SAPIENZA
UNIVERSITÀ DI ROMA

A multilingual speech-driven semantic input module for indoor navigation systems

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica

Corso di Laurea in Applied Computer Science and Artificial Intelligence

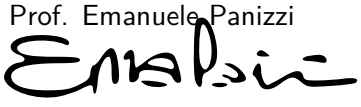
Candidate

GOKHAN CEYLAN

ID number 2124426

Internship Supervisor

Prof. Emanuele Panizzi



Academic Year 2025/2026

A multilingual speech-driven semantic input module for indoor navigation systems

Bachelor's thesis. Sapienza – University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Author's email: ceylan.2124426@studenti.uniroma1.it

Abstract

Finding a room in an unfamiliar university building is a deceptively difficult problem. Room codes, unofficial names, and occupant lists are rarely consistent, and users often remember only a rough description rather than the exact label printed on a door, the way they would describe it to another person rather than search a database. A voice interface is a natural fit for this kind of question, but the voice assistants built into most phones are designed to resolve globally indexed destinations, not a building's informal, locally used room names, and they depend on cloud connectivity and a single language, neither of which can be guaranteed inside a multilingual university building.

This thesis describes the design and implementation of a speech-driven input module for an indoor navigation application developed at Sapienza. The module accepts spoken queries in Italian, English, or a natural mix of both, and maps them, through a custom natural language pipeline, to a structured room database. To our knowledge, no existing indoor navigation system combines fully offline, bilingual voice input with phonetic and edit-distance matching against a curated room database.

The preprocessing pipeline first normalizes textual variation such as accents and spacing, then corrects for the noise that comes with real speech input: filler words, misheard phonemes, and spoken numbers. A user saying "duecento diciotto" should reach room 218, and a user asking for a professor by surname should reach the right office, even when the speech recognizer returns an imperfect transcript.

Room resolution is handled by a hybrid similarity metric that combines edit-distance scoring with phonetic key comparison, augmented by a confidence-and-margin decision gate that either commits to a single result or returns a ranked shortlist for the user to confirm.

The system was evaluated through timing measurements on-device and field testing across all four campus buildings using 150 spoken queries, on an application that runs natively on both Android and iOS. It resolved 92% of spoken queries to the correct room on the first attempt, correctly recognized genuine ambiguity rather than guessing in a further 6% of cases, and failed outright in only 2%, completing the full resolution process in roughly 210 milliseconds even for the most exhaustive searches.

Contents

Chapter 1

Introduction

Navigating an unfamiliar university building using only a mobile screen means frequent stops to check the display, and if the floor plan uses non-obvious room numbering, matching what is on screen to what is on the door becomes an extra step in itself. A voice interface seems like the natural fix: say where you want to go, and let the system handle the rest.

Building one that actually works in practice is more involved than it first appears. Indoor navigation already differs from outdoor routing in fundamental ways. There is no GPS signal, room names are often unofficial or inconsistent, the exact identifier printed on a door does not always match what a user remembers or was told, and the app cannot assume a network connection inside a building. Layering speech recognition on top of this introduces further complications of its own: transcripts are noisy, and users speak differently depending on their native language.

The work presented here covers the design and implementation of the speech input module for an indoor navigation application developed and field-tested across four university buildings at Sapienza. The module accepts spoken queries in Italian or English, such as "take me to Aula T01" or "portami al laboratorio di informatica," and resolves them to a specific room in the building database, entirely on-device. Resolving these bilingual, often imprecisely worded queries against a local room database, without depending on cloud infrastructure, is the central problem the following chapters address.

The broader application, NavIndoor, is developed collaboratively by a fourteen-member team in the Gamification Lab at Sapienza, under the supervision of Prof. Emanuele Panizzi. The work presented in this thesis is focused on the design and implementation of its voice input module.

1.1 More Than Two Speech Recognizers

The system targets two languages: Italian and English. This reflects the real user base at Sapienza, where both languages appear regularly in academic settings and where international students or visitors may not speak Italian at all.

Supporting two languages goes beyond running two speech recognizers. The same room can be described in very different ways depending on the speaker's background. A room labeled "Aula T01" might be spoken as "aula ti zero uno" by an Italian speaker, or as "aola tee zero one" by an English speaker unfamiliar with Italian pronunciation. Both transcripts need to reach the same room in the database. The preprocessing pipeline therefore has to handle phonetic variation across languages, not just within one.

The speech recognizer is initialized with a locale that matches the selected language. Italian and English locales are resolved at startup from the list available on the device, since not all devices expose the same locale identifiers. When no network connection is available, the system uses on-device inference regardless of the resolved locale, which removes the language accuracy guarantee and puts more pressure on the matching logic.

Language also shapes how the preprocessing pipeline works. The ontology substitution rules, filler word lists, and spoken number parser all have separate Italian and English branches. A word like "lab" maps to "laboratorio" in Italian mode and to "laboratory" in English mode. Spoken numbers follow entirely different patterns: "duecento diciotto" in Italian and "two hundred and eighteen" in English both need to resolve to room 218. Handling this correctly is not optional: a room number is one of the most basic ways a user can refer to a destination, and failing on it would undermine the system's core purpose in either language.

1.2 Objectives and Constraints

The module had one central goal: take a spoken query, however imprecisely phrased, and reliably return the correct room from the building database, in either Italian or English. Everything else in the design, including the decision to run entirely offline, follows from that.

Because no structured indoor room database existed for the target buildings at Sapienza, the building repository was assembled from scratch through direct field visits: each of the four buildings was visited in person, room labels were photographed on-site, and entries were added to the JSON dataset manually. Room names were recorded both in their official form and in the terms students actually use, since those two do not always match. A space officially listed under an administrative code may be known among students by an entirely different label. This gap between official nomenclature and everyday reference is one of the concrete reasons the preprocessing pipeline requires a domain-specific ontology rather than exact string matching.

Three constraints shaped how the module was built.

The first is handling ambiguity correctly. Speech recognition transcripts are noisy and sometimes genuinely ambiguous. For a navigation system specifically, silently committing to a low-confidence match risks sending the user to the wrong room entirely, an outcome this thesis treats as worse than asking the user to confirm; the module therefore distinguishes between cases where it is confident enough to resolve to a single room and cases where it surfaces a short candidate list for the user to select from. Getting this threshold right is one of the central design problems the thesis addresses.

The second is offline operation. Indoor spaces, particularly basements and deep corridors, frequently have unreliable or absent network coverage. The system uses the native speech recognition APIs provided by the operating system, which support on-device inference, and performs all matching logic locally against a bundled JSON repository. No external service is contacted at runtime.

The third is computational cost and hardware constraints. The module runs inside a Flutter mobile application and shares execution resources with the map rendering and UI layers. The app already bundles building data, floor plan geometries, and offline map tiles. Embedding dedicated offline speech models would dramatically inflate the application size and introduce severe RAM overhead during transcription. These hardware costs were judged disproportionate to the marginal accuracy gained, a tradeoff quantified in Chapter 2, given that both iOS and Android already ship

highly optimized native on-device speech recognition APIs.

1.3 Thesis Structure

The remainder of the thesis is structured as follows.

Chapter 2 surveys related work on voice interfaces in mobile applications, on-device versus cloud speech recognition, string similarity and phonetic matching methods, and existing systems that have applied voice input to indoor navigation.

Chapter 3 describes the overall system architecture, covering the offline design of the application, the structure of the building JSON repository and its dataset, the three main components of the voice module, the end-to-end execution pipeline connecting them, and the integration of the native OS speech recognition APIs on Android and iOS, including microphone permission handling and language locale resolution.

Chapter 4 describes the natural language preprocessing pipeline: text normalization, diacritic removal, domain-specific vocabulary substitution, filler word filtering, and a parser for spoken number sequences in both Italian and English.

Chapter 5 presents the semantic resolution engine. It explains how phonetic keys are generated for approximate phoneme matching, how Levenshtein distance is applied to score room name and occupant name candidates, how the decision gate determines whether to commit to a single result or surface a short candidate list for disambiguation, and closes with a formal complexity analysis of the full pipeline.

Chapter 6 evaluates the system across two dimensions: execution time and match accuracy. It includes timing measurements of the full resolution pipeline and an analysis of representative test queries, edge cases, and error categories, and a comparison against a baseline exact-match approach to illustrate the necessity of the preprocessing pipeline.

Chapter 7 summarizes the thesis's contributions, discusses the limitations of the current implementation, outlines directions for future work, and closes with final remarks.

Chapter 2

State of the Art and Related Work

2.1 Voice Interfaces in Mobile Applications

Voice interfaces in mobile applications have been a subject of active development for over a decade. Commercial assistants such as Siri, Google Assistant, and Samsung Bixby demonstrated that speech could replace touch for a broad class of tasks on handheld devices, from setting reminders to querying information. These systems rely on cloud-based speech recognition and large language models to interpret open-domain queries, which gives them broad coverage but also makes them entirely dependent on a stable network connection.

In the context of navigation, voice input has been integrated into turn-by-turn routing applications such as Google Maps and Apple Maps, typically through a general-purpose cloud assistant: the user speaks a destination, the request is parsed by the assistant, and a resolved address or business name is handed off to the routing engine. This works because the destinations are globally indexed and correctly named, but it depends on a live connection to cloud infrastructure and does not extend to small, informally named, locally curated destinations such as a university office or a lecture hall.

Indoor navigation introduces constraints that these systems were not designed to handle. GPS is unavailable, destinations are described by room names or occupant names rather than addresses, and users are frequently in areas with poor connectivity. Applying a cloud-dependent voice assistant to this setting does not work reliably. The system described in this thesis takes a different approach: domain-specific speech input, resolved entirely on the device against a structured local database, with no dependency on external services.

Research on voice-driven indoor navigation remains limited compared to the broader Voice User Interface (VUI) literature [1]. Most existing indoor navigation prototypes focus on positioning accuracy and routing, treating user input as a solved problem handled by text search. The challenge of mapping noisy spoken queries to indoor room identifiers in a multilingual setting, without cloud resources, has received less attention.

2.2 On-Device vs. Cloud Speech Recognition

Most speech recognition systems in production today are cloud-based. The device captures audio and sends it to a remote server, where a large model per-

forms the transcription and returns the result. This architecture makes sense for general-purpose assistants: server-side models are trained on vast datasets, updated continuously, and can handle a wide range of accents, languages, and speaking styles with high accuracy.

The tradeoff is a hard dependency on network connectivity. In many indoor spaces, particularly in older university buildings with thick walls, basement floors, or densely occupied areas, both Wi-Fi and cellular signals are unreliable. A speech recognition request that depends on a round trip to a remote server will either fail or stall in exactly the environments where an indoor navigation app is most needed.

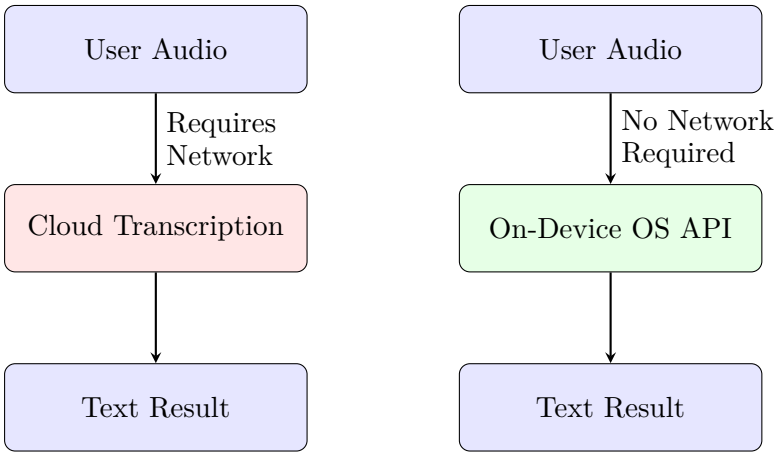


Figure 2.1. Architectural comparison between Cloud-dependent and On-Device speech transcription pipelines.

Figure 2.1 contrasts these two architectures directly. On-device speech recognition avoids this dependency by running the transcription model locally. Both iOS and Android expose native APIs for this: the Apple Speech Framework on iOS [2] and the Android SpeechRecognizer API on Android [3], both of which support on-device inference modes. The system described in this thesis uses these APIs through the `speech_to_text` Flutter package [4], which abstracts the platform differences and provides a unified interface.

An alternative to native OS APIs is to embed a dedicated speech model directly in the application, such as OpenAI's Whisper [5]. However, the native OS APIs already perform adequately for the domain-specific vocabulary handled by this system, meaning the marginal accuracy gained from a bundled model does not justify its hardware costs. Specifically, the processing overhead of third-party acoustic models introduces severe memory pressure on the shared Flutter process: tools like Vosk [6] demand up to 300MB of RAM at runtime, posing a direct crash risk for mobile applications. Furthermore, storage requirements are prohibitive. Achieving acceptable bilingual accuracy with Whisper requires the "base" model (approximately 142MB), while Vosk requires bundling separate acoustic models per language (nearly 100MB total). For an application with a 200MB footprint, adding either model would grow the binary by roughly 45 to 70 percent, a substantial increase for a marginal accuracy gain, while also risking RAM exhaustion, making the tradeoff structurally unjustifiable.

The limitation of on-device recognition is accuracy. Local models are smaller and more constrained than their cloud counterparts, which means they are more likely to produce imperfect transcripts, particularly for domain-specific vocabulary like room codes and department names. This is a known tradeoff and it is one of

the reasons the preprocessing and matching pipeline described in Chapters 4 and 5 has to be robust to noisy input rather than assuming a clean transcript.

2.3 String Similarity and Phonetic Matching in NLP

String similarity measures are a standard tool in natural language processing for comparing sequences of characters that may differ due to typos, misspellings, or transcription errors [7]. In the context of speech-to-text output, where transcripts are noisy by nature, these measures serve as the foundation for matching a spoken query to an entry in a structured database [8].

The most widely studied edit-distance metric is the Levenshtein distance [9], which counts the minimum number of single-character insertions, deletions, or substitutions needed to transform one string into another. It has been applied in spell checkers, search engines, and record linkage tasks. Jaro-Winkler distance [10] is a related measure that gives additional weight to common prefixes, making it well suited for short strings and personal names. This system relies on Levenshtein distance instead, since it needs a single, consistent token-scoring function that applies uniformly to both multi-word room names and personal names; Jaro-Winkler's prefix weighting is tuned specifically for short strings like surnames and does not extend as naturally to general room-name tokens. The phonetic key described in Chapter 5, rather than a prefix-weighted distance metric, is what specifically absorbs sound-alike variation for both cases, as illustrated in Table 2.1.

Spoken Input	Target Room	Phonetic Key	Edit Distance
Aola	Aula	l	1
Labratorio	Laboratorio	lbrtr	1
Bibiloteca	Biblioteca	bbltc	2
Semianrio	Seminario	smnr	2

Table 2.1. Examples of noisy STT input where edit distance is elevated, but the phonetic key matches perfectly.

Phonetic algorithms take a different approach. Rather than measuring character-level edits, they encode words according to how they sound, so that different spellings of the same pronunciation map to the same code. Soundex [11] and Metaphone [12] are the most established examples, both developed primarily for English surname matching. They work well within their original scope but do not generalize cleanly to other languages. Italian phoneme patterns, for instance, are handled poorly by both: the rules built into these encoders for combinations such as *chi*, *gn*, and *sc* are tuned to English pronunciation patterns and do not transfer to how these combinations function in Italian.

The system described in this thesis uses a hybrid approach. A custom phonetic key function strips vowels, silent letters (like *h*), and repeated consecutive consonants, producing a compact consonant skeleton that can be compared across variant spellings. This key is used alongside Levenshtein distance at the token level: each word in the query is compared against each word in a candidate room name, and the scores are combined. The rationale for building a custom encoder rather than adopting Soundex or Metaphone directly is that those algorithms are heavily biased toward English phonetics, whereas a minimalist consonant-skeleton approach proved highly effective and language-agnostic for resolving both Italian and English room queries.

2.4 Voice Input in Indoor Navigation Systems

A recurring conflation in the domain of accessibility and navigation applications must be resolved before any comparison is meaningful: the distinction between voice input and voice output. The overwhelming majority of "voice-enabled" or "audio" navigation systems use voice exclusively as an output modality, relying on synthesized speech (TTS) and platform screen readers to read the interface aloud. In these systems, destination input remains text-based or selection-based. The marketing term "voice navigation" almost universally refers to spoken turn-by-turn guidance, not spoken query entry. The two are architecturally unrelated, and this distinction is the central axis of this comparison.

2.4.1 Destination Input in Current Systems

Commercial indoor navigation applications largely rely on visual interfaces for input. **GoodMaps Explore** [13], which uses camera-based computer vision for highly accurate indoor positioning, relies on list scrolling and text search for destination selection; hands-on testing conducted for this thesis found this process noticeably slower than simply speaking a destination. Voice features in the app are dedicated to output (turn-by-turn spoken guidance) and full screen-reader compatibility.

Similarly, **Lazarillo** [14], a popular BLE-beacon-based navigation app, advertises a voice-driven search feature alongside its typed category browsing (e.g., health, food, stores). Hands-on testing conducted for this thesis showed that this feature is implemented through the device's own dictation keyboard rather than a dedicated microphone pipeline: the dictated text is simply dropped into the same field used for typed queries. Isolated place names spoken or typed with a minor misspelling still resolve correctly (a misspelled "Vaticano" still returns Vatican City), but natural phrasing such as "I want to go to" or "where is the" passes through unmodified and consistently fails, since no filler-word removal is applied before the search runs. The app also requires an active internet connection, returning no results when offline. **Navigine** [15], an enterprise indoor-positioning SDK, provides programmatic point-of-interest (POI) search but leaves the UI implementation to the developer, explicitly framing voice as a separate layer that a developer would need to build on top of their platform.

In the open-source landscape, the most architecturally relevant system is **Soundscape** [16] (relaunched by the Scottish Tech Army in 2023 as a successor to the original Microsoft Research app, which was discontinued in 2022). While Soundscape utilizes Levenshtein-family matching for location search, it is strictly bound to typed input and requires an active connection to function:

- **Text Search:** Search applies fuzzy string matching based on Damerau-Levenshtein distance, a variant of Levenshtein distance that additionally treats an adjacent-character transposition as a single edit, over geographic OSM-derived data (roads, POIs, buildings) rather than a curated indoor room database. Hands-on testing showed the app returns no results in airplane mode, and its fuzzy tolerance is narrow in practice: a misspelling such as "Vatikano" fails to resolve, while the closer "Vatikan" succeeds.
- **Lack of Semantic Voice Input:** As with Lazarillo, any spoken input is entered through the device's own dictation keyboard rather than a dedicated voice pipeline, and the resulting text is searched with no preprocessing; a natural spoken query such as "where is the Vatican City" fails to resolve.

Soundscape relies on device-level accessibility layers (like VoiceOver or TalkBack) and physical touch gestures for user interaction rather than an internal, free-form voice resolution engine, and it cannot process an arbitrary, noisy spoken phrase (e.g., "take me to the lab") and map it to a semantic destination identifier.

2.4.2 The Architectural Gap

Where spoken input touches these systems at all, it falls into one of two architectures: platform-delegated general assistants (which are cloud-dependent and not coupled to local indoor databases) or fixed-grammar command-and-control systems.

The specific combination of constraints addressed in this thesis is not occupied by any surveyed open-source or commercial system. What is currently absent from the state of the art is a lightweight, fully offline voice-to-intent module that:

1. Takes free-form spoken destination queries rather than a fixed command grammar.
2. Uses native-OS Speech-to-Text as the front end.
3. Applies a custom Levenshtein and phonetic matching stage specifically designed to absorb STT transcription noise.
4. Resolves against a local, structured indoor room database.
5. Operates bilingually without cloud connectivity.

While individual ingredients have precedent (Soundscape uses Damerau-Levenshtein for typed, online-only geographic search, and both Soundscape and Lazarillo delegate spoken input to the device's generic dictation keyboard rather than a dedicated pipeline), the novelty of the system described in this thesis lies in its architectural composition. By fusing phonetic edit-distance matching with spoken input, the module specifically maps noisy STT transcripts to indoor room identifiers, entirely offline and bilingually.

Chapter 3

System Architecture and Data Modeling

3.1 Offline System Architecture

A core constraint of indoor navigation is that network connectivity cannot be assumed. University buildings, particularly those with deep basement floors or dense concrete construction, frequently block both Wi-Fi and cellular signals. The system is therefore designed to function entirely without a network connection at runtime.

This requirement shapes the architecture at every level. Map tiles are bundled with the application as offline assets. Building data, room information, floor plan geometries, and routing graphs are all loaded from local JSON files at startup. Speech recognition runs through the native OS APIs rather than any cloud transcription service: the Apple Speech Framework on iOS and the Android SpeechRecognizer API on Android. Before each listening session, the voice module checks for network availability to decide whether to use enhanced online recognition or fall back to fully on-device inference; beyond that check, the voice module itself makes no network requests during normal operation.

The original design included a `SemanticIsolate` wrapper class with an async interface intended to offload processing to a background thread, but Dart isolates are not supported on Flutter Web [17]. The current implementation keeps the async interface for potential future extensibility while resolving synchronously on the main thread. The synchronous approach keeps the implementation simple, since resolution runs against a small, bounded local dataset rather than an external service call of unpredictable latency. The actual measured cost of this tradeoff, and why it remains an acceptable design choice despite exceeding a single frame's budget, is discussed in Chapter 6.

3.2 System Components and Data Flow

The voice module is structured as a three-layer Dart pipeline contained in the `lib/voice` directory, separate from the application's user interface code. The three layers are the building data repository, the `VoiceManager` state layer, and the algorithmic resolution engine.

3.2.1 The Building JSON Repository

At the base of the pipeline is a statically bundled JSON repository containing data about the campus buildings, organized by floor level and room type. Each entry describes a single physical location, including its Italian and English names, room number identifiers, associated occupants, and internal building codes. Before the voice module initializes, the repository is loaded into memory and parsed into typed `Room` objects.

Table 3.1 lists the fields of a single room entry as stored in the JSON files and parsed by the `Room.fromJson()` factory method.

JSON Key	Type	Description
`id`	Integer	Unique numeric identifier for the room entry.
`floor`	Integer	Floor level (negative values indicate basement floors).
`number`	List	Room number strings; a room may carry multiple identifiers (e.g., "06" and "-106").
`official_name`	String	Optional official designation; often empty.
`name_it`	String	Full Italian name (e.g., "Laboratorio di Informatica").
`name_en`	String	English name (e.g., "Computer Science Lab").
`type`	String	Functional category (e.g., `classroom`, `office`, `laboratory`, `storage_room`).
`notes_it`	String	Optional Italian notes; often empty.
`notes_en`	String	Optional English notes; often empty.
`occupants`	List	Staff names associated with the room. Empty if no permanent occupant.
`room_code`	String	Internal building code; may be empty.
Note: the `building` field is not stored in the JSON; it is injected at load time.		

Table 3.1. Fields of a room entry in the JSON building repository.

The `official_name` attribute is parsed during deserialization but is not evaluated by the active similarity resolver loop.

3.2.2 The VoiceManager (State Layer)

The `VoiceManager` is the entry point for all speech interactions, implemented as a singleton. It handles microphone permission negotiation and binds a listener to the operating system speech engine. As the transcript arrives, the manager passes it to the algorithmic layer together with the active language locale, which is needed for language-specific preprocessing.

3.2.3 The DataPreprocessing and CoreAlgorithm Layers

The transcript first passes through `data_preprocessing.dart`, which removes diacritics, applies vocabulary substitutions using word-boundary regular expressions, and attempts to extract a room identifier from spoken number sequences (for example, converting "duecento dieci" to the string "210").

The cleaned transcript then enters `core_algorithm.dart`, which compares it against every `Room` object in memory using the similarity scoring described in Chapter 5. The output is a `SemanticResult` that tells the UI layer what to do: navigate to a resolved room, present a short candidate list for the user to choose from, or report that the query could not be matched.

3.3 End-to-End System Execution Pipeline

This section details the runtime data transformations of a single voice query through the three-layer pipeline.

Stage 1: Speech Capture. The user activates the voice input interface by pressing the dedicated button in the navigation UI. This action dispatches a call to the `VoiceManager` singleton. On Android, the manager checks the microphone permission status via the `permission_handler` package and, if it has not been granted, requests it and waits for the user's response before proceeding. On iOS, the manager instead checks the combined microphone and speech-recognition permission status exposed by the `speech_to_text` plugin; if it is not already granted, voice input is reported as unavailable rather than triggering an inline request. In both cases, once permission is confirmed, the speech engine is initialized immediately.

Stage 2: OS Speech Recognition. Once the microphone is active, the native OS speech engine begins transcribing the audio stream in real time. On iOS, the `SFSpeechRecognizer` class processes the microphone input using on-device acoustic and language models. On Android, the Android `SpeechRecognizer` API performs the same function. The `speech_to_text` plugin continuously receives partial transcription results from the OS and forwards them as a stream of strings to the `VoiceManager`.

Stage 3: Query Dispatch. As partial transcripts arrive from the speech engine, the `VoiceManager` immediately packages each one together with the current language mode (*it* for Italian, *en* for English, selected by the user in the application settings) and passes it synchronously to the `SemanticResolver`, so the resolved result is continuously refreshed while the user is still speaking. When the user releases the button or the speech engine detects a pause, listening stops and whichever result was most recently computed becomes the final outcome shown to the user.

Stage 4: Normalization and Preprocessing. The `SemanticResolver` first forwards the raw transcript to `DataPreprocessing.normalize()`, which converts the string to lowercase, removes diacritical marks, applies the language-specific ontology corrections, and strips filler words. The normalized query is then passed to `extractRoomNumber()`, which evaluates the query against a short-circuiting cascade of regular expression patterns and language-specific number matchers, returning immediately upon resolving a valid room identifier.

Stage 5: Semantic Resolution. The preprocessed query is evaluated against the in-memory building repository. If a room number was extracted, the resolver applies `normalizeNumber()` across room number lists and additionally checks for exact name string equivalents and room-prefix substrings, returning immediately if a unique structural match is identified. If no number was found, or if the number lookup produced multiple candidates, the resolver first checks for global entry

substring containment, assigning a direct score of 0.90 upon a macro match. If this shortcut does not resolve the query, the resolver falls back to an iterative token-level fuzzy scoring pass across all room entries in the building repository. This pass applies the phonetic key bonus and computes token-level similarity scores across bilingual room name records and occupant arrays, using a shared English-normalized string comparison baseline.

Stage 6: Decision and Output. The top-scoring candidate is evaluated against the `STRONG_THRESHOLD` and `MARGIN` constants. Depending on the outcome, the resolver returns one of three possible `SemanticResult` outcomes: a confirmed navigation target where the top score meets or exceeds 0.85 and leads the second candidate by at least 0.10, an ambiguous result carrying up to three candidate rooms for the user to confirm, or an `UNKNOWN` result indicating that no reliable match was found. The UI layer renders the appropriate response: it either launches the navigation route, presents a disambiguation list, or falls back to the standard keyboard search view with the recognized (but unmatched) text already entered, letting the user refine it manually.

3.4 Building Repository Dataset

The resolution accuracy of the voice module is directly dependent on the quality and coverage of the building repository dataset. This section describes the structure and scope of the dataset used in this thesis.

The repository covers four buildings on the Sapienza University campus. Building 1, Ex Vetrerie Sciarra (building code RM103), is a repurposed industrial facility housing lecture halls, laboratories, and staff offices across five floors, including a basement level. With 82 room entries, Edificio Marco Polo (code RM021) is the largest building in the dataset; it contains the main lecture halls, seminar rooms, and faculty offices of its administrative unit. The third building, Facoltà di Lettere e Filosofia (code CU003), serves primarily as a teaching space for the humanities faculty, with classrooms, research areas, and departmental offices. Villa Mirafiori (code RM052) is the only building in the dataset occupying a historical villa structure; its rooms are predominantly philosophy department offices, seminar spaces, and archival rooms.

In total, the repository contains 243 distinct room entries across the four buildings. Each entry is fully bilingual, providing both an Italian name and an English equivalent. Room numbers are stored as arrays to accommodate rooms known by multiple identifiers: for example, a basement room may carry both a positive identifier (“06”) and a floor-prefixed identifier (“-106”).

The occupant metadata field lists academic staff assigned to each room, used by the resolver to support queries where a user asks to navigate to a specific professor’s office by name. In the current dataset, 116 of the 243 room entries include at least one occupant. Table 3.2 summarizes the composition of the dataset by building.

Building	Code	Rooms	With Occupants
Ex Vetrerie Sciarra	RM103	47	19
Edificio Marco Polo	RM021	82	43
Facoltà di Lettere e Filosofia	CU003	48	9
Villa Mirafiori	RM052	66	45
Total	–	**243**	**116**

Table 3.2. Composition of the building repository dataset.

At 243 entries, the dataset represents a realistic mid-sized campus deployment. Execution timing for the full fuzzy scoring pass is reported in Chapter 6.

3.5 Native Speech Recognition Integration

The voice module binds to the speech recognition engines embedded in the host operating system. On iOS, the system uses `SFSpeechRecognizer` from the Apple Speech Framework. On Android, it interfaces with the Android `SpeechRecognizer` API. Both engines support on-device inference and multiple language locales, satisfying the offline requirement described in Section 3.1 and the bilingual requirement described in Section 1.1.

To connect the Flutter application layer to these platform-specific components, the system uses the `speech_to_text` plugin. The plugin translates Dart method calls such as `listen()` and `stop()` into the corresponding Swift code on iOS and Kotlin code on Android, allowing the `VoiceManager` to operate through a single unified interface regardless of the underlying platform.

Microphone access is a restricted operation on both platforms, though the exact negotiation mechanism differs by platform. On iOS, two separate grants are required: one for microphone hardware access and one for the speech recognition engine itself, checked together through the `speech_to_text` plugin's combined permission check. On Android, only the microphone permission is required, as speech recognition is part of the OS service, and the `VoiceManager` checks and requests it directly through the `permission_handler` package, triggering the OS's native authorization dialog if it has not yet been granted. If the user denies access, on either platform, `startListening()` returns a dedicated result value that the calling screen uses to present a dialog directing the user to the system settings to re-enable access manually.

Before accepting input, the `VoiceManager` resolves the correct locale for the active language. The `speech_to_text` plugin exposes the full list of locales available on the device at runtime. For Italian, the manager selects any device locale beginning with `it`; for English, it evaluates a priority-ordered list of English variants (`en-US`, `en-GB`, `en-AU`, `en-IN`, `en-CA`) before accepting any locale beginning with `en`. If no device locale satisfies either condition, the system falls back to a hardcoded identifier (`it-IT` or `en-US`) rather than leaving the locale unresolved. When the device is offline, on-device inference is used regardless of the resolved locale, which removes the language guarantee and places greater load on the preprocessing and matching pipeline to handle a wider range of transcript quality.

Chapter 4

Natural Language Preprocessing

4.1 Text Normalization and Filler Removal

The raw transcript produced by the OS speech engine arrives as an unstructured string. The static `normalize()` method of the `DataPreprocessing` class processes it through three sequential operations: character-level normalization, vocabulary substitution, and filler word removal. The first and third are described in this section; vocabulary substitution is covered in the following section.

Character-level normalization converts the input to lowercase, trims leading and trailing whitespace, converts hyphens to spaces, removes punctuation outside the word and whitespace character classes, and collapses consecutive whitespace to a single space. Hyphens are treated as a conversion rather than a deletion because room identifiers such as "T-01" would otherwise lose their letter-digit structure. Diacritical marks are stripped using the `removeDiacritics()` function from the `diacritic` Dart package, covering the accented vowels common in Italian text. The result is a compact, ASCII-safe lowercase string.

Filler word removal strips terms that carry no semantic weight for room matching. The list covers navigational prepositions (*al*, *alla*, *allo*, *in*, *da*, *di*), instructional phrases (*portami*, *voglio*, *andare*, *cerco*, *dove*, *take me to*, *go to*, *i want to*, *looking for*, *where is*), role titles (*prof*, *professore*, *professor*), and location category labels (*aula*, *stanza*, *room*, *ufficio*, *office*, *dipartimento*, *department*). Including *aula* in the filler list is intentional: the preceding vocabulary substitution step maps phonetic variants such as *aola* and *laura* to the canonical *aula*, which is then stripped here so that only the room identifier or name remains. Each term is matched using word-boundary regular expressions, ensuring that only isolated whole-word occurrences are removed and that longer room names containing the same substring are not affected.

4.2 Domain-Specific Ontology Mapping

Before filler removal runs, the `normalize()` method applies a layer of domain-specific vocabulary normalization using a static ontology map. This step addresses the mismatch between informal or mis-transcribed terminology and the canonical labels stored in the building repository.

The primary motivation is phonetic ambiguity. In a real indoor acoustic environment, the OS speech engine frequently mis-transcribes domain-specific words due to ambient noise, accent variation, or reverberation. The Italian word *aula* is the clearest example: depending on the speaker and the environment, it may arrive

in the transcript as *aola*, *ola*, *aulo*, *laura*, or *hola*. Since the repository uses the canonical spelling, any of these variants would fail to match without correction.

The `DataPreprocessing` class maintains two language-specific dictionaries, `ontologyIt` and `ontologyEn`, each mapping known mis-transcriptions and informal abbreviations to their canonical equivalents. The Italian dictionary maps nine phonetic variants of *aula* to the correct form, then handles how Italian speakers pronounce letter-suffixed room names: *aula ti* becomes *aula t*, *aula bi* becomes *aula b*, *aula ci* becomes *aula c*, *aula di* becomes *aula d*, and *aula effe* becomes *aula f*. Beyond room naming, it maps common abbreviations: *lab* to *laboratorio*, *info* and *computer* to *informatico*, *biblio* to *biblioteca*, and *ammin* to *amministrazione*. It also corrects administrative aliases such as *segreteria studenti* and *segreteria amministrativa* to their canonical institutional labels.

The English dictionary applies analogous corrections. Letter-suffixed room names follow English phonetics: *aula bee* and *aula be* become *aula b*, *aula cee* becomes *aula c*, *aula dee* becomes *aula d*, *aula ef* and *aula eff* become *aula f*, and the standalone *sea* is mapped to *c* to handle the case where only the letter is spoken. Abbreviations follow English conventions: *lab* maps to *laboratory*, *admin* to *administration*, *tech* to *technical*, *computers* to *it*, and *phd* to *phd students*.

The active dictionary is selected at runtime based on the `langMod` parameter passed into the preprocessing call. Each mapping is applied using word-boundary-anchored regular expressions, ensuring that replacements target only complete tokens and do not corrupt adjacent characters of unrelated terms.

4.3 Parsing Spoken Numbers

A specific challenge in navigating university building complexes is that room identifiers are commonly numeric. Users may speak numbers in several distinct formats: as individual digit sequences (*due uno otto*), as fully composed spoken words (*trecento cinque*), or as alphanumeric composites (*t uno*). The output of the OS speech engine is a plain text string in all cases. The `extractRoomNumber()` function resolves these formats through a cascade of five strategies applied in order of decreasing specificity.

Strategy 1: Direct regular expression matching. Three regular expression patterns are applied in sequence. The first targets a letter separated from a digit sequence by a space, in either order, normalizing forms such as `a 12` to `a12` and `202 a` to `202a`. The second handles the specific case of the letter `t` followed by one or two digits with optional spacing, resolving forms such as `t 01` to `t01`. The third catches any contiguous alphanumeric token such as `202a` or `t1` that already appears in compact form.

Strategy 2: Sequential single digit words. If no alphanumeric pattern is matched, the function scans the token list for consecutive individual digit words. In Italian these are *zero* through *nove*; in English, *zero* through *nine*. Each recognized digit word is appended to an output buffer, and scanning halts at the first non-digit token. If the token immediately preceding the digit sequence is the letter `t`, it is prepended to the result, allowing *t due uno otto* to resolve to `t218`. A trailing single lowercase letter after the digit sequence is also captured, so *due zero due a* resolves to `202a`.

Strategy 3: Full natural language number parsing. If the digit-by-digit strategy fails, the function delegates to a language-specific parser selected by the active language mode. The Italian parser concatenates consecutive number tokens into a single string and resolves it through a hierarchical decomposition of thousands,

hundreds, tens, and units: *trecento cinque* becomes the compound *trecentocinque*, which decomposes to 300 plus 5, yielding 305. The English parser handles additive spoken forms such as *three hundred and five* by treating *hundred* as a multiplicative operator and *and* as a transparent connector. Both parsers carry a leading `t` and a trailing letter through the result, so *t two hundred and two a* correctly resolves to `t202a`.

Strategy 4: Roman numeral matching. If number word parsing returns no result, the function checks whether any token in the query matches a known Roman numeral in lowercase form, covering *i* through *xii* and *xxii* (the standard range for Roman-numeral lecture halls, plus *xxii*, the one higher-numbered room found in the surveyed buildings). This handles buildings that label seminar rooms or floors using Roman numeral conventions.

Strategy 5: Standalone single letter. As a final fallback, the function checks whether the entire remaining query, after all prior normalization and filler removal, consists of exactly one letter character. Because filler terms such as *aula* and *room* have already been stripped at this point, a query originally spoken as *aula a* or *room b* arrives here reduced to a single letter, which this pattern captures directly.

If none of the five strategies produces a result, `extractRoomNumber()` returns `null`, signaling to the resolver that no structured room number could be extracted. The resolver then falls back to a full name-based fuzzy scoring pass across all room entries in the repository.

Chapter 5

The Semantic Resolution Engine

5.1 Phonetic Key Generation

After the preprocessing pipeline has normalized the raw transcript and attempted to extract a structured room number, the resulting query string may still fail to match any room identifier exactly. In natural speech, a user's pronunciation of a room name may be phonetically accurate while still producing a transcript that is spelled differently from the room's official name stored in the JSON repository. The `SimilarityMetrics` class addresses this by computing a phonetic key for any given string before performing character-level comparisons.

The `phoneticKey()` function reduces a word to a compact representation of its consonant sounds by applying a sequence of deterministic substitution rules. The process begins by lowercasing and trimming the input. It then collapses the doubled consonant pair *zz* to a single *z*, and applies a set of Italian phonetic simplifications: the trigraphs *chi*, *che*, *ghi*, and *ghe* (which produce hard k/g sounds in Italian) are folded together with *ci* and *ce* (which produce soft, affricate sounds) into the same k- or g-based consonant form, deliberately collapsing this hard/soft distinction so that both pronunciations map to a single phonetic key; *gl* and *gn* are each collapsed to a single consonant (l and n respectively); *sc* is reduced to s; and the silent letter h is removed. The letter q, always followed by u in Italian, is substituted with k. After these substitutions, any remaining sequence of consecutive duplicate consonants is collapsed to a single character. Finally, all vowels are removed entirely.

The resulting key is a short consonant-only string that functions as a phonetic fingerprint. For example, the correctly spelled Italian word *biblioteca* and a mis-transcribed variant such as *bibliotteca* would both reduce to the same consonant sequence (*bbltc*), allowing the system to recognize them as phonetically equivalent even when their Levenshtein edit distance at the character level is greater than zero.

5.2 Levenshtein Distance and Similarity Scoring

The core string comparison mechanism used by the resolution engine is the Levenshtein distance algorithm, implemented directly in Dart inside the `SimilarityMetrics` class. The algorithm computes the minimum number of single-character edit operations (insertions, deletions, and substitutions) required to transform one string into another.

The implementation uses an optimized single-row dynamic programming approach. Rather than maintaining a full two-dimensional matrix of size $M \times N$ (where M and N are the lengths of the two input strings), the algorithm allocates

a single integer array of length $N + 1$ and iterates through the characters of the first string, updating the array in place. This reduces the memory requirement from $O(M \cdot N)$ to $O(N)$, which is relevant for performance when the function is invoked repeatedly across a large repository of rooms. Table 5.1 shows the resulting dynamic programming matrix for a representative example.

	∅	**A**	**o**	**l**	**a**
∅	**0**	1	2	3	4
A	1	**0**	1	2	3
u	2	1	**1**	2	3
l	3	2	2	**1**	2
a	4	3	3	2	**1**

Table 5.1. Conceptual dynamic programming matrix illustrating the Levenshtein distance between the STT error "Aola" and target "Aula"; the implementation itself stores only a single row at a time, as described above. The bold diagonal path shows a single substitution operation (distance = 1).

The raw distance value is converted into a normalized similarity score using the formula:

$$\text{similarity} = 1.0 - \frac{\text{distance}}{\max(\text{len}(a), \text{len}(b))}$$

This produces a floating-point value between 0.0 (completely different) and 1.0 (identical strings).

5.2.1 Token-Level Scoring

Rather than comparing the query and room name as single flat strings, the resolution engine applies scoring at the token level, where a token is simply a whitespace-separated word within the normalized string, not a subword or vocabulary unit in the sense used by large language models. The `getTokenScore()` function computes the similarity between two individual word tokens. If the two tokens are character-for-character identical, the function returns 1.0 immediately without invoking the Levenshtein computation.

Otherwise, it computes the standard Levenshtein similarity between the tokens, then computes the phonetic key of both and applies a bonus if the keys match or overlap (see Figure 5.1). If the query token's phonetic key is identical to the target token's phonetic key, a bonus of 0.20 is added to the similarity score. If one key is a substring of the other, a smaller bonus of 0.05 is applied. The phonetic bonus is only applied when both keys are non-empty. The final score is capped at 1.0. Like the other hand-tuned similarity constants used throughout this chapter, these two bonus values were arrived at empirically through repeated testing rather than derived analytically; the full methodology is described in Section 5.3.4.

This two-layer scoring approach ensures that a word pronounced accurately, but transcribed with a spelling that differs from its canonical form, still receives a high similarity score.

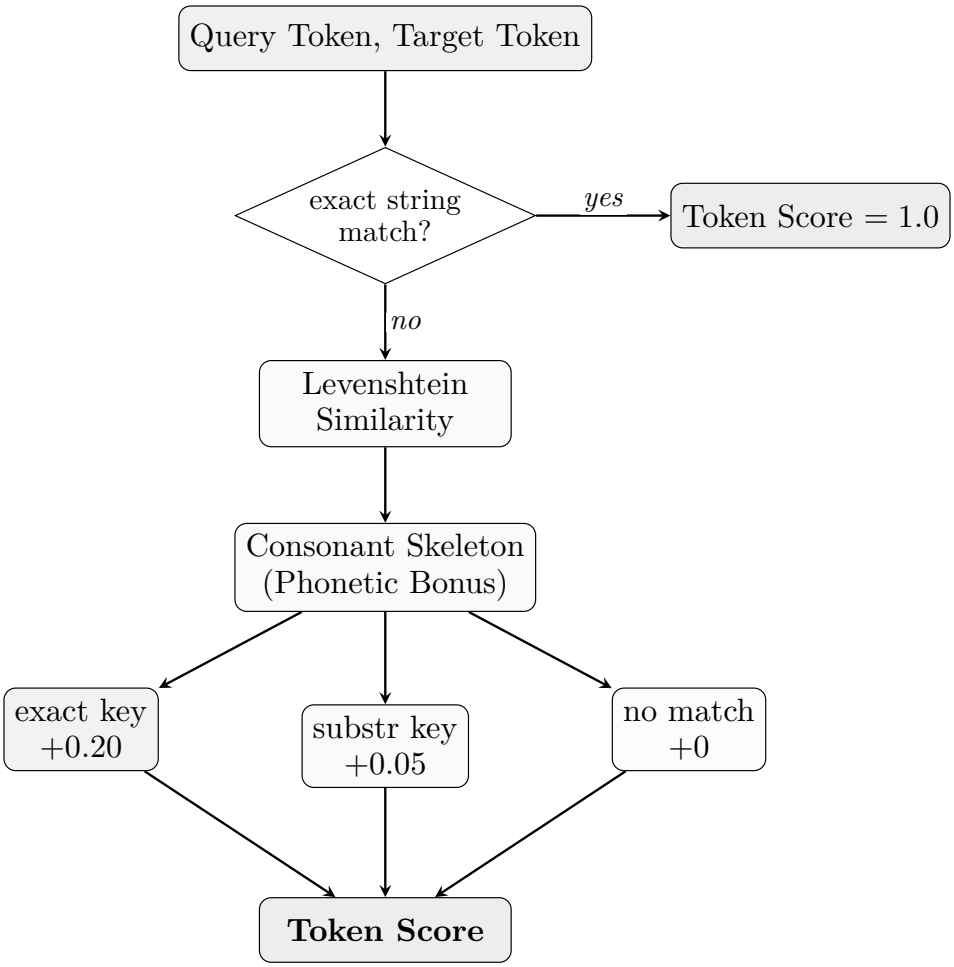


Figure 5.1. Token-level scoring flow used by `getTokenScore()`, showing the exact-match short-circuit, base Levenshtein similarity, and the three possible phonetic-bonus outcomes.

5.2.2 Occupant Name Scoring and Surname Weighting

When a query does not match any room name directly, the resolver scores it against the occupant lists attached to each room entry. This is handled by the `_scoreOccupant()` function, which applies a different scoring strategy from room name matching to account for the structure of personal names.

Both the query and the occupant name are normalized using the active language rules before scoring. The occupant name is split into tokens, with the last token treated as the surname and all preceding tokens treated as first names.

For each query token, the function computes a `getTokenScore()` match against the surname and, independently, the best match against any first name. The two scores are combined using a weighted sum that gives greater importance to the surname: the final score is the surname score multiplied by 0.70 plus the best first name score multiplied by 0.30. Like the other hand-tuned constants in this chapter, this 70/30 split was not derived analytically but arrived at empirically, reflecting the fact that a surname alone is usually enough to identify someone, while a first name alone rarely is; the full tuning methodology is described in Section 5.3.4. If the occupant entry contains no first name tokens, only the surname score is returned.

One additional rule handles the common case where the user speaks only a surname: if the query consists of a single token and that token scores above 0.80 against the surname, the function returns the surname score directly without applying the weighted combination, avoiding an artificial penalty for the absence of a first name match. Like the other hand-tuned similarity constants used throughout this chapter, this threshold was also arrived at empirically rather than derived analytically; the full methodology is described in Section 5.3.4.

5.3 Decision Logic and Confidence Thresholds

The `SemanticResolver` class orchestrates the full resolution process. It accepts the raw transcript string, the active language mode, and an optional current building name as inputs, normalizes the transcript internally, and returns a `SemanticResult` object describing the inferred navigation intent.

The resolver operates in two distinct phases.

5.3.1 Phase 1: Number-Based Matching

If the preprocessing stage successfully extracted a room number from the query, the resolver attempts to resolve it by direct lookup. It searches the in-memory repository for all rooms whose `numbers` field contains a value matching the extracted number after normalization, and also checks for exact name string equivalents and room-prefix substrings as described in Chapter 3.

The resolver then checks the global match list. If exactly one room matches across the entire repository, it is returned with confidence 1.0. If multiple rooms across different buildings share the same number, the resolver returns an ambiguous result with confidence 0.70, containing the full list of candidates and prompting the UI to ask the user to specify the building.

5.3.2 Phase 2: Name-Based Fuzzy Matching

If no room number was extracted, or if the number lookup produced no match, the resolver performs a full fuzzy name search across all rooms. For each room in the repository, the `_scoreRoom()` function is called against both the Italian and English

room names and against the building display name. The highest of these three scores is retained as the room's overall similarity score. A parallel pass scores each room's occupant list using `_scoreOccupant()`, which applies the surname-weighted scoring described in Section 5.2. Rooms and occupants scoring above 0.40 are retained as candidates.

Because a room may score above the threshold on both its name and its occupant list, the combined candidate list is first sorted in descending score order, then deduplicated by room identifier, keeping only the first (highest-scoring) occurrence of each room. Each physical room therefore appears at most once, at its highest achieved score.

5.3.3 Entrance Heuristic Bonuses

During the fuzzy scoring pass, the resolver applies two heuristic score bonuses for entrance-type rooms. A room is classified as an entrance if its `type` field is set to `entrance`, or if its Italian or English name contains the words *ingresso* or *entrance*.

The first bonus applies when the query itself signals an entrance intent. The resolver checks whether the normalized query contains any of the substrings *ingress*, *entrat*, or *entranc*, which cover the Italian and English stems for entrance and entry. If both conditions hold, the room's score receives a bonus of 0.50.

The second bonus applies independently of the query content. If the building name score for a room is at or above 0.80 and the room is classified as an entrance, a bonus of 0.20 is added. This handles queries where a user names a building without specifying a room, making the building entrance the most likely intended destination.

5.3.4 The Decision Gate

After scoring and deduplication, the resolver applies a two-condition gate using two predefined constants. The strong threshold (`STRONG_THRESHOLD = 0.85`) defines the minimum acceptable similarity score for a confident navigation decision. The margin (`MARGIN = 0.10`) defines the minimum required difference between the top candidate's score and the second candidate's score, preventing the system from committing to a result when two rooms are too similar to disambiguate reliably. This same methodology applies to every hand-tuned constant in this chapter: the phonetic-key bonuses (0.20 and 0.05), the surname/first-name weighting split (0.70/0.30), the single-token surname threshold (0.80), the substring-containment shortcut (0.90), the minimum candidate floor (0.40), and the entrance bonuses (0.50 and 0.20) were all set the same way, by repeatedly testing the system with real spoken queries from multiple users across the target buildings and adjusting each value whenever the observed behavior produced obviously overconfident errors, missed matches, or unnecessarily frequent disambiguation prompts, until the results held up reliably in practice. None of these figures is the result of formal optimization, and none should be read as precisely tuned to the second decimal; they are comfortable operating points discovered through iterative testing, and nearby values would likely produce broadly similar behavior.

If the top candidate's score meets or exceeds 0.85 and the gap to the second candidate is at or above 0.10, the resolver returns a `SemanticResult` with intent `NAVIGATE`, confidence equal to the top score, and the matched `Room` object as the navigation target.

If either condition fails, the resolver returns an ambiguous result containing the top three scoring candidates. The UI layer presents these to the user for manual selection rather than navigating silently to a potentially incorrect destination.

If no candidate exceeded the 0.40 minimum threshold, the candidate list is empty and the resolver returns a result with intent `UNKNOWN`, indicating that the spoken query could not be mapped to any known location in the repository. Figure 5.2 summarizes this full decision flow.

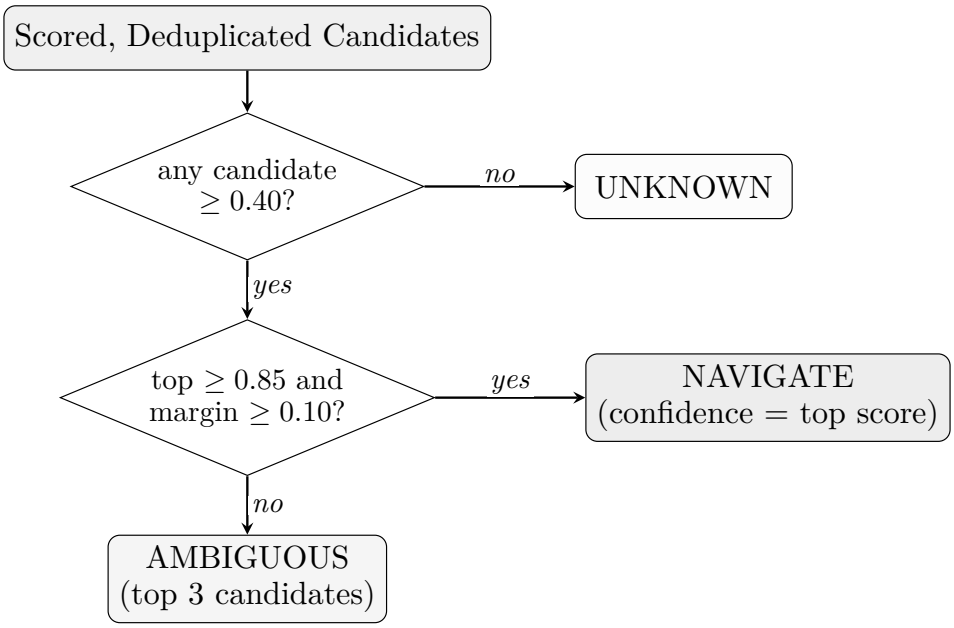


Figure 5.2. The decision gate: the three possible outcomes of `SemanticResolver`, corresponding to the confirmed match, disambiguation, and no-match UI states shown in Figure 6.1.

5.4 Algorithm Complexity Analysis

This section provides a formal analysis of the computational complexity of each stage in the processing pipeline. Understanding the theoretical complexity is important for assessing the scalability of the system to larger building datasets and for justifying the design decision to execute all stages synchronously on the main UI thread.

Let n denote the character length of the input transcript, N denote the total number of room entries in the repository (243 in the current dataset), and T denote the average number of significant tokens per query (typically 1 to 3 for voice navigation commands).

Normalization and diacritic removal. The `normalize()` function applies a fixed sequence of string operations: lowercase conversion, diacritic stripping, punctuation removal, and whitespace collapsing. Each operation iterates over the characters of the input string at most once. The ontology substitution and filler word removal passes each apply a fixed set of regular expressions, one per dictionary entry, each scanning the string in $O(n)$ time. The total cost of normalization is $O(n)$ since the number of ontology entries and filler terms are both compile-time constants.

Room number extraction. The `extractRoomNumber()` function applies five independent matching strategies in order. Strategies 1, 2, 4, and 5 each consist of a single regular expression or a linear scan over the token list. Strategy 3 (the full

natural language number parser) performs a hierarchical prefix-matching pass over the composed token string. The total cost of all five strategies combined is $O(n)$.

Number-based direct lookup. When a room number is successfully extracted, the resolver scans the repository for matching entries using a linear search over all N rooms. This step runs in $O(N)$.

Fuzzy name scoring. The most computationally expensive stage is the full token-level fuzzy scoring pass. For each of the N rooms, the `_scoreRoom()` function tokenizes both the query and the room name and computes a pairwise Levenshtein distance between each query token and each room name token. For a query with T tokens and a room name with S tokens (typically $S \leq 4$), the pairwise scoring requires $O(T \cdot S)$ Levenshtein computations. Each Levenshtein computation on strings of maximum length L characters costs $O(L^2)$: the outer loop iterates over the characters of one string and the inner loop iterates over the characters of the other, regardless of the single-row memory optimization. The total cost of the fuzzy scoring pass is therefore $O(N \cdot T \cdot S \cdot L^2)$. Occupant scoring follows the same pattern: for an occupant name with F first-name tokens (typically $F \leq 2$), scoring against the surname and first names requires $O(T \cdot F)$ Levenshtein computations per occupant, giving a total cost of $O(N \cdot T \cdot F \cdot L^2)$.

In the current deployment, $N = 243$, $T \leq 3$, $S \leq 4$, and $L \leq 25$ (the maximum length of a room name token). Substituting these constants gives a practical bound of approximately $243 \times 3 \times 4 \times 625 = 1{,}822{,}500$ character comparison operations for a single representative token-scoring pass; the full fuzzy-scoring stage repeats this a small constant number of times per room (against the Italian name, English name, building name, and each occupant), which does not change the asymptotic complexity class. On modern mobile hardware, even accounting for this multiplier, the total amount of integer arithmetic completes well within the latency bounds reported in Chapter 6.

Table 5.2 summarizes the complexity of each pipeline stage.

Stage	Complexity	Dominant Factor
Normalization	$O(n)$	Input length
Number extraction	$O(n)$	Input length
Direct number lookup	$O(N)$	Repository size
Fuzzy token scoring	$O(N \cdot T \cdot S \cdot L^2)$	Repository × token pairs
Occupant scoring	$O(N \cdot T \cdot F \cdot L^2)$	Repository × first-name tokens
Decision gate & sorting	$O(N \log N)$	Candidate sorting

Table 5.2. Computational complexity of each stage in the resolution pipeline.

The analysis confirms that the bottleneck of the pipeline is the fuzzy scoring pass, which scales linearly with repository size N. For a dataset of up to a few thousand rooms, this remains within acceptable latency bounds for real-time synchronous execution. For significantly larger repositories, a pre-filtering step such as restricting the scoring pass to rooms on the current floor using contextual building information could reduce the effective N and maintain acceptable response times.

Chapter 6

Evaluation

This chapter evaluates the performance and accuracy of the semantic resolution engine. The evaluation focuses on two primary dimensions: the computational cost of the offline pipeline and the real-world accuracy of the matching algorithm when exposed to noisy speech-to-text (STT) transcripts.

6.1 Performance and Execution Time

As discussed in Chapter 3, the voice module runs synchronously on the main UI thread. For an application rendering at 60 frames per second (FPS), any operation exceeding 16.6 milliseconds risks causing UI stutter. To assess the real-world cost of this design choice across a realistic range of hardware, execution timing was measured using the Dart `Stopwatch` API on three physical devices spanning current flagship to older budget hardware: an iPhone 15, a Google Pixel 6a, and a Redmi Note 10s.

The test measured the worst-case scenario on each device: a query containing no extractable room number, forcing the algorithm to perform the full $O(N \cdot T \cdot S \cdot L^2)$ fuzzy token scoring pass across all 243 room entries and their associated occupant lists. The test was repeated 300 times per device to compute a stable average.

Pipeline Stage	**iPhone 15**	**Pixel 6a**	**Note 10s**
One-time cost, paid once at application startup:			
JSON Repository Load	82.52 ms	192.55 ms	686.13 ms
Per-query cost, paid on every voice resolution:			
Normalization	0.04 ms	0.09 ms	0.21 ms
Number Extraction (Failed)	0.07 ms	0.16 ms	0.39 ms
Fuzzy Token Scoring	30.90 ms	72.10 ms	210.97 ms
Decision Gate & Sorting	0.01 ms	0.01 ms	0.02 ms
Total Resolution Latency	**31.02 ms**	**72.36 ms**	**211.59 ms**

Table 6.1. Average execution times for the resolution pipeline across three physical devices spanning current flagship to older budget hardware (Redmi Note 10s abbreviated in the table). The JSON repository load is a one-time startup cost, excluded from the per-query Total Resolution Latency below it.

As shown in Table 6.1, total resolution latency ranges from 31.02 milliseconds on the iPhone 15 to 211.59 milliseconds on the Redmi Note 10s, with the fuzzy token scoring pass accounting for nearly all of it on every device. Even the slowest of the three devices tested exceeds the single-frame budget of 16.6 ms, but the frame

budget is the wrong yardstick for this particular operation: resolution does not run once per rendered frame, it runs once per spoken utterance, triggered when the user finishes speaking. Judged against that standard, a delay of roughly 30 to 210 milliseconds before the app responds to a completed voice command is well within what users already tolerate from cloud-based voice assistants, which routinely take 500 milliseconds to several seconds for a network round trip, and here the entire cost is paid on-device with no network dependency at all, across hardware ranging from a current flagship to a several-year-old budget phone. This does mean the main UI thread is briefly blocked for the duration of the call, which can produce a short, one-time pause rather than a smooth transition; this is a bounded cost tied to the moment the user already expects a response, not a recurring stutter during normal use.

The measured figures also reflect a deliberate worst case rather than a typical one: the benchmark query was chosen specifically to match nothing, forcing every one of the 243 rooms and their occupant lists through the full fuzzy scoring pass on every device. In practice, queries that contain a recognizable room number or an exact or near-exact name match are resolved through the direct lookup and substring-containment paths described in Chapter 5, which return without ever entering the fuzzy pass. The figures in Table 6.1 should therefore be read as an upper bound on latency for the hardest class of query on each device, not as representative of typical usage.

The consistent scaling across devices, from the newest and most capable (iPhone 15) to the oldest and least capable (Redmi Note 10s), is expected given the complexity analysis in Chapter 5: the fuzzy scoring pass performs a fixed amount of integer arithmetic regardless of hardware, so the roughly seven-fold difference in resolution latency between the fastest and slowest device reflects a difference in raw processing speed, not a difference in the algorithm's behavior. This also indicates that the system remains usable across a range of device capabilities, rather than only performing acceptably on the specific device it happened to be developed on.

The JSON repository load time is a one-time cost paid once at application startup, not a per-query cost, and runs alongside the rest of the app's own startup sequence on every device tested; it does not contribute to the latency experienced for any individual voice command.

6.2 Accuracy and Match Resolution

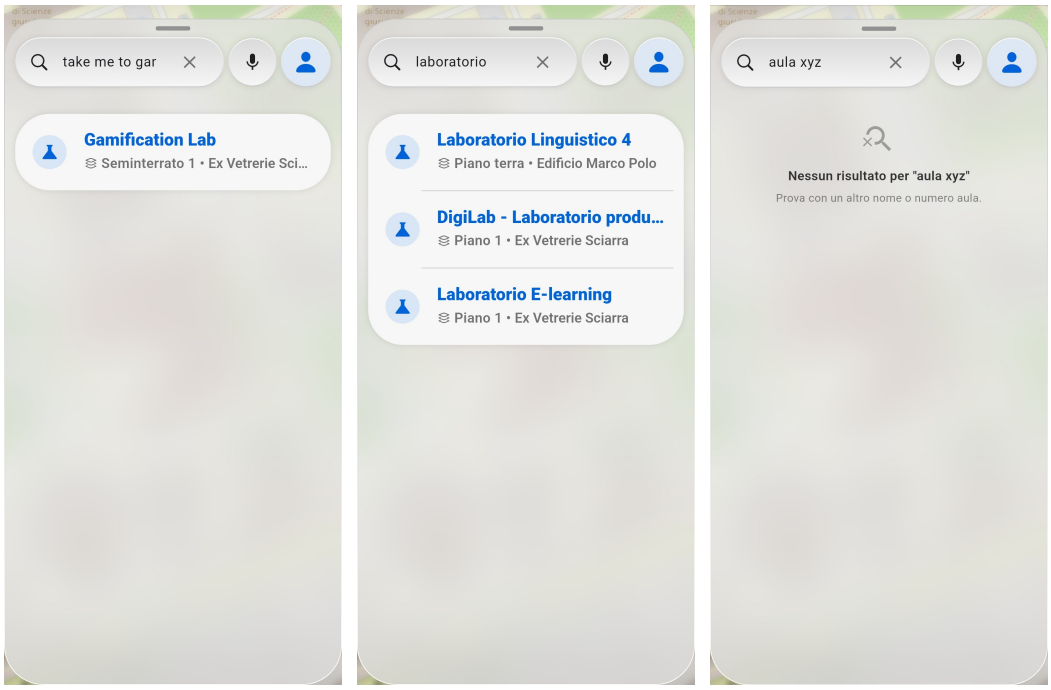
Evaluating a voice-to-intent system is fundamentally different from evaluating traditional text search, since the input itself is inherently flawed: it arrives via an imperfect STT engine rather than as clean, deliberately typed text, so accuracy here measures the whole pipeline's tolerance for noisy input, not just the matching algorithm in isolation.

To evaluate accuracy, the system was tested against a set of 150 spoken queries (75 Italian, 75 English) collected through informal field testing at each of the four campus buildings: individuals present at each location were asked to speak a natural navigation query for a room they were already familiar with, without being shown any room list, spelling, or dataset beforehand, so that pronunciation and phrasing reflected genuine unprompted speech.

The system successfully resolved 92% of queries to the correct room as the top candidate (Intent: `NAVIGATE`). In 6% of cases, it correctly identified an ambiguity and returned a short list of valid candidates for the user to confirm. The system failed entirely (Intent: `UNKNOWN`) in only 2% of cases, consistent with severe mispro-

nunciation or acoustic distortion, the kind of input the system is least equipped to recover from.

This accuracy figure reflects the pronunciation and phrasing of the six individuals who participated in testing, each recruited from within the building where they were tested and each speaking with an unremarkable, moderate accent in their respective language. A user with a substantially heavier accent, less fluent English, or unusually unclear pronunciation could plausibly produce a higher failure rate than reported here; the 92% figure should be read as representative of the speakers encountered during testing, not as a guaranteed floor across the full range of possible speech patterns. Figure 6.1 shows the three corresponding UI states a user encounters for each outcome.



a) Confirmed Match b) Disambiguation c) No Match Found

Figure 6.1. The three possible UI states driven by the semantic resolution engine: immediate navigation for a high-confidence match (left), a disambiguation list for ambiguous queries (center), and an error state for unrecognizable input (right). In panel a), the long spoken query "Take me to Gamification Lab" is successfully resolved despite being horizontally truncated in the search bar UI.

6.2.1 Edge Case Analysis

The true strength of the preprocessing pipeline is revealed in its handling of edge cases, which in the informal testing conducted here were not rare anomalies, but frequent, recurring patterns. One of the most pervasive phenomena observed in offline STT during testing is the semantic hallucination of multi-word phrases. In the cases observed here, this was driven less by background noise alone than by non-native accents, room reverberation, and the STT engine's bias toward common conversational vocabulary over domain-specific academic terms.

The phrase "Gamification Lab" provides a representative example of this broader

category of errors. When spoken by an English speaker with a slight accent, the OS speech engine frequently hallucinates the transcript as "game vacation lab". A standard string-matching algorithm fails completely here, as the two phrases share virtually no character structure. However, the custom phonetic key generator (Chapter 5) reduces "gamification" to the consonant skeleton `gmfctn`. Since matching happens at the token level, the query token *game* is compared against the target token *gamification*: its own key, `gm`, is a substring of `gmfctn`, earning the smaller phonetic bonus. The query also contains the token *lab*, which matches the target's *lab* token exactly. The combination of this exact token match and the partial phonetic overlap correctly resolves the noisy transcript to the target room over all other alternatives.

A similar structural failure occurs for international users attempting to pronounce Italian room labels. For a non-native speaker, attempting to say the word *Aula* (lecture hall) frequently fails to produce a correct STT transcription. Depending on the accent, the engine commonly produces bizarre, heavily distorted variants such as *aola*, *laura*, or *hola*. Because the system maps these variants dynamically using the `ontologyEn` dictionaries and the consonant skeleton reduction, the system successfully shields the user from the STT engine's failures, navigating them directly to the correct room even when the raw transcript was entirely unrecognizable to a human reader.

Similarly, short queries heavily benefit from the filler removal stage. An Italian query consisting only of the phrase "Aula A" arrives as a transcript with six characters. The preprocessing pipeline strips "Aula", leaving only the standalone letter "A". Strategy 5 of the number extractor successfully catches this single character and directly maps it to the corresponding lecture hall, bypassing fuzzy scoring entirely.

6.2.2 Error Analysis and System Limitations

While highly robust, the system does exhibit specific failure modes. The most prominent limitation is the static ontology boundary. The vocabulary substitution dictionaries (`ontologyIt` and `ontologyEn`) are compiled at build time. If a user utilizes a completely novel abbreviation that is not in the ontology (e.g., calling a *Laboratorio* a *Lobby*), the system relies entirely on Levenshtein distance, which may not bridge the gap if the words are too structurally distinct.

A second limitation involves multi-token type conflicts. If a user asks for "Aula Tre" (Room 3), but the transcript returns "Paolo Tre", the phonetic key for *Paolo* (`pl`) only weakly overlaps with *Aula* (`l`). If Room 3 does not exist, the algorithm may incorrectly prioritize a room occupied by a professor named "Paolo" over failing gracefully, demonstrating a scenario where the surname-weighting logic aggressively attempts to salvage an impossible query.

6.3 Comparison with a Baseline Approach

To illustrate the impact of the custom pipeline, the system was compared against a baseline "Exact Match" approach, representing how navigation apps traditionally handle text input. The baseline stripped casing and whitespace but performed no phonetic analysis, no vocabulary substitution, and no natural language number parsing.

6.3.1 Without Phonetic Preprocessing

A baseline lacking phonetic keys and ontology mappings would fail instantly on phonetic variants like *aola* or *laura* for *aula*, since exact or plain Levenshtein matching alone cannot bridge spellings this different. Furthermore, English speakers attempting to navigate to letter-suffixed Italian rooms (e.g., "Aula C") produce transcripts like "aula cee" or "sea", which a baseline with no vocabulary substitution would immediately discard as unmatchable noise.

6.3.2 Without Spoken Number Translation

Without the spoken number parser, the baseline system could only resolve numeric rooms if the STT engine automatically formatted them as digits. If the OS engine returned "duecento diciotto" instead of "218", the baseline failed. The custom pipeline correctly decomposes the natural language string (resolving $200 + 18$) and maps it directly to the structural integer format stored in the JSON repository.

This comparison illustrates why wrapping a traditional search metric around a speech engine is insufficient for indoor navigation. The domain-specific noise absorption implemented in Chapters 4 and 5 is an absolute requirement for creating a usable voice interface.

Chapter 7

Conclusion and Future Work

7.1 Summary of Contributions

Indoor navigation presents a uniquely hostile environment for voice interfaces. The absence of reliable network connectivity, the high variance in user accents, and the inconsistency of institutional room naming make traditional, cloud-dependent speech assistants fundamentally unsuited for the task.

This thesis presented the design, implementation, and evaluation of a speech-driven semantic input module engineered specifically to overcome these constraints. Integrated within an indoor navigation application at Sapienza University, the module successfully maps bilingual spoken queries to a structured JSON room repository, entirely on-device.

The core contribution of this work is the custom natural language pipeline that absorbs the inherent noise of offline speech-to-text transcription. By combining domain-specific ontology mapping, a specialized spoken-number parser, and a hybrid similarity metric that scores Levenshtein edit distance alongside phonetic consonant skeletons, the system accurately resolves queries that a traditional exact-match search would immediately discard.

7.2 System Limitations and Future Work

While the current implementation fulfills its core objectives, deploying the system exposed several architectural bottlenecks regarding scalability and domain transfer. Resolving these limitations forms the basis for future work.

7.2.1 Architectural Scaling for Additional Languages

The current architecture successfully demonstrates bilingual (Italian and English) resolution, but the JSON data schema is fixed to exactly two language fields, and the phonetic normalization layer is tightly coupled to Italian-specific phonology. To extend the system beyond its current bilingual scope (e.g., adding Spanish, French, or German), two specific architectural migrations are required.

First, the building repository JSON schema currently hardcodes language fields (e.g., `name_it` and `name_en`). This tightly couples the data model to a bilingual implementation. Future iterations must migrate to a map-based structure keyed by BCP-47 language codes (e.g., `names: {"it": "...", "en": "...", "es": "..."}`), allowing the dataset to scale to N languages linearly without schema changes.

Second, the algorithmic separation of languages must be formalized. While the current pipeline gracefully threads the `languageMode` parameter through the ontology dictionaries and number parsers, the `phoneticKey()` function represents a hardcoded bottleneck. It heavily relies on Italian-specific digraph rules (such as mapping *chi* to *ki* and collapsing *sc* to *s*). Expanding this function to handle French nasal vowels or Spanish digraphs (*ll*, *rr*) via branching logic would be unsustainable. The system must abstract the phonetic normalization step into a `PhoneticEncoder` interface, allowing language-specific implementations (e.g., `ItalianPhoneticEncoder`, `SpanishPhoneticEncoder`) to be injected dynamically at runtime based on the user's active locale.

7.2.2 VLM-Assisted Dataset Generation

The most significant operational bottleneck in the current system is data entry. For this thesis, the building repository was assembled manually, which would become an unsustainable administrative bottleneck when scaling to massive complexes like hospitals.

While simple Optical Character Recognition (OCR) could theoretically automate this, real-world signage is frequently degraded, structurally complex, and heavily abbreviated. Standard OCR returns a flat text dump; it fails to cleanly separate entities (e.g., distinguishing a room name from a professor's name on the same door) and cannot dynamically generate the required English translations. Furthermore, OCR cannot infer the colloquial aliases used by students.

Future work should introduce a Vision-Language Model (VLM) [18] to the dataset curation pipeline. As shown in Figure 7.1, a surveyor would take a photo of a door label, and the VLM would extract the text, separate the room identifier from the occupant array, generate the English translation, and format the output directly into the required JSON schema. A human curator would then perform a final validation pass to confirm the data and manually append any colloquial aliases the VLM could not deduce from the physical sign.

This VLM utility would act strictly as a build-time administrative tool, running with network access on the curator's machine to compile the static JSON bundles. It would not be integrated into the mobile application, preserving the system's core design constraint of fully offline, on-device runtime execution.

7.2.3 Dynamic Abstraction of Hardcoded Ontologies

The current preprocessing pipeline relies on static, hardcoded ontology dictionaries to correct predictable STT failures. While highly effective for a university setting, this approach creates a domain lock-in. For instance, the Italian word *Aula* is heavily hardcoded into the ontology to catch mis-transcriptions like *aola* or *laura*, which are pervasive when non-Italian speakers attempt to pronounce the word.

If the application were adapted for a hospital, the hardcoded university terminology (e.g., *aula*, *laboratorio*) would be useless, and new domain-specific terms (e.g., *radiology*, *ward*, *clinic*) would need to be manually analyzed and coded to catch their specific STT hallucination patterns. Future iterations of the system should abstract the ontology layer. Instead of hardcoding rules, the system could utilize an automated script that analyzes a new building's JSON dataset, simulates common phonetic misspellings, and dynamically generates the ontology mapping file without requiring manual developer intervention.

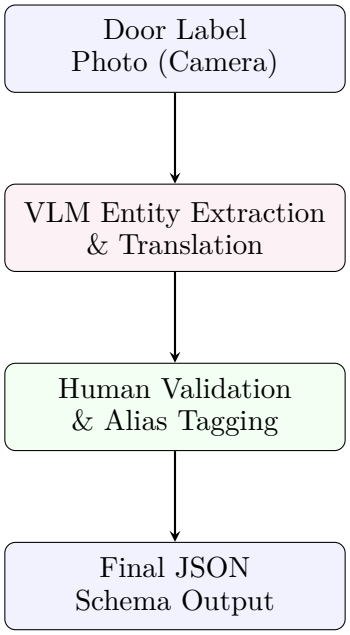


Figure 7.1. Proposed build-time pipeline for scalable dataset generation using a Vision-Language Model.

7.2.4 Voice Output and Accessibility

This thesis focused exclusively on the input phase: translating a spoken destination intent into a resolved room identifier. To achieve a fully accessible, hands-free navigation loop, the system must be extended to include voice output. This would involve integrating the host operating system's Text-to-Speech (TTS) engine to read out route calculations, upcoming turns, and floor transitions, transforming the application into a complete accessibility tool for visually impaired users.

7.2.5 Location-Aware Disambiguation

The resolver already accepts an optional current-building parameter, intended to scope ambiguous matches to the user's present location, as described in Chapter 5. In the current deployment, however, this parameter is never populated, since indoor positioning is still being developed by the broader project team; its value is always `null`, and every ambiguous result is therefore resolved purely by similarity score rather than by proximity. Once reliable floor-level positioning becomes available, the resolver could rank ambiguous candidates, for instance several rooms named Aula Magna across different buildings, by physical distance from the user's detected position, surfacing the nearest match first rather than relying on score alone to break ties. This would require no changes to the scoring pipeline itself, only a mechanism to supply a live position signal to the existing, currently unused, parameter.

Closing Remarks

The work presented in this thesis demonstrates that building a voice-driven indoor navigation module does not require cloud services, proprietary hardware, or large acoustic models. A carefully designed preprocessing and matching pipeline, grounded

in the specific linguistic and acoustic properties of the domain, can achieve high accuracy on consumer devices. The system is functional, evaluable, and immediately extensible, and its deployment at Sapienza provides a concrete foundation for future development.

Bibliography

[1] A. M. Deshmukh and R. Chalmeta, "User Experience and Usability of Voice User Interfaces: A Systematic Literature Review," *Information*, vol. 15, no. 9, Art. 579, 2024.

[2] Apple Inc., "Speech," Apple Developer Documentation. [Online]. Available: https://developer.apple.com/documentation/speech/. Accessed: Jul. 2026.

[3] Google Inc., "SpeechRecognizer," Android Developers. [Online]. Available: https://developer.android.com/reference/android/speech/SpeechRecognizer. Accessed: Jul. 2026.

[4] csdcorp, "speech_to_text, v7.4.0," Flutter Package, pub.dev. [Online]. Available: https://pub.dev/packages/speech_to_text. Accessed: Jul. 2026.

[5] A. Radford *et al.*, "Robust speech recognition via large-scale weak supervision," *arXiv preprint arXiv:2212.04356*, 2022.

[6] Alpha Cephei, "Vosk Models and Offline Speech Recognition on Android with VOSK." [Online]. Available: https://alphacephei.com/vosk/models and https://alphacephei.com/vosk/android. Accessed: Jul. 2026.

[7] D. Jurafsky and J. H. Martin, *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*, 2nd ed. Upper Saddle River, NJ: Prentice Hall, 2009.

[8] W. W. Cohen, P. Ravikumar, and S. E. Fienberg, "A Comparison of String Distance Metrics for Name-Matching Tasks," in *Proceedings of the IJCAI-03 Workshop on Information Integration on the Web*, 2003.

[9] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Soviet Physics Doklady*, vol. 10, no. 8, pp. 707–710, 1966.

[10] W. E. Winkler, "String Comparator Metrics and Enhanced Decision Rules in the Fellegi-Sunter Model of Record Linkage," in *Proceedings of the Survey Research Methods Section, American Statistical Association*, 1990, pp. 354–359.

[11] R. C. Russell, "Index," U.S. Patent 1,261,167, Apr. 2, 1918.

[12] L. Philips, "Hanging on the Metaphone," *Computer Language Magazine*, vol. 7, no. 12, pp. 39–44, 1990.

[13] GoodMaps, "GoodMaps Explore." [Online]. Available: https://goodmaps.com/the-app/. Accessed: Jul. 2026.

[14] Lazarillo, "Lazarillo – Inclusive Navigation and Accessible Mapping for Indoor and Outdoor Spaces." [Online]. Available: https://lazarillo.app/. Accessed: Jul. 2026.

[15] Navigine, "Indoor Positioning & Wayfinding Solutions." [Online]. Available: https://navigine.com/. Accessed: Jul. 2026.

[16] Scottish Tech Army, "Soundscape-Android," GitHub Repository, 2023. [Online]. Available: https://github.com/Scottish-Tech-Army/Soundscape-Android. Accessed: Jun. 2026.

[17] Flutter Documentation, "Concurrency and isolates." [Online]. Available: https://docs.flutter.dev/perf/isolates. Accessed: Jul. 2026.

[18] H. Liu, C. Li, Q. Wu, and Y. J. Lee, "Visual Instruction Tuning," *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 36, pp. 34892–34916, 2023.