Groupe 5
FOEX Vick, GHANDOUR Jad, RAJAUD Eliott, TAIDER Silia

Rapport Projet

# City Manager Reinforcement Learning

## 1. Sources

- Github: https://github.com/siliataider/CityManager-Reinforcement-Learning

- Teaser vidéo: https://youtu.be/YjdlPgHJ2UE

- Application déployé: https://citymanager-app.onrender.com/

- Présentation avec animations et vidéos: https://docs.google.com/presentation/d/1zDpo5Tk8YWrd6qleoZuT8AjfJGk2-HHX6jXKon48nGM/edit?usp=sharing

- Trello: https://trello.com/w/citymanagergr5

## 2. Organisation du projet (gestion du projet)

Le gestion du projet à été réalisée au travers de plusieurs outils.

Tout d'abord, nous avons utilisé GitHub pour la gestion du versionning de notre projet. Des branches ont été créées pour chaque développeur et pour des tâches spécifiques.

De plus, afin de se répartir correctement les différentes tâches à réaliser, un Trello a été mis en place avec plusieurs espaces de travail: Dashboard, Front, Back, IA.

Enfin le déploiement de notre application a pu être réalisé via Dockerhub et Render.

### Répartition des tâches par personne:

| Silia TAIDER | Eliott RAJAUD | Vick FOEX | Jad GHANDOUR |
|---|---|---|---|

| | | | |
|---|---|---|---|
| *Architecture<br><br>*IA: DQ-Learning<br><br>*Backend Agent<br><br>* Front<br><br>*Pipeline CI/CD | *Architecture<br><br>*IA: Q-Learning<br><br>*Front<br><br>*Backend Agent<br><br>*Communication Simulation - Agent | *Architecture<br><br>*Front<br><br>*Communication Simulation - Front<br><br>*Implémentation carte réelle<br><br>*Implémentation déplacement | *Architecture<br><br>*Front<br><br>*Simulation<br><br>*Communication Simulation - Front<br><br>*Communication Simulation - Agent |

## 3. Estimation de l'investissement

| Silia TAIDER | Eliott RAJAUD | Vick FOEX | Jad GHANDOUR |
|---|---|---|---|
| 25% | 25% | 25% | 25% |

## 4. Spécification fonctionnelles et techniques

### Généralités

Le but de l'application est de simuler les comportements d'individus dans un milieu urbain. Un individu est représenté par un modèle IA appelé agent dont les comportements seront influencés par un système de récompense associé au maintien de ses besoins en tant qu'individu. Ces différents besoins sont créés et remplis par des activités associés à des bâtiments dédiés, eux aussi faisant partie de la simulation. Cette simulation peut être configurée et observée continuellement via une interface homme-machine.

### Agents

- Trois besoins: Ils ont besoin de se nourrir, de travailler pour acheter de la nourriture, et de se reposer pour continuer à travailler.
- Nos agents peuvent êtres simuler via deux modèles: Q-Learning ou de DQ-Learning
- Ils possèdent 5 états: timestamp, weather, hunger, energy et money. Ces états sont discrétisés en 108 états pour le Q-Learning.
- Au cours de la simulation l'agent va choisir une action, calculer une reward en fonction de son état pour ensuite les modifier et enfin s'entraîner.
- Plus le temps de simulation augmente, plus l'agent va exploiter ses résultats au lieu d'explorer.

- L’agent possède des points de vie, calculés en fonction de son état, et meurs s’il est inférieur à 20% de l’état global.
- La reward moyenne est aussi calculée à chaque épisode pour tracer son évolution.
- Une possibilité de sauvegarder un agent au comportement particulièrement intéressant est aussi disponible, mais la possibilité de charger ce modèle n’as pas été complètement implémenter.

### Carte

- Elle comporte des bâtiments: Restaurants, où l’agent se nourrit en échange d’argent. Des lieux de travail où il gagne de l’argent en échange d'énergie. Et enfin des maisons, où les agents récupèrent de l'énergie.
- Une simulation de déplacements réels est réalisée grâce à l’API Openrouteservice. En effet, celle-ci nous permet de récupérer les coordonnées du chemin à suivre pour que nos agents atteignent leur objectif. De plus, pour éviter que les agents ne se superposent, nous faisons en sorte que les agents se repoussent s'ils sont sur le point de rentrer en collision.
  À chaque déplacement, l’agent prépare les vecteurs qui lui permettent de suivre le chemin vers le prochain objectif, ces vecteurs sont ensuite pondérés par la présence des autres agents proche de lui (la distance augmente proportionnellement la force de répulsion). Le mouvement est ensuite appliquer. Enfin, afin d’éviter de trop nombreux appels à l’API, les chemins utilisés sont mis en cache et réutilisés si un agent doit emprunter un chemin déjà utilisé.

### Interface Homme-Machine:

- Elle va permettre de configurer la simulation: nombre d’agents, emplacement des bâtiments, durée d’un épisode.
- Nous pourrons interagir avec la simulation en cours: changer la vitesse de la simulation, modifier la météo.
- Il est possible aussi de pouvoir arrêter et recommencer la simulation à tout moment.
- Une visualisation de données de l’agent est possible, notamment un graphique du reward moyen, l’état de l’agent, ses points de vie, ou encore s’il est mort.

## 5. Q-Learning vs DQ-Learning

Nous avons donc utilisé deux modèles différents pour l'entraînement de nos agents. Ces deux modèles utilisent la même fonction de reward afin de pouvoir comparer les résultats. Peu importe l’agent, l'entraînement va fonctionner de la même façon. Tout d’abord la simulation commence avec un exploration rate élevé afin d’explorer un maximum de possibilité, puis au cours de la simulation ce learning rate va diminuer afin

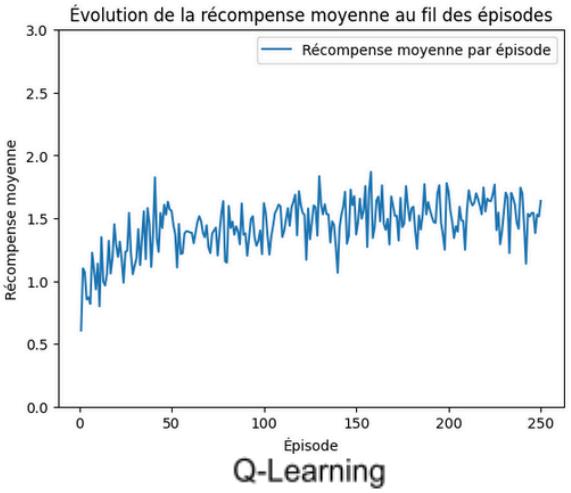
d’exploiter les résultats obtenus. Dans notre application, nous pouvons définir la durée d’un épisode pour impacter l’exploration rate et donc augmenter ou diminuer la durée d'apprentissage plus ou moins rapidement.
A la fin de chaque épisode, les états de tous les agents (morts ou pas) sont réinitialisés afin de continuer l'entraînement. Les agents morts reçoivent une pénalité.
Une action est donc choisie au départ (aléatoirement si exploitation, sinon la meilleure action à réaliser en fonction de l’état de l’agent).
Ensuite, le calcul de la reward en fonction de l’action et de l’état sont réalisés pour ainsi retourner l’état suivant ainsi que la reward à l’agent.
L’agent va ensuite pouvoir entraîner son modèle grâce à son état, son état suivant, l’action choisit et sa reward. L'entraînement va être différent selon le modèle.

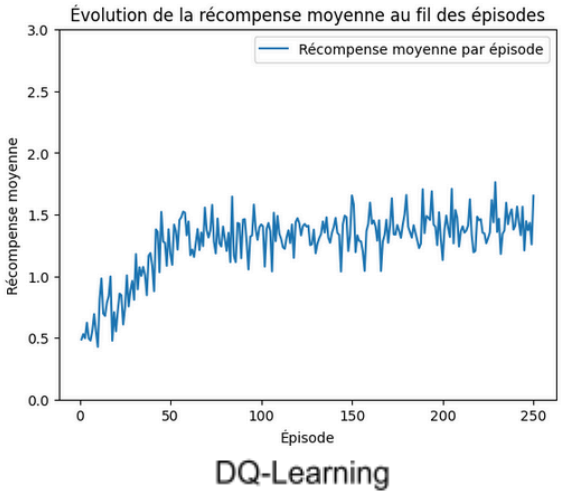
- Le premier modèle, Q-Learning, utilise une table qui associe une valeur pour un état et une action donnée. Ainsi, il nous faut donc discrétiser nos états afin de pouvoir créer notre table. A l’aide de l’équation de Bellman, nous allons calculer une valeur que nous allons associer à l’état actuel et l’action choisie. Après plusieurs itérations on se retrouve donc avec un table, qui pour chaque état va avoir une valeur plus ou moins élevée pour chaque action associée. La plus haute valeur représentant la meilleure action pour l'état désigné.

- La 2ème approche qu’on a utilisé pour entraîner nos agents est le Deep Q-Learning. Cette méthode utilise des Deep Q-Networks pour appliquer plus ou moins la même logique que la Q-table en utilisant des réseaux de neurones pour estimer les meilleures actions possibles. Notre modèle comprend 3 couches:

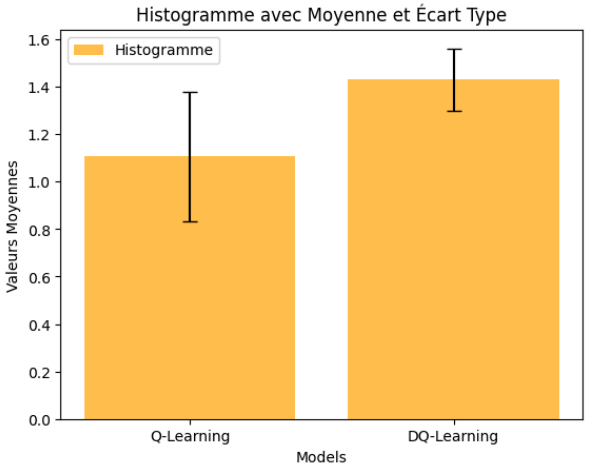
- **Notre couche d’entrée:** Contient les 5 composants de l'état de l'agent (heure, météo, faim, énergie, argent).
- **Une couche cachée pour l’apprentissage et prédiction**: Le réseau de neurones apprend à prédire la meilleure action possible dans un état donné en s’adaptant en continu pour maximiser la récompense totale que l'agent reçoit.
- **Une couche de sortie:** Cette couche représente le meilleur choix entre nos 3 actions possibles, chacune aura un score, L'agent choisit l'action avec la valeur prédite la plus élevée (stratégie d'exploitation), mais il explore aussi parfois aléatoirement pour découvrir de nouvelles stratégies (stratégie d'exploration).

Ainsi après plusieurs épisodes et les même paramètre de simulation, nous pouvons observer les résultats suivant:



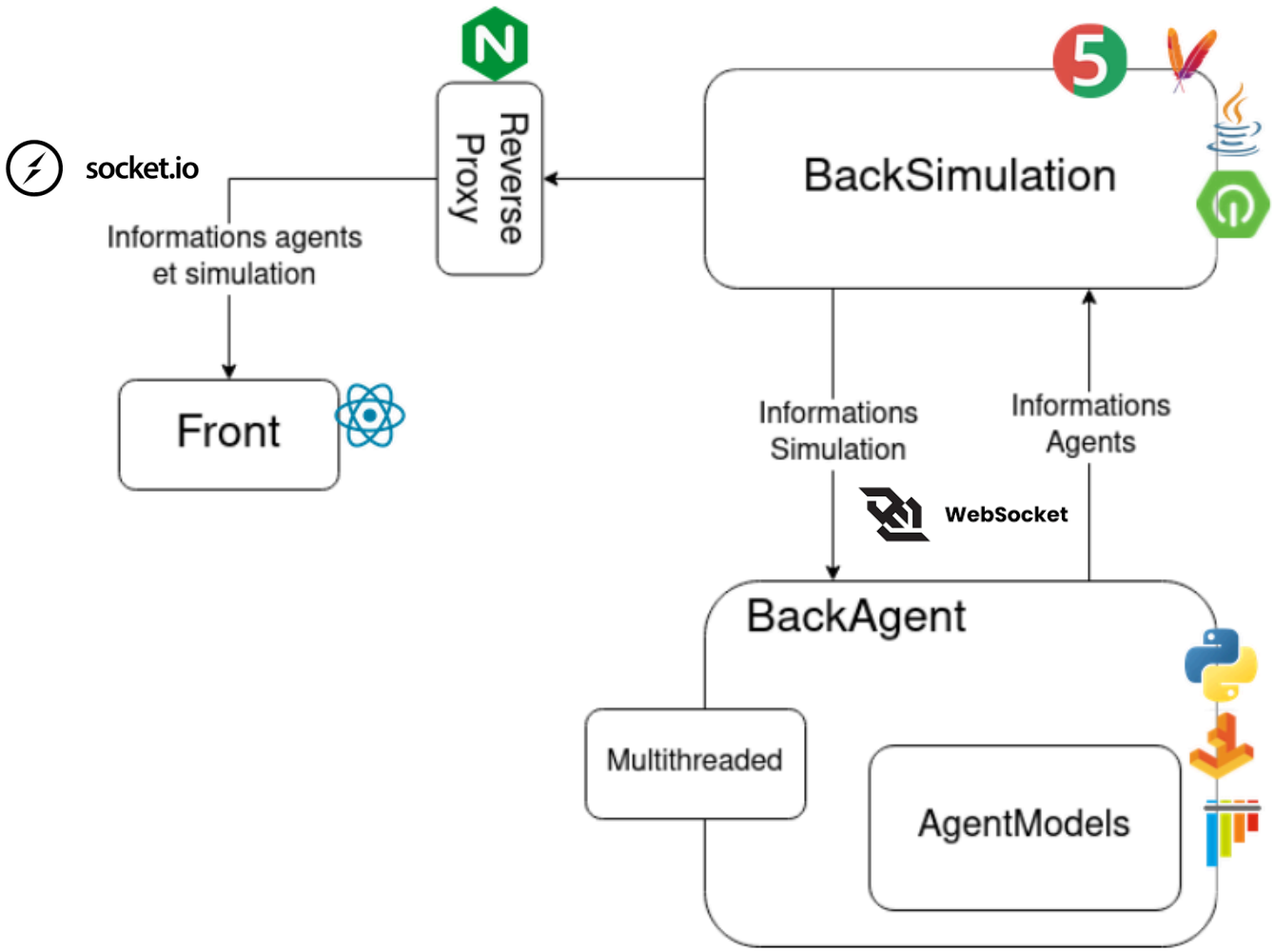

On remarque bien que nos deux modèles ont une période d'apprentissage, puis au bout d'environ 50 épisodes ils exploitent leur résultat. Cependant si on visualise la reward moyen pour les deux modèles et l'écart type avec les 10 mêmes données de tests utilisant les modèles entraînés précédent, on obtient l'histogramme suivant:




| | Avantages | Inconvénients |
|---|---|---|
| Q-Learning | ➔ Rapide à apprendre Efficace si peu d'états.<br>➔ Simple à mettre en place | ➔ Nécessite de discrétiser les états.<br>➔ Si on augmente le nombre d'état, on peut se retrouver avec une trop grosse table. |
| DQ-Learning | ➔ Facilement scalable<br>➔ Peut gérer des états complexes et des variables continues.<br>➔ Performant si on lui laisse le temps de s'entraîner.<br>➔ Capable de généraliser à partie d'expériences passées. | ➔ Complexe à mettre en place<br>➔ Prend plus de temps à apprendre<br>➔ Demande plus de puissance de calcul |

## 6. Schéma d'architecture technique et choix technologique

Architecture Globale




### Front:

- Framework **ReactJS** (Javascript)
- API OpenStreetMap pour la carte
- Gestion de CORS via reverse proxy

### Simulation:

- Framework Springboot (**Java**).
- Communication avec le Front via **Netty-socketio**, qui est une implémentation de serveur socketio (typiquement utilisé avec NodeJS) pour Springboot.

NodeJS est plus souvent utilisé pour servir de backend à un frontend en ReactJS. Cependant, nous souhaitions utiliser une conception orientée objet pour la simulation, à laquelle Java est bien plus adaptée. Tout problème d'incompatibilité avec React peut être résolu par des dépendances externes spécialisées comme Netty-socketio.

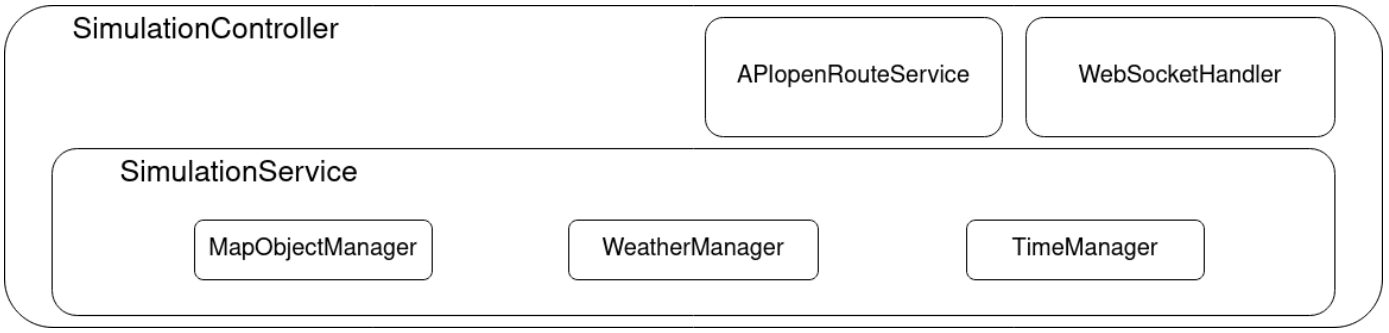
## Agent:

- **Python**
- TensorFlow pour la gestion des modèles IA.
- Serveur Python **Websockets** pour communiquer avec le bloc Simulation
- **MultiThreading** pour l'entraînement de nos agents. 3 Threads sont créés, ce qui va permettre d'avoir 3 agents qui s'entraînent en même temps.

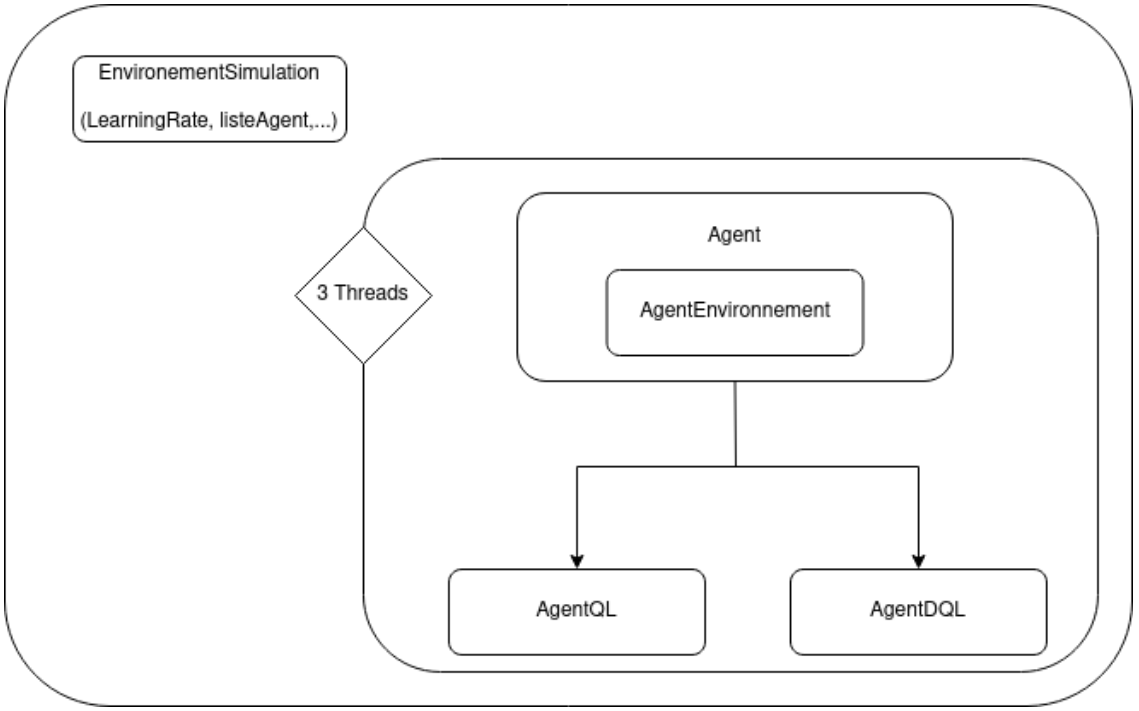
## Déploiement:

- Docker
- Render

## Block Simulation



## Block Agent (python)




## Conception Objet des éléments de la carte (batiments et agents)

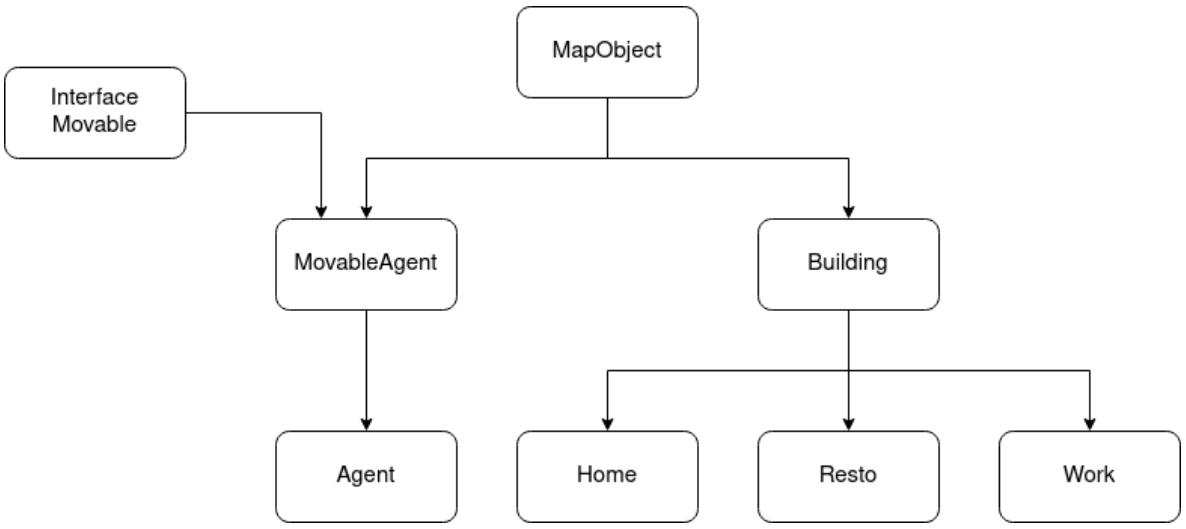
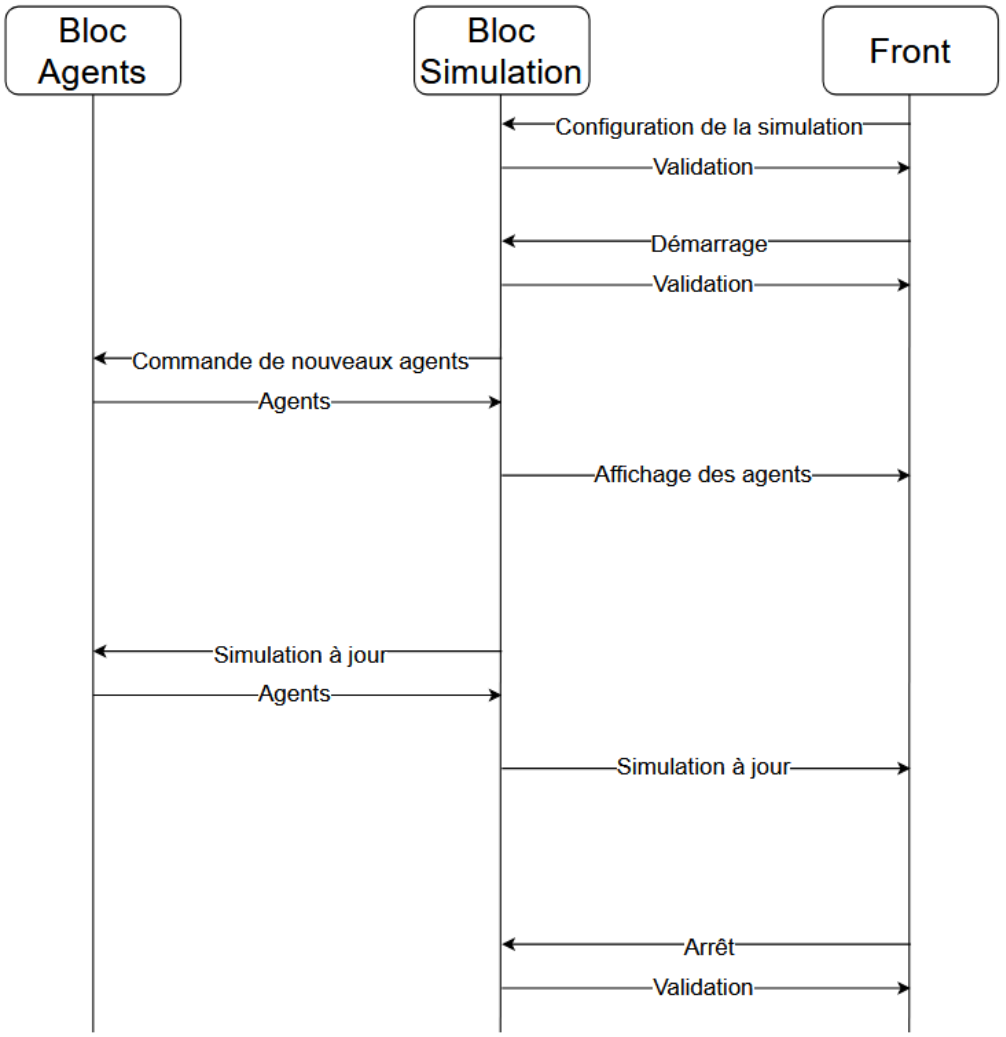


Diagramme de séquence: use case




## 7. Tests et Déploiement

### Tests BackSimulation:

- Initialiser un objet SimulationService, y ajouter des MapObjects (Agents, et bâtiments). Convertir cette simulation en JSON grâce à sa méthode dédiée toJSONString. Si le résultat est un JSON valide, le test est validé.
- Initialiser un objet SimulationService vérifier si la solution est valide grâce à sa méthode dédiée verify. Peu à peu ajouter des bâtiments de différents types et faire une nouvelle vérification à chaque fois. Une dernière vérification est faite avec plus de 0 agents et au moins un bâtiment de chaque type. Le test est validé si toutes les vérifications sont négatives sauf la dernière.

## Tests BackAgent:

- Des tests unitaires ont été réalisés afin de tester les principales fonctions de nos agents telle que le choix de l'action lors de l'exploitation, ou encore l'entraînement de nos modèles avec des données prédéfinies afin de visualiser l'état suivant de nos agents.
- Un test plus global a été aussi réalisé permettant de créer directement un client dans le back python afin de pouvoir tester la communication web-socket et les différentes actions en fonction des données envoyées.

## CI/CD:

Dans le cadre du développement et du déploiement de l'application CityManager-App, une pipeline CI/CD complète et robuste a été mise en place, implémentée via GitHub Actions et intégrée avec Docker et Render. Cette pipeline comprend plusieurs étapes clés pour garantir la qualité et la fiabilité de l'application.

D'abord, à chaque pull request sur la branche principale, le code est automatiquement récupéré et une image Docker est construite à partir d'un Dockerfile. Cette image inclut un environnement Java (OpenJDK 17), Python avec pip, Node.js via NVM, ainsi que d'autres dépendances nécessaires comme Maven et Nginx. Le script `docker_launcher.sh` est utilisé pour orchestrer le démarrage des services de l'application, y compris le proxy Nginx, les backends en Python et Java, et le frontend React.

Après la construction de l'image Docker, des tests automatiques sont exécutés séparément pour les composants backend Python et Java pour s'assurer de leur bon fonctionnement. À l'issue de ces tests, si aucune erreur n'est détectée, la pipeline procède au déploiement de l'application via Render, en utilisant un script dédié.

Enfin, une vérification post-déploiement est réalisée pour confirmer le bon fonctionnement de l'application déployée.