Non-Functional Requirements 11
3.4 Constraints and Assumptions 12
CHAPTER IV – SYSTEM ARCHITECTURE AND IMPLEMENTATION 14
4.1 Overview of the Architecture 14
4.2 Description of the NanoGPT_Marinello.py Code 15
4.3 User Interface 16
4.4 Training Phase and Checkpoint Management 17
4.5 Text Generation and Qualitative Evaluation 19
CHAPTER V – DATASET AND KNOWLEDGE INTEGRATION 21
5.1 Data Sources 21
5.2 Reprocessing and Balancing Procedures 21
5.3 Ethical and Legal Considerations 22
CHAPTER VI – SYSTEM EVALUATION AND LIMITATIONS 24
6.1 Training Metrics 24
6.2 Qualitative Performance 25
6.3 Discussion of Limitations 25
CHAPTER VII – GUIDELINES AND CONCLUSIONS 27
7.1 Guidelines for Future Development 27

7.2 Conclusions 28
CHAPTER VIII – SUPPORTING MATERIALS 30
8.1 Access to the Repository 30
8.2 Environment and Dependencies 30
8.3 Dataset 31
8.4 Checkpoints and Reproducibility 31
BIBLIOGRAPHY 33
NOTE ON THE THESIS TEXT 34

ABSTRACT

This thesis explores how to build conversational assistants based on Large Language Models — LLMs — to support the understanding and use of complex autonomous systems, such as robotic vehicles or intelligent boats.
The project starts from a first dataset of technical documents made available through the *BlueBoat* project, developed in collaboration with NTNU – Norwegian University of Science and Technology.
The aim is to design a pipeline for training, customizing, and effectively interacting with a domain-specific LLM. The final goal is to create an accessible and reusable system that helps human users interact effectively with autonomous systems.

CHAPTER I: OVERVIEW OF THE NANOGPT PROJECT

1.1 Objectives of the Work

The present work aims to develop a large language model — LLM — named *NanoGPT Marinello,* designed to support the research and development activities of the *Nautilus* project at the University of Padua, which is closely connected to the *BlueBoat* project of the Norwegian University of Science and Technology — NTNU.

The primary objective is the creation of a conversational assistant capable of answering technical questions related to the project, thereby facilitating the understanding of code, procedures, and implementation choices by students and researchers. In this way, the system seeks to reduce the learning curve, ease the integration of new team members, and provide documentation that is more accessible and dynamic compared to traditional static manuals.

The developed prototype integrates both text-based and voice-based interaction, allowing users to ask questions and receive answers either in written form or through speech synthesis. This feature is intended to make the assistant more versatile and usable in heterogeneous contexts, including experimental field settings.

The ultimate purpose of this thesis is therefore twofold: on the one hand, to demonstrate the technical feasibility of a specialized LLM; on the other, to provide a documented, replicable, and extensible platform that can serve as a solid foundation for future developments and enhancements by the research groups involved.

The code developed has been consolidated into the Python script *NanoGPT_Marinello.py*, which integrates all the main components of the system and will be described in the following chapters. This organization ensures the reproducibility of the work and facilitates its extension by future developers.

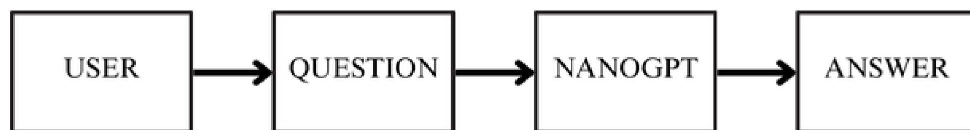


Figure 1.1.1 – Conceptual flow of interaction with NanoGPT: the user formulates a question, which is processed by the model to generate an appropriate answer.

1.2 Problem Statement and Specific Objectives

The development of an assistant based on large language models in a highly technical and specialized context such as the *Nautilus* project entails a number of significant challenges. The first difficulty concerns the need to train a small-scale model—specifically *NanoGPT Marinello*—on a limited dataset composed mainly of theses, reports, and documentation produced within the framework of the *BlueBoat* project at the Norwegian University of Science and Technology — NTNU. This entails dealing with issues such as the scarcity and heterogeneity of sources, the necessity of balancing the available data, and the requirement to ensure both coherence and quality in the generated responses.

Another critical aspect is the management of computational resources. The project was designed to operate even in environments with limited processing capacity, without relying on high-performance infrastructures. This constraint has guided architectural and methodological choices, influencing both the training and inference stages, with the aim of maximizing efficiency under resource-constrained conditions.

In addition to these challenges, there is the need to implement a multimodal interaction mode that integrates both text and voice input. This feature makes the system not only a tool to support software development but also a resource that can be employed in experimental field scenarios, where voice interaction may simplify use in operational conditions.

In lights of these considerations, the specific objectives of the thesis can be summarized as follows:

- Design and development of a prototype of an LLM, based on the NanoGPT architecture, in support of the *Nautilus/BlueBoat* project activities;
- Implementation of a complete pipeline for text and voice interaction, employing open-source tools for speech recognition and synthesis;
- Systematic evaluation of the feasibility of training and inference in resource-constrained environments, with the aim of demonstrating the possibility of building a specialized LLM from scratch;
- Provision of a clear, well-documented, and replicable platform that can serve as a solid foundation for students and researchers who will continue the development activity.

1.3 Research Contributions

The present thesis describes the activities carried out and the results achieved in the development of the *NanoGPT Marinello* system. In particular, the work has led to:

- Design and implementation of a large language model based on the NanoGPT architecture and developed using the *PyTorch* framework, specifically adapted to the requirements of the *Nautilus/BlueBoat* project;
- Construction of a dedicated dataset, derived from theses, reports, and technical documentation, reprocessed and structured in a question–answer format to effectively support model training;
- Integration of a multimodal interaction pipeline, encompassing both text and voice input, with speech recognition and synthesis, in order to make the assistant usable in diverse contexts, including experimental field activities;
- Development of a prototypical user interface, built with open-source tools, aimed at providing end users with immediate and intuitive access to the system;
- Experimental evaluation of the model, conducted through accuracy and latency metrics and accompanied by a critical analysis of the main limitations encountered;
- Drafting of guidelines for future development, intended to provide operational directions and methodological suggestions for students and researchers who will extend and improve the system in the future.

1.4 Structure of the Thesis

The thesis is organized into seven chapters, each addressing a specific aspect of the work carried out, following a logical progression that moves from the definition of the context to the development of the prototype and the presentation of future perspectives.

Chapter I – Overview of the NanoGPT Project: introduces the general objectives of the thesis, outlines the problems addressed, presents the main contributions, and provides an overall view of the work.

Chapter II – Background Concepts: provides a synthesis of the theoretical principles and essential tools required to understand the work, including an introduction to large language models, speech synthesis and recognition technologies, and the prototyping tools employed.

Chapter III – Requirements and Use Cases: defines the functional and non-functional requirements of the system, identifies the stakeholders and potential use scenarios, and discusses the relevant design constraints.

Chapter IV – System Architecture and Implementation: describes the structure of the model, the architectural choices made, and the organization of the code, while also illustrating the training and inference procedures, the user interface, and the audio pipeline.

Chapter V – Dataset and Knowledge Integration: presents the sources used for dataset construction, the procedures of reprocessing and balancing, and the ethical and legal considerations associated with data usage.

Chapter VI – System Evaluation and Limitations: analyzes the performance achieved by the prototype, describes the metrics adopted, and discusses the main critical issues encountered, with particular attention to aspects related to scalability and operation in resource-constrained environments.

Chapter VII – Guidelines and Conclusions: summarizes the results obtained, provides practical recommendations for extending the work, and outlines the future development perspectives of the project.

Chapter VIII – Supporting Materials: provides all the practical resources necessary to reproduce and extend the project, including access to the GitHub repository, setup of the software environment, details of the dataset, and management of checkpoints for training reproducibility.

CHAPTER II: BACKGROUND CONCEPTS

2.1 Large Language Models LLMs

Large Language Models — LLMs — represent one of the most significant innovations in the field of artificial intelligence. They are based on the Transformer architecture, introduced in 2017, which has progressively replaced traditional sequential approaches through the use of the self-attention mechanism.

Unlike recurrent networks, which process text step by step, Transformers analyze the entire sequence in parallel, assigning each element a “weight” according to its relevance to the context. This ability to model long-range relationships overcomes the limitations of earlier approaches and has enabled the construction of increasingly large and effective language models. The architecture is primarily composed of layers of self-attention, feed-forward networks, and normalization mechanisms, organized into blocks that are replicated multiple times depending on the scale of the model.

The main industrial-scale LLMs, such as Generative Pre-trained Transformer — GPT — and Bidirectional Encoder Representations from Transformers — BERT — are characterized by billions of parameters and require substantial computational resources for both training and inference. In applied research contexts, however, it is not always necessary to adopt models of such scale: smaller versions may be more suitable when datasets are limited and when available hardware imposes strict constraints.

Within this perspective lies *NanoGPT Marinello*, developed as part of the present work. Inspired by the GPT architecture, it adopts a character-level — char-level — approach, in which text is represented as sequences of individual characters rather than as sets of tokens obtained through tokenization algorithms. The char-level nature of the model results from the choices made during dataset preparation and training: the vocabulary was defined as the set of characters actually present in the collected documents, and training was conducted directly on sequences of these characters.

Although this approach is less common in large-scale models, it proves particularly suitable in scenarios such as the present one, characterized by small datasets and a specialized domain. It simplifies the vocabulary, reduces preprocessing complexity, and allows for more efficient training under resource-constrained conditions. [1] [2]

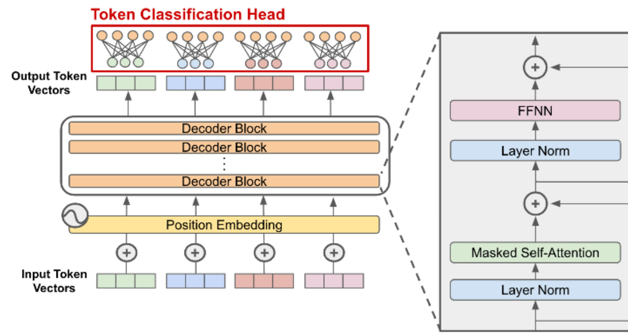


Figure 2.1.1 – Decoder-only Transformer architecture adopted by *NanoGPT Marinello*, consisting of stacked decoder blocks with masked self-attention, feed-forward layers, and a final classification head for next-token prediction. Source: C.R. Wolfe, "Decoder-Only Transformers: The Workhorse of Modern AI," Substack, 2023.

2.2 Software Tools Employed

The development of *NanoGPT Marinello* relied on open-source software tools that provided the project with flexibility, rapid implementation, and replicability. Among these, two played a central role: *PyTorch* and *Gradio*.

PyTorch is a widely used framework in both academic and industrial contexts for the implementation of machine learning models. It enables the modular and transparent development of neural networks, giving researchers direct control over both training and inference phases. The adoption of *PyTorch* made it possible to construct an architecture faithful to the GPT model and to adapt it to the specific needs of the project, without resorting to proprietary or closed solutions that would have required greater expenditure of time and resources.

Gradio was employed for the development of the user interface. It is a library that allows for the rapid creation of simple and interactive web interfaces, thereby facilitating the testing and demonstration of artificial intelligence models. Within the project, *Gradio* enabled the creation of a browser-based multimodal chat, making the assistant easily accessible from different devices.

The combined use of these tools significantly reduced development time while maintaining clarity, extensibility, and transparency of the code. Moreover, these libraries represent the fundamental prerequisites for reconstructing the prototype's development environment and therefore constitute the starting point for anyone wishing to reproduce or extend the work.

2.3 Speech Recognition and Synthesis Technologies

Speech recognition was implemented using *Whisper*, an open-source library developed by OpenAI and widely adopted in academic and research contexts. *Whisper* is characterized by its robustness in transcription and its ability to operate across multiple languages, while maintaining a satisfactory balance between accuracy and computational requirements. This technology was selected for its accessibility, comprehensive documentation, and adaptability to experimental scenarios with limited resources.

For speech synthesis, the *gTTS* — Google Text-to-Speech — library was employed, an open-source solution that converts the text generated by the model into spoken output. *gTTS* offers the advantages of being lightweight, replicable, and easily integrable into the prototype, thereby meeting the requirement of providing a system that is both usable and shareable within the research team. For the sake of reproducibility, it is therefore necessary to prepare a Python environment with support for *Whisper* and *gTTS*, which constitute the foundations of the speech pipeline.

Through this combination, *NanoGPT Marinello* takes the form of a multimodal assistant, capable of handling both voice and text input and delivering responses in both modalities. This increases the versatility of the system and facilitates its use even in field operations, where reliance on a keyboard may be less practical.

2.4 Hardware and Computational Constraints

Another relevant aspect concerns the constraints imposed by the available computational resources. Unlike large language models trained in cloud environments on high-performance infrastructures, *NanoGPT Marinello* was designed to operate even in low-capacity scenarios, without necessarily relying on dedicated GPUs or distributed computing systems.

This design choice had direct implications for both the architecture of the model and the training methodology. On the one hand, it was necessary to limit architectural complexity by prioritizing lightweight and optimized solutions; on the other, a character-level approach was adopted, which is better suited to small datasets and manageable with training sequences compatible with the available resources.

Despite the simplifications introduced, the project demonstrated the feasibility of developing an artificial intelligence system that can be generated, trained, and queried

using standard consumer technologies. Although this represents a scaled-down approach compared to large-scale models, it confirms the practicality and replicability of the prototype, in line with the operational needs of future developers.

CHAPTER III: REQUIREMENTS AND USE CASES

3.1 Stakeholders and Use Scenarios

The developed system, *NanoGPT Marinello*, was conceived to address heterogeneous needs and to involve different groups of stakeholders who, in various capacities, can benefit from its use. The main stakeholders can be identified as follows:

- Researchers and developers at the Norwegian University of Science and Technology — NTNU — involved in the *BlueBoat* project: they represent the primary group of users, as the dataset underpinning the system was built from theses and reports produced at the Norwegian university. This group benefits from more immediate and interactive access to accumulated knowledge, thereby optimizing research and development activities.
- Students and researchers of the *Nautilus* team — University of Padua: this group also benefits from the assistant, which facilitates the understanding of code, procedures, and technical documentation, reducing the learning curve and easing the integration of new members into the team.
- Faculty members and supervisors: the assistant can serve as a useful tool for evaluating progress, documenting activities, and providing didactic support to students engaged in the study and development of autonomous systems.
- End users in experimental contexts: in field scenarios characterized by practical constraints, the integrated voice interface allows interaction with the system without relying on a keyboard, thus extending its applicability beyond the laboratory setting.

The main use cases therefore lie within the domains of education and research support, with particular emphasis on knowledge transfer within the teams involved. These are complemented by operational experimentation contexts, in which the multimodal — text and voice — interaction mode ensures immediate usability even in practical situations.

3.2 Functional Requirements

Functional requirements define what the system must provide in order to meet the objectives of the project. In the case of *NanoGPT Marinello*, the main functionalities can be summarized as follows:

- Text-based interaction: the system must allow the user to formulate questions in natural language and receive coherent, context-aware responses in written form;
- Voice-based interaction: in addition to text input, the system must support voice input through automatic speech recognition — speech-to-text — and provide responses also in synthesized audio form — text-to-speech;
- Training on domain-specific datasets: the assistant must be trainable on targeted corpora built from theses, reports, and technical documentation, in order to specialize in the project's domain of reference.

These requirements ensure that the prototype fulfills its intended purpose: an interactive multimodal assistant capable of adapting to the needs of students and researchers engaged in the development of autonomous systems.

3.3 Non-Functional Requirements

In addition to the core functionalities, the system must comply with a set of non-functional requirements, which define how it should behave in order to ensure a satisfactory user experience and make the prototype effectively employable in research contexts. These requirements, fundamental to guaranteeing its overall effectiveness, can be summarized as follows:

- Portability: the system must ensure accessibility from heterogeneous devices through the launch of a local server accessible via browser. This enables users to interact with the assistant not only from the machine on which it is running, but also from other terminals—such as laptops, tablets, or smartphones—without requiring additional installations;
- Heterogeneity: the prototype must be able to run both on machines equipped with GPUs, leveraging hardware acceleration, and on CPU-only computers. This requirement ensures compatibility with diverse execution environments and broadens the range of possible applications;
- Performance: the system must maintain short response times, thereby enabling smooth interaction without excessive delays perceived by the user;
- Reliability and replicability: the implementation must be stable, featuring mechanisms for model saving and loading — checkpointing — as well as a modular code structure, thus facilitating the reproducibility of results;

- Usability: the interface must be simple and intuitive, accessible via browser, in order to support experimentation by users with varying levels of technical expertise;
- Openness and transparency: the system must rely on open-source tools, thereby reducing constraints related to proprietary licenses and allowing future developers to freely extend the work.

Taken together, these requirements ensure that the prototype is not only functionally correct, but also practically usable and replicable in research and development contexts.

3.4 Constraints and Assumptions

The development of *NanoGPT Marinello* was influenced by a series of technical and organizational constraints that limited the design choices available. The main ones can be summarized as follows:

- Limited dataset availability: the training material consists exclusively of theses, reports, and technical documentation from the *Nautilus/BlueBoat* project, resulting in reduced data variety and volume;
- Dependence on external libraries: the system relies on open-source components such as *PyTorch*, *Whisper*, *gTTS*, and *Gradio*, whose compatibility and availability directly affect overall stability;
- Connectivity requirement for speech synthesis: the use of *gTTS* requires an Internet connection to generate audio responses, limiting the full operability of the system in offline scenarios;
- Hardware limitations: performance is adequate on standard computers but not on resource-constrained platforms such as the *NVIDIA Jetson Nano*, a low-cost embedded board designed for artificial intelligence applications, which does not allow smooth and continuous use of the system.

Alongside these constraints, the project is based on several contextual assumptions which, while not strictly binding, have guided the design and use of the prototype:

- Basic user competencies: it is assumed that users possess basic computing skills sufficient to install dependencies, start the system, and interact with the interface;
- Availability of essential computing resources: it is assumed that execution takes place on a computer equipped with a modern CPU and adequate memory, with

the optional presence of a GPU to improve performance, although the latter is not indispensable;

- Circumscribed application domain: the system was designed to operate within the specific scope of the *Nautilus/BlueBoat* project; it is therefore assumed that interactions remain focused on this domain and do not extend to more general contexts.

Anyone seeking to reproduce or extend the system must regard these constraints and assumptions as reference operating conditions, in order to ensure the correct execution of the prototype and to avoid incompatibilities or non-compliant results.

CHAPTER IV – SYSTEM ARCHITECTURE AND IMPLEMENTATION

4.1 Overview of the Architecture

The architecture of *NanoGPT Marinello* was designed to integrate a specialized language model with a multimodal interaction pipeline and a simple, accessible interface, in order to make the system usable both in academic contexts and in experimental scenarios. The design followed principles of modularity and transparency, ensuring that individual components could be extended or replaced without compromising the overall functionality.

From a conceptual standpoint, the system can be described as consisting of four main interconnected blocks:

1. Dataset and training: the data, consisting of theses, reports, and technical documentation from the *Nautilus/BlueBoat* project, are preprocessed and organized into character sequences. These sequences feed into the model training phase, implemented in *PyTorch*.
2. Language model: at the core of the architecture lies *NanoGPT Marinello*, a char-level GPT model inspired by the Transformer architecture. The model receives character sequences as input and produces character-by-character predictions, generating coherent and context-aware text.
3. Interaction module: this component manages the processing of user requests. For text input, the sequence is passed directly to the model, while for voice input *Whisper* is used for transcription and *gTTS* for response synthesis. The output is then returned either in written form or, if requested, as audio.
4. User interface: system access is provided through an interface developed with *Gradio*, which allows users to query the assistant directly from a browser. This choice enables the system to be used not only from the computer running the model but also from other devices, ensuring portability and ease of use.

The entire implementation has been consolidated into a single Python script, *NanoGPT_Marinello.py*, which contains the model definition, training routines, inference functions, and interface management. Although this organization is not modular in a strict sense, it provides the advantage of a single reference point for the entire system,

simplifying the reproducibility of the work and offering a readily extensible basis for future developments.

Despite its simplicity, this architecture enables the integration of different technologies into a coherent and replicable system, suitable both for educational experimentation and for use in applied research contexts.

4.2 Description of the NanoGPT_Marinello.py Code

The file *NanoGPT_Marinello.py* constitutes the operational core of the project, bringing together in a single script all the components required for both training and interaction with the model. At its foundation lies a decoder-only Transformer language model, namely a network that relies exclusively on the decoding stack to generate text autoregressively, predicting each character on the basis of the preceding ones. This architecture has been enriched with a multimodal pipeline that enables the handling of both textual and voice input, coherently integrating transcription and synthesis modules.

From an implementation perspective, the program includes several classes and functions that together form the processing flow. The *CharDataset* class is responsible for data preparation: it builds the character-level vocabulary and generates the sequences required for training. The neural model is defined in the *GPT* class, which implements Transformer blocks with causal self-attention and residual connections, while also supporting an adaptive generation function capable of producing new text sequences in response to user prompts.

The learning process is managed by the *train_model* function, which enables the model to be trained on a user-provided corpus and to save the corresponding checkpoints, thereby making it possible to interrupt and resume training sessions. With respect to interaction, the *unified_chat* function represents the core of the pipeline: it accepts either text or voice input, performs transcription via *Whisper*, generates a response with the model, and, if required, produces a vocal output using *gTTS*.

An illustrative code fragment shows how interaction within the chat is managed:

```
def unified_chat(audio_input, manual_text, temperature, history):

    # handles voice or text input and generates the model’s response

    if manual_text.strip():

        user_text = manual_text.strip()
```

```
    logits, _ = model(context)      # forward pass

    generated = model.generate_adaptive(context, max_new_tokens=200)

    return history + [(f"USER: {user_text}", f"AI: {decode(generated)}")]
```

In this way, the script implements an integrated system that combines training, inference, and multimodal interaction within a single platform. While simple in structure, this organization ensures clarity, reproducibility, and extensibility, making the program suitable both for experimental purposes and for applied contexts. [1] [2]

4.3 User Interface

The assistant is accessed through a *Gradio*-based interface, an open-source library that enables the rapid and flexible development of browser-accessible web applications. This solution was adopted to ensure ease of use, portability, and the ability to test the system from devices other than the machine hosting the model, without requiring additional installations.

The interface is organized into functional tabs that reflect the main modes of system use. The multimodal chat section allows users to submit queries either in textual form, via an input field, or in spoken form, through a microphone connected to the device. In the latter case, the pipeline includes automatic transcription with *Whisper* and the delivery of the model-generated response, optionally converted into audio using *gTTS*.

In addition to chat functionality, controls are provided for adjusting certain generation parameters, such as temperature, which influences the creativity level of the responses. Session management features are also included, such as the ability to clear the conversation history or replay the last message produced by the system.

From an aesthetic standpoint, the interface was customized through a CSS stylesheet embedded in the script, which standardizes colors, typography, and component layout, thereby enhancing coherence and intuitiveness. While accessory to the project's technical core, this aspect contributes to improving usability and facilitates demonstration in academic or experimental settings.

A simplified example of how the main chat tab was defined is shown below:

```
with gr.Tab("💬 Voice & Text Chat"):

  audio_input = gr.Audio(sources=["microphone"], type="filepath")
```

```
text_input = gr.Textbox(label="Type your question")
chatbot = gr.Chatbot(label="Conversation")
send_button = gr.Button("Send")
clear_button = gr.Button("Clear Chat")
```

Overall, the choice of *Gradio* made it possible to seamlessly integrate the training and interaction functionalities previously described, completing the pipeline with a lightweight yet effective interface that makes the system immediately accessible even to non-expert users.

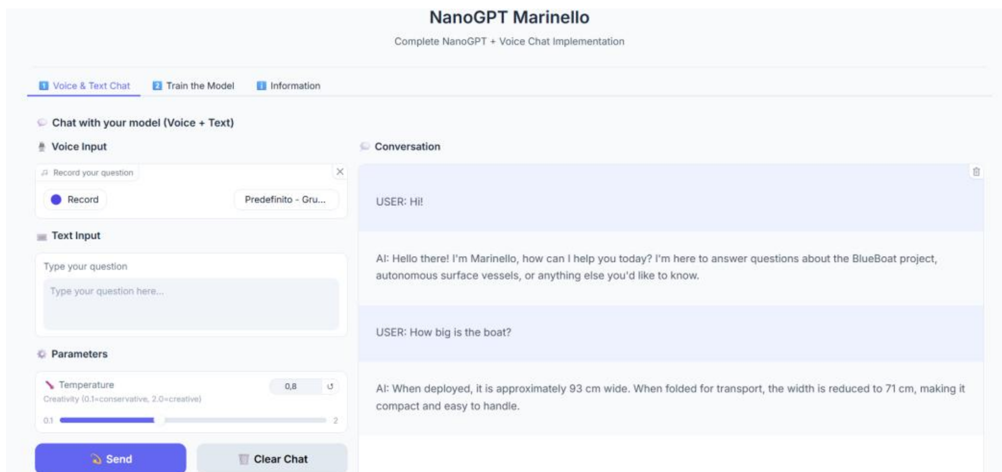


Figure 4.3.1 – Main interface of the *NanoGPT_Marinello.py* application, implemented with Gradio. The GUI provides multimodal interaction through voice and text input, adjustable parameters, and a conversation panel displaying the dialogue with the assistant.

4.4 Training Phase and Checkpoint Management

Model training represents one of the central components of the system and is fully managed within the *NanoGPT_Marinello.py* script. The goal is twofold: to enable the creation of a specialized model from a corpus of technical texts, and to allow interrupted sessions to be resumed at any time through the periodic saving of checkpoints.

The *train_model* function orchestrates this process: it receives the user-provided text, builds the vocabulary through the *CharDataset* class, initializes the GPT model, and starts a training loop for a predefined number of steps. During training, at each iteration the model receives character sequences as input and learns to predict the next symbol by minimizing the cross-entropy loss function, which measures how far the predicted

probability distribution deviates from the correct symbol, penalizing incorrect predictions. The system can also adapt automatically to the available resources: when a GPU is present, it exploits *CUDA* acceleration and mixed-precision training to optimize both performance and memory usage.

A key feature of the prototype is checkpoint management, implemented through the *load_checkpoint* function. At the end of training, the state of the model, the optimizer, and the vocabulary tables are stored in a binary file, which can later be reloaded without repeating the entire training process. This mechanism ensures reproducibility and facilitates the practical use of the system, especially in resource-constrained environments.

A simplified example illustrates the core of the training procedure:

```
for step in range(steps):
    xb, yb = dataset.get_batch(batch_size, block_size)
    logits, loss = model(xb, yb)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

This fragment highlights the typical optimization cycle: batch generation, loss computation, backpropagation, and weight update. The system also records performance metrics—such as the average loss value and training speed—allowing progress to be monitored during execution.

Overall, the training and checkpointing pipeline makes the model not only trainable from scratch on new datasets, but also easily reusable in subsequent sessions. This feature proves particularly valuable in experimental scenarios, where the ability to interrupt, resume, and compare different configurations represents an essential requirement for applied research.

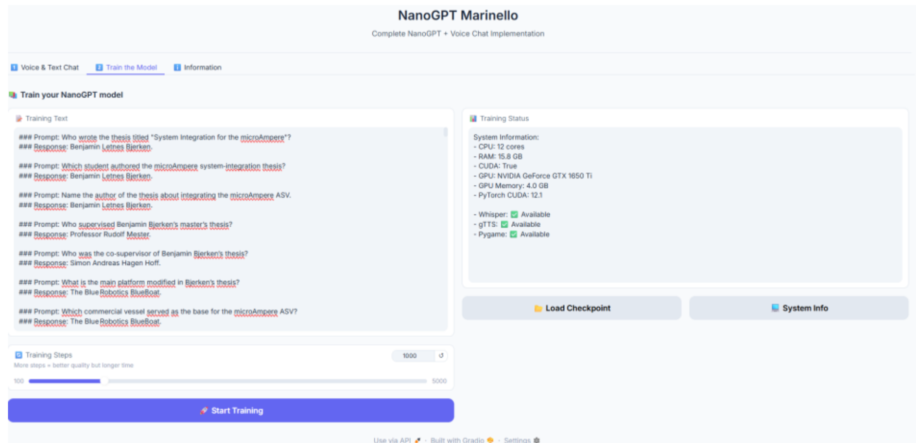


Figure 4.4 – Training interface of *NanoGPT_Marinello.py*. The user can provide a custom Q&A dataset, set the number of training steps, and monitor system resources and model status.

4.5 Text Generation and Qualitative Evaluation

Once the training phase is completed, the model is able to generate text sequences starting from a user-provided prompt. This process leverages the autoregressive nature of the model, which produces one character at a time conditioned on those already generated. The generate_adaptive function implements this logic, allowing parameters to be specified such as the maximum number of tokens to be generated and the temperature, which regulates the degree of variability in the responses: low values favor more conservative and predictable outputs, whereas higher values introduce greater creativity. The evaluation of response quality shows differentiated behavior depending on the type of question asked. When the request falls within the domain of the training data — for instance, questions related to the code or technical documentation of the *Nautilus/BlueBoat* project — the model produces coherent and accurate responses, even when the input contains spelling errors or missing characters. This indicates a solid generalization capability within the domain, supported by the char-level nature of the model.

However, when the question is entirely outside the dataset, the system lacks adequate knowledge and tends to generate irrelevant outputs, sometimes resembling random responses. This behavior reflects the limitations imposed by the reduced size of the corpus and the absence of pre-training on general-purpose texts.

Finally, through the *unified_chat* function, text generation is integrated into the interactive pipeline already described, enabling the user to directly assess the experience in both textual and voice-based modes.

Overall, the generation phase and its qualitative evaluation confirm that the system is effective at reproducing knowledge already present in the training dataset, while also highlighting its difficulty in handling completely new or out-of-domain questions.

CHAPTER V – DATASET AND KNOWLEDGE INTEGRATION

5.1 Data Sources

The construction of the dataset represented a crucial phase of the project, as the quality and relevance of the data directly influence the capabilities of the model. In this work, priority was given to sources within the domain of the *Nautilus/BlueBoat* project, in order to obtain a highly specialized language assistant.

The main data sources were:

- Master's theses and technical reports developed within the framework of the *BlueBoat* project at the Norwegian University of Science and Technology — NTNU — addressing topics such as software infrastructure, nonlinear predictive control, collision avoidance strategies, and reinforcement learning.
- Technical documentation and manuals relating to the *BlueBoat* platform and its control, perception, and navigation modules, used to integrate engineering and operational details.
- Q&A Dataset, derived from the documents mentioned above and generated with the support of commercial artificial intelligence systems, in order to increase the number of examples and simulate possible interactions with the assistant in dialogic form.

It should be emphasized that, although this experimentation employed specific academic and technical documents, the approach adopted is independent of the nature of the sources: in principle, the model can be trained on any type of text, provided that the data are relevant, coherent, and legally usable.

5.2 Reprocessing and Balancing Procedures

The process of dataset construction did not consist merely of collecting technical documents, but also involved targeted reprocessing aimed at transforming the sources into material directly usable for training. Specifically, instead of training the model on complete texts, a dedicated question–answer corpus was generated.

The questions were produced with the support of commercially available artificial intelligence systems, based on the content of the theses and reports used as references. This approach significantly increased the number of available examples, generating

hundreds of potential interactions and covering the key concepts from the sources in greater detail.
To structure the dataset in a way that was easily interpretable by the model, a syntactic convention was adopted:

- each question is preceded by the tag ###Prompt:,
- each answer by the tag ###Response:.

Thanks to this convention, the model learned to clearly delimit conversational turns, recognizing where the question ended and the answer began, without the need to impose an artificial character limit on generations.
Furthermore, to make the dataset more practical and closer to the behavior of commercial artificial intelligence systems, examples of colloquial conversations were also included, such as the following:

```
###Prompt: Hi!
###Response: Hello there! I'm Marinello, how can I help you today?
```

This strategy made the training process more robust and maximized the utility of the available sources, transforming them into a dialogic dataset more closely aligned with the intended real-world use of the system.

5.3 Ethical and Legal Considerations

The use of datasets constructed from existing sources requires careful ethical and legal reflection. First, the theses and technical reports employed for training the model were used exclusively for research and academic experimentation, in full respect of the context in which they were produced. They were neither distributed to third parties nor employed for commercial purposes.
Another aspect concerns the protection of intellectual property. The documents used remain the property of their respective authors and institutions; the training activity does not imply republication of the contents but rather their transformation into statistical form within a language model. In this sense, the model’s output does not correspond to a literal reproduction of the texts, but to a probabilistic re-elaboration based on the linguistic patterns it has learned.
Finally, it should be noted that the methodology adopted is consistent with practices commonly employed in the field of academic machine learning: limited use of texts for

research purposes, documentation of the provenance of the sources, transparency in reprocessing procedures, and careful attention to issues of responsibility in the use of data.

CHAPTER VI – SYSTEM EVALUATION AND LIMITATIONS

6.1 Training Metrics

Model training was primarily monitored through the cross-entropy loss function, which measures how far the probability distribution produced by the model deviates from the correct symbol, penalizing incorrect predictions.

Performance-related execution metrics were also recorded, such as the average time per step, the number of characters processed per second, and GPU/CPU memory usage. GPU utilization proved to have a decisive impact: training was significantly faster and more efficient compared to CPU execution, thereby reducing the time required to complete a training cycle.

Another aspect that emerged concerns the trend of improvement over time: beyond approximately 4,000 steps, the loss no longer decreased significantly, remaining essentially stable. This dynamic resembles a reverse logarithmic learning curve, in which initial progress is rapid but gradually diminishes, becoming marginal in the subsequent steps.

Figure 6.1.1 – Average training loss over 5000 steps. The x-axis represents the training steps, while the y-axis shows the average loss value. The curve highlights the progressive decrease in loss indicating the convergence of the model during training.

6.2 Qualitative Performance

The qualitative evaluation of the system highlighted different behaviors depending on the type of question.

- In-domain questions: when the input falls within the content of the NTNU theses and the technical documentation used for training, the model is able to provide coherent and relevant answers while maintaining terminological accuracy. Moreover, the char-level nature of the model enables it to tolerate spelling mistakes or typos: even in the presence of missing or misspelled characters, the assistant is still able to understand the request and produce an appropriate response.
- Out-of-domain questions: when requests deviate completely from the training corpus, the model lacks useful knowledge and tends to generate arbitrary or irrelevant responses. In such cases, the output may appear generic or resemble a random completion, reflecting the limitations imposed by the small size of the dataset and the absence of general-purpose texts.
- Colloquial conversations: to improve user experience, the dataset also included examples of conversational exchanges generated with the support of commercial AI systems. This allowed the model to handle simple interactions, such as greetings and introductory phrases, producing responses that are smoother and more natural. While this addition does not directly enhance the system's technical expertise, it increases usability and makes the assistant more comparable to widely available voice and text-based assistants.

Overall, the system performs well within the specific domain on which it was trained, showing robustness even in the presence of imperfect inputs, but it exhibits clear limitations when operating outside the scope of the training sources.

6.3 Discussion of Limitations

The performance analysis highlighted several structural and methodological limitations of the developed prototype:

- Dataset size: the amount of available data proved to be limited. This results in strong dependence on the specific content of the NTNU theses and the texts used, with the risk that the model may be unable to answer questions falling outside these sources.

- Char-level approach: adopting character-level tokenization has the advantage of simplifying vocabulary construction and making the model more tolerant of spelling errors. However, this choice reduces efficiency and the ability to capture long-range semantic relationships compared to more advanced approaches.
- Reduced architecture: the model implements a relatively simple configuration. While this choice was necessary to comply with resource constraints, it makes the assistant suitable as a demonstrative prototype but not comparable in performance to larger-scale language models.
- Hardware constraints: training on CPU proved inefficient and required very long processing times. The use of GPUs, particularly with *CUDA* acceleration, significantly reduced training time, yet it remains evident that without access to more powerful computing resources, system scalability is limited.
- Dependence on external components for the voice pipeline: the multimodal pipeline relies on third-party modules such as *Whisper* for transcription and *gTTS* for speech synthesis. This introduces portability constraints and potential points of failure associated with the integration of external software.

In summary, the identified limitations do not undermine the significance of the experimentation but instead help to delineate its scope. The prototype should be regarded primarily as an academic demonstrator and a training tool for new developers, rather than as a production-ready assistant. At the same time, these critical aspects offer concrete directions for improvement, including the expansion of the dataset, the adoption of more sophisticated tokenization strategies, and the exploration of more powerful model architectures. [1]

CHAPTER VII – GUIDELINES AND CONCLUSIONS

7.1 Guidelines for Future Development

The developed prototype constitutes a solid yet still preliminary foundation. To transform it into a more complete and versatile tool, it is useful to establish a set of guidelines to orient future developers. The following recommendations highlight the main areas for improvement identified during the project and propose possible directions for further work.

- Dataset: the model was primarily trained on question–answer pairs derived from technical texts. While this format ensured coherence, it limits the variety of possible interactions. To make the system more robust, the dataset should be enriched with broader and more diverse materials—descriptive documentation, multi-turn conversations, and, in particular, reasoning-oriented examples such as short logical problems or mathematical calculations. This would allow the assistant to go beyond factual responses and develop processing capabilities more aligned with real-world interactions.
- AI technology: the current prototype is based on a relatively simple char-level model. To improve performance, it will be necessary to adopt more advanced tokenization techniques and evaluate deeper architectures. Another promising evolution would be the integration of external modules—for instance, computational libraries—enabling the system to perform operations such as simple mathematical calculations. This would transform the assistant from a mere text generator into a truly interactive tool.
- Interface: the current GUI developed with *Gradio* is simple yet functional. In the future, it could be made more accessible by adding graphical elements, conversation history, and the ability to save sessions. An additional step would be to publish the assistant on a dedicated web domain, thereby enabling constant remote access without the need to manually launch the local server.
- Training: testing showed that after approximately 4,000 steps the model's improvements tend to plateau. It will therefore be useful to adopt optimization techniques such as learning rate schedulers and early stopping, in order to avoid wasting resources. To increase data variety, text augmentation techniques could

be explored—for example, the automatic generation of variants and paraphrases—thus reducing the need to manually expand the corpus. A systematic checkpoint management strategy, with intermediate versions saved and compared, would also help identify the most effective configuration.

- Further developments: the system could benefit from the integration of automatic evaluation metrics, such as perplexity — measuring how predictable a text is for the model — or BLEU scores — comparing generated responses against reference answers. These tools would enable more objective monitoring of output quality. The assistant could also be extended to support multi-user interactions, introducing parallel session management. Finally, linking the system to structured databases or external tools would pave the way toward an assistant capable not only of providing information but also of performing practical tasks, bringing it closer to the functionality of a commercial AI.

7.2 Conclusions

The work carried out led to the development of a specialized language assistant, built entirely from scratch — from dataset construction to model implementation. At the core of the project lies an LLM based on a character-level, decoder-only Transformer architecture, enriched with a multimodal pipeline integrating both text and voice input, and made accessible through a web interface. The main objective was to provide a tool enabling interaction with documentation and knowledge related to autonomous systems such as those addressed in the NTNU theses on the *BlueBoat* project, thereby facilitating the understanding and exploration of complex control and navigation architectures.

Despite resource constraints and the limited size of the corpus, the developed LLM demonstrated the feasibility of obtaining a functional prototype capable of producing coherent responses within the specific domain on which it was trained. This result is particularly significant as it was achieved without resorting to pre-trained models, but by building the entire pipeline of dataset, training, generation, and interface from the ground up.

The objectives of the thesis—designing a simple yet extensible architecture, training a model on targeted sources, integrating multimodal functionalities, and providing a useful reference for future developers—can be considered accomplished. The prototype should be understood not as a commercial product ready for deployment, but as an academic

demonstrator and experimental platform, offering insight into how an LLM can support the description and discussion of autonomous systems.

The identified improvement paths outline a clear trajectory for the system's evolution, with the aim of gradually bringing the prototype closer to the standards of commercial AI.

In conclusion, this thesis has shown that it is possible to develop an LLM from scratch and employ it as a support tool for interacting with knowledge related to an autonomous system, while at the same time providing a reference framework for those who wish to continue and extend the project in the future.

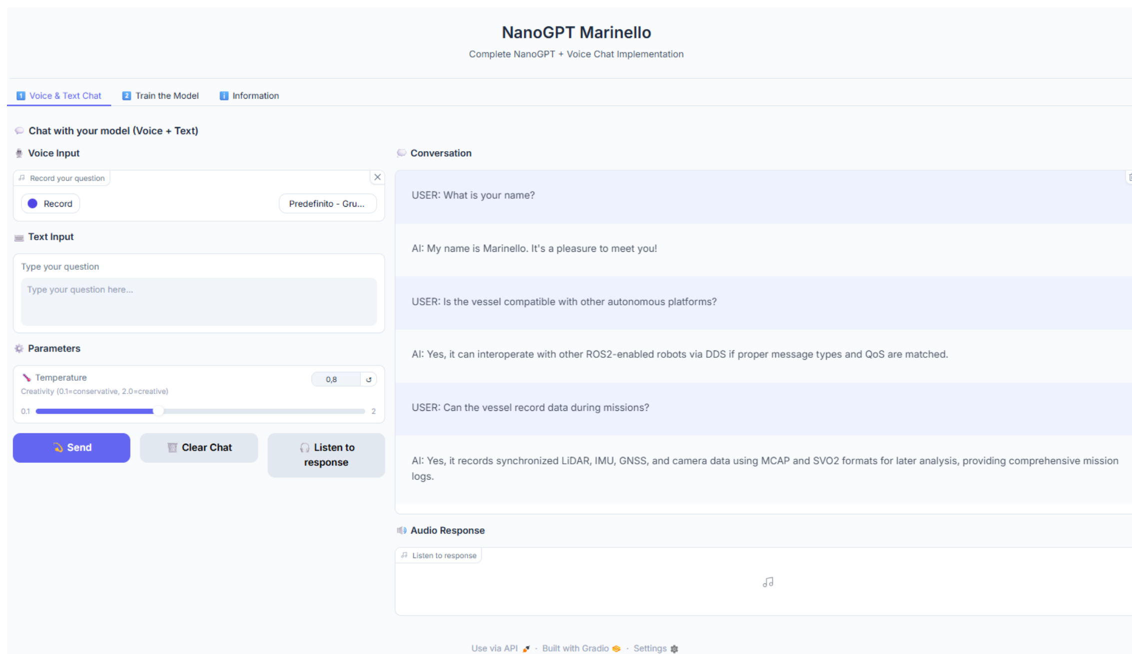


Figure 7.2.1 – Example interaction with NanoGPT Marinello. The assistant is able to introduce itself, answer technical questions about autonomous platforms, and provide detailed information on system interoperability and data recording. This dialogue illustrates the capability of the prototype to deliver domain-specific knowledge in a natural conversational format.

CHAPTER VIII – SUPPORTING MATERIALS

8.1 Access to the Repository

To promote reproducibility and provide future developers with an immediate starting point, all project materials have been made available in a dedicated GitHub repository associated with the author of this thesis.

The repository includes:

- the main script *NanoGPT_Marinello.py*;
- a sample Q&A dataset — *Q&A_Dataset.txt*;
- a requirements.txt file listing the necessary dependencies;
- a .gitignore file for handling temporary files and large checkpoints;
- a README file with detailed usage instructions;
- pre-trained checkpoints, made available through a linked Google Drive folder.

The repository is available at the following address:

https://github.com/FrannPizz/NanoGPT-Marinello

Checkpoints can be accessed via Google Drive:

NanoGPT Marinello – Checkpoints

8.2 Environment and Dependencies

To ensure correct system execution, it is necessary to set up an updated Python environment and install the required libraries.

The project was developed and tested on Python 3.10 and Python 3.11, versions that guarantee full compatibility with *PyTorch*.

The main dependencies include:

- *PyTorch*: library for defining and training the neural model;
- *torchvision* and *torchaudio*: auxiliary modules for handling multimedia data;
- *openai-whisper*: for audio transcription — speech-to-text;
- *gTTS* — Google Text-to-Speech: for speech synthesis;
- *pygame*: for local audio management;
- *gradio*: for building the browser-based user interface;
- *psutil*: for monitoring system resources.

All required libraries are listed in the *requirements.txt* file included in the GitHub repository. Installation can be performed with a single command:

```
pip install -r requirements.txt
```

From a hardware perspective, the system was tested both on CPU and on NVIDIA GPUs with *CUDA* support. The use of a GPU is strongly recommended, as it significantly reduces training time compared to CPU execution.

8.3 Dataset

The repository includes a sample file, *Q&A_Dataset.txt*, structured in a question–answer format.

Each entry in the dataset follows the schema:

```
###Prompt: [question text]
###Response: [answer text]
```

The provided file serves as a baseline corpus for training but can be easily extended. To add new knowledge, it is sufficient to insert additional prompt–response pairs in the same format, without the need to modify the code.

8.4 Checkpoints and Reproducibility

During training, the system automatically saves checkpoints that contain:

- the state of the neural model,
- the parameters of the optimizer,
- the vocabulary tables.

These files make it possible to resume training at any time without loss of information, as well as to compare different versions of the model.

To use a checkpoint, the file simply needs to be placed in the same directory as the main script — *NanoGPT_Marinello.py*. At startup, the program automatically detects the presence of the checkpoint and loads it without requiring additional commands.

BIBLIOGRAPHY

Scientific Articles:

1. **Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Polosukhin, I.** (2017). *Attention is all you need*. Advances in Neural Information Processing Systems, 30.
2. **Radford, A., Narasimhan, K., Salimans, T., & Sutskever, I.** (2018). *Improving language understanding by generative pre-training*. OpenAI technical report.

Norwegian University of Science and Technology Theses:

- **Bjerken, B.** (2025). *Microampere software infrastructure for autonomous surface vessels* (Master's thesis, Norwegian University of Science and Technology).
- **Reimers, H.** (2025). *Nonlinear Model Predictive Control and BlueBoat hardware integration* (Master's thesis, Norwegian University of Science and Technology).
- **Alstergren, J. K**. (2025). *Risk-Based Collision Avoidance for Autonomous USVs* (Master's thesis, Norwegian University of Science and Technology).
- **Karlsen, V**. (2025). *Reinforcement Learning for BlueBoat autonomous navigation* (Master's thesis, Norwegian University of Science and Technology).
- **Cao, L.** (2025). *BlueBoat hardware development and integration* (Master's thesis, Norwegian University of Science and Technology).

Software and Libraries

- **Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., ... & Chintala, S.** (2019). *PyTorch: An imperative style, high-performance deep learning library*. Advances in Neural Information Processing Systems, 32.
- **Gradio** (2023). *Gradio documentation*. Retrieved from https://www.gradio.app
- **OpenAI** (2022). *Whisper: Robust speech recognition via large-scale weak supervision*. Retrieved from https://github.com/openai/whisper
- **Google** (2023). *gTTS: Python library and CLI tool to interface with Google Translate's text-to-speech API*. Retrieved from https://pypi.org/project/gTTS/

NOTE ON THE THESIS TEXT

Some English-language sections of this thesis were subjected to grammatical and stylistic revision with the support of OpenAI - ChatGPT, which was used exclusively as a linguistic correction tool.