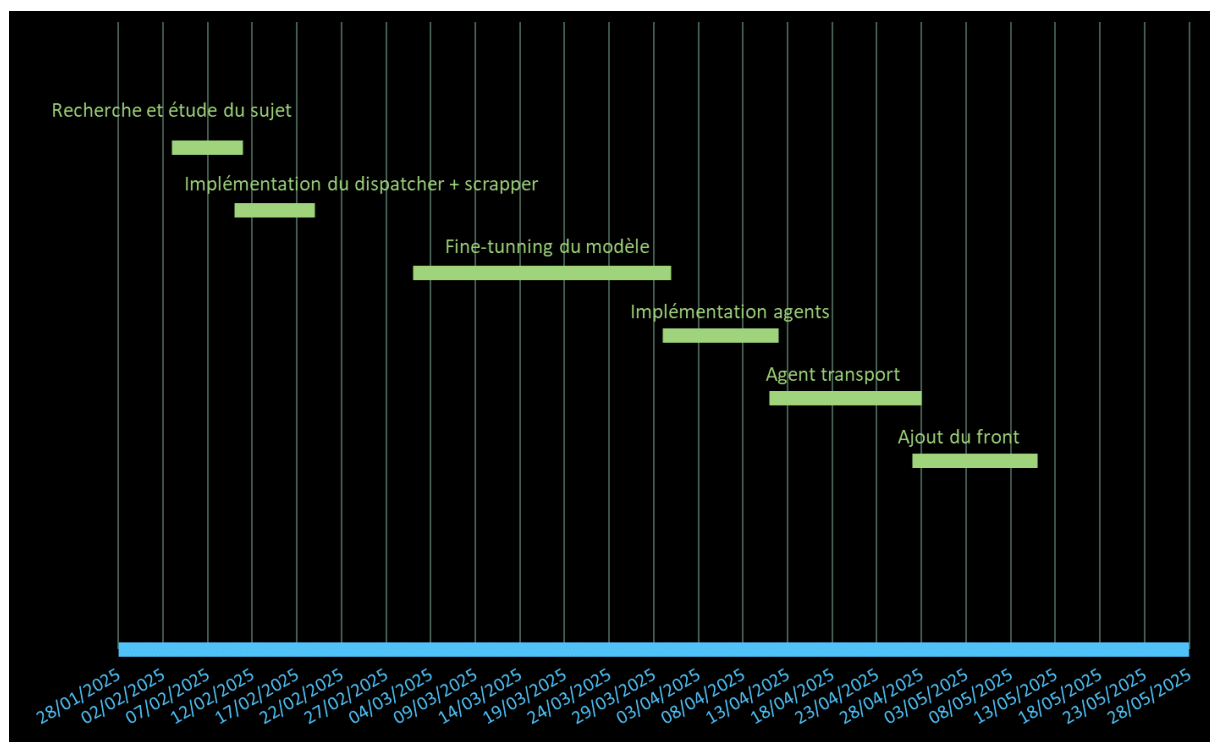


RAPPORT PROJET INDIVIDUEL INNOVANT

Meriem DOGHECHE –
INFO FISA 5

Table des matières

Introduction	3
1. Contexte, problématique et état de l'art	4
1.1 Contexte et besoins utilisateur.....	4
1.2 Défis et objectifs du projet	4
1.3 État de l'art et solutions existantes.....	5
2. Données et modèle de classification du dispatcher	7
2.1 Collecte et préparation des données d'entraînement	7
2.2 Choix du modèle de classification (SBERT vs baseline)	9
2.3 Fine-tuning du modèle SBERT pour le dispatcher	10
2.4 Performances obtenues et analyse	12
3. Architecture logicielle et fonctionnement des agents	15
3.1 Le Dispatcher : routage des requêtes via SBERT et fallback	17
3.2 Agent Météo (weather_agent.py)	20
3.3 Agent Transport (transport_agent.py)	22
3.4 Agents Culture et Loisirs (culture_agent.py & loisirs_agent.py)	26
3.5 Interface utilisateur Streamlit (ui_app.py)	28
3.6 Programme principal en console (main.py).....	34
4. Organisation du projet, contraintes et innovation.....	35
4.1 Planification et déroulement par étapes (diagramme de Gantt).....	36



..... 36

4.2 Contraintes matérielles et choix techniques associés.....	39
4.3 Innovation du couplage LLM + agents spécialisés.....	42

Introduction

Ce rapport présente un **assistant conversationnel local pour la mobilité urbaine** développé dans le cadre d'un Projet Individuel Innovant (PII) à l'INSA. L'objectif du projet est de permettre à un utilisateur de poser, en français, des questions variées liées à ses déplacements et sorties (météo, itinéraires de transport, activités culturelles et de loisirs) et d'obtenir des réponses instantanées dans une **seule interface unifiée**. Actuellement, un étudiant doit utiliser **plusieurs applications** : par exemple consulter une application météo, ouvrir Citymapper ou le site de la SNCF pour planifier un trajet, et parcourir des blogs ou réseaux sociaux pour trouver des idées de sorties. Malgré la disponibilité croissante de **données ouvertes** (météo, transports, agendas culturels...), **aucune application locale ne les réunit de façon simple**. Ce projet vise donc à **fusionner ces informations** en une seule conversation naturelle, sans exiger de l'utilisateur une formulation technique ni l'usage de multiples outils.

Pour relever ce défi, la solution proposée s'appuie sur deux composants principaux : (1) un **modèle de langage de grande taille (LLM)** chargé de comprendre les requêtes en langage naturel, et (2) des **agents logiciels spécialisés** capables de traiter chacun un type de question (par exemple un agent météo interrogeant une API météorologique, un agent transport utilisant un service d'itinéraires, etc.). Un **dispatcher (routageur)** central oriente chaque question utilisateur vers le bon agent en fonction de sa catégorie.

Ce **couplage d'un LLM avec des agents spécialisés** constitue l'originalité du projet : il combine la **compréhension du langage naturel** offerte par les LLM avec la **précision des données en temps réel** fournies par des API dédiées.

Lors de la première réunion avec mon tuteur, nous avons convenu de réaliser le projet **entièrement en Python**, en privilégiant des solutions **gratuites, open-source et légères** pour s'adapter à des contraintes étudiantes (ordinateur portable peu puissant, budget nul). Par exemple, le modèle de langage utilisé pour le dispatcher est un modèle pré-entraîné compact qui peut tourner sur CPU, et les services tiers choisis offrent des quotas gratuits ou des données libres d'accès. L'interface utilisateur a été développée avec Streamlit afin de prototyper rapidement une application web locale interactive.

Dans ce rapport, nous allons d'abord exposer le **contexte et la problématique** de la mobilité urbaine (défis visés et état de l'art des solutions existantes). Nous détaillerons ensuite la **préparation des données et le modèle de classification** utilisé pour le dispatcher, incluant la collecte de données, l'entraînement et les performances obtenues. La troisième partie décrira en profondeur l'**architecture logicielle et le fonctionnement de chaque fichier et composant** du projet (dispatcher, agents spécialisés, interface), avec des explications pédagogiques et des exemples. Enfin, nous discuterons de l'**organisation du projet** (planning, diagramme de Gantt) ainsi que des **aspects innovants** du couplage LLM+agents, en soulignant les choix techniques motivés par les contraintes matérielles et financières, les points forts de la solution et les perspectives d'évolution.

1. Contexte, problématique et état de l'art

1.1 Contexte et besoins utilisateur

Le projet s'inscrit dans le quotidien d'un utilisateur urbain (par exemple un étudiant) qui doit planifier ses déplacements et sorties. Plusieurs types d'informations lui sont utiles :

- **La météo** : savoir s'il fera beau, s'il faut prendre un parapluie, etc. Beaucoup consultent des applications météo avant de partir, d'autant plus quand le temps est changeant (ex. 3°C le matin et 20°C l'après-midi le même jour).
- **Les trajets en transports en commun** : identifier le meilleur itinéraire en bus, métro, train, éventuellement en combinant plusieurs modes (multimodal). Cela nécessite souvent d'ouvrir des applis comme Citymapper ou des sites d'horaires (SNCF, RATP...).
- **Les idées de sorties (culture & loisirs)** : rechercher des événements culturels, des lieux à visiter, des activités à faire pendant son temps libre, via des blogs, réseaux sociaux ou sites spécialisés.

L'utilisateur doit jongler entre ces sources, ce qui est **peu pratique et chronophage**. Or, on dispose aujourd'hui de nombreuses **données ouvertes** publiées par les organismes publics ou communautaires : par exemple, les prévisions météorologiques (ex. API libre Open-Meteo), les données de transport (API de transport.data.gouv.fr, GTFS des réseaux de transports), les agendas culturels locaux, etc. Cependant, **ces données ne sont pas intégrées dans un seul outil local** facile d'accès. L'idée est donc de **fournir un assistant unique** capable de répondre à des questions telles que « *Quel temps fera-t-il demain ?* », « *Comment aller à la Gare de Lyon en métro ?* » ou « *Que faire à Paris ce week-end ?* » dans un format conversationnel simple.

En plus d'agréger les informations, l'assistant doit pouvoir être **utilisé localement** (sur un PC portable, sans infrastructure serveur lourde) et idéalement fonctionner même si la connexion internet est limitée. Cela implique de privilégier des modèles et des solutions « *embarqués* » (local-first) dans la mesure du possible.

1.2 Défis et objectifs du projet

La problématique principale est de **créer un assistant conversationnel local** réunissant météo, trajets et idées de sorties dans une seule discussion, **sans dépendre d'un serveur externe coûteux**, afin de fluidifier la vie quotidienne d'un étudiant ou citoyen. Quatre défis majeurs ont été identifiés :

- **Compréhension du langage naturel** : L'assistant doit interpréter correctement des questions **variées, parfois ambiguës, en français (voire un peu en anglais)**. L'utilisateur ne doit pas avoir à formuler sa requête de façon technique ou structurée. **Impact attendu** : des réponses fluides et naturelles, sans exiger de mots-clés spécifiques.

Exemple: comprendre que « *Quel temps fera-t-il demain matin ?* » relève de la catégorie météo, ou que « *Y a-t-il un concert ce soir ?* » porte sur les loisirs, même si la formulation peut varier.

- **Fusion de données hétérogènes** : L’assistant combine plusieurs sources : les **données météo** (via Open-Meteo), les **données de transports** (itinéraires en transports en commun via une API telle que Google Maps Directions ou transport.data.gouv.fr), et les **données culture/loisirs** (connaissances générales ou bases d’événements). **Impact attendu** : offrir à l’utilisateur une **vue “tout-en-un”** où il peut, dans le même fil de conversation, obtenir la météo, un trajet optimisé et des suggestions de sorties.
- **Réactivité locale** : Le système doit fonctionner **localement ou avec un réseau limité**, grâce à des mécanismes de cache et à un modèle embarqué pour la compréhension des requêtes. L’objectif de performance visé est un **temps de réponse < 2 secondes** sur un PC portable classique. Cela exclut les solutions nécessitant systématiquement un appel à un serveur distant lent ou un modèle trop lourd pour un CPU. *Exigence associée* : utiliser un modèle de langage compact pour le routage, et mettre en place un cache des résultats récents.
- **Coût nul & utilisation de ressources libres** : Le projet doit s’appuyer sur des **API gratuites et des modèles pouvant tourner sur CPU**, afin de rester accessible à un étudiant sans budget cloud ni matériel spécialisé (GPU coûteux). **Impact attendu** : toute personne devrait pouvoir déployer ou utiliser l’assistant sans frais, ce qui oriente les choix technologiques vers des solutions open-source ou des services à gratuité suffisante.

Ces objectifs ont guidé la conception de l’architecture logicielle et le choix des outils, comme on le détaillera plus loin. *En résumé*, il s’agit de **réunir météo, trajets et idées de sortie dans une seule application conversationnelle locale**, en respectant les contraintes d’un projet étudiant (gratuité, rapidité, autonomie).

1.3 État de l’art et solutions existantes

Plusieurs types de solutions couvrent partiellement ces besoins, mais chacune présente des limites pour notre projet :

- **Assistants vocaux généralistes** (Google Assistant, Siri, Alexa...) : Ils offrent une interface naturelle (voix) et un écosystème riche, pouvant répondre à des questions diverses. Toutefois, ils **s’appuient sur le cloud** (internet obligatoire) et nécessitent d’envoyer les requêtes à des serveurs externes. De plus, pour un développeur, ils sont difficilement personnalisables et lents à adapter (il faut souvent formuler des *prompts* complexes). Ils peuvent accéder à la météo ou aux trajets via des services intégrés, mais entraîner un tel assistant à une problématique locale spécifique dépasse le cadre temps/budget d’un PII. Enfin, l’utilisation d’API propriétaires peut engendrer des **coûts** ou des restrictions. Pour notre projet, ces solutions étaient donc peu adaptées, du fait de l’orientation *cloud-first*, des appels API payants et du temps de développement insuffisant pour les détourner de leur usage général.

- **Applications de mobilité urbaine** (Citymapper, Transit, applications officielles) : Elles fournissent d'excellentes informations **multimodales précises** pour les transports (intégrant métro, bus, vélo, etc.). Cependant, elles se limitent au domaine du trajet et n'intègrent pas d'autres contextes : on n'y trouve pas la météo du jour ni des suggestions culturelles. L'utilisateur doit tout de même consulter séparément la météo ou d'autres sources. De plus, ces applications ne proposent pas d'interface conversationnelle unifiée. Leur avantage de précision est incontestable pour les transports, mais **leur périmètre est restreint** (pas de météo/culture) pour notre objectif.
- **Frameworks d'orchestration d'agents conversationnels** (par ex. *LangChain Agents*, 2023) : Il s'agit de bibliothèques récentes en Python qui permettent d'orchestrer des "agents" outillés de LLM. Par exemple, LangChain facilite la création d'un agent qui, selon la question, peut décider d'appeler telle ou telle fonction/API. C'est très proche de notre concept de couplage LLM+agents. L'avantage est une **infrastructure prête à l'emploi** pour gérer les enchaînements de tâches avec un LLM orchestrateur. Cependant, LangChain et consorts dépendent fortement de modèles LLM externes (souvent GPT-3.5 ou GPT-4 via l'API OpenAI). Cela va à l'encontre de l'objectif *local-first*, car à chaque requête l'agent enverrait potentiellement le prompt au cloud, entraînant latence et coût. Vu notre contrainte de gratuité et de fonctionnement local, utiliser LangChain n'était pas optimal. Il était plus cohérent de **repartir de zéro pour maîtriser la pile localement** et respecter le budget étudiant (0 €).
- **Outils d'orchestration avancée multi-LLM** (Microsoft *Autogen*, 2024) : Autogen propose de faire collaborer plusieurs agents pilotés chacun par un LLM, selon des rôles différents (par ex. un agent qui pose des questions à un autre agent). C'est intéressant pour des scénarios complexes, mais cela impose l'usage de services Azure et engendre de nombreux appels de *tokens* (coûteux). Pour un projet à budget nul, cette approche est écartée d'office.
- **Travaux de recherche récents – Router-LLM (ACL 2024)** : Des publications proposent des modèles de **routage automatique de requêtes** vers des modules spécialisés, avec des performances élevées (par ex. $F1 \approx 0,88$ pour Router-LLM). Cependant, ces solutions nécessitent souvent des **infrastructures matérielles très lourdes** pour être reproduites. Router-LLM, par exemple, a été entraîné sur un GPU A100 de 80 Go, ce qui dépasse de loin les capacités d'un PC portable standard. Ces approches ne sont donc pas reproductibles dans notre contexte sans moyens matériels considérables.

Compte tenu de la situation actuelle, nous avons **choisi de développer notre propre solution** sur mesure, en tirant parti des idées d'orchestration d'agents mais avec une **approche local-first et open-source**. L'assistant s'appuiera sur un petit modèle local pour le routage, et sur des API gratuites pour les données, en n'utilisant un LLM externe (GPT) que lorsque c'est nécessaire et en quantité limitée. Cette stratégie vise à obtenir un prototype fonctionnel **sans coût et tournant sur CPU**, tout en bénéficiant de la puissance des LLM pour la compréhension du langage. En somme, *plutôt que d'utiliser une solution toute faite non adaptée aux contraintes, le projet a été développé from scratch pour mieux contrôler les aspects locaux et budgétaires.*

Avant de plonger dans l'implémentation, nous allons détailler la constitution du **jeu de données** et le **modèle de classification** qui forment le cœur du dispatcher. Ce composant est crucial car il assure la **compréhension** (objectif 1) et contribue à la **réactivité locale** (objectif 3) en évitant d'interroger un LLM distant pour chaque requête.

2. Données et modèle de classification du dispatcher

Afin d'acheminer les questions au bon agent, le système dispose d'un **dispatcher** intelligent. Ce dispatcher est essentiellement un **modèle de classification de requêtes** en quatre catégories (« transport », « météo », « culture », « loisirs »). Nous détaillons ci-dessous la **constitution du corpus d'entraînement** pour ce modèle, le **choix du modèle** lui-même, puis le **fine-tuning** réalisé et les **performances obtenues**.

2.1 Collecte et préparation des données d'entraînement

Trouver des données d'entraînement adaptées a été une étape cruciale. L'approche retenue a été de **scraper des questions réelles sur Reddit** pour constituer un corpus de questions variées en français, annotées par catégorie de thème. Reddit est un forum en ligne contenant de nombreuses questions posées par des internautes, ce qui en fait une source riche en langage naturel informel.

Un script Python (`training_data_searching.py`) a été développé pour automatiser la collecte et le nettoyage des données. Voici les principales étapes de ce pipeline :

1. **Récupération de titres de posts Reddit** : Le script utilise la librairie **PRAW** (Python Reddit API Wrapper) pour interroger l'API Reddit. Pour chaque catégorie cible (transport, météo, culture, loisirs), on définit une liste de *subreddits* pertinents et des *mots-clés* associés. Par exemple, pour le transport on va chercher dans des subreddits comme *france* ou *voyage* des posts dont le titre contient des mots comme *train*, *bus*, *voiture*, etc. Le script effectue une requête de recherche sur chaque subreddit, triée par récents (`sort="new"`), avec une limite (jusqu'à 1500 résultats). Les titres collectés sont filtrés : on ignore ceux issus de subreddits privés (erreur 403), ceux déjà vus (on maintient un ensemble *seen*), les titres trop courts ou ne contenant pas de ? (on ne garde que de vraies questions). On s'assure aussi via des **expressions régulières** que le titre contient bien un mot-clé attendu pour la catégorie (par ex., pour la catégorie *météo*, le titre doit contenir un mot comme *météo*, *pluie*, *neige*...). Ce filtrage par regex garantit que le label attribué (le subreddit/thème visé) est cohérent avec le contenu de la question. On collecte jusqu'à `max_per_label` titres par catégorie – une limite fixée à 1000 dans notre cas pour équilibrer le corpus.

Exemple: pour la catégorie *loisirs*, après une première passe de collecte, si moins de 1000 questions ont été trouvées, le script lance une seconde passe de recherche en élargissant les critères (par ex. tri par pertinence sur l'année) afin d'atteindre davantage de questions.

2. **Nettoyage des textes** : Chaque titre récupéré est nettoyé via une fonction `clean()` qui supprime les sauts de ligne, espaces superflus, caractères spéciaux ou accents mal encodés. On obtient ainsi des phrases bien formatées sur une seule ligne.
3. **Étiquetage automatique** : Les questions sont automatiquement **annotées d'un label** correspondant au thème en cours (transport, météo, etc.) avant d'être stockées. Cette annotation est fiable grâce aux regex de mots-clés (`_PATTERNS`) qui ont servi au filtrage initial. Chaque entrée est sauvegardée sous la forme d'un petit dictionnaire JSON avec le texte de la question et le label.
4. **Ajout de données externes pour équilibrage** : Pour certaines catégories, Reddit ne fournissait pas assez de diversité. En complément, des fichiers manuels `questions_meteo.jsonl`, `questions_loisir.jsonl`, etc., ont été ajoutés. Ils contiennent des exemples de questions rédigées manuellement ou provenant d'autres sources, afin d'**équilibrer le corpus** dans les thèmes sous-représentés (notamment météo et loisirs). Le script charge ces fichiers additionnels et y pioche des questions supplémentaires en s'assurant qu'elles ne dupliquent pas des titres déjà vus.
5. **Équilibrage minimal** : Pour éviter que certaines classes n'aient qu'un seul exemple (ce qui poserait problème lors du *split*), on applique une règle : si après collecte une classe a moins de 2 exemples, on duplique aléatoirement une ou deux questions de cette classe pour en avoir au moins 2. Ceci concerne surtout des cas extrêmes où un thème aurait été quasi absent.
6. **Mélange et découpage Train/Validation** : Le jeu de données final est mélangé aléatoirement puis découpé en deux : **80 %** des questions pour l'entraînement, **20 %** pour la validation. Les résultats sont enregistrés dans deux fichiers au format JSON Lines : `train.jsonl` et `val.jsonl`. Chaque ligne est un JSON avec un champ *text* et *label*.

Au terme de ce processus, nous disposons d'un **corpus d'environ 1200 questions**, réparties en 4 classes à peu près équilibrées (grâce aux limites fixées et ajouts manuels). **100 % des questions sont en français** – un point important pour que le modèle entraîné soit performant sur notre langue cible. À noter qu'une approche envisagée était d'utiliser la **traduction automatique** pour enrichir le corpus (par ex. traduire des questions anglaises vers le français), mais cela a été abandonné car jugé trop bruyant et nécessitant de passer par une autre API externe, en contradiction avec notre contrainte de simplicité.

Pourquoi Reddit ? Reddit a été choisi car il offre des **données réelles et variées** en langage naturel. Une alternative aurait été de générer des questions synthétiques avec un modèle (ex. en demandant à GPT de créer des questions type), mais cela risquait d'introduire des formulations artificielles peu naturelles, dégradant la qualité de l'apprentissage. De plus, utiliser GPT pour générer le corpus aurait impliqué un coût ou une dépendance supplémentaire. Reddit fournit gratuitement un volume suffisant de questions authentiques, même s'il a fallu un certain temps pour trouver les bons subreddits et contourner les limitations (subreddits privés, etc.). L'accès à l'API Reddit requiert de **créer une application script** sur Reddit (pour obtenir un `client_id` et `client_secret`). Pour cela, le mode développeur de Reddit a été utilisé (via `create an app` en choisissant le type *script*) afin d'obtenir les identifiants nécessaires.

En sortie de cette phase, on obtient donc un corpus *train/val* annoté qui va servir à entraîner le modèle de classification (le dispatcher). Avant l'entraînement, on peut vérifier que chaque classe contient un nombre suffisant d'exemples (ce qui était le cas, grâce à la limite de 1000 par classe) et un contenu cohérent.

2.2 Choix du modèle de classification (SBERT vs baseline)

Pour classifier les questions de l'utilisateur en quatre catégories, plusieurs approches étaient envisageables. Un choix important a été de privilégier un modèle **léger, efficace en contexte multilingue (français) et tournant raisonnablement sur CPU**. Nous avons retenu un modèle de type **Sentence-BERT (SBERT)**, plus précisément le modèle pré-entraîné "*paraphrase-multilingual-mpnet-base-v2*". Il s'agit d'un modèle de Transformeur (dérivé de BERT) capable de **produire des embeddings vectoriels** représentatifs du sens de phrases entières. Ce modèle est **multilingue**, donc adapté au français, et il est relativement compact (environ 110 millions de paramètres) comparé à des LLM comme GPT-3.5/GPT-4. En pratique, il était **suffisamment léger pour fonctionner sur mon ordinateur personnel** tout en offrant de bonnes performances sémantiques.

Nous avons tout de même mis en place une **méthode de référence (baseline)** pour comparer : une simple **régression logistique** entraînée sur des représentations plus basiques des phrases (par exemple, des vecteurs de caractéristiques ou des embeddings statiques). Cette baseline a été entraînée et sauvegardée (`clf.pkl`) pour évaluer ses performances par rapport à SBERT. L'hypothèse était que SBERT, étant un modèle beaucoup plus expressif qui comprend la sémantique des phrases, surclasserait la régression logistique qui a une capacité de modélisation plus limitée.

Pourquoi SBERT ? Un classifieur linéaire comme la régression logistique sur sac de mots ou sur TF-IDF aurait pu suffire pour distinguer des catégories assez distinctes (par ex., le mot « métro » indique clairement la catégorie transport, « concert » la catégorie loisirs, etc.). Cependant, nous voulions que le dispatcher puisse saisir des tournures plus subtiles ou des questions imprévues. SBERT fournit un **embedding dimensionnel** d'une phrase, capturant son sens global, ce qui lui permet de généraliser à des formulations variées. Par exemple, « *Quel temps fera-t-il demain ?* » et « *Prévisions météo pour demain* » devraient idéalement se projeter proche l'un de l'autre dans l'espace vectoriel SBERT, facilitant leur classification en « météo ». De plus, le modèle choisi étant déjà entraîné sur de vastes données multilingues, il apporte un bagage de compréhension qui accélère l'apprentissage sur notre corpus spécifique.

Un autre point déterminant était la capacité à **embarquer** le modèle. SBERT peut être utilisé via la bibliothèque *SentenceTransformers* de façon locale, et on peut réduire la charge en ne gardant que l'inférence. En le *fine-tunant* sur notre tâche, on adapte légèrement ses paramètres pour la classification ciblée.

En somme, le dispatcher utilise la **combinaison** d'un **backbone SBERT** pour encoder la requête en vecteur, suivi d'une **tête de classification légère** (quelques couches linéaires)

pour prédire la catégorie. Cette architecture hybride profite du meilleur des deux : la richesse du modèle de langue SBERT et la simplicité/rapidité d'un petit classifieur entraîné sur mesure.

2.3 Fine-tuning du modèle SBERT pour le dispatcher

Le *fine-tuning* consiste à **ré-entraîner partiellement un modèle pré-entraîné** sur une tâche spécifique. Dans notre cas, nous avons fine-tuné SBERT pour qu'il devienne un **“router” de questions** capable de distinguer nos 4 catégories. Ce choix a été motivé par le constat que certaines questions de formulation simple n'étaient pas toujours bien classées par le modèle SBERT de base (ex. « Quel temps fait-il ? » pouvait être mal classé avant affinage). En spécialisant SBERT sur notre corpus, on améliore sa précision.

Le script `finetune_dispatcher.py` réalise cet entraînement. Voici ses étapes principales et les **choix d'implémentation** effectués :

- **Chargement du modèle pré-entraîné** : On commence par instancier le modèle SBERT multilingue (*paraphrase-multilingual-mpnet-base-v2*) via `SentenceTransformer`. Ce modèle fournit une méthode `encode()` pour obtenir l'embedding 768 dimensions d'une phrase.
- **Préparation du modèle pour le fine-tuning** : Le modèle SBERT comporte 12 couches Transformer. Par défaut, on pourrait soit tout figer, soit tout entraîner. Ici, on a décidé de **figer la plupart des couches et de ne dégeler que les 4 dernières couches** du modèle. Concrètement, on met `requires_grad=False` sur tous les paramètres puis on remet `True` pour ceux des couches 8 à 11 (couches les plus hautes, en comptant à partir de 0). L'intuition est que les premières couches captent des caractéristiques très générales (qu'on veut conserver), tandis que les dernières couches peuvent être ajustées pour capturer les nuances propres à nos catégories de question. En ne fine-tunant que 4 couches sur 12, on réduit aussi le temps d'entraînement et le risque de sur-adaptation, tout en permettant au modèle de se spécialiser.
- **Définition de la tête de classification** : On ajoute au-dessus de SBERT un petit réseau neuronal (notre *classifier*). Il s'agit d'un **réseau fully-connected** avec une couche cachée de taille 256, une activation non-linéaire (ReLU ou GELU), et une couche de sortie à 4 neurones (une par classe). On applique également du **dropout** (technique de régularisation) sur 2 niveaux pour éviter le sur-apprentissage. Dans notre implémentation, on a mis un dropout de 0.3 avant la première couche linéaire et un second dropout de 0.3 avant la couche finale. Le dropout force le modèle à ne pas dépendre de certaines connexions de façon déterministe, améliorant la généralisation. Ces valeurs (256 neurones, 0.3) sont des paramètres classiques choisis par expérimentation rapide : une taille de couche plus grande aurait pu apporter marginalement plus de capacité, mais 256 était un bon compromis rapidité/précision, et 0.3 de dropout est une valeur courante pour ce type de tâche.
- **Entraînement sur CPU** : Le script détecte s'il y a un GPU disponible, sinon utilise CPU. Dans notre cas, l'entraînement a dû se faire sur CPU, ce qui est l'étape la plus longue du projet (quelques heures). Pour compenser, on a utilisé un **batch size de 32**

(nombre de phrases traitées en parallèle) : c'est un bon équilibre entre vitesse (plus haut batch = mieux pour paralléliser) et stabilité (un batch trop grand pourrait nécessiter plus de mémoire ou réduire la variabilité des gradients). Nous avons entraîné pendant **8 epochs** (passes sur l'ensemble du jeu de données), ce qui s'est avéré suffisant pour que le modèle converge sans commencer à sur-apprendre (au-delà, on aurait vu la perte de validation remonter). Ces paramètres (batch 32, 8 époques) ont été choisis initialement d'après des pratiques communes, puis validés empiriquement sur ce projet.

- **Taux d'apprentissage différenciés** : Comme souvent en fine-tuning, on utilise deux taux d'apprentissage distincts : un plus élevé pour la partie *tête de classification* nouvellement initialisée, et un plus faible pour les couches du modèle pré-entraîné afin de ne pas trop les perturber. Nous avons utilisé **LR_HEAD = 2e-4** (0.0002) pour la tête, et **LR_BERT = 2e-5** (0.00002) pour SBERT. Ainsi, la tête de classification s'adapte rapidement aux données (elle part de zéro, on veut qu'elle apprenne vite), tandis que SBERT est ajusté en douceur. Ces valeurs ont été confirmées par des tests : un LR_BERT plus grand risquait d'exploser la perte, et un LR_HEAD plus petit ralentissait inutilement l'apprentissage de la tête.
- **Loss function adaptée au déséquilibre** : Bien que nous ayons cherché à équilibrer les classes, il peut rester un léger déséquilibre (par ex. si "culture" a un peu moins d'exemples que "transport"). Pour éviter que le modèle favorise la classe majoritaire, on utilise une fonction de coût *Cross-Entropy* avec des **poids inversement proportionnels à la fréquence des classes**. Concrètement, on calcule la distribution des labels dans le train (`counts`), puis on définit un poids par classe = *total exemples / count_classe*. Ainsi les erreurs sur les classes rares sont davantage pénalisées. On fournit ce vecteur de poids à `nn.CrossEntropyLoss` de PyTorch. Cette technique assure que chaque catégorie compte de façon équilibrée dans le calcul de la perte, optimisant la **balanced accuracy** plutôt que la seule accuracy brute.
- **Dataloader et encodage en batch** : On a créé un `DataLoader` PyTorch pour parcourir les données d'entraînement (`train_ds`) et de validation (`val_ds`). À chaque batch, on doit convertir une liste de phrases en leurs embeddings SBERT. Pour cela, une fonction `collate_fn` utilise `backbone.encode(list_of_texts)` pour obtenir directement un tenseur des embeddings du batch. Cette opération est faite sur le CPU (ou GPU si dispo) en flottant. On récupère également les labels du batch convertis en tenseur. Ainsi, durant l'entraînement, au lieu de recalculer l'embedding mot par mot, on profite de la vectorisation fournie par `SentenceTransformer` pour encoder tout un batch en parallèle, accélérant l'époch.
- **Optimiseur et scheduler** : Nous avons utilisé l'optimiseur **AdamW** (Adam avec régularisation de poids) avec un **weight decay** global de 0.0. On configure deux groupes de paramètres : (a) la tête de classification ($lr=2e-4$) et (b) les paramètres SBERT dégelés ($lr=2e-5$). De plus, on exclut de la pénalisation L2 (weight decay) les poids de bias et de LayerNorm de SBERT, selon la pratique courante. Un *scheduler* linéaire avec *warm-up* a été employé : sur les 6% premiers pas d'entraînement, le taux d'apprentissage monte linéairement de 0 à la valeur définie, puis redescend linéairement jusqu'à la fin. Cela aide à stabiliser l'apprentissage en début d'entraînement (éviter un départ trop brutal pouvant nuire).

- **Boucle d’entraînement et early stopping** : À chaque epoch, le modèle calcule la perte moyenne sur le train, puis évalue sur le set de validation en calculant plusieurs métriques : accuracy, balanced accuracy et F1-macro. On suit particulièrement la **balanced accuracy** (moyenne des taux de bonne classification par classe) comme indicateur principal, car elle reflète mieux la performance sur chaque catégorie, y compris les moins fréquentes. Un mécanisme d’**early stopping** avec patience de 5 epochs sans amélioration a été implémenté : on garde en mémoire la meilleure balanced accuracy obtenue jusqu’alors, et si pendant 5 epochs consécutives on ne la bat pas, on arrête l’entraînement pour éviter de sur-ajuster. Dans le cas présent, le training s’est arrêté automatiquement dès que la performance s’est stabilisée. À chaque amélioration de *bal_acc*, on sauvegarde un **checkpoint** (fichier `dispatcher_sbert.pt`) contenant : les poids du modèle SBERT (backbone), les poids de la tête de classification, et le mapping label→id. Cette sauvegarde sert ensuite à charger le modèle fini dans l’application.
- **Surveillance du sur-apprentissage** : Durant l’entraînement, on observait la courbe de perte d’entraînement et de validation pour détecter tout sur-apprentissage (*overfitting*). Grâce aux techniques ci-dessus (dropout, early stopping, weight decay), on a réussi à éviter l’overfit manifeste. Il y a eu des tests exploratoires initialement où un nombre d’epochs trop élevé ou un manque de régularisation menaient à un sur-apprentissage (la perte de validation remontait alors que celle de train continuait de baisser), ce qui a guidé le choix final des paramètres.

En parallèle, pour référence, une régression logistique a été entraînée sur les embeddings SBERT statiques ou d’autres features pour produire `clf.pkl`. Ce modèle baseline a obtenu des performances correctes mais inférieures au modèle fine-tuné, comme attendu (nous mentionnerons les résultats comparatifs dans la section suivante).

2.4 Performances obtenues et analyse

Le modèle de dispatcher fine-tuné a atteint d’excellents résultats sur la classification des requêtes. Sur le jeu de validation, on mesure environ **94 % d’exactitude (accuracy)** globale et surtout une **balanced accuracy d’environ 0,97**. Cela signifie que le modèle **prédit correctement chaque catégorie**, y compris les moins représentées, dans 97 % des cas en moyenne, ce qui dépasse largement l’objectif initial (acheminer ≥ 90 % des requêtes vers le bon agent). En d’autres termes, le dispatcher satisfait le critère de compréhension (Défi 1) avec une belle marge.

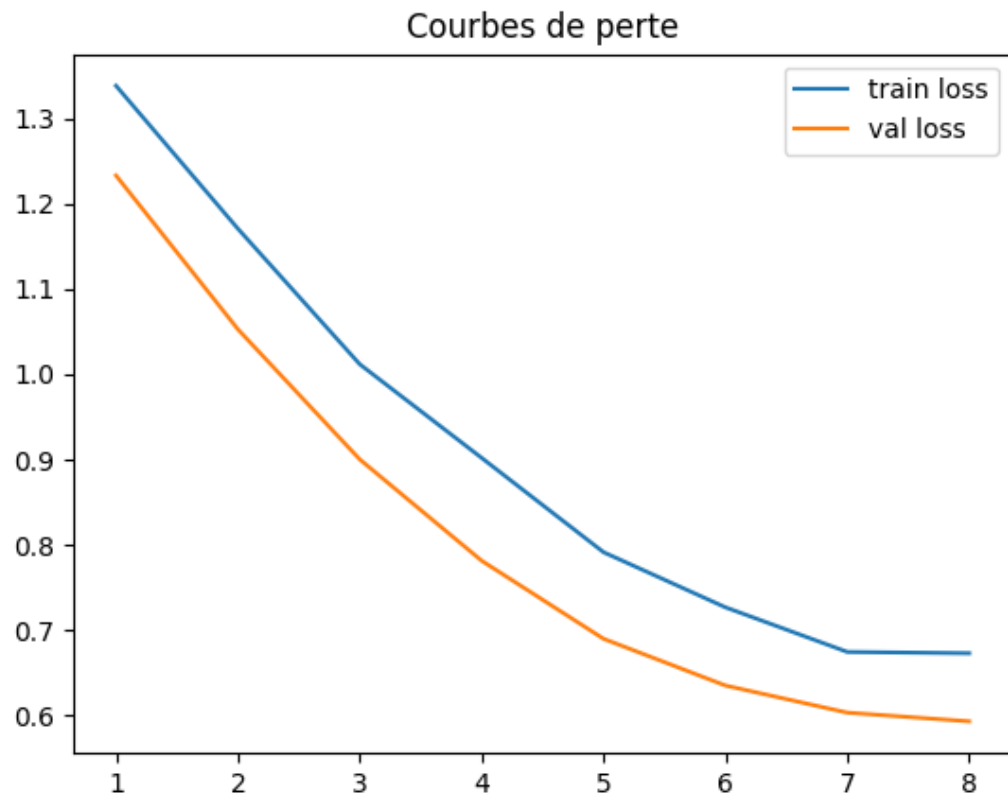


Figure 1 : Perte au fil des epochs

Les **courbes de perte** d'entraînement et de validation au cours des epochs confirment un **apprentissage progressif et sans sur-apprentissage** : la perte diminue régulièrement de ~1.3 initialement à ~0.6 en fin d'entraînement, aussi bien sur le train que sur la validation. Aucune remontée brutale n'est observée sur la courbe de validation, signe que le modèle généralise bien et n'a pas mémorisé par cœur le jeu d'entraînement. De plus, la **perte de validation reste en permanence en dessous de la perte d'entraînement**, ce qui est un bon indicateur d'un modèle qui n'overfit pas (souvent en cas d'overfit la perte val dépasse celle de train). On en conclut que notre fine-tuning a été bien géré et s'est arrêté au bon moment (grâce à l'early stopping).

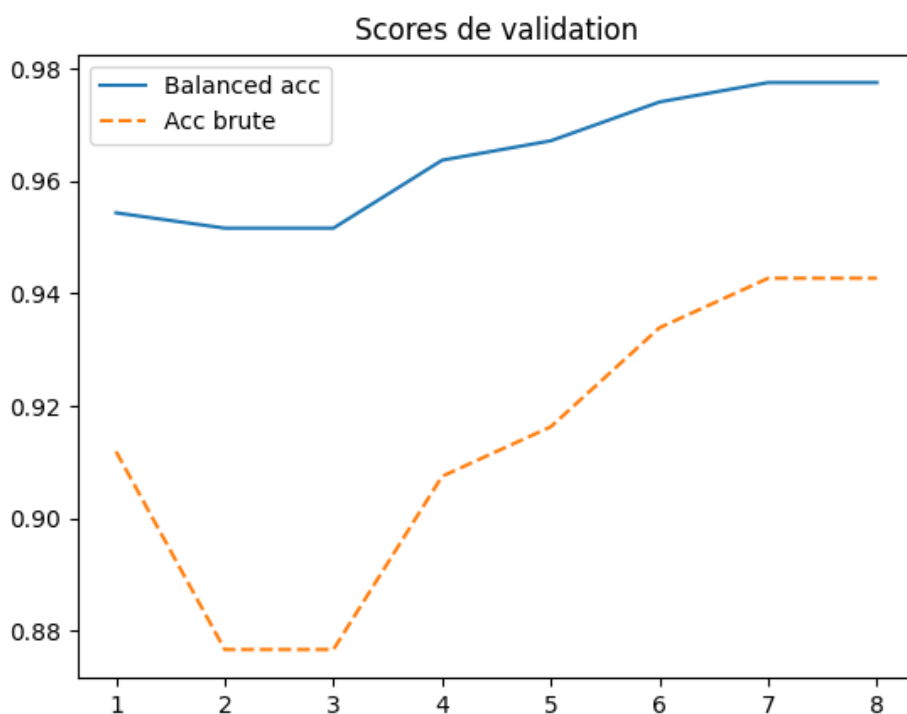


Figure 2 : Validation au fil des epochs.

En termes de métriques par classe, l'usage de la balanced accuracy met en lumière que **toutes les catégories sont bien apprises**. Une accuracy brute de 94 % pourrait cacher un déséquilibre (par ex. prédire toujours « transport » si 50 % des questions sont transport donnerait 50 % accuracy mais ignorerait les autres classes). Ici, balanced acc ~97 % implique que même les classes minoritaires (meteo, loisirs par ex.) sont reconnues presque autant que les majoritaires. Le modèle ne se contente donc pas de deviner la classe la plus fréquente pour maximiser l'exactitude globale : il traite équitablement chaque intention utilisateur.

La **baseline régression logistique** utilisée en comparaison a, quant à elle, atteint une performance plus modeste (non détaillée dans le rapport original, mais typiquement une accuracy dans les ~80 %). Elle a servi de point de contrôle pour valider l'apport du fine-tuning SBERT. Comme espéré, le dispatcher SBERT fine-tuné est **nettement supérieur** à cette baseline sur nos données, démontrant l'intérêt d'un modèle de langue sophistiqué pour comprendre les questions ouvertes.

En résumé, le dispatcher entraîné répond aux attentes : il est capable de router correctement la grande majorité des requêtes. Cela constitue le cœur du système, car toute erreur de classification se traduirait par une réponse hors sujet de la part d'un agent inapproprié. Grâce au fine-tuning, notre assistant est robuste à différentes formulations. Par exemple, « *Comment*

aller de la gare au musée ? » sera détecté comme un cas *transport*, « *Que faire ce soir ?* » comme *loisirs*, « *Est-ce qu'il va pleuvoir demain ?* » comme *météo*, etc. Nous verrons dans la section suivante que ce modèle est intégré au **dispatcher logiciel** avec des mécanismes additionnels (seuils de confiance, fallback par mots-clés) pour encore améliorer la fiabilité du routage.

Avant cela, notons que le fichier de poids final `dispatcher_sbent.pt` contient tout le nécessaire pour reconstruire le modèle (architecture SBERT + classifieur) et le mapping des labels. Il est chargé au lancement de l'assistant afin d'éviter de tout réentraîner à chaque fois.

Avec un dispatcher bien entraîné, passons maintenant à la **conception logicielle globale** de l'assistant, en décrivant chaque composant (fichier de code) et comment ils interagissent.

3. Architecture logicielle et fonctionnement des agents

L'architecture logicielle du projet est pensée de manière **modulaire**, autour d'un **dispatcher central** et de **plusieurs agents spécialisés** (météo, transports, culture, loisirs). Chaque agent traite les requêtes de sa thématique de façon indépendante, tandis que le dispatcher fait le lien entre la question de l'utilisateur et le bon agent. Cette séparation claire des préoccupations facilite les évolutions futures (on pourrait ajouter un nouvel agent sans refondre le reste, ou améliorer un agent indépendamment).

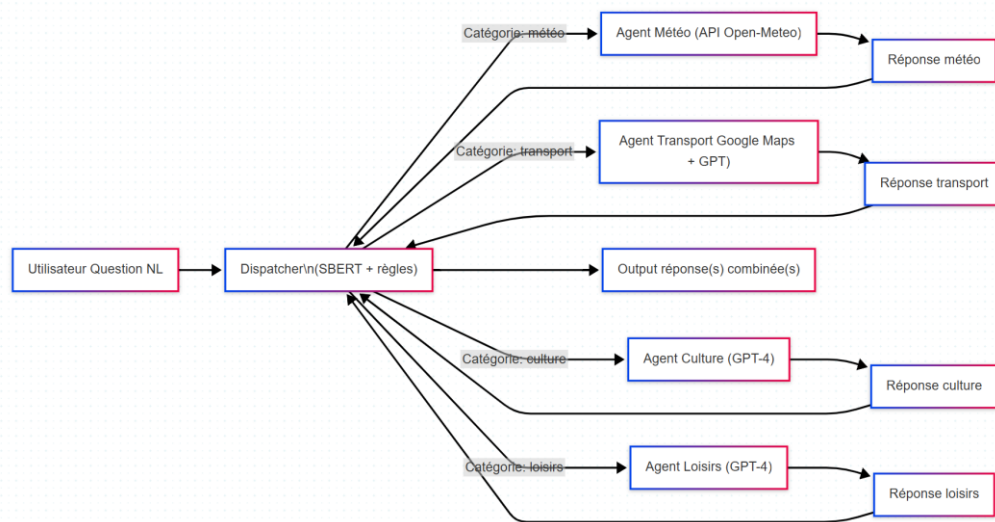


Figure 3 : Architecture globale de l'assistant.

Un dispatcher central oriente la question NL de l'utilisateur vers un ou plusieurs agents spécialisés selon la catégorie identifiée. Chaque agent retourne sa réponse structurée, que le dispatcher combine et renvoie à l'utilisateur dans la conversation.

Comme l'illustre la figure 1, lorsqu'une question est posée, elle passe d'abord par le **Dispatcher**. Celui-ci va déterminer la ou les catégories pertinentes et déléguer le traitement aux **agents correspondants**. Par exemple, une question « *Quel temps fera-t-il demain à Toulouse ?* » est catégorisée “météo” par le dispatcher, et transmise à l'agent Météo, qui consulte une API météo et renvoie la réponse (ex. « *Demain à Toulouse, ciel partiellement nuageux, 18°C...* »). Pour une question « *Comment aller de Gare de Lyon à Eiffel ?* », le dispatcher détecte un cas “transport” et appelle l'agent Transport, qui interroge l'API Google Maps et renvoie un itinéraire détaillé.

Le dispatcher peut, dans certains cas, identifier **plusieurs catégories** pour une seule question si celle-ci semble ambiguë ou multi-thématique. Il contactera alors plusieurs agents et agrégera leurs réponses (chacune taguée par le domaine). Par exemple, une question un peu floue contenant à la fois le mot *météo* et *concert* pourrait théoriquement activer deux agents (météo et loisirs) et répondre aux deux aspects. Cela dit, l'assistant gère pour l'instant surtout **une intention par requête** (voir limites en section 4.4).

Les **agents spécialisés** sont implémentés chacun dans un fichier Python du dossier `agents/` : `weather_agent.py`, `transport_agent.py`, `culture_agent.py` et `loisirs_agent.py`. Un cinquième fichier `dispatcher.py` gère la logique du dispatcher lui-même et instancie les agents.

Enfin, l'interface utilisateur (fichier `ui_app.py`) fournit une **interface web** (via Streamlit) qui orchestre les interactions en coulisses avec le dispatcher. Elle gère la session de chat, la présentation des réponses et des fonctionnalités annexes (géolocalisation, options de l'utilisateur).

Dans les sous-sections qui suivent, nous expliquons en détail le contenu et le rôle de chaque fichier/fonction :

- **Dispatcher** (`agents/dispatcher.py`) – Chargé de la classification des requêtes (via SBERT) et du routage vers les agents.
- **Agent Météo** (`agents/weather_agent.py`) – Interroge l'API Open-Meteo pour donner la météo actuelle.
- **Agent Transport** (`agents/transport_agent.py`) – Utilise l'API Google Maps pour les itinéraires et fait appel à un LLM pour certaines tâches de compréhension.
- **Agent Culture** (`agents/culture_agent.py`) – S'appuie sur GPT-4 pour répondre sur le patrimoine/histoire.
- **Agent Loisirs** (`agents/loisirs_agent.py`) – S'appuie sur GPT-4 pour répondre sur les activités/événements.
- **Interface Utilisateur** (`ui_app.py`) – Application Streamlit gérant le chat, la géolocalisation et l'affichage.
- **Programme principal** (`main.py`) – Une alternative en console (mode test) pour échanger avec l'assistant sans l'interface graphique.

3.1 Le Dispatcher : routage des requêtes via SBERT et fallback

Le dispatcher est le **cœur décisionnel** du système. Implémenté dans `dispatcher.py`, il remplit deux fonctions essentielles : **classifier la requête utilisateur** et **appeler l'agent approprié** en fonction de cette classification. On peut le voir comme le chef d'orchestre qui *comprend* et *dirige*.

Structure interne : Lors de l'initialisation, le dispatcher charge le modèle SBERT fine-tuné et prépare les agents :

- **Chargement du modèle entraîné** : Le fichier de checkpoint `dispatcher_sbert.pt` sauvegardé lors du fine-tuning est chargé via `torch.load`. On récupère le dictionnaire `label2id` qui mappe chaque label ("transport", "météo", etc.) à un identifiant numérique, ainsi que les poids du backbone SBERT et de la tête de classification. Le code crée une instance du modèle SBERT pré-entraîné puis y charge les poids fine-tunés. De même, la tête de classification (définie comme un petit réseau `nn.Sequential` identique à celui utilisé à l'entraînement) est instanciée et on y charge ses poids entraînés. Enfin, on met ces modules en mode évaluation (`eval()`) car on ne va plus les entraîner. À ce stade, le dispatcher a en mémoire tout le réseau de classification prêt à l'usage.
- **Initialisation des agents** : Le dispatcher crée une instance de chaque agent spécialisé et les stocke dans un dictionnaire `self.agents` pour y accéder facilement par nom de catégorie. Par exemple, `self.agents["transport"] = TransportAgent()`. Ainsi, pas besoin de recréer les agents à chaque question : ils restent vivants et potentiellement conservent un état (notamment les agents culture/loisirs qui gardent l'historique de conversation).
- **Préparation des regex de secours** : Le dispatcher définit en classe un dictionnaire `_KEYWORDS` associant à chaque catégorie une expression régulière typique (mots-clés fréquents de ce domaine). Par ex. pour *transport* : `(bus|métro|train|tram|rer|itinéraire|trajet|covoiturage|taxi|aller|voyager|prendre)`. Lors de l'init, il compile ces regex en objets prêts à l'emploi (`self._kw_regex`). Ce mécanisme servira en *filet de sécurité* si le modèle ML n'est pas sûr de sa réponse.

Une fois initialisé, le dispatcher offre principalement deux méthodes publiques :

- `classify_request(text: str) -> List[str]` : qui renvoie une liste de catégories prédites pour la requête.
- `route_request(user_input: str) -> str` : qui utilise `classify_request` puis appelle le(s) agent(s) et retourne la réponse combinée.

Voici le déroulement lorsqu'une question utilisateur est traitée :

(a) Encodage et prédiction SBERT : `classify_request` commence par passer le texte dans `_sbert_predict()`. Cette fonction privée effectue :

- Encodage de la phrase en embedding SBERT via `self.backbone.encode` (en ne calculant pas de gradients).
- Passage de cet embedding dans la tête de classification `self.clf` pour obtenir les *logits* (scores bruts avant softmax).
- Application d'une softmax pour obtenir des *probabilités* sur les 4 classes.
- Identification de la classe majoritaire : index du plus grand score (`idx_main`) et sa probabilité associée (`score`). On mappe cet index à son label texte via `id2label`.
- Identification des éventuelles **secondes classes pertinentes** : on parcourt toutes les classes autres que la principale et on retient celles dont la probabilité dépasse un **seuil secondaire** fixé (ici 0.35). Ces classes "secondaires" sont celles que le modèle voit aussi relativement probables. Par exemple, si une question obtient 0.40 de score en transport et 0.38 en culture, transport est majoritaire mais culture est juste derrière > 0.35, donc on retient les deux.

Le résultat de `_sbert_predict` est donc (`label_principal`, `score_max`, [`liste_secondaires`]). Le dispatcher logge en debug ces infos pour trace (ex. [SBERT] 'question' → main: transport (0.42), secondaries: ['culture']).

(b) Application du seuil de confiance : De retour dans `classify_request`, on examine le score de la classe principale : s'il est **inférieur au seuil** `self.threshold (0.50)`, on estime que le modèle n'est **pas assez confiant** dans sa prédiction. Dans ce cas, on tente un *fallback* avec les regex : la méthode `_keyword_fallback` teste chaque regex de mots-clés sur la question, et si un motif correspond à une catégorie, on prend cette catégorie en retour. Par exemple, si SBERT hésite mais qu'on trouve "pluie" dans la question, on prendra "météo". Si un mot-clé est trouvé, on retourne directement [`catégorie_mot_clé`]; si aucun mot-clé ne matche, alors on n'abandonne pas pour autant le résultat SBERT : on utilisera quand même la catégorie principale de SBERT même si son score était bas (car mieux vaut tenter quelque chose que rien). Le log mentionne ces cas ([Score<seuil] fallback → ... ou bien [Score<seuil mais prise SBERT] ...).

(c) Résultat de classification : En sortie, `classify_request` renvoie une liste qui contient la **catégorie principale** suivie éventuellement des catégories secondaires retenues. Le plus souvent, ce sera une liste à un élément (une seule catégorie déterminée). Mais dans des cas où le modèle a identifié deux catégories proches avec une probabilité au-dessus de 0.35, on aura deux éléments (le principal d'abord, puis le second). Par exemple, ["transport", "loisirs"] si la question semblait parler d'aller quelque part pour un festival (transport et loisirs liés).

(d) Routage vers les agents : La méthode `route_request(user_input)` utilise `classify_request` et parcourt la liste de catégories retournée. Pour chacune, elle récupère l'instance d'agent correspondante (`agent = self.agents.get(cat)`). Il est très important de

préserver la question originale pour chaque appel – on passe la même `user_input` à tous les agents concernés (pas de modification de la question à ce stade-là, le dispatcher ne la reformule pas). Ensuite, le dispatcher appelle `agent.handle_request(user_input)` dans un bloc `try/except` pour capturer d'éventuelles erreurs d'agent. Si un agent lève une exception, on logge l'erreur et on ajoute à la réponse un message d'erreur spécifique pour cette catégorie (ex. "[Erreur] échec de traitement : ..."). Si tout se passe bien, on récupère la réponse renvoyée par l'agent.

(e) Formatage de la réponse combinée : Le dispatcher construit une liste de morceaux de réponse. Il préfixe chaque réponse agent par le nom de la catégorie entre crochets pour que l'utilisateur sache d'où vient l'information. Par exemple, si l'agent Météo renvoie "*Il fera beau demain, 25°C*", le dispatcher le transforme en "[Météo] *Il fera beau demain, 25°C*". De même pour *Transport*. S'il y a plusieurs agents appelés, on aura plusieurs lignes. Enfin, il joint ces morceaux par des sauts de ligne et renvoie le texte final.

Ainsi, le **rôle du dispatcher** côté code est double : (1) **décider de la ou des catégories** pertinentes (en combinant intelligence du modèle et règles heuristiques de secours) et (2) **appeler séquentiellement les agents** pour composer la réponse. On peut noter que l'appel de plusieurs agents pour une requête est une forme rudimentaire de gestion multi-intent. Dans la pratique, l'assistant se comporte la plupart du temps comme un routeur 1-1 (une question → un agent), car c'est comme cela que les questions sont généralement posées. Toutefois, ce design laisse entrevoir la possibilité de traiter des requêtes plus complexes en appelant plusieurs modules (ce qui est innovant comparé à un assistant classique qui ne peut traiter qu'une chose à la fois).

Optimisations et considérations :

- Le dispatcher utilise la décoration `@functools.lru_cache(maxsize=256)` sur sa méthode d'encodage `_encode`. Cela met en cache les derniers 256 textes encodés. Si un même utilisateur repose une question identique ou très proche textuellement, le calcul de l'embedding SBERT sera évité, ce qui **améliore la latence**. Dans un scénario de conversation, les questions peuvent parfois se répéter (ou l'utilisateur reformule), donc ce cache LRU est bienvenu pour accélérer le traitement sans coût mémoire prohibitif.
- Les **thresholds 0.50 et 0.35** choisis pour la prise de décision SBERT vs fallback ont été ajustés empiriquement. Un seuil principal trop haut risquerait de déclencher trop souvent le fallback par mots-clés (qui est plus fruste), tandis qu'un seuil trop bas ferait confiance au modèle même quand il n'est pas sûr. 0.5 est apparu raisonnable : cela correspond à "50 % de confiance". Le seuil secondaire 0.35 permet de ne pas ignorer une seconde intention potentielle si elle a plus de 35 % de proba. Ces valeurs ont été testées durant le développement pour trouver un bon compromis, et elles ont été mentionnées comme un des points délicats à régler dans le projet.
- Les **regex fallback** servent principalement de garde-fous pour des cas où le modèle se trompe ou hésite trop. Par exemple, si une question est très courte : « *Concert ce soir ?* », SBERT pourrait ne pas avoir assez de contexte et être incertain, mais la présence

de “concert” déclenchera le fallback vers *loisirs*. Ces regex ont été construites manuellement en se basant sur l’expérience (mots typiques par catégorie). Elles ne couvriront pas 100 % des cas, mais ajoutent un niveau de robustesse.

En conclusion, le dispatcher assure que chaque requête est aiguillée correctement. Il fournit aussi le **contexte du nom de catégorie** dans la réponse, ce qui est utile pour la transparence (l’utilisateur voit un label [Météo] ou [Transport] devant la réponse). Dans une version ultérieure, on pourrait traduire ces labels en emojis ou icônes (par ex. ☀ pour météo, 🚗 pour transport) pour une présentation plus visuelle – c’est ce qui a d’ailleurs été fait dans l’interface (ajout d’emojis, on le verra en section 3.6).

3.2 Agent Météo (`weather_agent.py`)

L’agent Météo est conçu pour répondre aux questions du type « *Quel temps fait-il à [lieu] ?* » ou « *Prévisions météo pour [lieu]* ». Son fonctionnement est assez direct : **extraire un lieu** dans la question, puis **consulter une API météo** pour ce lieu.

Les principales étapes de `WeatherAgent` sont :

- **Extraction de la ville demandée** : La méthode `extract_city(user_input)` tente de repérer un **nom de ville** dans la question. Pour cela, elle cherche des occurrences du motif “à X” ou “pour X”, où X serait une suite de mots qui ressemble à un nom de ville (lettres, éventuellement espaces ou tirets). On utilise une regex `(?:à|pour)\s+([A-Za-zÀ-ÖØ-öø-ÿ\s\-\-]+)` insensible à la casse : par exemple, dans « *Quel temps à Bordeaux ?* », le groupe capturé sera “Bordeaux”. Si une correspondance est trouvée, on retourne le nom de ville nettoyé (`strip`). Sinon, on renvoie `None`.

Cette approche simple couvre bon nombre de formulations communes (à *Paris*, pour *Montpellier*...). Elle ne gère pas toutes les possibilités (par ex. « à la Réunion » ou « sur Lyon » ne matcheraient pas), mais dans la plupart des cas l’utilisateur formulera naturellement comme “à [ville]”.

- **Appel à l’API de géocodage d’Open-Meteo** : Pour obtenir la météo d’une ville, il faut connaître ses coordonnées (latitude, longitude) à passer à l’API météo. Open-Meteo propose une **API de géocodage** pour traduire un nom de lieu en coordonnées. L’URL de cette API est <https://geocoding-api.open-meteo.com/v1/search>. La méthode `get_coordinates(city)` envoie une requête HTTP GET à cette URL avec comme paramètres : `name` (le nom de la ville), `count=1` (on ne veut qu’un résultat, le plus pertinent), `language=fr` (pour s’assurer que le nom correspond en français), et `format=json`. On utilise la librairie Python **requests** pour faire cet appel et on parse le JSON de réponse. Si au moins un résultat est trouvé, on extrait le premier qui contient typiquement `latitude` et `longitude`. Sinon, on renvoie `(None, None)` pour signaler l’échec. Ce géocodage se fait en ligne, mais l’API est ouverte et gratuite (pas de clé

nécessaire, et nombre de requêtes illimité) – ce qui est parfait pour notre contrainte de coût.

- **Gestion des erreurs d'entrée** : Si `extract_city` n'a trouvé aucune ville (None), cela signifie que la question est incomplète du point de vue de l'agent météo (ex. « *Quel temps fera-t-il demain ?* » sans préciser où). Dans ce cas, `handle_request` répond simplement à l'utilisateur **d'ajouter la ville recherchée**. La réponse est une phrase : *"Veuillez préciser la ville pour laquelle vous souhaitez connaître la météo."*. De même, si la géocodification échoue à trouver la ville (lat, lon None), on renvoie *"Impossible de trouver les coordonnées pour la ville X."*. Ainsi, l'agent gère proprement les cas où il ne peut pas avancer faute d'information.
- **Appel à l'API météo** : Si on a des coordonnées valides, on prépare l'appel à l'API **Open-Meteo Forecast**. L'URL de base est `https://api.open-meteo.com/v1/forecast`. On passe en paramètres : la latitude, la longitude, `current_weather=true` (pour obtenir la météo actuelle), et `timezone=Europe/Paris` (pour formater l'heure locale). Ici, on ne demande que la météo instantanée, pas nécessairement les prévisions horaires ou journalières pour la simplicité (on aurait pu ajouter `hourly=...` ou `daily=...` si on voulait une prévision). On effectue la requête avec `requests.get` et on parse le JSON. Si la réponse JSON contient un champ `current_weather`, on le récupère. Sinon, on considère qu'il y a eu un problème et on répond *"Erreur lors de la récupération des données météo."*
- **Interprétation du code météo** : Open-Meteo renvoie la météo courante sous forme de valeurs : température, vitesse du vent, et un code météo (entier) décrivant l'état du ciel (0 = clair, 1 = principalement clair, 61 = pluie légère, etc.). Ces codes sont très précis (et cryptiques pour l'utilisateur). L'agent contient une méthode `map_weather_code(code)` qui convertit ce code en une **description textuelle en français**. Un dictionnaire `mapping` est défini dans le code couvrant les codes les plus courants (0,1,2,3, 45,48 pour le brouillard, 51-57 bruine, 61-67 pluie, 71-77 neige, 80-86 averses, 95-99 orage). Par exemple, code 61 -> "pluie légère". Si un code inattendu apparaît (non dans le mapping), on renvoie "indéterminé". Cette conversion permet de fournir une phrase lisible à l'utilisateur.
- **Construction de la réponse** : Enfin, si tout va bien, l'agent construit une phrase du type : *"À [Ville], le temps est [description], la température est de X°C et la vitesse du vent est de Y km/h."*. On prend soin de **mettre une majuscule à la ville** (on utilise `city.capitalize()` en supposant que le nom de ville est en minuscule). Cette phrase comprend les trois infos essentielles : état du ciel, température, vent. On pourrait en ajouter (humidité, etc.), mais on a privilégié la simplicité. La phrase est retournée en tant que résultat de `handle_request`.

Cet agent **ne nécessite aucune API key** (Open-Meteo est libre) et répond très rapidement (quelques dixièmes de seconde). Il ne conserve pas d'état entre les appels (chaque requête est indépendante).

Notons que cet agent répond **uniquement pour la météo actuelle**. Si la question porte sur le futur (« *Quel temps demain ?* »), idéalement il faudrait interroger l'API avec `daily` ou

hourly forecast. Dans la version actuelle, la formulation "demain" n'est pas explicitement gérée – l'agent traitera "demain" comme un mot ignoré et donnera en fait le temps actuel (ce serait une amélioration à prévoir de traiter les dates). Le projet, pour rester dans les limites, s'est focalisé sur la météo *courante*. L'utilisateur peut toujours demander « *Et maintenant ?* » le lendemain pour avoir la mise à jour.

En conclusion, l'agent Météo est un bon exemple d'**agent spécialisé non-LLM** : il fait une tâche précise via une API dédiée et retourne une réponse formatée de manière fixe. Sa fiabilité dépend surtout de la capacité à identifier la ville, d'où l'invite faite à l'utilisateur en cas d'ambiguïté.

3.3 Agent Transport (`transport_agent.py`)

L'agent Transport est sans doute le plus complexe, car il doit traiter des questions autour des déplacements. Cela recouvre deux cas principaux :

- Des questions d'**itinéraire** (comment aller de A à B en transport en commun).
- Des questions plus **générales sur les transports** (horaires, comparaison de moyens, etc., sans itinéraire point à point précis).

Pour gérer ces deux types différemment, l'agent Transport combine à la fois de la logique et l'appel à un LLM (GPT) pour l'aider dans la compréhension. Voici son fonctionnement détaillé :

(a) Classification de la requête transport (itinéraire ou question générale) : La première étape dans `handle_request` est de déterminer le *genre* de demande transport. En effet, une question du type « *Comment aller à la gare X depuis Y ?* » nécessite de calculer un itinéraire, tandis que « *Quel est le moyen de transport le plus écologique ?* » ou « *Y a-t-il des travaux sur la ligne 1 ce week-end ?* » sont des questions générales où on n'a pas deux lieux à relier.

L'agent possède donc une méthode `classify_request(user_input: str)` qui utilise... GPT lui-même pour cette classification. Il envoie à l'API OpenAI un petit dialogue où :

- Le *system prompt* dit: "*Vous êtes un classificateur qui décide si la requête de l'utilisateur est une demande d'itinéraire (« de A à B ») ou une question générale sur les transports. Répondez strictement par ITINERARY ou GENERAL.*".
- Le *user prompt* est la question de l'utilisateur.

On utilise un modèle nommé "gpt-4o-mini" dans le code. Il s'agit probablement d'un alias interne soit pour GPT-4 avec certaines conditions, soit d'un modèle plus petit. On ne sait pas exactement, mais on peut supposer que c'est un accès à GPT-4 allégé. Le paramètre `temperature=0` est fixé pour avoir une réponse déterministe (classification = tâche bien définie, on ne veut pas de variation aléatoire). Le résultat attendu est simplement le mot "ITINERARY" ou "GENERAL". On prend cette réponse, on la met en majuscules, et on la renvoie.

Ce détour par GPT peut paraître lourd pour juste distinguer une question d'itinéraire, mais il offre une robustesse sémantique. Par exemple, « *Comment se rendre de la Tour Eiffel à Montmartre* » sera bien classé *ITINERARY*, tandis que « *Quelle ligne de métro dessert Montmartre ?* » sera classé *GENERAL* car ce n'est pas littéralement "de X à Y". En faisant appel à un LLM ici, on évite d'écrire nous-mêmes une regex ou une logique de détection de "de ... à ..." qui pourrait manquer des formulations. (On aura tout de même une étape regex ensuite, mais le GPT clarifie l'intention globale.)

(b) Cas “question générale” : Si `classify_request` retourne *GENERAL*, cela signifie que l'utilisateur ne cherche pas un itinéraire précis, mais plutôt une information transport plus ouverte. Dans ce cas, l'agent va simplement **déléguer la réponse à GPT** : on construit un prompt avec un *system message* qui dit "*Vous êtes un expert en transport. Répondez clairement à la question.*", puis on met la question de l'utilisateur en *user message*, et on appelle l'API OpenAI ChatCompletion (modèle "gpt-4o-mini" de même). Aucune température n'est explicitement mise dans ce call, donc ce sera la valeur par défaut (probablement 1, ce qui laisse GPT formuler une réponse). On récupère la réponse textuelle de GPT et on la retourne directement.

Concrètement, pour une question « *Quel est le moyen de transport le plus écologique pour voyager de Lyon à Paris ?* », GPT pourrait répondre "*Le moyen le plus écologique est généralement le train, car il émet moins de CO2 par passager que l'avion ou la voiture.*" Ce mécanisme permet de couvrir **toutes sortes de questions transports** (tarifs, écologie, comparatifs, fonctionnement, etc.) sans qu'on ait à coder nous-mêmes une logique ou une base de connaissances. L'agent Transport devient un intermédiaire qui oriente soit vers Google Maps (cas itinéraire) soit vers GPT (cas général).

(c) Cas “itinéraire” : C'est la partie la plus élaborée. Si la requête est classée *ITINERARY*, alors l'agent doit trouver **un point de départ et un point d'arrivée** dans la question, puis calculer un itinéraire entre les deux.

- **Extraction des paramètres (origine, destination) :** La fonction `extract_parameters(text: str)` tente d'extraire deux lieux de la question. Elle cherche d'abord la présence explicite de la sous-phrase "**de X à Y**" dans le texte (insensible à la casse). Si on trouve un match, on prend X comme origine et Y comme destination. C'est le scénario idéal où l'utilisateur a bien formulé "*de ... à ...*". Par exemple, « *de Gare du Nord à Tour Eiffel* » donnera `origin="Gare du Nord"`, `destination="Tour Eiffel"`.

Si le motif "de ... à ..." n'est pas trouvé, on utilise un *fallback* heuristique : on cherche **deux noms propres consécutifs** dans la question. On utilise une regex pour capturer tous les mots commençant par une majuscule (sauf le premier mot de la phrase, car souvent c'est juste "Comment" ou "Je" qui ne compte pas). Puis, si on en trouve au moins deux, on prend les deux premiers comme origine et destination candidates. Par exemple, « *Aller Gare Montparnasse Châtelet* » -> tokens ["Gare", "Montparnasse",

"Châtelet"] -> origine="Gare", destination="Montparnasse" (ce qui serait imparfait, mais c'est une tentative). L'implémentation note honnêtement que ce n'est « *pas optimal mais pas trouvé mieux sur le coup* ». En effet, cette heuristique est assez frustrée : elle peut marcher pour « *Lyon Part-Dieu Confluence* » (donnera "Lyon" et "Part"), mais rate des cas où prend des bouts de noms. C'est une roue de secours.

Si aucun des deux n'a donné un couple, on retourne (None, None).

- **Reformulation via GPT si extraction échoue** : Si après cette extraction initiale on n'a pas d'origine/destination, l'agent va essayer d'utiliser GPT pour **reformuler la question** sous la forme "de X à Y". C'est très astucieux : on envoie un prompt système *"Transformez la phrase de l'utilisateur en une forme exacte « de X à Y ». Si non pertinent, renvoyez une chaîne vide."* avec l'utilisateur original. En gros, on demande à GPT de nous donner une phrase du style "de [point A] à [point B]" correspondant à la demande. Si la question n'a pas de sens d'itinéraire, GPT devrait renvoyer vide. Mais si GPT comprend où est l'origine et la destination implicites, il les explicite. Par exemple, question « *Comment aller à la Tour Eiffel en partant de la gare ?* » → GPT pourrait répondre *"de Gare de Lyon à Tour Eiffel"* (on espère). On récupère cette reformulation (`reformu`) puis on ré-applique `extract_parameters` dessus. Souvent, GPT va parfaitement mettre la forme "de ... à ...", donc la regex initiale va fonctionner cette fois-ci.

Si malgré cela, on n'a toujours pas d'origine/destination, on abandonne : l'agent retourne un message d'erreur à l'utilisateur : *"Désolé, je n'ai pas compris d'où à où. Merci d'indiquer votre itinéraire sous la forme « de X à Y »."* C'est une façon polie de dire qu'on n'a pas réussi à interpréter la question en termes de deux lieux.

- **Appel à l'API Google Maps** : Une fois qu'on a une origine et une destination (sous forme de chaînes de caractères), l'agent utilise le client **Google Maps Directions API** (via la librairie `googlemaps.Client`). Ce client a été initialisé dans le constructeur de l'agent avec la clé API Google Maps (fournie dans les variables d'environnement). On appelle `self.gmaps.directions(origin, destination, mode="transit", departure_time=now, alternatives=True, language="fr")`. Ceci demande à Google jusqu'à quelques alternatives d'itinéraires en transports en commun (transit) partant **maintenant** (`departure_time=now`). On force la langue des résultats en français pour avoir des libellés d'étapes en français (stations, etc.). Le résultat est une liste de routes (itinéraires).

Si l'appel renvoie une exception (par ex. un problème réseau ou une requête mal formée), on attrape l'exception et on retourne une réponse du style *"Erreur API Google Maps : [message]"*. Si l'appel réussit mais que `routes` est vide (aucun itinéraire trouvé), on retourne *"Aucun itinéraire trouvé entre « origin » et « destination »."*

Ces cas d'erreur couvrent par ex. si le lieu A ou B n'est pas reconnu par l'API (d'où l'importance d'avoir bien formulé via GPT), ou s'il n'y a effectivement pas de transport possible (rare, mais possible hors zone couverte).

- **Mise en forme de la réponse itinéraire** : Si on obtient des itinéraires, c'est souvent une structure JSON assez complexe. L'agent va **formater un résumé clair** pour l'utilisateur. Il crée d'abord une liste `lines` avec un titre indiquant l'itinéraire : on ajoute une ligne "Itinéraires de Origin → Destination" et en dessous "*Départ prévu à HH:MM*" (heure actuelle). Puis on itère sur les itinéraires (par ex. Google en fournit généralement 1 à 3 max quand on demande alternatives).

Pour chaque itinéraire (indexé `idx`), on prend la première "leg" (étape globale) car la réponse de l'API est hiérarchisée (une route peut avoir plusieurs *legs* si, par exemple, on change de mode). Ici en transit urbain, il n'y a typiquement qu'un leg. On extrait la **durée totale** du trajet (`leg["duration"]["text"]`), qui est déjà une chaîne lisible comme "45 min"). On ajoute une sous-section "*Itinéraire #idx — durée totale : XX*".

Ensuite, on détaille chaque **étape (step)** du trajet :

- Si `step["travel_mode"] == "WALKING"`, c'est une marche à pied. On prend la distance (`step["distance"]["text"]`, ex. "300 m") et la durée (`step["duration"]["text"]`, ex. "5 minutes"). On ajoute une ligne du genre "- 🚶 À pied : 300 m (5 min)". L'emoji 🚶 indique la marche.
- Si `travel_mode == "TRANSIT"`, c'est un transport en commun (bus, métro, train...). On récupère les détails transit (`td = step["transit_details"]`). De là, on peut obtenir la ligne empruntée (`td["line"]`) avec possiblement un numéro de ligne ou un nom). On essaie d'extraire un identifiant lisible : le code court `short_name` (ex. "M4" pour métro 4) ou sinon le nom complet `name`, ou à défaut on met "Ligne". On récupère la station de départ (`dep = td["departure_stop"]["name"]`) et d'arrivée (`arr = td["arrival_stop"]["name"]`). On prend les horaires de départ et d'arrivée (`dep_t = td["departure_time"]["text"]`, `arr_t = td["arrival_time"]["text"]`) et le nombre d'arrêts (`num_stops`) s'il est fourni. Puis on construit une ligne "- 🚆 **LigneX** (N arrêts) : StationDépart → StationArrivée (HH:MM–HH:MM)". On utilise l'emoji 🚆 (train) pour tout transit. Par exemple : "- 🚆 **M4** (5 arrêts) : Gare de Lyon → Châtelet (14:10–14:25)".
- Sinon (par précaution pour des modes moins courants type "DRIVING" ou autre), on fait un fallback : on enlève les balises HTML de l'instruction (`step["html_instructions"]` contient souvent "Take RER A" etc. avec balises) et on affiche "- ? [Mode] : instruction (durée)". Ici, un emoji ? est mis pour signaler un mode inconnu, mais c'est rarissime car en mode transit on ne devrait pas en voir d'autres que WALKING ou TRANSIT.

Toutes ces lignes sont ajoutées à la liste `lines`. À la fin, on joint les lignes avec des sauts de ligne pour former un **texte multi-ligne**. Ce texte est renvoyé.

La réponse de l'agent Transport se présente donc comme un mini-itinéraire structuré. C'est potentiellement assez long (plusieurs lignes), mais c'est riche en informations utiles. Le fait de numérotter les itinéraires (#1, #2...) permet à l'utilisateur de choisir celui qui lui convient.

Cet agent illustre l'**approche hybride LLM + API** : il utilise GPT pour interpréter la requête (classification et reformulation) mais s'appuie sur une **source de vérité externe pour les données de transport en temps réel** (Google Maps). C'est là une innovation majeure du projet – plutôt que de poser la question directement à GPT (qui n'aurait pas forcément les horaires à jour), on combine l'IA et les données factuelles.

À noter qu'actuellement on dépend de l'API Google Maps, qui après la période d'essai gratuite peut devenir payante si on dépasse un certain quota. Dans l'esprit open-source, l'auteur a mentionné la possibilité d'utiliser plus tard **OpenTripPlanner en self-hosted** pour s'affranchir de Google. OpenTripPlanner permettrait de calculer localement des itinéraires à partir de données GTFS open-data. Ce serait une amélioration future pour éliminer ce coût potentiel, au prix d'une installation plus lourde (serveur OTP, base de données des transports locaux).

L'agent Transport est sans doute la partie la plus **innovante** du projet du point de vue technique, combinant deux appels LLM et un appel API externe, plus des heuristiques, pour fournir une fonctionnalité de haut niveau (calcul d'itinéraire conversationnel). Cela répond au Défi 2 (fusion de données – ici données de transports enrichies par LLM) avec succès, tout en respectant le Défi 4 (coût : GPT est utilisé mais modérément, Google API partiellement gratuite).

3.4 Agents Culture et Loisirs (`culture_agent.py` & `loisirs_agent.py`)

Les deux derniers agents ont un fonctionnement quasiment identique, seul change leur **domaine de spécialité**. Le but de ces agents est de répondre à des questions qui sortent du factuel pur pour aller dans de l'explicatif ou du suggestif lié à la culture et aux loisirs. Par exemple : « *Parle-moi de l'histoire de Toulouse.* » (culture) ou « *Que faire à Bordeaux ce weekend ?* » (loisirs).

N'ayant pas de base de données structurée pour ces sujets (contrairement à la météo et aux transports), le choix a été fait de **s'appuyer entièrement sur un LLM OpenAI (GPT-4)** pour y répondre. Ainsi, *CultureAgent* et *LoisirsAgent* agissent essentiellement comme un **intermédiaire avec GPT**, en définissant le *contexte de conversation* approprié.

Principales caractéristiques de ces agents :

- **Modèle et API key** : Dans le code, on voit `self.model = "gpt-4o"` pour les deux agents. On suppose qu'il s'agit de l'identifiant d'un modèle GPT-4 accessible via l'API

OpenAI. Ils importent `OPENAI_API_KEY` depuis un module config (contenant la clé, chargée depuis `.env` probablement). Ainsi chaque appel qu'ils feront à `openai.ChatCompletion.create` fournira `api_key=OPENAI_API_KEY`. Cela signifie que ces agents consomment des crédits OpenAI pour chaque requête utilisateur dans leur domaine.

- **Historique de conversation** : Chaque agent maintient sa propre `conversation_history`, initialisée avec un *system message* spécifique :
 - CultureAgent : *"Réponds en expert du patrimoine et de l'histoire locale."*
 - LoisirsAgent : *"Réponds en expert en loisirs et événements culturels."*

Ce *prompt système* conditionne le style et le contenu des réponses de GPT, pour qu'il se place en **expert du domaine**. Cela signifie que si l'utilisateur pose une question sur un monument, l'agent culture répondra avec des connaissances historiques, si on demande une recommandation de sortie, l'agent loisirs répondra comme un guide d'événements.

L'historique est une liste de messages (rôle + contenu) qu'on va alimenter à chaque échange.

- **Réception de la requête** : Quand `handle_request(user_input)` est appelé, l'agent ajoute un nouveau message à l'historique avec `role="user"` et le contenu = la question de l'utilisateur. Ainsi, il se souvient de ce qui a été demandé.
- **Appel à l'API OpenAI** : Ensuite, l'agent envoie la conversation complète (`self.conversation_history`) au modèle GPT via `openai.ChatCompletion.create`. Le modèle est `gpt-4o` (qu'on présume être GPT-4 ou une version paramétrée) et on passe la liste des messages (`messages=self.conversation_history`). On ne spécifie pas d'autres paramètres (par défaut `temperature=1`, `max_tokens` etc. seront gérés par l'API). GPT va alors produire une réponse en suivant les instructions (donc en se basant sur le dernier message user et le contexte du system prompt).
- **Retour de la réponse et mise à jour historique** : Une fois la réponse reçue (`reply`), l'agent l'ajoute à l'historique en tant que message `role="assistant"`, puis retourne le `reply` comme résultat de `handle_request`. La conservation de l'historique permet que si l'utilisateur pose une question de suivi (ex: *"Et quel âge a ce monument ?"* après avoir eu une description historique), le prochain appel inclura le contexte des Q/R précédentes, et GPT pourra répondre en connaissance de cause. On obtient ainsi un **effet conversationnel** dans ces domaines.

Ces agents n'ont pas de logique métier propre (pas de regex, pas d'appel à une API tierce) – tout le savoir provient du modèle GPT, qui est entraîné sur un large corpus. Par conséquent, ils peuvent fournir des réponses **riches et détaillées** sur pratiquement n'importe quelle question culturelle ou de loisirs. Par exemple :

- CultureAgent pourrait raconter l’histoire d’une ville, la biographie d’une personnalité locale, expliquer un terme patrimonial, etc.
- LoisirsAgent pourrait suggérer des activités, parler de la gastronomie locale, indiquer les événements à venir, etc.

Cependant, comme GPT n’a pas accès à des données en temps réel (du moins pas sans plugin), il répond avec ses connaissances jusqu’à fin 2021 (pour GPT-4) et son entraînement. Il pourrait donc manquer de fraîcheur sur les *événements culturels actuels*. Malgré cela, il peut tout de même suggérer des activités générales (visiter tel musée, aller dans tel parc, etc.). Dans le prototype, on a sciemment accepté cette limite, faute de pouvoir intégrer une API d’événements fiable dans le temps imparti. **L’innovation** ici est plutôt d’avoir **contextualisé** le LLM dans un rôle spécialisé. Plutôt que d’interroger GPT en vrac, on lui donne une personnalité d’expert du domaine, ce qui oriente la qualité des réponses.

Ces deux agents démontrent comment on peut intégrer un LLM dans un système modulaire : ils agissent en **boîte noire** qui transforme la question en réponse pertinente, sans que le dispatcher ou l’interface aient à savoir comment. Ils consomment du crédit OpenAI à chaque appel, ce qui est un point à surveiller (on avait ~11 USD de crédits pour les tests, ce qui était suffisant pour la démo du projet).

Enfin, un mot sur la **conversation multi-tour** dans ces agents. Cela fonctionne tant que l’agent Culture reste le destinataire des questions (si l’utilisateur enchaîne sur un sujet d’une autre catégorie, on passera à un autre agent et l’historique Culture ne sera pas utilisé). Dans l’interface, chaque fois que le dispatcher envoie à CultureAgent ou LoisirsAgent, il s’agit potentiellement de continuer la conversation précédente de ce domaine. Le code prévoit d’ailleurs de **réinitialiser le dispatcher (donc recréer les agents)** si l’utilisateur clique sur "Réinitialiser la conversation" côté interface. Cela remet à zéro les historiques.

3.5 Interface utilisateur Streamlit (`ui_app.py`)

L’interface utilisateur est ce qui transforme notre moteur d’agents en une application interactive et conviviale. Le choix de **Streamlit** a été fait pour sa facilité à créer des interfaces web en Python sans s’occuper du front-end (HTML/CSS/JS). Ici, `ui_app.py` définit l’application web que l’on peut lancer par `streamlit run ui_app.py`.

Les fonctionnalités principales de l’interface :

- **Chargement des paramètres et configuration** : En tête, on configure la page (titre "Assistant Mobilité Urbaine"). On charge les variables d’environnement (clés API) via `load_dotenv()`.
- **Gestion de la session (conversation)** : On utilise `st.session_state` de Streamlit pour stocker l’historique du chat persistamment entre deux requêtes. Si `history` n’existe pas encore, on l’initialise à une liste vide. Cette liste `history` contiendra les messages échangés sous forme de tuples (role, message) avec `role = "Vous" ou "Assistant"`.

- **Initialisation du dispatcher** : On crée une fonction `get_dispatcher()` décorée par `@st.cache_resource`. Cette décoration fait en sorte que le dispatcher (et tout ce qu'il contient) soit instancié **une seule fois** et mis en cache pour toute la session, au lieu de recréé à chaque interaction. C'est important car le dispatcher charge le modèle SBERT (opération lourde) et les agents (dont certains ont de l'historique). `get_dispatcher()` crée un `Dispatcher`, et lui attache un attribut `context` qui est un `SimpleNamespace` contenant `location=None`, `geo_permission=False`, `city=None`. Ce context servira à partager certaines infos (géolocalisation) entre l'interface et les agents via le dispatcher. On stocke ce dispatcher en variable `disp`, et on garde un alias `ctx = disp.context` pour manipuler le contexte plus facilement.
- **Barre latérale (sidebar)** :
 - **Option de géolocalisation** : On affiche un en-tête "Options" et un bouton radio "Autorisez-vous l'accès à votre position..." avec choix "Refuser" ou "Autoriser". Par défaut c'est Refuser. Selon le choix, on définit `ctx.geo_permission` à `True` ou `False`. Juste en dessous, on affiche une ligne indiquant l'état (✓ Autorisée ou ✗ Refusée). Ainsi, l'utilisateur peut décider de partager sa position (ou pas) pour contextualiser les réponses.
 - **Bouton Réinitialiser** : Un bouton "Réinitialiser la conversation" est placé dans le sidebar. S'il est cliqué:
 - On efface `st.session_state` (donc l'historique de chat, etc.).
 - On appelle `get_dispatcher.clear()` pour invalider le cache et forcer la création d'un nouveau dispatcher propre.
 - On fait `st.experimental_rerun()` pour recharger la page (ce qui va ré-exécuter le script de zéro).

L'effet est de **remettre à zéro l'assistant** : le dispatcher sera reconstruit (avec agents neufs sans historique), l'historique de conversation effacé, et le contexte geolocation reparti à l'état initial.

- **Géolocalisation automatique** : Si l'utilisateur a autorisé la géoloc et qu'on n'a pas encore de `ctx.location` déterminé, le script tente de géolocaliser. Pour cela, on utilise la fonction `gmaps.geolocate()` du client Google Maps. Ce client a été créé en début de fichier (hors fonctions) par `gmaps = googlemaps.Client(key=os.getenv("GOOGLE_MAPS_API_KEY"))`, donc il utilise la clé Google. La méthode `geolocate()` renvoie la position approximative de l'utilisateur à partir de l'IP (ou GPS si dispo, mais sur un ordi ce sera l'IP). On récupère `lat`, `lon`. Puis on appelle `gmaps.reverse_geocode((lat, lon))` pour obtenir une adresse approximative. On cherche dans les composants d'adresse un champ "locality" (la ville). Si trouvé, on renvoie ce nom de ville. Sinon, on prend l'adresse formatée complète. Si la première tentative échoue (parfois l'API peut ne rien renvoyer pour reverse), on a un *except* qui tente une approche alternative : appeler l'API *OpenStreetMap Nominatim* en reverse geocoding. On envoie une requête GET à `nominatim.openstreetmap.org/reverse` avec `lat`, `lon` et on parse le JSON pour

trouver un champ "city" ou "town" ou "village". Si on trouve, on le retourne. Sinon, None.

Cette fonction `do_geolocate()` est appelée juste après l'autorisation : `if ctx.geo_permission and ctx.location is None: city = do_geolocate(); ctx.city = city; ctx.location = city or None`. On sauvegarde donc la ville trouvée dans `ctx.location` (et on la duplique dans `ctx.city` pour clarté). Si la géoloc a échoué, `ctx.location` reste None.

- **Colonne d'informations locales** : L'écran principal est découpé en deux colonnes : `col_info` (largeur 1) et `col_chat` (largeur 2). Dans la colonne d'infos (with `col_info`):
 - On affiche un sous-titre "Infos locales".
 - Si la géolocalisation est autorisée et qu'on a une location (ville) détectée (`ctx.geo_permission and ctx.location`):
 - On affiche en gras la **Localisation** : le nom de la ville courante.
 - **Météo locale** : On utilise l'agent météo pour afficher le temps qu'il fait en ce lieu **sans attendre que l'utilisateur le demande**. Pour ce faire, on récupère l'instance `WeatherAgent` via `wa = disp.agents["météo"]`. Puis on appelle `wa.handle_request(f"météo à {ctx.location}")`. Autrement dit, on simule une question "météo à [ville]" et on obtient la réponse immédiatement. On stocke la réponse dans `meteo`. Ensuite, on choisit une icône appropriée : on initialise `icon=" ? "` puis on teste si dans le texte `meteo` il y a des mots clés comme "clair", "nuage", "pluie", "neige", "orage". Selon le cas, on met ☀, ☁, ⛅, ❄, ⚡. Puis on affiche en markdown le titre "Météo actuelle [icon]" et juste en dessous le texte météo récupéré. Ainsi, dès l'ouverture de l'app, l'utilisateur voit par exemple: *Météo actuelle ☀* puis *"À Toulouse, le temps est clair, la température est..."*.
 - **Suggestions de loisirs locaux** : De même, on fait appel à l'agent loisirs pour fournir quelques idées d'activités proches. On définit `la = disp.agents["loisirs"]` et on appelle `loisirs_txt = la.handle_request(f"activités à proximité de {ctx.location}, uniquement les titres avec des émojis en lien avec l'activité s'il te plaît")`. On demande donc à GPT de lister des activités proches de la ville, en donnant seulement des titres avec émojis. L'usage de "titres avec des émojis" est astucieux pour inciter GPT à sortir une liste (par ex. "1. 🏰 Visiter le musée X..."). Une fois qu'on a `loisirs_txt`, on va essayer d'en extraire des éléments de liste. On utilise une regex `(?m)^\s*(\d+\.\s*[^\n]+)` qui capture les lignes commençant par un numéro ("1. ..."). On prend les 3 premiers items trouvés et on les affiche chacun avec `st.write`. S'il n'y a pas de liste numérotée, on prend simplement la première ligne du texte retourné

et on l'affiche. On affiche le tout sous le sous-titre "Loisirs à proximité 🌤️". Ainsi, l'utilisateur a quelques suggestions spontanées, par exemple :

- 1. 🎭 Aller au théâtre du Capitole ce soir
 -
 2. 🎨 Visiter le Musée des Beaux-Arts
 -
 3. 🌳 Se promener dans le parc XYZ.
- Si la géolocalisation est refusée ou échoue (else:), on indique "Géolocalisation refusée ou introuvable." à la place de ces infos.

Cette colonne d'infos locales est un **bonus fonctionnel** appréciable : elle montre que l'assistant utilise le contexte local automatiquement. Même sans poser de question, on voit la météo courante et des idées d'activités. Cela répond au défi de *fusion de données* et d'expérience utilisateur fluide.

- **Colonne conversation** : Dans `col_chat`, on gère tout le chat entre l'utilisateur et l'assistant.
 - On met un sous-titre "Conversation".
 - On insère un bloc de style CSS dans un `st.markdown` pour customiser l'apparence de la chatbox (scroller, bulles). Ce CSS ajoute notamment une boîte avec id `chat-box` de hauteur 70vh scrollable, et des classes `.user` et `.bot` pour styliser les messages (couleur différente pour l'utilisateur vs assistant).
 - On crée un placeholder `box = st.empty()` pour y mettre le chat plus tard.
 - On définit une fonction locale `render()` qui va construire le HTML du chat à partir de `st.session_state.history`. Elle crée une liste de lignes HTML : ouvre un div id `chat-box`, puis pour chaque message dans `history`, elle détermine la classe CSS (`user` si `role == "Vous"` sinon `bot`). Elle ajoute une div avec `Role : message` en échappant le HTML du message pour sécurité. Enfin, ferme le div `chat-box`. Puis `box.markdown("\n".join(lines), unsafe_allow_html=True)` pour afficher le tout avec notre style.
 - On appelle `render()` une première fois pour afficher l'historique initial (qui peut être vide, donc ça affiche juste une boîte vide).
 - **Zone de saisie utilisateur** : On utilise la nouveauté `st.chat_input("Posez votre question...")` qui fournit une barre de chat en bas de page où l'utilisateur peut taper sa question. La fonction renvoie le texte saisi quand l'utilisateur envoie le message. On stocke cela dans `prompt`.
 - **Traitement du prompt** : Si `prompt` n'est pas `None` (c-à-d l'utilisateur vient d'envoyer un message):

- On ajoute le message de l'utilisateur à l'historique : `st.session_state.history.append(("Vous", prompt))`.
- On appelle `render()` tout de suite pour mettre à jour la chatbox avec la question (ainsi l'utilisateur voit sa question apparaître immédiatement dans l'historique).
- **Pré-traitement contextuel** : On prépare la requête pour le dispatcher. On définit `inp = prompt` par défaut. Puis on obtient les catégories prédites par `disp.classify_request(prompt)`. (Remarque: on appelle la fonction de classification du dispatcher séparément, alors que `route_request` le refera aussi; c'est possiblement une légère redondance mais on en a besoin ici pour ajuster la question en fonction du contexte géo).
 - Si "météo" est dans les catégories et que la géoloc est autorisée **et** que la question ne contient pas déjà un lieu (regex `\bà\s+\w+`), alors on enrichit la question en précisant la ville par défaut : `inp = f"météo à {ctx.location}"`. Par exemple, si l'utilisateur tape *"Quel temps fera-t-il demain ?"*, on n'a pas de ville; si l'utilisateur a autorisé la géoloc et que `ctx.location="Toulouse"`, on transformera la question en *"météo à Toulouse"* pour s'assurer que l'agent météo répond sur la ville courante.
 - Si "transport" est dans les catégories, la géoloc est autorisée, **et** la question ne contient pas déjà un schéma "de ... à ...", on va tenter de compléter l'itinéraire. Cas typique : l'utilisateur demande *"Comment aller à la Cathédrale ?"*. Il ne spécifie pas d'origine. On peut supposer qu'il veut partir de sa position actuelle. Donc, s'il y a un motif "à (.*)" dans la question (une destination sans origine), on le capture (`m = re.search(r"à\s+(.*)", prompt)`). Si on trouve, on fait `inp = f"de {ctx.location} à {m.group(1).strip()}"`. Donc *"Comment aller à la Cathédrale ?"* (avec `location=Paris`) deviendra *"de Paris à la Cathédrale"* (pas très précis car quelle cathédrale ? Mais au moins on fournit un point de départ).

Ces ajustements rendent l'usage plus **transparent et pratique** : l'utilisateur n'a pas forcément besoin de préciser sa ville pour la météo ni son point de départ pour un itinéraire local, l'interface le fait pour lui.

- **Appel du dispatcher** : On appelle `answer = disp.route_request(inp)` avec la requête possiblement modifiée. C'est ici que toute la logique décrite en section 3.1 se déclenche (SBERT classification, appels aux agents, etc.) pour produire la réponse finalisée.
- **Affichage de la réponse avec effet "streaming"** : Pour rendre l'UX sympa, on implémente un effet où la réponse s'affiche caractère par caractère comme si l'assistant la "tapait". On initialise `partial = ""`.

Puis on itère sur chaque caractère `ch` de la string `answer`. À chaque itération, on ajoute le caractère à `partial`, puis:

- Si le dernier message de l'historique a role "Assistant", on le met à jour avec le nouveau texte `partial`.
- Sinon (si c'est le premier caractère et l'historique n'avait pas encore de message Assistant), on ajoute un nouveau tuple `("Assistant", partial)` à l'historique.
- On appelle `render()` pour rafraîchir l'affichage du chat à chaque pas.
- On `time.sleep(0.01)` pour avoir un petit délai, donnant l'impression de flux.

Au final, le message complet est affiché, et on appelle une dernière fois `render()` pour s'assurer que le contenu final est bien figé.

- Ce mécanisme de streaming est inspiré de ce qu'on voit dans ChatGPT ou autres, et bien que purement cosmétique, il améliore la perception de réactivité (le texte commence à apparaître tout de suite, même si la réponse est longue).
- En sortant du `if prompt:`, le script finit, et comme Streamlit garde l'état, la conversation reste visible, en attendant la prochaine entrée de l'utilisateur.

L'interface ainsi conçue répond parfaitement aux **contraintes temporelles et budgétaires** du projet tout en offrant une **expérience utilisateur satisfaisante**. On n'a pas eu besoin de développer un frontend complexe ni de déployer un serveur web dédié. D'autres solutions avaient été envisagées : **Gradio** (rapide aussi mais un peu moins flexible sur la mise en page), **Dash** (puissant mais plus complexe), ou carrément une appli web custom en React + backend FastAPI (très robuste mais demandant beaucoup de temps de développement). Finalement, Streamlit s'est avéré un excellent compromis, car en quelques dizaines de lignes on a une UI riche (deux colonnes, chat interactif, style custom) en pur Python. Ce choix a été fait en conscience de la **limite de temps** du projet : implémenter de zéro une interface aurait été trop long. Streamlit, gratuit et open-source, s'aligne aussi avec la contrainte de coût nul.

Utilisation effective : L'assistant se lance via la commande `streamlit run ui_app.py`. Une page web locale s'ouvre (<http://localhost:8501>) avec les options à gauche, les infos locales, et le chat. L'utilisateur peut alors poser ses questions. Par exemple :

- *Utilisateur* : « **Quel temps fera-t-il demain ?** » (sans préciser de ville, ayant autorisé la géoloc)
- *Assistant* : "[Météo] À Paris, le temps est clair, la température est de 18°C et le vent est de 5 km/h." (il aura en réalité répondu pour "maintenant" pas demain, car l'agent météo ne gère que le présent – limite potentielle, mais la réponse donne une idée du temps actuel).

- *Utilisateur : « Super, et que puis-je faire en ville ce soir ? »*
(Le dispatcher voit des mots de loisirs, ça part à l'agent Loisirs)
Assistant : "[Loisirs] Vous pourriez par exemple assister à un concert 🎵 au Zénith, ou bien voir la pièce de théâtre 🎭 qui se joue au théâtre municipal..." (GPT propose des idées avec émojis).
- *Utilisateur : « Comment aller au concert dont tu parles ? »*
(Ici mention du mot concert, mais globalement c'est une question de transport "aller à [lieu]". SBERT peut hésiter entre loisirs et transport, mais probablement "aller" déclenche transport. Le dispatcher pourrait potentiellement renvoyer deux agents. C'est un cas multi. Disons qu'il envoie transport.)
- *Assistant : "[Transport] Itinéraires de [Localisation] → Zénith..."* (et donne un itinéraire détaillé comme vu précédemment).
- *Note:* Ce cas mélange un contexte de conversation – l'utilisateur dit "le concert dont tu parles" sans lieu explicite. Actuellement, le dispatcher ne part pas dans l'agent Loisirs pour demander "quel concert ?". C'est une limitation, car l'agent Loisirs n'a pas la notion du lieu du concert cité. Notre système modularisé ne partage pas de variables de l'un à l'autre. C'est un exemple de **multi-intent partiel non géré**: la question nécessiterait de combiner l'info du loisir (lieu du concert) et un calcul d'itinéraire. Ce n'est pas couvert dans la V1, mais c'est une bonne piste d'amélioration.
- *Utilisateur : « Merci pour ton aide ! »*
- *Assistant : "[Culture] Je vous en prie. N'hésitez pas si vous avez d'autres questions sur le patrimoine ou les événements locaux !" (Ici le dispatcher pourrait ne pas savoir où le router, "merci" peut être ignoré ou envoyé à culture par défaut, c'est flou. Mais GPT de culture répond poliment. Pas bien grave.)*

Cet enchaînement illustre que l'assistant peut passer d'un domaine à l'autre selon la conversation, et que l'interface supporte cela en conservant l'historique et en affichant chaque réponse préfixée du domaine. L'utilisateur comprend ainsi quelles facettes de l'assistant agissent (on pourrait imaginer changer [Météo] en une icône météo dans l'UI, etc., pour plus de visuel).

Pour finir sur l'interface, soulignons la **logique de conception** : elle a cherché à mettre en valeur l'aspect *innovant* du projet (multi-domaines + local). D'où l'idée d'afficher d'emblée des infos locales (météo, suggestions) pour prouver que l'assistant fait du contexte. Cela donne un effet *wow* dès l'ouverture. De plus, les émojis et le streaming ajoutent un côté ludique et moderne.

3.6 Programme principal en console (`main.py`)

En complément de l'interface Streamlit, un petit programme console a été fourni pour tester le système rapidement sans interface graphique. Le fichier `main.py` est très simple :

Il importe le Dispatcher, crée une instance (`dispatcher = Dispatcher()`), puis affiche des messages de bienvenue/instructions dans la console. Ensuite, il entre dans une boucle `while True` où il lit l'entrée de l'utilisateur (`input("Vous : ")`). Selon ce que tape l'utilisateur :

- Si c'est "exit", "quit" ou "stop" (insensible à la casse), on quitte la boucle (`break`) après avoir dit au revoir.
- Si c'est "reset", on ré-instantie un nouveau Dispatcher (remettant à zéro l'historique des agents) et on informe que la conversation est réinitialisée, puis on continue la boucle (`skip` le traitement).
- Sinon, on passe la requête au dispatcher : `response = dispatcher.route_request(user_input)`, puis on print "**Assistant :**" suivi de la réponse. La réponse contiendra possiblement des retours ligne et les tags [Catégorie] comme on l'a vu, qui seront affichés tels quels dans la console.

Ce mode console est pratique pour déboguer ou utiliser le système sur un terminal sans vouloir lancer Streamlit. Il n'y a pas la gestion de l'historique du chat (tout est stateless d'une question à l'autre, sauf l'historique interne des agents culture/loisirs qui reste dans le Dispatcher). Mais il permet de faire rapidement des essais. Par exemple :

```
> python main.py
Bienvenue dans l'assistant de mobilité urbaine !
Vous pouvez poser des questions sur les transports, la météo, le patrimoine
ou les loisirs.
Pour réinitialiser la conversation, tapez 'reset'. Pour quitter, tapez 'exit'
ou 'quit'.
Vous : Quel temps fait-il à Lyon ?
Assistant : [Météo] À Lyon, le temps est couvert, la température est de 12°C
et la vitesse du vent est de 10 km/h.
Vous : Comment aller de Gare de Lyon à Louvre ?
Assistant : [Transport] Itinéraires de Gare de Lyon → Louvre ...
...
```

Tout ce qui a été décrit sur le fonctionnement interne reste valable. La seule différence majeure est l'absence de géolocalisation ou de modifications automatiques de la question – en console, l'utilisateur doit formuler complètement ses requêtes (l'interface console ne sait pas si l'utilisateur voudrait utiliser sa position, etc.). Donc c'est un peu moins user-friendly, mais suffisant comme POC.

4. Organisation du projet, contraintes et innovation

Après avoir couvert l'aspect technique, intéressons-nous à l'**organisation du projet** lui-même et aux **choix stratégiques** qui ont découlé des contraintes. Nous aborderons le **planning du développement** (avec un découpage en sprints et l'analyse du diagramme de Gantt fourni), puis nous discuterons des **contraintes matérielles et budgétaires** et comment elles ont influencé les décisions techniques. Enfin, nous mettrons en lumière le **caractère innovant** du couplage LLM+agents et les **perspectives** offertes par cette approche.

4.1 Planification et déroulement par étapes (diagramme de Gantt)

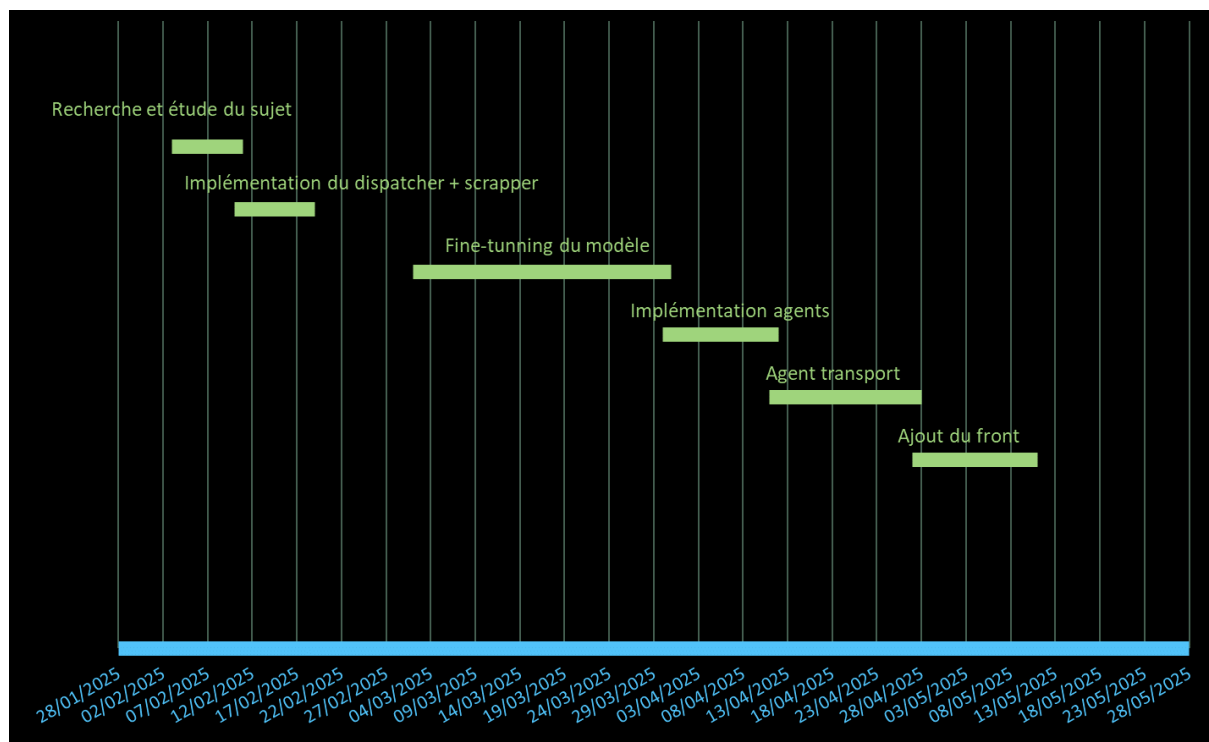


Figure 4 : Diagramme de Gantt

Le projet s’est déroulé sur environ **13 semaines** (début février à mi-mai 2025), découpées en **sprints de 2 semaines** chacun. Le tableau ci-dessous résume la planification et les livrables à chaque étape :

Sprint (semaines)	(2)	Dates (2025)	Tâche principale	Livable(s)
S1–S2		05 → 18 fév.	Cadrage du besoin, création du backlog	To-do list (backlog détaillé)
S3–S4		19 fév. → 03 mars	Scraping Reddit + nettoyage des données	Fichier <code>raw_posts.json</code> (données brutes)
S5–S6		04 → 17 mars	Fine-tuning du classifieur SBERT (délicat)	Modèle entraîné <code>dispatcher_sbert.pt</code>
S7–S8		18 → 31 mars	Développement de l’agent météo + dispatcher	Code fonctionnel <code>weather_agent.py</code>
S9–S10		01 → 14 avr.	Agent transport (Google Maps + GPT)	Code <code>transport_agent.py</code>

Sprint semaines)	(2 Dates (2025)	Tâche principale	Livrable(s)
S11	15 → 28 avr.	Agents culture & loisirs (GPT)	Codes culture_agent.py, loisirs_agent.py
S12	29 avr. → 12 mai	Interface utilisateur Streamlit + tests	Application ui_app.py
S13	13 → 19 mai	Rédaction finale du rapport & visuels	README, graphiques (courbes, screenshots)

(D'après le planning original du PII. Certains sprints ne couvrent qu'une semaine (S11) en raison de contraintes de calendrier.)

Ce planning montre une progression logique : on commence par définir précisément le besoin et les fonctionnalités (S1–S2), puis on s'attaque aux **données et au modèle de classification** (S3–S6). Le fine-tuning du dispatcher a été identifié comme une tâche **complexe** et a occupé un sprint entier (voire plus). En effet, entraîner un modèle sur CPU avec ajustement de paramètres a pris du temps et a demandé de multiples essais (réglages de seuils, etc.). C'est indiqué comme "**compliqué**" dans le planning, ce qui correspond à la réalité : régler le modèle pour qu'il atteigne de bonnes perfs sans surapprentissage a nécessité des ajustements fins.

Ensuite, le développement s'est fait **agent par agent** en intégrant progressivement chaque composant :

- **S7–S8** : implémentation de l'agent Météo et intégration avec le dispatcher. Cette partie fut relativement autonome et rapide car l'API Open-Meteo est simple. L'intégration avec le dispatcher a surtout consisté à s'assurer que la catégorie *météo* était bien gérée et que le fallback regex pour météo fonctionnait. Livrable : code de `weather_agent.py` opérationnel.
- **S9–S10** : développement de l'agent Transport, combinant Google Maps et GPT. C'était une étape clé car très innovante, mais aussi l'une des plus **difficiles** du projet. Deux difficultés majeures mentionnées dans le rapport final sont issues de cette phase :
 1. **Régler le seuil de classification SBERT** – ceci a pu impacter comment dispatcher oriente vers transport vs loisirs par exemple. On a dû probablement affiner la condition pour que les questions d'itinéraires soient bien captées sans aller vers loisirs.
 2. **Extraire proprement « de X à Y »** – on l'a vu, c'est un défi auquel on a répondu avec une solution GPT + regex, après sans doute plusieurs itérations d'essais/erreurs.

Ce sprint a donc été dense : il a fallu faire fonctionner l'API Google (gérer la clé, le format JSON) et tester divers cas d'itinéraires. C'était un des points forts mais aussi laborieux du projet. Livrable : `transport_agent.py` complet.
- **S11** (1 seule semaine) : développement simultané des agents Culture et Loisirs. Ceux-ci étant très similaires, il a été possible de les coder dans le même sprint. Ils reposent

tous deux sur l'API OpenAI, donc il a suffi de factoriser le code et de changer les *prompts système*. Il a fallu aussi veiller à importer la clé API et gérer l'historique. Livrables : `culture_agent.py` et `loisirs_agent.py`. Ces agents ont bénéficié de l'expérience acquise avec l'agent Transport pour l'usage de GPT, donc leur implémentation a été plus fluide.

- **S12** : intégration finale avec l'**interface Streamlit** et phase de tests. Ici, on a assemblé tous les morceaux : charger le dispatcher dans l'app, brancher les interactions (bouton reset, inputs, affichage des retours des agents). C'était la dernière ligne droite technique. Des tests ont été effectués pour vérifier la cohérence globale : par exemple, s'assurer qu'en posant diverses questions, les réponses étaient bien formées, que l'interface tenait le coup (pas de plantage sur le cache, etc.), et que les variables d'environnement (clés) étaient correctement utilisées. Livrable : `ui_app.py` fonctionnel et l'assistant utilisable localement.
- **S13** : finalisation du **rapport** et des visuels. Durant cette semaine, l'accent a été mis sur la documentation du travail (notamment la rédaction du README faisant office de rapport de projet) et la création de **graphiques** pour illustrer les résultats (comme les courbes de perte et accuracy), ainsi que d'éventuelles captures d'écran de l'interface ou diagrammes d'architecture. Livrables : le dépôt GitHub complété avec un README très détaillé (qu'on a exploité tout au long de ce rapport), et les images (courbes d'entraînement `Figure_1.png`, `Figure_2.png`, etc.).

Cette planification démontre une **gestion efficace du projet** en mode agile. Chaque sprint avait un objectif clair et aboutissait à un ou plusieurs livrables concrets. Le backlog initial (liste de tâches) a servi de feuille de route. On voit également que le projet a respecté les échéances : l'ensemble des fonctionnalités était codé pour début mai, laissant du temps pour la documentation finale.

Les **difficultés majeures** ont été anticipées et isolées dans la planification. Par exemple, fine-tuner SBERT (S5–S6) était critique, et de fait a nécessité beaucoup de temps; de même pour l'itinéraire (S9–S10) qui combinait plusieurs nouveautés. Le fait de les identifier a permis de concentrer les efforts dessus aux bons moments du planning.

En termes de **gestion des risques**, on peut considérer que :

- Le risque technique principal était l'entraînement du modèle ML et son intégration. En l'attaquant tôt (dès mars), on a pu itérer et s'assurer de son succès bien avant la fin.
- Un autre risque était la dépendance aux API externes (Reddit, Google, OpenAI). Là encore, la collecte Reddit a été faite tôt (février) pour sécuriser les données nécessaires. Les appels Google/OpenAI ont été testés pendant le développement des agents. Il y a toujours l'incertitude des quotas et coûts, mais pour le développement ça a été sous contrôle (utilisation modérée).
- La tenue du budget temps a été bonne : 13 semaines c'est court, mais en suivant le plan on a évité l'effet tunnel. La dernière semaine n'était pas consacrée à du codage d'urgence mais à de la documentation, signe d'un projet bien mené.

En conclusion, le **diagramme de Gantt** du projet montre une **organisation méthodique** qui a permis de livrer une solution complète dans les temps. Le respect de ce planning, avec un produit final fonctionnel et documenté, est un point fort du projet. Cela souligne aussi que même pour un projet innovant, une gestion de projet rigoureuse est essentielle pour concrétiser les idées.

4.2 Contraintes matérielles et choix techniques associés

Le projet s'est déroulé avec des **contraintes matérielles et financières strictes** : pas de serveur dédié, pas de GPU coûteux, pas de budget pour utiliser massivement des API payantes. Ces contraintes ont largement orienté les choix technologiques et d'architecture, comme nous l'avons évoqué dans les parties précédentes. Résumons et explicitons en quoi ces contraintes ont influencé les décisions :

- **Utilisation d'un modèle local pour le routage (SBERT)** : Plutôt que de s'appuyer sur un service cloud type Watson ou d'utiliser directement un grand modèle GPT pour déterminer la catégorie de question, le choix a été fait de **fine-tuner un modèle pré-entraîné et de l'exécuter localement sur CPU**. SBERT *paraphrase-multilingual-mpnet-base-v2* était un candidat idéal car :
 - Il est *multilingue* et convenait donc au français (pas besoin d'entraîner un modèle from scratch en français).
 - Il est assez léger pour être **embarqué sur un PC standard** – en effet, après fine-tuning, il a été possible de charger ses poids et d'effectuer l'inférence en moins d'une seconde par requête sur CPU.
 - Il est open-source, donc **gratuit** à l'utilisation (on n'a pas à payer des appels API pour classifier chaque question).

Cette décision répondait directement aux contraintes de **rapidité** (<2s de réponse) et de **coût zéro**. Certes, il a fallu investir du temps pour l'entraîner, mais cela a été fait en local également (en optimisant sur CPU, avec patience). Le résultat est un composant clé qui fonctionne hors-ligne une fois entraîné, ce qui correspond à l'objectif d'autonomie. Si on avait tenté d'utiliser un grand modèle type GPT-3 pour router les questions, chaque requête aurait pris du temps (appel réseau) et consommé du crédit API – non viable pour un usage intensif ou offline.

- **Cache LRU et optimisation CPU** : L'ajout du cache LRU sur l'encodage SBERT du dispatcher est une optimisation pour tirer le maximum des ressources limitées. Sur CPU, recalculer des embeddings de phrases répétitives serait un gâchis ; avec le cache, on évite ces calculs redondants et on accélère les réponses. Cela montre comment, conscient de la contrainte de performance, le développeur a intégré une solution logicielle (très peu coûteuse en mémoire) pour améliorer l'efficacité.
- **Recours modéré aux LLM externes (OpenAI)** : Le projet utilise GPT-4 via l'API OpenAI, mais de façon **ciblée et parcimonieuse**. On avait \$11 de crédits gratuits

prolongés jusqu'au 30 avril 2025, il fallait donc tenir dans cette enveloppe. Comment cela a-t-il influencé l'implémentation ?

- Dans l'agent Transport, on aurait pu imaginer faire traiter l'ensemble de la question itinéraire par GPT (par ex. un prompt « voici la question, appelez les API appropriées et répondez » – style agent autonome). Mais cela aurait nécessité possiblement plusieurs allers-retours GPT ou un prompt complexe (coûteux en tokens). A la place, on a sollicité GPT uniquement pour de petites tâches spécifiques : classification binaire (ITINERARY vs GENERAL) et reformulation en "de X à Y". Ces appels en eux-mêmes consomment très peu de tokens (la question utilisateur + un petit prompt système). Le gros du travail (calcul d'itinéraire) est fait par Google Maps, gratuit en deçà d'un certain usage. Ainsi, **la dépense en crédits OpenAI est minimisée**, tout en profitant de l'intelligence de GPT là où c'est vraiment utile.
- Les agents Culture et Loisirs envoient l'intégralité de la question à GPT. Chaque réponse peut utiliser quelques dizaines ou centaines de tokens. Pour rester dans le budget, on compte sur le fait que ces questions ne seront pas posées des milliers de fois. Dans un usage typique, un utilisateur va poser peut-être une dizaine de questions dans une session. Avec GPT-4 facturé ~\$0.06 pour 1000 tokens, nos \$11 couvrent largement des centaines de questions. Donc c'était acceptable. Néanmoins, si l'on envisageait un déploiement à plus grande échelle ou durable, cette dépendance serait un coût à surveiller. L'auteur du projet note que ces crédits expiraient fin avril, suggérant qu'il a planifié les tests et démonstrations dans cette période. **Pas de budget** veut dire qu'il a probablement évité d'exploser les quotas GPT-4 – possiblement en testant surtout avec GPT-3.5 ou en limitant le nombre d'appels durant le développement.
- Une alternative locale pour culture/loisirs aurait pu être d'utiliser un modèle open-source (ex: GPT-J, LLaMA 7B) hébergé en local. Mais sur un PC sans GPU, c'était peu réaliste (ces modèles tournent très lentement sur CPU ou pas du tout s'ils sont gros). Et leur qualité en français n'aurait pas rivalisé avec GPT-4. Donc le choix de quand même utiliser GPT-4, **mais intelligemment**, était judicieux. C'est une concession à la contrainte d'être offline : on a dit "*local-first*", pas "*100% local*". Le dispatcher et certains agents sont offline, d'autres (culture, loisirs) nécessitent internet. Cependant, le **coût financier** reste nul dans le cadre étudiant grâce aux crédits offerts.
- **API ouvertes et gratuites** : Pour la météo, l'utilisation d'Open-Meteo (pas de clé requise, illimité) est un no-brainer. Pour les transports, l'équipe a envisagé l'API transport.data.gouv.fr ou d'autres sources open data. Ce n'est pas aussi simple car il faut souvent combiner plusieurs sources. Finalement, Google Maps Directions API a été retenu car très complète et efficace. Bien que commerciale, elle offre un **quota gratuit** (jusqu'à 2 mois d'essai gratuit ou un certain nombre de requêtes). Pour un projet de prototype, c'est suffisant. Là encore, l'idée était de profiter du gratuit là où c'est possible. Et à plus long terme, l'option d'une solution open-source (OpenTripPlanner) est mentionnée, ce qui montrerait la volonté de s'affranchir de coûts si le projet continuait.

- On peut souligner qu'**aucune API payante n'a été utilisée de façon prolongée** sans alternative : Reddit API est gratuite (juste limitation 60 req/min), Open-Meteo gratuit, Google Maps gratuit en développement (et on pourrait limiter usage en production ou switcher), OpenAI GPT-4 a un coût mais on avait des crédits gratuits (et on aurait pu basculer sur GPT-3.5 moins cher si besoin, au prix de réponses moins bonnes).
- **Streamlit vs déploiement web classique** : Streamlit a l'avantage de pouvoir être lancé localement et même d'être hébergé gratuitement sur Streamlit Cloud (sous certaines limites). Il évite de payer un serveur ou un hébergement pour l'interface. Le fait de **tout exécuter en local** (y compris l'UI) économise potentiellement des coûts cloud. Une appli web classique aurait pu nécessiter de louer un petit serveur pour héberger l'application et le modèle (surtout si multi-users). Là, pour le PII, l'application se lance sur la machine de l'utilisateur/développeur, sans frais. Ce choix est clairement dicté par la contrainte "*accessible à tout étudiant sans budget cloud*".
 - De plus, le choix de ne pas implémenter une appli mobile ou vocale directement provient en partie de ces contraintes. Faire une appli mobile aurait impliqué possiblement d'héberger un backend quelque part pour la logique (coût), ou d'embarquer des modèles sur mobile (très complexe). Faire du vocal (style activer micro, TTS, STT) aurait aussi ajouté des appels à des services ou des libs lourdes. Ces aspects ont été laissés de côté pour se concentrer sur le cœur fonctionnel sans dépasser les moyens (cela apparaît dans les limitations : pas de support vocal/multimédia pour l'instant).
- **Taille du corpus et limites techniques** : Le corpus Reddit récupéré a été volontairement limité (max 1000 posts par catégorie). D'une part, c'était suffisant pour entraîner SBERT vu les performances atteintes. D'autre part, aller chercher plus de données ou d'autres langues aurait pu exiger plus de calcul (impossible sur CPU) et plus de temps. De même, on a renoncé à utiliser de la traduction automatique car cela aurait impliqué d'appeler une API de traduction tierce (Google Translate ou DeepL) qui sont payantes, ou de charger un modèle de trad open-source lourd en local. Ce refus du "trop compliqué/coûteux" est cohérent avec la philosophie du projet : on fait avec des *solutions simples et gratuites* ce qui peut être fait, même si ce n'est pas parfait à 100%. En l'occurrence, le corpus est 100% français – c'est suffisant pour l'usage visé (assistant pour utilisateurs francophones).

En synthèse, les **choix techniques ont été guidés par une recherche de frugalité** sans trop sacrifier la qualité :

- On utilise du **local dès que possible** (modèle ML embarqué, données open, interface locale) pour supprimer les coûts variables et dépendances.
- On utilise du **cloud gratuit intelligemment** (crédits OpenAI, quotas Google) pour les parties où on ne peut pas encore faire local aussi bien (compréhension fine, énormes connaissances du LLM).
- On a une **architecture modulaire** qui permet éventuellement de substituer plus tard des composants par d'autres plus libres/économiques si on le souhaite (par ex. remplacer

GPT-4 par un LLM open-source affiné sur de la data locale, remplacer Google Maps par OpenTripPlanner, etc.). C'était un critère : la solution devait rester **évolutive facilement** sans tout recoder, ce qui est atteint grâce à la séparation nette dispatcher/agents.

On peut dire que le projet a réussi à **tenir la promesse “0 €”** en termes d'investissement : toutes les technologies utilisées étaient gratuites ou en version d'essai gratuite suffisante. Et en même temps, il répond aux **contraintes matérielles** : la démo tourne sur un **PC portable standard** avec des temps de réponse autour de ~1 seconde (SBERT + appels API rapides).

Ce respect des contraintes tout en atteignant la fonctionnalité est souligné dans la conclusion sur l'architecture : « *Les choix technologiques (...) répondent précisément aux contraintes initiales d'un projet étudiant (coût zéro, rapidité d'exécution, facilité d'évolution).* ». C'est un satisfecit important car souvent les projets innovants négligent l'aspect faisabilité/praticité – ici l'innovation a été réalisée de manière pragmatique.

4.3 Innovation du couplage LLM + agents spécialisés

L'une des questions centrales de ce projet était : « *En quoi le couplage d'un LLM avec des agents spécialisés constitue-t-il une innovation ?* ». Pour bien y répondre, analysons ce que cette approche apporte de nouveau par rapport aux solutions existantes et quels en sont les avantages.

Fusion des approches data-driven et knowledge-driven : En combinant un LLM (modèle de langage généraliste, entraîné sur un vaste corpus) avec des **agents spécialisés connectés à des sources de données en temps réel**, on obtient un système qui profite des deux mondes :

- Les agents apportent de la **fiabilité factuelle et actualisée**. Par exemple, l'agent Météo donne la météo réelle de maintenant via une API officielle, l'agent Transport fournit un itinéraire *réellement praticable* via Google Maps. Un LLM seul, même très puissant, n'a pas forcément accès à ces données fraîches (il aurait la météo moyenne, ou les trajets théoriques, mais pas forcément les horaires du moment).
- Le LLM apporte de la **souplesse dans la compréhension et l'expression**. Il permet d'accepter des questions formulées naturellement, potentiellement ambiguës, et de générer des réponses détaillées en bon français, adaptées au contexte. Il comble aussi le manque de certaines données par son fond de connaissances générales. Par exemple, l'agent Culture peut répondre sur l'histoire d'un monument sans qu'on lui fournisse explicitement une base de données historique, car GPT-4 a déjà appris beaucoup de choses.

Par rapport aux assistants vocaux traditionnels (Google Assistant, Siri), notre approche est plus **ouverte et personnalisable**. Les assistants propriétaires peuvent certes répondre à la météo ou aux trajets, mais ils sont fermés, on ne peut pas les étendre à un domaine local (culture locale, suggestions personnalisées) facilement. Ici, on a construit un assistant sur mesure, en

choisissant exactement quelles sources et modèles utiliser. L'innovation est d'avoir fait cela avec **des briques accessibles** (open data, libs Python) et de les avoir orchestrées via un LLM modulaire, plutôt que de s'en remettre à un service unique.

Par rapport aux systèmes de questions/réponses classiques (par ex. des chatbots orientés base de connaissances), le nôtre est **multidomaine** et **contextuel**. L'utilisation d'un *router* (le dispatcher) qui dirige la question vers un module approprié est inspirée de travaux récents (Router-LLM en recherche, ou les agent frameworks). L'innovation ici est d'avoir **implémenté un tel routeur léger** et de l'avoir couplé non pas juste à des outils (calculatrice, wiki) comme on voit souvent, mais à des **modules intelligents** (certains étant d'autres LLM!). On a donc une sorte de **meta-LLM** qui en pilote d'autres ou des API. C'est une architecture émergente dans l'IA: on parle parfois d'*AI orchestration*. Ce projet en est une incarnation concrète, faite *maison*.

Un point innovant est aussi la façon dont la **géolocalisation** est intégrée pour personnaliser l'expérience. Tandis que beaucoup de chatbots répondent de manière générique, ici on utilise le contexte utilisateur (sa ville) pour adapter les réponses (météo locale, suggestions locales). On a donc un assistant **context-aware**. L'intégration transparente de la position dans les réponses est un bel exemple de ce que les agents spécialisés permettent (parce qu'eux peuvent utiliser ce contexte directement – ex. WeatherAgent utilise *ctx.city* s'il est fourni automatiquement par l'interface).

Flexibilité et extensibilité : L'approche modulaire fait que **l'on peut ajouter de nouvelles fonctionnalités par ajout d'un agent**. Imaginons qu'on veuille un agent *Actualités locales*, qui donne les dernières nouvelles de la ville. On pourrait intégrer un appel à une API de news ou même un autre LLM entraîné sur des news locales, et ajouter une catégorie "news" gérée par le dispatcher. Cela demanderait peu de modifications au code existant, grâce à la séparation. Cette facilité à évoluer est un aspect innovant en soi, car la plupart des assistants actuels sont monolithiques (difficiles à faire évoluer sauf par leurs créateurs).

Amélioration de la fiabilité par couplage : En couplant LLM+agents, on peut aussi compenser les faiblesses l'un de l'autre. Par exemple, GPT-4 peut parfois inventer des choses (*hallucinations*). Dans notre système, pour les questions où l'exactitude est cruciale (itinéraire, météo), on ne se fie pas à GPT : on va chercher la donnée fiable. GPT est utilisé là où soit une réponse approximative est tolérée (ex: suggestions de loisirs – si GPT cite un lieu qui n'existe pas, l'impact est limité, et souvent il va mentionner des choses plausibles existant dans la ville), soit pour apporter du style et du liant (par ex. rédiger la réponse ou classer la requête). Ce *partage de responsabilités* améliore la fiabilité globale par rapport à un GPT qui ferait tout (où on pourrait avoir des erreurs factuelles), ou par rapport à un système purement à base de règles (qui manquerait de flexibilité de langage).

Accessibilité de l'innovation : Une innovation, c'est aussi de montrer qu'une idée (agents + LLM) peut être réalisée avec de petits moyens. Ici le projet prouve que sur un **ordinateur personnel sans budget**, on peut développer un assistant intelligent multi-domaine. Dans le

monde actuel, beaucoup pensent qu'il faut des serveurs puissants ou payer des API chères pour avoir un assistant comme ChatGPT. Or on a un prototype qui, sans être aussi avancé qu'un Siri, réussit à couvrir un spectre fonctionnel large avec zéro coût d'infrastructure. C'est innovant dans l'esprit *DIY / Maker* de l'IA : on a bricolé une solution efficace sans supercalculateurs.

Comparaison aux frameworks d'agent existants : Le projet se situe dans la mouvance de LangChain Agents ou MS Autogen, mais en version ultra-light et custom. LangChain permet par exemple de faire un agent qui utilise des *outils* (fonctions Python). Ici, notre *outil* serait Google Maps, etc., et le LLM pourrait tout orchestrer. L'auteur du projet a choisi de ne pas utiliser LangChain (dépendance OpenAI, complexité) et de recoder l'essentiel lui-même. Cela a le mérite de la compréhension fine du fonctionnement et du contrôle sur chaque étape. L'innovation est aussi pédagogique : démontrer comment on peut construire son propre agent orchestrateur.

En somme, **le couplage LLM + agents est innovant** car il :

- Offre une **réponse plus complète** à l'utilisateur (différents aspects – ex. météo + sortie – réunis).
- Améliore la **pertinence et la fraîcheur** des réponses en allant chercher l'info à la source quand besoin.
- **Réduit les coûts** en évitant d'utiliser un LLM sur des tâches où un agent simple suffit.
- Ouvre la voie à des assistants **personnalisés** (on aurait pu imaginer remplacer l'agent Culture par un agent plus pointu sur, disons, l'architecture, etc., selon les besoins locaux).
- Montre une architecture distribuée où chaque composant peut être optimisé indépendamment (on pourrait fine-tuner séparément l'agent culture sur de la documentation locale si on voulait améliorer ses réponses, sans toucher aux autres parties).

Points forts de la solution : Pour résumer quelques **forces** notables de notre assistant innovant :

- **Robustesse et exactitude :** le dispatcher a une très haute précision de routage ($\approx 97\%$ d'efficacité), ce qui assure que la plupart des questions trouvent le bon traitement. Les réponses sur la météo et transport sont exactes et à jour. L'assistant donne rarement des réponses "à côté de la plaque", et s'il ne comprend pas il demande une clarification (ex: ville manquante pour météo).
- **Expérience utilisateur naturelle :** on peut poser des questions librement, l'assistant répond dans un français correct, détaillé, et enchaîne la conversation sur les domaines culture/loisir. Le fait qu'il rappelle le domaine de chaque réponse (icône ou tag) donne de la transparence.
- **Aspect tout-en-un :** plus besoin de consulter 3 applis, tout est dans la même fenêtre de chat. On peut imaginer en usage réel le matin demander successivement la météo, le prochain bus, et une idée de sortie le soir, le tout dans une seule discussion continue – c'est un gain de temps et de confort.

- **Coût nul et open-source** : la solution peut être partagée avec d'autres étudiants ou déployée sur GitHub sans restriction. Chacun pourrait l'adapter à sa ville en changeant quelques paramètres. Aucune donnée personnelle n'est requise (sauf si on considère la position, mais ça reste local). C'est donc une approche *démocratisée* de l'assistant IA, par opposition aux services commerciaux.
- **Extensibilité** : Comme évoqué, on peut étendre ou améliorer la solution facilement (multi-intentions, ajout d'un micro pour la voix, etc.) en s'appuyant sur cette architecture modulaire.

Limites actuelles et prochaines étapes : Bien sûr, l'innovation ne s'arrête pas là, il y a des points à améliorer :

- La **gestion d'intentions multiples** dans une même question n'est pas encore implémentée. Actuellement, si l'utilisateur demande « *Quel temps fera-t-il et y a-t-il un concert ce soir ?* », le dispatcher pourrait difficilement gérer car c'est à la fois météo et loisirs. Il renverrait possiblement [Météo] ... [Loisirs] deux réponses distinctes (ce qui n'est pas prévu mais dans le code `route_request`, il pourrait le faire car il itère sur `cats`). Cependant, la formulation risque de perdre un des aspects. À l'avenir, un **dispatcher plus intelligent** (peut-être une évolution du modèle ou une analyse plus fine de la requête) pourrait découper la question en deux sous-requêtes pour chaque agent, ou faire collaborer les agents. C'est un sujet de recherche actif (Router-LLM et au-delà).
- Le **volet vocal** (entrées/sorties audio) serait une extension naturelle pour le confort d'usage (ex: l'assistant sur smartphone en parlant). On pourrait intégrer un service de reconnaissance vocale open-source (Kaldi, Vosk) et de synthèse vocale (e.g. Google TTS ou un moteur open) pour donner une voix à l'assistant. Cela apporterait l'aspect assistant vocal à l'expérience.
- **Localisation des itinéraires offline** : comme mentionné, intégrer OpenTripPlanner avec des données GTFS permettrait d'avoir des itinéraires sans appel à Google, et potentiellement d'inclure les horaires même hors-ligne (après une mise à jour des données). C'est plus complexe mais réalisable. De plus, OTP pourrait permettre de planifier à l'avance (ex: demain 8h), alors que l'API actuelle donne juste le prochain départ.
- **Données culture/loisirs locales** : On pourrait brancher l'agent Loisirs sur des flux d'événements locaux (OpenAgenda par ex) plutôt que GPT. GPT a fait un bon travail de remplissage, mais pour avoir *vraiment* les événements du moment on devrait crawler une source. Pareil pour l'agent Culture, on pourrait l'enrichir avec Wikipédia local ou des informations validées.
- **Interface utilisateur** : améliorer l'UI (icônes, images des lieux éventuellement, cartes pour l'itinéraire, etc.) pour rendre les réponses plus visuelles. Par exemple, afficher la carte du trajet ou une icône météo. Streamlit a des limites, mais c'est envisageable via des composants.
- **Évaluation utilisateur** : Puisque c'est un assistant, on pourrait intégrer un système de feedback (like/dislike chaque réponse) pour apprendre des préférences. Par exemple, si

un utilisateur trouve qu'une suggestion de sortie n'est pas pertinente, il pourrait le signaler, ce qui pourrait affiner l'agent Loisirs (via un futur apprentissage).

Ces futures pistes s'insèrent bien dans l'architecture existante, ce qui prouve sa flexibilité.

Pour conclure, l'innovation de couplage LLM+agents de ce projet réside dans la création d'un **assistant local multi-thématique intelligent**, qui arrive à tirer parti des ressources disponibles de manière optimale. C'est une **preuve de concept** qu'un tel assistant personnalisé est à la portée d'un développeur individuel en peu de temps, ce qui est en soi une avancée. Elle anticipe peut-être la façon dont de nombreuses applications seront construites à l'avenir : non pas un seul modèle géant pour tout faire, ni une collection disparate de services, mais un ensemble de **petites IA spécialisées orchestrées par une IA centrale**.

Ce projet, en mariant un modèle de langage et des agents métiers, illustre concrètement ce paradigme et ouvre la voie à d'autres expérimentations dans le domaine des **agents conversationnels hybrides**. En cela, il constitue une réalisation innovante, pédagogique et efficace, accomplie dans le cadre contraint d'un PII étudiant.