ChatGPT

Structure modulaire proposée (intégration SDK OpenAI Agents)

Avant de passer au code, voici un plan d'architecture en dossiers/modules séparés pour le système multi-agents Codex. L'idée est de distinguer clairement l'orchestrateur, le planificateur de tâches et le(s) codeur(s), en s'appuyant sur le SDK OpenAI Agents pour les implémenter. Chaque composant correspond à un agent ou à un groupe d'agents spécialisés, utilisant les modèles OpenAI appropriés.

```
codex_project/                  # Racine du projet
├── orchestrator/               # Module Orchestrateur (coordination
générale)
│   └── orchestrator_agent.py   # Définit l'agent orchestrateur ou la logique
de pilotage
├── planner/                    # Module Planification (Architecte/Manager de
tâches)
│   ├── task_planner_agent.py   # Agent de planification qui décompose les
demandes
│   └── embed_util.py           # Outil de recherche par embeddings (contexte
rapide)
├── coder/                      # Module Codeur(s) (génération de code)
│   └── code_generator_agent.py # Agent(s) codeur utilisant GPT-4.1 pour
produire le code
├── utils/                      # Utilitaires communs (configuration,
modèles, etc.)
│   ├── model_config.py         # Config des modèles (ex : choix de "o4-
mini", GPT-4.1...)
│   └── openai_client.py        # Initialisation du SDK OpenAI Agents (clés
API, Runner)
└── main.py                     # Script principal lançant l'orchestration
multi-agents
```

Orchestrateur (module orchestrator/)

- **Rôle :** Coordonne le workflow global. L'orchestrateur reçoit la requête initiale (par exemple la demande de l'utilisateur ou un **prompt** synthétisé) et décide de la séquence des étapes.
- **Implémentation :** Il peut être implémenté soit comme un agent directeur (un agent OpenAI doté d'instructions lui permettant de planifier et déléguer), soit tout simplement via du code Python orchestrant manuellement les agents. Pour plus de déterminisme, on optera ici pour une orchestration par le code (flow contrôlé manuellement) [1].
- **Fonctionnement :** Sur réception d'une tâche, l'orchestrateur appelle d'abord l'agent planificateur pour analyser la demande (classification ou découpage). Ensuite, il enchaîne les appels aux agents codeurs pour chaque sous-tâche identifiée, éventuellement en boucle ou en parallèle. Ce chaînage de plusieurs agents, où la sortie de l'un sert d'entrée au suivant, est une approche recommandée pour décomposer des tâches complexes [2].

- **OpenAI Agents SDK :** L'orchestrateur utilise le SDK pour exécuter les agents dans le bon ordre. Par exemple, il peut utiliser `Runner.run_sync(...)` sur l'agent planificateur puis sur l'agent codeur. On peut aussi configurer un agent orchestrateur qui délègue automatiquement certaines actions grâce aux **handoffs** du SDK [3]. (Le *handoff* permet à un agent de **déléguer** une tâche à un sous-agent spécialisé [3]). Par exemple, la documentation illustre un agent de triage qui redirige vers des agents spécialisés selon la question posée [4]. De même, notre orchestrateur pourrait, à terme, décider de remettre une tâche de codage à un agent codeur via un handoff automatique.

Planification des tâches (module planner/)

- **Rôle :** C'est le module *Architecte/Manager*, chargé de découper la requête en sous-tâches gérables et de déterminer comment les distribuer aux codeurs. Il sert de **cerveau rapide** du système, filtrant et préparant le travail pour le codeur.
- **Agent de planification :** On y définira un `Agent` OpenAI (par exemple `TaskPlannerAgent`) dont le but est d'analyser la demande initiale et de produire soit une **classification** de la tâche, soit une **liste de sous-tâches** structurées. Cet agent sera configuré avec un modèle **léger et rapide** – par exemple un modèle OpenAI de petite taille (« o4-mini » ou équivalent GPT-4 rapide) [5] – afin d'obtenir une inférence ultra-rapide. Son **prompt** (instructions système) décrira comment il doit formater le plan (éventuellement en JSON ou liste, facilitant le traitement automatique).
- **Utilisation d'embeddings :** Le planificateur peut s'appuyer sur un composant d'**embedding** pour gagner en efficacité. Par exemple, avant de décider du plan, il peut interroger une base de connaissances vectorielle pour extraire du contexte pertinent (code existant, documentation, exemples) lié à la requête. Le fichier `embed_util.py` contiendra les fonctions nécessaires pour encoder la requête et retrouver les informations associées. Ce mécanisme d'augmentation de contexte aide l'agent planificateur à prendre des décisions éclairées rapidement.
  Exemple : si l'utilisateur demande d'implémenter une fonctionnalité X, le planificateur pourrait rechercher par embeddings si une partie du code ou une librairie liée à X existe déjà, et utiliser ce contexte pour formuler les sous-tâches.
- **Déroulement :** Concrètement, l'orchestrateur va appeler le `TaskPlannerAgent` via le SDK (`Runner.run(...)`) en lui passant la requête utilisateur. La réponse de cet agent sera une structure de données (par ex. un dictionnaire ou une liste d'actions à mener) plutôt qu'un simple texte. L'utilisation d'un **output structuré** est d'ailleurs préconisée pour ce genre de coordination par code [6] : on peut demander à l'agent planificateur de **classer** la demande en catégories ou de renvoyer des étapes numérotées, ce qui facilitera la logique de branchement côté orchestrateur.
- **Filtrage / Guardrails :** Ce module peut aussi jouer le rôle de garde-fou en amont. Grâce à un modèle rapide, on peut filtrer ou analyser la requête utilisateur pour détecter d'éventuels problèmes (ex. demande hors-scope ou malveillante). Le SDK Agents propose les guardrails pour effectuer en parallèle des vérifications sur l'input avant de lancer les modèles coûteux [7]. Dans notre architecture, le planificateur (ou une fonction dédiée dans `planner/`) peut réaliser ce contrôle : si la requête est invalide ou non pertinente, il le signale immédiatement, évitant de mobiliser GPT-4 inutilement [7].

Génération de code (module coder/)

- **Rôle :** C'est le module des *Codeurs*, responsable de produire le code source requis pour chaque tâche ou sous-problème identifié. Il contient un ou plusieurs agents spécialisés dans la génération de code à partir d'instructions précises.

- **Agent codeur :** On y définira par exemple un `Agent` nommé `CodeGeneratorAgent` utilisant le modèle GPT-4.1 (ou équivalent) pour la **synthèse de code**. Ses instructions de base (prompt système) préciseront qu'il doit suivre fidèlement les spécifications fournies par le planificateur (par ex. *« Tu es un assistant codeur expert. Écris le code en suivant exactement les indications fournies, en respectant les conventions. »*). Étant donné que GPT-4 est plus lent/cher, cet agent n'est invoqué **qu'après** la phase de planification filtrante, avec des instructions bien ciblées – ce qui maximise la qualité du résultat tout en minimisant les appels.
- **Multi-codeurs spécialisés (optionnel) :** L'architecture est extensible pour inclure plusieurs agents codeurs spécialisés par domaine ou langage. Par exemple, on pourrait avoir `PythonCoderAgent`, `WebUICoderAgent`, etc., chacun ayant des connaissances/outils spécifiques. L'orchestrateur ou le planificateur déciderait alors de quel codeur a besoin chaque sous-tâche. Ce principe suit la recommandation d'utiliser **des agents spécialisés excellant chacun dans une tâche particulière plutôt qu'un agent généraliste à tout faire** [8]. Dans un premier temps, un seul agent codeur général peut suffire, mais nommer le module *coder/** laisse la place à en ajouter d'autres.
- **Résultat :** Pour chaque sous-tâche reçue (par exemple « Écrire la fonction X qui fait Y »), l'agent codeur génère le code source correspondant. L'orchestrateur collecte ces morceaux de code. Si nécessaire, une étape d'**évaluation** ou de test peut être ajoutée après chaque génération (possiblement via un autre agent critique ou en utilisant un outil de test automatisé).

Utilitaires et configuration (module utils/)

- **Config du SDK et API :** Ce dossier contient les éléments transverses. Par exemple, `openai_client.py` peut initialiser la configuration du SDK OpenAI Agents (clé API, paramètres globaux) pour que les autres modules l'importent. On peut y définir une fonction utilitaire pour obtenir un objet `Runner` ou pour centraliser l'appel à `Runner.run` avec les bons paramètres.
- **Configuration des modèles :** Le fichier `model_config.py` listera les identifiants de modèles et réglages à utiliser pour chaque agent. Par exemple, on peut y définir `MODEL_PLANNER = "o4-mini"` et `MODEL_CODER = "gpt-4.1"`, ainsi que les paramètres de température, top_p, etc., spécifiques à chaque rôle. Ainsi, les agents pourront être instanciés avec ces valeurs en évitant la duplication.
- **Outils réutilisables :** Outre l'outil d'embedding (qui pourrait aussi résider ici si on veut le partager entre modules), ce dossier peut accueillir des fonctions communes (par ex. formatage de sortie, évaluation de code généré, gestion de logs/traces). Le SDK offre un système de traçage intégré pour visualiser le déroulement des agents [9], dont l'initialisation pourrait être centralisée dans un utilitaire (ex: activer le mode tracing pour debug).

Intégration des Agents OpenAI (considérations spéciales)

- **Agents en tant qu'endpoints :** Chaque agent défini dans cette architecture (orchestrateur, planificateur, codeur) est essentiellement un *endpoint* qui fait appel aux modèles OpenAI en back-end. Autrement dit, quand un agent est exécuté, cela envoie une requête à l'API OpenAI (Chat Completion) avec le prompt construit selon ses instructions, outils et contexte. Cette conception "agent = endpoint" est importante : au début du projet, on utilise directement les agents du SDK d'OpenAI tels quels (appels API externes). Ils occuperont donc une place centrale dès le démarrage de l'application. Ce n'est que plus tard, une fois l'ensemble fonctionnel, qu'on envisagera éventuellement de **miroiter** certaines connexions (par exemple remplacer un appel externe par un équivalent local ou optimiser certaines étapes).

- **Définition des agents via le SDK :** Grâce au SDK, créer un agent se fait en quelques lignes. Chaque module peut exposer une fabrique ou une instance d'agent prête à l'emploi. Par exemple, dans `task_planner_agent.py`, on peut avoir :

```
from agents import Agent, function_tool
# ... éventuelle définition de tool @function_tool pour embeddings ...
planner_agent = Agent(
    name="TaskPlanner",
    instructions="Analyse la demande et crée un plan détaillé de sous-tâches
en JSON...",
    model=MODEL_PLANNER,  # par ex. "o4-mini"
    tools=[embedding_search_tool]  # si on utilise un outil de recherche
)
```

De même, `code_generator_agent.py` définira l'agent codeur :

```
coder_agent = Agent(
    name="CodeGenerator",
    instructions="Tu es un assistant qui génère du code selon les
instructions fournies...",
    model=MODEL_CODER,  # par ex. "gpt-4.1"
    # éventuellement pas de tools, ou des tools utilitaires si besoin
)
```

- **Enchaînement et délégation :** On peut soit orchestrer manuellement ces appels (via `Runner.run_sync(planner_agent, input)` puis boucle sur `Runner.run_sync(coder_agent, subtask)`), soit tirer parti des *handoffs*. Par exemple, on pourrait configurer l'agent planificateur pour qu'il **fasse un handoff** automatique vers le codeur une fois le plan établi. Le SDK supporte cela nativement : on peut fournir la liste des sous-agents codeurs à l'agent principal, et inclure dans son prompt des instructions du type *"si le plan contient du code à écrire, délègue à tel agent"*. Cette délégation automatisée est exactement le concept de **handoff** du SDK [3] et permet un enchaînement fluide sans écrire la boucle en code (c'est l'LLM qui décide quand appeler le codeur). Pour démarrer, toutefois, on peut garder un contrôle explicite via le code Python, ce qui facilite le débogage.

En résumé, cette architecture sépare les préoccupations en trois modules principaux (Orchestrateur, Planification, Codeurs) et s'appuie sur le SDK d'OpenAI pour créer des agents spécialisés qui collaborent. Ce squelette modulaire servira de base : une fois mis en place, on pourra envoyer les invites appropriées à Codex pour qu'il génère automatiquement le code de chaque composant de l'API. Chaque partie ayant un rôle bien défini, il sera plus simple de développer et tester progressivement l'ensemble du système. Les agents OpenAI agiront comme moteurs d'inférence à chaque étape (rapide pour la planification, puissant pour le codage), conformément au workflow souhaité [2] [10]. Ainsi outillée, l'équipe pourra par la suite étendre ou optimiser chaque module (y compris réécrire certaines parties en Rust ultérieurement, si nécessaire, une fois la logique validée en Python).

---

[1] [2] [3] [6] [8] [10] Orchestrating multiple agents - OpenAI Agents SDK
https://openai.github.io/openai-agents-python/multi_agent/

4 5 Agents - OpenAI Agents SDK
https://openai.github.io/openai-agents-python/agents/

7 Guardrails - OpenAI Agents SDK
https://openai.github.io/openai-agents-python/guardrails/

9 OpenAI Agents SDK
https://openai.github.io/openai-agents-python/