Sorbonne Université
Master ANDROIDE

Reconnaissance audio par des modèles de fondation pour un robot manipulateur mobile

UE de projet M1

Inés Tian RUIZ-BRAVO PLOVINS – Ines RAHAOUI

2025

Table des matières

Chapitre 1

Introduction

Dans le cadre du master informatique parcours ANDROIDE à Sorbonne Université, nous avons travaillé sur la mise en place d'une interface homme-machine permettant à un robot manipulateur mobile d'interpréter des instructions vocales.

Ce projet a été encadré par Stéphane DONCIEUX et Emiland GARRABE, au sein de l'ISIR (Institut des Systèmes Intelligents et de Robotique).

L'objectif était d'intégrer des modules de reconnaissance vocale (Speech-to-Text) et de synthèse vocale (Text-to-Speech) avec une architecture cognitive pilotée par un modèle de fondation, afin de permettre au robot d'interagir de manière naturelle avec un utilisateur humain.

Le dépôt contenant l'ensemble de notre travail est accessible à l'adresse suivante : https://github.com/iinnesperado/pandroide-reconnaissance-audio

Chapitre 2

État de l'art

L'émergence des modèles de fondation, tels que les grands modèles de langage (LLM) ou les modèles vision-langage (VLM), permet de transformer en profondeur le domaine de la robotique. Grâce à leur capacité à traiter un langage ouvert et à contextualiser des instructions complexes, ces modèles deviennent des interfaces puissantes entre des utilisateurs non-experts et des systèmes robotiques.

Au sein de l'ISIR (Institut des Systèmes Intelligents et de Robotique), des travaux sont actuellement menés autour de robots manipulateurs mobiles capables de saisir et manipuler des objets ainsi que d'interagir socialement avec des humains. Ces travaux visent à rendre les robots capables de collaborer efficacement dans des environnements partagés avec les humains.

Dans ce cadre, l'ISIR explore également l'intégration de modèles de fondation, notamment les LLM, pour leur capacité à interpréter des commandes en langage naturel et à raisonner sur des tâches complexes. L'utilisation de ces modèles offre une nouvelle approche dans le développement d'architectures cognitives robotisées, en s'appuyant sur des représentations linguistiques riches et flexibles.

Cependant, un des principaux défis dans l'intégration des LLMs dans des systèmes robotiques interactifs réside dans leur fonctionnement séquentiel : une interaction typique consiste à fournir un prompt puis à attendre une réponse. Or, dans le cadre d'un dialogue oral en condition réelle, cela demande l'utilisation de modèles performants permettant de répondre rapidement et efficacement tout en considérant l'utilisation d'une puissance de calcul limitée.

Chapitre 3

Evaluation des outils

Pour permettre aux robots d'interagir efficacement avec des instructions vocales, il est essentiel que l'audio soit transcrit de manière précise et rapide. L'exploration et l'évaluation des outils de transcription automatique constituent donc une étape cruciale de notre projet.

Parmi les nombreux outils *Speech-to-Text* disponibles, notre attention s'est portée sur la famille *Whisper* développée par OpenAI, reconnue pour son efficacité dans de nombreuses études. Nous avons choisi de nous concentrer plus particulièrement sur *Faster-Whisper*, une version optimisée pour la vitesse, critère essentiel pour notre projet — notamment dans l'optique d'un déploiement à destination d'utilisateurs non experts.

3.1 Méthodologie des tests

Pour évaluer les performances du modèle, nous avons constitué un jeu de données audio multilingue (français et anglais), correspondant aux principales langues utilisées dans notre application cible. Les audios sont volontairement courts afin de pouvoir en tester un grand nombre dans des conditions variées.

Chaque audio a été décliné avec différents niveaux de bruit de fond (de 0 à 100 %) pour simuler des environnements plus ou moins bruyants. Cela nous a permis de construire un benchmark qualitatif.Nous avons ensuite évalué la qualité des transcriptions en s'inspirant du **taux de reconnaissance de mot** (*Word Accuracy*), en nous basant sur la formule classique du taux d'erreur de mot (WER - *Word Error Rate*) :

$$\mathrm{WER} = \frac{S + D + I}{N} \tag{3.1}$$

où :

- S est le nombre de substitutions,
- D est le nombre de suppressions,
- I est le nombre d'insertions,
- N est le nombre total de mots dans la transcription de référence.

Le **taux de reconnaissance de mot** (WAcc) est alors défini comme :

$$\text{WAcc} = 1 - \text{WER} \tag{3.2}$$

Ce taux représente la proportion de mots correctement reconnus dans la transcription automatique, et constitue donc un indicateur plus intuitif de performance pour notre étude.

Tous les tests ont été effectués en utilisant le modèle *large* de Faster-Whisper, qui, bien que plus lourd, offre de meilleures performances. À noter que les tests ont été réalisés sur CPU, ce qui peut impacter les vitesses d'exécution, un GPU offrant de meilleures performances dans un contexte réel.

3.2 Résultats

La figure 3.1 illustre l'évolution du taux d'erreur sur un ensemble de 10 audios selon différents niveaux de bruit. On observe une dégradation marquée à partir de 60 % de bruit ajouté, ce qui reste un bon résultat, car en conditions réelles, le bruit ambiant est souvent moins important. Par ailleurs, ce bruit pourrait à terme être réduit par des techniques de traitement audio (filtrage, réduction de bruit) que nous envisageons pour les versions futures du projet. Les audios bruités utilisés dans les tests peuvent être écoutés sur notre dépôt GitHub (répertoire `samples/withNoise`) pour se faire une idée de leur qualité.

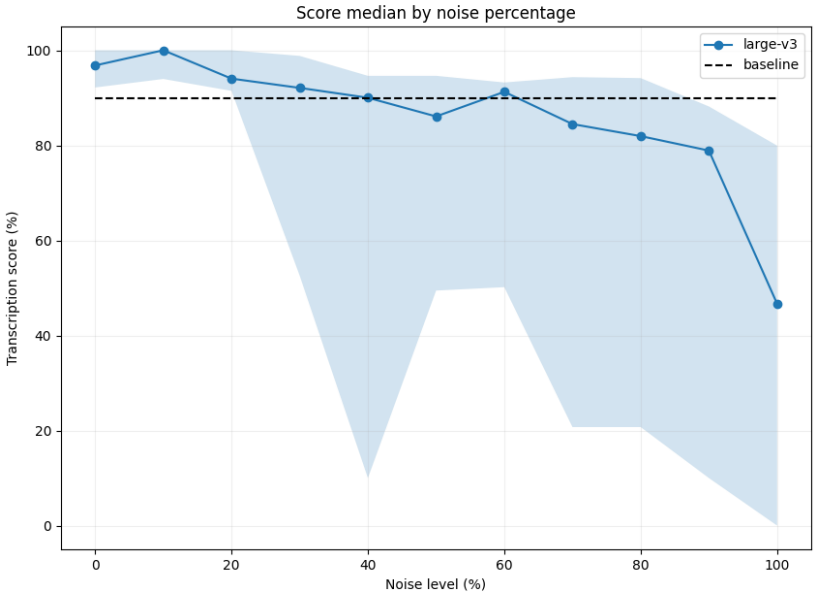

FIGURE 3.1 – Taux d'erreur de transcription (WAcc) pour le modèle "large-v3" en fonction du niveau de bruit (0-100%). Chaque point correspond à la médiane calculée sur 10 échantillons, avec les intervalles interquartiles (q1-q3) représentés par la zone ombrée. La ligne en pointillés indique le seuil minimal de WER pour une transcription fiable avec Faster-Whisper. Les résultats indiquent que ce modèle est performant même pour des environnement de bruit modérer.

Un des problèmes identifiés concerne la nature des erreurs de transcription. En effet, le taux d'erreur ne tient pas compte de l'importance sémantique des mots. Par exemple, dans la phrase *« Va dans la cuisine »*, une erreur sur le mot *la* est peu impactante, alors qu'une erreur sur *cuisine* rend la commande incompréhensible. Une solution envisagée serait de pondérer les mots selon leur importance dans la phrase, mais cette approche s'est révélée trop coûteuse en temps pour être mise en œuvre dans le cadre de ce projet. En pratique, ce problème est partiellement contourné grâce à l'utilisation du LLM, qui peut demander à l'utilisateur de répéter la commande en cas d'ambiguïté. Cela s'avère d'ailleurs plus robuste en conditions réelles, où l'on ne dispose pas de la transcription correcte pour faire une comparaison directe.

Un autre avantage important de *Faster-Whisper* est sa capacité à reconnaître correctement des mots rares ou spécifiques, notamment les termes techniques, les sigles et les noms utilisés en laboratoire, ce qui est particulièrement utile dans notre contexte.

3.3 Comparaison avec d'autres modèles

Après avoir testé le modèle *large*, nous avons évalué les performances de versions plus légères sur les mêmes échantillons audio. Ces modèles, moins gourmands en ressources, sont plus adaptés notamment sur des systèmes embarqués aux capacités limitées.

Cependant, cette légèreté se traduit par une baisse notable des performances. Nous avons rapidement écarté les modèles *tiny* et *base*, dont les résultats en transcription étaient trop dégradés pour une utilisation réaliste (cf. Annexe C.1). Leurs taux d'erreur élevés et leur sensibilité au bruit les rendent peu fiables, même dans des conditions d'enregistrement relativement propres.

En revanche, le modèle *small* offre un compromis plus acceptable. Comme le montre la figure 3.2, ses performances restent correctes tant que le bruit ambiant reste modéré, c'est-à-dire en dessous de 30 %, et sa vitesse de transcription est elle bien plus rapide (cf. Annexe C.2). Bien qu'il soit en retrait par rapport au modèle *large*, notamment en termes de précision, il constitue une alternative intéressante pour les scénarios où les ressources matérielles sont contraintes.

Cette comparaison nous permet donc de justifier le choix du modèle *large* pour les expérimentations actuelles, tout en gardant à l'esprit que des optimisations futures pourraient permettre d'envisager l'usage du modèle *small* dans un contexte déployé.

3.4 Conclusion

Au terme de cette évaluation, nous avons décidé de conserver *Faster-Whisper* avec le modèle *large* comme outil principal de transcription dans notre projet. Malgré quelques défauts, il reste plus performant que la plupart des autres solutions disponibles, et son usage dans des applications réelles s'avère robuste et prometteur.

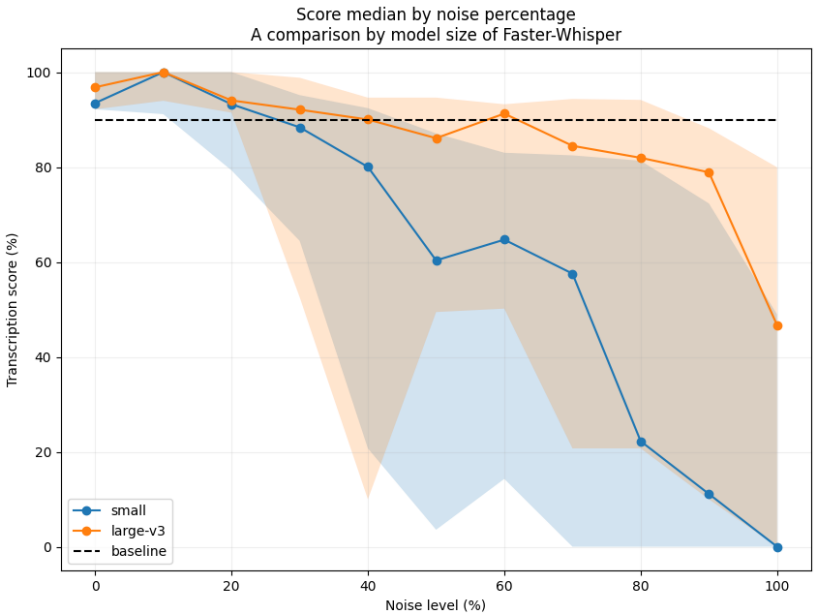

FIGURE 3.2 – Comparaison du taux d'erreur de transcription entre les modèles "small" et "large-v3" de Faster-Whisper pour différents niveaux de bruit (0-100%). Chaque point représente la médiane calculée sur 10 échantillons, avec les intervalles interquartiles (q1-q3) indiqués par les zones ombrées. La ligne en pointillés correspond au seuil minimal de WER pour une transcription fiable.

Chapitre 4

Implémentation de la reconnaissance vocale

Après avoir sélectionné le modèle optimal pour la transcription, nous avons implémenté la reconnaissance vocale à l'aide de `Python`, qui offre une bonne compatibilité avec `Faster-Whisper`.

Pour gérer l'audio en temps réel, nous avons utilisé la bibliothèque `PyAudio`, qui permet de capturer l'audio depuis le microphone. Ne pouvant pas travailler directement sur le robot, nous avons opté pour l'utilisation de `Ollama` (version 3.2). Bien que la version 3.1 soit la plus performante actuellement disponible, elle n'était pas supportée par nos machines, limitant ainsi les performances. Ces résultats pourront toutefois être améliorés avec des machines plus puissantes.

Comme mentionné précédemment, nous avons choisi le modèle *large* de `Faster-Whisper` pour sa précision, bien qu'il soit plus lent sur nos machines.

Notre première idée était de capturer l'audio en continu avec `PyAudio`, de le découper en fragments, et de transmettre ces morceaux à `Faster-Whisper` pour une transcription mot par mot. Cependant, cette stratégie s'est avérée inefficace : le temps nécessaire à la transcription d'un fragment entraînait la perte des paroles suivantes de l'utilisateur. La puissance de calcul de nos ordinateurs s'est révélée insuffisante pour supporter cette approche, et les résultats étaient de qualité médiocre.

Nous avons donc opté pour une solution plus simple et plus fiable : l'utilisateur appuie sur une touche pour démarrer l'enregistrement, puis appuie à nouveau pour l'arrêter. Le fichier audio ainsi généré est ensuite transmis à `Faster-Whisper` pour la transcription. Cette méthode est plus optimale en termes de complexité et de temps de calcul, ce dernier étant comparable à celui analysé dans la partie précédente puisque les mêmes fonctions sont utilisées.

Nous supposons que chaque enregistrement ne contient qu'une seule instruction de l'utilisateur, comme par exemple : « Va chercher de l'eau dans la cuisine. » Cette instruction transcrite est ensuite envoyée à notre modèle LLM, configuré de manière à comprendre le contexte et à produire une réponse sous forme de code.

Pour cela, nous lui fournissons un *prompt* précisant qu'il s'agit d'un **robot manipulateur**, comme ceux utilisés dans le cadre de ce projet. Le prompt inclut également une série de fonctions que le modèle peut appeler, selon le principe du *Code as Policies*. Cela

signifie que le LLM ne se contente pas de décrire ce qu'il ferait, mais génère directement du code exécutable.

```
Transcription of audio 'snack-20' took : 7.54s
class Robot:
    def __init__(self):
        self.current_location = 'living room'
        self.inventory = []

    def current_location(self):
        return self.current_location

    def inventory(self):
        return self.inventory

    def go_to(self, place):
        self.current_location = place

    def report_status(self):
        print(f"Current location: {self.current_location}")
        print("Inventory:", self.inventory)

    def pick_up(self, item):
        self.inventory.append(item)
        print(f"Picked up: {item}")

# Initialize the robot
robot = Robot()

# Move to kitchen
robot.go_to('kitchen')

# Pick up snack from kitchen (assuming there's a 'snack' in inventory)
robot.pick_up('snack')

# Give the snack to someone
robot.give_to('snack', 'recipient')
```

FIGURE 4.1 – Exemple de génération de code avec le principe *Code as Policies*, où le LLM choisit directement les fonctions à exécuter. Ollama commence par donner le code formel de ces capacités puis le code exécutable.

L'instruction issue de la transcription est ensuite injectée dans un *prompt* structuré comme suit :

```
messages = [
  {
    'role': 'user',
    'content': f'You are a robot assistant with the following capabilities
       in the Python class:  {coda} '
    'The users asks : {query} '
    f'Some useful informations: "{best_doc}". '
    'What action(s) do you take? Respond with only the necessary function
       calls in Python-like syntax.',
  },
]
```

Listing 4.1 – Prompt utilisé pour interagir avec le modèle LLM

Après plusieurs itérations, le prompt présenté en figure 4.1 s'est révélé le plus efficace pour obtenir une réponse correcte et exécutable. Les prompts précédents donnaient des résultats incohérents ou inexploitables.

Pour améliorer encore la précision des réponses, nous avons également expérimenté l'intégration de la méthode **RAG** (*Retrieval-Augmented Generation*). Cette technique permet au LLM d'interroger des documents pertinents pour enrichir ses réponses. Dans notre cas, cela permet par exemple de fournir au modèle des informations sur la localisation des objets, en les stockant dans des documents accessibles via RAG.

Enfin, pour pallier les erreurs de transcription vues précédemment, nous avons configuré le prompt de sorte qu'`Ollama` puisse redemander à l'utilisateur de reformuler une instruction ambiguë. Cela constitue une solution pratique dans le cadre d'une interaction orale, où il est difficile de détecter automatiquement les erreurs de sens sans texte de référence.

Chapitre 5

Ordre asynchrone

L'intégration du LLM avec la méthode RAG se révèle particulièrement pertinente dans le cas des ordres asynchrones. En effet, il est fréquent que l'instruction initiale fournie par l'utilisateur soit incomplète ou peu claire. Il devient alors utile de permettre à l'utilisateur de formuler des instructions supplémentaires pour affiner la requête et améliorer ainsi la réponse du robot.

Une des idées principales envisagées dans ce projet consiste à enrichir progressivement la base documentaire utilisée pour le RAG avec les informations fournies par l'utilisateur au fil de l'interaction. Toutefois, un défi majeur réside dans la capacité à distinguer les données pertinentes à intégrer dans la base des simples instructions ponctuelles ne nécessitant pas d'archivage.

Une solution simple que nous avons testée consiste à considérer le premier audio comme l'instruction principale, puis à interpréter les suivants comme des compléments d'information. Ces ajouts successifs permettent au modèle LLM d'améliorer sa compréhension du contexte et de générer une réponse plus précise.

Par exemple, dans un premier temps, si l'utilisateur dit simplement *"va chercher de l'eau"*, le modèle renvoie la commande suivante :

```
report_status("Going to get water")
```

Cependant, après avoir ajouté une information complémentaire dans un second message, telle que *"l'eau est dans le salon"*, le modèle est alors capable de produire une réponse plus complète et pertinente :

```
go_to("living room")
pick_up("water")
```

Les fonctions disponibles dans le prompt sont les suivantes :

- `current_location()` → Donne la position actuelle du robot (départ dans le salon).
- `inventory()` → Liste des objets actuellement dans l'inventaire (initialement vide).
- `go_to(place)` → Permet au robot de se déplacer vers un lieu donné.
- `report_status()` → Permet au robot de communiquer ce qu'il est en train de faire.
- `pick_up(item)` → Permet au robot de ramasser un objet et de l'ajouter à son inventaire.

— `give_to(item, recipient)` → Permet au robot de donner un objet à un destinataire.

Cette approche de base pourrait être considérablement améliorée en développant une capture audio en continu, plutôt qu'un enregistrement ponctuel déclenché manuellement. Une telle amélioration permettrait, par exemple, de détecter un mot déclencheur (*trigger word*), à l'image des assistants vocaux grand public comme Siri ou Alexa. En utilisant cette stratégie, le système pourrait reprendre automatiquement la capture lorsqu'il détecte une tentative de clarification ou une réponse de l'utilisateur à une requête du robot, rendant ainsi la communication plus fluide et naturelle.

Chapitre 6

Conclusion

Ce projet nous a permis de mettre en place les fondations d'une interface vocale fonctionnelle, destinée à être utilisée sur un robot mobile. L'intégration complète avec le robot sera réalisée après la rédaction de ce rapport.

Nous avons procédé à une analyse comparative rigoureuse des outils existants afin de sélectionner ceux les plus adaptés à notre usage. Cette démarche nous a conduits à utiliser des solutions telles que *Faster-Whisper* pour la transcription audio, et la méthode RAG pour améliorer la précision des réponses générées par le modèle de langage.

Bien que la partie sur les ordres asynchrones n'ait pas pu être pleinement développée faute de temps, elle constitue une piste prometteuse pour les évolutions futures du projet.

Ce travail nous a également permis de découvrir de nouveaux outils et bibliothèques, mais aussi de mieux appréhender la méthodologie propre à un projet de recherche, notamment en ce qui concerne l'évaluation rigoureuse des solutions techniques. Nous avons ainsi pris conscience que l'analyse d'outils, que nous pensions initialement simple et rapide, requiert en réalité rigueur, tests empiriques et méthodologie.

Nous tenons à remercier particulièrement nos encadrants, Stéphane Doncieux et Emiland Garrabe, pour leur accompagnement constant et la qualité de leurs conseils tout au long de ce projet.

Annexe A

Cahier des charges

A.1 Introduction

A.1.1 Contexte du projet

Les grands modèles de fondation (Foundation Models) représentent une avancée significative dans le domaine de l'intelligence artificielle. Leur capacité à comprendre et générer du langage naturel, ainsi qu'à établir des relations entre différentes modalités (texte, image, son), en fait des outils particulièrement prometteurs pour la robotique moderne. Ces modèles peuvent agir comme une interface à la fois ouverte et contextuellement informée entre des utilisateurs non-experts et des systèmes robotiques complexes.

L'Institut des Systèmes Intelligents et de Robotique (ISIR) développe actuellement des robots manipulateurs mobiles qui intègrent différents travaux du laboratoire sur la saisie d'objets, leur manipulation et sur l'interaction sociale. Ces développements intègrent déjà des modèles de fondation (LLM, VLM - Large Language Models et Vision-Language Models) pour exploiter leur capacité à traiter le langage naturel avec un vocabulaire ouvert.

A.1.2 Objectif général

Ce projet s'inscrit dans cette dynamique et vise le développement d'une interface homme-machine acceptant des instructions vocales pour un robot manipulateur mobile équipé d'une architecture cognitive basée sur le langage. L'objectif principal est de permettre une interaction naturelle et intuitive entre les utilisateurs et le robot, en utilisant la voix comme modalité d'entrée principale.

A.2 Description des besoins

A.2.1 Besoins fonctionnels

Le laboratoire ISIR exprime les besoins fonctionnels suivants :

— Réaliser une transcription précise de la parole en texte, même dans des environnements avec bruit ambiant

— Intégrer ces transcriptions comme une prompt vers un LLM
— Implémenter un mécanisme d'interaction avec un LLM permettant à l'utilisateur d'interrompre une actions en cours ou ajouter des information complémentaires
— Développer une mécanisme pour réaliser l'interaction avec LLM en synchrone et asynchrone
— Implémenter un système de feedback vocal pour que le robot puisse confirmer sa compréhension et communiquer ses réponses

A.2.2 Besoins non-fonctionnels

Les besoins non-fonctionnels identifiés sont :

— Performance : temps de réponse rapide pour assurer une interaction fluide
— Portabilité : assure un bon compromis entre performance et taille du modèle, favorisant à la fois la précision et la facilité de déploiement
— Robustesse : fonctionnement dans des environnements avec bruit variable
— Maintenabilité : code bien documenté et modulaire pour faciliter les évolutions futures

A.3 Contraintes

A.3.1 Contraintes temporelles

La planification dans un premier temps est le suivant :

— Phase 1 (2 semaines) : Mise en place de l'environnement de développement, et premiers tests avec Faster-Whisper
— Phase 2 (1 mois) : Développement du module de reconnaissance vocale et intégration avec les modèles de langage
— Phase 3 (2 mois) : Implémentation du mécanisme d'interaction
— Phase 4 (2 semaines) : Tests, validation, et documentation

A.3.2 Contraintes matérielles

— Puissance de calcul : contraint par les capacités de calcul des machines personnelles, les différents outils devront être exécutés en CPU
— Compatibilité : le projet devra être compatible avec les systèmes opératifs de Windows et Apple (au moins)
— Robots : des tests avec des bras robotiques disponibles à l'ISIR seront envisageables dans des créneaux limités

A.3.3 Proposition d'outils

Aucune autre contrainte (technique ou logicielle) n'est imposée par les encadrants, mais des propositions sont faites :

— Reconnaissance vocal : le module Faster-Whisper (https://github.com/SYSTRAN/faster-whisper) sera utilisé pour la transcription de la parole en texte

— Modèles de langage : le framework Ollama sera utilisé pour tester et exécuter des petits modèles de langage (LLM) localement.
— Compatibilité OS : le software Docker permettra de créer, tester et déployé les modules du projet indépendamment du système opératif utilisé pour les exécuter

A.4 Solution proposée

A.4.1 Module de speech-to-text (STT)

Ce module utilisera Faster-Whisper pour transformer les commandes vocales en texte. Il définira une méthode responsable de la transcription ainsi que des méthodes outils pour réaliser de test de performance. Plus spécifiquement, les tests réalisés pour ce module seront :

— Tests préliminaires simples (reconnaissance des chiffres, heures, dates, homophones, etc.)
— Tests avec différents niveaux de bruits ambiant (faible, moyen, élevé)
— Test pour évaluer la performance (temps d'exécution, précision)

A.4.2 Module d'intégration de LLM

Ce module s'appuiera sur des modèles de langage déployés via Ollama. Il sera responsable de :

— La gestion du contexte conversationnel
— La génération de la réponse à une commande
— La définition du format de réponse à générer
— La demande de clarifications en cas d'ambiguïté

Les tests pour ce module incluront :

— Tests avec des commandes simples et bien structurées
— Tests avec des commandes ambiguës ou incomplètes
— Tests utilisant le protocole Code As Policy ou RAG pour se rapprocher au contexte d'application sur un robot assistant

A.4.3 Module d'interaction

Ce module assurera la communication entre le système de reconnaissance/interprétation et l'architecture cognitive du robot. Il devra :

— Permettre l'insertion de la transcription vocal en temps réel
— Gérer le retour de LLM
— Assurer un suivi de l'état d'exécution des actions
— Fournir un retour sur l'exécution des commandes
— Définir le mécanisme d'écoute de LLM

Pour vérifier le fonctionnement du module, on fera de :

— Tests de bout en bout simples (commande vocale jusqu'au retour de LLM)
— Tests de séquences d'ordres synchrones
— Tests de commandes asynchrones (interruptions ou modifications d'ordres en cours)

A.4.4 Module de text-to-speech (TTS)

Pour permettre au robot de communiquer verbalement, un module de synthèse vocale sera implémenté en utilisant des bibliothèques TTS, en principe nous envisageons d'utiliser la librairie gTTS (Google Text-To-Speech). Ce module permettra transformer la réponse textuelle de LLM en audio et le retransmet à l'utilisateur.

A.4.5 Extensions possibles

Si le temps le permet, les extensions suivantes pourront être envisagées :

— **Interface graphique complémentaire** : Développement d'une interface visuelle permettant au robot de communiquer ses intentions de manière passive.

Annexe B

Manuel utilisateur

B.1 Structure

B.1.1 Exécutables

— `whisper_processor.py` : chargé de faire les transcriptions, on retrouve aussi les tests de performance des différents modèles de Faster-Whisper.
— `llm_integration.py` : on retrouve les tests d'interaction simples avec ollama ainsi que les tests *Code as Policy* et *RAG*.
— `main.py` : responsable de faire les tests d'interaction synchrone et asynchrone.

B.2 Utilisation

B.2.1 Faire des transcriptions

Dans le main du fichier `whisper_processor.py`, il est possible d'obtenir la transcription d'un des fichiers audio disponibles dans le répertoire samples. Décommentez la ligne de code suivante en choisissant le modèle de Faster-Whisper :

```
fw_model_size = "large-v3"
getTranscript("samples/juin.m4a", fw_model_size, record=False)
```

Ceci permettra de faire la transcription du fichier audio `juin.m4a` avec le modèle "large-v3". Il est aussi possible de traiter l'audio et obtenir le score de la transcription à travers la fonction `getScore()`.

B.2.2 Pour lancer Ollama

Il est nécessaire de faire un pull sur le modèle `3.2:3b` et `mxbai-embed-large` en faisant sur terminal :

```
ollama pull <model name>
```

B.2.3 Faire les tests avec Ollama

Pour faire le test *Code as Policy* ou *RAG* il suffit de décommenter la fonction correspondante dans le main du fichier `llm_integration.py`.

```
if __name__ == "__main__":
    # test_RAG()
    # test_CAD()
```

B.2.4 Pour exécuter l'interaction avec Ollama

Lancez le fichier `main.py` pour tester l'interaction en temps réel avec Ollama. Le premier enregistrement réalisé en cliquant sur la touche b enregistre la requête. Puis tous les enregistrements qui suivraient permettraient de donner à Ollama des informations supplémentaires.

Disclaimer : Il est nécessaire de donner la permission à VSCODE (ou votre IDE de préférence) d'accéder au clavier.

B.3 Répertoires

Le répertoire samples contient les fichiers audio utilisés pour les tests faits. Le répertoire samples/withNoise contient plus spécifiquement les fichiers audio avec les différents niveaux de bruit (%) sous le format `fileName-noiseLevel.mp3`.

Puis les données générées par les tests sont sauvegardées dans les répertoires 'transcriptions' et 'data' selon le modèle de Faster-Whisper utilisé.

```
./
 src/
    main.py
    whisper_processor.py
    llm_integration.py
 data/
    tiny/
       exec_time.txt
       comparison_exec_time.txt
    small/
       exec_time.txt
       comparison_exec_time.txt
    medium/
       exec_time.txt
       comparison_exec_time.txt
    large-v3/
        exec_time.txt
        comparison_exec_time.txt
 transcriptions/
    tiny/
    small/
    medium/
```

```
   large-v3/
samples/
   withNoise/
   audio files (.m4a, .mp3)
code_as_policy.txt
README.md
```

B.4 Dépendances

— faster-whisper
— ollama
— pyaudio
— numpy
— matplotlib

Annexe C

Figures

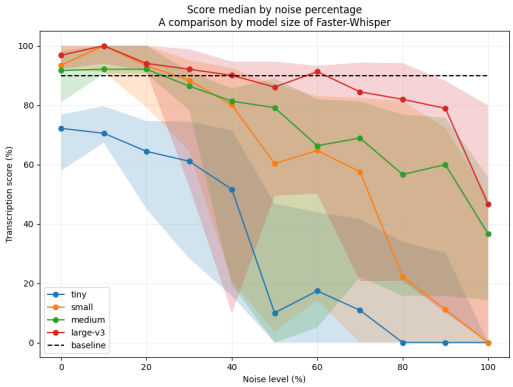

FIGURE C.1 – Comparaison des taux d'erreur de transcription (WER) pour quatre modèles de Faster-Whisper (tiny, small, medium, large-v3) en fonction du niveau de bruit (0-100%). Chaque courbe représente la médiane calculée sur 10 échantillons, avec les intervalles interquartiles (q1-q3). La ligne en pointillés indique le seuil minimal de WER (baseline) pour une transcription fiable

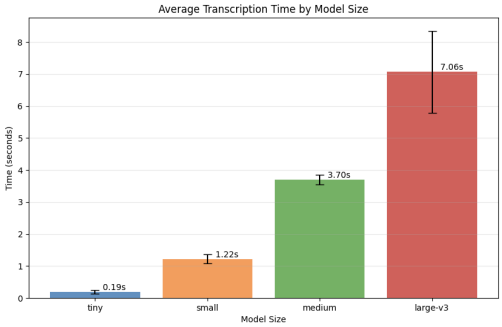

FIGURE C.2 – Temps moyen de transcription pour différents modèles de Faster-Whisper (tiny, small, medium, large-v3). Les barres représentent la moyenne du temps d'exécution de la méthode `getTranscript` calculée sur 118 échantillons audio, avec des barres d'erreur indiquant l'écart-type (variabilité).