



Developing a Chatbot for Internal Documents

Alexandre Zürcher

Haaga-Helia University of Applied Sciences
Bachelor of Business Information Technology
Product-based thesis

Abstract

Author Alexandre Zürcher
Degree Business Information Technology
Thesis Title Developing a Chatbot for Internal Documents
Number of pages and appendix pages 53 + 0
<p>This product-based thesis presents the development of a document-based chatbot for the Swiss company Innovatim. The chatbot's main aim is to provide accurate, document-based responses to end-user queries, and to be embeddable onto any existing website. The end goal of Innovatim is for the product to be a commercial product that they can sell to other companies, in a B2B business model.</p> <p>One distinctive feature of this project is its use of entirely open-source technologies, enabling the creation of a privacy-focused software, tailored to the commissioning company's individual client needs.</p> <p>The report starts with the introduction chapter, which introduces the concepts of Generative AI chatbot and Retrieval Augmented Generation, and states the various objectives of the project, as well as their feasibility.</p> <p>The chatbot's goals are to have an intuitive user interface, cite his sources, support PDFs, Word documents, Excel sheets and plain text files, and answer in English, French, German and Italian. The software as a whole needs to be sufficiently secure, to be deployed on a Swiss infrastructure, to not use any AI-related API, to be able to change the LLM model used on the fly, to maintain continuous operation, and to compensate carbon footprint as much as possible.</p> <p>Following the introduction, the methodology chapter covers the Scrum-inspired project management, and showcases the 10 Usability Heuristics for User Interface Design principles by Dr. Nielsen, which are applied during the development of both the user and administrative interfaces.</p> <p>Thereafter, the architecture of the solution is explained, which breaks down the choice of using OpenStack and Docker for the infrastructure. Additionally, this section is supported by a diagram which shows how the various docker containers interact with each other.</p> <p>After that, each of the project's objective is explored separately in the form of user stories, which present into their implementation, the technology choices, and the various challenges faced.</p> <p>The thesis wraps up on the discussion chapter, which concludes that the project is qualitative as most goals are met and as the commissioning company is satisfied. However, more development time is still necessary for the chatbot to cite its sources, be able to answer in multiple languages, and be deployed on OpenStack.</p>
Keywords AI, chatbot, RAG, open source, software development, LLM, Generative AI, LangChain, LangServe, FastAPI, Milvus, Astro, Docker, OpenStack

Contents

Abbreviations.....	1
1 Introduction.....	2
2 Methodology.....	8
2.1 Project management.....	8
2.2 Principles to build design of the user interface.....	9
3 Architecture.....	11
3.1 Infrastructure as Code.....	11
3.1.1 Docker.....	11
3.1.2 OpenStack.....	11
3.2 Diagram.....	12
4 User story – The chatbot will use RAG techniques to provide document-based responses.....	13
4.1 Main backend framework – LangChain.....	13
4.2 REST API framework – LangServe / FastAPI.....	14
4.3 Storing the document on a vector store – Milvus.....	15
4.4 The chatbot will need to cite the source it uses for its answers.....	17
4.5 User story - Support PDFs, Word documents, Excel sheets and plain text files.....	19
5 User story – Easily change which LLM model is used.....	20
5.1 Choosing an open-source LLM model.....	20
5.2 Containerization of the LLM.....	20
5.3 User story – The chatbot will be able to answer in English, French, German and Italian. .	21
6 User story – Have an intuitive user interface.....	23
6.1 Using Web Components.....	23
6.2 The chatbot user interface.....	26
6.2.1 Displaying message bubbles.....	28
6.2.2 Minimizing the popup and embedding the chatbot in an iframe.....	30
6.2.3 Managing errors.....	32
6.3 The administrator user interface.....	33
6.3.1 Managing documents on the vector store.....	34
6.3.2 Managing chatbot prompts.....	35
6.4 Functionalities shared between chatbot user and administrator user interface.....	37
6.4.1 Implement internationalization with 4 languages.....	37
6.4.2 Displaying popups.....	38
7 User story – Good security, so only authorized users are able to access private information...	41
7.1 CORS validation.....	41
7.2 Firewall rules.....	41
7.3 Token-based authentication.....	42
8 User story – Deploy the chatbot on a Swiss infrastructure.....	44

8.1	Managing project level variables.....	44
8.2	Running the application in containers.....	45
8.3	Orchestration of the containers with docker-compose.yaml files.....	47
8.4	User story – Compensate as much of the carbon footprint as possible.....	47
9	User story – Guarantee continuous operation, even when third party services used are down.	48
9.1	Circumvent the unavailability of Hugging Face servers.....	48
9.2	Circumvent the unavailability of Ollama servers.....	49
10	Discussion.....	50
	References.....	52

Abbreviations

AI – Artificial Intelligence

API – Application Programming Interface

B2B – Business-to-business

CSS – Cascading Style Sheets

HTML – HyperText Markup Language

i18n – Internationalization

LLM – Large Language Model

RAG – Retrieval Augmented Generation

UI – User Interface

1 Introduction

This project was commissioned in 2023 by the Swiss-based B2B company Innovatim, an agency specialized in no-code solutions and AI. Their main business is to help other companies in their digital transformation, and also to boost workers productivity (Innovatim, no date).

The main goal of this project is to produce a software that provides a chatbot experience that would be able to answer user queries using a company's internal documents. The required document formats that need to be supported are PDFs, Word documents, Excel sheets and plain text documents.

Being a project-based thesis commissioned by a company, the aim was since the beginning to produce a viable product, that can actually be sold to other companies following a B2B business model. For this reason, this thesis will explain the different challenges and solutions I ran across, in the form of user stories. I will go into detail into the implementation of the solutions, and talk about the reasoning behind them. The architecture of the software will also be showcased.

Table 1: List of initial goals, with personal comment made at the start of the project

Initial goal	Personal comment at start of the project
The chatbot will use RAG techniques to provide document-based responses	The key part of the project, should be doable, as many concrete examples already exist online.
The chatbot will need to cite the source it uses for its answers	Good prompt engineering will be needed for this.
The chatbot will be able to answer in English, French, German and Italian	This will be a defining factor in choosing one (or more) LLM model(s) for the chatbot.
Support PDFs, Word documents, Excel sheets and plain text files	Python libraries exist for each of those document types.
Good security, to guarantee that only authorized users will be able to access private information	The framework used for the backend will need to support authentication.
Deploy the chatbot on a Swiss infrastructure, without using existing AI-related APIs	As the plan is to use open source technologies only, no outside API are required.
Have an intuitive user interface	The commissioned user interface has few elements, which will make for a simple UI.
Possibility to easily change LLM model used	To be determined if needed on the frontend / backend / both.
Guarantee continuous operation, even when third party services used are down	The various providers used by the software could go down. The application should still stay online in this case.
Compensate as much of the carbon footprint as possible	The hosting company will be required to demonstrate a strong commitment to ecological sustainability.

Many requirements were given at the start of the project, such as being able to embed the chatbot on a third party website after being logged in on there. Another important need was to run an open source large language model. The idea behind this decision was to create a product that a company can trust with their internal documents without them being sent to a US company such as OpenAI, Google, Meta, etc. This also allows Innovatim to be in maximum control of the costs, with the disadvantage that they have to manage the hosting of the model themselves.

I also personally added the goal to put in place an OpenStack Heat Orchestration Template (HOT) file, to deploy for easy deployment on Open Stack based public cloud.

The chatbot will use RAG techniques to provide document-based responses

This is the main goal of the project. Innovatim wanted to use a chatbot with generative conversational AI and Retrieval Augmented Generation capabilities.

In order to understand what a chatbot with Retrieval Augmented Generation is, it helps to make a distinction between the different types of chatbots that exist.

The oldest and simplest type of chatbot is **rule-based**, where “the user’s utterance gets matched to a chatbot intent. In rule-based chatbots, you can use regular expressions to match a user’s statement to a chatbot intent” (Codeacademy, no date). For a more concrete example, when the user writes “what’s the weather in Helsinki”, the chatbot’s detects the “weather” and the “in” keywords, store “Helsinki” as a variable as it comes right after the “in” keyword. Finally, it returns information on the weather in the asked location using a third party API.

As Innovatim’s clients can be in many different industries, and their queries will vary drastically, using a simple rule-based chatbot was out of the question.

In contrast, **conversational AI** based chatbots are way more advanced, as they use “large volumes of data, machine learning and natural language processing to help imitate human interactions, recognizing speech and text inputs and translating their meanings across various languages” (IBM, 2021).

This method requires the use of a Large Language Model (LLM), which IBM defines as “a category of foundation models trained on immense amounts of data, making them capable of understanding and generating natural language and other types of content to perform a wide range of tasks” (IBM, 2024a). ChatGPT, which was released in November 2022 and only took 2 months to reach 100 million users, is a well-known example of a LLM (Livemint, 2023). It’s also possible to execute a LLM locally, on one’s own computer, or on a server in a datacenter.

As opposed to the rule-based chatbots that look for a set of keywords, conversational AI chatbots takes every word of the user query into account. Then, they answer based on the most statistically correct way possible, based on the data that was fed to them from various sources. It's important to note a conversational AI currently only understand “how words relate statistically”, but not what they actually mean (Martineau, 2023).

For this reason, the biggest downside of this method is that answers from this type of chatbot can range from correct and accurate to nonsensical and factually wrong. Still, it's currently the best way to achieve this goal because of the versatility that comes with it.

Still, this project uses those technologies, but for the chatbot to be able to “read” documents in order to answer queries, we need to add Retrieval Augmented Generation (RAG) to conversational AI.

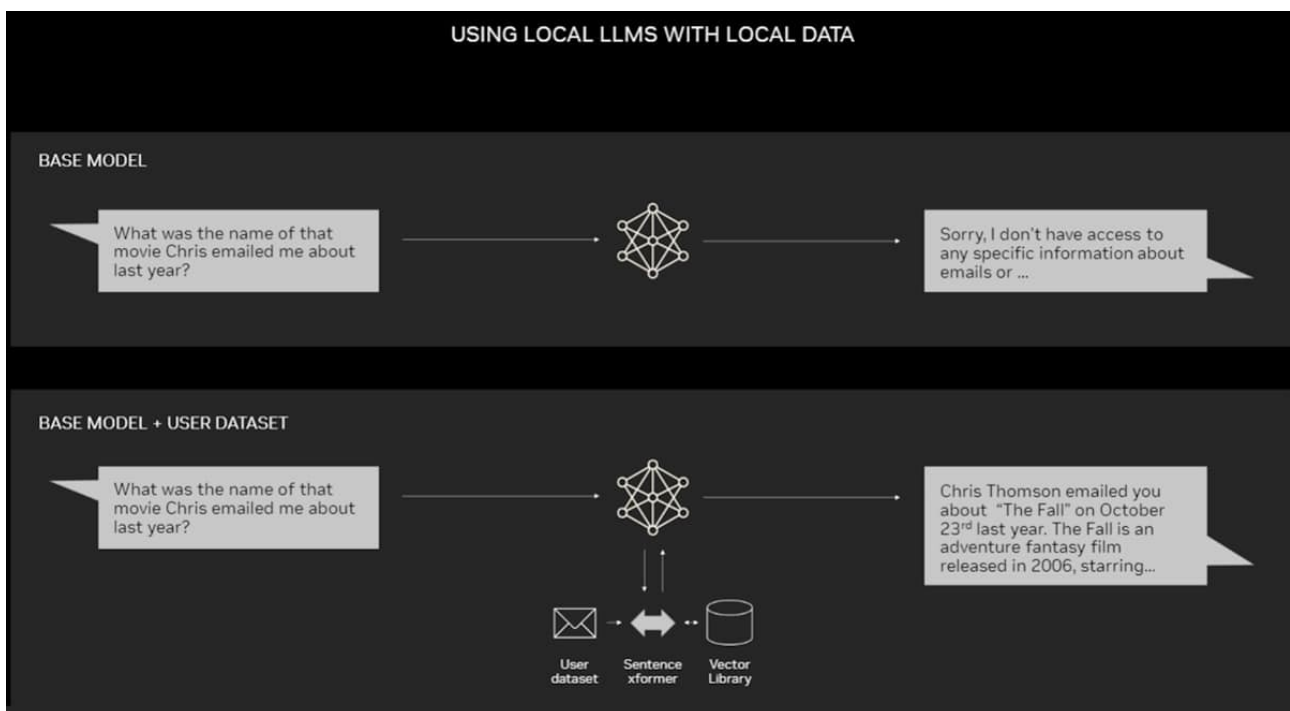


Figure 1: Diagram showing the difference between LLM with and without RAG capabilities, published on the NVIDIA blog (Merritt, 2023)

RAG is an “AI framework for improving the quality of LLM-generated responses by grounding the model in external sources of knowledge to supplement the LLM’s internal representation of information”. On simpler terms, it’s a method used to give a conversational AI chatbot the ability to “read” and “understand” documents, which can be used by the chatbot to answer queries. This method has the advantage that it doesn’t require to train the LLM model to add data to it, which reduces the time and required computational power.

Further technical explanations on RAG can be found later in this thesis.

The chatbot will need to cite the source it uses for its answers

As explained in the previous goal, the project uses conversational AI with RAG capabilities. However, in order for the bot to accurately get information from documents and cite the source(s), it will be important to use good prompt engineering, which is a way to “guide generative artificial intelligence (generative AI) solutions to generate desired outputs” (Amazon Web Services, Inc., no date b).

An example of a prompt could be : “You are a chatbot responsible for answering questions about legal documentation on NIPD in Switzerland. You should never answer a question with another question, and you should always answer with the most relevant documentation page. Do not answer questions that do not concern NLPD. Always start with 'from the legal point of view'.”

This is something that can be drastically different from one of Innovatim’s client to the next, as many different factors can come into play: the main language used in the company will be used in the prompt, and the client’s industry will determine what the chatbot role will be.

The goal here is to find a good baseline prompt that can be adapted and extended while still citing the source of the knowledge.

The chatbot will be able to answer in English, French, German and Italian

As Innovatim is based in Switzerland, it’s important to support the main languages spoken in the country. Supporting them will come down to choosing LLM models that can “speak” each of those languages.

I made clear to the commissioning company at the start of the project that I was fluent in French and English, knew a bit of German, but no Italian. For this reason, I will only be able to test the French and English capabilities of a given LLM model.

Support PDF documents, Word documents, Excel sheets and plain text documents

Multiple type of documents must be supported by the software, in order to be as easy to use as possible for the end user.

Good security, to guarantee that only authorized users will be able to access private information

This goal evolved during the duration of the project. The first requirement was to provide a secure login, to guarantee that only authorized users will be able to access private information.

Later on, Innovatim wished to move from credential based security for the chatbot, in order to make the user experience as lean as possible. Researching the best options to secure the chatbot in an invisible way for the end-user became the new goal.

Deploy the chatbot on a Swiss infrastructure, without using existing AI-related APIs

Innovatim explained to me that they wanted to deploy LLM models themselves, and not simply use ChatGPT or other generative AI platforms. This decision was motivated by the privacy concerns of using such services, as well as the lower cost and increased flexibility of deploying LLM themselves.

About the Swiss infrastructure choice, it came down again to Innovatim being a Swiss company. That being said, there still is the possibility that the chatbot could be deployed in another country in the future. Because of that, it's important to choose an infrastructure with as little vendor locking as possible, meaning that it will be possible to deploy on another web hosting company with little to no hassle.

Have an intuitive user interface

In order to be as simple to use as possible to avoid end-user frustrations, the user interface will need to be intuitive and simple to use.

Possibility to easily change LLM model used

This goal came into be for two reasons. The first one is the need to change the LLM model used depending on Innovatim's client and their industry, and spoken language(s).

Also, as new, more powerful models are released regularly, migrating easily to them will be a necessity. As such, updating which LLM is used by an instance should be as simple as possible.

Guarantee continuous operation, even when third party services used are down

In order for the software to work, online resources are needed to be accessed, such as the LLM models themselves. Finding ways for the software to continue operation even if those resources would become unavailable is key to maintain customer satisfaction.

Compensate as much of the carbon footprint as possible

From a sustainability perspective, it's important to note that IT in general and AI in particular are extremely energy hungry. In 2028, AI is projected to use more power than the entire country of Iceland used in 2021 (Ammanath, 2024).

It's essential to reverse this tendency as much as possible, without necessarily abandoning this promising technology. In order to do that, we want to choose a hosting company that was as green as possible to host the finished product in order to balance our carbon footprint.

2 Methodology

2.1 Project management

Innovatim and I decided to conduct a meeting every two weeks, on which I showcased the various features I would work on, as well as the challenges I faced and sometimes needs to be able to continue working.

Concurrently, a centralized to-do list was maintained in the README file of the GitHub repository, which was also available to Innovatim. This way of working is inspired by the sprints made famous in the Scrum methodology, but in a more informal way.

The commissioning company provided guidance when it was needed, and acted as both product owner and client depending on the situation.

We agreed at the beginning of the project on a list of features to be developed, as well as a general timeline for the milestones. Here's what we agreed on at the beginning:

Table 2: Initial timeline for the project milestones.

Date	Description
2024-02-02	Proof of concept – Streamlit + LangChain example
2024-02-09	Research – Test different vector databases solutions
2024-02-23	Back-end – Put in place a Rest-API using FastAPI
2024-02-23	Back-end – Persist the vector database
2024-03-01	Frontend – Basics for testing (text input, streaming response from AI, embeddable)
2024-03-01	Research – Test different LLM models
2024-03-08	Back-end + Frontend – Authentication
2024-03-15	Back-end + Frontend – Ability to easily switch which LLM model to use
2024-03-22	Frontend – Implement multiple languages support (English, Italian, French and German)
2024-03-29	Deployment – Create OpenStack Heat Orchestration Template (HOT) file
2024-04-05	Deployment – deploy test on Infomaniak's Public Cloud

At first, we followed the schedule closely, but as the scope of the thesis grew, we ended up working in sprints where I focused on a specific feature or problem, until Innovatim validated the feature.

2.2 Principles to build design of the user interface

When talking about user interface design, I personally like to keep in mind the 10 Usability Heuristics principles from Dr. Jakob Nielsen (Jakob Nielsen, 2024).

General rules of thumb, the concepts formulated in 1994 still rings true today. During my studies where I learned about UX design, I found these rules to be flexible enough to still express myself, but specific enough to keep me from experimenting too much.

I'll be referencing to the principles used when discussing the frontend interfaces. Here they are for reference:

1: **Visibility of System Status** - The design should always keep users informed about what is going on, through appropriate feedback, within a reasonable amount of time.

2: **Match Between the System and the Real World** - The design should always keep users informed about what is going on, through appropriate feedback within a reasonable amount of time.

3: **User Control and Freedom** - Users often perform actions by mistake. They need a clearly marked "emergency exit" to leave the unwanted action without having to go through an extended process.

4: **Consistency and Standards** - Users often perform actions by mistake. They need a clearly marked "emergency exit" to leave the unwanted action without having to go through an extended process.

5: **Error Prevention** - Good error messages are important, but the best designs carefully prevent problems from occurring in the first place. Either eliminate error-prone conditions, or check for them and present users with a confirmation option before they commit to the action.

6: **Recognition Rather than Recall** - Minimize the user's memory load by making elements, actions, and options visible. The user should not have to remember information from one part of the interface to another. Information required to use the design (e.g. field labels or menu items) should be visible or easily retrievable when needed.

7: **Flexibility and Efficiency of Use** - Shortcuts — hidden from novice users — may speed up the interaction for the expert user so that the design can cater to both inexperienced and experienced users. Allow users to tailor frequent actions.

8: Aesthetic and Minimalist Design - Shortcuts — hidden from novice users — may speed up the interaction for the expert user so that the design can cater to both inexperienced and experienced users. Allow users to tailor frequent actions.

9: Help Users Recognize, Diagnose, and Recover from Errors - Error messages should be expressed in plain language (no error codes), precisely indicate the problem, and constructively suggest a solution.

10: Help and Documentation - Error messages should be expressed in plain language (no error codes), precisely indicate the problem, and constructively suggest a solution.

3 Architecture

Innovatim plan to use the service internally, but most importantly to deploy it for their clients. For this reason, it's very important to have as simple as possible of a deployment process.

I'm a big advocate of the DevOps philosophy, which aims to release software as fast as possible using a set of tools, practices and set of principles (Amazon Web Services, Inc., no date a).

One of those practices is Infrastructure as Code, which is the idea that an "infrastructure is provisioned and managed using code and software development techniques, such as version control and continuous integration." (Amazon Web Services, Inc., no date a). This means that a text file describes an infrastructure, which enables replication, versioning and fast deployment.

3.1 Infrastructure as Code

We wanted to take advantage of this for this project, which is why we settled on docker as a containerization technology, and OpenStack for the deployment platform.

3.1.1 Docker

Why Docker ? According to Stack Overflow 2023 user survey, Docker was the n°1 tool used by developers (Stack Overflow, no date), and while the most popular tool is not necessary perfect, their documentation is thorough and clear, a lot of resources are available on the subject, and it's also continually updated and improved.

Another key factor in choosing Docker was my personal experience with it, which helped me to get started quicker with the project than with a technology I wouldn't know as well.

As Innovatim wanted to keep the project lean, the use of the Docker Compose orchestration tool also made sense, as it is a relatively simple format to write in, in comparison to the behemoth that is Kubernetes. It also helped that I had more experience in working with Docker Compose than Kubernetes.

3.1.2 OpenStack

As for the Infrastructure as Code practice, we chose OpenStack and its templating system called Heat Orchestration Template (HOT), which is what Infomaniak uses for their Public Cloud product (Infomaniak, no date b).

OpenStack is explained as such : "OpenStack is a cloud operating system that controls large pools of compute, storage, and networking resources throughout a datacenter, all managed and provisioned through APIs with common authentication mechanisms. A dashboard is also available, giving administrators control while empowering their users to provision resources through a web interface. Beyond standard infrastructure-as-a-service functionality, additional components provide orchestration, fault management and service management amongst other services to ensure high availability of user applications." (OpenStack, no date).

Instead of using a proprietary cloud service such as AWS or Microsoft Azure, multiple hosting companies deploy OpenStack, which enables Innovatim to either switch to another provider, or reuse the same templating file elsewhere, if needed. This approach enables more options, and in our case, more cost-effective hosting than the competition.

3.2 Diagram

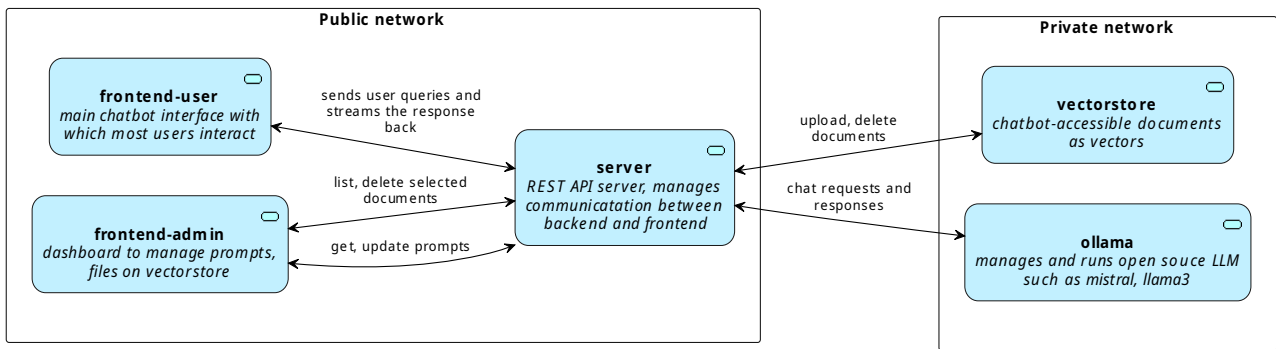


Figure 2. Architecture diagram describing how the different Docker containers interact.

Another key practice in DevOps is the use of containers, which segregates an application in smaller services. In our case, the following containers are present :

- **ollama** – the LLM itself.
- **vectorstore** – the vector database (also known as a vector store), which will contains the documents that the LLM can access.
- **frontend-user**– the main interface, used to chat with the agent in a ChatGPT-like experience.

As this is embedded on a parent website, the frontend for the user can be integrated in an iframe element on a page only available after logging in. As for security, an API token is passed from the parent site to the iframe for authentication.

- **frontend-admin** – the administrative interface, which will allow managing the documents available to the LLM, and also to change the LLM's prompts.
- **server** – the center point where all other services intersect. It will handle the requests from the frontends, manage the file operations (uploading, removing), and handle communication between the vector store and the LLM.

Only requests that contain a valid API key are accepted by the serve.

4 User story – The chatbot will use RAG techniques to provide document-based responses

4.1 Main backend framework – LangChain

When they submitted their proposition for a Bachelor Thesis project, Innovatim already conducted some tests themselves, and found the LangChain framework to be a great choice for the main backend framework. Written in Python, one of its features is to build a complete chatbot agent with augmented retrieval capabilities.

Let's explain the needed components for a chatbot agent with retrieval capabilities (the ability to "read" documents) to work.

The LangChain framework provides many integrations with LLM providers, most notably Ollama, which we ended up using – more on that later. It's also possible to connect to LLMs from OpenAI, Google's Generative AI, IBM Watson, and many others (LangChain, no date a). It's necessary that while it's possible to run a LLM on a CPU only, having access to an NVIDIA GPU greatly improves its performance.

Next, you need to store the documents in a format the LLM can understand, which is vectors, typically stored in a **vector database**, also called a **vector store**. As IBM puts it, "Unlike traditional relational databases with rows and columns, data points in a vector database are represented by vectors with a fixed number of dimensions, clustered based on similarity. This design enables low latency queries, making them ideal for AI-driven applications".

So, we need to store the documents we want the chatbot to interact with as vectors in a vector database. But how can we convert a PDF, or a PowerPoint document to vectors? For this, we need to convert each file to plain text using a python library, then run what's called a **text embedding model**, which converts plain text into vectors (LangChain, no date b). As Innovatim wished to only use open source technologies and no third-party APIs, we ended up using the Hugging Face integration from LangChain, where many open source text embeddings models can be found.

LangChain is the glue between all of those elements. Each of these components are integrated natively into the framework, as well as other necessary features, such as keeping a chat history – so the chatbot agent remembers what you just said.

4.2 REST API framework – LangServe / FastAPI

In order for a frontend client to communicate with the chatbot agent managed by LangChain, a web server was needed. For this, we ended up choosing LangServe, which is the sister project of LangChain that enables it to be accessed via a REST API. Under the hood, LangServe uses the popular Python library FastAPI, which supports multiple authentication methods, including using an API key, and credentials-based login.

Another great feature of LangServe is his JavaScript library, which enables our frontend for the user to run and communicate with LangServe. Here's the required code to send a request from the frontend.

```
export async function chat(question: string, button: HTMLElement) {
  let chatBubble = addChatBubble("", chatBubbleTypes.Bot) as HTMLElement

  const remoteChain = new RemoteRunnable({
    // the docker environmental variable is used to get the server address
    url: `${import.meta.env.PUBLIC_API_SERVER}`
  })

  // creates a promise with empty chat history as it's managed by the server
  // and the user question. Only one request at a time is allowed.
  const stream = await remoteChain.stream(
    {
      "chat_history": [],
      "question": question
    },
    { maxConcurrency: 1 }
  )

  // streams the response chunk by chunk and updates the corresponding
  // chat-bubble custom element
  for await (const chunk of stream) {
    chatBubble.setAttribute(
      "message",
      chatBubble.getAttribute("message")! + chunk
    )
  }
}
```

Code 1. Sending a request from the frontend and displaying the result in a chat bubble, written in TypeScript.

In this code snippet, a request is sent from the frontend to the LangServe server, containing the question and an empty chat history (as it's managed by the backend). The server then streams a response in chunk, which are appended to the "message" attribute of the newly created chat bubble.

LangServe and the LangChain code is running in a docker container called "server". It requires the containers "ollama" (serving the LLM) and "vectorstore" (where the documents are stored) to run.

4.3 Storing the document on a vector store – Milvus

As explained earlier, a vector database / vectorstore is needed in order to store the documents in a format the LLM can understand. We searched for the most performant and mature open source vector store, and ended up using Milvus.

According to the open source VectorDBBench benchmark by Zilliz, Milvus is the winner in handling the most query per second (QPS) (Zilliz, no date). But even as the benchmark is open source, Zilliz is the company behind Milvus. In full transparency, I didn't know that in advance and trusted this benchmark.

Nonetheless, other vector benchmarks also rate Milvus highly in performance. Qdrant, which is another vector database company, rank Milvus second place in their own benchmark (Qdrant, 2024), behind their own product of course.

According to Google Trends, the popularity of searching for the term “vector database” really exploded last year, with an all-time high popularity between the March 10th and March 16th. It is currently hard to find independent research on the most performant vector database products, so deciding which vector store to choose is also difficult.

Despite that, Milvus still works well for our use-case. At the risk of sounding like a broken record, I couldn't test Milvus on production hardware, but it worked well enough on my personal computer.

Taking all of that into account, and in the case that another solution would replace Milvus would come in the future, I've put in place an abstract class that serves as a wrapper to a LangChain's “VectorStore” object. All the required methods to manage a vector store with the server are declared, making it possible to extend the application to use another provider more easily.

```

class VectorstoreManager(ABC):
    @property
    def vectorstore(self) -> VectorStore:
        return self._vectorstore

    @property
    def collection_name(self):
        return self._collection_name

    @collection_name.setter
    def collection_name(self, value):
        self._collection_name = value

    @property
    def embedding_function(self):
        return self._embedding_function

    @embedding_function.setter
    def embedding_function(self, value):
        self._embedding_function = value

    @property
    def connection(self):
        return self._connection

    def __init__(self, vectorstore: VectorStore, connection_args, collection_name,
embedding_function):
        self._vectorstore = vectorstore
        self._embedding_function = embedding_function
        self._connection = self.connect(connection_args)
        self._collection_name = collection_name

    @abstractmethod
    def connect(self, connection_args):
        """Connects to the vectorstore db"""
        pass

    @abstractmethod
    def get_collection(self):
        """Get vectorstore db collection"""
        pass

    @abstractmethod
    def delete_document(self, document_id: str):
        """Removes a document from the vectorstore"""
        pass

    @abstractmethod
    def get_collection_json(self):
        """Get vectorstore collection for client consumption"""
        pass

    @abstractmethod
    def collection_exists(self):
        pass

    @abstractmethod
    def search_value(self, value):
        pass

```

Code 2. Custom abstract Python class named *VectorStoreManager*, used to manage a vector store.

4.4 The chatbot will need to cite the source it uses for its answers

Currently, the chatbot gets access to a single document with the highest probability to be the right one. This prevents the chatbot from returning conflicting information.

Regarding prompt engineering, here's a template I've found to be working :

```
You are a chatbot responsible for answering questions from the user about documents you have access to.
```

```
Never refer to online documents, only the documents you have at your disposal. Always cite the exact name of your source.
```

```
Here's the history of messages exchanged with the user:
```

```
{chat_history}
```

```
Next question: {question}
```

The answer template used is the following :

```
Answer the question using this context:
```

```
{context}
```

```
Question: {question}
```

The prompts are a modified version of the default ones provided by LangChain in their LangServe RAG example (LangChain, 2024).

The variables in {brackets} will be replaced by the server to their corresponding value. More information on the subject can be found in the “Managing chatbot prompts” chapter.

To give an example, the chatbot has access to two unrelated documents. One of them is the file “PDF.pdf” with the following content :

Example document

```
Alexandre Zürcher is an exchange student from Geneva, Switzerland and is currently in the last month of his Bachelor in Business Information Technology. As a typical Swiss citizen, he really loves cheese and banks. Sometimes, people confuse his name to be Christopher, but he doesn't mind.
```

The other document is a short text about Docker called “docker.txt”, entirely unrelated to the first text.

When the chatbot is asked “What do you know about Alexandre Zürcher ?”, it gives the following answer:

```
Based on the context provided in the example document, Alexandre Zürcher is an exchange student from Geneva, Switzerland who is nearly finishing his Bachelor's degree in Business Information Technology. He is known to enjoy cheese and banking, which are typical aspects of Swiss culture. Sometimes, people mistake his name for Christopher, but he doesn't mind. However, the document does not provide any specific details related to your query. If you have a particular question regarding Alexandre Zürcher or if there is
```

a specific document or source that you can share with me, please let me know and I will try my best to help answer your query based on that information.

No mention of docker was made in the chatbot's response, and the answer is based on the text only, without referencing to an online article about Alexandre Zürcher.

Similarly, asking "what do you know about docker ?", the chatbot answers:

Based on the context you provided and the whitepaper description, Docker is an open-source platform that uses containerization technology to package applications and their dependencies into lightweight, portable containers. These containers ensure consistent runtime and can run across different environments while being isolated from each other and the host system for security and reliability purposes. Docker has both free and premium tiers, with its service called Docker Engine, which was first released in 2013 by Docker, Inc.

No mention of Alexandre Zürcher is made this time, yet no mention of "docker.txt" is made, either.

After looking a bit more into the issue of not mentioning the source, I saw that this actually wasn't an information passed to the chatbot directly. Only the textual content of a document similar to the prompt is passed to the chatbot, but not its title.

Even so, it is possible for LangChain to return the document name used as source as metadata in the response. This could be used to display in a separate and visual manner the document name in the chatbot response. I discussed this proposal with Innovatim, and both parties agreed that it would be a "nice-to-have" feature, but wasn't the main focus for now.

As the thesis grew in scope, this feature didn't make it in the current version of the software.

4.5 User story - Support PDFs, Word documents, Excel sheets and plain text files

LangChain provides access to a huge amount of the LangChain's "BaseLoader" python classes, which is used to convert documents to plain text, which can then be converted into vector data.

In particular, the Unstructured.IO library integration provides all the classes necessary to support the 4 types of files, and more (LangChain, no date c).

```
def document_loader_strategy(filepath: str) -> tuple[Optional[BaseLoader], str]:
    """Uses supported langchain-supported document loaders depending on file extension"""
    # get the extension of the file
    file_name, file_extension = os.path.splitext(filepath)

    file_extension = file_extension.replace(".", "")

    match file_extension:
        case "pdf":
            return UnstructuredPDFLoader(filepath), "PDF"
        case "xls" | "xlsx":
            return UnstructuredExcelLoader(filepath), "Excel"
        case "csv":
            return UnstructuredCSVLoader(filepath), "CSV"
        case "doc" | "docx":
            return UnstructuredWordDocumentLoader(filepath), "Word"
        case "ppt" | "pptx":
            return UnstructuredPowerPointLoader(filepath), "PowerPoint"
        case "txt" | "text" | "md" | "log":
            return UnstructuredFileLoader(filepath), "Text"
        case _:
            return None, ""
```

Code 3. Python function used to return the right BaseLoader class depending on the file's extension.

This function, which follows the strategy design pattern, is used each time a document is added to the vector store. It's also used to set the metadata of the file to an easily readable file format name, which is displayed in the administrative user interface's list of files on the vector store.

5 User story – Easily change which LLM model is used

5.1 Choosing an open-source LLM model

As the LLM field develops very rapidly at the time of writing this thesis, Innovatim and I thought it would be necessary to use a system where the model used could be easily updated without making changes to the codebase.

During the development phase of this project, I didn't have access to a test server where a LLM would run, so I had to run the model on my own hardware.

The process of running a LLM locally is very intensive, and typically require a Graphics Processing Unit (GPU) to run at sufficient speed. During the development process, I used my own GPU, whose specific model is NVIDIA RTX 3070, to run various models to test which would be the most suitable for the project's scope.

As the LLM field develops very rapidly at the time of writing this thesis, Innovatim and I thought it would be necessary to use a system where the model used could be easily updated without making changes to the codebase, which is why we ended up using Ollama to manage the chatbot's LLM.

The "right" model isn't an easy answer. Many factors could influence the model to choose, including the main language spoke at the company, whether their activity is related to a particularly niche field, the nature of documents that the chatbot would interact with, but also how demanding the model is on the hardware.

LLMs benchmarking is a subject of sometimes heated discussion, as some say that some LLMs may have been trained on test data, which results in great result in benchmarks, but less stellar results in actual use (AIM Research, 2023). After having tested the Llama 2 and Mistral 7B models ourselves for a while, Innovatim and I felt that it was the most versatile of the two was Mistral 7B.

More testing will have to be conducted in order to arrive at a more definitive answer, but for this thesis, we settled on using that model.

5.2 Containerization of the LLM

In order to manage the LLM model used, Innovatim wanted to use Ollama, a frontend for the llama.cpp project, available here : <https://github.com/ggerganov/llama.cpp>.

The main goal of both Ollama and llama.cpp is to provide easy access to high performance, pre-trained LLMs that can respond to a user query. Ollama helps further by granting access to various models such as Llama3, Mistral 3B v0.2, and many others.

Ollama can be deployed in a docker container with support for NVIDIA GPU, which as I mentioned earlier enable way better performance than running on CPU alone.

To get an idea of how easy it is to deploy a LLM using Ollama, here's the 2 docker commands needed to get started on a machine with an NVIDIA GPU (Ollama, 2023) :

```
# "-d" - the output of the container won't be shown in the current terminal
# "--gpus=all" - grants the container access to all the GPUs of the system
# "-v ollama:/root/.ollama" - persists the cached models
# "--name ollama" - set name of container to "ollama"
# "ollama/ollama" - uses Docker image https://hub.docker.com/r/ollama/ollama
docker run -d --gpus=all -v ollama:/root/.ollama --name ollama ollama/ollama

# first pulls the mistral LLM model, then the user can query the LLM in the
# terminal
docker exec -it ollama ollama run mistral
```

Code 4. Running a LLM using Ollama in a Docker container using bash commands.

Ollama is running in one of the containers described in the main docker-compose.yaml file, and is required by the main server to run.

I searched online for alternatives to Ollama, but no other options provided the same easy docker deployment and seamless installation of a new model in just one command. Also, a python library is available, which was used on the server to check the presence of the LLM model, and automatically download it in case it's not cached.

As it uses HTTP, it is also possible to connect to a remote Ollama instance. My laptop connects to my NVIDIA-powered computer at home, and get access to the Ollama docker container running on it. This gave me the freedom to work anywhere, while still using the power of my NVIDIA GPU.

5.3 User story – The chatbot will be able to answer in English, French, German and Italian

Choosing a LLM model directly influences which languages the chatbot itself speak. As explained in the previous section, I was limited with my personal NVIDIA GPU's available VRAM, so larger models than Mistral 7B couldn't be run. I personally tried to query the chatbot in both French and English, and had better results in English than in French, which sometimes would still answer in English.

Various models derived from Mistral with additional specialization in other languages are available from various third parties on Hugging Face, notably Mistral-Ita-7b by DeepMount00 for the Italian

language (Michele Montebovi, 2024), `Claire-Mistral-7B-0.1` from OpenLLM-France for French (OpenLLM France, 2023), and `em_german_leo_mistral` by user `jphme` for German (Jan Philipp Harries, 2023).

I initially planned on testing more models after gaining access to the production environment, but as it didn't happen during the duration of the thesis, this is something that I will focus on privately with Innovatim. If a general usage model isn't available, I plan on testing these models in collaboration with the commissioning company.

6 User story – Have an intuitive user interface

As the solution has to be embedded in a parent website, I suggested using either vanilla JavaScript, or a very light framework, in order for the user interface to be as light as possible. After some research, Astro seemed like the most promising one, for a few reasons.

Firstly, no JavaScript is included in the built package of Astro by default, only HTML and CSS. Less JS means less potential slowdowns, and as the chatbot will be embedded onto another site with most likely already a lot of JavaScript, it's necessary to keep that from happening.

The framework also supports TypeScript out-of-the-box, which is a great way to keep unexpected bugs from appearing at build time, and have an overall better developer experience with great auto-completion functionalities.

6.1 Using Web Components

Web Components are defined by MDN Web Docs as “a suite of different technologies allowing you to create reusable custom elements – with their functionality encapsulated away from the rest of your code – and utilize them in your web apps” (MDN, 2024b).

This is a modern and simple way to built web applications, which still supports all the major browsers (Chrome, Edge, Firefox and Safari) since 2020.

I wanted to explore this frameworkless, standardized way of building UI components, as I was tired of often switching from one frontend frameworks to another on different projects. I also thought about the performance benefit of not having to ship React or a similar framework to the client, which seemed appealing for our use case.

To illustrate, let's now see a concrete example of using custom components.

Here, a “CustomComponentEx” class will be defined, and then added to the DOM with the attribute “text” set to “Hello”. This will result by the text “Hello” written in the browser. After 1 second, the “text” attribute is updated to “Hello World”, which updates the text shown to that value.

```

class WebComponentEx extends HTMLElement {
  // sets to observes the text attribute, like this :
  // <custom-component-ex text="hello world"></custom-component-ex>
  static observedAttributes = ["text"]

  // Class attribute for a paragraph, that will be used to update the
  // text when corresponding "text" attribute is updated
  textParagraph: HTMLParagraphElement

  // Executes when a component instance is created
  constructor() {
    super()

    // Instanciate the span element that will contain the message
    this.textParagraph = document.createElement("p")
  }

  // executed when component is inserted into the DOM
  connectedCallback() {
    // Sets the initial content of the paragraph to the value of
    // the "text" attribute
    this.appendChild(this.textParagraph)
    this.textParagraph.innerText = this.getAttribute("text")!
  }

  // Triggers when an observed attribute is modified
  attributeChangedCallback() {
    // Updates the content of the paragraph as soon as the value of
    // the "text" attribute changes
    this.textParagraph.innerText = this.getAttribute("text")!
  }
}

// Maps the "web-component" tag name to the WebComponentEx
// class in the custom elements registry
customElements.define("web-component-ex", WebComponentEx)

// Generates a new instance of a WebComponent.
// Executes the constructor() method
let webComponentEx = document.createElement("web-component-ex")

// Sets the initial value of the "text" attribute to "Hello"
webComponentEx.setAttribute("text", "Hello")

// Adds the WebComponent instance to the top of the page's <body>.
// Executes the connectedCallback() method, which sets innerText of
// the paragraph to "Hello"
document.body.prepend(webComponentEx)

// updates the "text" attribute to "Hello World" after 1 second
setInterval(()=>webComponentEx.setAttribute("text", "Hello World"), 1000)

```

Code 5. Example of a Web Component lifecycle.

The kind of logic illustrated in this code snippet is used multiple times in both the admin and user frontends.

Regarding the styling, most of the CSS is managed in the main “index.astro” file of each project. There also is a CSS file called “variables.css”, shared by both the user and admin frontends, which serves as a central place where CSS variables that affect both frontends are declared.

This makes for an easier way to customize a chatbot for each of Innovatim’s client.

Here’s an example of how a few CSS variable in “variables.css” :

```
:root {
  --btn-padding: 0.5rem;
  --btn-border: none;
  --btn-background-color: #9cd34c;
  --btn-text-color: black;
  --btn-active-border: none;
  --btn-active-background-color: green;
  --btn-active-text-color: lightgreen;
}
```

Code 6. CSS variables used in the frontend applications, as declared in the file “variables.css”.

I made sure the variables name are as self-explanatory as possible. In the case of this file, the variables shown manage the color, padding, border-radius and margin of a button.

This strategy was applied for many elements of the design of both user and admin frontends: margin, padding, colors, shadows, border (width, color and style), font family, transition time for animations, and the dimensions (width and height) of various elements such as the button that toggles the chatbot.

6.2 The chatbot user interface

The main user interface used by the end-user is a chatbot window that is minimized by default, and only a small icon is shown, with a robot icon.

Clicking the button fades in the window, and automatically focuses the text input in order to let the user type right away.

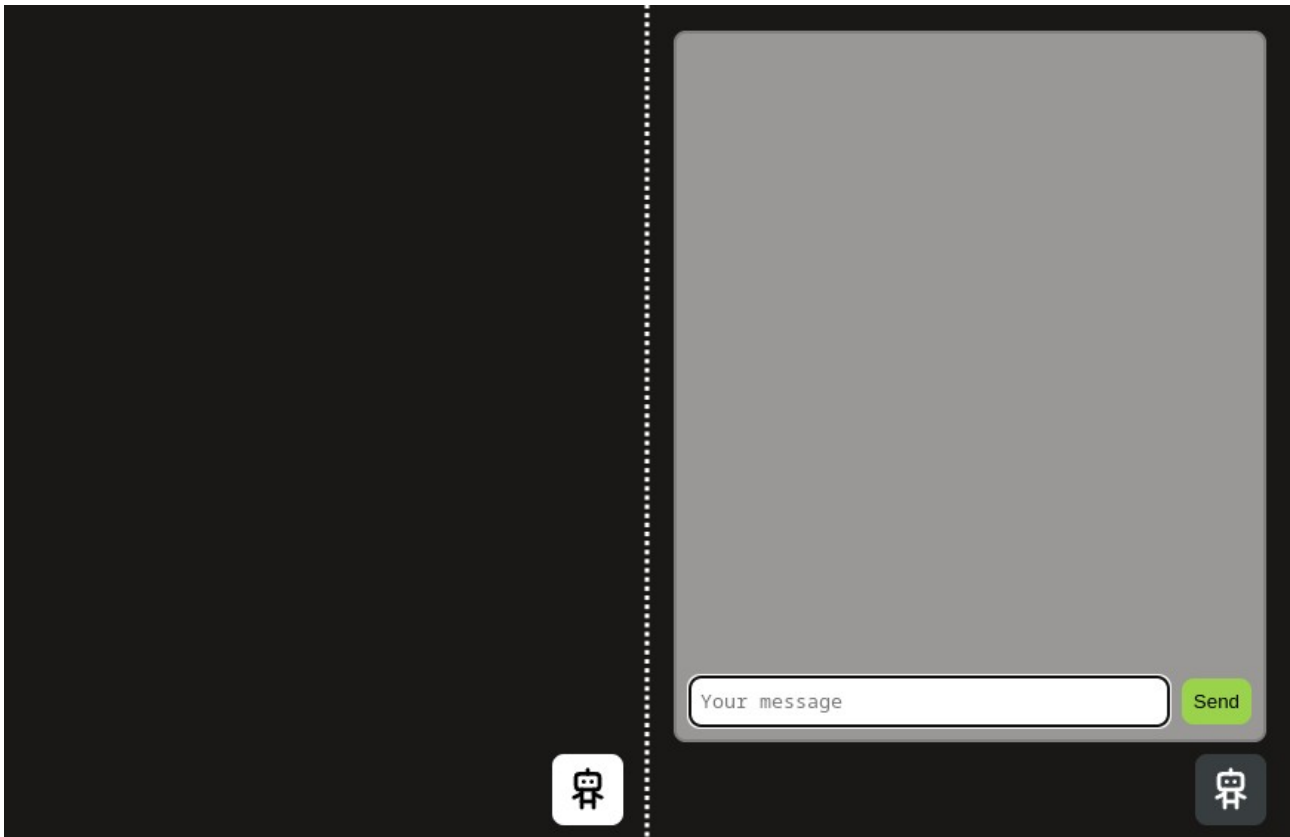


Figure 3: 2 screenshots of the chatbot user interface, on one the chatbot window is minimized, on the other it is shown.

The design is kept as simple as possible to make sure non-technical user can interact without an issue with the chatbot, following the “Aesthetic and Minimalist Design” principle from Jakob Nielsen’s 10 general principles (Jakob Nielsen, 2024).

Most elements of the design can be modified to suit the parent website’s own graphic design / branding.

After writing a message, the user can either send it by clicking the green “Send” button, or by using the “Enter” key. This behavior was made to be, again, as intuitive as possible.

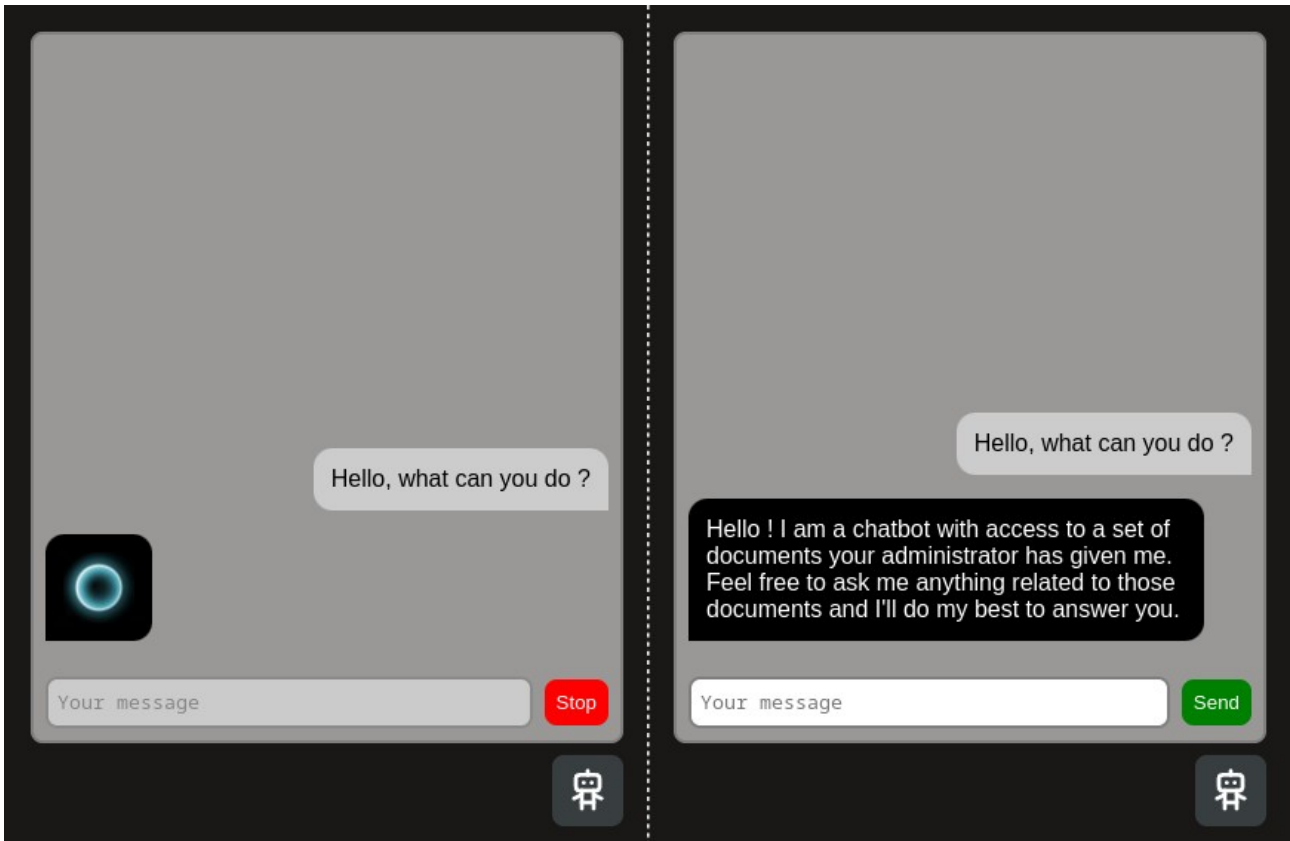


Figure 4: Side by side screenshots of the frontend for an user, showing a response being generated, and a full, generated response.

As soon as a message is sent, a new chat bubble containing the user's message is displayed on the right side of the window. This follows the way that most messaging application use, such as WhatsApp, Telegram, Signal, etc.

Until the chatbot starts answering, a loading animation is displayed (see left screenshot on above figure), which makes the end-user understands that something is going on. The text is then streamed in chunks in the same way that ChatGPT uses. During this time, the user input is disabled, to keep the user from sending a new query. The current response can be stopped by clicking the "Stop" button (see left screenshot on above figure).

When the chatbot is finished answering, the text input is focused again, and the "Stop" button turns back into a "Send" button (see right screenshot on above figure).

6.2.1 Displaying message bubbles

For the user frontend with which a user interacts by asking a question with a simple chat UI, here's what was used to create a message bubble :

```
class ChatBubble extends HTMLElement {
  static observedAttributes = ["message"]

  messageSpan: HTMLSpanElement

  constructor() {
    super();
    // instanciate the span element that will contain the message
    this.messageSpan = document.createElement("span")
  }

  connectedCallback() {
    const containerP = document.createElement("p")
    this.appendChild(containerP)

    const loadingImg = document.createElement("img")
    loadingImg.classList.add("fade-in")
    loadingImg.src = "/animations/loading.gif"
    containerP.appendChild(loadingImg)

    containerP.appendChild(this.messageSpan)
    this.messageSpan.innerText = this.getAttribute("message")!
  }

  attributeChangedCallback() {
    this.messageSpan.innerText = this.getAttribute("message")!
  }
}

// maps the "chat-bubble" name to ChatBubble class in custom elements registry
if (!customElements.get("chat-bubble")) {
  customElements.define("chat-bubble", ChatBubble)
}
```

Code 7. Custom ChatBubble Web Component in JavaScript, used to display a chat bubble in the user interface.

Then, the helper method “addChatBubble()” is used to append either a user or a bot chat bubble :

```
export function addChatBubble(message: string, type: chatBubbleTypes) {
  const messageList = document.getElementsByClassName("message-list")[0] as HTMLElement
  let element = document.createElement("chat-bubble")
  element.setAttribute("message", message)
  element.setAttribute("type", type)
  messageList.prepend(element)

  return element
}
```

Code 8. Helper function to add a chat bubble to the conversation UI, written in TypeScript.

As for the styling, most of it is managed in the file “index.astro” :

```

.message-list {
  display: flex;
  flex-direction: column-reverse;
  height: 100%;
  border-radius: var(--border-radius);
  overflow-y: auto;
  padding-right: calc(var(--main-padding) / 2);
}

/* required to keep message list sticky to bottom */
.message-list #anchor { overflow-anchor: auto; }

chat-bubble {
  max-width: 100%;
  padding: var(--bubble-padding);
  margin-bottom: var(--bubble-margin);
  transition: all var(--transition-time);
}

/* user and bot have different style, and are aligned in separate directions */
chat-bubble[type=user] {
  background: var(--user-bubble-color);
  margin-left: 2rem;
  color: var(--user-text-color);
  align-self: flex-end;
  border-radius:
    var(--bubble-border-radius)
    var(--bubble-border-radius)
    0
    var(--bubble-border-radius);
}

chat-bubble[type=bot] {
  background: var(--bot-bubble-color);
  margin-right: 2rem;
  color: var(--bot-text-color);
  align-self: flex-start;
  border-radius:
    var(--bubble-border-radius)
    var(--bubble-border-radius)
    var(--bubble-border-radius)
    0;
}

chat-bubble p { margin: 0; }

/* the loading image is shown if the message attribute is empty */
chat-bubble[message=""] { aspect-ratio: 1/1; }
chat-bubble[message=""] img { display: block !important; }
chat-bubble img {
  display: none;
  height: 3rem;
}

```

Code 9. CSS used for styling the chat bubble components.

Here, various CSS variables defined in the central “variables.css” file are used, such as “--border-radius”, “--main-padding”, “--transition-time” and “--bubble-border-radius”. The list of messages is a flexbox, which contains the various bubbles in the conversation, from both the user and the bot.

6.2.2 Minimizing the popup and embedding the chatbot in an iframe

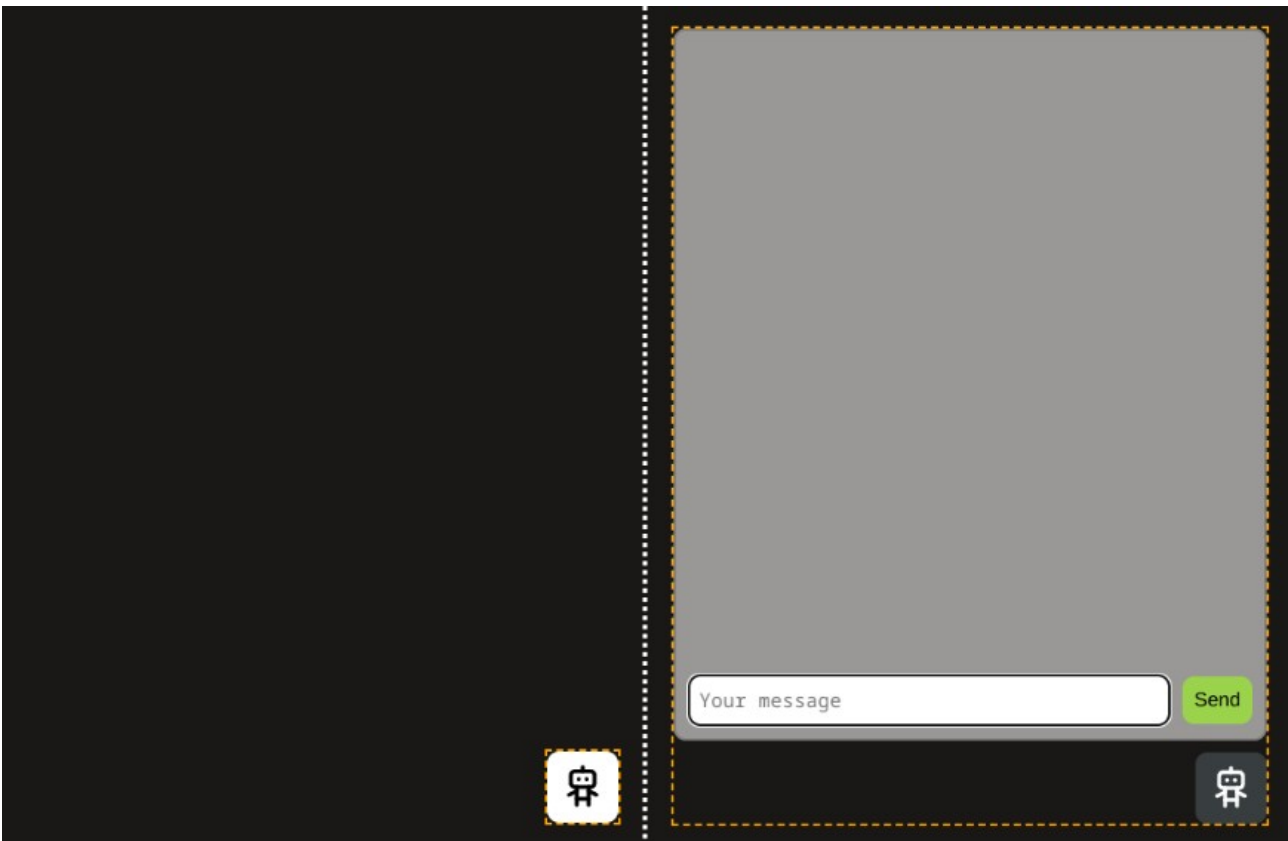


Figure 5: Screenshots of the chatbot window, one minimized and one maximized. An orange dashed border is used to represent the width and height of the iframe containing the chatbot.

When the time to integrate the chatbot on a website with an iframe came, a problem arose: by default, an `<iframe>` HTML tag's width and height isn't reactive to the children's. This means that the iframe was either always too big and covered the parent's page elements, or too small and never showed the maximized chatbot window.

I found the library `iframe-resizer` (Bradshaw, 2024), that worked well enough by sending messages from the child (the chatbot window) to the parent (the website that embed the chatbot window). However, the build size grew drastically from adding this library, which I wasn't happy with. For this reason, I decided to implement the same concept myself directly in the project, without a library dependency.

```
function postDimensionsToParent(): void {
  const boundingClientRect = document.body.getBoundingClientRect() as DOMRect
  window.parent.postMessage({ "ChatbotHeight": boundingClientRect.height }, "*")
  window.parent.postMessage({ "ChatbotWidth": boundingClientRect.width }, "*")
}
```

Code 10. TypeScript code used to send the chatbot dimension to the parent website.

This function is fired on the initial page load, where the chatbot window is minimized. The method `postMessage()` is used to communicate between the window objects of the chatbot and the parent. Another advantage of this method is that it works on different origins, which supports our use-case, as the chatbot is on a different domain than the parent website.

Now, the parent receives a message from the children. But how can it use this message to adapt the iframe's dimensions ?

For this, a separate JavaScript file named "resize-iframe.js", containing the following code :

```
const chatbotId = document.currentScript.getAttribute("chatbot-id"),
      chatbot = document.getElementById(chatbotId)

// listens for "ChatbotHeight" and "ChatbotWidth" messages containing possibly new iframe
// dimensions
window.addEventListener('message', (event) => {
  if (event.data.hasOwnProperty("ChatbotHeight")) {
    chatbot.style.height = event.data.ChatbotHeight + "px"
  }

  if (event.data.hasOwnProperty("ChatbotWidth")) {
    chatbot.style.width = event.data.ChatbotWidth + "px"
  }
})
```

Code 11. JavaScript code used by the parent to update the chatbot's width and height.

As this file was added in the "public" subdirectory, it's shipped as-is and can be called by the parent. It works by getting the chatbot element using its ID attribute, passed by the parent as a parameter. Then, it applies the values of the chatbot it received via the "ChatbotHeight" and "ChatbotWidth" messages directly to the iframe.

```
<iframe
  id="chatbot"
  src="http://localhost:9191?x-token=secret-token"
  frameborder="0"
  style="position:fixed;right:1rem;bottom:1rem;z-index:100;height:2rem;width:2rem;">
</iframe>

<script src="http://localhost:9191/resize-iframe.js" chatbot-id="chatbot"></script>
```

Code 12. HTML iframe and script declared on the parent.

Each time the toggle button of the chatbot window is clicked, the "postDimensionsToParent()" mentioned earlier is fired, and the dimension update automatically.

6.2.3 Managing errors

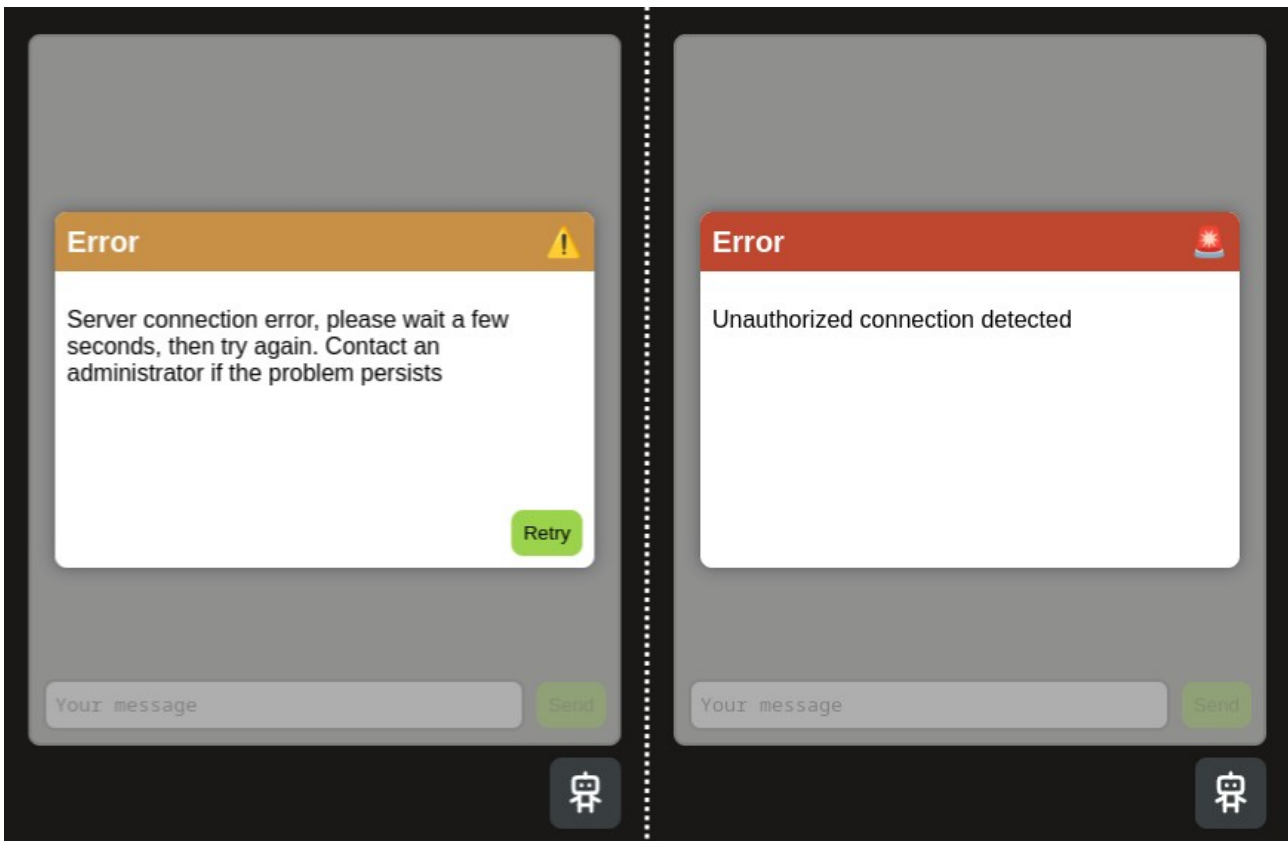


Figure 6: Screenshots of the chatbot window, with an error message for connecting to the server, and an unauthorized error message.

Sometimes, things break. Having an error message that explains the situation is a necessity, which is why a message is displayed in the form of a popup when certain errors are thrown.

On the left screenshot of the above figure, the frontend hasn't managed to make a connection to the server. It is possible that the server was restarted to switch the model used, the embeddings, the prompts could have been updated, a maintenance update, ...

Many options are possible, but as the server is only temporary down for most of them, the error message let the user the option to retry using a button of the same name. A timeout of 1 second is present in order to prevent user spamming the button. The text also indicates that an administrator should be contacted in the event that the problem persists. This is also the Jakob Nielsen's principle "Help Users Recognize, Diagnose, and Recover from Errors" (Jakob Nielsen, 2024).

The right screenshot is less informative by design. For any request from the chatbot interface to the server, the right API key has to be specified in an HTTP header. In case this API key is incorrect or unspecified, the server will return an "Unauthorized" response with a 401 status code.

If this happens, it's probably because someone is trying to access the chatbot interface from somewhere else than the parent website. Therefore, the error message stays vague to give no hint to the potential attacker as to how authentication work.

More details on popups can be found in the later chapter "Displaying popups".

6.3 The administrator user interface

Chatbot admin

Prompt - Context template

```
You are a chatbot responsible for
answering questions from the user.

Follow these rules :
- Never answer a question with another
question.
- Always answer with the most relevant
documentation page.
```

Prompt - Answer template

```
Answer the question using this context:
{context}

Question: {question}
```

Update

Files on vectorstore

Select files to upload

or drag and drop files here

PDF.pdf

Upload

<input type="checkbox"/>	File name	Type	Date added
<input type="checkbox"/>	Excel.xlsx	Excel	2024-05-19
<input checked="" type="checkbox"/>	PowerPoint.pptx	PowerPoint	2024-05-19
<input type="checkbox"/>	CSV.csv	CSV	2024-05-19

Delete selected

Figure 7: Screenshot of the administrator panel, with the chatbot's prompts, a list of files present on the vectorstore, and a drop box to upload new files

The administrator interface has multiple uses: it's possible to see and update the chatbot's context and answer prompts, a list of files on the vector store can be found, new files can be uploaded, and also deleted.

This user interface uses the same shared CSS variables from the central "variables.css" file, keeping a similar look and feel as the chatbot user interface. The Astro framework is used as well.

Every string of the interface is translated in 4 languages, using the "t(translation_key)" function that was explained previously.

Making use of a flexbox layout, the items automatically wrap on smaller screen sizes, making it mobile friendly as well.

6.3.1 Managing documents on the vector store

Adding a document to the vector store was made to be as easy as possible, in the pursuit of a simple user interface.



Figure 8: Steps to add a new document to the vector store.

In order to add one or more files to the vector store, it's possible to either click the “Select files to upload” button, or to drag and drop the files on the green box.

After this, the green box with split in two, which reveals the list of the files that will be uploaded to the server. Next to each file is the package emoji “📦”, to symbolize that the file is ready to be sent to the server.

Instead of immediately uploading the file, the user can verify if he added the file(s) they wanted to, and finally press “Upload” when ready. The “Upload” button is then disabled, and the files are sent from the browser to the server.

While it's uploading, a loading bar is shown, and the emoji is updated to a shipping truck “🚚” (not pictured on the above figure). As the server receives the entire file, the emoji is updated again to an inspecting emoji with a monocle “🕵️”, which conveys that the server is inspecting the file to add to the vector store.

Finally, when the file is added, the emoji is updated to a green checkmark “✅”, showing that the upload was finished successfully. The table below also automatically updates with the newly added file's name, file type and date of upload.

The button “Delete selected” also appears, initially disabled, indicating a new option for the end-user.

6.3.2 Managing chatbot prompts

In order for the chatbot to work, certain essential variables need to be present : “**{question}**” contains the user's query, “**{context}**” passes documents from the vector store, and “**{chat_history}**” holds the chat history between the current user and the bot.

To verify the presence of said variable, a check on both the frontend and the backend is made. If one of the template doesn't contain the variable, no request is sent to the server, and instead a popup is shown explaining to the user that missing variables were found.

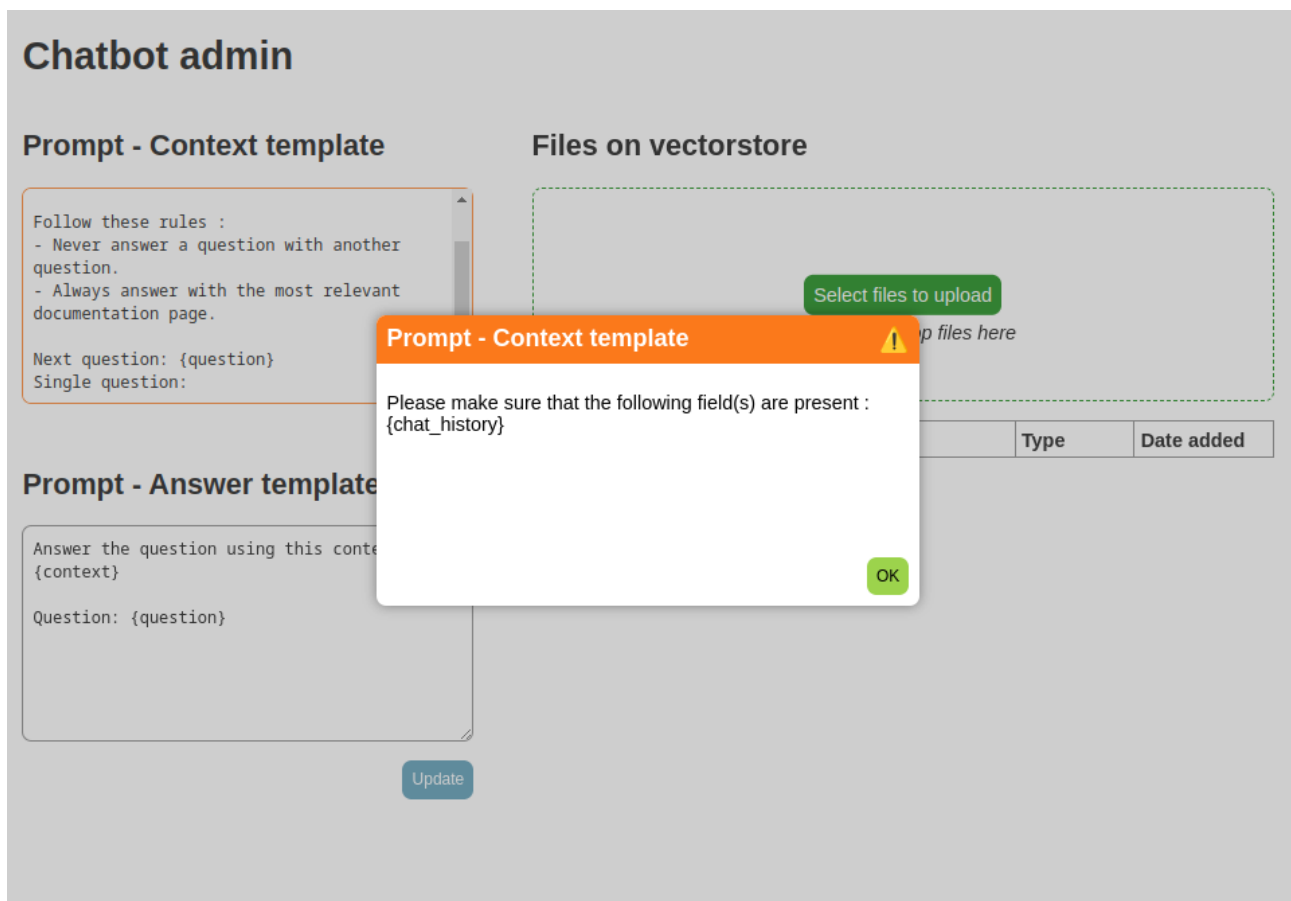


Figure 9: Administrative interface, where a variable is missing from the context prompt template

After clicking “OK”, the pop-up disappears and the text area with the missing variable(s) is highlighted in orange to indicate visually where the problem is. This is in accordance to the “Help Users

Recognize, Diagnose, and Recover from Errors” principle from Jakob Nielsen (Jakob Nielsen, 2024).

6.4 Functionalities shared between chatbot user and administrator user interface

Some functionalities are shared between both the chatbot and the administrator users. This section goes through the ones that were used in pursuit of the “have an intuitive user interface” goal.

6.4.1 Implement internationalization with 4 languages

In order to support multiple languages, the helper functions “t(translation_key)” is shared to the admin and user frontends :

```
// gets the language based on the user agent of the browser
function getLangFromUserAgent(): string {
  if (navigator.languages != undefined)
    return navigator.languages[0]
  return navigator.language
}

// return the supported language code, english is the fallback
function getSupportedLanguage(navigatorLanguage: string): string {
  if (/fr.*/.test(navigatorLanguage)) {
    return "fr"
  } else if (/it.*/.test(navigatorLanguage)) {
    return "it"
  } else if (/de.*/.test(navigatorLanguage)) {
    return "de"
  } else {
    return "en"
  }
}

const defaultLang = "en"

export function useTranslations() {
  let lang = getSupportedLanguage(getLangFromUserAgent()) as keyof typeof ui

  return function t(key: keyof typeof ui[typeof defaultLang]) {
    return ui[lang][key];
  }
}

// was shorten for the sake of brevity
export const ui = {
  en: {
    "admin.title": "Chatbot admin",
    "input.send": "Send",
  },
  fr: {
    "admin.title": "Chatbot admin",
    "input.send": "Envoyer",
  },
  it: {
    "admin.title": "Amministratore di chatbot",
    "input.send": "Inviare",
  },
  de: {
    "admin.title": "Chatbot-Verwaltung",
    "input.send": "Senden",
  }
} as const
```

Code 13. TypeScript code used to translate the user interface in 4 languages.

This works by first retrieving the language from the user's browser, and translate the UI based on that value. In case the language is not supported, English is used as a fallback.

The function “t(translation_key)” can then be used anywhere it's imported, and depending on the language of the user, the corresponding string of text will be shown :

```
<script>
  import { useTranslations } from '../../../shared/utils'
  const t = useTranslations()

  let title = document.querySelector("#page-title") as HTMLHeadingElement

  // i18n
  document.title = t("admin.title")
  title.innerText = t("admin.title")
</script>
```

Code 14. Assigning translation helper functions to title in TypeScript.

6.4.2 Displaying popups

Both the user and admin frontends use the same popup logic and styling, and are used to display closable information message, error messages, actionable error messages, and critical error messages. As such, the logic and styling of popups are centralized in a shared directory between the two frontend projects.

```
export enum PopUpTypes {
  Info = "info",
  Error = "error",
  ErrorRetry = "error-retry",
  Critical = "critical"
}
```

Code 15. PopUpTypes in TypeScript, used for type safety.

Each of the 4 pop-up type has different behavior, header color and emoji. Using an enumerated type permits to easily call a specific type of pop-up from elsewhere in the frontend apps.

```
import "./pop-up.css"
import { useTranslations } from "../../../shared/translations"
const t = useTranslations()
```

Code 16. Various imports used for the PopUp component.

As the popup element displays messages, the “t(translation_key)” is imported. A separate “pop-up.css” CSS file that manages the component styling is also imported.

```

class PopUp extends HTMLElement {
    static observedAttributes = ["title"]

    callbackFunction?: Function
    button: HTMLButtonElement
    titleHeading: HTMLHeadingElement
    typeEmojiSpan: HTMLHeadingElement
    contentParagraph: HTMLParagraphElement
    timeout: number

    constructor(callbackFunction?: Function) {
        super()
        // (not shown: instantiation of the class' properties)
    }

    closePopUp(): void {
        this.button.disabled = true
        setTimeout(()=>this.remove(), this.timeout)
    }

    connectedCallback() {
        let message = document.createElement("div") as HTMLDivElement
        this.appendChild(message)

        let header = document.createElement("div") as HTMLDivElement
        message.appendChild(header)
        this.titleHeading.innerText = this.getAttribute("title")!
        header.appendChild(this.titleHeading)

        const type = this.getAttribute("type") as PopUpTypes

        if (type == PopUpTypes.Critical) {
            this.typeEmojiSpan.innerText = "🚨"
        } else if (type == PopUpTypes.Error || type == PopUpTypes.ErrorRetry) {
            this.typeEmojiSpan.innerText = "⚠️"
        } else if (type == PopUpTypes.Info) {
            this.typeEmojiSpan.innerText = "ℹ️"
        }

        header.appendChild(this.typeEmojiSpan)

        this.contentParagraph.innerText = this.getAttribute("content")!
        message.appendChild(this.contentParagraph)

        if (type != PopUpTypes.Critical) {
            message.appendChild(this.button)

            if (type == PopUpTypes.ErrorRetry && this.callbackFunction) {
                this.button.innerText = t("retry")
                this.button.addEventListener("click", _=> {
                    this.closePopUp()
                    setTimeout(this.callbackFunction!, this.timeout)
                })
            } else {
                this.button.innerText = "OK"
                this.button.addEventListener("click", _=> this.closePopUp())
            }
        }
    }

    attributeChangedCallback() {
        this.titleHeading.innerText = this.getAttribute("message")!
    }
}

```

Code 17. TypeScript code used to manage the pop-up component logic.

The pop-up component is the most complex web component present in the project. It contains a header, itself with a title, a background color, and an emoji. The last two are determined by the `PopUpTypes` enumerated type.

The types `ErrorRetry` is a bit special, as the “Retry” button executes a function given as an optional parameter to the constructor, to then close the pop-up. The other types simply show an “OK” button that only closes the pop-up. The “Critical” type cannot be closed, as the error stops the application from running.

```

if (!customElements.get("pop-up")) {
  customElements.define("pop-up", PopUp)
}

export function showPopUp(
  parent: HTMLElement,
  title: string,
  content: string,
  type: PopUpTypes,
  callbackFunction?: Function
) {
  let popUp

  if (callbackFunction) {
    popUp = new PopUp(callbackFunction)
  } else {
    popUp = new PopUp()
  }

  popUp.setAttribute("title", title)
  popUp.setAttribute("content", content)
  popUp.setAttribute("type", type)

  popUp.classList.add("fade-in")

  parent.prepend(popUp)
}

```

Code 18. Define custom components, and exported helper function to display pop-up in TypeScript.

After the declaration, the custom component is registered to use the “pop-up” tag.

Finally, the helper function “`showPopUp()`” is exported for usage anywhere in the app. It is called in multiple try-catch blocks in both the user and admin interfaces. In order to pass the callback function parameter to the constructor, the component is instantiated via JavaScript by calling “`new PopUp(callbackFunction)`”.

7 User story – Good security, so only authorized users are able to access private information

At first, the goal was to implement a credential-based authentication for both the client and the administrative interfaces. However, Innovatim deemed this flow to be too involved for the end-user, as the chatbot will only be present on pages where the user is already logged-in. Therefore, I had to research other security methods that would remain invisible to the end-user.

7.1 CORS validation

We started by exploring an insufficient but still necessary security feature, which is CORS, which stands for “Cross-Origin Resource Sharing”, a method to validate requests based on the “Origin” HTTP header’s value (MDN, 2024a). What we wanted to achieve by using this was to only allow the parent website’s URL to call the server.

However, I quickly understood that this won’t be possible, as the chatbot is embedded in an iframe. This means that the “Origin” header’s value would always be the chatbot’s URL, and not from the parent’s, which made CORS useless for this purpose. Spoofing CORS is also very easy, as a CURL script could easily add a fake “Origin” header :

```
curl 'https://chatbot-server.com/token/validate' \  
  -H 'x-token: secret-token' \  
  -H 'Origin: https://authorized-origin.com'
```

Code 19. CORS request with fabricated “Origin” header to bypass CORS.

Still, implementing CORS isn’t a bad thing, and keeps third party websites from making requests on the server.

7.2 Firewall rules

A possibility that I didn’t have the chance to explore is adding custom firewall rules. The only option I see possible would be to block IP addresses if a non-authorized request was made. This would only be a mitigation, as an attacker likely has access to multiple IP addresses.

As I didn’t have access to a production environment during the project, I couldn’t experiment with this option.

IP and domain-based are also out of the question, as the request all come from the end-user’s IP address, and not a verifiable central IP address.

7.3 Token-based authentication

The best way I found to secure the server is to use token-based authentication. Any request will need to have an “X-Token” header with a correct header, which can be easily changed by changing a docker environmental variable (see deployment chapter for more details).

On the frontend, the token is manually set as a query string parameter called “x-token”, which is then passed to each request to the server, as “http://localhost:9191/?x-token=secret-token”.

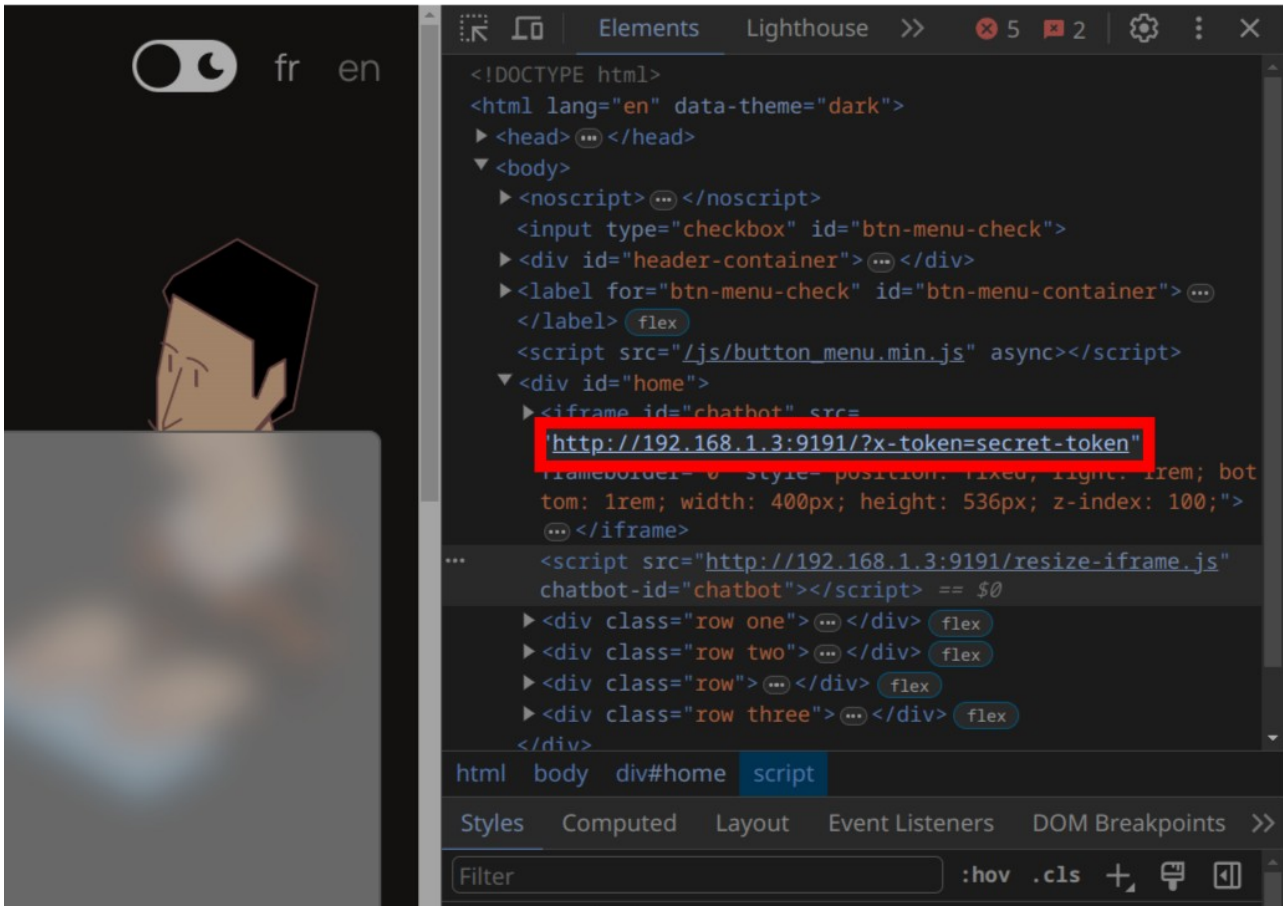


Figure 10: Opening the developer console and looking through the inspect tab reveal the secret token.

This is definitely not the strongest security mechanism: if someone malicious has access to the chatbot by logging into the parent's website, they can easily find the secret token in the query string of the iframe, letting them use the chatbot anywhere they want, even if they are not connected to the website.

I explained this compromise between security and convenience to the commissioning company, and they accepted the risk.

As for the administrative interface, Innovatim wanted a credential based authentication flow, which couldn't be implemented during the time of the thesis due to time constraints. As of now, it uses the same mechanism as the chatbot user interface.

8 User story – Deploy the chatbot on a Swiss infrastructure

As explained in previous sections, we chose Infomaniak’s Public Cloud product to deploy the application. Sadly, no environment with a GPU was available during the thesis process, which prevented me from working on actual deployment.

With this limitation, I focused on making the docker deployment as easy as possible. This will hopefully make the process of writing an OpenStack’s HOT template quick when I get access to a production environment.

8.1 Managing project level variables

Multiple project-wide variables are set in an “.env” file at the project root, and an “.env.example” is also present to get started quicker:

```
PUBLIC_API_SERVER="http://localhost:8000"
OLLAMA_MODEL="mistral"
# OLLAMA_HOST="http://192.168.100.3:11434" # for development ONLY
EMBEDDINGS_MODEL="hkunlp/instructor-large"
CORS_ORIGIN_USER="http://localhost:9191"
CORS_ORIGIN_ADMIN="http://localhost:9192"
AUTH_TOKEN="secret-token"
```

Code 20. Required environmental variables’ default value, as declared in the “.env.example” file.

“**PUBLIC_API_SERVER**” is used by both frontends, at build time the reference to that variable is replaced by the value, in this case “<http://localhost:8000>”, which is the server’s root.

“**OLLAMA_MODEL**” is the LLM model that the project uses. This value is read by the server, which first checks if the model is already downloaded in the “ollama” container. In case it’s not present, the model is pulled from Ollama’s servers, and cached into a docker volume.

“**OLLAMA_HOST**” is also passed to the server. It uses this address as a remote Ollama instance, instead of the default local one. This was used during development when I wanted to have access to my NVIDIA GPU from outside my apartment.

“**EMBEDDINGS_MODEL**” is, as its name indicates, the embeddings model used to transform text into vectors. This is a reference to a hugging face repository, which the server reads. In case it’s not already present, it’s downloaded from hugging face, and cached into a docker volume.

“**CORS_ORIGIN_USER**” and “**CORS_ORIGIN_ADMIN**” are both used by the server to determine which URL are allowed to make requests.

Finally, “**AUTH_TOKEN**” is the token’s value that the server checks when it receives any request.

This method is really useful, as for example to update the authorization token, updating the value in the “.env” file and recreating the server container is all it takes.

8.2 Running the application in containers

In production, 3 docker images have to be built for deployment: the server, and both user and administrative backends. The ollama and vectorstore containers run pre-made images.

For brevity's sake, I will only go through how the server is set up, which is the most complex container in use in the project. Here's what it looks like in "docker-compose.dev.yaml":

```
services:
  server:
    build:
      context: server
      args:
        CPU_OR_NVIDIA: "cpu"
        PYTHON_VERSION: "3.11"
    environment:
      CORS_ORIGIN_ADMIN: ${CORS_ORIGIN_ADMIN}
      CORS_ORIGIN_USER: ${CORS_ORIGIN_USER}
      OLLAMA_MODEL: ${OLLAMA_MODEL}
      OLLAMA_HOST: ${OLLAMA_HOST}
      EMBEDDINGS_MODEL: ${EMBEDDINGS_MODEL}
      AUTH_TOKEN: ${AUTH_TOKEN}
    depends_on:
      vectorstore:
        condition: service_healthy
    volumes:
      - ./server/app:/code/app
      - server_cache:/root/.cache
      - server_nltk_data:/root/nltk_data
    ports:
      - "8000:80"
volumes:
  server_cache: {}
  server_nltk_data: {}
```

Code 21. The server container, as defined in the main docker-compose.yaml file.

Many things are happening here.

First, the "context" directive specifies that the Dockerfile used to build the server is in the "server" directory. The arguments "CPU_OR_NVIDIA" and "PYTHON_VERSION" are both passed to the Dockerfile, and will change how the image is built.

After that, multiple environment variables are passed to the container itself (not the image!), which are described in more detail in the previous section.

The "depends_on" directive makes the orchestrator wait for the vector store to be in a healthy state before running the server container.

Some volumes are used to keep the embeddings from Hugging Face even if the container is destroyed. They are also defined lower in the file, as the docker-compose.yaml specs requires.

Finally, the port 8000 of the docker host is mapped to the port 80 of the container, so it can receive requests from the clients.

```
ARG CPU_OR_NVIDIA
ARG PYTHON_VERSION

#####
# used to build the base image running on CPU only
FROM python:${PYTHON_VERSION} as base-cpu

WORKDIR /code
COPY ./requirements.txt /code/requirements.txt

RUN apt update && apt install libgl1 -y
RUN pip install --no-cache-dir --upgrade -r /code/requirements.txt
RUN pip install --upgrade llama-cpp-python

#####
# used to build the base image running on NVIDIA GPU
FROM nvidia/cuda:12.3.2-devel-ubuntu22.04 as base-nvidia

ARG PYTHON_VERSION

WORKDIR /code
COPY ./requirements.txt /code/requirements.txt

RUN apt update && apt install python${PYTHON_VERSION} python3-pip ccache git libgl1 -y
RUN CMAKE_ARGS="-DLLAMA_CUBLAS=on" FORCE_CMAKE=1 pip3 install llama-cpp-python
RUN pip install --no-cache-dir --upgrade -r /code/requirements.txt

#####
# uses CPU or NVIDIA image and serves the FastAPI app
FROM base-${CPU_OR_NVIDIA}
CMD ["uvicorn", "rest_server:app", "--app-dir=/code/app", "--host=0.0.0.0", "--port=80",
"--reload", "--reload-include", "*.txt"]
```

Code 22. The Dockerfile used for building the server's image.

This is the actual Dockerfile used to build the server image. Let's go through it step by step.

First, the arguments "CPU_OR_NVIDIA" and "PYTHON_VERSION" dictate which steps are taken in the build process.

If the value of "CPU_OR_NVIDIA" is set to "nvidia", the instructions starting with "FROM nvidia/cuda:12.3.2-devel-ubuntu22.04 as base-nvidia" will be executed. If it's set to "cpu", then the instructions starting with "FROM python:\${PYTHON_VERSION} as base-cpu" are executed.

"PYTHON_VERSION" sets the python version used by either a CPU or NVIDIA based server image. Both versions copy the requirements inside the container, and install the dependencies, which varies again depending on CPU and GPU based images.

Finally, the main command that serves the LangServe app is declared in the last build step (starting with "FROM base-\${CPU_OR_NVIDIA}").

8.3 Orchestration of the containers with docker-compose.yaml files

Every component of the application live in a Docker container. This is managed by a centralized set of docker-compose.yaml files and Dockerfiles.

As it is possible to use multiple docker-compose.yaml files, I took advantage of this to support different use-cases. Here's a list of the files in question :

“docker-compose.dev.yaml”: declare the “server”, “ollama” and “vectorstore” containers, which used for both development and production purposes. The frontend projects run without docker.

“docker-compose.prod.yaml”: extends “docker-compose.dev.yaml” for the “server”, “ollama” and “vectorstore” containers, and declares the two frontend containers, which are built using a Dockerfile in the frontend directory.

“docker-compose.llm-nvidia.yaml”: extends only the “server” container, and passes an NVIDIA GPU to it. As my GPU isn't powerful enough to be shared between all the containers that would make use of it, this file was created. Here's what the content of the file looks like:

```
services:
  ollama:
    deploy:
      resources:
        reservations:
          devices:
            - capabilities: ["gpu"]
              driver: nvidia
              count: 1
```

Code 23. Extending the ollama service and passes the GPU to it.

“docker-compose.nvidia.yaml”: extends the server, ollama and vectorstore containers from “docker-compose.dev.yaml”, and passes an NVIDIA GPU to them. This is made for production purposes, where a more powerful GPU would be available.

8.4 User story – Compensate as much of the carbon footprint as possible

After a bit of research, we decided to use a Swiss web hosting company called Infomaniak. They are known for their actions to reduce their footprint since 2007, and currently offset all of their greenhouse gas emission by 200 % (Infomaniak, no date a).

They also have been collaborating with the Geneva's local energy company SIG in an innovative project to heat homes in the city with the heat produced by their datacenters (Grandjean, no date).

Additionally, the energy used to power their datacenters is entirely local and renewable (Infomaniak, no date a).

9 User story – Guarantee continuous operation, even when third party services used are down

9.1 Circumvent the unavailability of Hugging Face servers

In the event that either Hugging Face or Ollama servers couldn't be reached, it's important that the application still runs normally.

Here's the Python class responsible to manage embeddings:

```
# import and logger instantiation are hidden for brevity

class EmbeddingsManager:
    def __init__(self, embeddings_model):
        self.embeddings_model = embeddings_model

        self.embeddings_model_dir_name = self.embeddings_model.replace("/", "_")
        self.cache_dir = f"/root/.cache/{self.embeddings_model_dir_name}"

    def initialize(self) -> Embeddings:
        try:
            logger.info(f"EMBEDDINGS - Downloading/verifying presence of model
'{self.embeddings_model}' on directory "
                f"'{self.cache_dir}'")
            huggingface_hub.snapshot_download(
                repo_id=self.embeddings_model,
                local_dir=self.cache_dir
            )
            logger.info(f"EMBEDDINGS - Model '{self.embeddings_model}' was found !")
            embeddings = HuggingFaceInstructEmbeddings(model_name=self.cache_dir)
            return embeddings
        except (urllib3.exceptions.MaxRetryError, requests.exceptions.ConnectionError) as
warning:
            logger.warning(f"EMBEDDINGS - Couldn't download model {self.embeddings_model}
from HuggingFacs - {warning}")
            except EnvironmentError as error:
                logger.critical(
                    f"EMBEDDINGS - Model {self.embeddings_model} isn't cached locally, and
couldn't be downloaded - {error}")
                )
            exit(1)
```

Code 24. Python code that manages the embedding models.

The server first checks whether the selected embedding model is cached locally.

In case it is, and it's the latest version or if Hugging Face servers can't be reached to check for a new version, it uses the local file and logs a non-critical warning.

In case no files are available, it tries to download them from Hugging Face, and finally if it can't reach the servers, a critical error is logged, and the server shuts down.

9.2 Circumvent the unavailability of Ollama servers

Managing connections problems with Ollama servers is quite similar to the equivalent problem with Hugging Face's for the embeddings. Here's how it works.

```
# import and logger instantiation are hidden for brevity

class OllamaManager:
    def __init__(self, ollama_host: str, ollama_model: str):
        self.ollama_host = ollama_host
        self.ollama_model = ollama_model
        self.ollama_client = ollama.Client(host=ollama_host)

    def initialize(self) -> ChatOllama:
        logger.info(
            f"OLLAMA - creating new ollama.Client using host '{self.ollama_host}' and
model '{self.ollama_model}'"
        )
        chat_ollama = ChatOllama(base_url=self.ollama_host, model=self.ollama_model)

        # first checks if Ollama model is available locally, if it's not, tries pulling
it
        # if pulling doesn't work, logs an error and stops the server
        try:
            logger.info(f"OLLAMA - Verifying presence of model '{self.ollama_model}'")
            self.ollama_client.show(self.ollama_model)
            logger.info(f"OLLAMA - Model '{self.ollama_model}' was found !")
            return chat_ollama
        except (httpx.ConnectTimeout and httpx.ConnectError):
            logger.critical(f"OLLAMA - Couldn't connect to remote host
'{self.ollama_host}'")
            exit(1)
        except ollama.ResponseError:
            try:
                logger.info(f"OLLAMA - Trying to pull model '{self.ollama_model}'")
                self.ollama_client.pull(self.ollama_model)
                return chat_ollama
            except (ollama.ResponseError, httpx.ReadTimeout) as error:
                logger.critical(f"OLLAMA - Couldn't pull model '{self.ollama_model}' :
{error}")
                exit(1)
```

Code 25. Python code that manages the Ollama client, which itself manages the LLM.

The server will first try to connect to the Ollama host. If the connection isn't successful, a critical error is logged and the server shuts down. Otherwise, it keeps going.

The server checks if the requested model is already present locally, for example, "mistral". If it is, it logs that it found it and returns a ChatOllama python object to be used by LangChain.

If it's not cached, the server tries pulling it, and logs what's happening. In the case that it's impossible to do so, a critical error is logged and the server shuts down.

If it manages to pull the model, the function returns a ChatOllama object ready to be used.

10 Discussion

The produced software was finished in a timely manner, and accepted by the commissioning company as being successfully completed. Innovatim plan on commercializing this chatbot software and host it on their own servers, which demonstrates the general necessity and usability of the product.

Nevertheless, I think it might be interesting to see how each initial goal was accomplished, if at all :

The chatbot will use RAG techniques to provide document-based responses: the key part of the project, the project indeed uses RAG techniques, in conjunction with generative conversational AI. With the right prompt engineering, the chatbot only answers using information found in the documents given to it. The target was met with success.

The chatbot will need to cite the source it uses for its answers: the chatbot doesn't mention the exact name of the file used, as this information is not given to it directly. However, it is still possible to display this information on the frontend by passing the source file name as metadata in the chatbot's response. As the scope of the thesis grew, this feature wasn't prioritized and ended up not being implemented in the current version.

The chatbot will be able to answer in English, French, German and Italian: as explained in the corresponding user story, the model used throughout the project is only really proficient in English. Other models from Mistral AI were considered, as they support more than just English, but couldn't be run on personal hardware. This goal still remains to be achieved.

Support PDFs, Word documents, Excel sheets and plain text files: using the Unstructured library, every required document type is now supported by the software.

Good security, to guarantee that only authorized users will be able to access private information: Innovatim wanted to have as transparent as possible user experience, which made security a hard topic to approach. Still, I feel that the provided token-based solution for the chatbot frontend and the credential-based authentication for the administrative backend is sufficient, and would consider that the target was hit.

Deploy the chatbot on a Swiss infrastructure, with no communication with outside APIs: Again, this has yet to be done, but shouldn't take too long once I get access to a production environment. Still, the objective has yet to be met.

Have an intuitive user interface: this aspect was probably the most tested in the project, and was what I felt the most comfortable working on. A lot of small details have been considered to make

the experience as smooth as possible, and the feedback I received from showing the product to Innovatim pushes me to say that the objective was met.

Possibility to easily change LLM model used: updating a “.env” file to then restart a docker container is a very simple way to manage something as complex as LLMs, the objective was achieved.

Guarantee continuous operation, even when third party services used are down: many precautions were taken to keep the risk low, consequently, the desired outcome was attained.

Compensate as much of the carbon footprint as possible: thanks to Infomaniak’s ecological involvement, the greenhouse gas produced to run this project are offset by 200 %. For this reason, I’d say that the goal was achieved.

Overall, I’m personally pleased with the work I produced, and learned a lot on the subject of generative AI, deployment, frontend and backend development. Work still remains to be done for the project to be viable commercially and for all the objectives to be achieved. I hope that I will be able to continue working on this project.

Going in depth with the technologies I used for the project was a pleasure, and I will definitely continue working with as Astro, Web Components and Docker for other projects in the future.

References

AIM Research (2023) 'Leaders Opinion: The Problems with LLM Benchmarks', *AIM Research*, 14 September. Available at: <https://aimresearch.co/leaders-opinion/leaders-opinion-the-problems-with-llm-benchmarks> (Accessed: 29 April 2024).

Amazon Web Services, Inc. (no date a) *What is DevOps? - DevOps Models Explained - Amazon Web Services (AWS)*, Amazon Web Services, Inc. Available at: <https://aws.amazon.com/devops/what-is-devops/> (Accessed: 27 April 2024).

Amazon Web Services, Inc. (no date b) *What is Prompt Engineering? - AI Prompt Engineering Explained - AWS*, Amazon Web Services, Inc. Available at: <https://aws.amazon.com/what-is/prompt-engineering/> (Accessed: 19 May 2024).

Ammanath, B. (2024) *How to manage AI's energy demand — today and in the future*, *World Economic Forum*. Available at: <https://www.weforum.org/agenda/2024/04/how-to-manage-ais-energy-demand-today-tomorrow-and-in-the-future/> (Accessed: 19 May 2024).

Bradshaw, D.J. (2024) 'davidjbradshaw/iframe-resizer'. Available at: <https://github.com/davidjbradshaw/iframe-resizer> (Accessed: 20 May 2024).

Grandjean, O. (no date) *Un datacenter vert pour chauffer des logements*, *Vive la Vie*. Available at: <https://vive-la-vie.magnum3.ch/article/un-datacenter-vert-pour-chauffer--des-logements/186> (Accessed: 31 January 2024).

Infomaniak (no date a) *Ecology – Find out why we pollute*. Available at: <https://www.infomaniak.com/en/ecology> (Accessed: 31 January 2024).

Infomaniak (no date b) *Public Cloud - cloud infrastructure for your company*. Available at: <https://www.infomaniak.com/en/hosting/public-cloud> (Accessed: 27 April 2024).

Innovatim (no date) *Innovatim - Agence d'intelligence artificielle*, *Innovatim*. Available at: <https://innovatim.ch/> (Accessed: 27 April 2024).


Jakob Nielsen (2024) *10 Usability Heuristics for User Interface Design*, *Nielsen Norman Group*. Available at: <https://www.nngroup.com/articles/ten-usability-heuristics/> (Accessed: 20 May 2024).

Jan Philipp Harries (2023) *jphme/em_german_leo_mistral* · *Hugging Face*. Available at: https://huggingface.co/jphme/em_german_leo_mistral (Accessed: 20 May 2024).

LangChain (2024) *langserve/examples/conversational_retrieval_chain/server.py at main · langchain-ai/langserve*, *GitHub*. Available at: https://github.com/langchain-ai/langserve/blob/main/examples/conversational_retrieval_chain/server.py (Accessed: 20 May 2024).

LangChain (no date a) *Providers* |  *LangChain*. Available at: <https://python.langchain.com/docs/integrations/platforms/> (Accessed: 28 April 2024).

LangChain (no date b) *Text embedding models* |  *LangChain*. Available at: https://python.langchain.com/docs/modules/data_connection/text_embedding/ (Accessed: 28 April 2024).

LangChain (no date c) *Unstructured* |  *LangChain*. Available at: <https://python.langchain.com/v0.1/docs/integrations/providers/unstructured/> (Accessed: 21 May 2024).

MDN (2024a) *Cross-Origin Resource Sharing (CORS) - HTTP* | MDN. Available at: <https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS> (Accessed: 20 May 2024).

MDN (2024b) *Web Components - Web APIs* | MDN. Available at: https://developer.mozilla.org/en-US/docs/Web/API/Web_components (Accessed: 28 April 2024).

Michele Montebovi (2024) *DeepMount00/Mistral-Ita-7b · Hugging Face*. Available at: <https://huggingface.co/DeepMount00/Mistral-Ita-7b> (Accessed: 20 May 2024).

Ollama (2023) *Ollama is now available as an official Docker image · Ollama Blog*. Available at: <https://ollama.com/blog/ollama-is-now-available-as-an-official-docker-image> (Accessed: 28 April 2024).

OpenLLM France (2023) *OpenLLM-France/Claire-Mistral-7B-0.1 · Hugging Face*. Available at: <https://huggingface.co/OpenLLM-France/Claire-Mistral-7B-0.1> (Accessed: 20 May 2024).

OpenStack (no date) *Open Source Cloud Computing Platform Software, OpenStack*. Available at: <https://www.openstack.org/software/> (Accessed: 27 April 2024).

Qdrant (2024) *Vector Database Benchmarks - Qdrant*. Available at: <https://qdrant.tech/benchmarks/> (Accessed: 20 May 2024).

Stack Overflow (no date) *Stack Overflow Developer Survey 2023, Stack Overflow*. Available at: https://survey.stackoverflow.co/2023/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2023 (Accessed: 28 April 2024).

Zilliz (no date) *VectorDBBench: An Open-Source VectorDB Benchmark Tool*. Available at: <https://zilliz.com/vector-database-benchmark-tool> (Accessed: 29 April 2024).