Project Q2
Generative AI
Automatic GUI Translation

SDP project a.a. 24/25 - Quer

s342937 - Oggero Paolo

s341882 - Pagano Mariano Michele

s323141 - Pochettino Alberto

September 19, 2025

Contents

Chapter 1

Introduction

This project aims to develop a module that integrates with web-based applications to provide real-time, automatic translation of the Graphical User Interface (GUI). The module leverages locally installed generative AI models to dynamically translate visible text elements into a target language selected by the user.

Chapter 2

Problem Definition

Web applications often need to support multiple languages to reach a wider user base. Traditional localization methods involve manually translating UI strings and managing language files, which can be time-consuming and costly. This project explores the use of large language models (LLMs) for on-the-fly GUI translation.

Key challenges include:

- **Integration:** Linking the translation module with existing web frameworks (React/Angular) to intercept and translate text elements without breaking functionality.
- **Context Preservation:** Ensuring translations are semantically correct (e.g., button labels vs. descriptive text).
- **Performance:** Achieving real-time translation speed using locally hosted LLMs.
- **Dynamic Content:** Handling text dynamically generated or updated within the app.
- **Accuracy:** Evaluating translation quality and context-awareness.
- **Local Deployment:** Managing the setup and computational requirements of running LLMs locally.

Chapter 3

Methodology

The project methodology is structured around the following components:

- Designing the translation module architecture.
- Identifying integration points within React-based front-ends.
- Intercepting and replacing text nodes dynamically.
- Developing a back-end pipeline for model loading, inference, and caching.
- Evaluating translation accuracy and speed.

Chapter 4

System Requirements and Dependencies

4.1 Introduction

The implementation of the real-time GUI translation module requires specific software dependencies to function properly. This chapter outlines the essential components that must be installed and configured to support the translation functionality.

4.2 Ollama Platform

4.2.1 Overview

Ollama is an open-source platform designed to simplify the deployment and execution of large language models on local systems. It provides a unified interface for downloading, managing, and serving various AI models through a REST API, eliminating the complexity typically associated with running large language models locally.

The platform abstracts the underlying model management complexities, allowing developers to focus on application logic rather than model deployment concerns. Ollama handles memory management, model loading, and API serving automatically, making it an ideal choice for applications requiring local AI inference capabilities.

4.2.2 Installation

For this project, Ollama was installed on the local development system to provide the translation back-end services. The installation process involves downloading the appropriate installer for the target operating system and following the standard installation procedures provided by the Ollama platform.

4.3 Language Model Selection

4.3.1 Model Requirements

The translation module requires a large language model capable of multilingual translation tasks. For this implementation, we selected and installed models from different families, which provide robust translation capabilities across multiple language pairs.

4.4 System Integration

The combination of Ollama and the selected language models provides the necessary infrastructure for the module to function. Ollama serves as the intermediary between the web application and the AI models, handling model inference requests and returning translation results through its REST API interface.

This local setup ensures that translation operations can be performed without relying on external cloud services, providing better privacy, reduced latency, and independence from internet connectivity for the core translation functionality.

Chapter 5

back-end Implementation

This chapter describes in detail the back-end architecture and implementation of the automatic GUI translation module. The back-end is responsible for orchestrating the Large Language Model (LLM), managing translation requests, providing caching, and serving data efficiently to the front-end.

5.1 back-end architecture

The back-end handles interactions with a locally hosted Large Language Model (LLM) managed through the `Ollama` framework. To enable interaction from the testing web application, the `Ollama` framework is wrapped within a `FastAPI` server, which exposes RESTful endpoints for translation requests.

5.1.1 back-end structure

The back-end is separated in four main files:

- `app.py`: Defines a REST API back-end consisting of two endpoints, configures a CORS middleware, and mounts a static file directory. The `app.mount("/static", StaticFiles(...))` call serves files from the `../front-end` directory at the `/static` URL path, allowing a webpage to get the JavaScript translator module remotely, like described in **??**.

- `app_utils.py`: Provides essential utilities for constructing translation prompts and managing token limits for Large Language Model (LLM) requests. It integrates with the tiktoken library to estimate token usage and ensure that prompts remain within model constraints. The functions here are used internally by the FastAPI back-end (app.py) to build consistent prompts and to split large batches of input text into smaller, model-compliant requests.

- `app_conf.py`: Defines configuration parameters for the translation system, including the model to be used in case the environment variable is not set, token limits for various LLM models considered during development, and a map of supported languages containing the ISO 639-1 language codes and their corresponding English full name. By centralizing these values, the rest of the application remains flexible and easy to maintain.

- `start.py`: Launches the system composed of Ollama, the selected LLM model, and the FastAPI serve,r giving enough time to each component to initialize before passing to the next step. A background theread is also started to log inthe console LLM model output.

5.1.2 Endpoints

Two endpoints are exposed by the `FastAPI` server:

- `/translate-one`: Accepts a single string and target language, validates input, generates a prompt, queries the LLM, caches the result, and returns the translation.

- `/translate-many`: Accepts a list of strings and a target language, splits the input into token-limited prompts, and streams generated translations back to the client.

5.1.3 Endpoint: /translate-one

This endpoint handles single-word or phrase translations:

```
GET /translate-one?text={encoded_text}&lang={target_language}
```

This endpoint handles individual text translation requests and returns JSON responses in the following format:

```
{
  "translation": "translated_text_content"
}
```

1. Validates input parameters `text` and `lang`.

2. Checks the cache for a previous translation.
3. Strips leading and trailing punctuation but preserves them for reinsertion after translation.
4. Selects the appropriate prompt type ("word" vs. "phrase").
5. Calls the Ollama model:

   ```
   response = ollama.generate(model=app_conf.model_name, prompt=prompt)
   ```

6. Returns the translation as JSON:

   ```
   { "translation": "<translated_text>" }
   ```

5.1.4 Endpoint: /translate-many

This endpoint translates a list of texts in a single request:

```
POST /translate-many?lang={target_language}
Content-Type: application/json

{
  "texts": ["text1", "text2", "text3"]
}
```

This endpoint supports streaming responses, returning translations in the format:

```
index -> translated_text\n
```

1. Accepts a JSON body with `texts: List[str]` and a query parameter `lang`.
2. Checks the cache for a previous translation.
3. Splits the list into multiple prompts, respecting the LLM token limit (via `app_utils.split_prompts_by_token_limit`).
4. Returns a `StreamingResponse` that yields translations progressively.

5.1.5 Streaming Response

The function `stream_AI_response(prompts)` streams partial translations:

- Uses `ollama.generate(..., stream=True)` to receive incremental output.
- Buffers text until newline characters appear, then yields complete lines.
- Enables clients to render results incrementally instead of waiting for the entire response.

5.2 Optimizations

5.2.1 Prompt Templates

Three prompt templates are defined to get a better model behavior during translation:

- `word` – designed for translating a single word.
- `phrase` – designed for translating a single phrase or sentence.
- `phrase_list` – designed for batch translation of multiple texts, returning output in a strict format (`index -> translated_text`).

Each template tries to get the model to follow three main points:

- Only the translation is returned (no explanations or commentary).
- Numerals and symbols are preserved as-is.
- Leading and trailing punctuation is not modified.

5.2.2 Caching Layer

To avoid redundant model calls, an in-memory cache is implemented:

```
cache = {}
cache_lock = threading.Lock()
```

A thread lock ensures safe concurrent access.

5.2.3 Supported Languages

The module also defines a mapping of ISO 639-1 language codes to human-readable language names. This dictionary supports over 150 languages, allowing the system to dynamically handle translation requests across diverse linguistic contexts. The use of standardized codes ensures compatibility with external systems and datasets.

5.3 Workflow Summary

The complete translation workflow is as follows:

1. The client sends one or more texts along with the target language.
2. The back-end validates the input against supported languages.
3. Input text is tokenized and, if necessary, split into chunks respecting token limits.
4. Structured prompts are generated using predefined templates.
5. The prompts are sent to the selected LLM via the Ollama API.
6. Responses are cached and returned to the client:
   - As a single JSON object for `/translate-one`.
   - As a streaming text response for `/translate-many`.

5.4 Design Rationale and Performance Considerations

The back-end design emphasizes:

- **Separation of concerns:** The LLM runs in a separate process from the FastAPI server to isolate model initialization, memory consumption, and crash handling.
- **Scalability:** Token-limited chunking and async endpoints allow the back-end to handle multiple clients efficiently.
- **Observability:** Logging and structured monitoring provide visibility into model inference and API request handling.

- **Maintainability:** Centralized configuration and utility functions simplify model updates and prompt template modifications.

Chapter 6

Real-Time GUI Translation Module Implementation

6.1 Introduction

This chapter presents the design and implementation of a JavaScript-based translation module that provides real-time, automatic translation of graphical user interface elements in web applications. The module, implemented as `translator.js`, integrates with modern web frameworks to deliver dynamic language translation capabilities using locally installed generative AI models.

The primary objective of this implementation is to enhance the accessibility and usability of web applications for a global audience by eliminating the need for developers to manually create translations for every supported language. The module operates by dynamically identifying and translating all visible text elements within an application's interface, regardless of the original source language.

6.2 System Architecture

6.2.1 Core Components

The translation module comprises several interconnected components that work together to provide seamless real-time translation:

1. **addTextNodes()**: Adds a text node or all its text nodes children if it is not a text node to the active text nodes map. This function is called on module mounting to detect every text node present in the page.

2. **DOM Observer System**: Utilizes the MutationObserver API to monitor changes in the Document Object Model (DOM) and identify new text content requiring translation, passing them to `addTextNodes()`.

3. **Text Node Management**: Implements a dual-mapping system using JavaScript Map objects to efficiently track and manage text nodes and their corresponding content.

4. **Translation Service Interface**: When translation is requested the module provides communication with the back-end translation server through RESTful API endpoints.

5. **User Interface Controller**: Manages the language selection interface and user interactions.

6. **State Management System**: Coordinates translation states and prevents race conditions during DOM modifications.

6.2.2 Data Structures

The module employs two primary data structures for efficient text node management:

- `textNodeToStringAll`: A Map object that maintains a one-to-one relationship between DOM text nodes and their original string content.
- `stringToTextNodes`: A Map object that groups text nodes by their content, enabling batch translation of identical strings.

Additionally, a WeakSet object (`temporarilyIgnoredNodes`) is utilized to temporarily exclude nodes from observation during translation operations, preventing infinite loops in the mutation observer.

6.3 Implementation Details

6.3.1 Module Initialization

The module is initialized through the `startTranslationObserver` function, which accepts three parameters:

- `serverUrl`: The endpoint URL for the translation service (default: `http://127.0.0.1:8000`)

- `selectedPosition`: The position of the language selector UI element
- `languages`: An array of ISO 639-1 language codes to support

Upon initialization, the module performs the following operations:

1. Validates input parameters and language code compliance
2. Establishes the initial DOM state and identifies existing text nodes
3. Injects the language selection user interface
4. Configures the MutationObserver for real-time DOM monitoring

6.3.2 DOM Monitoring and Text Node Detection

The module implements a comprehensive DOM monitoring system using the MutationObserver API. The observer is configured to monitor:

- `childList`: Detection of added or removed DOM elements
- `subtree`: Recursive monitoring of all descendant elements
- `characterData`: Changes to text content within existing nodes
- `attributes`: Modifications to element attributes (filtered for translation-related attributes)

6.3.3 addTextNodes()

This function adds text nodes starting from a passed parent element, and it:

1. Recursively examines all child nodes of a given element
2. Filters out script and style elements to prevent interference
3. Respects the `no-translate` class designation for elements that should remain untranslated
4. Validates text content to ensure non-empty, meaningful strings

6.3.4 Translation Strategies

The module implements two distinct translation strategies to optimize performance and user experience:

Single Node Translation

The `translateTextNodeSafely` function handles individual text node translation through the following process:

1. Temporarily excludes the target node from observation
2. Increments the active translation counter
3. Sends an HTTP GET request to the `/translate-one` endpoint
4. Updates the node content with the translated text
5. Manages the translation state and UI indicators

Batch Translation

The `loadPageTranslated_translate_many` function implements batch translation for improved efficiency:

1. Collects all unique text strings requiring translation
2. Sends a POST request to the `/translate-many` endpoint
3. Processes streaming responses to update multiple nodes simultaneously
4. Utilizes the string-to-nodes mapping for efficient bulk updates

6.3.5 User Interface Implementation

The language selection interface is dynamically injected into the target web page using the `injectSelectHTML` function. The interface features:

- A compact, hoverable language indicator showing the current language code
- An expandable dropdown with search functionality for language selection
- Visual feedback during translation operations through loading indicators
- Positioning flexibility with four predefined locations

The interface implements event-driven interactions that:

- Filter available languages based on user input
- Trigger translation operations upon language selection
- Provide visual feedback during processing states

6.4 State Management and Concurrency

6.4.1 Translation State Coordination

The module implements a multi-layered state management system to handle concurrent operations:

1. **Global Translation State**: The `isTranslating` flag is used to update the ui element, allowing the user to visualize when a translation is happening
2. **Active Translation Counter**: Tracks individual node translation operations to coordinate UI updates
3. **DOM Stability Monitoring**: The `isDomChanging` flag ensures translations occur only when the DOM is stable

6.4.2 Error Handling and Resilience

The current implementation includes basic error handling for network failures and malformed responses. Translation errors are logged to the console while maintaining application stability.

6.5 Performance Considerations

6.5.1 Optimization Strategies

The module implements several optimization strategies to minimize performance impact:

- **Deduplication**: Identical text strings are translated once and applied to multiple nodes
- **Selective Processing**: Only visible, meaningful text content is processed for translation

- **Asynchronous Operations**: Translation requests are handled asynchronously to prevent UI blocking
- **State-based Execution**: Translation only occurs when the target language differs from the original

6.6 Discussion

The translator.js module demonstrates a practical approach to real-time GUI translation that balances functionality with performance considerations. The implementation successfully addresses the core requirements of dynamic text detection, efficient translation processing, and seamless user experience integration.

The module's architecture provides a solid foundation for further development and could serve as a reference implementation for similar translation systems. The use of modern web APIs and thoughtful state management creates a robust solution that can be adapted to various web application frameworks and deployment scenarios.

While current limitations exist, particularly in areas of error handling and offline functionality, the module represents a significant step toward making web applications more accessible to global audiences through automated, AI-powered translation capabilities.

Chapter 7

Module Integration

7.1 Introduction

This chapter describes the integration methods for incorporating the translation module into web applications. The module supports two distinct integration approaches, each designed to accommodate different development workflows and deployment scenarios.

7.2 Integration Methods

7.2.1 Remote Module Loading

The first integration method involves loading the translation module directly from a remote server. This approach is particularly useful for testing environments or when the module is served as a shared resource.

Implementation

The remote loading method utilizes ES6 module imports within an HTML script tag:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <title>Translation Observer Test</title>

  <!-- Load the translation observer as a module -->
  <script type="module">
```

```
    import startTranslationObserver from
           'http://127.0.0.1:8000/static/translator.js';
    startTranslationObserver("http://127.0.0.1:8000");
  </script>
</head>
```

Characteristics

This integration method offers several advantages:

- **Centralized Updates**: Module updates are automatically available to all consuming applications
- **Minimal Local Dependencies**: No need to include the module in the application bundle
- **Development Flexibility**: Easy switching between different module versions
- **Cross-Origin Resource Sharing**: Requires proper CORS configuration on the serving server

7.2.2 Local Module Integration

The second integration method involves importing the module as a local dependency within the application's build system. This approach is more suitable for production environments and applications with specific dependency management requirements.

Implementation

Local integration utilizes standard ES6 import syntax within the application's JavaScript files:

```
import startTranslationObserver from '../../module/front-end/translator.js'

startTranslationObserver("http://127.0.0.1:8000", "Bottom-Left", ["en", "fr"]);
```

7.3 Configuration Options

7.3.1 Parameter Specification

The module accepts three configuration parameters:

1. **serverUrl** (string): The back-end translation service endpoint
2. **selectedPosition** (string): UI positioning - accepts ”Bottom-Left”, ”Top-Left”, ”Top-Right”, or ”Bottom-Right”
3. **languages** (array): ISO 639-1 language codes to restrict available translation options

7.3.2 Default Behavior

When parameters are omitted, the module applies default configurations:

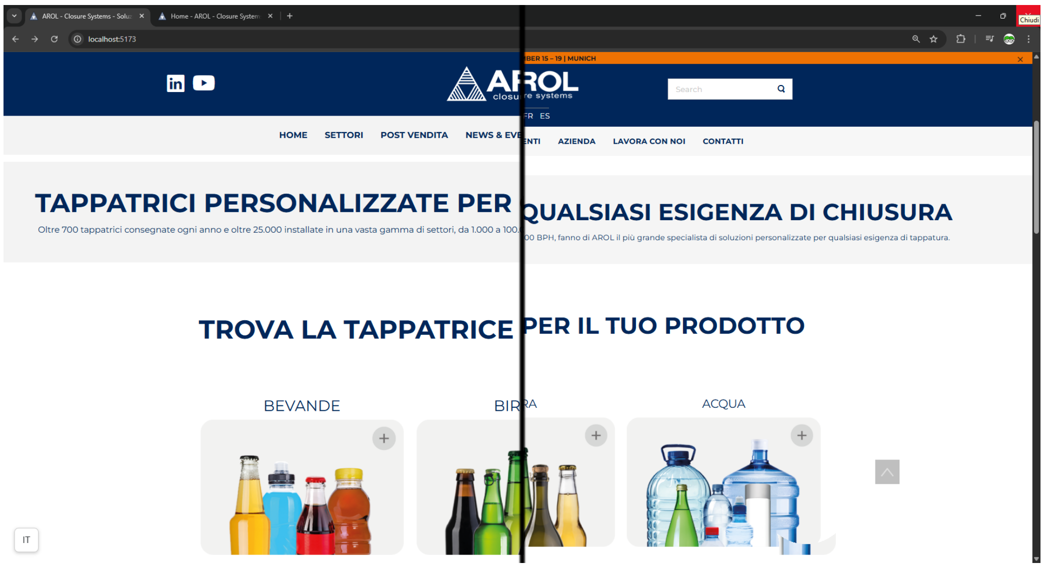
- Server URL defaults to `http://127.0.0.1:8000`
- Position defaults to ”Bottom-Right”
- Language selection includes all supported ISO 639-1 language codes

Chapter 8

Testing Web Application

To validate the translation module, we built a minimal web application using React using as a reference the `Arol Group` webpage. The goal of this test app was to simulate a realistic GUI scenarios such as navigation between pages, static text rendering, and integration of Bootstrap components. Since the main scope of the project was the translation module, only three pages of the webpage were implemented: `Homepage`, `Azienda`, and `Bevande`.

8.1 Example page


localhost:5173
AROL closure systems
HOME SETTORI POST VENDITA NEWS & EVENTI AZIENDA LAVORA CON NOI CONTATTI
TAPPATRICI PERSONALIZZATE PER QUALSIASI ESIGENZA DI CHIUSURA
TROVA LA TAPPATRICE PER IL TUO PRODOTTO
BEVANDE
BIRRA
ACQUA
IT

OUR VERSION ORIGINAL VERSION

Chapter 9

Evaluation and Analysys

9.1 Model Loading Strategies

During development, we experimented with two different approaches for executing the language models.

9.1.1 Hugging Face Transformers

Using the `transformers` library, we loaded pretrained language models directly through `AutoModelForCausalLM` and `AutoTokenizer`. Using `.to("cuda")`, we made sure the model was running on the GPU and the output was retrieved using the appropriate decoding method.

This approach required a longer processing time to obtain the translations and a more complex implementation in terms of code.

9.1.2 Ollama Runtime

Alternatively, we integrated the `ollama` runtime, which abstracts the complexity of model loading and optimization. Ollama runs the model as a background process and handles low-level optimizations automatically. In our back-end implementation, Ollama was launched as a subprocess, and its output was streamed into the FastAPI service, enabling seamless interaction between the model and the REST API.

This approach was more efficient and lightweight, making it better suited for real-world deployment.

9.2 Library Comparison

To compare the two frameworks, we executed the same translation task on 1000 sentences using the `tinyllama-1.1b-chat-v0.6-fp16` model. The evaluation considered both execution time and translation quality, measured using the BLEU score.

Framework	Language	BLEU Score	Time Taken
Ollama	Italian	3.47	12m 35s
	French	6.47	10m 59s
	German	3.08	10m 42s
Hugging Face	Italian	5.41	15m 49s
	French	10.90	15m 14s
	German	5.30	15m 24s

Table 9.1: Comparison between Hugging Face and Ollama using the `tinyllama-1.1b-chat-v0.6` model.

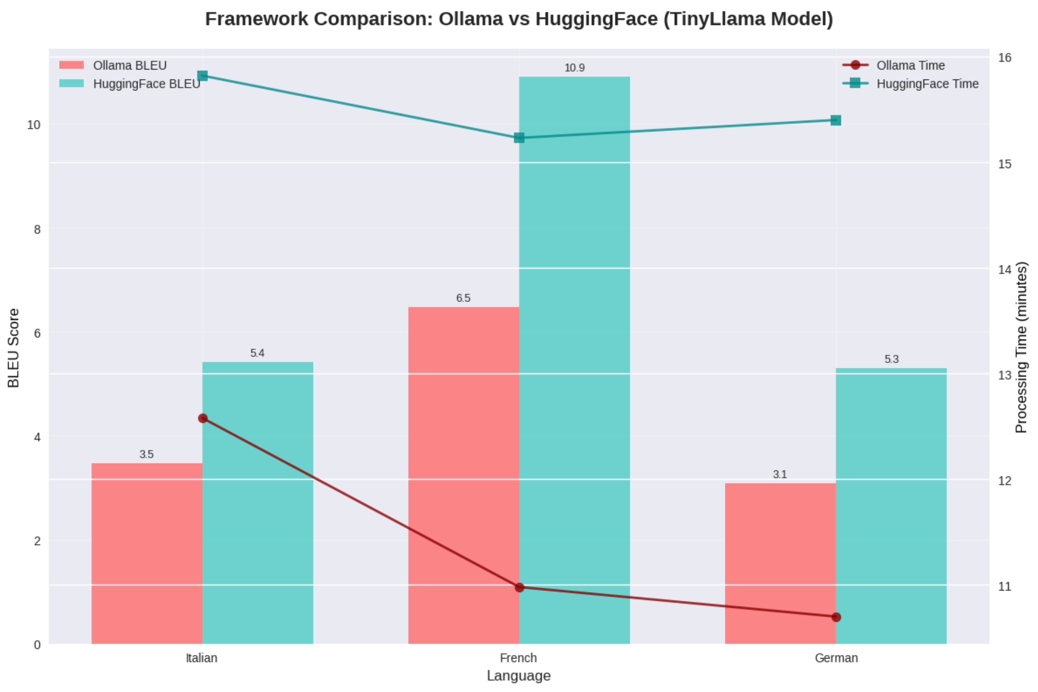


Figure 9.1: Comparison of BLEU scores and processing time for Ollama and Hugging Face across three languages. The bar plot shows BLEU scores, while the line plot represents execution time.

9.2.1 Discussion

The figure 9.1 clearly illustrates that Hugging Face achieved slightly higher BLEU scores across all three languages, while Ollama processed the data faster. This highlights the classic trade-off between translation quality and efficiency, with Hugging Face prioritizing accuracy and Ollama offering lower latency.

Ollama was therefore used due to the more efficient performance and an easier set up of the models, making it preferable for scenarios where latency is critical.

9.3 Evaluation of Model Performance

We evaluated multiple models across three target languages: Italian, French, and German. The evaluation considered BLEU scores to measure translation quality, and total processing time to assess efficiency.

The plots in Figures 9.2 and 9.3 summarize the performance of all tested models.

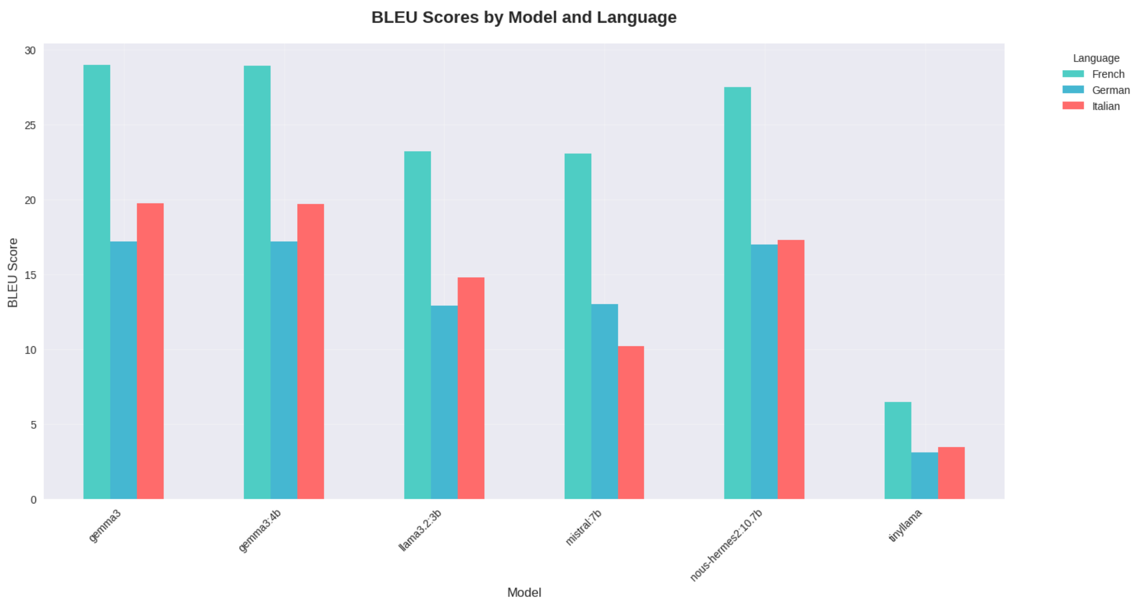


Figure 9.2: BLEU scores by model and target language. Higher scores indicate better translation quality.

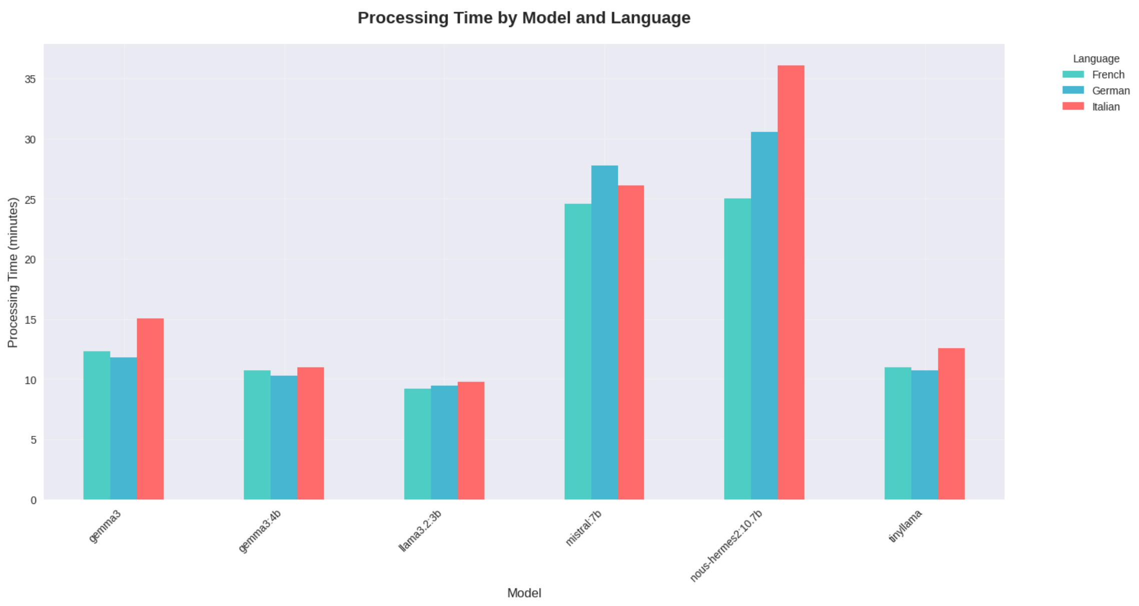


Figure 9.3: Processing time by model and target language. Lower values indicate better efficiency.

9.3.1 Discussion

All tested models are relatively small due to the hardware limitations of our systems. Since latency was one of the key concerns only models that could fit entirely in the GPU memory were considered. All models were tested using an i9-14900HX and an NVIDIA RTX4070 with 8GB of dedicated memory. Every model was tested using different language pairs, and we discovered that each LLM has a different BLEU score based on the languages involved. As it can be seen in Figure 9.2, it is clear that the `gemma3` family (`gemma3` and `gemma3:4b`) achieved the highest BLEU scores, particularly for French, where the scores approached 29. The `nous-hermes2:10.7b` model also performed well, with slightly lower BLEU scores but much higher resource demands and longer runtimes.

Smaller models such as `tinyllama` exhibited very poor BLEU performance while still requiring significant processing time, demonstrating that they are not viable for high-quality translation tasks.

Figure 9.3 highlights the trade-off between performance and runtime. While the `gemma3` models achieved strong results, they also required moderate processing time (10–15 minutes). The `nous-hermes2:10.7b` model, despite its high quality, was the slowest, with runtimes exceeding 30 minutes.

Overall, the `gemma3` family provided the best balance between translation quality and efficiency for the available hardware, making it ideal for deployment in our case. It is also worth considering that the models considered are generic-purpose models and, therefore, the quality of the translations is lower than what could be achieved with a model trained with the unique purpose of translation.

Across all pages and methods, Italian-to-French translations consistently require 25-35% more time than Italian-to-English translations. This pattern is visible in both figures and likely reflects the increased token count and processing complexity for French output generation in the `gemma3:4b` model.

9.3.2 Other Libraries and Models

During the development of the back-end, we considered the possibility of experimenting with additional frameworks and model providers, such as `vLLM`, `ExLlama`, and optimized GPU runtimes (e.g., NVIDIA TensorRT or CUDA-specific inference engines). These libraries are designed to significantly improve inference speed and reduce memory consumption, offering state-of-the-art performance in large-scale deployments.

However, a practical limitation emerged: many of these back-ends and optimizations are available exclusively for Linux-based systems. Since the development and testing environment used in this project was based on Windows, it was not possible to evaluate them within the scope of this work. As a result, the comparison has been restricted to the two accessible back-ends, the Hugging Face `transformers` pipeline and the `Ollama` runtime.

Despite this limitation, the two selected approaches provide a representative evaluation of the challenges and trade-offs in integrating language models into a translation pipeline, while leaving open the possibility for future extensions and experiments on Linux-based systems.

9.4 Performance Evaluation on Real Web App Translation

To evaluate the real-world performance of the translation module, we measured the time required to translate three representative pages of the testing web application using the `gemma3:4b` model. The tests were conducted for two target languages: English (*it-en*) and French (*it-fr*).

For each page, we compared three translation approaches:

- **/translate-one**: Each text element is sent individually as a separate API request.
- **/translate-many**: All elements are grouped into a single batch request, reducing network overhead.
- **After caching**: The same translation is repeated after enabling the caching mechanism to measure the improvement for repeated requests.

The metrics include:

- **front-end Elapsed Time**: The total time perceived by the user from the moment the translation starts until the page is fully updated with translated text.
- **back-end Elapsed Time**: The aggregated server-side execution time, including the number of calls, average latency per call, and the total processing time.

9.4.1 Experimental Results

Tables 9.2, 9.3, and 9.4 summarize the translation times for the three test pages, each containing varying numbers of text elements (34, 51, and 75 elements, respectively).

Table 9.2: Translation times for the Homepage using `gemma3:4b`

Lang	Method	Front	Back Avg	Back Total	Calls
it-en	/translate-one	4.77 s	0.746 s	25.37 s	34
	/translate-many	8.25 s	8.12 s	8.12 s	1
	after caching	0.04 s	—	—	—
it-fr	/translate-one	6.26 s	0.978 s	33.26 s	34
	/translate-many	9.13 s	9.12 s	9.12 s	1
	after caching	0.03 s	—	—	—

Table 9.3: Translation times for the Azienda using `gemma3:4b`

Lang	Method	Front	Back Avg	Back Total	Calls
it-en	/translate-one	8.14 s	0.905 s	46.14 s	51
	/translate-many	15.64 s	15.54 s	15.51 s	1
	after caching	0.05 s	—	—	—
it-fr	/translate-one	10.30 s	1.131 s	57.67 s	51
	/translate-many	16.10 s	15.97 s	15.97 s	1
	after caching	0.06 s	—	—	—

Table 9.4: Translation times for the Bevande using `gemma3:4b`

Lang	Method	Front	Back Avg	Back Total	Calls
it-en	/translate-one	9.91 s	0.756 s	56.721 s	75
	/translate-many	20.62 s	20.494 s	20.494 s	1
	after caching	0.07 s	—	—	—
it-fr	/translate-one	13.26 s	0.999 s	74.992 s	75
	/translate-many	21.07 s	20.969 s	20.969 s	1
	after caching	0.07 s	—	—	—

9.4.2 Performance Analysis

front-end Performance Comparison

Figure 9.4 illustrates the front-end performance across all tested pages and translation methods. The visualization reveals several key patterns:

The `/translate-one` method consistently outperforms `/translate-many` in terms of front-end elapsed time across all pages and language pairs. For the Homepage, `/translate-one` achieves 4.77s (it-en) and 6.26s (it-fr) compared to 8.25s and 9.13s for `/translate-many`, respectively. This counterintuitive result suggests that parallel processing of individual requests provides a better user experience despite generating more network calls.

This evaluation has been made running the back-end server locally; therefore, the network traffic analysis and latency is to be evaluated with care. In a real environment, the server that host the LLM and implements the API call used by the front-end module would take much longer to respond to the APIs.

The performance gap between methods widens significantly with page complexity. On the most complex Bevande page, the difference reaches 10.71s for it-en translations (9.91s vs 20.62s) and 7.81s for it-fr translations (13.26s vs 21.07s).

Scaling Behavior

Figure 9.5 demonstrates how translation performance scales with the number of elements per page. The scaling analysis reveals non-linear performance degradation, with the `/translate-many` method showing more pronounced deterioration as page complexity increases.

The individual translation approach maintains relatively stable per-element efficiency, while the batch method's efficiency decreases substantially with larger payloads. This suggests potential bottlenecks in processing large batched requests that outweigh the theoretical advantages of reduced network overhead.

Caching Impact

The caching mechanism provides dramatic performance improvements, reducing translation times to under 0.1 seconds across all scenarios (0.03-0.07s range). This represents over 99% improvement compared to initial translation times, making repeated translations virtually instantaneous and the system highly suitable for production environments with recurring content.

9.4.3 Discussion

The experimental results reveal important trade-offs between different translation approaches. While intuition might suggest that batching requests in `/translate-many` should reduce overhead and improve performance, our findings demonstrate the opposite: individual API calls provide superior front-end performance.

This counterintuitive result can be explained by the parallel processing capabilities available when making multiple individual requests. The back-end processing times show that while `/translate-one` generates significantly more total processing time (e.g., 25.37s vs 8.12s for Homepage it-en), the parallel execution of these requests results in better front-end performance. In contrast, `/translate-many` processes all elements sequentially within a single request, creating a bottleneck that increases with payload size.

The `/translate-one` approach also offers progressive UI updates, allowing users to see translations appear incrementally rather than waiting for the entire batch to complete. This progressive feedback can significantly improve perceived responsiveness, particularly important for user experience in web applications.

The caching mechanism's effectiveness confirms the system's readiness for production deployment, where content repetition is common and near-instantaneous response times are achievable for previously translated material.

These experiments establish that:

- The `/translate-one` method provides superior front-end performance and user experience across all tested scenarios.
- Performance scales non-linearly with page complexity, with batch processing showing more dramatic degradation.
- Caching mechanisms are essential for production deployment, providing over 99% performance improvement for repeated content.

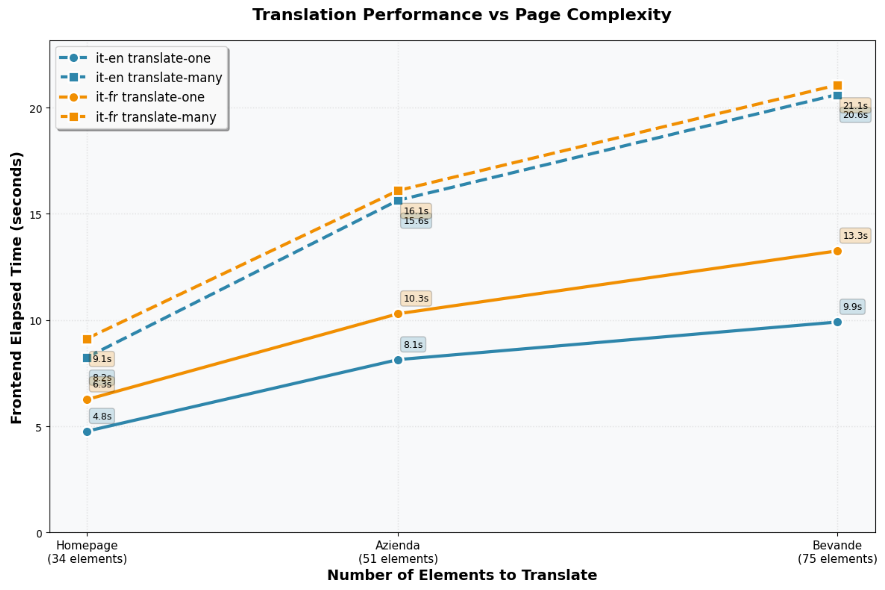


Figure 9.4: front-end performance comparison across pages and translation methods using `gemma3:4b`.

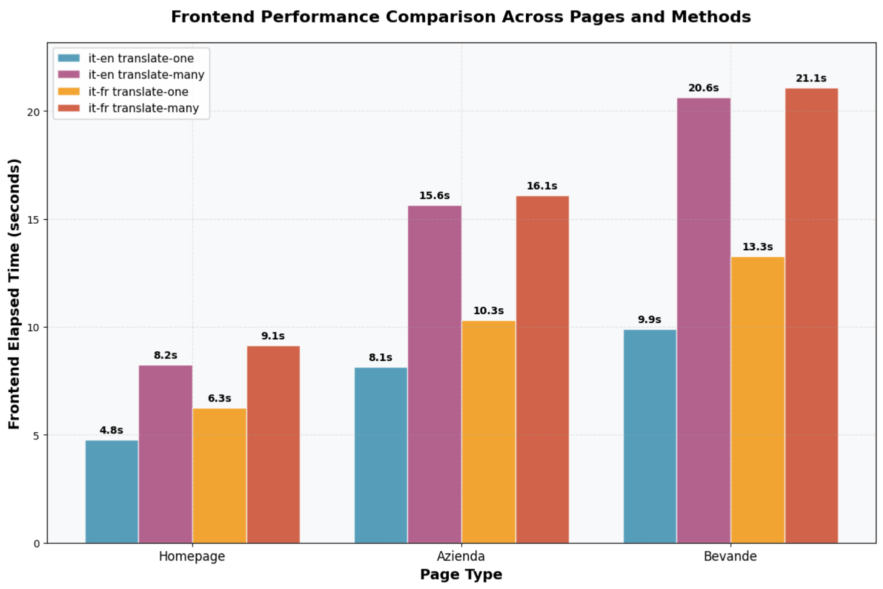


Figure 9.5: Translation performance versus page complexity, showing the effect of increasing the number of elements on front-end elapsed time.

Chapter 10

Conclusion

10.1 Design Overview

10.1.1 Back-End Choices

We chose `FastAPI` to host the back-end server due to its asynchronous design and ease of integration, which made it better suited for high-frequency translation requests than alternatives. For model inference, we compared Hugging Face with Ollama. While Hugging Face models offered slightly higher BLEU scores, Ollama was ultimately preferred because of its lighter setup, faster response times, and more stable runtime behavior. This decision reflected a focus on user experience and system robustness rather than translation accuracy.

A caching mechanism and prompt templates were introduced to further enhance responsiveness and reliability, ensuring the system remained efficient and coherent even under repeated or varied inputs.

10.1.2 Front-End Choices

The front-end module was designed around a `MutationObserver`, chosen over polling because it detects changes immediately without unnecessary overhead. We also adopted a minimal injected UI and a flexible integration approach (remote or bundled) to maximize compatibility with different applications. Between the two back-end strategies, `/translate-one` is used every time a dynamic update is selected, whereas on mount for static elements both `/translate-many` and `/translate-one` were analysed.

In practice `/translate-many` provided a worse performance with respect to `/translate-one`, but this could easily change in a real world enviroment where the back-end is not running locally.

10.1.3 Evaluation Insights

Evaluation showed that the `gemma3` models provided the best balance between accuracy and speed on the available hardware. More importantly, caching proved essential: it reduced repeated translation latency to near zero, making the system feel instant in real-world use where multiple hosts access the web application.

10.2 Future Directions and Improvements

The results obtained suggest several directions for future development and optimization:

Model Optimization

- Exploring advanced runtimes such as vLLM, ExLlama, or TensorRT to reduce inference latency.
- Incorporating models trained specifically for translation tasks to improve BLEU scores and semantic accuracy.

Scalability and Deployment

- Extending the system to cloud or hybrid environments to leverage more powerful GPUs.
- Supporting multi-user concurrent sessions while preserving low latency.

Evaluation Beyond BLEU

- Incorporating human-in-the-loop evaluation methods to assess contextual correctness and usability.
- Expanding test cases to include more diverse languages, including those with complex morphology and scripts.

10.3 Final Remarks

This project successfully delivered a working prototype of a **real-time GUI translation module** powered by LLMs. The architecture—combining a

FastAPI back-end, Ollama runtime, and JavaScript front-end observer—proved effective in addressing the challenges of dynamic content translation.

While limitations remain in terms of translation accuracy and runtime optimization, the integration of LLMs into web front-ends opens new opportunities for building **globally accessible applications**. The system not only validates the feasibility of this approach but also provides a solid foundation for further research and production-ready implementations.