Development of a Natural Language to Structured API Data Processing System with a Smart Home Use Case Demonstration

Albert Sagarra

January 19th, 2025

Abstract

This project presents the development of **NL2API**, a **Natural Language to Structured API Data Processing System** that enables intuitive human–machine communication by converting natural language (text and voice) commands into executable API calls. Leveraging advancements in **Large Language Models (LLMs)**, the system detects user intent and translates it into structured workflows that operate across diverse APIs.
A profiling framework provides an abstraction layer to accommodate heterogeneous API definitions, allowing straightforward integration into multiple domains. We evaluated several on-premises and cloud-based LLMs to identify optimal configurations for accuracy, multilingual capability, and performance.
To illustrate its effectiveness, the system was deployed in a smart home environment, automating control of various IoT devices through natural language commands. By bridging the gap between human language and machine-oriented interfaces, this work demonstrates the transformative role of NLP in simplifying and broadening access to complex, API-driven systems.

1 Motivation

Cambridge Dictionary defines natural language as distinct from computer languages: "language that has developed in the usual way as a method of communicating between people, rather than language that has been created, for example, for computers" [1]. This distinction underscores the core challenge of this project: enabling humans to interact with systems that are traditionally governed by structured, machine-oriented grammars, using natural language.

The foundations of computational linguistics emerged from two distinct but influential approaches in the late 1950s. **Noam Chomsky** introduced *generative grammar* [2], a formal framework for describing the syntactic structures of natural language, which emphasized rule-based systems and theoretical precision. In contrast, **Zellig Harris** focused on *distributional semantics* [3], an empirical approach that analyzed patterns and relationships in linguistic data to understand meaning. These paradigms, though different in focus, collectively shaped the early development of computational linguistics and Natural Language Processing (NLP).

In the 1980s and 1990s, NLP began to move away from rigid, rule-based systems and adopt statistical methods, inspired by the probabilistic ideas of **Claude Shannon**'s 1948 paper *"A Mathematical Theory of Communication"* [4]. These methods allowed systems to handle the variability and ambiguity of natural language by modeling probabilities instead of relying solely on predefined rules. Techniques like *Hidden Markov Models (HMMs)* [5] and *IBM's Statistical Machine Translation* [6] introduced frameworks for analyzing patterns in language, making it possible to process linguistic data at scale. This marked a major shift toward data-driven approaches, building on earlier theoretical foundations while addressing their limitations.

One of the key advances in this statistical era was the ability to model sequences and states, which provided a foundation for representing the context and structure of language. This idea paved the way for the emergence of *Recurrent Neural Networks (RNNs)* in the late 1990s and early 2000s. RNNs, and later *Long Short-Term Memory (LSTM)* networks, brought a new level of sophistication to NLP by enabling systems to "remember" relevant information across longer sequences. These breakthroughs led to significant progress in tasks such as speech recognition, text generation, and machine translation. However, RNNs and LSTMs struggled with scaling to large datasets and capturing long-range dependencies in text.

The 2010s saw critical advancements addressing these limitations, including the development of word embeddings like *Word2Vec* and *GloVe*, which allowed models to represent words as vectors in a high-dimensional space [7, 8]. These embeddings enabled models to capture semantic relationships more effectively, improving the performance of many NLP tasks. Additionally, techniques like unsupervised pre-training and transfer learning became widely adopted, allowing models to leverage massive datasets to generalize better across tasks.

The publication of the paper *"Attention is All*

You Need" [9] in 2017 revolutionized the field further. It introduced the *transformer model*, which replaced the sequential nature of RNNs with a *self-attention mechanism* capable of processing entire sequences in parallel. Transformers excelled at capturing long-range dependencies and relationships in text, making them highly efficient for large-scale language processing and generation. These innovations paved the way for today's *Large Language Models (LLMs)*, the culmination of decades of progress in NLP.

These advancements have created a vast array of opportunities for emerging natural language applications. It is the breakthrough of LLMs that this project leverages for user intent detection, serving as the first step in human-to-machine interaction. Our project is motivated by the potential to offer a user-friendly alternative to complex front-end applications by enabling natural language interaction with system RESTful APIs. With this approach, users ("people," as referenced in the Cambridge Dictionary) can communicate their intentions to the system as naturally as they would to another person. The potential applications of our NL2API project are numerous, and implementation becomes straightforward wherever an OpenAPI specification exists. Examples include restaurant reservations, healthcare appointment scheduling, or, as showcased in this project, interaction with a Smart Home system.

2 Objectives

To address the challenges and opportunities outlined in the motivation, this project is guided by the following objectives:

- Leverage **Large Language Models (LLMs)** to detect and interpret user intent to translate natural language commands into structured API calls, allowing users to interact with machine systems through intuitive, accessible interfaces.
- Design a profiling framework capable of handling diverse API definitions flexibly and ensuring compatibility with heterogeneous systems.
- Demonstrate the practical application of this system in a smart home use case, showcasing its ability to control devices across various technologies (Zigbee, Z-Wave, Wi-Fi, Radiofrequency, IR) via automated workflows across a RESTful API.
- Evaluate the scalability and adaptability of the system to other domains, ensuring its potential for broader applications beyond the initial smart home environment.

By pursuing these objectives, this project aims to illustrate how cutting-edge NLP technologies can be harnessed to simplify human-machine interactions and enable more efficient and user-friendly systems.

3 State of the Art

The rapid development of **Large Language Models (LLMs)**, such as OpenAI's GPT-4[10] and Meta's Llama[11], has significantly advanced natural language processing (NLP) by enabling systems to understand and generate human-like text with remarkable accuracy. These models excel at reasoning and handling unstructured inputs, but their integration into structured workflows, such as translating natural language into API calls, remains a challenge due to the gap between natural language flexibility and machine-oriented syntax.

Recent innovations address these challenges through modular approaches that decompose tasks into distinct phases, such as intent recognition, reasoning, and execution. For example, techniques like *chain-of-thought prompting* [12] guide models to articulate intermediate steps, improving performance in complex reasoning tasks. Similarly, *decomposed prompting* [13] divides tasks into smaller subtasks, enhancing modularity and adaptability. These advancements underscore the importance of structured, multi-step reasoning in enabling scalable and flexible workflows.

Frameworks such as *LangChain*[14] and OpenAI's Plugins[15] further demonstrate how LLMs can interact with structured systems by bridging the gap between unstructured language inputs and precise API execution. Additionally, reasoning methods like semantic parsing [16] equip models with tools to translate user intent into structured API workflows, aligning closely with this project's goals.

Rather than competing with state-of-the-art LLM implementations, this project focuses on developing a flexible, modular pipeline designed to integrate input methods and a profiling system. This foundation is intentionally built to support future extensions, including advanced multi-step reasoning techniques, ensuring scalability and adaptability for diverse use cases.

4 Methodology

The development of this project followed an **Agile approach**[17], which proved essential for handling the system's complex and modular architecture. This decision was driven by the need to continuously refine and integrate key components, including bots for multi-channel input (WhatsApp, Telegram), voice-to-text processing, natural language understanding, and an API execution controller.

Each module was developed and iterated upon independently, adhering to Agile's principle of continuous improvement. For instance, the integration with Whisper[18] for voice-to-text processing and the implementation of semantic intent mapping were delivered incrementally, ensuring seamless interaction with other system components. This approach allowed the project to adapt dynamically to challenges encountered during development and facilitated early testing and validation

of individual components.

The need for frequent testing and feedback loops was especially critical given the reliance on external input channels, such as WhatsApp and Telegram. Agile's flexibility ensured that evolving platform changes, such as updates to APIs or communication protocols, were smoothly incorporated within iterative development cycles. This responsiveness minimized disruption and maintained system functionality throughout the project lifecycle.

The modular architecture—comprising Stream Harmonization, Voice-to-Text, LLM, and API Command Execution Controller—was particularly suited to Agile development. Independent development and testing of these modules allowed for early identification and resolution of integration issues. For example, the interaction between the LLM and profiling API grammars was tested incrementally, ensuring alignment with system requirements and user needs.

In retrospect, a traditional waterfall approach would have been ill-suited for this project. The need for frequent adaptation to external dependencies, iterative refinement of user interactions, and the rapid evolution of platform technologies necessitated Agile's iterative and flexible process. This methodology not only enabled timely delivery of individual components but also ensured the overall system met its design objectives efficiently.

Because of the parallel development of the different modules, we identified the managed to A detailed Currently, with the selection of a specific LLM finalized, the focus has shifted to building the profiling framework to support multi-user and multi-API functionality. This phase aims to enable the system to dynamically handle diverse user profiles and adapt to various API specifications, ensuring scalability and versatility for broader applications.

5 System Architecture

5.1 Overview

Based on the functional diagram conceived during the initial project definition, technologies were carefully selected to address the system's specific challenges. At the time of this report, all prototype iterations have been completed, and components have been validated or modified, resulting in a finalized architecture that is largely operational.

The architecture adheres to the principles of **Infrastructure as Code (IaC)**, enabling seamless deployment across on-premises hardware or cloud-based environments. Defined through a few configuration files, the IaC setup includes containers, storage, secrets, networking, and code. This flexibility allows the infrastructure to be deployed either on lightweight edge devices via Docker Compose or in robust cloud environments using Kubernetes.

5.2 System Components

System Inputs

The system accepts inputs from both Telegram and WhatsApp platforms. Telegram integrates through its public Bot API [19], which supports webhooks for posting messages to an HTTPS endpoint, requiring only a valid SSL/TLS certificate. For WhatsApp, the Business API[20] did not meet the project's requirements. Instead, we used mywhin[21], a custom RESTful API Go implementation of WhatsApp Web. This client operates in a containerized environment, establishing socket communication with WhatsApp servers to handle bidirectional message exchange. Incoming messages are forwarded to the Node.js server, and outgoing messages are sent in the reverse direction. Although lightweight in terms of CPU and memory usage, the WhatsApp client relies on a relational database management system (RDBMS) for storing and managing necessary data entities. A PostgreSQL[22] server is deployed as a container within the system architecture to meet this requirement. Once the input setup is complete, messages (text or voice) from both Telegram and WhatsApp are directed to the main Node.js runtime for processing.

Node.js Server Runtime

The system's functional workflows are implemented using Node-RED[23], which operates on a Node.js server. Node-RED runs as an Alpine[24] Linux-based container within the system. This choice was made for its performance efficiency and suitability for designing and controlling functional flows. While lightweight in terms of system resource consumption, Node-RED serves as the core of the architecture, orchestrating the entire workflow. All machine learning inference tasks are handled externally, keeping the runtime's resource demands minimal.

Our node-red server runs the five blocks represented in the diagram inegrating them into a single pipeline that orchestrates external requests and ensures the right execution without additional components.

Voice-to-Text: Whisper

Whisper, a state-of-the-art Speech-to-Text (STT) system, is based on a transformer architecture and has been trained on a vast dataset. Its end-to-end design eliminates intermediate processing stages, simplifying the pipeline and enabling accurate transcriptions across 98 languages. Despite its high accuracy, Whisper's computational requirements are significant. During testing, the Nvidia Jetson Orin 16GB[25] achieved highly accurate transcription results, but the processing speed was a limiting factor. Tests showed that OpenAI's Whisper API [26] was approximately three times faster than local deployment. Given the minimal cost difference and the higher electricity consumption of local inference, the system architecture opted for OpenAI's API for Whisper-based transcription. In summary, while Whisper

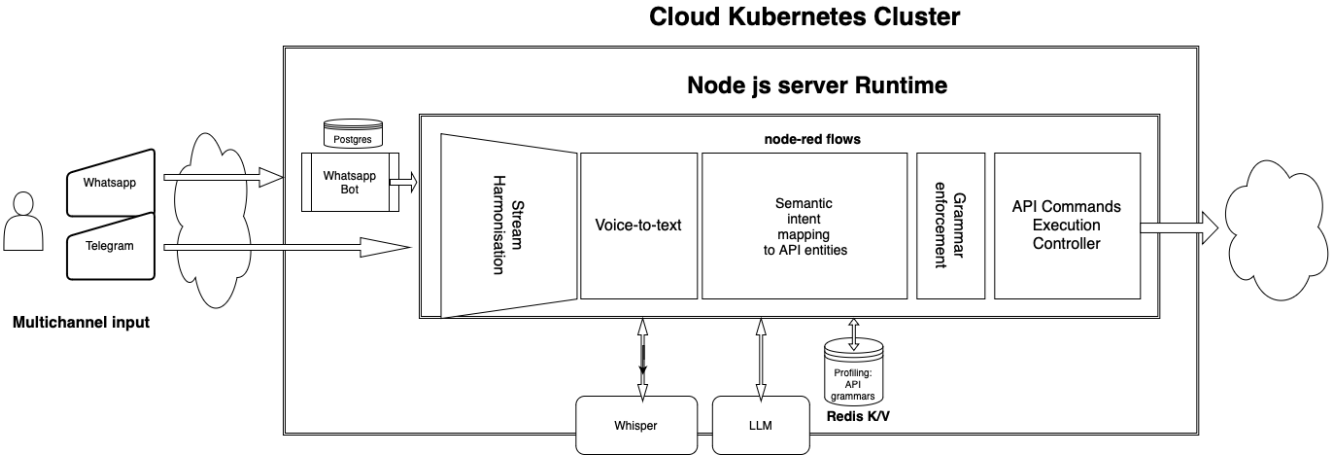

Figure 1: High-level system diagram

remains integral to the system's architecture, it is used via OpenAI's API, foregoing local deployment to prioritize efficiency and performance.

Semantic Intent Mapping: LLMs
This component represents the core of the project and directly drives its primary objectives. As discussed in the Motivation section, advancements such as semantic codification and the attention mechanism introduced by transformers have revolutionized this domain. Large Language Models (LLMs) have transformed the task of semantic intent mapping from a highly complex challenge into a streamlined inference process. Multiple strategies exist to address this task, each with various dimensions and experimental configurations. Given the extensive range of possibilities, we dedicated weeks to a thorough analysis to identify the optimal solution and architecture for our system. A detailed summary of experimental results is provided in a dedicated section of this paper. To briefly outline our chosen architecture, we have opted to use OpenAI's API [27] for semantic intent mapping, similar to our approach with Whisper. Consequently, there is no requirement to deploy our own inference engine, although local inference alternatives that were tested ran using Llama.cpp and Ollama engines. Our final product, while initially wired to OpenAI's cloud solution can be configured to run any of the alternatives that were considered and thoroughly tested during evaluation. Our Infrastructure-as-Code deployment file includes Ollama container although the local models must be downloaded from huggingface[28] as part of the configuration.
Profiling & grammars: Redis
Redis [29] is an ultra-fast, in-memory key-value store perfect for high-performance data pipelines, especially in systems requiring real-time access to user-specific data. In our setup, Redis efficiently stores user profiles and OpenAPI definitions, enabling rapid retrieval of personalized data to generate dynamic prompts tailored to each user and input method. This capability ensures quick and seamless integration with large language models (LLMs), where low-latency access to contextual information is crucial for generating accurate and relevant outputs. Redis is the ideal choice for systems that demand speed and scalability, making it a key component in optimizing LLM-driven workflows. As part of our system, we run a Redis server in a container. Unlike other system components, running Redis locally provides a significant performance boost, as it operates with minimal resource usage, ensuring both speed and efficiency.

5.3 Network & Security

Although showcased in a SmartHome scenario, our system architecture has potential applications involving sensitive activities, such as accessing bank account balances or medical records. Consequently, ensuring robust security measures is a critical aspect of the design.

The first layer of security relies on the user's phone, which is assumed to be a controlled and secure environment. For scenarios involving highly sensitive data, we recommend implementing a second-factor authentication mechanism. This precaution safeguards against potential compromises in input channels (e.g., WhatsApp or Telegram) or vulnerabilities in the user's phone.

To protect the system's internal architecture, all containers operate within an isolated virtual network. This setup ensures that no container ports are exposed, even to the host machine running the containers. Communication between system components occurs exclusively within this virtual network, eliminating the risk of network sniffing as there is no external physical network involved.

The system has two controlled external entry points:

1. **WhatsApp**: All communication with WhatsApp servers is secured using end-to-end encryption

(E2EE) based on the Signal Protocol[30]. Additionally, connections with WhatsApp servers are secured through TLS encryption.

2. **Telegram**: Messages are received via a public endpoint engineered with a secure Cloudflare tunnel[31]. This tunnel provides SSL/TLS encryption, eliminates public IP exposure, and includes DDoS protection and malicious traffic filtering. Only requests originating from Cloudflare's trusted network are accepted, ensuring that only legitimate traffic can interact with the system.

Beyond these entry points, the system is fully sealed from external access. The design allows for optional restrictions, such as disabling shell access to containers, creating an armored deployment for environments requiring additional security.

Enterprise-class authentication is utilized for external services, including Cloudflare, OpenAI, and GitHub. These services require API tokens or keys, which must be securely generated and obtained during the system onboarding process. Token-based authentication ensures that all external interactions are validated and authorized, enhancing the system's overall security.

Outbound communications, such as those to external APIs (e.g., OpenAI), are strictly controlled. All outgoing requests originate from within the containerized network and utilize HTTPS with authentication and certificates, ensuring secure and validated communication.

6 Software & Configuration

6.1 Infrastructure Deployment

To create a replicable and scalable environment, this project employs a virtualized infrastructure that can be deployed using container-based solutions. Whether one chooses Docker Compose, Docker Swarm, Kubernetes, or Red Hat OpenShift, the infrastructure can be adapted with minimal adjustments to the configuration.

The infrastructure is defined in a single YAML file, which describes how the architecture is instantiated by the orchestrator. This file specifies five services: *nl2api-nr*, *nl2api-pg*, *nl2api-redis*, *nl2api-whin*, and *nl2api-cf*. Container images and environment variables are predefined, enabling automatic deployment of all components.

A `secrets.json` file stores user credentials for external services, such as Telegram Bot, Cloudflare, OpenAI and Antrophic's api keys. Additionally, a *flows.json* file includes nodeJS runtime code. Before the system operates, a setup process configures communication channels, input profiles, and target APIs.

All containers operate within an isolated virtual network. A *cloudflared* tunnel container provides a secure ingress point for Telegram webhooks. The project package includes *infrastructure.yaml*, *secrets.json*, and *flows.json* files to simplify deployment.

6.2 Telegram Configuration

Telegram configuration begins by creating a dedicated bot using Telegram's "Botfather"[32] service. Once the bot is created, Botfather provides an API token used to define a webhook endpoint. This webhook forwards incoming messages to the *nl2api* endpoint, enabling multiple users to interact through a single bot instance. Assigning one bot per RESTful API simplifies the overall setup.

Using the API token, the system exposes the *nl2api* public URL, which requires TLS/SSL with a valid certificate. The *cloudflared* tunnel establishes a TLS-encrypted connection between the local environment and Cloudflare's edge network. Configuring a basic Cloudflare tunnel and storing the necessary token allows the *nl2api* endpoint to register with Telegram's public API, ensuring secure message delivery.

6.3 WhatsApp Configuration

Unlike Telegram, WhatsApp lacks native bot support. To overcome this limitation, we developed "mywhin" as a side-project. Mywhin is a custom WhatsApp Web client implemented in Go that provides a RESTful API wrapper to WhatsApp functionalities such us bidirectional messaging.

During deployment, a container is automatically created for the *nl2api-whin* service alongside a PostgreSQL container for data storage. Upon initiating the *nl2api-whin* container, the console displays a QR code, which the user scans with WhatsApp's "Linked Devices" feature to link their phone.

Once our phone line is linked, incoming messages are securely routed from *nl2api-whin* to *nl2api-nr* within the isolated network. It is important to note that communication between our WhatsApp client and WhatsApp network is initiated by out client as an outgoing websocket that operates only with end to end Whatsapp signaling protocol.

6.4 Input Harmonization & Voice-to-Text

Once Telegram and WhatsApp channels are configured, the system begins receiving text-based requests. The first pipeline stage, Input Harmonization, parses incoming messages from various sources and converts them into a standardized command format.

For voice commands, the input is transcribed using Whisper's service. While an on-premises Whisper container is available, the current implementation relies on OpenAI's cloud-based API for simplicity. An OpenAI API key, stored in `secrets.json`, enables this functionality. Regardless of the original input type, all commands are standardized for downstream processing.

6.5 Semantic Intent & Grammar enforcement Configuration

As explained across different sections in the document, we tested different alternatives for capturing the semantic intent of the user request. Although our use case had a winner in our tests, all preselected alternatives are usable inside Nl2API with little configuration.

Switching from OpenAI's GPT-4o default selection to Antrophic's model (Claude Sonnet 3.51) requires only a wire change into node-red visual pipeline tool, assuming the api key credentials for both providers are stored into the secrets.json file.

In case we want to swithch to an on-premise model, there are a few additional considerations that need to be taken care of. For simplicity, our IaC deployment fine includes Ollama engine. Depending on the containers host architecture to access GPU compute resources, the inference engine may require tailored parameters. LLM models can be downloaded using Ollama's interface or directly from huggingface website. Once the desired model is locally available, a simple switch will enable it inside NL2API.

Target APIs demand specific syntax and grammar configurations to execute the actions. These are defined as part of the profiles and incorporated to the LLM requests. While in most of the tested APIs most of the system responses are compliant to the requirements, sometimes a posterior grammar enforcement step is necessary to elminate redundancies, repetitions or explanations. LLMs are not deterministic systems by definition and it is always neccessary to check the compliance of our output to the taget API requirements. To that end, our quality check might need to be adapted to other requirements. Typically, however, URL and JSON parsing are sufficient to comply with most of the APIs.

6.6 Profiling: API Grammars Configuration & Commands Execution Controller

User and API profiles are stored in a Redis[29] database. These profiles define authorized users (e.g., Telegram usernames or WhatsApp accounts) and specify API details, such as SmartHome endpoint configurations.

Profile data is managed via the `flows.json` file, which runs within the Node-RED server. Profiles are Currently created manually although future enhancements may include endpoints for remote profile management.

The pipeline concludes by executing commands derived from grammar-defined operations. Since the grammar is specified within the profiling module, no additional configuration is needed for the controller.

Future versions of the system could implement safeguards to prevent errors or restrict user actions. Assessing the necessity of such features depends on the specific APIs being targeted by the *nl2api* system.

7 Results

This section presents the key achievements of our project, demonstrating how the system works end-to-end. We also outline the results of the comparative evaluation of various Large Language Models (LLMs), which helped inform our final choice for the NL2API framework.

7.1 Achieving Our Objectives

We aimed to investigate whether state-of-the-art Natural Language Processing (NLP) technologies—particularly Large Language Models—could serve as intuitive interfaces to complex API-based systems. The following objectives were central to our study:

- **Robust Intent Extraction.** We sought to convert complex, multilingual user requests into structured API calls without relying on rigid keywords or predefined phrases.
- **Modular Piping and Profiling.** We needed a flexible system architecture capable of integrating diverse APIs, input/output methods, and multiple LLM backends.
- **SmartHome Demonstration.** We aimed to validate our approach in a practical setup, controlling multiple IoT devices seamlessly through natural-language commands.
- **Generality Beyond SmartHome.** We also tested the framework in a different domain (3D model visualization) to confirm its adaptability.

Overall, we found these objectives were broadly met. The following subsections provide further details on how each component of our framework performed, culminating in the results of our LLM experimentation.

7.2 End-to-End Flow

Once all modules were implemented, our end-to-end system successfully interpreted diverse user commands and executed corresponding API calls. Below is a brief outline of the process:

- **Input (WhatsApp Audio).** Users sent audio messages (e.g., *"Let the sun enter the office"*) to a designated WhatsApp bot.
- **Harmonization.** The raw audio was extracted and standardized to a consistent format for processing.
- **Voice-to-Text.** Using OpenAI Whisper (or an on-premises speech-to-text alternative), the audio was transcribed into text.
- **API Profiling and Prompt Generation.** The system selected the appropriate API (e.g., a SmartHome service) and created a structured

prompt encapsulating both user intent and API syntax rules.

- **LLM Interpretation.** A Large Language Model analyzed the prompt, identified user intent (e.g., "raise the office blind"), and produced a suggested API endpoint or method call.
- **Grammar Enforcement.** The LLM's output was validated and normalized against a strict API schema. Any extraneous text was stripped out, and the final call was formatted as an HTTP request.
- **Execution.** The request was queued and executed, effectively raising the office blind in the example scenario.

Through this pipeline, the command *"Let the sun enter the office"* triggered the blinds to open, even though "blind" was never explicitly mentioned—demonstrating more flexible understanding than typical keyword-driven systems.

7.3 LLM Experimentation

To determine the best-performing LLM for our NL2API framework, we evaluated two on-premises and two cloud-based models under identical conditions. We measured both *accuracy* (as an average score from 1–5, gauging correctness in mapping user intent) and *latency* (in seconds). The experiment detailed process, together with the dataset used can be found in the appendix A.

Table 1: LLM evaluation average results.

Model	Score (1–5)	Latency (s)
LLaMA-3.2-3B	2.6	5.8
Mistral-NeMo	2.3	8.2
GPT-4o	4.7	1.5
Claude 3.5 Sonnet	4.4	2.8

Accuracy and Multilingual Support. Both smaller on-premises models (LLaMA-3.2-3B and Mistral-NeMo) handled basic requests adequately (e.g., *"Open the garage door"*), particularly in English. However, they struggled with more nuanced or multilingual commands, such as Catalan queries. By contrast, the two cloud-based solutions—GPT-4o and Claude 3.5 Sonnet—handled complex linguistic variants more consistently, yielding higher average scores (4.7 and 4.4, respectively).

Latency and Hardware Constraints. On our Jetson edge device, smaller models remained within acceptable latency ranges (5.8–8.2 seconds) when quantized effectively. The cloud-based services averaged latency below 3 seconds thanks to powerful remote infrastructure. For low-frequency tasks in a typical SmartHome context, the additional local overhead did not confer a clear advantage.

Cost Considerations. Given a moderate command volume, the cost of cloud inference was not prohibitive. On-prem solutions could become more viable if high traffic significantly increases cloud usage costs, if data confidentiality is paramount, or if internet dependencies must be avoided. However, for our primary use case, GPT-4o's balance of accuracy and speed made it the most practical choice.

Summary of Findings.

- *Cloud-Based Models Outperform On-Premises for Complexity and Multilinguality.*
- *GPT-4o Achieves the Highest Accuracy (4.7) and Lowest Latency (1.5s).*
- *Claude 3.5 Sonnet is a Close Second and a Good Backup.*
- *On-Premises Models Are Cheaper Only at High Usage or Under Strict Privacy Constraints.*

7.4 Discussion and System-Level Performance

Overall, our experiments confirmed that advanced LLMs dramatically improve intent recognition over naive keyword matching. The system's modular design simplifies switching between on-premises and cloud-based LLMs, supporting flexibility for different deployment scenarios. Combined with robust grammar enforcement, the NL2API framework can adapt to other domains beyond SmartHome, as demonstrated in a pilot project guiding 3D model visualization through voice commands.

8 Conclusions

This work demonstrates how recent advances in Natural Language Processing (NLP) can provide accessible interfaces to API-driven environments, reducing the need for specialized front-end applications. By leveraging Large Language Models (LLMs), we enabled users to issue complex, multilingual voice commands without referencing explicit keywords or device IDs. NL2API framework accommodates different APIs, LLMs, and input methods, making it adaptable across domains.

Beyond these core successes, our results indicate that voice-driven, language-based interfaces are not only more intuitive but also more powerful than traditional approaches—especially when augmented by modern LLM capabilities. This aligns with broader research efforts focused on natural language interfaces, including those translating text into structured queries or commands [33, 34].

Nevertheless, certain limitations highlight the need for continued refinement. For instance, tailoring models to specific domains or user nuances requires ongoing training and robust grammar enforcement. The reliance on cloud-based inference, while beneficial for most SmartHome scenarios, raises questions about cost, data privacy, and latency under different usage patterns.

9 Future Work

Despite the challenges involved in setting up a complete end-to-end architecture for our system, we have only begun to explore its possibilities. The results of this project have demonstrated the potential for integrating diverse APIs, but further enhancements and expansions are required to fully unlock the framework's capabilities. Below, we outline the most promising directions for future development:

- **Parallel models.** The NL2API framework architecture is robust enough to support the deployment of multiple Large Language Models (LLMs) in parallel. By incorporating model selection and parameterization into the profiling module, the system could dynamically adapt to different use cases. While our testing focused on the SmartHome use case, future applications could involve diverse scenarios, enabling optimal LLM selection and configuration for specific needs.

- **Two-pass approach for modular tasks.** In its current implementation, the system processes intent detection and grammar specification as a single step. Introducing a two-pass approach, where these tasks are handled sequentially using distinct stages or models, could improve task clarity and accuracy. This modular design would provide a foundation for handling more complex workflows and serve as an intermediate step toward advanced reasoning techniques.

- **Multi-step reasoning for complex queries.** Building upon the two-pass approach, the framework could adopt multi-step reasoning techniques such as *chain-of-thought prompting* [12] to handle intricate queries or workflows. This would involve breaking down user intents into multiple reasoning steps, with intermediate outputs guiding subsequent stages. Such techniques would be particularly beneficial in domains requiring layered decision-making or intricate automation scenarios.

- **Input methods.** The current input methods, based on popular instant messaging applications, may not be sufficient for all use cases. Although the harmonization module allows easy integration with other systems, expanding functionality will require the development of specific parsers for new input payloads. This enhancement could enable compatibility with a wider range of devices and communication platforms, facilitating broader adoption.

By addressing these areas, the NL2API framework can evolve into a more versatile and powerful tool for enabling natural language interactions with API-driven systems. These enhancements will ensure broader applicability, improved accuracy, and greater adaptability across diverse domains.

References

[1] Cambridge Dictionary. Natural language, 2023. Accessed: January 2025, Available: `https://dictionary.cambridge.org/dictionary/english/natural-language`.

[2] Noam Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124, 1956. Accessed: December 2024.

[3] Zellig S. Harris. Distributional structure. *Word*, 10(2-3):146–162, 1954. Accessed: December 2024.

[4] Claude E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, 623–656, 1948. Accessed: December 2024.

[5] Lawrence R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989. Accessed: December 2024.

[6] Peter F. Brown, Vincent J. Della Pietra, and Robert L. Mercer. The mathematics of statistical machine translation: Parameter estimation. *Computational Linguistics*, 19(2):263–311, 1993. Accessed: December 2024.

[7] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2013. Accessed: December 2024.

[8] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543. Association for Computational Linguistics, 2014. Accessed: December 2024.

[9] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. Accessed: December 2024.

[10] OpenAI. Gpt-4 technical report, 2023. Accessed: January 2025, Available at `https://openai.com/research/gpt-4`.

[11] Meta AI. *LLaMA: Large Language Model Meta AI*, 2023. Accessed: January 2025, Available: https://ai.meta.com/llama/.

[12] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, 2022. Accessed: January 2025.

[13] Tushar Khot, Harsh Trivedi, Matthew Finlayson, Yao Fu, Kyle Richardson, Peter Clark, and Ashish Sabharwal. Decomposed prompting: A modular approach for solving complex tasks. In *International Conference on Learning Representations*, 2023. Accessed: January 2025.

[14] Xiaoyang Li, Wei Zhang, Yanan Liu, Zhen Wang, Ming Li, Zhen Li, and Zhi Li. Application development exploration and practice based on langchain+chatglm+rasa. In *2023 IEEE 6th International Conference on Computer Communication and the Internet (ICCCI)*, pages 309–313. IEEE, 2023. Accessed: January 2025.

[15] OpenAI. Openai plugins documentation, 2023. Accessed: January 2025, Available at https://platform.openai.com/docs/plugins.

[16] Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah D. Goodman. Star: Bootstrapping reasoning with reasoning. In *Advances in Neural Information Processing Systems*, 2022. Accessed: January 2025.

[17] Wikipedia contributors. Agile software development. Wikipedia, The Free Encyclopedia, 2025. Accessed: January 2025, Available: https://en.wikipedia.org/wiki/Agile_software_development.

[18] Alec Radford et al. Robust speech recognition via large-scale weak supervision. *arXiv preprint arXiv:2212.04356*, 2022. Accessed: January 2025, Available: https://arxiv.org/abs/2212.04356.

[19] Telegram. Telegram bot api. https://core.telegram.org/bots/api, 2024. Accessed: January 2025, Available: https://core.telegram.org/bots/api.

[20] Meta Platforms. Whatsapp business platform documentation, 2023. Accessed: January 2025, Available at https://developers.facebook.com/docs/whatsapp.

[21] Inutil Labs. Mywhin api documentation, 2023. Accessed: January 2025, Custom RESTful API client developed using a Go implementation of WhatsApp Web. Available at https://rapidapi.com/inutil-inutil-default/api/mywhin.

[22] The PostgreSQL Global Development Group. Postgresql: The world's most advanced open source relational database, 2023. Accessed: January 2025, Available at https://www.postgresql.org.

[23] IBM Emerging Technologies. Node-red, 2023. Accessed: January 2025, Available: https://nodered.org/.

[24] Alpine Linux Community. Alpine linux, 2023. Accessed: January 2025, Available: https://www.alpinelinux.org/.

[25] NVIDIA Corporation. Nvidia jetson orin nx 16gb developer kit, 2023. Accessed: January 2025, Available at https://developer.nvidia.com/embedded/jetson-orin-nx.

[26] OpenAI. Whisper: Robust speech recognition model, 2023. Accessed: January 2025, Available: https://openai.com/research/whisper.

[27] OpenAI. *OpenAI GPT API Documentation: Overview*, 2024. Accessed: January 2025, Available: https://platform.openai.com/docs/overview.

[28] Hugging Face, Inc. *Hugging Face: The AI Community Building the Future*, 2024. Accessed: January 2025, Available: https://huggingface.co.

[29] Redis Ltd. Redis: Open source in-memory data store, 2023. Accessed: January 2025, Version 7.2, Available: https://redis.io.

[30] Open Whisper Systems. The signal protocol, 2023. Accessed: January 2025, Available at https://signal.org/docs/.

[31] Inc. Cloudflare. Cloudflare tunnel documentation, 2023. Accessed: January 2025, Available at https://developers.cloudflare.com/cloudflare-one/connections/connect-apps/.

[32] Telegram. Botfather: The one bot to rule them all, 2023. Accessed: January 2025, Available at https://core.telegram.org/bots#botfather.

[33] Ruoxi Sun, Sercan Ö Arik, Alex Muzio, Lesly Miculicich, Satya Gundabathula, Pengcheng Yin, Hanjun Dai, Hootan Nakhost, Rajarishi Sinha, Zifeng Wang, and Tomas Pfister. Sql-palm: Improved large language model adaptation for text-to-sql (extended). *arXiv preprint arXiv:2306.00739*, 2023. Accessed: January 2025, Available: https://arxiv.org/abs/2306.00739.

[34] Zhao Tan, Xiping Liu, Qing Shu, Xi Li, Changxuan Wan, Dexi Liu, Qizhi Wan, and Guoqiong Liao. Enhancing text-to-sql capabilities of large language models through tailored promptings. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*. ELRA and ICCL, 2024. Accessed: January 2025.

[35] Anthropic. Introducing claude 3.5 sonnet, 2024. Accessed: January 2025, Available at https://www.anthropic.com/news/claude-3-5-sonnet.

A Experimental Evaluation for NL2API LLM Selection

Selecting an appropriate Large Language Model (LLM) is central to the NL2API project. The conversion of natural language into accurate API calls fundamentally relies on utilizing the right inference engine with the optimal model. Our initial tests with cloud-based systems yielded satisfactory results. However, exploring the deployment of a successful open-source on-premises solution would represent a significant breakthrough for several reasons.

To evaluate the suitability of cloud-deployed versus on-premises models, we compare their respective advantages and disadvantages in Table 2.

Aspect	**Cloud-Deployed Models**	**On-Prem Deployments**
Pros	- No infrastructure or maintenance overhead - Access to state-of-the-art models - Scalable for variable workloads - Low upfront cost - Easy integration and advanced tooling	- Full control over data and models - Ensures data privacy and compliance - Cost-efficient for frequent use - Customizable and fine-tunable - Works offline and ensures reliability
Cons	- High long-term costs for frequent use - Data privacy concerns - Dependent on internet connectivity - Limited customization - Vendor lock-in	- High upfront cost for hardware - Requires technical expertise to manage - Scaling requires additional hardware - May not match cloud model performance - Maintenance burden

Table 2: Cloud-Deployed Models vs On-Prem Deployments

As is often the case, there is no single best solution; different use cases require different approaches. When compliance and confidentiality are critical requirements, an on-premises deployment becomes imperative. Conversely, when these factors are less stringent, one must balance quality and speed against cost considerations.

Because there is no *one size fits all* solution, our architecture is designed to be modular, allowing the LLM component to be interchangeable and tailored to specific tasks. A natural progression for the project would be to incorporate this flexibility into the profiling process, enabling the existing infrastructure to dynamically select the appropriate inference engine and model based on the request.

Cloud-based services were significantly easier to test and evaluate. Although several options are available for comparison, the on-premises architecture offers a broader range of combinations due to its customizable nature.

A.1 Model Candidates Preselection

We examined both open-source models (deployed on-premises) and leading cloud-based offerings, focusing on their performance, resource efficiency, and suitability for edge applications.

On-Premises Models Hugging Face [28] is a platform that offers tools, libraries (such as Transformers), and a hub for hosting machine learning models, datasets, and code, thereby fostering collaboration in AI development. It has become the standard repository for open-source LLMs due to its comprehensive ecosystem, user-friendly APIs, and a large, active community of researchers and developers.

As of mid-December 2024, there are over 1.2 million models available on Hugging Face. Selecting three models is therefore not straightforward, as navigating through the extensive options can be overwhelming. Although narrowing the task significantly reduces the list, it still includes tens of thousands of models. Based on extensive research and user trends, we selected two model families: LLaMA and Mistral.

The LLaMA (Large Language Model Meta AI) [11] family comprises open-source large language models developed by Meta (formerly Facebook). Designed for a wide range of natural language processing tasks, these models prioritize efficiency, enabling high performance even with relatively modest computational resources. Available in various sizes, LLaMA models allow researchers and developers to fine-tune them for specific applications, making them suitable for our NL2API project.

The Mistral family of models represents a new wave of open-source large language models engineered for exceptional efficiency and accuracy in natural language processing tasks. Developed by Mistral AI, these models have quickly gained recognition for their innovative architectures and ability to deliver high-quality results across diverse applications. Designed with scalability in mind, Mistral models can be deployed in various environments, ranging

from resource-constrained devices to high-performance computing clusters, aligning well with the requirements of our NL2API project.

There are various approaches to utilizing large language models (LLMs), including using off-the-shelf models, selecting models fine-tuned for specific tasks (such as instruction-tuned models), or choosing smaller models for independent fine-tuning. Determining the optimal method for a given application is often challenging and cannot be reliably predicted in advance. In this study, since our comparisons are against off-the-shelf, cloud-based alternatives, we did not pursue fine-tuning. Instead, we focused on selecting specialized, task-optimized versions within each model family to achieve the desired performance.

At the time the experiment was conducted, LLaMA 3.2 models were newly released, yet numerous fine-tuning options and quantization levels were already available. For Mistral models, we selected NeMo as the free alternative. Both LLaMA and Mistral models support multiple quantization levels and methods. However, selecting the appropriate combination was challenging, as it required balancing task complexity, performance requirements, and infrastructure constraints. Larger models generally offer better performance for complex tasks but demand significant computational resources. This can be mitigated through aggressive quantization, albeit at the cost of precision. Conversely, smaller models are more resource-efficient but may underperform on tasks that require rich contextual understanding.

Considering our hardware constraints and system requirements (context vs. training), we selected Q4_0_8_8 quantization for both models. This configuration entailed 4-bit weights (Q4), a specific quantization variant (0), and 8-bit precision for both activations and auxiliary data. This setup reduced memory usage and computational demands while maintaining stability and reasonable performance, making it ideal for resource-constrained environments.

Cloud-Based Models For cloud-based inference, we utilized provider APIs without downloading model files. This approach allowed easy access to advanced models without the complexity of local deployment. The two primary cloud-based models used in our evaluation were:

1. **OpenAI's GPT-4o**: Accessible via the OpenAI API, GPT-4o is renowned for its high performance and versatility in understanding and generating natural language. Trained on a diverse and extensive dataset that includes a wide range of internet text, GPT-4 excels at comprehending and producing human-like responses. Its robust architecture effectively handles intent detection by accurately interpreting user commands and context, making it highly suitable for converting natural language into precise API calls. GPT-4o requires valid API credentials and incurs billing based on usage. Additionally, GPT-4 operates on a token-based pricing model, where each API call consumes a certain number of tokens based on the input and output length. Free tiers are available with limited token allocations, allowing initial testing without significant costs.

2. **Anthropic's Claude 3.5 Sonnet**[35]: Accessed through Anthropic's API, Claude 3.5 Sonnet emphasizes safety and alignment, ensuring that generated responses are both accurate and ethically appropriate. It was trained to minimize harmful outputs and enhance contextual understanding, demonstrating strong capabilities in intent detection by reliably interpreting user intentions and reducing ambiguities. Similar to GPT-4o, Claude 3.5 Sonnet requires API credentials and incurs usage costs. Claude 3.5 Sonnet also utilizes a token-based pricing structure, with free tiers providing a limited number of tokens for initial experimentation.

These cloud-based models served as benchmarks for performance and accuracy, often considered state-of-the-art, against which the on-premises models were evaluated. Their advanced training and specialized features in intent detection set a high standard for achieving the objectives of our NL2API project.

A.2 Experiment

A.2.1 Test Commands

We defined the test commands as a collection of sentences containing commands that the system should analyze and map to our API specifications. These commands are categorized into four levels of difficulty: Simple, Complex, Very Difficult, and Absurd. Additionally, we incorporated a multilingual dimension, challenging our system in three languages: Catalan, Spanish, and English.

It is important to note that the "Absurd" category was included to detect and prevent future hallucinations. In the context of LLMs, hallucination refers to the tendency to generate plausible-sounding but factually incorrect, unsupported, or entirely fabricated information based on patterns in their training data.

Regarding languages, some models have been extensively trained in English but have received minimal training in other languages. For instance, Catalan poses a significant challenge to most LLMs.

Finally, it is important to recognize that test commands are inherently biased. A new problem or use case may benefit from rerunning tests tailored to that specific problem and language, as model selection could vary to optimize the results.

Table 3: Test Sentences in English, Spanish, and Catalan

Category	**English**	**Spanish**	**Catalan**
Simple	Open all blinds in the living room	Abre todas las persianas del salón	Obre totes les persianes de la sala d'estar
Complex	Let the sun enter the living room	Deja que entre el sol en el salón	Deixa que entri el sol a la sala d'estar
Very Difficult	It is too dark in the living room	Está demasiado oscuro en el salón	Fa massa foscor a la sala d'estar
Absurd	My tailor is rich	Mi sastre es rico	El meu sastre és ric
Simple	Open the garage door	Abre la puerta del garaje	Obre la porta del garatge
Complex	Need to park, open up	Necesito aparcar, abre la entrada	Necessito aparcar, obre l'entrada
Very Difficult	I'm arriving home by driving	Estoy llegando a casa conduciendo	Estic arribant a casa conduint
Absurd	Home car light blue	Hogar coche azul claro	Casa cotxe blau clar
Simple	Switch the pool pump off and turn on the pool lights	Apaga la bomba de la piscina y enciende las luces de la piscina	Apaga la bomba de la piscina i encén els llums de la piscina
Complex	Prepare the pool for a night swim	Prepara la piscina para un baño nocturno	Prepara la piscina per a un bany nocturn
Very Difficult	It's 10 PM but I fancy a quick swim	Son las 10 de la noche pero me apetece un baño rápido	Són les 10 de la nit però em ve de gust un bany ràpid
Absurd	Swan swim swing	Cisne nadar columpio	Cigne nedar gronxador
Simple	Switch on all garden lights	Enciende todas las luces del jardín	Encén tots els llums del jardí
Complex	Prepare the garden for an outdoor dinner	Prepara el jardín para una cena al aire libre	Prepara el jardí per a un sopar a l'aire lliure
Very Difficult	Friday night, garden party	Noche de viernes, fiesta en el jardín	Nit de divendres, festa al jardí
Absurd	Green garden watermelon	Sandía verde de jardín	Síndria verda de jardí
Simple	Turn on the office's air conditioning	Enciende el aire acondicionado de la oficina	Encén l'aire condicionat de l'oficina
Complex	It's too hot here at the office	Hace demasiado calor aquí en la oficina	Fa massa calor aquí a l'oficina
Very Difficult	I can't concentrate on my work because it's too hot in here	No puedo concentrarme en mi trabajo porque hace demasiado calor aquí	No puc concentrar-me en la meva feina perquè fa massa calor aquí
Absurd	Air blue laptop thing	Cosa de portátil azul de aire	Cosa de portàtil blau d'aire
Simple	Shut all house blinds	Cierra todas las persianas de la casa	Tanca totes les persianes de la casa
Complex	We're leaving for a few days, close everything	Nos vamos por unos días, cierra todo	Ens n'anem per uns dies, tanca-ho tot
Very Difficult	We left town yesterday; I don't think we secured everything	Nos fuimos de la ciudad ayer; no creo que hayamos asegurado todo	Vam marxar de la ciutat ahir; no crec que ho hàgim assegurat tot
Absurd	Bye all means things stuff	Adiós todas las cosas medios	Adéu a totes les coses mitjans

We did not define the correct API call for each command, as interpretations may vary. For example, with the command "*It's 10 PM but I fancy a quick swim*," one might expect the pool lights to be turned on, but should the filtration pump be switched off? What about the garden lights? In many of these sentences (Complex & Very Difficult), there is no single correct answer.

A.2.2 Procedure

The evaluation process involved the following steps for each of the 25 natural language prompts:

1. **Provide System Context**: Supplied the same system message outlining the available API endpoints, parameters, and required output formats to each model.
2. **Submit User Query**: Entered the user command into the model and recorded the generated API call(s).
3. **Score Correctness**: Manually assessed the accuracy of each generated API call on a scale from 1 (completely incorrect) to 5 (perfect API call), based on predefined criteria.
4. **Measure Latency**: Recorded the time taken from submitting the prompt to receiving the response.

A.2.3 Analyzing Cost

On-premises models require a substantial setup with significant GPU compute power. Factoring a portion of the cost into our system would surpass any calculation from our cloud-based competitors. Such a comparison would make sense in the scope of an enterprise-class application where (1) data confidentiality matters and, (2) cost analytics have proper accounting support.

Our experimental hardware, the NVIDIA Jetson ORIN NX 16GB, has a cost of approximately $1,000. When examining the results in the next section, it is evident that this hardware is not sufficiently powerful to complete the task in a timely manner for a real-time API. Although spreading the cost across a four-year horizon and neglecting electricity costs, even a simple hour per month on maintenance would surpass the costs of cloud-based models for our application.

For estimating cloud-based model costs, we introduced the concepts of tokens and free tiers. Tokens represent chunks of text processed by the models, with each API call consuming a certain number of tokens based on the input and output length. Both GPT-4o and Claude 3.5 Sonnet offer free tiers with limited token allocations, allowing for initial testing without significant costs. Obviously, high-frequency applications will eventually outweigh the on-premises costs.

Cloud-based service cost estimations require attention to two main factors: (1) the number of tokens used for prompts, context, and responses, and (2) the number of expected daily requests. With the actual prices from both providers (Anthropic & OpenAI), and assuming the number of tokens measured and fewer than 100 requests per day, our service cost would be contained within a few cents (<$0.03) per day or a few dollars (<$10) per month. Based on these numbers, a tenfold increase in usage would be necessary to reach the cost breakpoint against the on-premises cost estimation.

A.3 Results

Table 4: Per-Model Aggregate Results

Model	**Avg. Score (1–5)**	**Avg. Latency (s)**
LLaMA-3.2-3B-Instruct-Q4_0_8_8	2.6	5.8
Mistral-NeMo-Instruct-2407-Q4_0_8_8	2.3	8.2
GPT-4o	4.7	1.5
Claude 3.5 Sonnet	4.4	2.8

Table 4 presents a significant advantage for the cloud-based services analyzed, both in terms of latency and accuracy. In fairness, the accuracy differences were anticipated. Cloud-based services not only run state-of-the-art models but also feature higher numbers of parameters. Comparing a 450B-parameter model against a 3B-parameter model is inherently unbalanced. However, our experiment also measured the extent to which the task was straightforward enough for a small model to handle successfully.

Results showed that both small on-premises models struggled with anything beyond straightforward commands, such as "*Open the garage door*." Both performed better in English and failed in Catalan, as anticipated based on the training documentation. In contrast, cloud-based services demonstrated robust multilingual support and a remarkable ability to handle complex scenarios.

Regarding latency, our modest Jetson edge device confirmed its capability to handle small-sized models in the 1B-8B range when using the right quantization levels due to memory constraints. Latency could be matched with some optimizations, but stronger compute power would definitively improve performance.

Finally, considering the SmartHome use case with low usage frequency, the cost comparison favors cloud-based services. Overall, our cloud-based services achieved an easy win for our use case. An on-premises model might be the best choice in scenarios where:

- The number of requests is significant enough to surpass the cost breakpoint, or
- There is sensitivity around confidential information, or
- Internet dependency needs to be avoided, or
- The user has the resources to manage and amortize a powerful environment.

Among the cloud-based contenders, OpenAI's model outperformed Anthropic's. The differences were specific and are likely to change with each new model release.

A.3.1 Conclusion

Based on the pre-selection research and subsequent experimental analysis, we concluded that OpenAI's cloud-based service with **GPT-4o** was the best choice for our problem. However, having a robust alternative is prudent; therefore, we integrated Anthropic's model into our workflows as a secondary option to be used in the event of an outage or if a new release offers advantages over the primary choice.

It is important to note that there are many other alternatives that could influence the results, particularly for an enterprise-class application. Our research utilized base pretrained models and identified several avenues for future investigation to complement the model with advanced pipelining in multi-step reasoning as described in the future work section.