Le génie pour l'industrie

RAPPORT TECHNIQUE
PRÉSENTÉ À L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
DANS LE CADRE DU COURS LOG795

PROJET DE FIN D'ÉTUDES
PFE020 CONCEPTION ET IMPLÉMENTATION D'UNE APPLICATION EN MICROSERVICES POUR LA PRÉDICTION DES PRIX DES ACTIONS EN BOURSE

Khalil Anis ZABAT
ZABK86080008
Samuel FORTIN
FORS15109805
Basile PARADIS
PARB30059704
David Vermette Nadeau
VERD18019800
Marc-Olivier Fillion
FILM09089803

DÉPARTEMENT DE GÉNIE LOGICIEL ET DES TI

Professeur-superviseur

MANEL ABDELLATIF

NOUAL MOHA

MONTRÉAL, 22 AVRIL 2025
HIVER 2025

REMERCIEMENTS

Nous tenons à remercier Hakim Ghlissi pour sa collaboration tout au long du projet. Son encadrement et ses conseils ont été très utiles à l'équipe pour mener le projet à terme.

RÉSUMÉ

À l'ère de l'intelligence artificielle, l'intégration rapide de modèles d'IA dans des systèmes logiciels existants soulève des défis majeurs de scalabilité, de modularité et de maintenabilité. Les architectures monolithiques, bien que simples à mettre en œuvre, deviennent rapidement un goulot d'étranglement lorsque l'on vise à servir un grand nombre d'utilisateurs tout en garantissant la sécurité et la résilience des services. Dans ce contexte, notre projet de fin d'études illustre comment migrer une application monolithique de prédiction de prix boursiers vers une architecture en microservices, tout en respectant les bonnes pratiques de conception logicielle.

Nous avons d'abord conçu un prototype monolithique intégrant trois composants clés d'IA : un modèle de prédiction de séries temporelles basé sur LSTM, un analyseur de sentiments FinBERT appliqué à des flux de nouvelles financières, et un chatbot à connaissance de domaine pour l'interaction utilisateur. Forts de cette première version, nous avons procédé à la décomposition en microservices : chaque module (authentification, gestion de portefeuille, collecte et indexation des données de marché, services IA, etc.) est désormais déployé indépendamment, communiquant via RabbitMQ et sécurisé par des certificats TLS et JWT.

Cette migration a mis en évidence l'importance d'une infrastructure distribuée robuste : nous avons développé des bibliothèques partagées pour l'authentification et la messagerie, assurant cohérence et testabilité dans un environnement Kubernetes. Les microservices IA incluent un service de prédiction, un service d'entraînement, un service de collecte de données et un service d'analyse de sentiments. Cette fragmentation permet une mise à jour et une montée en charge ciblées, ainsi qu'une meilleure isolation des responsabilités.

Au terme de ce projet, nous démontrons qu'une transition maîtrisée vers les microservices renforce la flexibilité, la performance et la qualité du développement d'applications enrichies en intelligence artificielle.

Table des matières

INTRODUCTION

Lorsque vient le temps d'amorcer le développement d'une application, mettre tous les fichiers dans un seul répertoire est en quelque sorte la chose la plus intuitive puisqu'ils sont créés pour servir un seul et même projet. L'architecture système dite « monolithique » est la structure la plus connue et la plus simple. C'est en partie pour cette raison qu'un grand nombre de projets en font usage. Cependant, à mesure qu'un projet grossit et accroit en complexité, une architecture monolithique devient rapidement contraignante. Cette architecture apporte un couplage élevé et un service plus gros et plus lourd à maintenir. Pour remédier à ce problème, employer une architecture composée de « microservices » dès le départ peut être avantageux. Cette architecture permet de réduire fortement le couplage en minimisant la responsabilité des services pour plus facilement les maintenir. Cependant la migration d'une architecture monolithique en micro-service n'est pas une tâche simple. Il faut bien comprendre le domaine d'affaires dans lequel l'application œuvre afin d'effectuer correctement la séparation de ses modules. Ajoutons à cela un module d'intelligence artificiel et toutes ses fonctionnalelités, la tâche à réaliser n'est pas trop évidente. L'objectif au final est d'utiliser des stratégies de décomposition adaptées aux systèmes d'IA afin de mettre l'accent sur la modularité des services, la communication entre eux et les performances globales.

Ce rapport présente notre démarche de migration d'une application monolithique intégrant un module d'intelligence artificielle vers une architecture en micro-services. Nous y détaillons les défis spécifiques rencontrés lors de la fragmentation d'un système complexe comportant des modèles d'IA, ainsi que les solutions mises en œuvre pour assurer une transition harmonieuse. L'accent est mis sur les stratégies d'identification des domaines fonctionnels, la gestion des interfaces entre les services, et l'optimisation des performances dans un environnement distribué.

CHAPITRE 1 SURVOL DU PROJET

1.1 Motivation

Le projet nous a été présenté comme étant deux volets de développement effectué en tandem. Le premier étant le développement d'une preuve de concept, c'est-à-dire la plateforme de prédiction de valeur boursière qui ne sera pas utilisée par de vrais utilisateurs. L'objectif est de fournir une suite de fonctionnalités avancées qui permettront de simuler un système qui possède de l'âge et qui apporte une certaine complexité. Le deuxième se porte sur la migration du système monolithique vers un système distribué en microservices dans laquelle nous devons porter attention à la manière de décomposer la composante IA du système. Le but étant de découvrir et analyser les manières de décomposer efficacement les systèmes basés sur l'IA. La complexité de la tâche survient lors de l'attribution des frontières à chaque micro-service afin de garantir une bonne cohésion et garder le couplage à un minimum.

1.2 Objectifs

Nos tâches sont donc initialement le développement d'un backend qui inclut un système d'authentification, les fonctionnalités nécessaires pour la gestion du portefeuille ainsi que l'affichage de données boursières. Nous devons également développer un module IA qui communiquera avec le backend afin de lui fournir les données sur l'analyse de sentiment des nouvelles financières ainsi que les prédictions effectuées sur les stocks. Nous devons aussi développer un frontend qui gèrera le reste des fonctionnalités et qui fournit une interface simple et plaisante à regarder. Une fois toutes les fonctionnalités intégrées, nous devons ensuite effectuer la décomposition des composants backend en services distincts, tant au niveau de notre backend classique que pour notre module IA.

CHAPITRE 2 ÉTAT DE L'ART

2.1 Terminologies et concepts

2.1.1 Architecture monolithique

2.1.1.1 Contexte et compromis

Il est très difficile de donner une réponse fermée et sérieuse à l'état de l'art en architecture monolithique. La réalité c'est que plusieurs auteurs, entreprises et personnes d'influences poussent différentes idées comme étant la bonne solution. L'architecture est avant tout un exercice de détermination de sur quoi nous pouvons faire des compromis. Afin d'éviter d'approcher la question trop vaguement, spécifions certains paramètres. Une application déployée à grande échelle, dans un contexte d'entreprise qui souhaite l'utiliser pour plusieurs années. Nous arrivons habituellement à un regroupement d'architectures semi-standard selon le domaine simplement parce que nous pouvons rarement compromettre la qualité ou la modificabilité d'un système. La qualité peut être définie en boite noire comme un système qui nous donne des résultats précis de manière prévisible. De prime abord, il s'agit ici d'une pierre angulaire ayant des répercussions vastes. Examinons quel outil est essentiel pour qu'un système soit *jugé* prévisible, soit une pyramide de test adéquate. Concentrons-nous sur les tests unitaires et d'intégration.

2.1.1.2 Tests unitaires : isoler l'expertise métier

L'idée sémantique derrière un test *unitaire* est qu'il teste une unité de code. Code ici est souvent le mauvais terme, un peu trop vaste. Il doit tester une unité de connaissance. Un système qui prétend avoir la compétence requise pour évaluer la valeur d'un portfolio est une expertise qui transcende le code. Cette expertise est immuable temporellement comparé à du code qui est sujet à des changements fréquents. Quelle entreprise serait prête à investir dans une série de tests précis pour du code sujet à des changements fréquents? Il faut donc un code immuable qui communique clairement son expertise pour être bien testable en unité. Une comparaison qui est faite en littérature est que dans l'architecture du bâtiment, un plan d'hôpital ressemble à un hôpital. Qu'un plan d'appartement ressemble à des logements. Un logiciel de gestion de stock devrait ressembler à un système de gestion de stock. C'est pourquoi nous isolons une couche entière de technologie (dans la mesure du possible) pour exprimer notre expertise dans un domaine particulier.

2.1.1.3 Tests d'intégration et CQRS

Maintenant que notre système est un expert domaine en isolation, nous devons l'intégrer afin qu'il puisse servir ses fonctions. Introduisons les tests d'*intégration*. Jusqu'à maintenant, le style architectural est plutôt vague, sauf pour sa capacité à isoler une partie de son code pour le rendre plus expressif. Les options sont ouvertes ici et il n'y a pas une seule bonne réponse. Cependant, une de ces réponses valides est l'utilisation de CQRS (Command/Query Responsibility Segregation). L'idée est qu'il faut forcer un plus grand niveau de garanties sur les opérations qui impactent l'état du système que celles qui en représentent simplement l'état. Lors d'une commande (modification potentielle d'état), il est important de vérifier avec l'expert (le domaine) que le système demeure dans un état consistant. Il faut donc prendre en mémoire toute l'information nécessaire à l'expert (souvent appelé agrégat) afin qu'ils puissent publier des évènements de domaine, invalidé, validé ou même retourné une valeur à la suite d'une requête et sauvegarder le tout comme une unité de travail à la base de données (une opération atomique).

2.1.1.4 Lecture et persistance : pragmatisme technique

Ces opérations atomiques qui modifient le système sont dispendieuses. C'est pourquoi nous acceptons souvent d'être moins stricts sur les lectures. Un peu comme une base de données tend à avoir un niveau d'isolation plus faible pour ses lectures que pour ses écritures. À cet effet, au lieu d'utiliser des répertoires pour garantir l'atomicité de transaction, en lecture nous injectons parfois les détails de la technologie de persistance des données dans la couche applicative. Certains s'indignent de cette pratique argumentant que c'est un détail qui devrait être une abstraction. Cependant, dans les faits, il est excessivement rare qu'un service utilisant SQL modifie sont implémentation pour une technologie sans schéma. Il est aussi peu réaliste de penser que ce changement n'ait pas de répercussion sur le reste du système.

2.1.1.5 Validation des choix et performance

Tout de même, si une architecture est implémentée sans schéma, c'est simplement parce que les méthodes de répertoire sont tellement spécifiques à chaque requête, qu'il faudra tout de même les réécrire. Ceci éloigne la solution du problème et a pour conséquence de découpler ce qui est couplé par définition, retrouvé des données de la technologie de persistance, menant à des opérations inefficaces qui n'utilise pas correctement la technologie choisie.

2.1.1.6 Tests bout en bout et environnement réel

Ultimement, ce sont les tests d'intégration qui valident que le tout fonctionne, peu importe les choix faits en cours de route. Les applications modernes utilisent de vraies dépendances, par exemple une base donnée SQL déployée sur Docker) pour la durée du test afin de vérifier que l'entièreté des modules de l'application fonctionnent de manière cohérente. C'est en utilisant une architecture où les tests sont traités de façon primordiale, avec un mandat de qualité qu'une grande partie des décisions architecturales sont prise pour nous.

2.1.2 Architecture micro-services

L'architecture en micro-services est en l'évolution de l'architecture monolithique dans un environnement qui le pousse à être plus modifiable et maintenable à grande échelle. À mesure que le système monolithique grandit, le travail requis pour comprendre la structure et y apporter des changements prend de l'ampleur. Des problèmes de couplage majeur peuvent alors monter à la surface, ce qui peut coûter cher en dette technique et en argent. Une architecture en micro-services peut aider à remédier au problème. Chaque micro-service est conçu afin de répondre à une fonctionnalité propre au domaine. Dans notre cas le portefeuille, les stocks ainsi les nouvelles sont tous des fonctionnalités du domaine. Chacune d'entre elles possède une instance de base de données propre à elle-même, ce qui vient réduire le couplage massivement. Cela permet ainsi à des équipes potentiel de se spécialiser dans un seul micro-

service et en devenir l'expert. Chaque micro-service étant dans un environnement propre à lui-même fait en sorte que chacun peut adopter les technologies qui sont les mieux adapté pour fournir à leur besoin et également déployer de nouvelles versions du micro-service sans déranger les autres. Ils sont également utiles lorsque l'on doit entreprendre la migration d'un projet *legacy*. Les fonctionnalités du domaine sont alors identifiées et une à une, sont isolé dans un micro-service propre à elle. Les micro-services ne sont par contre pas la solution pour tous les maux. Cette architecture est beaucoup plus complexe à déployer. Les communications interservices sont également plus difficiles à mettre en place, elles doivent être asynchrone et doivent être robuste aux pannes. Dans certains cas, construire un système en micro-services peut s'avérer être une solution trop coûteuse pour un projet de plus petite taille. Dans ce genre de cas, une architecture monolithique pourrait très bien servir les besoins des développeurs.

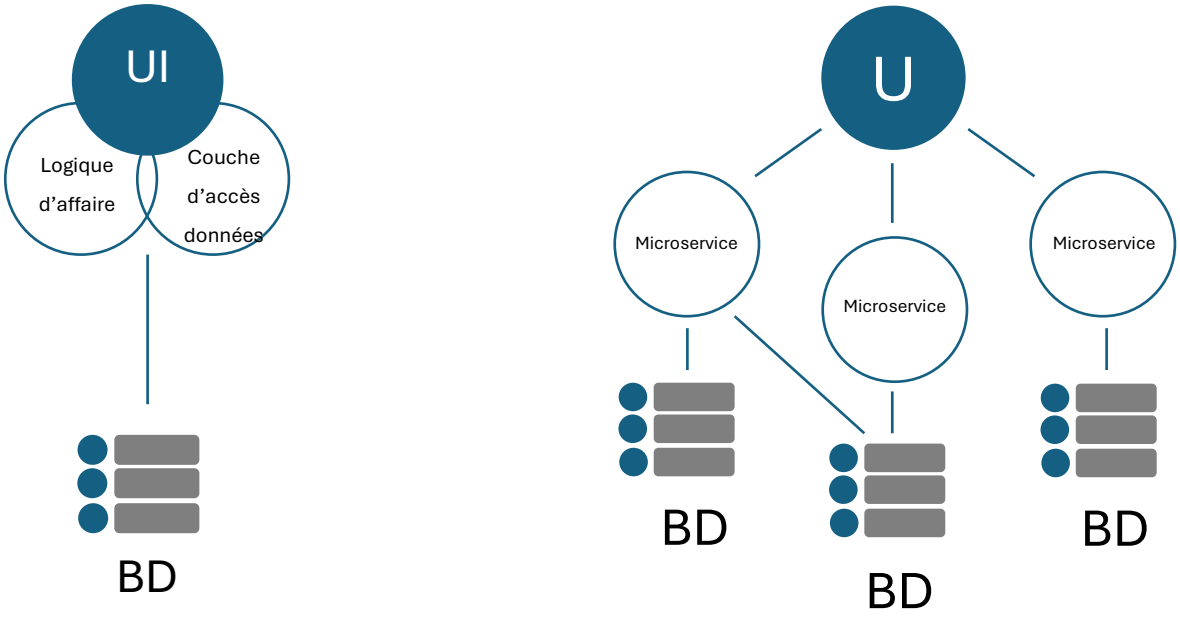

Figure 1.0 Schéma représentant une architecture monolithique (gauche) et en micro-services (droite)

2.1.3 Modèles d’IA

Afin de refléter la diversité des choix de conception propres aux applications basées sur l’intelligence artificielle, nous avons élaboré un système reposant sur l’intégration de plusieurs types de modèles. Certains de ces modèles sont préentraînés, tandis que d'autres ont été développés et entraînés entièrement depuis zéro. Cette approche nous a permis d’explorer en profondeur les implications techniques et conceptuelles liées à l’intégration de chaque type de modèle, tant au niveau de leur comportement que de leur performance. Elle nous a également offert une meilleure compréhension des enjeux associés à la conception modulaire, notamment dans la perspective d’une architecture orientée microservices.

2.1.3.1 Prédiction de prix

2.1.3.1.1 LSTM et Prophet

L'architecture du service de prédiction boursière utilise deux modèles complémentaires : un LSTM (Long Short-Term Memory) et un modèle Prophet, chacun ayant des caractéristiques distinctes.

Le modèle LSTM est une architecture de réseau de neurones récurrent spécialisée dans l'analyse de séries temporelles. Il est composé de deux couches LSTM empilées (100 puis 50 unités) avec des couches de dropout (0.2) pour prévenir le surapprentissage, suivies de couches denses (50 et 25 unités avec ReLU) et une couche de sortie à une unité. Le modèle utilise l'optimiseur Adam et la fonction de perte MSE (Mean Squared Error). Les données d'entrée sont prétraitées avec un MinMaxScaler et organisées en séquences temporelles d'une longueur définie. Le modèle est particulièrement efficace pour capturer les dépendances à court et long terme dans les données boursières.

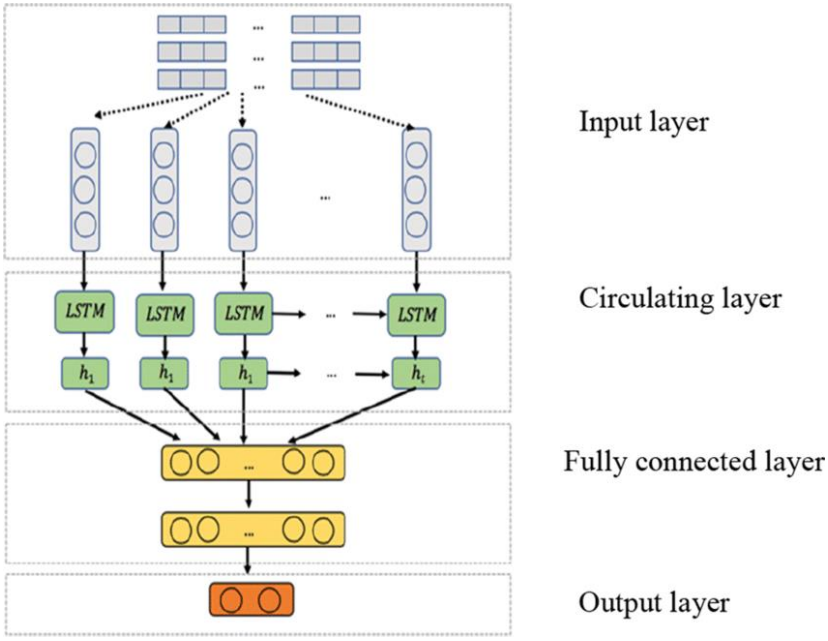

Figure 2.0 Schéma des différentes couches par lesquels les données traverse

Le modèle Prophet, développé par Facebook, est un modèle additif qui décompose la série temporelle en trois composantes principales : la tendance, la saisonnalité et les jours fériés. Il utilise une approche bayésienne pour modéliser la tendance non linéaire et gère automatiquement les effets saisonniers. Prophet est particulièrement robuste aux données manquantes et aux points aberrants, et il peut incorporer des régresseurs supplémentaires pour améliorer ses prédictions.

Dans le service de prédiction, ces deux modèles travaillent en parallèle. Le LSTM excelle dans la capture des patterns complexes et des dépendances temporelles, tandis que Prophet est plus performant pour la modélisation des tendances à long terme et des effets saisonniers. Les prédictions des deux modèles sont combinées pour fournir une vue plus complète des mouvements futurs des prix, avec des scores de confiance calculés pour chaque prédiction. Cette architecture hybride permet de bénéficier des forces de chaque approche tout en atténuant leurs faiblesses respectives.

2.1.3.2 Analyse de sentiment

2.1.3.2.1 finBERT

Pour construire un portrait fidèle du sentiment du marché, nous nous sommes appuyés sur **FinBERT**, une version de BERT pré-entraînée spécifiquement pour le domaine financier. Un modèle BERT (Bidirectional Encoder Representations from Transformers) est conçu pour traiter et comprendre du texte via des techniques de NLP (Natural Language Processing). FinBERT, quant à lui, a été affiné sur des corpus financiers de référence afin d'attribuer à chaque extrait de texte une distribution de probabilités sur trois classes : négatif, neutre et positif.

2.1.3.2.1.1 Collecte des données

Dans un premier temps, nous avons implémenté des scrapers destinés à extraire des articles de sources variées (The Wall Street Journal, Reuters, Bloomberg, etc.). Cependant, ces robots ont rapidement été soumis à des restrictions de quotas et à des blocages IP. Faute de temps pour développer des contournements robustes, nous avons pivoté vers l'API de Yahoo Finance, qui nous offrait un accès limité, mais fiable aux dernières actualités financières. Afin de ne pas dépasser les quotas imposés par l'API, nous avons restreint notre collecte à un maximum de dix articles par titre boursier pour chaque requête.

2.1.3.2.1.2 Prétraitement et pondération

Les articles financiers ne présentent pas nécessairement un texte homogène : l'avis sur une action peut se situer dans l'introduction, le corps ou la conclusion, avec des intensités variables. Pour en tenir compte, nous avons découpé chaque article en sections pondérées (introduction, développement, conclusion) avant traitement.

2.1.3.2.1.3 Segmentation et agrégation

FinBERT ne peut prendre en entrée que 512 tokens à la fois. Nous avons donc fragmenté chaque section en segments de 512 tokens, calculé un score de sentiment pour chacun, puis recombiné ces scores en une note agrégée par article selon leur poids respectif. Enfin, pour offrir une vision globale du sentiment sur un titre donné, nous calculons la moyenne pondérée des scores de tous les articles collectés, reflétant ainsi la diversité des opinions exprimées.

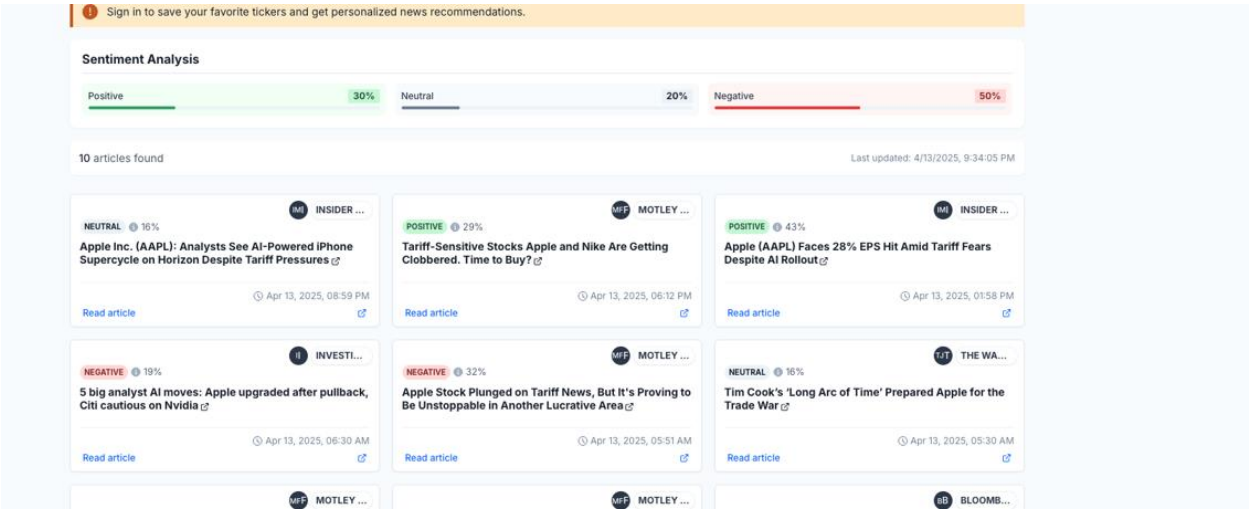

Figure 3.0 Affichage des nouvelles de la journée avec l'analyse de sentiment par action

2.1.3.3 Chatbot interactif

2.1.3.3.1 Contexte et champ d'action

Notre agent conversationnel s'inscrit dans un rôle de conseiller financier : dès la première requête, il se restreint à l'univers boursier et économique, refusant poliment toute question hors de ce périmètre. Cette limitation garantit à l'utilisateur des réponses précises et cohérentes, exclusivement alignées sur les thématiques d'investissement, de produits financiers et d'analyse de marché.

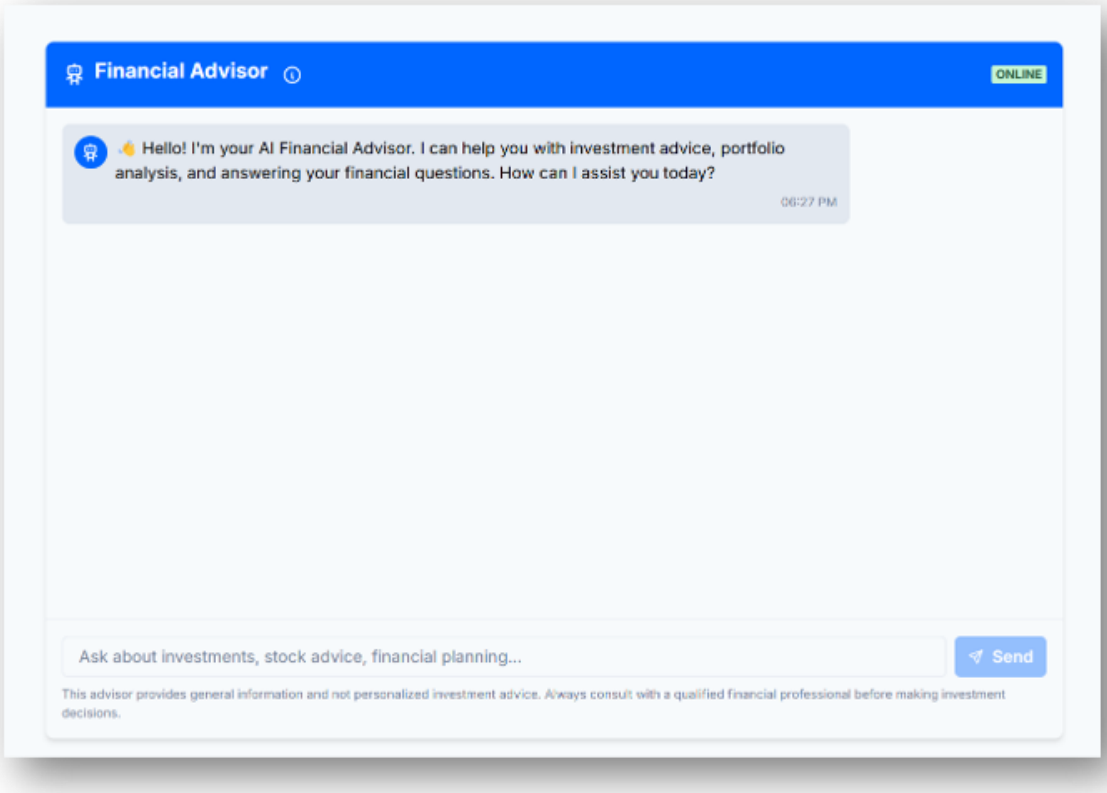

Figure 4.0 Vue du chatbot interactif

2.1.3.3.2 Mémorisation et personnalisation

Pour que chaque échange soit plus pertinent que le précédent, nous avons intégré à l'aide de LangChain un module de mémoire contextuelle. Au fil de la conversation, l'IA enregistre les préférences sectorielles, le profil de risque et le style d'investissement de l'utilisateur. À la clôture de la session, elle génère un résumé synthétique et met à jour un profil stocké en base de données, que le chatbot restaure à la reconnexion suivante. Ainsi, l'agent conserve en permanence la continuité du dialogue et affine sa compréhension des besoins individuels.

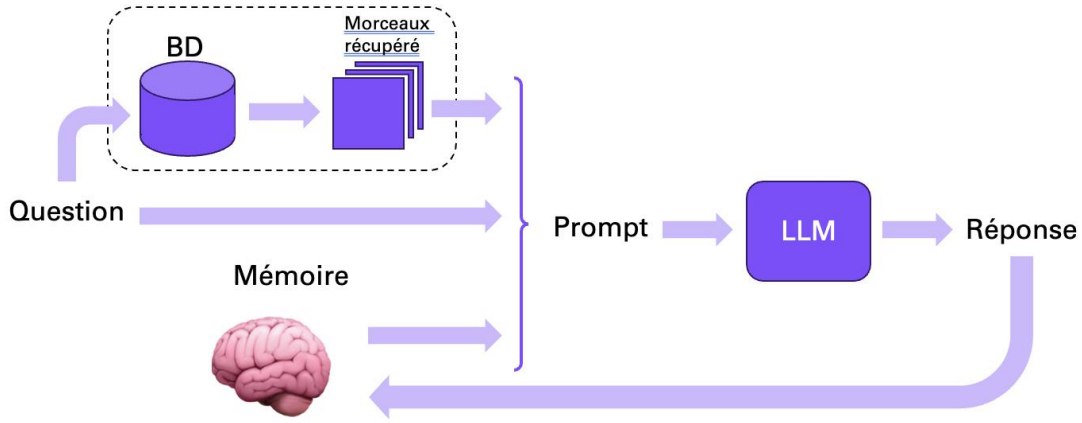

Figure 5.0 Fonctionnement du module de mémoire LangChain

2.1.3.3.3 Exploitation des données personnelles

Pour affiner ses recommandations, le chatbot dispose d'un accès en lecture aux données de portefeuille (titres détenus, répartition sectorielle) et à l'historique des ordres de l'utilisateur. Cette vision détaillée lui permet d'ajuster automatiquement ses conseils d'achats, ventes ou rééquilibrages en fonction des performances passées et de la composition actuelle du capital.

2.1.3.3.4 Intégration des microservices IA

Enfin, lorsqu'une requête exige une analyse de marché ou une projection de cours, l'agent appelle en temps réel les microservices dédiés : d'un côté, le service de prédiction de prix, basé sur LSTM/Prophet ; de l'autre, le service d'analyse de sentiment FinBERT. En combinant ces résultats avec le contexte personnel, le chatbot propose des conseils fondés à la fois sur le profil de l'utilisateur et sur des données financières actualisées, offrant ainsi une expérience fluide, sûre et parfaitement adaptée à chaque investisseur.

CHAPITRE 3 MÉTHODOLOGIE

3.1 Gestion du travail

Dans le cadre de notre projet, nous avons adopté une approche structurée pour la gestion du travail, en tirant parti d'outils collaboratifs modernes et en répartissant les tâches selon les compétences de chaque membre de l'équipe. Discord a été notre principal canal de communication, nous permettant d'organiser les tâches et de discuter en temps réel. Cette plateforme nous a offert la flexibilité nécessaire pour créer des canaux dédiés à des sujets spécifiques, facilitant ainsi la concentration sur des aspects particuliers du projet sans encombrer les discussions générales.

Parallèlement, nous avons utilisé GitHub pour le contrôle de version et la gestion des tâches. Le système de Kanban intégré à GitHub nous a permis de visualiser l'état d'avancement des différentes tâches, de suivre les progrès et d'identifier rapidement les éventuels goulots d'étranglement. Cette transparence a été cruciale pour maintenir une bonne coordination au sein de l'équipe.

La répartition des tâches a été pensée de manière à maximiser l'expertise de chaque membre. Basile et Anis se sont concentrés sur la conception et le développement des composants d'intelligence artificielle, apportant leur savoir-faire dans ce domaine complexe. Samuel et David ont pris en charge la logique du backend, assurant la robustesse et la performance de l'infrastructure. Enfin, Anis et Marc-Olivier ont collaboré sur la conception et le développement du frontend, veillant à offrir une expérience utilisateur fluide et intuitive.

Cette organisation nous a permis de travailler de manière efficace et harmonieuse, chaque membre pouvant se concentrer sur ses points forts tout en bénéficiant du soutien et de l'expertise des autres. L'utilisation combinée de Discord et de GitHub a renforcé notre capacité à collaborer à distance, à maintenir une communication ouverte et à gérer le projet de manière

agile. En résumé, notre méthodologie de gestion du travail a été centrée sur la communication, la collaboration et l'exploitation optimale des compétences individuelles, soutenue par des outils adaptés à nos besoins.

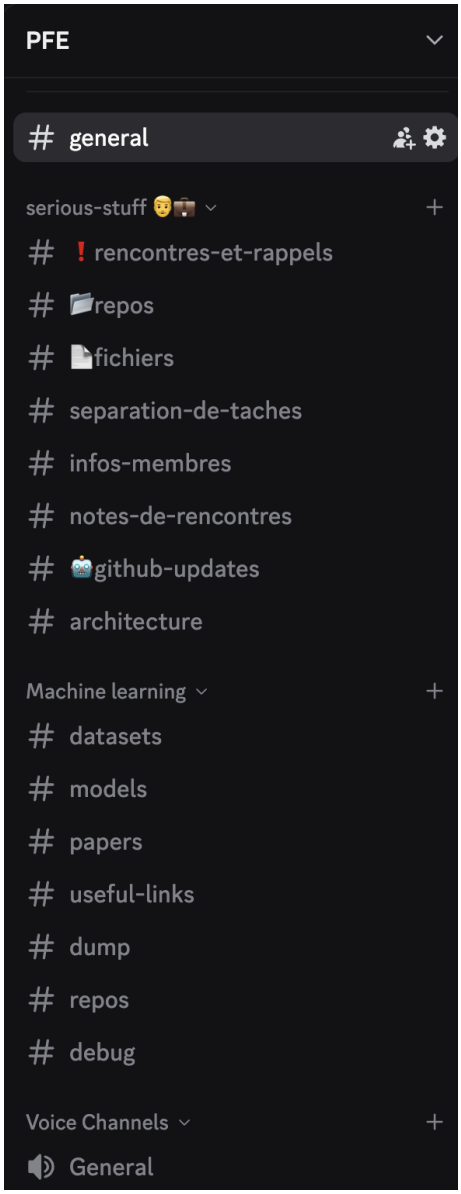

Figure 6.0 Capture d’écran du serveur Discord

3.2 Outils

Pour le développement de notre projet, nous avons mobilisé une série d'outils sélectionnés en fonction des besoins techniques identifiés au fil de l'implémentation. L'entraînement de nos modèles d'intelligence artificielle a été réalisé sur Google Colab, en exploitant la puissance des TPUs pour accélérer le processus d'apprentissage. Afin d'assurer la compatibilité et la robustesse de nos solutions, nous avons travaillé à la fois sur des processeurs ARM et sur des architectures x86, ce qui nous a permis de valider le comportement de nos applications sur des environnements hétérogènes.

Le développement backend a été réalisé avec Microsoft Visual Studio, un environnement intégré particulièrement adapté au développement en C#. Pour la conteneurisation et la gestion de nos environnements d'exécution, nous avons utilisé Docker Desktop, facilitant ainsi la portabilité et la reproductibilité de nos applications. Le prototypage rapide ainsi que le développement de premières versions fonctionnelles ont été accélérés grâce à l'utilisation de Cursor, un outil optimisé pour l'itération rapide et la génération de code.

Concernant le frontend, plusieurs technologies ont été envisagées, parmi lesquelles React, Vue, Angular ainsi que des solutions plus spécialisées comme Ghost. Après analyse, nous avons retenu React, considérant sa simplicité d'apprentissage, sa rapidité d'implémentation et sa grande flexibilité. React nous offre un système de gestion d'état efficace, utilisant des observables, ce qui favorise la réactivité de l'interface et la modularité du code. L'intégration du HTML et du TypeScript au sein de composants unifiés simplifie la maintenance et améliore la clarté du projet. De plus, l'écosystème riche de React permet d'incorporer facilement des bibliothèques externes, notamment pour la création de graphiques interactifs, ce qui s'est révélé particulièrement utile pour l'affichage dynamique des données boursières.

CHAPITRE 4 IMPLÉMENTATION

4.1 Frontend

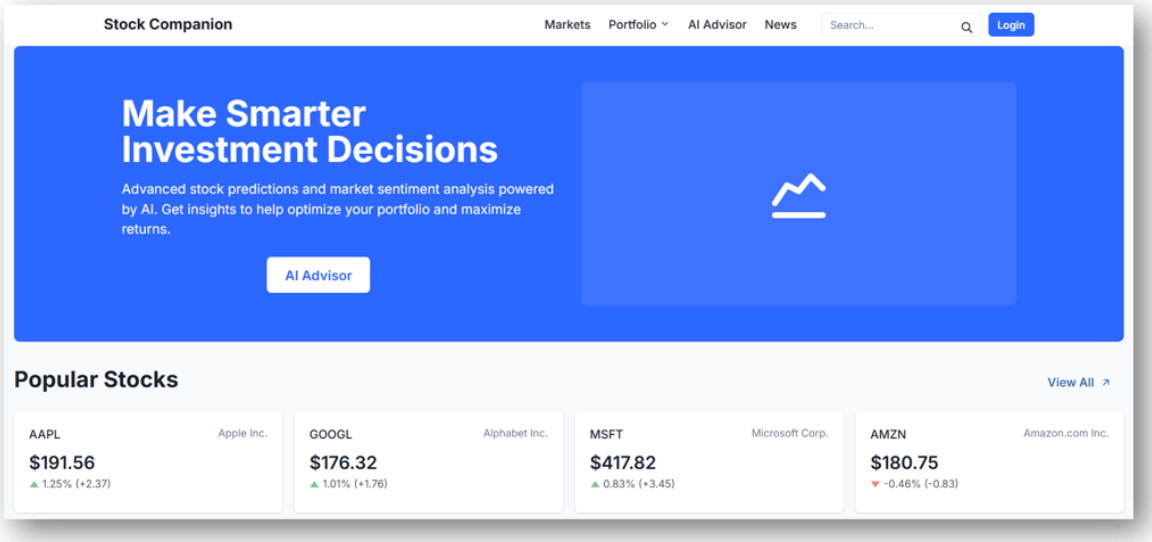

Figure 7.0 Page d'accueil de l'application

4.1.1 Architecture déclarative de React

L'architecture déclarative de React repose sur la notion de composants qui décrivent l'interface utilisateur à partir d'un état unique. Grâce aux hooks comme useState et useEffect il devient possible de synchroniser automatiquement l'affichage avec les données locales et partagées, qu'elles proviennent du Context API ou d'une librairie de gestion d'état. Chaque composant réagit aux changements de son état sans que l'on ait à manipuler manuellement le DOM, ce qui simplifie la maintenance et encourage la réutilisation de fragments d'interface isolés.

4.1.2 Environnement de développement Vite

Vite tire parti des modules ES natifs des navigateurs pour lancer instantanément un serveur de développement sans bundling préalable. Les fichiers Typescript et Javascript sont compilés à

la volée par l'intermédiaire d'une chaîne de plugins rapide, et le rafraîchissement à chaud applique uniquement les changements de modules concernés. Cette approche réduit drastiquement le temps d'attente au démarrage et lors des itérations quotidiennes, tout en offrant un système de build optimisé pour la production.

4.1.3 Typage strict avec Typescript

Typescript équipe chaque couche du frontend d'un typage statique qui élimine un grand nombre d'erreurs avant même l'exécution. Les interfaces et les types génériques définissent avec précision les schémas JSON renvoyés par les microservices et guident l'autocomplétion de l'éditeur. L'intégration de règles de linting spécifiques garantit un code cohérent et documenté automatiquement, ce qui facilite la collaboration au sein de l'équipe et la montée en compétences des nouveaux contributeurs.

4.1.4 Routage client avec React Router

React Router orchestre la navigation interne sans jamais recharger la page en mappant chaque URL à un arbre de composants. Les routes imbriquées permettent de composer simplement un layout parent et ses sous-vues tandis que l'importation asynchrone de modules via React.lazy et Suspense fragmente le code afin d'accélérer le premier rendu. Cette configuration assure une expérience fluide même sur des applications à large périmètre.

4.1.5 Fetching et mise en cache avec React Query

TanStack Query centralise les requêtes HTTP et maintient un cache intelligent qui invalide ou rafraîchit automatiquement les données selon des clés de requête définies. Les optimisations comme les mutations optimistes anticipent les mises à jour du serveur pour offrir une réactivité maximale et les mécanismes de pagination et de préfetching facilitent la manipulation de gros volumes d'informations financières. Cette couche intermédiaire dispense l'équipe de gérer manuellement les états de chargement et d'erreur.

4.1.6 Runtime Bun

Bun a été choisi comme runtime JavaScript alternatif pour sa capacité à fournir un bundler, un minificateur et un test runner intégrés à très haute performance. La vitesse d'installation des dépendances, la compilation native et l'exécution des scripts de développement ou de linting sont considérablement accélérées grâce au moteur écrit en Zig. Cette plateforme garantit une expérience de développement homogène et un time-to-market réduit.

4.2 Backend monolithique

Nous nous sommes fortement inspirés par les dernières tendances en matière de décomposition modulaire. Nous avons aussi l'avantage inestimable de savoir que nous souhaitons éventuellement passer en microservice. Le monolithe modulaire était donc l'approche de choix. Une application en couche utilise traditionnellement 3 couches (Présentation, Application, Données). Une application en Onion avec un domaine va pousser la donne plus loin vers la couche centrale étant le domaine, la couche secondaire est la couche applicative, la couche tierce divisée en présentation et infrastructure. Ici la couche présentation et infrastructure sont fermées, elles ne communiquent pas entre elles, mais plutôt à travers la couche applicative par inversion de contrôle. Cependant, un monolithe modulaire nous force à dépasser l'architecture Onion. Nous devons interdire le couplage direct entre les différents domaines présent dans notre monolithe de telle sorte que les séparer ait un impact minimal. Nous ajoutons donc des couches verticales théoriques entre les différentes capacités de notre application. C'est par passage de messages, un peu comme le modèle acteur, que nous transférons de l'information. Cette séparation se poursuit dans les bases de données. Nous avons les fonctionnalités suivantes :

- Authentification et autorisation
- Gestion des nouvelles
- Gestion du temps
- Gestion des stocks
- Gestion de portfolio

Elles peuvent être séparées en 2 grandes catégories : utilitaire et domaine.

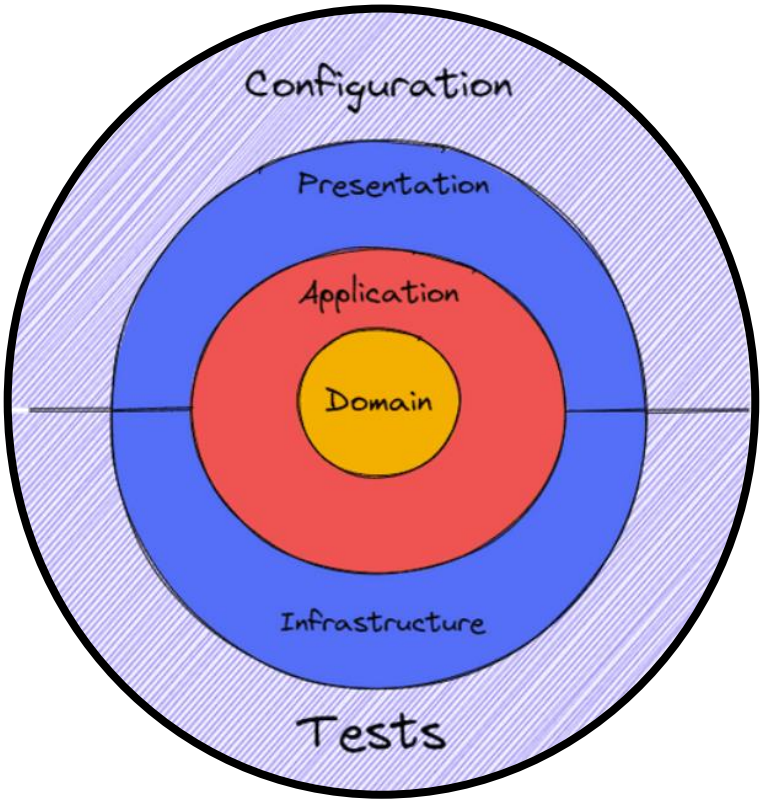

Figure 8.0 Représentation de la structure en onion de l'architecture

Authentification et Autorisation est une fonctionnalité utilitaire pour toutes les autres fonctionnalités. Elle a été implémentée à l'aide de Identity Framework de Microsoft, conjointement avec le système d'authentification et d'autorisation de ASP.net core. Cette combinaison permet à chaque route d'être protégé d'accès par des intrus. Cette fonctionnalité gère aussi les utilisateurs, à l'aide d'une base de données Postgresql. Elle s'intègre aussi avec RabbitMq pour la publication d'évènement comme la création d'un nouvel utilisateur.

Gestion des Nouvelles est une fonctionnalité du domaine. Elle fait partie d'un processus de ETL dans laquelle et s'occupe du **L**. Elle ingère et indexe les nouvelles envoyées par messages à RabbitMq. Pour cette solution nous utilisons deux bases de données. Une base de données MongoDB pour l'information concernant une nouvelle (metadata) comme son nom, auteur, le stock auquel elle est associée, si l'opinion est positive, négative ou neutre. Le texte de nouvelle est en réalité rarement nécessaire, la plupart des utilisateurs lisent le titre, regardent le score, mais ne lisent pas tous les articles. C'est pour cette raison que le texte lui-même est sauvegardé dans un Storage Blob de Azure. De cette manière lorsqu'un utilisateur demande à lire une nouvelle, est envoyé par Stream à partir du Blob. Chaque entrée MongoDB a bien évidemment une référence vers sa nouvelle dans le Blob. C'est un système performant et utilisé dans l'industrie pour le bas cout de sauvegarde en Blob.

Gestion du temps est une fonctionnalité en partie domaine et utilitaire. Pour qu'une preuve de concept soit intéressante, il faut simuler plus d'échange d'information qu'une fois par jour. Nous devons prendre contrôle du temps. Nous avons donc une horloge monotonique qui avance selon un multiplicateur donné et modifiable par REST. Nous pouvons accélérer le temps pour l'entièreté du système à n'importe quel instant. Nous allons voir que c'est essentiel pour les prochaines sections. Chaque fonctionnalité peut demander le temps actuel ou écouter pour un message (journée complétée) qui indique quel jour vient de finir.

Gestion des stocks a une responsabilité similaire à celle des nouvelles. C'est d'ingérer par message RabbitMQ (AMQP) et d'indexer les données pour un accès durable et consistent aux données de chaque symbole (stock) à travers le temps. Sa responsabilité domaine est aussi de rejouer les données d'un stock jusqu'à la date demandée et de donner les informations voulues pour le moment souhaité, similairement au *Event Sourcing*. Il faut savoir que la base de données a de l'information de prédiction future pour un stock et parfois il faut avoir sa valeur présente ou historique. Nous gardons une suite d'état daté en sauvegarde sur MongoDb. Cette

base de données est utilisée ici pour sa performance et pour l'agilité supplémentaire nécessaire lorsque nous ingérons des données d'une source distance par ETL.

Gestion de portfolio, fonctionnalité sommaire qui utilise plusieurs autres services. Chaque utilisateur à un portfolio. Celui-ci permet d'avoir une balance, d'acheter et de vendre des stocks. La balance du portfolio évolue avec le temps. Il se doit donc d'être au courant du temps actuel virtuel, de la valeur de ses stocks et de quel utilisateur il dessert. Juste en décrivant ces fonctionnalités, il est apparent que sa responsabilité et relationnel. Postgresql est la base de données employée pour cette fonctionnalité. Il a par définition un couplage avec d'autres modules décrits plus haut. C'est pour cette raison que même à l'intérieur du monolithe nous ne communiquons jamais latéralement dans une couche, mais purement par passage de messages entre le contrôleur et la couche applicative. Il n'y a jamais de couplage par référence dans le code. Le défi ici est de demeurer découplé et sans état, ce qui est réussi grâce à notre architecture Clean/Onion avec CQRS.

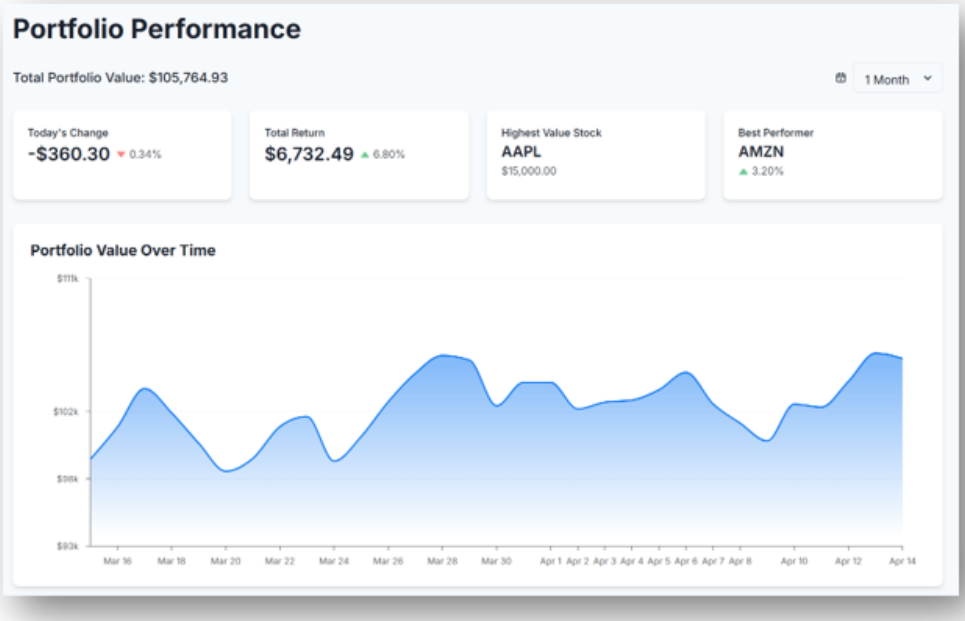

Figure 9.0 Vue de la perfomance du portfolio

Cette architecture de Monolithe Modulaire remplit donc nos attentes de n'avoir aucune dépendance inter module par référence (le couplage fonctionnel est nécessaire) ainsi que d'être entièrement testé par test d'intégration avec les vraies dépendances (Azurite, RabbitMq, MongoDb, Postgresql) qui sont déployées et réellement utilisées pour valider leur bon fonctionnement lors des tests.

4.3 Module IA monolithique

Dans la version monolithique de notre architecture, les deux modules qui composent notre application sont le module d'analyse de sentiment et le module de prédiction boursière. Chacun de ces modules est développé de façon indépendante, possède son propre environnement et expose ses endpoints sur un socket différent. Cette architecture présente plusieurs inconvénients majeurs :

1. **Redondance des ressources** : Chaque module maintient sa propre base de données et ses propres modèles, ce qui duplique les ressources et complique la synchronisation des données.
2. **Difficulté de maintenance** : Les mises à jour doivent être effectuées séparément sur chaque module, augmentant le risque d'incompatibilités.
3. **Scalabilité limitée** : L'architecture ne permet pas de scaler individuellement les composants en fonction de la charge.
4. **Gestion complexe des dépendances** : Les dépendances sont partagées entre les modules, rendant les mises à jour logicielles plus risquées.
5. **Points de défaillance uniques** : Un problème dans un module peut affecter l'ensemble du système.

Cette architecture a donc été remplacée par une approche microservices qui permet une meilleure séparation des responsabilités, une scalabilité indépendante des composants et une maintenance simplifiée.

CHAPITRE 5 FRAGMENTATION EN MICROSERVICES

5.1 Frontend

Dans notre contexte, découper l'interface utilisateur en micro-frontends aurait introduit une complexité disproportionnée par rapport aux bénéfices attendus. Le cœur de notre application repose sur une seule pile technologique React associé à TypeScript, avec Vite et Bun pour le développement et le bundling garantissant une cohérence visuelle et fonctionnelle sans nécessiter la coordination de multiples frameworks ou versions de bibliothèques. Avec une équipe restreinte de cinq étudiants, multiplier les dépôts, pipelines de build et processus de déploiement pour chaque fragment aurait alourdi la maintenance, augmenté les risques de régression et dilué la responsabilité de chacun.

Le micro-frontend révèle tout son intérêt dans des projets à grande échelle, intégrant des modules réutilisables dans plusieurs produits ou des systèmes *legacy* hétérogènes. Or, notre portail de prédiction boursière n'exige ni la gestion de portails externes ni la cohabitation de technologies disparates. Conserver un unique dépôt frontend permet de bénéficier d'un temps de compilation et de déploiement optimisé, d'une gestion centralisée des routes et d'un cache client unifié, tout en accélérant la mise à jour des dépendances et en assurant la cohérence des styles via une architecture de composants partagés et un système de design unifié. Cette approche, à la fois simple et robuste, correspond parfaitement à la taille de l'équipe et à l'étendue fonctionnelle de notre application.

5.2 Backend

Séparer un Monolithe Modulaire semble être une tâche relativement simple. Après tout, l'architecture est conçue pour être divisée et la division des concepts de domaine ne pourrait être plus simple. Dans les faits, il s'agissait simplement de copier-coller des fichiers. Les noms demeurent identiques. Cependant, le vrai défi est plus subtil : l'infrastructure. Alors que nous nous efforçons à rendre la couche applicative et le domaine découplé, ils sont tous ultimement couplés à des concepts non négociables. Par exemple, la gestion de l'authentification et de l'autorisation et la gestion des messages par RabbitMq. Comment s'assurer d'une logique distribuée exacte pour l'autorisation et l'authentification qui ne demande pas à chaque service d'appeler le service qui gère cette fonctionnalité? La solution n'est pas un micro-service, mais un code distribué. Nous avons implémenté une librairie de gestion de certificats HTTPS, d'encryptions et de gestion d'accès. Cette librairie au démarrage d'un service fait un appel au service autorisation et d'authentification pour demander sa clé publique. Par la suite tout JWT qui est reçu par un micro-service valide la signature de celui-ci avec la clé publique de la clé privée qui l'a généré. Nos communications sont elles aussi encryptées par TLS et puisque nous utilisons un certificat de développement qui doit être reconnu par la machine, cette librairie contient celui-ci pour qu'il soit réutilisé par chaque service afin de garantir que tous nos échanges inter services sont sécuritaires. Toutefois, notre travail ne s'arrête pas là. La librairie doit aussi offrir à chaque service une façon de faire leurs tests d'intégrations d'une manière valide et sécurisée. Chaque requête de tests doit elle aussi être proprement authentifiée. De plus, cette librairie qui contient du code pour chaque service et du code d'aide au test à chaque service doit elle aussi voir tout son code testé. Des tests d'intégrations avec un réel déploiement ont été ajouté à la libraire. De plus elle doit désormais être facile à distribué sur tout ordinateur qui construit une image docker dans laquelle elle est une dépendance. La librairie a donc été ajoutée (mais non listée) sur Nuget.org, la source des librairies standard pour les images docker d'AspNet Core. Tout ce travail a ensuite été imité pour la librairie de communication par RabbitMq. Chaque service doit être capable de définir des messages, des publicateurs, des

consommateurs et le tout avec une topologie cohérente de messagerie qui comprend comment router chaque message avec le protocole AMQP 0.9.1 qui est la version plus complexe, mais aussi plus complète de AMQP 1.0 qui est utilisé par plusieurs services infonuagiques comme Azure Service bus. Une fois cela géré, il restait à transformer certaines commandes en client HTTPS. Par exemple le portfolio doit appeler le service d'Authentification pour avoir l'identifiant du portefeuille d'un utilisateur, le service de temps pour l'heure virtuel actuel et le service de stock pour le prix actuel d'un stock avant de déterminer si un achat est valide.
En somme, ce qui est habituellement considéré comme la complexité première de la division en micro-services était en réalité bien préparé et c'est l'infrastructure qui était plus complexe à distribuer de façon cohérente. Ironiquement, les tests d'intégration ont aussi perdu en qualité avec la division parce qu'un appel du portfolio doit mock la réponse de ses différentes dépendances REST.

5.3 Module AI

La décomposition de l'architecture monolithique de stock-ai en microservices a été réalisée selon une approche méthodique et progressive. La première étape a consisté en une analyse approfondie du système existant pour identifier les composants fonctionnels distincts et définir les limites de service appropriées. Cette phase de planification a permis d'établir clairement les interfaces de communication entre les différents services et de préparer une stratégie de migration progressive.
La séparation des services a été effectuée en créant des composants indépendants et spécialisés: le **DataService** pour la gestion des données boursières, le **ModelService** pour la gestion des modèles d'IA, le **TrainingService** pour l'entraînement des modèles, le **PredictionService** pour la génération des prédictions, et le **NewsService** pour l'analyse des nouvelles financières. Chaque service a été conçu pour être autonome et responsable d'une fonctionnalité spécifique. La définition des interfaces a été une étape cruciale, avec la création d'APIs REST pour chaque service, la standardisation des formats de données et la documentation complète des endpoints.

Cette standardisation a permis d'assurer une communication fluide entre les services tout en maintenant une certaine flexibilité pour les évolutions futures.

La migration des données a nécessité une restructuration complète des bases de données, avec la création de schémas de données spécifiques à chaque service. Cette étape a été accompagnée de la mise en place de mécanismes de synchronisation pour assurer la cohérence des données entre les différents services. L'implémentation des services a été réalisée de manière progressive, avec un accent particulier sur les mécanismes de communication et de résilience. Chaque service a été développé indépendamment, avec ses propres tests unitaires et d'intégration, tout en assurant la compatibilité avec les autres composants du système. Le déploiement a été effectué de manière progressive, en utilisant Docker et Docker Compose pour faciliter la mise en production. Cette approche a permis de déployer chaque service individuellement, tout en maintenant la stabilité du système global. La documentation et la formation ont été des aspects essentiels de cette transformation, avec la création de guides détaillés et la formation des équipes aux nouvelles pratiques de développement et de maintenance.

Finalement, une phase d'optimisation et d'amélioration continue a été mise en place, avec l'analyse régulière des performances et l'ajustement des configurations pour répondre aux besoins évolutifs du système. Cette décomposition en micro-services a finalement permis d'obtenir une architecture plus modulaire, plus facile à maintenir et à faire évoluer, tout en améliorant significativement la scalabilité et la résilience du système.

CHAPITRE 6 ÉTAT DU SYSTÈME

6.1 Architecture Backend

L'architecture du backend est conçue avec une approche modulaire et distribuée, centrée sur la communication entre microservices. Le AuthNuget est un package NuGet qui encapsule toute la logique d'authentification et d'autorisation. Il implémente des mécanismes de validation de jetons JWT, de gestion des rôles et des permissions, ainsi que des interfaces pour l'intégration avec différents fournisseurs d'identité. Le package AuthNuget.Testing fournit des outils et des tests pour valider le comportement du package dans différents scénarios, facilitant ainsi le développement et la maintenance.Le LocalNugetFeed agit comme un dépôt local pour les packages NuGet, stockant les différentes versions des packages AuthNuget et RabbitMqNuget. Il contient également les symboles de débogage (.snupkg) qui permettent un débogage plus efficace en fournissant des informations détaillées sur l'exécution du code. Ce dépôt local est essentiel pour le développement et les tests, car il permet de travailler avec des versions spécifiques des packages sans dépendre de sources externes.Le RabbitMqNuget est un package qui simplifie l'utilisation de RabbitMQ dans les applications .NET. Il fournit des abstractions pour la publication et la souscription aux messages, la gestion des files d'attente, et la configuration des échanges. Le package RabbitMqNuget.Testing offre des outils pour tester l'intégration avec RabbitMQ, simulant différents scénarios de communication et de défaillance. Le dossier Services contient les implémentations des microservices qui utilisent ces packages. Chaque microservice est conçu pour être autonome, avec sa propre base de données et sa propre logique métier. Les microservices communiquent entre eux via RabbitMQ, en utilisant le RabbitMqNuget pour gérer les messages. L'authentification et l'autorisation sont gérées par le AuthNuget, assurant que seuls les utilisateurs et les services autorisés peuvent accéder aux ressources.Cette architecture permet une séparation claire des responsabilités, une maintenance simplifiée, et une scalabilité accrue. Les microservices

peuvent être déployés indépendamment, et les packages NuGet facilitent la réutilisation du code et la standardisation des pratiques de développement.

Voici donc l'architecture finalisé de notre système:

- Le Client (frontend)
- Le backend qui roule sous une grappe Kubernetes
- Différent micro-services
    - Auth: Authentification des utilisateurs.
    - Portfolio: Gestion du portefolio des utilisateurs.
    - Stock: Persistance sur le prix des actions.
    - News: Persistance sur les actualités reliées aux actions
    - Prédictions (AI): Modèle IA pour la prédiction des actions.
    - Intention (AI): Modèle IA pour l'analyse de sentiments des actualités
    - Rabbitmq: Message Queuing entre Prédictions/Stock et Intention/News.
    - PostgresSQL, MongoDb, Azurite: Persistance.

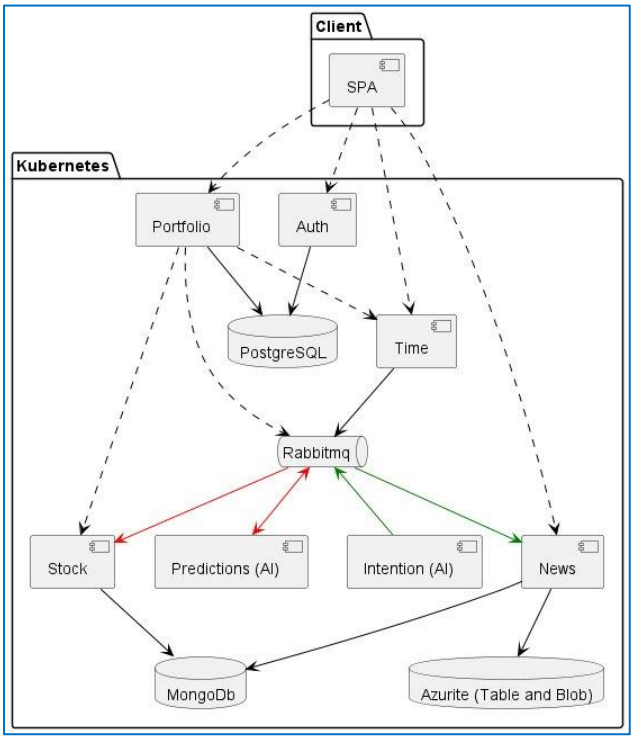

Figure 10.0 Schéma relationnel des différents microservices

6.2 Architecture IA

La création des services indépendants a été réalisée avec une attention particulière aux responsabilités spécifiques de chaque composant.

Le **DataService** a été conçu pour gérer exclusivement les données boursières, avec des fonctionnalités de collecte, de nettoyage et de stockage des données historiques et en temps réel. Il implémente des mécanismes de cache et de mise en cache pour optimiser les performances et réduit la charge sur les sources de données externes.

Le **ModelService**, quant à lui, est responsable de la gestion complète des modèles d'IA, incluant leur versioning, leur stockage et leur déploiement. Il gère à la fois les modèles LSTM pour la prédiction des prix et les modèles d'analyse de sentiment, avec un système de métadonnées pour suivre les performances et les paramètres de chaque modèle.

Le **TrainingService** a été développé comme un service autonome pour l'entraînement des modèles, avec des capacités de traitement parallèle et de gestion des ressources. Il implémente des algorithmes d'optimisation des hyperparamètres et des mécanismes de validation croisée pour assurer la qualité des modèles entraînés.

Le **PredictionService**, conçu pour la génération des prédictions, intègre des mécanismes de mise en cache des prédictions fréquemment demandées et des algorithmes de fusion pour combiner les prédictions de différents modèles. Il gère également la calibration des scores de confiance et l'ajustement des prédictions en fonction des conditions de marché.

Enfin, le **NewsService** a été développé pour l'analyse des nouvelles financières, avec des capacités de traitement du langage naturel et d'extraction d'entités. Il implémente des algorithmes d'analyse de sentiment spécifiques au domaine financier et des mécanismes de pondération pour tenir compte de la pertinence et de la fraîcheur des nouvelles.

Chaque service a été conçu avec ses propres mécanismes de résilience, de monitoring et de logging, tout en maintenant des interfaces claires et bien définies pour la communication avec les autres services. Cette séparation des responsabilités a permis d'obtenir une architecture plus modulaire et plus facile à maintenir.

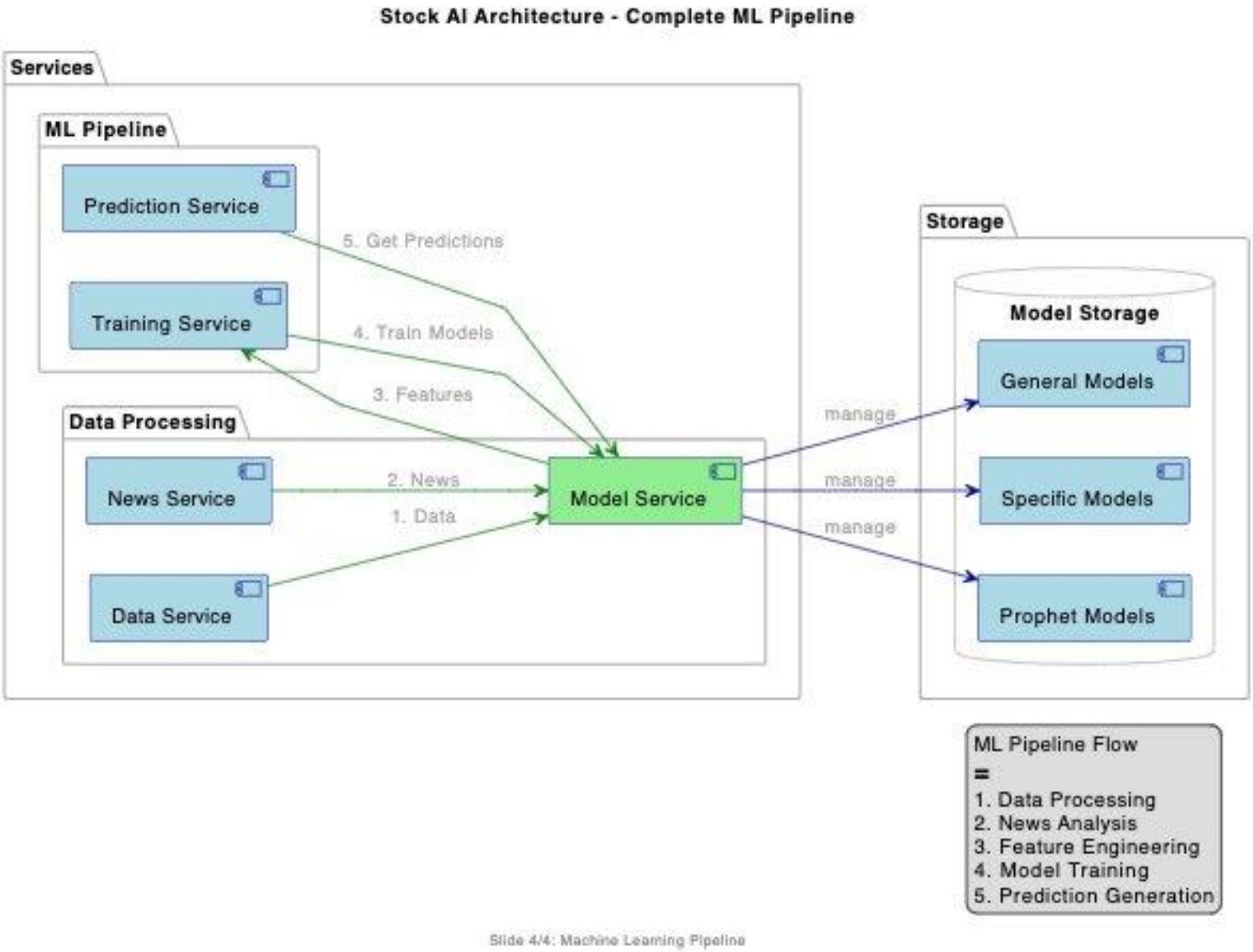

Figure 11.0 Schéma relationnel des différents services du backend IA

CHAPITRE 7 CONCLUSION

Ce projet de fin d'études nous a menés à concevoir et implémenter une plateforme web de prédictions boursières propulsée par l'intelligence artificielle. Dans un premier lieu, nous avons utilisé une architecture monolithique, puis nous avons migré avec une architecture en micro-services. Plusieurs leçons ont été apprises à travers tous les problèmes qui se sont présentés à nous. Nous avons réalisé à quel point la communication entre membres de l'équipe est importante lors de la réalisation de tâches et du partage de connaissance. L'énonciation claire des tâches à réaliser tout au long d'un projet est également extrêmement importante afin de garder le cap sur l'objectif final. Avec du recul, l'utilisation d'un répertoire de cas d'utilisation aurait probablement allégé une bonne partie de nos problèmes. De plus, nous avons quelque peu sous-estimé le travail de liaison du frontend avec le backend. En effet, aux termes du projet, chacun était individuellement fonctionnel, mais le raccord des deux a suscité quelques problèmes tardifs. Ces problèmes auraient pu être palliés en effectuant plus de tests d'acceptation.

En somme, la réalisation de ce travail fut très enrichissante pour tout le monde dans l'équipe. Nous en retirons beaucoup tant d'un aspect théorique que sur les manières de travailler efficacement en équipe. Pour la suite, plusieurs améliorations sont possibles. Par exemple, la décomposition des services dans le backend IA afin d'offrir un service de gestion des modèles plus généraliste. Cela permettrait d'entraîner un modèle pour tous les stocks à la place d'entrainer un modèle pour chaque stock. Enfin, bien que notre plateforme ne puisse pas nous permettre de devenir millionnaires du jour au lendemain (quoique, on peut toujours rêver!), elle démontre la démocratisation des outils d'analyse financière autrefois réservés aux grandes institutions. Dans un monde où l'intelligence artificielle continue de redéfinir les règles du jeu, notre projet n'est qu'un avant-goût de ce que la fusion entre finance et technologie nous réserve. Après tout, si nous avons réussi à faire collaborer des modèles d'IA avec des cours boursiers imprévisibles, l'avenir ne peut être que prometteur... ou du moins, légèrement plus prévisible!

BIBLIOGRAPHIE

- Fischer, T., & Krauss, C. (2018). Deep learning with long short-term memory networks for financial market predictions. *European Journal of Operational Research*, *270*(2), 654-669. https://doi.org/10.1016/j.ejor.2017.11.054
- Scott Millet, Nick Tune (2015) Patterns, Principles, and Practices of Domain-Driven Design
- Evafachria. (2023). Stock price prediction with FB Prophet. *Medium*. https://medium.com/@eva21fachria/stock-price-prediction-with-fb-prophet-d67fe46ebc9
- *Refactoriser un monolithe en microservices*. (2024, 26 juin). Google Cloud. https://cloud.google.com/architecture/microservices-architecture-refactoring-monoliths
- Modèle finbert tone utilisé https://huggingface.co/yiyanghkust/finbert-tone
- Module mémoire de LangChain https://python.langchain.com/api_reference/langchain/memory.html#langchain-memory
- API yfinance : https://yfinance-python.org/