Projet ingénierie et entreprise

Dassault Aviation : Assistant virtuel pour les opérations cockpits civils

Domaine : Neuro-ergonomie et Intelligence Artificielle

Rapport

Élève(s) :
Fiona Benedito
Dinh-Viet-Toan Le
Claire Raulin
Maxime Sabbah

Encadrant(s) :
Hervé Girod
Timon Ther
Antoine Casta
Caroline Chanel

6 mars 2022

Table des matières

Résumé **2**

1 Introduction **3**
1.1 Problématique générale . . . 3
1.2 Objectifs du projet . . . 3
1.3 Parties prenantes et attendus . . . 3

2 Partie gestion de projet **4**
2.1 Equipe et méthode de travail . . . 4
2.1.1 Présentation de l'équipe . . . 4
2.1.2 Stratégie de travail . . . 4
2.1.3 Définition des jalons . . . 4
2.2 Décomposition du produit et des activités . . . 5
2.2.1 PBS et WBS . . . 5
2.2.2 Organisation de l'équipe . . . 6
2.3 Organisation temporelle du projet . . . 7
2.3.1 Planning . . . 7
2.3.2 Budget prévisionnel . . . 8
2.3.3 Plan de charge . . . 9

3 Contexte **10**
3.1 État de l'art . . . 10
3.1.1 Description du PIE précédent . . . 10
3.1.2 Compréhension du langage naturel . . . 12
3.1.3 Intégration de données dynamiques via API . . . 12
3.2 Intérêt d'un assistant virtuel pour pilotes . . . 13
3.2.1 Questionnaire : exploitation des données en temps réel . . . 13
3.2.2 Retours sur la maquette qu'on veut réaliser . . . 14

4 Développement et intégration des solutions retenues **17**
4.1 Enrichissement de l'ontologie . . . 17
4.1.1 Données statiques du monde réel . . . 18
4.1.2 Ontologie et données dynamiques . . . 18
4.2 Intégration des données dynamiques . . . 18
4.2.1 Communication avec les API . . . 19
4.2.2 Représentation des données dynamiques . . . 19
4.3 Requêtes . . . 21
4.3.1 Reconnaissance du langage naturel . . . 21
4.3.2 Traitement des requêtes . . . 22
4.4 Architecture de l'application . . . 23

5 Utilisation de l'assistant de vol virtuel **26**
5.1 Performances de la maquette . . . 26
5.1.1 Tests internes . . . 26
5.1.2 Retours pilotes sur la maquette réalisée . . . 26
5.1.3 Entretiens individuels pilotes . . . 28

5.2 Pistes d'amélioration 28
5.2.1 Potentielle suite de projet 28

6 Conclusion **29**

Références **30**

Annexes **31**
6.1 Ontologie 31
6.2 Requêtes disponibles 32
6.3 Intentions implémentées pour le NLU 33
6.4 Liste des questions pour tester la maquette 34

Résumé

L'aviation civile se dirige aujourd'hui vers les cockpits monopilotes, dans lesquels le pilote peut se retrouver, seul, face à une charge de travail importante. Dans ce cadre-là, Dassault Aviation cherche à développer un assistant de vol permettant au pilote de requêter diverses informations, telles que l'aéroport le plus proche ou la météo à cet aéroport.

Notre projet se base sur de précédentes itérations qui ont abouti sur une maquette reconnaissant une requête annoncée à l'oral en langage naturel, et exploitant une ontologie pour pouvoir y répondre et fournir un retour audio à l'utilisateur.

Notre projet enrichit cette ontologie et le panel de requêtes disponibles avec des données provenant du monde réel, ainsi que des données de trafic et météo en temps réel. La maquette implémentée repose donc sur une application web pouvant mettre en relation une ontologie et des services API extérieurs, et implémentant des modules de reconnaissance vocale et du langage naturel. Les performances quantitatives de la reconnaissance vocale atteignent 70% de requêtes bien comprises, et nous obtenons des retours qualitatifs positifs concernant l'utilisation de la maquette .

Mots clés – Assistant virtuel, Traitement du Langage Naturel, Ontologie, API, Requêtes.

1 Introduction

1.1 Problématique générale

Le pilotage d'un avion qu'il soit commercial, cargo, civil ou militaire est une tâche qui demande une charge mentale très élevée et de savoir acquérir, traiter et prioriser un grand nombre de données. Il est donc envisageable de créer un assistant virtuel afin d'aider les pilotes à obtenir des informations sur leur vol en temps réel le plus rapidement possible. Dans le cadre des développements de cockpits pour vols monopilotes civils, Dassault Aviation cherche à développer un assistant de vol virtuel pour leur gamme Falcon Jet afin d'aider et d'accompagner le pilote lors de phases de vol particulièrement chargées : déroutement, panne d'un moteur, panne d'une gouverne, etc... Cet assistant permettrait donc au pilote d'obtenir les informations nécessaires à la poursuite de son vol : aéroports les plus proches, longueur nécessaire d'atterrissage recalculée en cas de panne d'une gouverne, météo etc.

En théorie, pour pouvoir être implémenté dans un avion civil, l'assistant pilote virtuel doit pouvoir communiquer en temps réel avec le pilote, c'est-à-dire comprendre ses questions et y répondre précisément. Lors de précédents PIE [1] une maquette a été réalisée, celle-ci fonctionne pour un plan de vol précis déjà pré-enregistré.

1.2 Objectifs du projet

Alors que précédent PIE [1] se concentrait sur un plan de vol précis et déjà pré-enregistré, nous avons pour objectif de pouvoir suivre un vol en temps réel afin de rendre la maquette dynamique. Pour cela, nous devons enrichir le nombre de scénarios de vols accessibles à l'aide de l'API FlightRadar24 et sélectionner les données qui seront pertinentes pour le pilotage. En conséquence, nous devrons enrichir la base de données ontologique et rajouter des concepts. Nous devrons améliorer en parallèle la base de données utilisées par la bibliothèque "Snips NLU" pour la partie Natural Language Processing (NLP).
Un des autres objectifs est de rendre l'Interface Homme-Machine (IHM) plus ergonomique notamment en y intégrant un bouton "push-to-talk", ce qui permettrait aussi d'améliorer le temps de réponse de notre maquette.
L'équipe de Dassault Aviation qui nous encadre nous offre la possibilité de proposer toute piste d'amélioration de la maquette qui nous semble pertinente. Nous avons donc pris l'initiative de rajouter une base de données météo en temps réel ainsi que les fréquences radios des aéroports.
Enfin, tout ceci doit être corrélé avec une étude de terrain auprès de pilotes confirmés et en formation afin de coller au plus proche de leurs attentes lors des différentes phases de vol et situations auxquelles ils pourront être confrontés dans un cockpit monopilote.

1.3 Parties prenantes et attendus

Notre PIE sera encadré par quatre personnes se partageant les différentes responsabilités.
Hervé Girod et Timon Ther (respectivement ingénieur système senior et expert data scientist chez Dassault Aviation) représentent la partie client du projet. Après avoir définies

et étudiées ensemble leurs attentes, nous leur ferons part de nos avancées et les consulterons sur les principaux choix techniques. Lors d'une réunion de cadrage de projet, nous avons convenu de deux livrables, le premier étant ce rapport, le second étant la maquette d'assistant virtuel dont le développement est détaillé ci-dessous.
Pour la partie gestion de projet, nous serons accompagnés par Antoine Casta qui nous aidera à définir clairement le plan de développement et nous aider à nous organiser de manière optimale de façon à mener à bien le projet. Nous devrons lui fournir le plan de développement mis à jour après la réunion de cadrage de projet, la réunion à mi-projet ainsi que celle en fin de projet. Il aura aussi accès à ce rapport et à la maquette d'assistant virtuel.
Notre référente à l'ISAE-SUPAERO est Caroline Chanel, qui pourra nous aider notamment pour la partie intelligence artificielle ou nous mettre en relation avec les personnes compétentes dans le domaine recherché.
Enfin, dans un objectif de prospection, notre maquette servira de démonstration dans le cadre de vols civils : d'abord pour des ingénieurs Dassault Aviation pour différents cas de vols, puis dans un second temps, auprès des pilotes afin de connaître leur ressenti.

2 Partie gestion de projet

2.1 Equipe et méthode de travail

2.1.1 Présentation de l'équipe

Le projet que nous allons mener se divise en plusieurs grandes parties, une partie consistera en l'enrichissement des bases de données à l'aide de l'API FlighRadar et l'ajout d'une base de données météo, une autre l'amélioration de la base de données de language naturel "Snips NLU" et la dernière consistera à échanger avec des pilotes tout au long de la conception de la maquette afin de savoir quelles données exploiter et améliorer l'ergonomie de l'IHM.

— Fiona Benedito : Cheffe de projet, responsable Speech Recognition et NLU
— Dinh-Viet-Toan Le : Responsable développement, données temps réel et ontologie
— Claire Raulin : Responsable requêtes, tests et Interface Homme-Machine (IHM)
— Maxime Sabbah : Responsable collaboration avec les pilotes

2.1.2 Stratégie de travail

Nous avons fait le choix d'une méthode de gestion de projet traditionnelle. Le projet est divisé en tâches et sous-tâches qui sont interdépendantes et doivent donc être réalisées dans un ordre précis. Compte tenu de la grande incertitude sur les difficultés rencontrées dans les phases de programmation, de la potentielle latence des retours des pilotes, il sera important d'actualiser régulièrement le plan de développement.

2.1.3 Définition des jalons

Les jalons que nous avons respecté au cours du développement de notre projet sont les suivants :
— 11 Décembre 2021 : Première revue côté gestion de projet ;

— 11 Janvier 2022 : Réunion de mi projet ;
— 31 Janvier 2022 : Deuxième revue côté gestion de projet ;
— 5 Mars 2022 : Remise du rapport, de la maquette et de son guide utilisateur ;
— 14 Mars 2022 : Soutenance.

Tout au long de cette phase de développement, nous avons eu des réunions bi-mensuelles avec le client afin de le tenir au courant de notre avancement et répondre au mieux à ses attentes. Nous avons pu échanger sur les différentes pistes d'amélioration que nous envisagions au fur et à mesure de nos avancées.

2.2 Décomposition du produit et des activités

2.2.1 PBS et WBS

Ci-dessous le PBS, dont les feuilles en rose sont les composants ajoutés au précédent prototype (issus du PIE de l'année dernière). Les autres éléments sont les composants déjà présents sur le prototype et sur lesquels nous avons apporté des améliorations plus ou moins importantes.

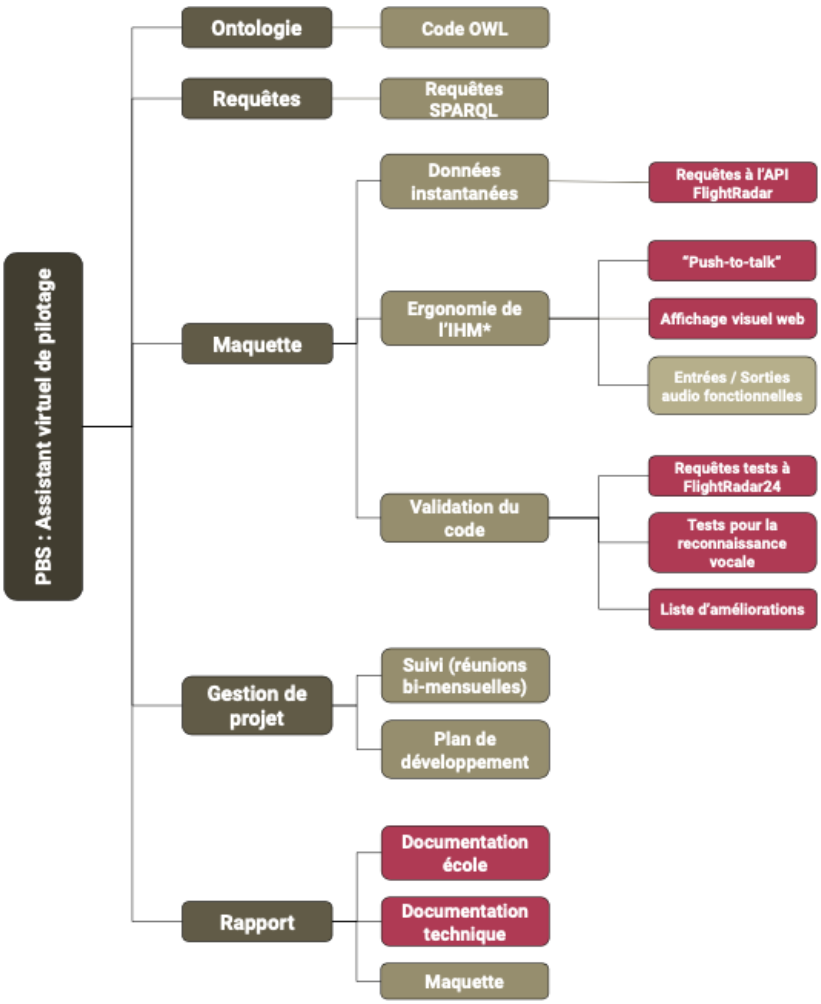

FIGURE 2.1 – Product Breakdown Structure

De cet arbre découle le WBS, qui contient toutes les tâches réalisées lors du projet, en

cohérence avec le planning qui détaille la répartition du travail. Les tâches sont regroupées par working package (WP), qui sont les différentes grandes parties du projet.

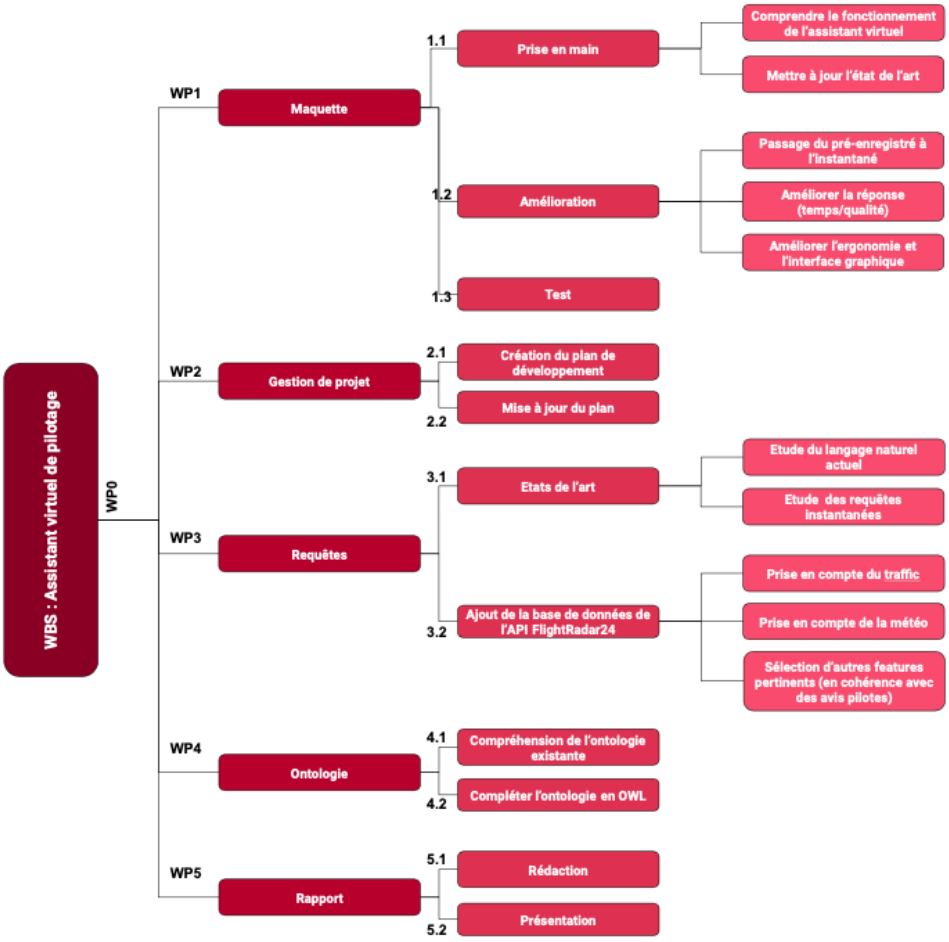

FIGURE 2.2 – Work Breakdown Structure

2.2.2 Organisation de l'équipe

En fonction de la décomposition de notre projet en tâches plus ou moins dépendantes les unes des autres nous avons décidé de gérer en parallèle les parties développement de la maquette, prise de contact avec les pilotes et la partie gestion de projet. Chaque membre du groupe sera affecté sur une tâche principale et assistera un autre membre de l'équipe sur une tâche annexe. Pour la tâche numéro 5, c'est-à-dire la rédaction du rapport et la présentation de notre projet, nous avons décidé de ne pas désigner de responsable. Tout ceci est visible sur la matrice RACI ci dessous.

Tâches	Claire RAULIN	Maxime SABBAH	Dinh-Viet-Toan LE	Fiona OLIVEIRA BENEDITO	Hervé GIROD	Timon THER
WP 2.1 - Création du plan de développement	R	R	R	A	I	I
WP 2.2 - Mise à jour du plan de développement	R	R	R	A	I	I
WP 1.1 - Prise en main de la maquette	R	A	R	R	C	C
WP 1.2 - Amélioration de la maquette	A	C	R	R	C	C
WP 1.3 - Tests et tests facteurs humains	A	R	C	R	C	C
WP 3.1 - États de l'art	R	R	R	A	C	C
WP 3.2 - Ajout de la base de données de l'API FlightRadar	R	C	A	C	C	C
WP 4.1 - Compréhension de l'ontologie en OWL	R	A	R	C	C	C
WP 4.2 - Complétion de l'ontologie en OWL	C	R	A	C	C	C
WP 5.1 - Rédaction du rapport	R	R	R	R	I	I
WP 5.2 - Présentation	R	R	R	R	I	I

Légende :

(R) Responsible ; (A) Accountable ; (C) Consulted ; (I) Informed

FIGURE 2.3 – Matrice RACI

2.3 Organisation temporelle du projet

2.3.1 Planning

Nous avons établi le planning au début du projet, en essayant d'anticiper la durée des différentes tâches et l'interdépendance de certaines tâches. Mais au cours du projet nous avons mis à jour le planning qui a changé, d'une part pour la durée des tâches mais aussi pour l'ordre et la répartition. Le budget prévisionnel de la section suivante permettra de mettre des chiffres sur ces différents changements.

Globalement, le temps nécessaire à la gestion de projet a été sous-estimé au début. Cette partie du projet permet de cadrer le travail, en imposant des jalons et des dates à respecter - telles que les revues de décembre et janvier. D'autre part, la plupart des tâches sont corrélées mais réalisables en parallèle. Ainsi nous avons pu travailler sur le code Python, l'ontologie et les requêtes simultanément bien que ces tâches soit interdépendantes. Cela nous a permis de répartir les compétences utilisées entre les membres de l'équipe. Enfin nous pensions finir les parties techniques plus tôt et passer sur la finalisation du projet, mais nous avons pris un peu de retard au cours du projet (que nous avons rattrapé sur la fin).

Ainsi, nous avons obtenu le planning final suivant :

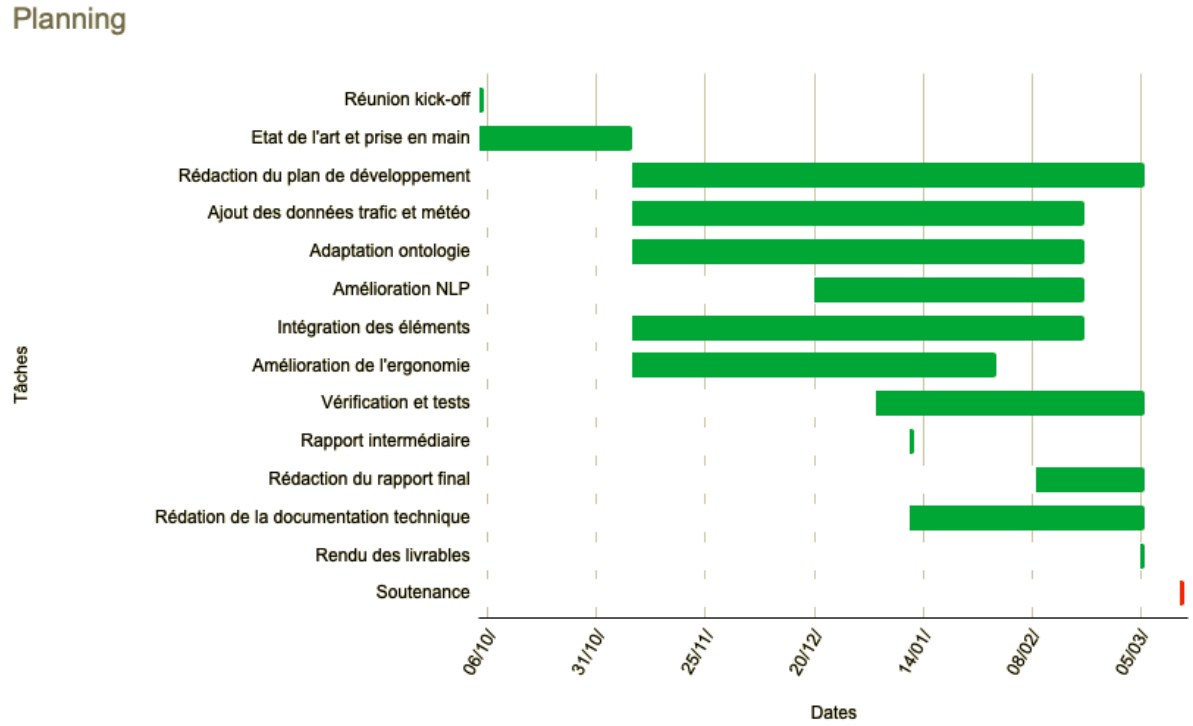

FIGURE 2.4 – Planning réel du projet

2.3.2 Budget prévisionnel

Le projet ingénierie et entreprise de l'ISAE-SUPAERO représente 5 crédits dans notre année, c'est-à-dire 120 heures par personne en théorie. En pratique nous avons 80 heures allouées par personne pour ce projet au cours de l'année, soit 320 heures. Nous avons donc travaillé chez nous en parallèle ce qui offre plus de souplesse sur les temps et répartition du travail. Nous sommes donc partis d'une estimation initiale de 224 heures assez faibles et avons vite ré-estimé le temps nécessaire qui était bien plus élevé.

La répartition des heures est présentée dans le tableau ci-dessous, qui permet de comparer les estimations et temps effectifs du projet. Une telle différence sur un projet coûteux aurait été problématique, mais en considérant notre projet dont le seul coût est le temps, cela n'a pas posé de vrai problème. Nous avons désormais plus d'outil pour estimer le coût d'un projet et une tendance à sous-estimer globalement les tâches en ressort. Nous avions décidé un budget prévisionnel au début du projet en tâchant d'estimer les nombres d'heures sans vraiment d'indice. La révision de mi-Décembre nous a permis de trouver de gros écarts pour certaines tâches, notamment les tâches techniques. Ce tableau nous a bien servi au cours du projet pour prendre conscience de la nécessité de passer d'une tâche à l'autre ou de paralléliser pour ne pas se retrouver à la fin du projet avec des tâches non-réalisées.

Tâches	Prédiction budget	Budget réel
Maquette		
Comprendre le fonctionnement de l'assistant virtuel	20H	30H
Mettre à jour l'état de l'art	10H	10H
Passage du pré-enregistré à l'instantané	20H	30H
Améliorer la réponse (temps/qualité)	5H	20H
Améliorer l'ergonomie et l'interface graphique	20H	20H
Tests	6H	12H
Sous-total	*81H*	*122H*
Gestion de projet		
Création du plan de développement	20H	20H
Mise à jour du plan	5H	20H
Sous-total	*25H*	*40H*
Requêtes		
Etude du langage naturel actuel	8H	8H
Etude des bases de données et requêtes instantanées	12H	12H
Prise en compte du trafic	25H	25H
Prise en compte de la météo	5H	20H
Sélection d'autres features pertinents	10H	40H
Sous-total	*60H*	*105H*
Ontologie		
Compréhension de l'ontologie existante	12H	12H
Compléter l'ontologie en OWL	16H	10H
Sous-total	*28H*	*22H*
Rapport		
Rédaction	20H	40H
Présentation	10H	20H
Sous-total	*30H*	*60H*
Total	**224H**	**359H**

2.3.3 Plan de charge

Nous avons établi le planning et le budget prévisionnel prévu pour chaque tâche, ce qui nous permet maintenant de représenter le plan de charge jusqu'à la livraison des livrables au client. La charge de travail sera répartie de façon croissante, avec notamment plus d'heures en janvier et février pour la fin de développement de la maquette, les tests ainsi que la rédaction de ce rapport.
Nous avons remarqué au fur et à mesure du projet, en particulier lors de la revue de mi projet que nous avions sous estimé le nombres d'heures dont nous aurions besoin pour mener à bien ce projet. À noter que seule la charge de travail effectuée en mars reste encore une charge estimée.

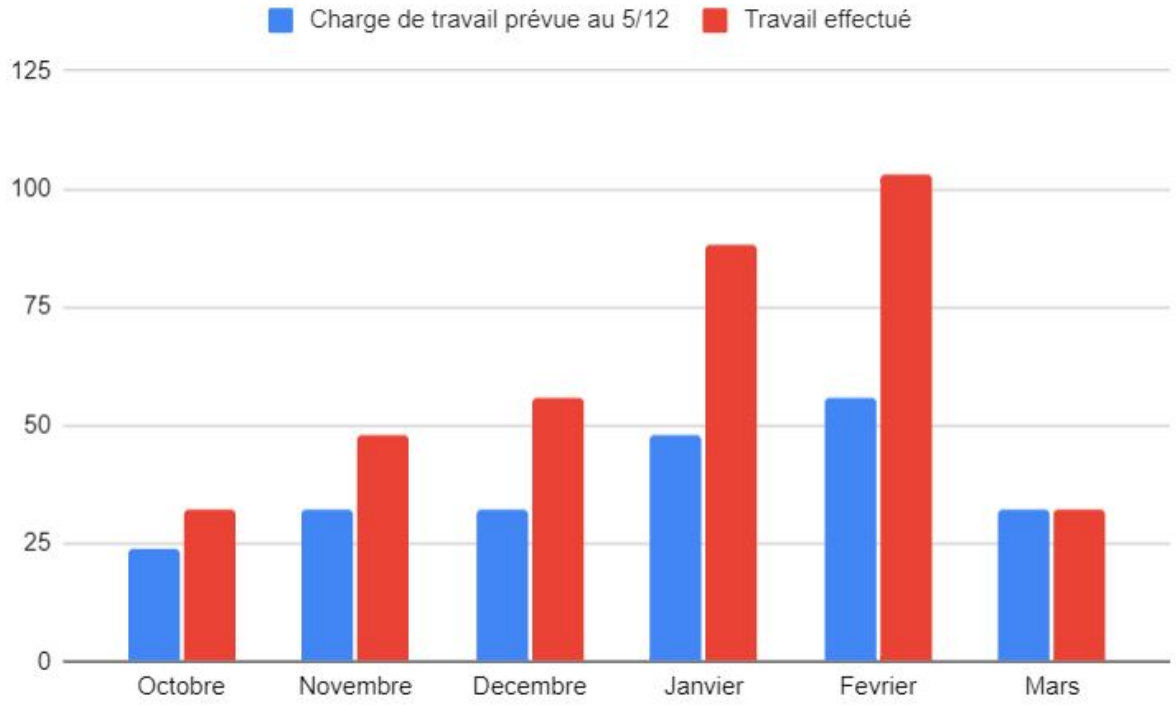

FIGURE 2.5 – Plan de charge

3 Contexte

3.1 État de l'art

3.1.1 Description du PIE précédent

Ce PIE fait suite à un PIE précédent réalisé en 2020-2021 par un groupe d'étudiants de l'ISAE-SUPAERO composé de Côme Lavisse, Daniela Andrioaie, William Angelier et Antoni Garzon Joli également pour le compte de Dassault Aviation. Leur PIE prenait aussi la suite d'un projet antérieur qui avait fourni un premier prototype de maquette d'agent conversationnel et d'ontologie pouvant être requêté par écrit.

Face aux limitations auxquelles se heurtaient ce prototype, l'objectif de leur PIE était de fournir une nouvelle maquette dont les requêtes pourraient être formulées à l'oral en anglais et en langage naturel et dont l'ontologie serait adaptée à des nouvelles requêtes pouvant être implémentées. Du point de vue technique, les fonctionnalités implémentées ont été les suivantes :

Pour l'ontologie L'objectif principal était d'élargir l'ontologie. Pour cela, certains scénarii ont été ajoutés. Le premier était le déroutement car c'est une phase à forte charge mentale, stressante et très demandeuse en termes de concentration, qui devait être traitée efficacement pour aider le pilote de manière optimale. Il fallait que la maquette fournisse un rendu visuel clair sur les terrains proches et l'état de la piste en fournissant le METAR, TAF, fréquence de l'ATIS, NOTAM et orientation et prenant en compte d'éventuelles pannes pour les divers calculs de distance d'atterrissage. Dans les faits, seul le déroutement propre à l'aviation d'affaire a été implémenté et la maquette fournit après

une requête à l'oral pour chaque aéroport, les horaires d'ouvertures de l'aéroport et du contrôle aérien, le carburant disponible au sol, la largeur des taxiways et sa capacité de handling.

Le projet a aussi permis la mise en place de checklists review que le pilote peut appeler directement. L'assistant renvoie le premier mot de la checklist que le pilote doit répéter pour amorcer le processus, puis répéter tous les items pour que la checklist soit aboutie.

Enfin, la maquette met en place la possibilité d'établir le CPDLC qui consiste à mettre en relation le pilote et un contrôleur aérien. Ici, le pilote peut requêter la mise en contact avec un contrôleur de n'importe quel aéroport à proximité, faire sa requête qui sera transmise par l'IHM au contrôleur. Il faut encore implémenter la possibilité de recevoir la réponse du contrôleur.

Pour la programmation en langage naturel (NLP) La maquette reconnaît le langage naturel et y répond en utilisant la bibliothèque Snips NLU pour cela, avec des résultats très satisfaisants. Au départ, "trop" de libertés furent données dans la formulation des requêtes, ce qui portait à confusion dans les réponses que la maquette donnait. Le choix de normaliser le format des requêtes a été retenu pour éviter cet écueil.

De plus, la reconnaissance vocale utilise les librairies PortAudio, PyAudio et SpeechRecognition pour traiter les entrées et sorties audio en temps réel, faire les liens avec les entrées sur l'ordinateur et le code et s'occuper du traitement du signal vocal pour le transformer en chaîne de caractère respectivement. Enfin, pour assurer la robustesse de la solution, le choix d'énoncer le nom complet de l'aérodrome plutôt qu'une abréviation a été retenu.

Pour l'interaction pilote-assistant Il a été décidé d'ajouter une réponse vocale à l'assistant dans certaines situations, notamment au cours des checklists ou lors d'une demande de déroutement.

Enfin l'architecture finale de la solution était la suivante :

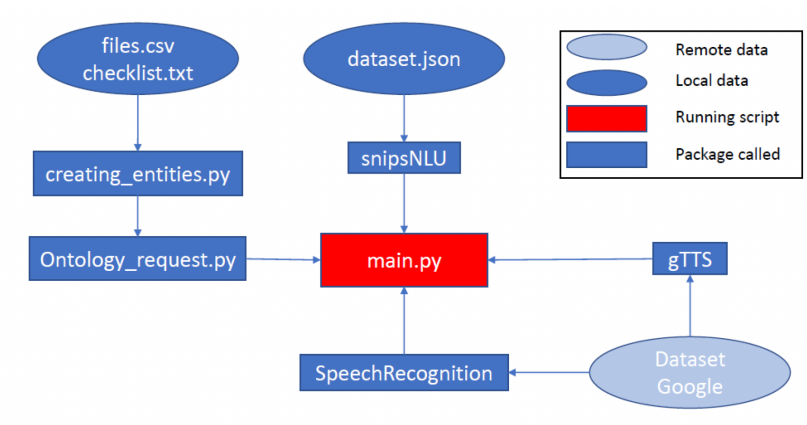

FIGURE 3.1 – Architecture de la solution 2020. Figure extraite de [1].

En ce qui concerne les performances de la maquette, elle a eu de bons retours d'utilisation par des pilotes avec des temps d'exécution raisonnables sans être "trop" gourmand en ressources. Néanmoins certaines pistes d'amélioration comme l'amélioration de l'interface

graphique, l'ajout de nouvelles requêtes, l'enrichissement de l'ontologie, l'amélioration de l'ergonomie d'utilisation et l'ajout d'un bouton push-to-talk ont été soulevées.

3.1.2 Compréhension du langage naturel

Comme expliqué précédemment, la maquette reconnaît le langage naturel et y répond grâce à la bibliothèque Snips NLU. Cette bibliothèque a été choisie lors du PIE précédent pour diverses raisons : il s'agissait de faire un compromis entre performance, rapidité de la réponse, facilité d'utilisation de l'assistant et objectif.

En effet, il s'agissait pour nous de rendre la maquette plus réaliste en permettant une liberté sur l'expression des requêtes orales, tout en conservant un assistant réactif avec des réponses rapides (utilisation instantanée). Ainsi le choix du PIE précédent d'utiliser Snips NLU est le plus judicieux. D'autre part il n'y a pas eu d'évolution majeure sur les bibliothèques en Python de NLU. Ce sont les bibliothèques de NLP générales qui ont évolué, bibliothèques extrêmement coûteuses en temps et puissance. Il n'était donc pas judicieux d'explorer une autre bibliothèque (typiquement Rasa NLU qui fonctionne de manière très similaire à Snips NLU).

Dans notre cas, le sujet étant très spécifique et les requêtes finalement assez limitées par les données utilisées, il est possible de compléter la base de données Snips NLU à la main. Ainsi nous avons rajouté les termes techniques, aéroports et de nombreux synonymes dans la base de données. D'autre part une bibliothèque trop complète est aussi plus propice aux erreurs dans un domaine d'utilisation plutôt normalisé. Dans notre cas, la maquette a été optimisée pour reconnaître certains bouts de mots incomplets par exemple.

Finalement, une modification du code brut avec des ajouts ponctuels de vocabulaire est la solution que nous avons retenu et qui nous a donné des résultats concluants.

3.1.3 Intégration de données dynamiques via API

La communication avec des API (*Application Programming Interface*) extérieures est également nécessaire afin d'inclure des données en temps réel et provenant du monde réel. En particulier, nous nous focalisons sur l'intégration de données de trafic et de météo.

Trafic aérien en temps réel. Il existe de nombreux sites présentant des données en temps réel sur le trafic aérien, notamment FlightAware, OpenSkyNetwork ou FlightRadar24. Ces données proviennent de récepteurs ADS-B (Automatic Dependent Surveillance-Broadcast) au sol permettant de réceptionner les données des vols dans un certain rayon autour de ceux-ci. De ce fait, les réseaux de ADS-B étant propres à chaque service (33.000 récepteurs pour FlightRadar24, 24.000 pour FlightAware), la couverture du trafic est différente selon le service utilisé. Par exemple, les activités d'aviation légères sont généralement absents dans OpenSkyNetwork, contrairement à FlightRadar24.

Ces sites proposent des API ouvertes, permettant à un programme de communiquer avec leurs services, et présentant des interfaces différentes. En l'occurrence, FlightAware et OpenSkyNetwork reposent sur une identification par clé (API Key), propre à chaque utilisateur, et possiblement payantes suivant l'usage commercial ou non du service. FlightRadar24 délivre ses données sur un flux continu de `.json` accessibles sans clé. Au-delà de l'interface avec le script, les données issues des différents services sont également plus ou moins complètes, notamment les informations de route souvent lacunaires sous OpenSkyNetwork.

Météo en temps réel. Il existe beaucoup d'API permettant d'obtenir des données météo, notamment OpenWeatherMap, WeatherAPI ou Weatherbit pour n'en citer que quelques unes, qui proposent plus ou moins des services similaires. Leurs données se basent sur diverses sources, dont des stations au sol, des satellites d'observation météorologiques, ou des services météo d'aéroport. Ces API proposent en plus d'autres services, tels que les prévisions ou des historiques météo. D'après leur documentation, les données météo sont accessibles en entrant n'importe quelle coordonnée géographique sur la Terre. Aussi, via un système de géocodage, elles sont également accessibles en entrant le nom d'une ville ou lieu dont on ignore les coordonnées géographiques.

Outre les paramètres météo, plusieurs services proposent également d'obtenir les METAR et TAF bruts en temps réel des aéroports, notamment AviationWeather REST API, CheckWX ou AviationAPI. Ces services nous donnent des informations sur les stations et fournissent des METAR brut, ainsi que leur équivalent sous forme de dictionnaire.

Pour tous ces services, leur accès se fait via des requêtes GET envoyées sur une route particulière, avec éventuellement une clé d'authentification.

3.2 Intérêt d'un assistant virtuel pour pilotes

3.2.1 Questionnaire : exploitation des données en temps réel

Il est important de sonder les utilisateurs potentiels de notre maquette. Pour cela, un questionnaire Google Form a été créé dans le but d'être envoyé à des pilotes, ou à des personnes ayant une vision de ce qu'est le pilotage pour recueillir des avis et des suggestions sur les éventuelles requêtes et fonctionnalités que nous devrions implémenter. Également, le but était d'obtenir un avis sur la manière de présenter les informations disponibles et sur l'interface graphique ainsi que de se projeter vers certains scénarios auxquels nous n'aurions pas forcément pensé et qui pourraient altérer la façon dont nous allions développer notre projet. Le questionnaire a été pensé comme suit :

Recueillir l'identité et l'expérience des participants. Les premières questions du questionnaire nous ont permis de récupérer les adresses mail des participants car nous voulons les recontacter par la suite pour qu'ils testent notre maquette ainsi que leur nom, prénom, métier et expérience de pilotage dans le but de situer notre panel de sondés.

Attester la pertinence du projet ainsi que les informations essentielles que l'agent doit fournir. Les questions suivantes demandaient au participants leur avis quant à l'utilité d'un tel agent conversationnel et de jauger la priorité à laquelle les informations fournies par FlightRadar24 et la météo devraient être implémentées. Nous leur demandions de les classer sur une échelle de 1 (pas indispensable) à 5 (incontournable). De plus pour les données FlightRadar, une question avec une liste exhaustive de paramètres (incluant aéroport le plus proche, altitude, vitesse verticale, vitesse sol, waypoint, état/largeur de la piste, fréquence d'aéroport, type d'avion ou la possibilité de rajouter une autre option "libre") leur a été posée.

Prise en compte des checklists pilote. De la même manière que précédemment, il leur été demandé de classer l'importance de la prise en compte des checklists sur une échelle de 1 à 5.

Ergonomie de l'agent et interface graphique. La suite du questionnaire portait sur l'importance qu'accordaient les pilotes à un éventuel retour visuel (notamment sur la compréhension de la requête par l'agent, et la réponse à cette requête) que pourrait fournir l'IHM. De plus, l'utilité de l'implémentation d'un bouton push-to-talk pour effectuer les requêtes leur a été questionnée.

Questionnements pratiques sur l'utilisation de l'agent La fin du questionnaire portait sur des interrogations "pratiques" que l'on se posait dans le design de la maquette. Était-il possible que les réponses aux requêtes et l'interaction homme/agent rentre en contact avec certaines tâches de pilotage et si oui, dans quelle mesure ? Enfin, nous leur laissions la liberté de nous indiquer d'éventuels scénarii auxquels nous n'aurions pas pensé, ou certains conditions pratiques d'utilisations que nous aurions pu omettre ainsi que la liberté d'exprimer toutes autres suggestions.

Dans sa globalité, le questionnaire se devait d'être suffisamment court pour maximiser le taux de réponse mais également suffisamment complet pour obtenir des résultats utiles.

3.2.2 Retours sur la maquette qu'on veut réaliser

Nous avons soumis le questionnaire précédent à plusieurs pilotes, allant de ceux qui pratiquent l'activité à l'ISAE-SUPAERO à des pilotes de ligne professionnels pour garantir un maximum d'ouverture sur les réponses possibles. Nous avons obtenu 11 réponses à notre questionnaire, ce qui nous a déjà permis d'affiner correctement la ligne directrice de notre projet ainsi que de déterminer les points essentiels à développer. Parmi ces onze participants, nous pouvions recenser les profils suivants :

Quel type de pilote êtes-vous et combien d'années d'expériences ?	
Pilote cadette Air France, toujours en formation	Étudiant pilote
En fin de formation	Pilote privé, 9 ans d'expérience
Débutant	Pilote de ligne A320 (débute bientôt)
Pilote en formation / 150h de vol	3 ans. 130 heures. Qualif pro en cours
PPL avion, hydravion et voltige, 4 années	

TABLE 1 – Métier et expérience des pilotes sondés.

Les réponses aux questions sont présentées dans l'ordre du questionnaire :

- Concernant l'utilité d'implémenter un tel agent, les réponses furent positives à hauteur de 63,6 %, ce qui atteste d'un besoin assez réel d'implémenter un agent conversationnel pour aider au pilotage.

- Concernant l'importance des informations FlightRadar24, les réponses données sont présentées en Figure 3.2. 81,8 % des sondés estiment l'importance de FlightRadar24 a au moins 3 sur 5, ce qui montre un besoin conséquent d'inclure ces données à la maquette. Vu la multitude de données qu'offre cet outil, il était important de sélectionner les plus pertinentes. Les participants ont estimé assez largement qu'il était indispensable d'avoir accès à l'aéroport le plus proche (90,8 %), l'altitude courante et l'état/largeur de la piste (63,6 %) mais aussi que les waypoints (0 %, possibilité de non compréhension de l'item par les participants) et le type d'avion (27,3 %) étaient largement dispensables . Parmi

les autres éléments, les avis étaient moins tranchés et les informations vitesses (sol et verticale) et fréquence de l'aéroport n'étaient plébiscitées que par environ 45 à 50 % des sondés. Nous savions donc qu'il était nécessaire que les requêtes fonctionnelles de la maquette devaient porter sur l'aéroport le plus proche, l'altitude courante et l'état de la piste. Un des pilotes a également signifié sa volonté d'inclure le METAR/TAF des aéroports à proximité.

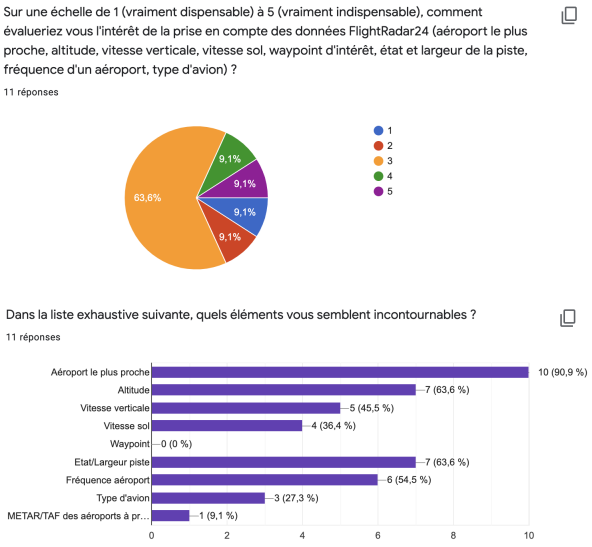

FIGURE 3.2 – a) Utilité des données FlightRadar24 b) Détails des données indispensables

- Concernant l'ajout des données météo, 100 % des sondés estiment son importance a au moins 4 sur 5 (avec 63,6 % qui la note à 5 !), ce qui montre la nécessité absolue d'inclure les données météo à la palette de l'agent conversationnel. C'est d'ailleurs l'un des point principal qu'ont soulevé les pilotes.

- Concernant les checklists pilotes, les avis étaient un peu plus diffus avec une forte majorité des personnes (45,5 %) qui estiment leur importance à 3 sur 5. Néanmoins, 27,3 % des participants estiment également son utilité à 5, ce qui nous conforte dans l'idée d'implémenter les checklists pilotes.

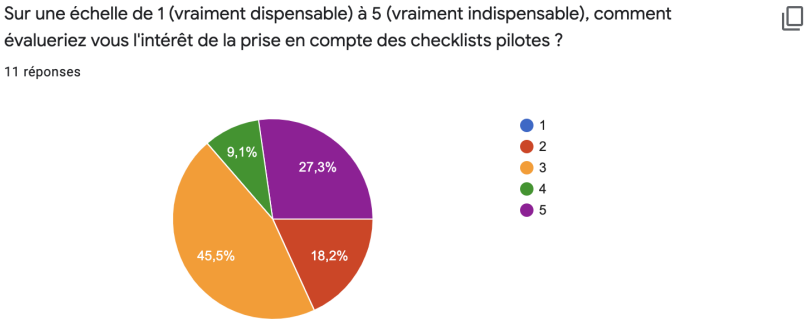

FIGURE 3.3 – Proportion des avis sur l'utilité des checklists

- Concernant l'ergonomie de l'IHM, 72,7 % des pilotes estiment qu'il est essentiel d'avoir un retour visuel de l'agent, notamment pour les réponses aux requêtes. Un pilote a aussi suggéré la possibilité d'ouvrir le QRH sur une checklist sur l'EFB (lors de checklists anormales par exemple). Également, 90,9 % des participants estiment que l'implémentation d'un bouton push-to-talk soit essentielle pour le bon fonctionnement du processus de requête.

- Aussi d'un point de vue pratique, une grande majorité des pilotes (81,8 %) craignaient que l'utilisation de l'agent conversationnel rentre en conflit avec les tâches de pilotages (notamment les communications radios). C'est également un point dont il faut tenir compte, si une mise en installation pratique d'un tel agent devait voir le jour (notamment dans un futur PIE ?).

- Enfin, certains pilotes ont proposé les réponses suivantes lorsque nous leur en laissions la liberté :

Des scénarios précis où l'utilisation de l'agent conversationnel aiderait énormément ?	
Lorsque la charge de travail est élevé en phase critique vol	Panne en l'air pour décharger des tâches au pilote
Annonces cabine et passagers	Ouverture d'une checklist anormale dans le QRH sur l'EFB (gain de temps), obtenir la météo de terrains environnants
Conscience situationelle, aéronef présent dans le secteur	
Lors d'une panne, cela permettrait de réduire le workload du pilote	En déroutement ou lors de phases critiques, où l'on a pas le temps de chercher l'info dans les documents de vol. Ça peut permettre de continuer à regarder dehors. Mais c'est surtout utile pour les pilotes seuls à bord, car à deux, c'est le travail du second pilote. Mais ça reste quelque chose d'envisageable.
Changement de fréquence Com/Nav. "GARMIN, mets 123.120 en com1 stby", dictée de plan de vol dans Garmin.	
Affichage de carte Vac, Aéroports le plus proche, Fréquences, Altitude tour de piste, Vent .	
Autres remarques ou commentaires que vous pourrez apporter ?	
Je pense que l'assistant vocal doit etre vraiment limité a des taches style radio, dictée de plan de vol (avant vol).	Pas vraiment compris la question sur Flight radar. Il me semble pas que ça soit la banque d'infos la plus pertinente. Le fait de pouvoir demander confirmation pour une fréquence radio et d'avoir un retour visuel en plus, ça semble plus utile !
Cependant, en se reposant trop sur l'assistant, les pilotes perdront en compétence ce qui est dangereux	
J'ai travaillé sur ce sujet pour intégrer une commande vocale dans le Rafale chez Thales si ça vous intéresse	La solution d'un push to talk avec l'affichage des informations sur un écran me semble pertinent, le feedback vocal me semble pouvoir rentrer en conflit avec les communications radio.

TABLE 2 – a) Scénarios possibles d'utilisation b) Commentaires quelconque additionnels

Globalement, les pilotes ont conforté les choix de design que nous avions pris en amont du développement du projet et ont apporté d'utiles suggestions quant à l'utilisation de l'agent et les données que nous devions implémenter de manière prioritaire. La météo, l'altitude et l'aéroport le plus proche furent les informations les plus demandées. L'ergonomie de l'agent était également estimé comme élément principal du bon fonctionnement de celui-ci. Néanmoins, il demeure certains pilotes qui ne considèrent pas particulièrement comme important la présence d'un tel agent, de par les interférences que celui-ci pourrait causer ou la possibilité de rajouter de la charge mentale et de faire perdre en attention le pilote.

4 Développement et intégration des solutions retenues

Nous présentons dans cette section les aspects techniques des solutions retenues permettant d'adapter le PIE précédent pour répondre aux objectifs fixés dans les précédentes sections, soient :

1. Enrichissement de l'ontologie avec des éléments réels.
2. Prise en compte des données en temps réel de trafic aérien et de météo.
3. Réponse à des requêtes en utilisant ces nouvelles données.

Comme précisé en section 3.1.1, le contexte du PIE précédent était le suivi d'un vol fictif pré-programmé à l'avance, et n'incluant aucune donnée du monde réel. Les requêtes formulées concernait donc ce vol en particulier, et des données extérieures "rentrées à la main". Dans le cadre de notre PIE qui comprend uniquement des données du monde réel, évoluant éventuellement en temps réel, nous devions donc déterminer à qui sont destinées les requêtes. Nous avons choisi de porter les requêtes sur un trafic existant que nous pouvons sélectionner, et émettre des requêtes en nous mettant "à la place" du pilote du vol sélectionné.

Le code source est disponible sur un dépôt Github [1], et une version déployée [2] est également accessible.

4.1 Enrichissement de l'ontologie

Une ontologie est un modèle de données permettant de stocker et représenter un ensemble de termes et concepts de manière structurée. L'ontologie utilisée dans ce projet suit le langage OWL (Web Ontology Language) [3]. OWL est une extension à l'ontologie RDFS (*Resource Description Framework Schema*) qui permet de décrire un ensemble de connaissances. Dans le cadre de ce projet, nous n'avons eu besoin que des notions de *Class*, *DatatypeProperty*, *ObjectProperty* et *Individual*, bien que OWL puisse être beaucoup plus expressif.

- **Class** : Ensemble d'individus partageant les mêmes propriétés. Exemple : `Airport`.
- **DatatypeProperty** : Attribut d'une classe. Exemple : `AirportAltitude` pour la classe `Airport`.
- **ObjectProperty** : Propriété entre deux classes. Exemple : `Has`, entre les classes `Airport` et `Runway` de sorte que `Airport Has Runway`.
- **Individual** : Instance d'une classe. Exemple : `Toulouse-Blagnac` est un individu de la classe `Airport`.

L'ontologie utilisée comme point de départ provient du PIE précédent, qui contenait déjà un certain nombre de classes et de propriétés exploitables. Néanmoins, l'ajout de données du monde réel permet, d'une part, d'ajouter un grand nombre d'individus, mais nous contraint également à altérer les propriétés des classes, en fonction de la disponibilité de telle ou telle propriété. L'utilisation de données évoluant en temps réel nous contraint aussi à manipuler des objets hors de l'ontologie, contrainte qui sera développée en Section 4.1.2. Une description des classes de l'ontologie utilisée dans ce projet est disponible en annexes.

1. `https://github.com/dinhviettoanle/PIE_flight_assistant_2021`
2. `https://pie2021-flightassistant-v21.herokuapp.com/`

4.1.1 Données statiques du monde réel

Les données statiques sont toutes stockées dans l'ontologie, et proviennent de sources ouvertes rassemblant ces différents éléments (notamment `https://ourairports.com` et `https://opennav.com`). En l'occurrence, nous prenons en compte dans l'ontologie les objets statiques suivants, en définissant leur classe correspondante :

- Aéroports
- Checklists
- Pistes d'atterrissage
- Aides à la navigation (VOR, DME, NDB, TACAN)
- Fréquences d'aéroport
- Points de passage (waypoints)

Le stockage de ces données se fait effectivement sous forme d'individus dans l'ontologie. En revanche, en ce qui concerne l'accès aux données par l'application, toutes les données sont lues une unique fois puis stockées en mémoire vive sous forme de `pandas.DataFrame`. En effet, du fait de la taille relativement restreinte de l'ensemble des données (environ 25 Mo de données brutes pour l'ensemble de ces données statiques, et 4 Mo en mémoire sous forme de `DataFrame`) et après plusieurs tests, nous observons une plus faible latence de lecture des `DataFrame` comparée à la lecture via requêtes SPARQL à l'ontologie. Seules certaines requêtes plus complexes, nécessitant les relations entre les classes, utilisent l'ontologie : par exemple « Quelles sont les pistes d'atterrissage de tel aéroport ? » qui nécessite la propriété `Has` entre `Airport` et `Runway`.

4.1.2 Ontologie et données dynamiques

L'ontologie du PIE précédent incluait également les classes `Aircraft` et `Weather`, représentant respectivement le trafic suivi, et des données météo à une position.

Néanmoins, la prise en compte de données réelles et évoluant en temps réel rend leur intégration à l'ontologie moins commode. Tout d'abord, inclure les données de trafic et de météo en tant que classe de l'ontologie nous contraindrait à stocker en mémoire toutes les données du monde entier. Cela impliquerait alors une place en mémoire importante (par exemple, tous les paramètres météo à chaque aéroport et ville du monde entier) et donc un temps de lecture potentiellement long. Aussi, étant donné que ces données évoluent en temps réel, cela demanderait également d'actualiser l'ensemble de ces données régulièrement, donc à nouveau rajouter de la latence due à l'écriture dans l'ontologie. Enfin, les restrictions des API, en termes de nombre de requêtes, ne nous permettent tout simplement pas d'avoir un taux de rafraîchissement très élevé (OpenWeatherMap n'acceptant que 60 appels par minutes, et FlightRadar24 restreignant le nombre d'appels aux données complètes d'un vol).

Nous développerons donc dans la section suivante les solutions mises en place pour prendre en compte ces données dynamiques.

4.2 Intégration des données dynamiques

Le second objectif de ce projet est d'intégrer à la maquette des données dynamiques en temps réel, en particulier, des données de trafic aérien et de météo. Nous expliquerons dans cette section les différents outils utilisés et leur traitement par la maquette.

4.2.1 Communication avec les API

L'extraction de ces données dynamiques évoluant en temps réel s'effectue via la communication avec des API (*Application Programming Interface*) ; en pratique, via des requêtes HTML sur une URL particulière. Nous avons présenté en Section 3.1.3 plusieurs possibilités d'API permettant de récupérer des données météo en temps réel et de trafic aérien, et nous détaillerons donc ici les solutions choisies.

Données météo. De par sa facilité d'accès et ses données fournies assez complètes, nous utilisons l'API d'OpenWeatherMap pour intégrer les données météo. Plus particulièrement, nous utilisons la bibliothèque PyOWM [3] qui est simplement un wrapper pour Python de l'API Web d'OpenWeatherMap. La bibliothèque propose un grand nombre de services, mais en pratique, au vu des requêtes que nous avons choisies nous n'utilisons que la fonction permettant d'extraire les données météo à un endroit particulier. Outre OpenWeatherMap pour les données météo, nous utilisons également AviationAPI [4] pour pouvoir répondre aux requêtes sur les METAR à un aéroport.

Données de trafic. Le service de trafic en temps réel choisi est celui de FlightRadar24. Les informations sont données sous forme de flux de `json` accessibles sur une URL accessible sans clé d'authentification. Pour gérer la communication entre l'API FlightRadar et notre script Python, il suffit donc simplement de faire des requêtes HTTP sur différentes routes : nous avons donc utilisé et adapté un code [5] existant permettant cette communication. L'API FlightRadar a deux points d'accès pour fournir différents éléments.

La première route permet d'acquérir les paramètres de vol (position, vitesse, altitude...) de tous les trafics à l'intérieur d'une certaine zone. Par exemple, ce *lien* renvoie les paramètres de vol du trafic aérien autour de Toulouse Blagnac. En particulier, ce point d'accès ne semble pas être restreint en termes de nombre d'appel, et nous permettra donc de mettre à jour périodiquement les données de vol des trafics alentours.

Le second point d'accès permet d'obtenir les informations complètes sur un vol spécifique. En particulier, comparé à la première entrée, nous avons en plus accès au modèle de l'avion, à son temps d'arrivée estimé (ETA), et d'autres informations que nous n'avons pas exploitées telles que l'historique de vol, ses précédentes positions... En revanche, ce point d'accès est restreint en termes de nombre d'appel.

Une version dégradée de la maquette utilisant les données d'OpenSkyNetwork a également été implémentée. En effet, les événements de fin février 2022, entraînant entre autres la fermeture de l'espace aérien ukrainien, ont provoqué un afflux d'utilisateurs sur FlightRadar24 et donc son indisponibilité durant quelques jours. Le service d'OpenSkyNetwork fournit des informations relativement similaires, mais son temps de réponse et sa couverture sont bien plus mauvais que FlightRadar24.

4.2.2 Représentation des données dynamiques

Comme expliqué précédemment, des contraintes et restrictions ne nous permettent pas de stocker ces données dynamiques comme des instances d'une classe `Aircraft` et `Weather`

3. `https://pyowm.readthedocs.io`
4. `https://docs.aviationapi.com/`
5. `https://github.com/alexbagirov/py-flightradar24`

dans l'ontologie. De ce fait, puisque les requêtes demandées par le pilote concernant le trafic ou la météo sont assez ponctuelles, il ne nous a pas semblé nécessaire d'envoyer d'appeler constamment les API et les stocker dans l'ontologie. Les solutions choisies sont représentées en Figure 4.1 et expliquées ci-dessous.

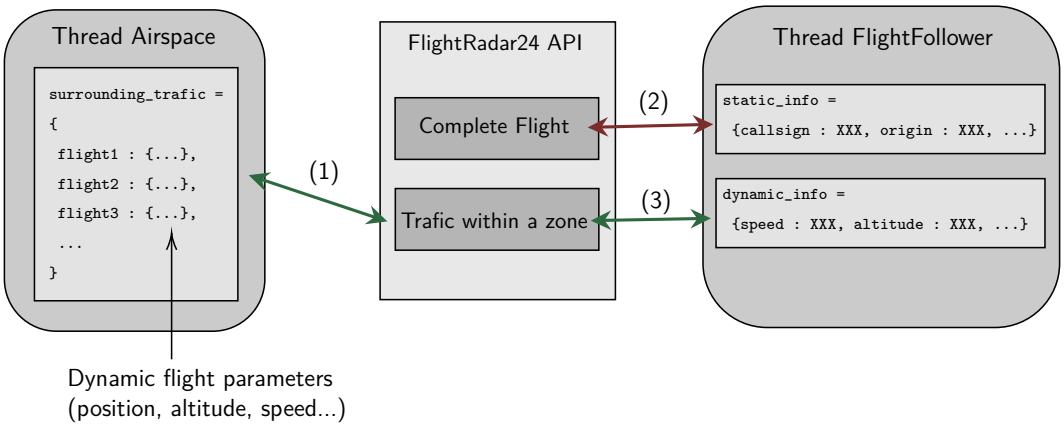

FIGURE 4.1 – Gestion des données dynamiques. Le premier *thread* `Airspace` gère le trafic alentour, en requêtant périodiquement le trafic dans une zone (1), appel qui n'est pas limité. Le second *thread* `FlightFollower` gère le suivi de l'avion sélectionné. Il appelle une unique fois le service permettant d'avoir les données du vol (2), appel qui est limité. Puis met à jour ses paramètres de vol en requêtant le service donnant le trafic dans la zone autour de sa dernière position (3).

Données météo. Les données météo ne sont stockées nulle part dans la mémoire de l'application. Un appel à l'API OpenWeatherMap est effectué seulement quand le pilote effectue une requête sur la météo.

Données de trafic. Les données de trafic font intervenir deux *threads* (processus en parallèle) : un *thread* gérant le trafic aux alentours et l'autre permettant de gérer le trafic suivi.

Le premier *thread* est composé d'un attribut, simplement de type `dict`, contenant la position et les paramètres de vol de tous les trafics autour du vol sélectionné. Ce *thread* lance alors des appels périodiques à l'API FlightRadar, en demandant le trafic dans une zone donnée (qui, pour rappel, ne sont pas restreints). De ce fait, on parvient à avoir un taux de rafraîchissement correct des données de vol du trafic alentour.

Le second *thread* gère le trafic sélectionné par l'utilisateur. Ainsi, lorsque l'utilisateur sélectionne l'avion, un unique appel l'API FlightRadar sur les données complètes du vol est lancé, afin d'initialiser tous ses paramètres statiques (nom de l'aéroport de départ et d'arrivée, modèle d'avion...) et sa position actuelle. En revanche, nous ne pouvons pas appeler périodiquement cette partie de l'API pour mettre à jour ses paramètres de vol (position, altitude...), du fait que celle-ci soit restreinte en nombre d'appels. Pour pallier ce problème, nous utilisons l'API pour demander le trafic dans une zone autour de la position précédente du trafic suivi. Cet appel n'étant pas restreint, on parvient alors rafraîchir périodiquement ses paramètres de vol, tout en ayant les informations complètes du vol.

Enfin, puisque les requêtes de l'utilisateur sont totalement asynchrones, le second rôle de ce *thread* est de pouvoir lancer le gestionnaire de requêtes de manière asynchrone, composante qui sera décrite dans la section suivante.

4.3 Requêtes

En utilisant ces nouvelles données statiques, et en incorporant des données de trafic et de météo, la maquette doit pouvoir répondre à des requêtes utilisateur, formulée à l'oral et en langage naturel. Nous expliquerons dans cette section notre utilisation du module de traitement du langage naturel et la gestion des requêtes à partir d'intentions.

4.3.1 Reconnaissance du langage naturel

La partie reconnaissance du langage naturel permet de transformer une phrase en langage naturel en une intention, que le gestionnaire de requêtes pourra traiter. Plus précisément, le module prendra en entrée une chaîne de caractères transcrite par la reconnaissance vocale, et devra renvoyer une intention "normalisée". De même que pour le précédent PIE, nous n'avons besoin que de la partie compréhension du langage naturel (NLU - *Natural Language Understanding*) et non d'une partie génération de langage naturel. De ce fait, nous pouvons nous baser sur la solution précédemment développée, soit l'utilisation de la bibliothèque Snips NLU [2].

Fonctionnement de Snips NLU. Cette bibliothèque repose sur ces deux concepts.

Les **intentions** relient de possibles énoncés (*utterance*) à une formulation normalisée. Dans l'exemple en Figure 4.2, on relie les différentes formulations de la même idée « quel est l'objet le plus proche », à la forme normalisée `nearestEntity`.

Les **entités** sont les valeurs possibles prises par les champs d'une intention, avec possiblement des synonymes. L'exemple décrit que dans l'intention, `[object]` peut prendre une des valeurs de l'entité `aviationEntity`. Aussi, la liste de valeurs possibles dans la définition de l'entité est une liste de synonymes. Par exemple, *"What is the nearest airfield"* sera compris de la même manière que *"What is the nearest airport"*. En l'occurrence, quand un synonyme est utilisé, Snips NLU renvoie le premier élément de la liste de synonymes.

```yaml
# nearestEntity Intent
---
type: intent
name: nearestEntity
slots:
  - name: object
    entity: aviationEntity
utterances:
  - give me the nearest [object]
  - can you give me the nearest [object]
  - what is the nearest [object]
```

```yaml
# aviationEntity Entity
---
type: entity
name: aviationEntity
values:
  - [airport, airfield]
  - [traffic, airplane]
  - [runway]
```

FIGURE 4.2 – Exemple d'intention et d'entité utilisées par Snips-NLU, au format `yaml`.

Une fois les intentions et entités déclarées dans ce fichier `yaml`, un modèle peut être entraîné à partir de ces données et être utilisé pour prédire l'intention à partir d'un énoncé quelconque.

Correction d'erreurs. Nous exploitons également le NLU pour pouvoir corriger les erreurs de la reconnaissance vocale. En effet, expérimentalement, la transcription par reconnaissance vocale peut être entachée d'erreur. De ce fait, en supposant que certains mots ne seront a priori jamais utilisés, nous utilisons le système de synonymes pour corriger ces erreurs de transcriptions. Plus précisément, nous ajoutons à la liste de synonymes tous les mots ressemblant au mot souhaité.

Par exemple, l'entité `aviationEntity` effectivement utilisée est décrite en Figure 4.3 : le mot *"port"* ne sera, a priori, jamais utilisé dans son sens original, mais risque d'être retranscrit si la reconnaissance vocale coupe la syllabe *"air-"* de *"aiport"*. Ainsi, le NLU comprendra quand même que l'on souhaitait parler d'aéroport.

```
# aviationEntity Entity
---
type: entity
name: aviationEntity
values:
  - [airport, airfield, airports, airfields, port]
  - [traffic, airplane, airplanes, aeroplane, aeroplanes]
  - [runway, railway, runways, railways, runawayt, runaways]
```

FIGURE 4.3 – Ajout de synonymes dans la définition d'une entité dans un but de correction d'erreur de reconnaissance vocale.

Grâce à cela, on parvient à empiriquement corriger un certain nombre d'erreurs, même s'il reste des éléments qui ne parviennent pas à être correctement compris, éléments que nous décrirons en Section 5.1.1.

4.3.2 Traitement des requêtes

Le module de traitement des requêtes est finalement le coeur de l'application. Ce module permet donc de prendre en entrée l'intention comprise par le module NLU, et en donner la réponse. Pour cela, celui-ci doit pouvoir manipuler requêtes SPARQL à l'ontologie et appels à des API extérieures, dont les notions et solutions ont été présentées en Sections 4.1 et 4.2.

Requêtes considérées. Tout d'abord, outre l'aspect technique, nous avons dû choisir une liste de requêtes à prendre en compte. Nous nous sommes alors basés sur le PIE précédent pour avoir un premier point de départ, que nous avons ensuite altéré en fonction du nouveau contexte de données réelles. Par exemple, des requêtes pour demander l'*anti-icing* ou la *landing distance*, présentes dans la maquette précédente, nous sont indisponibles, alors que des nouvelles requêtes comme le trafic le plus proche nous sont alors disponibles. La liste des requêtes implémentées est disponible en annexes, et est reprise du manuel utilisateur écrit pour ce PIE.

Lien avec le NLU. Sur l'aspect technique, toutes les requêtes présentées dans le tableau en annexes ne sont, en réalité, pas liées à une intention de Snips NLU, déclarée dans le `yaml`. En effet, certains types de requêtes sont très ressemblants et, après plusieurs tests, sont très sensibles aux erreurs de retranscription. Par exemple, la différence dans les énoncés pour `NearestAirport`, `NearestTrafic` et `NearestRunways` se joue à un seul

mot : *"what is the nearest airport / trafic / runway ?"*. De ce fait, dans la base de données d'entraînement pour le NLU, il y a en réalité beaucoup moins d'*intent* déclarées. Les *intent* prédites par NLU sont d'abord pré-traitées pour en extraire la réelle intention derrière l'énoncé. Plus visuellement, on présente dans les annexes un arbre permettant de lier les requêtes effectivement déclarées dans les données d'entraînement et les requêtes disponibles à l'utilisateur.

Nous avons donc détaillé les éléments majeurs de fond de la maquette, soit les nouveaux éléments ajoutés à l'ontologie, les appels aux API extérieures, l'utilisation du NLU et l'interaction entre tous ces éléments. Nous détaillerons dans la section suivante la forme de la maquette, et sa structure globale.

4.4 Architecture de l'application

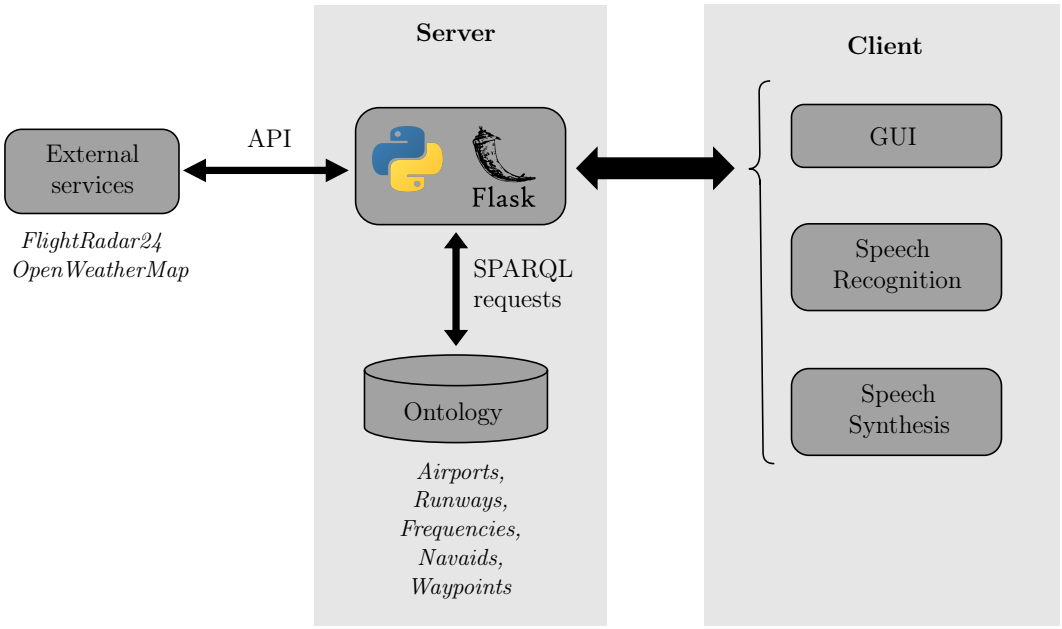

FIGURE 4.4 – Architecture globale de l'application.

L'architecture finale choisie pour le Flight Assistant est sous forme d'application web. Alors que le PIE précédent avait choisi de l'implémenter sous forme d'application *standalone*, nous nous sommes orientés vers une application web pour plusieurs raisons, notamment :

- Pour nous, une facilité de mise en place de l'environnement de travail (notamment avec les complications entre WSL et les fenêtres graphiques).
- Une meilleure connaissance des technologies web front-end pour implémenter l'interface utilisateur, par rapport aux bibliothèques Python orientées GUI (Tkinter, Pyglet ou Qt5).
- Après déploiement sur un serveur distant, on évite également les pertes de performances dûes au matériel de la machine locale ou dûes à la connexion internet.
- Les déploiements réguliers permettent de présenter l'avancée du travail auprès du client, sans que celui n'ait besoin d'installer ou mettre à jour des dépendances.

L'architecture globale de l'application est présentée en Figure 4.4. On décrira dans les paragraphes suivants les éléments non développés précédemment, en particulier les éléments côté client.

Interface graphique. L'interface graphique et les différents éléments interactifs ont été réalisés entièrement en HTML / CSS / Javascript. En particulier, nous n'avons pas utilisé de framework front-end (Angular, Vue, React...) par manque de compétences sur leur utilisation, et considérant que les technologies web de base soient suffisantes pour créer cette première interface.

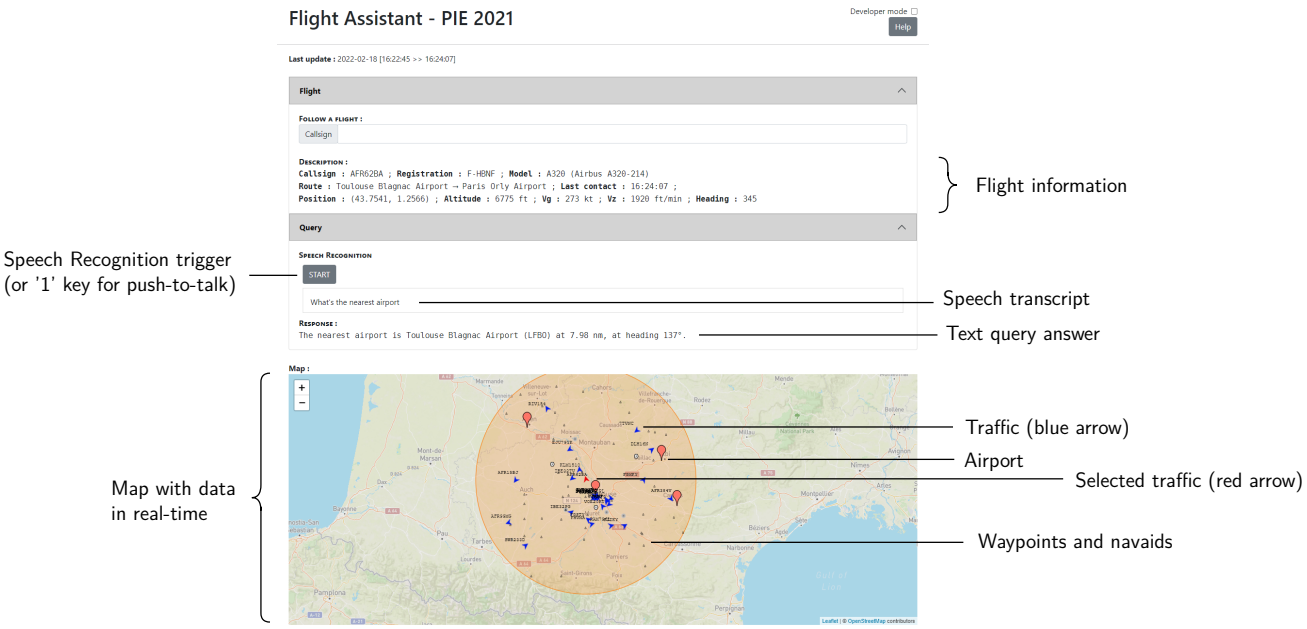

FIGURE 4.5 – Interface utilisateur.

Une capture d'écran de l'interface graphique client est présentée en Figure 4.5. L'utilisateur, n'a théoriquement besoin que d'interagir avec la carte une seule fois, lors de la sélection du vol à suivre, puis toutes les requêtes et réponses se font ensuite à l'oral. L'interface graphique ne servant qu'à donner un retour supplémentaire visuel.

Reconnaissance vocale. La reconnaissance vocale est le premier élément du processus de requêtes. Elle permet de transformer le signal audio de la voix, en une transcription de la phrase prononcée sous forme de chaînes de caractères. Celle-ci est donc implémentée du côté client, en Javascript et en se utilisant l'interface SpeechRecognition du *Web Speech API*.

En particulier, nous n'avons pas choisi de l'implémenter côté Python comme l'avait fait le PIE précédent. Pour rappel, celui-ci avait implémenté la reconnaissance vocale en utilisant la bibliothèque SpeechRecognition de Python. En effet, en accord avec les pistes d'amélioration proposées dans leur rapport, nous avons remarqué une très forte latence due à la reconnaissance vocale dû, entres autres, au traitement du signal audio en différé. Par ailleurs, l'implémentation d'un bouton push-to-talk (donc nécessairement situé au niveau client) nous contraint à implémenter la reconnaissance vocale du côté client –

l'envoi d'un signal « appui » et « relâchement » au serveur rajoutant probablement encore de la latence. De ce fait, cela nous a poussé à directement implémenter la reconnaissance vocale côté client.

Du point de vue utilisateur, celui-ci utilise un bouton push-to-talk (en l'occurrence, la touche « 1/& » du clavier) pour énoncer oralement sa requête, et lors du relâchement, la transcription est directement envoyée au serveur pour y être traitée.

Synthèse vocale. Enfin, la synthèse vocale est le dernier élément de la chaîne. Elle transforme la requête envoyée par le serveur sous forme de chaîne de caractère en un retour audio pour l'utilisateur. La synthèse vocale est donc également en Javascript, basée sur l'interface SpeechSynthesis du *Web Speech API*.

En définitive, on présente en Figure 4.6 un exemple de traitement en interne d'une requête, telle "What is the nearest airport ?".

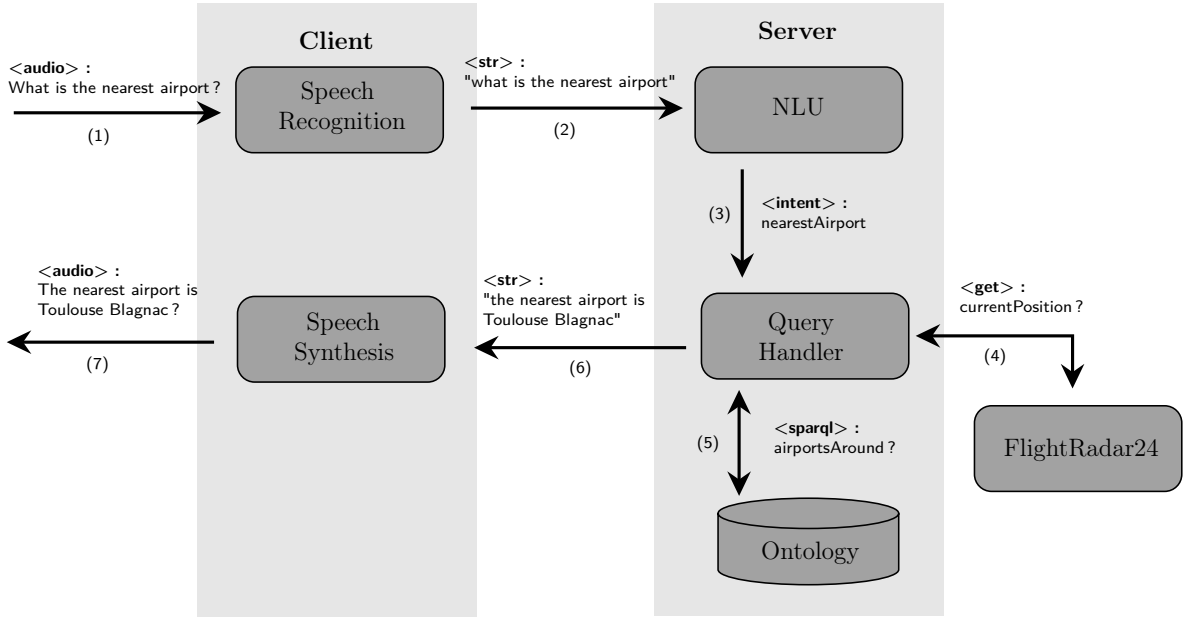

FIGURE 4.6 – Exemple de traitement d'une requête en interne. L'utilisateur formule à l'oral et en langage naturel une requête (1) qui est envoyée au module de reconnaissance vocale. Ce dernier transcrit l'audio (2) et l'envoie au serveur. Le module de Natural Language Understanding transforme cette phrase formulée en langage naturel en une requête « normalisée » (3). Le gestionnaire de requête traite celle-ci en effectuant les tâches nécessaire. En l'occurrence, pour cette requête, appeler l'API FlightRadar (4) et requêter des aéroports à l'ontologie (5). Le gestionnaire de requêtes renvoie alors une chaîne de caractère (6) contenant la réponse à la requête et l'envoie au module de SpeechSynthesis chez le client. Ce dernier transforme alors la chaîne de caractère en données audio (7).

5 Utilisation de l'assistant de vol virtuel

5.1 Performances de la maquette

5.1.1 Tests internes

Afin de tester la robustesse de notre maquette, nous avons établi une liste de 20 questions représentatives de celles qui pourraient être posées par un pilote dans le cockpit lors d'un vol. Notre but est aussi d'explorer au maximum le spectre de questions possible sur la maquette. Pour cela, nous avons établi le protocole suivant :

— L'utilisateur choisit un vol.
— Il énonce la question telle qu'elle est écrite.
— Il nous dit si la réponse était en accord avec la requête énoncée.

En tout, 22 personnes nous ont permis de faire ces tests, ce qui nous permet aussi de tester la robustesse la reconnaissance vocale couplée à la compréhension de langage naturel. La liste des questions ainsi que les résultats obtenus sur chaque question est disponible en annexes.

Nous constatons un taux de fiabilité global de notre maquette de 70%. Les questions classiques telles que "What is the ground at arrival ?" ou "When will I arrive ?" ont une fiabilité de 100 %. Alors que d'autres telles que "Can you give me the app at LFRU ?" ont une fiabilité de 0%. En effet, l'algorithme de reconnaissance vocale est moins performant pour reconnaître une suite de lettres.

L'indicatif d'aéroport est d'autant plus compliqué à comprendre que "RU" se prononce "are you", c'est donc la deuxième option que comprend notre algorithme. Une autre problématique est la compréhension des noms de ville, en effet, les villes anglaises telles que "Cambridge" ou "Gloucester" alors que "Lille" et "Charles de Gaulle" sont moins bien reconnues par notre algorithme. La reconnaissance automatique de langage est par ailleurs un domaine de recherche actuellement actif [4], et le module SpeechRecognition natif des navigateurs web ne sont entraînés que sur un seul langage.

Enfin, le français est la langue maternelle de la majorité des participants à ce test, ainsi certains mots sont mal compris car leur prononciation de l'anglais n'est pas parfaite, ceci est visible sur les mots tels que "arrival", souvent compris comme "a rival" ou "are evil". Ce type d'erreur est en revanche plus facile à corriger, via un post-processing sur le module de compréhension du langage naturel, comme expliqué en Section 4.3.1.

Les difficultés mentionnées sont courantes pour les algorithmes de reconnaissance vocale et c'est ce que nous avons voulu mettre en lumière en choisissant nos 20 questions. En l'occurrence, les problèmes de détection de mots épelés (ICAO d'aéroports) et des mots dans d'autres langues sont les principales faiblesses et premières pistes d'amélioration de notre maquette.

5.1.2 Retours pilotes sur la maquette réalisée

De la même manière qu'en 3.2, nous avons pris l'initiative de soumettre un nouveau questionnaire aux mêmes pilotes (sous forme de Google Form également) pour recueillir leur avis sur la maquette que nous avons réalisé. Le but du questionnaire était d'évaluer l'ergonomie de notre IHM, notamment vis à vis du choix d'afficher les informations ou de les faire énoncer par l'agent, les fonctionnalités de celui-ci et d'avoir d'éventuels retours lors de leur usage de la maquette.

Ainsi une fois l'identité et l'expérience des personnes sondés recueillies, le questionnaire s'articule de la manière suivante avec une prépondérance des questions portant sur des points d'ergonomie :

Choix des informations que l'agent doit afficher de manière prioritaire. Parmi les informations suivantes sur un vol que fournit notre maquette (Callsign, Registration, Model, Route, Last Contact, Altitude, Position, Vg, Vz, Heading), nous demandons aux pilotes lesquelles devait être affichées par l'agent. De plus, nous questionnons l'intérêt de réserver un affichage dédié pour ces informations afin qu'elle soit affichées tout au long du vol.

A propos de requêtes basiques. En terme pratique, nous demandions si il était intéressant pour un pilote de pouvoir requêter et recevoir une réponse à des requêtes basiques et récurrentes par simple appui d'un bouton (par exemple tactile) sur un écran. Nous demandons également si la réponses à ces requêtes devaient être de préférence longues avec des détails, courtes avec moins de détails, affichées sur un écran ou alors annoncées par audio.

A propos de requêtes plus élaborées et des checklists. Pour les checklists, nous interrogeons les pilotes sur la manière dont nous devions les fournir, doivent-elles être affichées, énoncées et/ou répétées par l'agent ? Concernant des requêtes plus longues et complexes, nous questionnons la nécessité d'adapter le module de SpeechRecognition de manière à être capable de les traiter avec efficacité. De la même manière que les réponses aux requêtes basiques, nous demandions si les réponses à ces requêtes complexes doivent être longues avec des détails, courtes avec moins de détails, affichées sur un écran ou alors annoncées par audio. Enfin, une question, dont la réponse était laissée libre, portait sur le cadre d'utilisation dans lequel les pilotes pourrait formuler des requêtes à l'oral.

Retour d'utilisation et avis sur l'ergonomie de la maquette. La question suivante demandait sur une échelle de 1 (très désagréable) à 5 (très agréable), l'évaluation que donneraient les pilotes à l'utilisation de la maquette. Nous demandions également si leur utilisation de la maquette avait collé à l'idée qu'ils s'en étaient fait et si la maquette avait répondu à un maximum de leurs espérances.

Enfin, une dernière question concernait de soulever d'éventuels dysfonctionnements de la maquette et leur cadre d'apparition. L'ultime question était laissée libre pour tout commentaire ou suggestion.

Malheureusement, par manque de temps, de possibles difficultés à utiliser notre maquette (malgré un guide utilisateur et une disponibilité de notre part par mail) et parce que le travail demandé aux pilotes était un peu plus conséquent, nous n'avons pas pu recueillir un nombre représentatif de réponses au questionnaire et en présenter les statistiques en détail ne serait pas pertinent. Néanmoins, les quelques réponses reçues nous ont permis d'évaluer les performances de notre maquette et de recevoir des premiers avis sur notre travail.

Globalement, les pilotes ont trouvé leur utilisation de la maquette très agréable et intuitive et ont trouvé que le projet était prometteur et bien réalisé. Ils ont indiqué que les vitesses Vg et Vz et l'altitude du vol en cours était intéressant à afficher dans un

espace dédié tout au long du vol. Ils avaient également formulé le souhait que l'agent puisse répondre de manière prioritaire à des requêtes basiques et que l'idée de lancer les requêtes par appui d'un bouton tactile semblait digne d'intérêt. Ils ont également indiqué que certaines requêtes n'ont pas toujours pu être reconnues par la maquette, notamment quand cela impliquait des suites de lettres ou des mots français pouvant porter à confusion en anglais. De manière générale, les pilotes trouvaient la maquette suffisamment bien pensée et qui pouvait être possiblement intégrée dans un cockpit sous réserve d'un travail supplémentaire, par exemple pour un prochain projet.

5.1.3 Entretiens individuels pilotes

L'une des élèves pilotes ayant répondu à notre questionnaire a effectué un projet sur le thème de la reconnaissance vocale dans un cockpit d'avion militaire. Lors d'un entretien personnel avec elle, elle a soulevé l'importance d'implémenter par la suite une partie traitement du signal vocal en amont de notre maquette car l'environnement cockpit est très bruyant.

5.2 Pistes d'amélioration

Les améliorations techniques principales de ce projet interviennent dans le module de reconnaissance du langage, notamment la prise en compte des mots épelés et la reconnaissance multilingue, comme citées précédemment. D'autres corrections dans les fonctions de réseaux ainsi que des tests plus approfondis sont également à envisager, mais nous identifions surtout une suite de projet qui nous apparaît comme essentielle pour comprendre l'intérêt d'un agent conversationnel au sein d'un cockpit.

5.2.1 Potentielle suite de projet

Comme indiqué en début de Section 4, l'utilisateur se mettait "à la place" du pilote d'un vol réel pour passer ses requêtes. Les possibilités de requêtes sur les éléments extérieurs sont assez satisfaisantes, mais les requêtes portant sur le vol en lui-même se restreignaient à des paramètres de vol classiques (altitude, vitesse, cap...), seules données que nous avions à notre disposition avec FlightRadar24.

Or, il peut être judicieux et utile pour le pilote d'avoir des requêtes sur l'avion en lui-même, comme ses performances (distance d'atterrissage nécessaire, rayon d'action maximal, altitude optimale...) ou le statut de certains systèmes (anti-icing, fuel restant, état des volets...). Certaines de ces requêtes avaient été implémentées dans le PIE précédent qui disposait de toutes ces données provenant d'un vol fictif. De ce fait, il peut être intéressant d'incorporer ce prototype dans un logiciel de simulation de vol [6] permettant à la fois de de requêter des informations d'éléments du monde réel, mais également des paramètres avion. Au-delà de ces possibilités de requêtes, ce nouveau prototype pourra être testé en simulateur, dans des conditions plus réalistes que celles de notre projet (incluant notamment la communication radio qui peut entrer en conflit avec la maquette, comme présenté dans le questionnaire aux pilotes), et permettant donc de réellement tester l'efficacité d'un agent conversationnel au sein d'un cockpit.

6. `https://github.com/nasa/XPlaneConnect`

6 Conclusion

En conclusion, notre projet PIE a donné lieu à une amélioration de l'agent conversationnel précédemment développé, permettant à un pilote de requêter certaines informations. En particulier, dans le cadre de notre projet, des informations sur le trafic et la météo en temps réel ainsi que des données statiques du monde réel. Cette maquette a donc été implémentée suite à des réflexions et recherches sur les données disponibles, l'architecture adéquate ainsi que des sollicitations pour établir les requêtes les plus appropriées pour un pilote.

Pour cela, différentes compétences académiques et techniques ont dû être sollicitées, en particulier sur l'aspect programmation de la maquette. Pour ne citer que la principale, le choix de l'architecture d'application web pour notre maquette nous a demandé de mettre en place nos connaissances de programmation réseau et des relations client-serveur. Au delà du code en lui-même, nous avons souhaité exploiter cette architecture web pour faciliter la démonstration de notre maquette au client. Plus concrètement, nous avons alors exploité nos connaissances de conteneurisation d'applications via *Docker*, et de déploiement via *Heroku* et *OpenShift*, pour déployer la maquette et plus facilement communiquer nos résultats avec Dassault Aviation.

Également, les principaux objectifs de ce PIE ont été atteints puisque le déploiement de notre maquette est opérationnel et que les tests sont en grande partie concluants. Néanmoins, il est possible de soulever quelques points de développement que nous aurions aussi voulu faire tel que l'implémentation concrète de notre maquette dans un cadre plus réaliste, la prise en compte de plusieurs langages (autre que l'anglais) et une rencontre physique avec d'éventuels pilotes qui pourrait l'utiliser ainsi qu'une utilisation en "temps réel" de notre maquette par ces mêmes pilotes.

Les relations de travail entretenues entre notre équipe et les ingénieurs de Dassault Aviation furent très cordiales et instructives. Nous avons pu mieux appréhender un travail avec un tuteur industriel et nous projeter efficacement pour développer une application qui fonctionne. Hervé Girod et Timon Ther étaient également très disponibles et pédagogues et nous ont accompagné durant toute la durée du projet en organisant notamment des réunions bi-mensuelles pour faire des points récurrents sur notre travail. Ils étaient également très à l'écoute concernant toute suggestion que nous avons pu apporter et ont permis d'instaurer un cadre beaucoup plus concret d'application de nos connaissances. Nous les remercions d'ailleurs grandement pour l'implication et le temps qu'ils ont investi pour nous dans ce projet.

De la même manière que le PIE précédent nous ait ouvert cette perspective d'ajout de données réelles et dynamiques, notre projet permettrait également un nouveau pas vers l'implémentation d'un agent conversationnel au sein d'un cockpit. Pour aller dans cette direction, il serait donc judicieux de réintroduire les requêtes sur les paramètres des systèmes de l'avion, au delà des simples paramètres de vol, en permettant par exemple la communication avec un logiciel de simulation de vol. Cette évolution permettrait, entres autres, des environnements et contextes plus réalistes d'utilisation d'un agent conversationnel pour cockpit civil.

Références

[1] D. Andrioaie, W. Angelier, A. Garzon Joli et L. Côme, « PIE Dassault Aviation : Assistance monopilote, » ISAE-SUPAERO, 2021.

[2] A. Coucke, A. Saade, A. Ball et al., « Snips Voice Platform : an embedded Spoken Language Understanding system for private-by-design voice interfaces, » *arXiv preprint arXiv :1805.10190*, p. 12-16, 2018.

[3] J.-B. Lamy, « Owlready : Ontology-oriented programming in Python with automatic classification and high level constructs for biomedical ontologies, » *Artificial intelligence in medicine*, t. 80, p. 11-28, 2017.

[4] H. Yadav et S. Sitaram, « A Survey of Multilingual Models for Automatic Speech Recognition, » *arXiv preprint arXiv :2202.12576*, 2022.

Annexes

6.1 Ontologie

L'ontologie complète provenant du PIE précédent comprenait également les classes `Aircraft` et `Weather`, ainsi que des `ObjectProperty` supplémentaires. Comme expliqué en Section 4.1, ces deux objets, évoluant en temps réel, ne sont ni stockés ni gérés par l'ontologie.

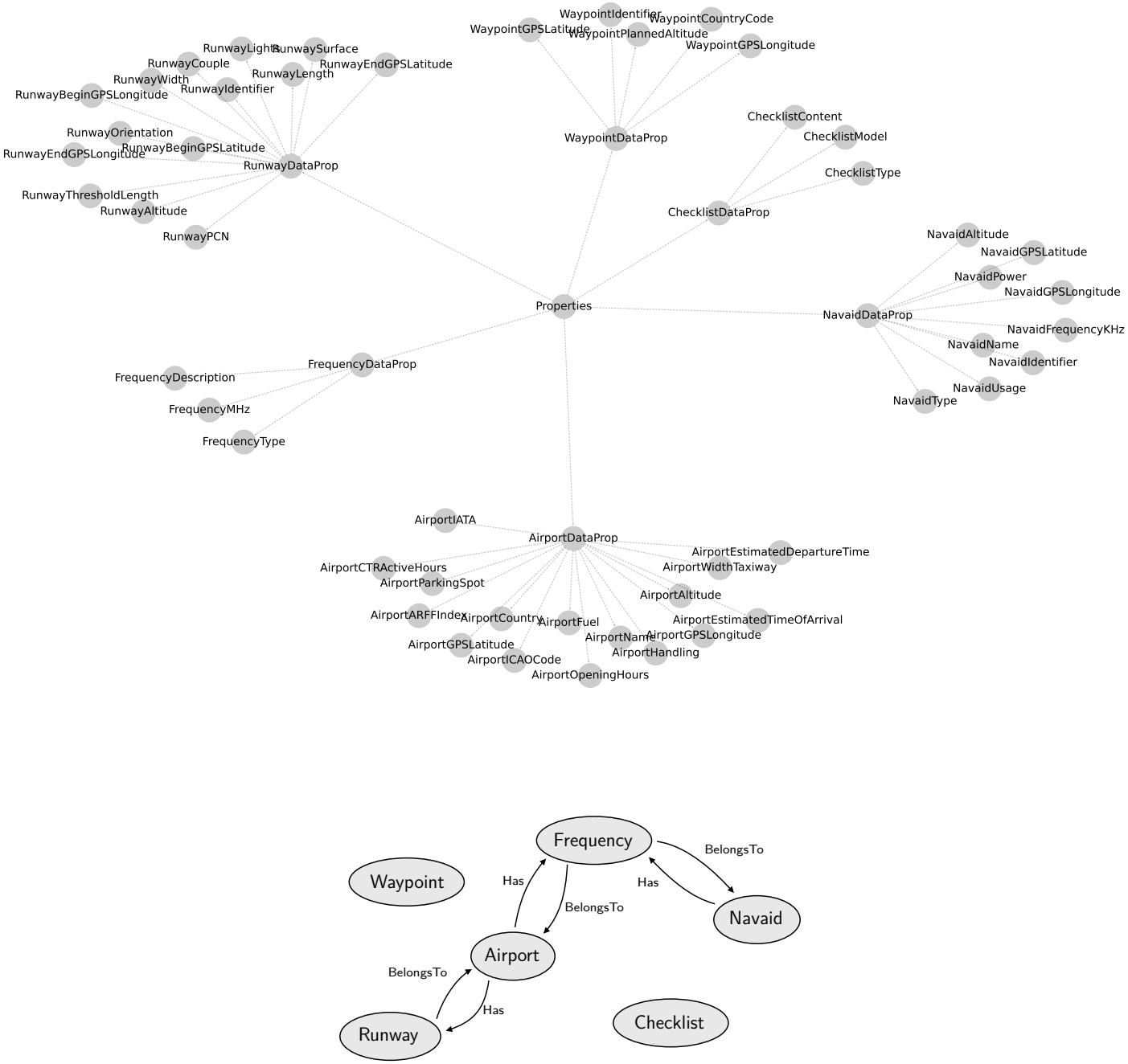

FIGURE 6.1 – `DatatypeProperty` et `ObjectProperty` des classes de l'ontologie

6.2 Requêtes disponibles

Intent	Description	Examples
Static queries (that will not change during the flight)		
Departure / Arrival airport	Returns the name and the ICAO of the departure / arrival airport	"What is the arrival airport ?"
Runways at an airport	Returns the list of runways at the an airport	"What are the runways at [Toulouse Blagnac]* ?"
Frequency at an airport	Returns the frequency value of a frequency channel at an airport	"[TWR] frequency at [Toulouse Blagnac]*"
Dynamic queries (that may change during the flight)		
Current flight parameters	Returns a current flight parameter (speed, altitude, Vz, heading...)	"What is my [altitude] ?"
Nearest airport	Returns the nearest airport from the current position	"What is the nearest airport ?"
Nearest trafic	Returns the nearest trafic from the current position	"Where is the nearest trafic ?"
Nearest runways	Returns the list of runways at the nearest airport	"What are the nearest runways ?"
Length of the nearest runway	Returns the length of the longest nearest runway	"What is the length of the nearest runway ?"
Estimated time of arrival	Returns the ETA	"Estimated time of arrival"
Weather queries		
Weather at airport	Returns a global status of the weather or a specific parameter at an airport	"What is the weather at [Toulouse Blagnac]* ? "What is the [visibility] at [Toulouse Blagnac]* ?"
Weather at location	Returns a global status of the weather or a specific parameter at a location	"Give me the weather at [Toulouse] ?" "[Temperature] at [Paris]"
Checklists		
Approach checklist	Returns the approach checklist	"Give me the approach checklist"
Landing checklist	Returns the landing checklist	"Landing checklist"

* The airport can also be [arrival] or [departure]. **Current available airport names are French and English airports in this version.**

Available frequencies – (Depending on the airport) TWR (Tower), APP (Approach), GND (Ground), INFO, AFIS, ATIS
Available flight parameters – Heading, Speed, Altitude, Vertical speed, Latitude, Longitude, Callsign, Registration
Available weather parameters – Temperature, Pressure, Wind, Gust, Clouds, Visibility, Rain, METAR

6.3 Intentions implémentées pour le NLU

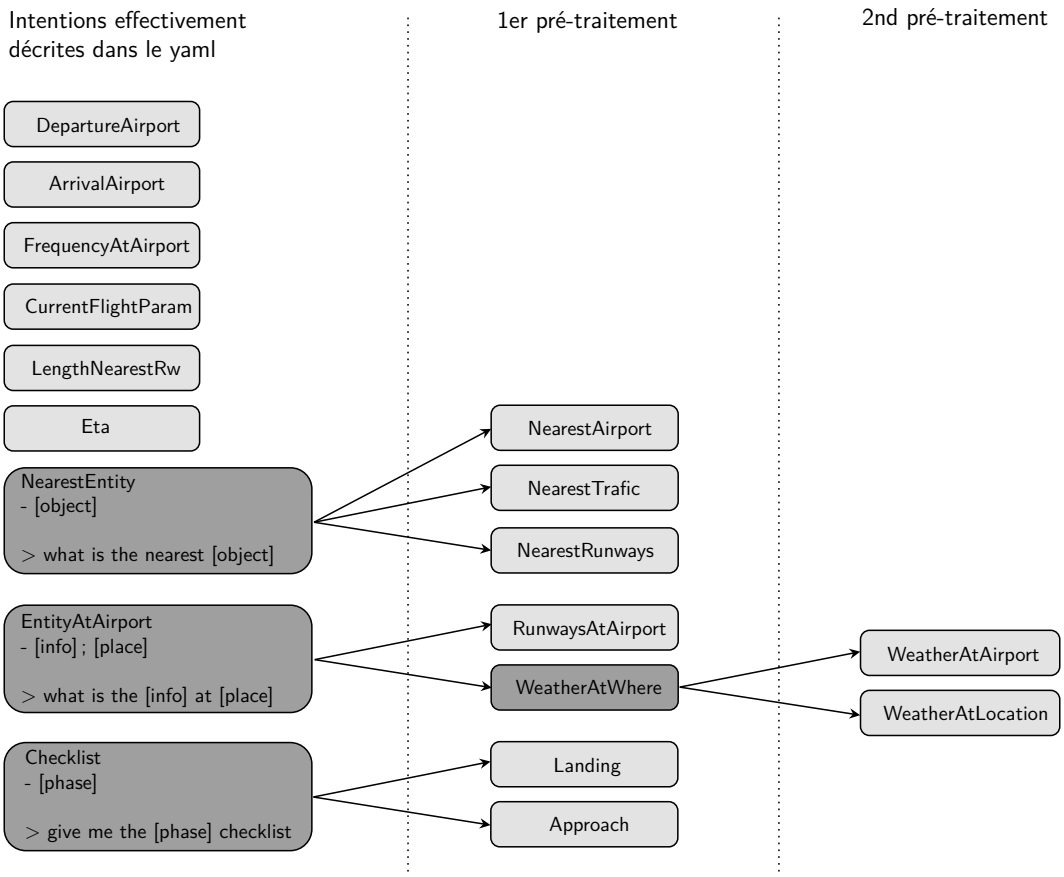

FIGURE 6.2 – Arbre liant les *intent* déclarées pour Snips NLU (en gris foncé) et les intentions réelles, présentées dans le tableau précédent (en gris clair). Pour les trois intentions `NearestEntity`, `EntityAtAirport` et `Checklist`, sont décrits les arguments (*slots*) et un exemple d'énoncé.

L'intention `WeatherAtWhere` est un cas particulier, et est traitée assez spécifiquement : si le champ `[place]` est trouvé dans la base de données d'aéroport, on renvoie la requête `WeatherAtAirport`, sinon on effectue une opération de géocodage de cet argument `[place]` et on appelle `WeatherAtLocation` sur les coordonnées trouvées.

`FrequencyAtAirport` n'est pas non plus dans `EntityAtAirport` pour des raisons de performance.

6.4 Liste des questions pour tester la maquette

Ci dessous, la liste des questions que nous avons utilisées pour tester notre maquette ainsi que leur taux de bonnes réponses. Nous avons posé 3 questions simples, 10 questions avec un seul paramètre, et 7 questions avec 2 paramètres.

Question	**% de bonnes réponses**
What is the departure airport ?	100
Can you give me the arrival airfield ?	91
When will I arrive ?	100
Give me the tower frequency at arrival	64
What is the ground at arrival ?	64
Give me the nearest runway	73
Can you give me the nearest airport ?	91
What is the nearest airplane ?	100
What is my speed ?	91
What is my current altitude ?	91
Current callsign	18
Start the approach check please	91
Can you give me the landing checklist ?	100
Give me the ATIS frequency at Lille	9
Can you give me the app at LFRU ?	0
Ground frequency at Paris Charles de Gaulle	46
Give me the visibility at EGBB	55
I need the temperature in Cambridge	100
Is there wind at Gloucester Gloucestershire Airport ?	64
What is the weather at QCY ?	55

TABLE 3 – Questions énoncées pour tester la reconnaissance du langage par la maquette.