# 🥡 Simulation de Livraison de Nourriture avec Agents (Mesa + Flask)

|  |
| --- |
| Une image contenant texte, capture d’écran, conception  Le contenu généré par l’IA peut être incorrect. |

**🔎 Introduction**

Ce projet est une **simulation basée sur des agents (Agent-Based Model)** développée avec **Mesa 3.x** (un framework Python pour les ABM) et une interface web interactive construite avec **Flask** et **JavaScript**.

L’objectif principal est de **modéliser un système de livraison alimentaire** où des agents transporteurs récupèrent de la nourriture auprès de donneurs (par exemple, des restaurants, supermarchés...) et la livrent à des bénéficiaires dans le besoin (ONG, banques alimentaires...).

La simulation est alimentée par des **données réelles de gaspillage alimentaire**, extraites d’un fichier CSV, afin de rendre les scénarios plus crédibles.

**🧩 Fonctionnalités implémentées**

✅ **Représentation visuelle sur grille** des agents :

* Donneurs (bleu), Bénéficiaires (vert), Transporteurs (rouge)

✅ **Mise à jour en temps réel** de la simulation :

* Déplacements des agents visibles sur une grille dynamique
* Suivi automatique du nombre de livraisons réussies et des échecs

✅ **Statistiques en direct** affichées sur l’interface web

✅ **Cycle de simulation contrôlable** via boutons :

* Démarrer, Mettre en pause, Réinitialiser

✅ **Lecture de données externes (CSV)** pour définir la quantité de nourriture initiale des donneurs

✅ **Serveur web Flask** pour gérer la logique backend et servir la page

✅ **JavaScript côté client** pour afficher dynamiquement les agents et interagir avec les contrôles utilisateur

**🧪 Technologies utilisées**

| **Technologie** | **Utilisation principale** |
| --- | --- |
| **Python 3** | Langage principal du projet |
| **Mesa 3.x** | Moteur de simulation orienté agents |
| **Flask** | Backend léger pour le serveur web |
| **JavaScript** | Dynamique de la grille et des boutons sur la page |
| **HTML/CSS** | Interface utilisateur (visualisation grille + stats) |
| **Pandas** | Lecture des données CSV |
| **Dash (Plotly)** | création d'applications web |

**🧠 agent.py – Définition des agents**

Ce fichier contient trois classes d’agents :

* **DonorAgent** (bleu) : Fournit de la nourriture (quantité initiale définie depuis le CSV).
* **RecipientAgent** (vert) : Reçoit la nourriture.
* **TransportAgent** (rouge) : Va chercher la nourriture et la livre.

**Logique principale du transport :**

* Si le transporteur ne transporte rien, il cherche un donneur avec de la nourriture.
* Sinon, il cherche un bénéficiaire pour livrer.
* Échec ou succès mis à jour dans model.

**🧩 model.py – Modèle principal Mesa**

Contient :

* Une grille (MultiGrid) où les agents sont placés.
* Lecture des données à partir du fichier CSV réel.
* Création des agents (donneurs, bénéficiaires, transporteurs).
* Méthode step() pour avancer la simulation.
* Méthode get\_grid\_state() pour retourner l’état visuel.

**Variables clés :**

* successful\_deliveries : livraisons réussies.
* failed\_attempts : tentatives échouées.

**🖼️ visualization.py – Configuration visuelle des agents**

Configure la **couleur des agents** pour le rendu :

* Bleu = Donneur
* Vert = Bénéficiaire
* Rouge = Transporteur

Utilisé surtout dans une version avancée avec mesa.visualization, mais tu peux l’étendre ou le remplacer si tu veux un canvas dynamique dans JS (ce que tu fais déjà !).

**🌐 gui\_mesa.py – Serveur Flask**

Routes :

* / : charge la page HTML.
* /grid\_data : appelle model.step() et retourne les données de la grille + statistiques.
* /reset : réinitialise la simulation.

Tu as aussi un try/except très utile pour le debug avec traceback 👍

**🧠 script.js – Interaction côté client**

Responsable de :

* Afficher la grille en HTML (agents positionnés dynamiquement).
* Mettre à jour les stats en temps réel (✅ Deliveries, ❌ Failures).
* Gérer les boutons **Démarrer**, **Pause**, **Réinitialiser**.

**Fonctions clés :**

* fetchGridData() : appelle le backend et met à jour la vue.
* renderGrid() : crée les divs pour chaque cellule agent.
* startSimulation() / pauseSimulation() / resetSimulation() : contrôles utilisateur.

**🧾 index.html – Interface HTML**

Composants :

* Un conteneur #simulation-container pour la grille (200x200px).
* Boutons de contrôle.
* Un div #status pour les statistiques en direct.
* Chargement du JS via /static/js/script.js.

**📊 cleaned\_food\_wastage\_data.csv**

Contient les données réelles de gaspillage alimentaire. Chaque ligne représente une quantité de nourriture gaspillée, utilisée pour initialiser les DonorAgent.

**⚙️ Installation & Lancement**

**🧪 1. Environnement virtuel (optionnel mais recommandé)**

python -m venv venv

source venv/bin/activate # ou venv\Scripts\activate sur Windows

**📦 2. Dépendances à installer**

pip install mesa flask pandas

**🚀 3. Lancer le serveur Flask**

python gui\_mesa.py

Puis ouvre <http://127.0.0.1:5000> dans ton navigateur.

# Interface de Visualisation Dash pour la Simulation de Livraison Alimentaire

|  |
| --- |
| Une image contenant texte, capture d’écran, Police, ligne  Le contenu généré par l’IA peut être incorrect. |

l'interface de visualisation construite avec **Plotly Dash** pour une simulation de distribution alimentaire, basée sur un modèle agent Mesa. Cette interface permet d'analyser les performances, les activités des agents et de produire des prédictions sur les livraisons à partir de la base de données cleaned\_food\_wastage\_data.csv.

L'objectif est de respecter rigoureusement les exigences pédagogiques suivantes :

* Intégration des données du modèle Mesa avec Dash
* Trois onglets (Métriques, Logs Agents, Prédictions)
* Filtres interactifs prévus
* Affichage : graphique temporel, diagramme circulaire, carte thermique

**🧩 Structure du fichier dash\_gui.py**

**Chargement des données CSV**

***food\_data = pd.read\_csv("data/cleaned\_food\_wastage\_data.csv")***

La base contient les colonnes comme : type\_of\_food, geographical\_location, wastage\_food\_amount, etc.

**Initialisation de l'application Dash**

**server = Flask(\_\_name\_\_)**

**app = dash.Dash(\_\_name\_\_, server=server, url\_base\_pathname="/dash/", ...)**

**Fonction de simulation des métriques**

Fonction temporaire simulant les données dynamiques du modèle (Mesa à connecter plus tard).

|  |
| --- |
| Une image contenant texte, capture d’écran, Police  Le contenu généré par l’IA peut être incorrect. |

**Layout et tabs Dash**

3 onglets sont prévus avec le composant dcc.Tabs :

* **tab-metrics** : Graphiques en direct
* **tab-logs** : Logs simulés d'agents
* tab-predictions : Prévision du volume

**Callbacks principaux**

En fonction de l'onglet actif, le contenu est généré dynamiquement :

**1. Graphiques des métriques**

* **px.line** : évolution temporelle des livraisons
* **px.pie** : répartition des types de nourriture
* **px.density\_heatmap** : pénuries par région

**2. Logs Agents**

* Liste simulée d'événements d'agents

**3. Prédiction**

* Graphe de prévision de la demande future

**🛠️ Installation et lancement**

**1. Installer les dépendances**

**pip install dash pandas plotly flask**

**2. Structure du projet**

distribution-alimentaire/

├── Dashboard/

│ └── dash\_gui.py

├── data/

│ └── cleaned\_food\_wastage\_data.csv

**3. Lancer l'application**

**python Dashboard/dash\_gui.py**

Puis accédez à <http://localhost:8051/dash/> dans le navigateur.

# 🧪 Exécution en ligne de commande (CLI) – Documentation

Ce script permet de **lancer une simulation de distribution alimentaire directement depuis le terminal**, avec des paramètres personnalisés. Il est situé dans le fichier run\_simulation.py.

**🎯 Objectif**

Permettre :

* de tester rapidement la simulation sans interface graphique,
* de varier les paramètres du modèle dynamiquement,
* de générer un journal (log) des étapes de simulation.

**⚙️ Fonctionnalités du script**

**1. Analyse des arguments**

Utilisation de la librairie argparse pour injecter les paramètres :

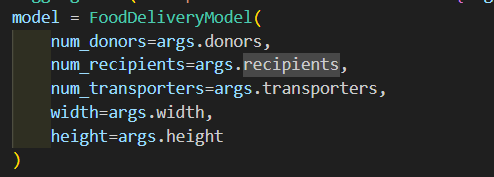
|  |
| --- |
| Une image contenant texte, capture d’écran, Police  Le contenu généré par l’IA peut être incorrect. |

Tu peux donc lancer une simulation personnalisée comme ceci :

**python Dashboard/run\_simulation.py --donors 15 --recipients 8 --steps 300**

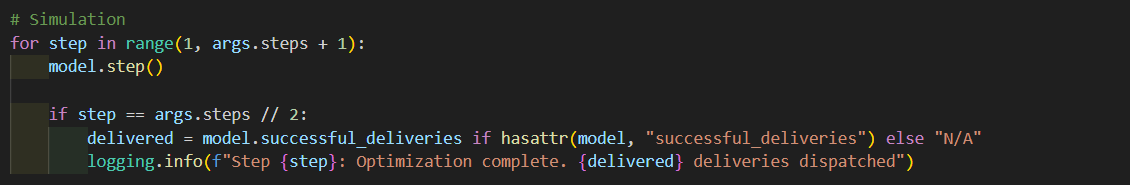
**2. Initialisation du modèle**

Création d’un objet FoodDeliveryModel avec les paramètres définis par l’utilisateur :



**3. Boucle de simulation**

Le modèle avance d’un pas à chaque itération :



**4. Journalisation (Logging)**

Tous les événements importants sont enregistrés dans un fichier logs/run-YYYY-MM-DD.log :

Une image contenant texte, capture d’écran, Police

Le contenu généré par l’IA peut être incorrect.

logging.info(f"Step 1: Initialized model with {args.donors} donors...")

logging.info(f"Step {step}: Optimization complete...")

logging.info(f"Step {args.steps}: Simulation ended.")

**🧾 Exemple d'utilisation**

**python Dashboard/run\_simulation.py --donors 20 --recipients 10 --transporters 5 --steps 500**

Résultat attendu :

✅ Simulation terminée avec succès. Voir les logs : logs/run-2025-04-24.log

Et dans le fichier de log :

|  |
| --- |
|  |
|  |
|  |