ChatGPT

Spécification détaillée – Agent IA de Messagerie (Email AI Agent)

1. Contexte et Objectif

L'**Agent IA de messagerie** a pour objectif d'automatiser la gestion des emails entrants sur une boîte Outlook (Office 365) en utilisant des techniques d'intelligence artificielle. Cet agent doit pouvoir **récupérer les nouveaux emails**, les **analyser** (par exemple grâce à un modèle IA type GPT-4) et éventuellement **effectuer des actions** en réponse : générer une réponse automatique, extraire des informations clés, créer des tickets dans un système interne, etc. L'agent fonctionnera en continu sur un serveur, intégré à l'infrastructure existante via Docker, et sera accessible via un sous-domaine dédié (`agentmail.padaw.ovh`).

Les bénéfices attendus sont : - Un traitement plus rapide des emails entrants (réponses automatisées ou aide à la décision). - Une diminution du travail manuel de tri ou d'analyse des courriels. - Une intégration fluide avec les services existants (via API internes, base de données, etc.) en fonction du contenu des emails.

2. Architecture Générale de la Solution

Composants principaux : - **Application Azure (Microsoft Entra ID)** : Une application enregistrée Azure AD fournit les identifiants OAuth2 nécessaires pour accéder à l'API Microsoft Graph. Elle dispose des permissions requises sur la boîte mail (lecture, envoi). - **Service Python (Agent)** : Application Python qui tourne en tâche de fond. Elle utilise l'API Microsoft Graph pour se connecter à la messagerie Office 365, soit en mode *pull* (polling régulier) soit via *webhooks* (notifications push). Ce service intègre également un module IA (ex. appel à une API OpenAI ou modèle local) pour analyser le contenu des emails. - **Interface Web (endpoint)** : Le service expose une interface HTTP REST (via un micro-framework comme Flask ou FastAPI) permettant de recevoir les notifications de nouveaux emails (dans le cas de l'usage de webhooks Microsoft Graph). Ce endpoint sera accessible publiquement à l'URL `https://agentmail.padaw.ovh/...`. Il ne s'agit pas d'une interface utilisateur, mais d'un point d'entrée machine (webhook) pour Microsoft Graph et éventuellement pour d'autres services internes. - **Infrastructure Docker et Nginx** : L'agent sera containerisé. Le container Docker rejoindra le réseau Docker existant (`nginx-proxy`) afin de bénéficier du reverse-proxy Nginx déjà en place. Le sous-domaine `agentmail.padaw.ovh` sera configuré pour pointer vers ce service (via les variables d'environnement du proxy automatique *autoconf*). Nginx se chargera de la terminaison TLS (génération automatique de certificat Let's Encrypt) et du routage HTTP vers le container de l'agent.

Schéma d'interaction (vue d'ensemble) : 1. Un nouvel email arrive sur la boîte surveillée. 2. L'agent en prend connaissance (soit en le récupérant via l'API Graph, soit via réception d'une notification push Graph). 3. L'agent extrait le contenu pertinent (expéditeur, sujet, corps du message, pièces jointes éventuellement). 4. Le module IA de l'agent traite ces informations : - Par exemple, il peut résumer le message, détecter l'intention (demande de rendez-vous, question fréquente, commande, etc.), ou extraire des données structurées (noms, dates, numéros). - Si l'email contient une instruction destinée à un système (par ex. « créer un nouvel utilisateur » ou « générer un rapport »), l'agent peut formuler une requête correspondante à un API interne (par ex. appeler le backend `optigo` approprié). 5. En

fonction du résultat de l'analyse, l'agent peut prendre une action : - Générer automatiquement une réponse par email (et l'envoyer via l'API Graph) si c'est pertinent ou fournir un brouillon de réponse assisté par IA. - Marquer l'email comme traité (lu) ou le déplacer dans un dossier spécifique pour archivage. - Enregistrer un log ou un résumé de l'email dans une base de données interne, ou créer un ticket de support. 6. L'agent répète ce cycle pour chaque nouveau message entrant en restant actif en permanence.

3. Pré-requis et Configuration Initiale

3.1 Application Azure AD et Permissions

- **Application Azure** : Une application doit être créée dans Azure AD (Entra ID) pour permettre à l'agent d'accéder à l'API Microsoft Graph. Les informations d'authentification suivantes sont nécessaires :
- *Client ID* (identifiant de l'application) [1]
- *Client Secret* (secret d'application généré) [2]
- *Tenant ID* (ID du tenant Azure AD où est enregistrée l'app) [3]
- *Redirect URI* : (dans le cas d'une auth déléguée interactive – probablement inutile ici car on utilisera un flux *credentials* côté serveur).
- **Permissions API** : L'application Azure doit avoir les droits **Microsoft Graph** appropriés consentis :
- Au minimum, la permission **Mail.Read** (ou Mail.ReadBasic) en type application pour pouvoir lire les emails d'un utilisateur [4]. Pour lire le contenu complet (sujet, corps), **Mail.Read** est recommandé plutôt que Mail.ReadBasic.
- Si l'agent doit pouvoir envoyer des emails, ajouter la permission **Mail.Send**.
- Éventuellement **Mail.ReadWrite** si on prévoit de marquer les messages comme lus/déplacés via l'API.
- **User.Read** (basique) peut être nécessaire en délégation, mais dans un contexte *daemon* (application serveur), on utilisera plutôt les permissions applicatives (application permissions) avec consentement administrateur.
- **Consentement** : Une fois les permissions assignées, un administrateur du tenant doit consentir à ces permissions pour que l'app soit autorisée à accéder aux boîtes aux lettres en question.

3.2 Compte de Messagerie Cible

- **Boîte aux lettres surveillée** : Décider quelle boîte mail l'agent va surveiller. Deux cas possibles :
- **Boîte dédiée (compte de service)** : Par exemple `agent@padaw.ovh` sur Office 365, sur laquelle on redirige ou copie les emails à traiter. L'agent utilisera l'API Graph sur ce compte. Avantage : évite d'interférer avec une boîte personnelle principale.
- **Boîte utilisateur existante** : l'agent se connecte sur la boîte d'un utilisateur réel (ex: votre boîte de réception). Dans ce cas, si on utilise les permissions *application* (client credentials), l'app Azure doit avoir accès à *toutes* les boîtes ou à la boîte spécifique. On peut limiter en scope en n'utilisant que l'ID de l'utilisateur cible dans les requêtes Graph.
- **Dossiers surveillés** : Par défaut, l'agent surveille la **Boîte de réception** (Inbox). On peut étendre ou personnaliser (ex: surveiller un dossier «AI Agent» particulier, ou surveiller tous les mails entrants y compris spams – moins probable). Pour la spécification, on considérera Inbox.
- Assurez-vous que l'**authentification moderne** (OAuth) est activée et l'**IMAP/POP désactivé** si on choisit Graph (ce qui est le cas par défaut sur O365). Ici on utilise Graph API plutôt qu'IMAP pour robustesse et sécurité.

3.3 Infrastructure Docker et Nom de Domaine

- **Sous-domaine DNS** : Configurer le DNS pour que `agentmail.padaw.ovh` pointe vers l'adresse IP du serveur hébergeant l'agent (même IP que padaw.ovh existant). Ceci permettra à Nginx de router le trafic HTTP(S) correctement.
- **Nginx Proxy automatique** : Le serveur utilise un **reverse proxy Nginx** avec configuration automatique (par ex. basé sur l'image `jwilder/nginx-proxy` et un companion Let's Encrypt). On le voit dans la configuration du réseau `nginx-proxy` fournie.
- Le container de l'agent doit rejoindre ce réseau Docker (`nginx-proxy`) afin d'être accessible via le proxy.
- Utiliser des variables d'environnement dans le container de l'agent pour que le proxy le reconnaisse :
  - `VIRTUAL_HOST=agentmail.padaw.ovh` (indique le host virtuel à créer).
  - `LETSENCRYPT_HOST=agentmail.padaw.ovh` et `LETSENCRYPT_EMAIL=votre_email@domaine` (pour générer un certificat SSL valide automatiquement).
  - (Si le setup autoconf utilise d'autres variables spécifiques, les adapter en conséquence).
- Le port interne sur lequel écoute le service web Python doit être exposé au proxy. Par exemple, si l'application Flask écoute sur le port 8000 à l'intérieur du container, on s'assurera de publier ce port et d'indiquer au proxy quel port utiliser (`VIRTUAL_PORT=8000` si nécessaire). Par défaut, certains proxies détectent le port exposé automatiquement.
- **Réseau Docker** : D'après l'inspection, le réseau `nginx-proxy` existe déjà avec des containers (nginx-proxy, optigo-frontend, backend, etc.). Le Docker-compose ou commande de run du container agent devra inclure `--network nginx-proxy` pour y adhérer.
- **Nom du container** : Choisir un nom unique, par ex. `agent-mail` pour le container, afin de faciliter les logs et le diagnostic (on le verra dans `docker ps` et il ne doit pas entrer en conflit avec un autre nom).
- **Backup config nginx** : S'assurer que la configuration automatique de nginx ne rentre pas en conflit avec `padaw.ovh` déjà en place. Ici, on utilise un sous-domaine donc cela devrait s'ajouter sans conflit.

3.4 Environnement Logiciel du Container

- **Langage et runtime** : Python 3.x (idéalement la dernière version stable). On pourra partir d'une image de base officielle (ex : `python:3.10-slim`).
- **Bibliothèques nécessaires** :
- **MSAL** (Microsoft Authentication Library) pour gérer l'authentification OAuth2 avec Azure AD facilement [5].
- Une librairie HTTP pour appeler l'API Graph. Le choix simple est d'utiliser `requests` (ou bien le SDK officiel Microsoft Graph Python, mais ce n'est pas strictement nécessaire).
- Un framework web léger pour le webhook : **Flask** ou **FastAPI**. FastAPI pourrait être un bon choix pour performance et simplicité (plus facile pour définir plusieurs routes). Flask est aussi suffisant.
- Bibliothèque pour l'IA :
  - Si on utilise l'API OpenAI : installer `openai` (le SDK Python) et prévoir la variable d'API Key.
  - Si on utilise un modèle local ou HuggingFace, inclure les dépendances correspondantes.
- Autres utilitaires : `python-dotenv` (pour gérer config), `schedule` (si on utilise un polling périodique), etc.
- **Variables d'environnement à définir** (dans le Docker ou via compose secrets) :
- `AZURE_CLIENT_ID` : l'ID client de l'app Azure.

- `AZURE_TENANT_ID` : le tenant ID Azure.
- `AZURE_CLIENT_SECRET` : le secret d'application.
- `MAILBOX_USER_ID` : l'identifiant de la boîte mail cible. Peut être l'UPN (email) de l'utilisateur, par ex. `agent@padaw.ovh`, ou son GUID Azure AD. (Utile pour construire les requêtes Graph).
- (Si on utilise Graph en mode *me*, on pourrait stocker l'UPN dans l'app and utiliser /me, mais en mode application *me* n'est pas valide, il faut spécifier l'user).
- `OPENAI_API_KEY` : si on utilise OpenAI pour l'IA.
- `OPENAI_MODEL` : nom du modèle (par ex. `gpt-4` ou `gpt-3.5-turbo`).
- Variables de configuration du webhook :
  - `WEBHOOK_HOST` ou le domaine pour vérification interne (optionnel, on sait que c'est agentmail.padaw.ovh).
  - Une `WEBHOOK_SECRET` ou code secret pour vérifier les notifications (par ex. la valeur de clientState, voir section suivante).
- Variables pour le proxy Nginx (décrites plus haut): `VIRTUAL_HOST`, etc.
- **Repository code** : (si pertinent) stocker le code sur Git, config CI etc, mais hors du scope immédiat.

4. Fonctionnement de l'Agent

4.1 Authentification et Connexion à Microsoft Graph

L'agent utilise OAuth2 via la librairie MSAL pour obtenir un jeton d'accès à l'API Graph: - **Mode d'authentification** : On privilégie le **flux Client Credentials** (identité d'application) puisque l'agent tourne sans intervention utilisateur. - Initialiser une instance de `ConfidentialClientApplication` avec le tenant, clientId et secret [6] [7]. - Demander un token pour la portée souhaitée. Avec client credentials, on utilise la portée `https://graph.microsoft.com/.default` qui utilise les permissions applicatives accordées.
- MSAL fournit `acquire_token_for_client()` pour obtenir un token en utilisant le client_id et secret. Le token émis aura les permissions définies (Mail.Read etc.). - **Stockage du token** : Le jeton access_token obtenu est valable ~1 heure. L'agent doit pouvoir en obtenir un nouveau lors de l'expiration. MSAL gère un cache en mémoire par défaut. On peut utiliser ce cache ou simplement redemander un token à chaque opération (MSAL évite les appels inutiles si le token est encore valide). - **Gestion des erreurs d'auth** : Prévoir le cas où l'obtention du token échoue (mauvaise config, secret expiré, absence de consentement). Dans ce cas, logguer l'erreur et éventuellement tenter une nouvelle tentative après un délai. L'agent ne pourra pas fonctionner tant que le token n'est pas obtenu.

4.2 Surveillance des Nouveaux Emails – Polling vs Webhooks

Deux approches sont envisageables pour détecter les nouveaux emails :

A. Polling périodique : - L'agent interroge régulièrement l'API Graph pour lister les messages non lus de la boîte de réception.
- Exemple d'appel : `GET /users/{MAILBOX_USER_ID}/mailFolders/Inbox/messages?$filter=isRead%20eq%20false` pour récupérer uniquement les mails non lus. On peut sélectionner certains champs (expéditeur, sujet, reçu le, etc.) pour limiter la charge. - L'intervalle peut être ajusté (par ex. toutes les 1 minute, 5 minutes). Plus c'est fréquent, plus l'agent réagit vite, mais attention aux limites de l'API et au quota (Graph a des limites de taux). - À chaque cycle, comparer la liste des ID de messages non lus avec ceux déjà vus auparavant pour déterminer les nouveaux. - Avantage: simplicité d'implémentation. Inconvénient: délai potentiellement plus long et inefficace si très fréquent (appel constant même sans nouveaux mails). - **Marquage comme lu** : Après traitement, l'agent peut appeler

`PATCH /messages/{id}` pour mettre `isRead=true` (ou utiliser l'action `/messages/{id}/move` vers un dossier "Processed") afin de ne plus le traiter lors du prochain poll. - **Cas d'usage** : Le polling peut être utile en première version pour valider le fonctionnement de base, ou si les webhooks ne sont pas fiables dans un contexte donné.

B. Webhooks (Notifications Graph push) :
Microsoft Graph peut appeler notre service dès qu'un nouvel email arrive, évitant le polling [8]. Voici comment mettre en place : - **Endpoint webhook** : Dans l'agent, implémenter une route HTTP (exemple : `POST /graph/notifications`) qui recevra les notifications de Graph. - Lors de l'enregistrement, Microsoft Graph enverra d'abord une requête de validation **GET** sur cette URL, contenant un jeton de validation en paramètre (`validationToken`). Notre agent doit répondre à cette requête en renvoyant textuellement le token dans les 10 secondes pour confirmer la réception [9]. Si c'est correct, le webhook est enregistré. - Important : Le service web de l'agent doit donc pouvoir traiter un **GET /graph/notifications?validationToken=XYZ** et renvoyer `XYZ` en texte brut. (Exemple d'implémentation avec FastAPI: un endpoint qui retourne `PlainTextResponse(token)`). - **Abonnement (subscription)** : L'agent utilise l'API Graph pour créer un abonnement: - Requête `POST /subscriptions` vers Graph, avec un corps JSON contenant les paramètres : - `"resource": "/users/{MAILBOX_USER_ID}/mailFolders('Inbox')/messages"` : la ressource à surveiller (ici les nouveaux mails dans la boîte de réception de l'utilisateur cible). - `"changeType": "created"` : on veut être notifié uniquement des créations de nouveaux messages (optionnellement on pourrait inclure `"updated, deleted"` mais ici le besoin principal est nouveaux mails). - `"notificationUrl": "https://agentmail.padaw.ovh/graph/notifications"` : l'URL publique de notre webhook (doit être HTTPS et accessible). - `"clientState": "SOME_SECRET_VALUE"` : une valeur secrète aléatoire de notre choix. Graph renverra cette même valeur dans chaque notification, ce qui nous permettra de valider que la notification reçue est bien authentique et liée à notre souscription [10]. - `"expirationDateTime": "<DateTime ISO>"` : date/heure d'expiration de l'abonnement. Par défaut Graph limite à 3 jours (~4320 minutes) maximum pour les mails [11]. On met la valeur max autorisée initialement. - Éventuellement d'autres champs comme `"scope": "..."` selon besoin, mais pour un abonnement de type application sur un user spécifique, les permissions de l'app suffisent. - **Réponse** : Si la création est réussie, Graph retourne un objet `subscription` contenant un `id` d'abonnement, la resource, la expirationDateTime réelle, etc. - **Renouvellement** : L'agent doit anticiper le renouvellement, car l'abonnement expire au bout de 3 jours maximum [11]. Stratégie: - Stocker l'`id` de l'abonnement et son `expirationDateTime`. - A intervalles réguliers (ex: via un thread ou une tâche planifiée chaque heure ou déclenchée lors de notifications), vérifier le temps restant. Par exemple, dès qu'il reste moins d'un jour avant expiration, renvoyer une requête `PATCH /subscriptions/{id}` avec un nouveau `"expirationDateTime"` dans ~3 jours pour prolonger. - En cas d'échec du renouvellement (ex: token expiré, app permissions révoquées, etc.), logguer et éventuellement recréer un nouvel abonnement de zéro si nécessaire. - **Réception des notifications** : Lorsque Graph envoie une notification (HTTP POST): - Le corps du POST contiendra les informations sur l'élément changé. Par défaut, Graph envoie un petit objet avec l'`id` du message, le type de changement (created/updated) et éventuellement le `resourceURL`. **Note** : Par défaut, le contenu du mail n'est pas inclus pour des raisons de sécurité/performance. On doit alors appeler Graph pour récupérer le mail complet. - Exemple de payload (simplifié) :

```
{
  "value": [{
     "subscriptionId": "<ID souscription>",
     "changeType": "created",
     "resource": "users/<userid>/messages/<messageid>",
     "tenantId": "<tenantID>",
```

```
      "clientState": "SOME_SECRET_VALUE"
    }]
}
```

On vérifiera que `clientState` correspond bien à la valeur attendue pour éviter les faux appels [10] . - Une fois la notification reçue et validée, l'agent extrait l'`id` du message (dans `resource` ou un champ `resourceData` si configuré) et utilise l'API Graph pour récupérer les détails complets du message: - Appel Graph pour récupérer le message : `GET /users/{userId}/messages/{messageId}` (peut inclure des paramètres `$select` pour champs spécifiques, ou `$expand=attachments` si on souhaite récupérer des pièces jointes également). - Graph répond avec l'objet du message (sujet, corps, sender, etc). - **Remarque** : Il est possible de demander à Graph d'inclure directement les données du message dans la notification en paramétrant `includeResourceData: true` lors de l'abonnement, mais cela requiert de gérer un certificat pour déchiffrer les données (les données seraient envoyées chiffrées par Graph) [12] [13] . Pour simplifier, on peut ignorer cette option dans un premier temps et simplement faire une requête pour obtenir le mail lors de chaque notification. - **Fiabilité** : Implémenter la gestion des cas où l'agent pourrait manquer des notifications (ex: downtime). Microsoft envoie des *lifecycle notifications* si l'abonnement est supprimé ou nécessite une reauthorization, que l'agent doit gérer. De plus, on peut coupler avec un mécanisme de rattrapage : à chaque démarrage de l'agent, faire un check manuel des mails non lus récents au cas où un événement aurait été manqué pendant une interruption.

Choix recommandé : Utiliser les **webhooks Graph** pour quasi-temps réel, avec un mécanisme de renouvellement d'abonnement. Le polling peut servir de secours ou de solution de rechange en cas de difficulté de mise en place des webhooks.

4.3 Traitement IA des Emails

Une fois qu'un nouvel email est récupéré (via polling ou via webhook): - L'agent extrait les éléments clés : **expéditeur**, **destinataires**, **objet**, **corps du message**, et liste des **pièces jointes** le cas échéant. - Ces informations sont passées au module d'**IA** pour analyse. Les traitements IA possibles incluent par exemple : - **Analyse de l'intention** : déterminer le but de l'email. Par exemple catégoriser s'il s'agit d'une demande d'information, d'un rapport d'incident, d'une commande, d'un spam, etc. - **Extraction d'entités** : extraire des données structurées du texte (noms, dates, numéros de commande, adresses...). - **Résumé automatique** : générer un résumé concis de l'email, utile pour un compte-rendu ou pour un affichage rapide. - **Classification de priorité ou sentiment** : évaluer l'urgence ou le ton (positif/négatif) du message afin de prioriser un traitement humain si nécessaire. - **Réponse suggérée** : proposer une ébauche de réponse appropriée à l'email. Par exemple, si l'email demande les horaires d'ouverture, l'agent peut préparer une réponse contenant ces informations. - Pour réaliser cela, on peut utiliser une **API de modèle de langage** (LLM) : - Envoi du prompt approprié au modèle (par ex. "Voici un email : ... Résume-le en 3 phrases." ou bien "Détermine si cet email concerne une demande technique, commerciale ou autre : ..."). - Le choix du modèle (GPT-3.5, GPT-4 via OpenAI, ou un modèle open-source hébergé) dépend des contraintes. OpenAI GPT-4 offrirait la meilleure qualité pour compréhension du langage. - Veiller à inclure dans le prompt ou les instructions de l'IA le contexte nécessaire (par ex. style de réponse professionnel, langue de l'email si ce n'est pas toujours en français, etc.). - **Exploitation du résultat IA** : Le résultat du modèle est ensuite interprété par l'agent pour décider de la suite : - Si c'est un résumé, l'agent peut l'enregistrer ou l'envoyer à un système (ou l'ajouter en haut du mail pour un collaborateur). - Si c'est une intention catégorisée, l'agent peut router l'email ou créer un ticket dans le système correspondant (ex: incident IT -> créer un ticket Jira, demande commerciale -> notifier l'équipe Sales, etc.). - Si c'est une réponse suggérée, l'agent peut soit l'envoyer automatiquement (dans les cas routiniers approuvés) soit la soumettre à une **validation humaine**. Une approche prudente est

d'envoyer la réponse en **brouillon** à l'expéditeur: c'est-à-dire via l'API Graph, créer un message de réponse et le sauvegarder dans les brouillons du compte, ou utiliser la fonction d'envoi différé si existante. Ainsi un humain peut vérifier. - **Exemple** : Email: "Bonjour, je n'arrive pas à me connecter à mon compte." -> L'IA classifie en « support technique », extrait "problème de connexion compte", l'agent pourrait automatiquement répondre avec un message type ("Cher utilisateur, avez-vous essayé de réinitialiser votre mot de passe…") ou créer un ticket support et répondre que le problème est pris en charge. - **Module IA modulaire** : Il est conseillé d'implémenter le traitement IA de façon modulaire/configurable. Par exemple, une fonction `process_email(content)` qui retourne un objet avec les champs souhaités (intention, résumé, réponse proposée, etc.). Ainsi on peut ajuster la logique sans toucher au reste (par exemple changer de modèle ou affiner le prompt).

4.4 Actions consécutives et Réponses

Après l'analyse IA, l'agent effectue les actions appropriées : - **Marquer comme traité** : L'email d'origine peut être marqué comme **lu** et éventuellement déplacé dans un dossier "Traité" pour archivage. Cela évite de le reprocesser. (API Graph: `PATCH /messages/{id}` pour set `isRead=true` ou `move`). - **Envoi d'un email de réponse** (facultatif selon les cas) : - Si une réponse automatique a été jugée pertinente, l'agent utilise l'API Graph pour envoyer le mail. L'API Graph offre l'endroit `/users/{id}/sendMail`. On doit construire un objet message avec les champs `subject`, `body`, `toRecipients` etc., puis appeler `POST /sendMail` [14] [15] (documentation Graph). Il faut la permission Mail.Send. - L'expéditeur sera le compte au nom duquel l'agent se connecte (si c'est un compte de service dédié, la réponse viendra de ce compte). **Attention** à bien définir le contenu du mail (texte formaté en HTML ou texte brut) selon le besoin. - Logguer l'envoi et idéalement attacher l'ID du message envoyé (Graph renvoie généralement 202 Accepted sans body pour sendMail). - **Intégration avec systèmes internes** : Si l'agent doit appeler l'application interne (par exemple `optigo-backend`) suite au mail, il peut le faire via HTTP (puisqu'il est sur le même réseau Docker, il pourrait appeler `http://optigo-backend:port/api` directement). Il conviendra de sécuriser cet appel (jeton d'API interne, etc.). - Exemples d'actions internes : Créer un nouvel objet (utilisateur, commande) dans la base, déclencher un traitement, enregistrer un événement. - **Stockage des résultats** : Pour suivi, l'agent peut écrire dans une base de données ou un fichier log les informations sur chaque email traité et l'action entreprise. Par exemple : Email X de A->B, sujet Y, classé en "commande", ticket #123 créé, réponse envoyée oui/non. Ceci permet un audit a posteriori et le débogage. - **Notifications** : En complément, l'agent pourrait notifier une personne ou un canal (Teams, Slack) lorsqu'il traite un email important, afin de garder un humain "dans la boucle" lors des débuts de l'automatisation.

4.5 Gestion des Erreurs et Scénarios Exceptionnels

- **Échecs de récupération mail** : Si l'API Graph renvoie une erreur lors de la lecture d'un mail (ex: mail déjà supprimé, perte de droits), l'agent loggue l'erreur. En mode webhook, c'est possiblement que le mail a été déplacé ou supprimé rapidement. L'agent pourra ignorer cette notification après log. En mode polling, une erreur peut être réessayée au prochain cycle.
- **Échecs du module IA** : Si l'IA ne retourne pas de résultat exploitable (par ex. appel à l'API OpenAI échoue ou dépasse le temps), définir une politique :
- Réessayer une fois éventuellement.
- Ou passer le mail en « non traité automatique » et éventuellement notifier un humain que cet email nécessite une attention (puisque l'IA n'a pas pu).
- **Timeouts** : Veiller à ce que les appels externes (Graph, OpenAI) aient des timeouts raisonnables pour ne pas geler le traitement. Utiliser éventuellement des appels asynchrones ou multi-thread si on veut traiter plusieurs emails simultanément (en cas de rafale d'emails).
- **Duplication** : Éviter de traiter deux fois le même mail. Avec webhooks, on peut recevoir deux notifications (par ex. *created* puis *updated* sur le même mail). Il faut garder trace des IDs déjà

traités récemment. Une structure en mémoire (set) ou stockage persistant simple (fichier des derniers IDs) peut suffire.
- **Maintien de l'abonnement** :
- Si le renouvellement du webhook échoue (ex: le token n'avait plus le droit Mail.Read, etc.), l'agent devrait détecter l'erreur (réponse 401 ou 403 de Graph) et tenter de recréer un abonnement neuf après avoir éventuellement régénéré un token ou rafraîchi les permissions. Il pourrait y avoir un court laps sans notifications.
- Microsoft Graph peut aussi envoyer une notification `reauthorizationRequired` en cas de problème (changement de mot de passe admin, etc.) [16]. L'agent doit logguer un message clair dans ce cas et possiblement alerter (car une action manuelle peut être requise pour réautoriser l'application).
- **Arrêts / redémarrages** : Si l'agent redémarre (déploiement, crash), il doit :
- Recréer un abonnement webhook actif (les anciens abonnements ne seront plus valides si l'URL change ou si on a dépassé l'expiration pendant l'arrêt).
- Traiter les mails non lus qui seraient arrivés pendant son absence (d'où l'intérêt d'un check initial au démarrage via un appel Graph filtré sur isRead=false par exemple).
- Avoir une stratégie pour éviter de renvoyer des réponses en double si déjà envoyées précédemment (on peut stocker l'ID des mails auxquels on a répondu dans un stockage persistant pour cross-check au démarrage).

5. Déploiement Docker de l'Agent

5.1 Dockerfile (aperçu)

Le Dockerfile contiendra par exemple :

```
FROM python:3.11-slim

# Installation des dépendances
RUN pip install msal requests fastapi uvicorn[standard] openai

# Copie du code de l'agent
COPY . /app
WORKDIR /app

# Exposer le port (exemple 8000)
EXPOSE 8000

# Démarrer l'application (exemple avec Uvicorn server si FastAPI)
CMD ["uvicorn", "agentmail:app", "--host", "0.0.0.0", "--port", "8000"]
```

(Remarque: le nom du module/app et détails seront adaptés selon l'implémentation effective.)

Points à noter : - Utiliser une image `slim` ou alpine pour réduire la taille, mais attention aux dépendances (ex: `msal` pure Python, pas de souci, `uvicorn[standard]` apportera uvloop, etc.). - Inclure éventuellement l'installation de drivers SSL si l'image de base ne les a pas (souvent déjà présents). - Ne pas copier de fichiers de config sensibles dans l'image (les secrets resteront dans les variables d'environnement au runtime, pas *hardcodés*).

5.2 Configuration du Container

Lors du lancement du container (via `docker run` ou compose): - Joindre le réseau existant : `--network nginx-proxy` (ou dans docker-compose, sous `networks:` utiliser `nginx-proxy:`). - Définir les variables d'env décrites en 3.4 (`AZURE_CLIENT_ID`, etc.). En docker-compose, on peut les définir dans un fichier `.env` ou directement dans le yaml sous le service. - Définir les variables pour nginx proxy: - `VIRTUAL_HOST=agentmail.padaw.ovh` - `VIRTUAL_PORT=8000` (si le service écoute sur 8000, sinon adapter) - `LETSENCRYPT_HOST=agentmail.padaw.ovh` - `LETSENCRYPT_EMAIL=contact@padaw.ovh` (par exemple, l'adresse email du propriétaire pour les notifications SSL). - (Optionnel) Montages de volumes: - Si l'agent écrit des logs dans un fichier ou stocke un cache DB (sqlite, etc.), monter un volume vers le host pour persistance. - Sinon logs peuvent être juste stdout/err et gérés par `docker logs`. - **Redémarrage** : Configurer une politique de redémarrage automatique (`restart: unless-stopped` par ex. en compose) pour que l'agent reprenne après un crash ou un reboot de la machine. - **Scale** : Normalement un seul container instance suffit (pas de scaling multi-instance car on surveille une source unique et on veut éviter les traitements doublons; de plus un abonnement Graph redirigerait toutes les notifs à tous les containers identiques, compliquant la concurrence).

5.3 Intégration Nginx Proxy

- Le container `nginx-proxy` existant va détecter le nouveau container grâce aux variables d'env et générer la config. On pourra vérifier que `agentmail.padaw.ovh` est bien accessible en HTTP(s).
- **Certification SSL** : Le companion Let's Encrypt (s'il est configuré) devrait générer un certificat pour le domaine. Il faut s'assurer que le port 80/443 du serveur est ouvert et que le DNS du domaine est bien propagé. Le premier démarrage peut prendre quelques minutes pour obtenir le cert.
- **Test d'accès** : On peut ajouter dans l'agent un endpoint simple de santé (ex: GET `/health` retournant "OK") pour tester via `curl https://agentmail.padaw.ovh/health`. Ceci permet de valider que le routage fonctionne à travers le proxy.
- **Sécurité** : Le proxy offre TLS. Vérifier que l'agent n'écoute qu'en interne (0.0.0.0 sur le container, exposé via proxy mais pas exposé en host mode directement). Ainsi, seules les requêtes via le domaine passeront.

6. Sécurité, Permissions et Confidentialité

Étant donné la nature sensible des emails, plusieurs aspects de sécurité doivent être couverts :

- **Stockage des Secrets** : Ne jamais stocker en clair le client_secret Azure ou l'API key OpenAI dans le code source. Utiliser les env variables comme indiqué, et restreindre leur visibilité. Si possible, intégrer avec un service de secrets (Azure Key Vault, Docker secrets) pour éviter même la présence en variable d'env en clair sur la machine.
- **Permissions minimales** : L'application Azure n'a que les droits nécessaires (principe du moindre privilège). Par exemple, Mail.Read sur une boîte utilisateur plutôt que sur *toutes* les boîtes de l'organisation si possible [17]. Si on peut limiter à un seul utilisateur via des configurations d'application (Application Access Policy dans Exchange Online pour restreindre l'app à une mailbox spécifique), ce serait idéal.
- **Client State Validation** : Comme mentionné, utiliser `clientState` dans l'abonnement Graph et vérifier sa présence dans chaque notification [10]. Cela évite qu'un tiers malveillant appelle

notre webhook avec de fausses données. Si le `clientState` ne correspond pas, on ignore la notification.

- **Signature des webhooks** : Microsoft Graph n'envoie pas de signature HMAC par défaut (d'où l'usage de clientState). Pour une sécurité accrue, on pourrait héberger le webhook derrière une validation supplémentaire, par ex. un token d'API dans l'URL que seul Graph (et nous) connaissons. Mais ce n'est pas natif; le clientState suffit comme secret partagé.
- **Traffic sortant** : Le container va appeler l'API Graph et possiblement l'API OpenAI sur Internet. S'assurer que ces communications utilisent bien TLS (HTTPS) – ce qui est le cas par défaut avec Graph et OpenAI endpoints.
- **Pare-feu** : Limiter l'accès au endpoint webhook aux IP de Microsoft Graph pourrait renforcer la sécurité, mais les adresses IP de Graph ne sont pas fixes et peuvent provenir de multiples datacenters, ce qui rend cette approche difficile. On se repose donc sur le secret et TLS.
- **Logs sensibles** : Si on logge le contenu des emails ou réponses AI, faire attention à où sont stockés ces logs. Éviter de stocker du contenu trop sensible en clair sur disque sur le long terme. Éventuellement nettoyer ou chiffrer si nécessité de conservation.
- **RGPD** : Si des données personnelles sont traitées dans les emails, veiller à la conformité. L'agent ne doit pas exposer ces données ni les envoyer à un service non autorisé. Par exemple, si envoi au modèle OpenAI, on doit être conscient que le contenu du mail est transmis à une API tierce – vérifier les termes d'utilisation et la confidentialité.

7. Tests et Validation

Avant la mise en production complète, effectuer une batterie de tests :

- **Test d'authentification** : Lancer l'agent et vérifier qu'il obtient bien un token d'accès Graph (logguer le succès, sans forcément afficher le token, mais l'expiration ou scopes pour confirmer). Tester un appel Graph simple (par ex. liste des dossiers ou des mails) pour valider la connectivité.
- **Test du Polling (si implémenté)** : Envoyer un email de test à la boîte surveillée, attendre le prochain intervalle, et voir si l'agent le détecte et logge l'événement. Tester que l'agent ne détecte pas un email déjà lu ou déjà traité.
- **Test du Webhook** :
- Créer l'abonnement via l'agent. Vérifier dans les logs que l'abonnement a été créé (id reçu) et que la phase de validation Graph a eu lieu (Graph devrait apparaître dans les logs du serveur web de l'agent au endpoint de validation).
- Envoyer un email test : Observer que presque immédiatement, le endpoint webhook reçoit une notification POST. Vérifier que l'agent fetch le mail et passe au module IA.
- Observer le traitement IA sur cet email test. Par exemple, si c'est un email connu ("Hello world"), on peut configurer le module IA pour simplement retourner un résultat constant pour le test, afin de voir que le flux complet fonctionne.
- Vérifier les actions suite au traitement : si l'agent doit répondre, voir la réponse envoyée dans la boîte d'envoi; si un ticket devait être créé, vérifier l'appel correspondant.
- **Simulation d'erreurs** : Tester la robustesse :
- Stopper la connexion internet du container (si possible) puis la rétablir pour voir si l'agent récupère bien (par ex, comment il gère l'impossibilité temporaire d'appeler Graph).
- Envoyer un email mal formaté ou très volumineux (avec pièce jointe lourde) pour voir comment l'agent réagit (peut imposer une limite de taille à traiter via l'IA, etc.).
- Tester un contenu en anglais ou dans une autre langue si le modèle IA doit gérer plusieurs langues.
- **Performance** : Envoyer, disons, 10 emails quasi simultanément et voir si l'agent les traite tous (si le webhook envoie plusieurs notifications d'affilée, assure qu'on traite en parallèle ou file d'attente sans en oublier).

- **Vérification finale** : Une fois satisfait, on pourra activer l'agent sur la vraie boîte avec des emails de production. Surveiller de près les premiers jours le comportement (logs et éventuellement mettre un mode debug verbose initialement).

8. Améliorations Futures

Une fois l'agent de base en place, des fonctionnalités supplémentaires pourront être envisagées : - **Interface d'administration** : Développer une petite interface web (protégée) pour visualiser les emails reçus et les actions prises par l'agent, ajuster des paramètres (seuils de confiance de l'IA, templates de réponse, etc.), voire intervenir manuellement si besoin. - **Apprentissage continu** : Si l'IA fait de la classification/réponse, recueillir les cas où l'agent s'est trompé ou a dû escalader à un humain, afin d'améliorer les prompts ou le modèle. Possibilité d'intégrer un apprentissage supervisé (feedback loop). - **Support multicanal** : Étendre l'agent à d'autres canaux que l'email. Par exemple, traiter des messages Teams ou Slack de manière similaire (Graph API permet de souscrire à des messages Teams par exemple). - **Optimisation du contexte** : Pour les réponses, connecter l'agent à une base de connaissances interne. Par ex, si c'est une question fréquente, l'agent pourrait aller chercher la réponse dans une FAQ interne plutôt que générer de zéro. - **Gestion des pièces jointes** : Analyser le contenu des pièces jointes (PDF, images) via OCR ou autres techniques si pertinent, et intégrer ces données dans l'analyse. Exemple: un email avec un bon de commande PDF -> l'agent lit le PDF, extrait le numéro de commande. - **Sécurité avancée** : Implémenter la vérification de signature du contenu du mail ou l'utilisation de l'option `includeResourceData` du webhook pour éviter un aller-retour Graph (mais cela nécessite de gérer les clés de chiffrement [13]). - **Scalabilité** : Si le volume d'emails augmente, on pourrait imaginer plusieurs agents ou un découpage (un service gère les webhooks et répartit les mails à plusieurs workers IA via une queue). Actuellement on part sur un seul processus séquentiel ou légèrement multi-thread.

En résumé, cet agent IA de messagerie combinera les capacités de l'API Microsoft Graph pour interagir avec la boîte mail et la puissance des modèles d'intelligence artificielle pour analyser et répondre aux emails. La présente spécification détaille la configuration nécessaire (Azure AD, Docker, DNS) [1], l'architecture logicielle, et le fonctionnement étape par étape (de la détection d'un nouvel email [9] jusqu'à la réponse automatique). En suivant ces directives, l'implémentation Python pourra être réalisée par Codex ou un développeur, aboutissant à un agent opérationnel sur `agentmail.padaw.ovh`. Les citations incluses se réfèrent à la documentation Microsoft Graph et des ressources pertinentes pour garantir que les meilleures pratiques sont respectées (permissions de lecture des mails [4], utilisation correcte des webhooks et renouvellement [11], etc.). Avec cet outil, la gestion des emails gagnera en efficacité et intelligence, tout en s'intégrant harmonieusement à l'environnement existant.

---

[1] [2] [3] [5] [6] [7] How to collect Email data from Microsoft Graph API with Python
https://www.515tech.com/post/collect-email-data-from-microsoft-graph-api_with_python

[4] [12] [13] [17] Change notifications for Outlook resources in Microsoft Graph - Microsoft Graph | Microsoft Learn
https://learn.microsoft.com/en-us/graph/outlook-change-notifications-overview

[8] [9] [10] [11] Microsoft Graph Webhooks - What, Why, How & Best Practices
https://www.voitanos.io/blog/microsoft-graph-webhook-delta-query/

[14] Python Azure Graph API How to Read Emails, Python ... - YouTube
https://www.youtube.com/watch?v=ah8Fm5Rtr7M

15 Microsoft Graph API Read Mail with Python - Stack Overflow
https://stackoverflow.com/questions/70025901/microsoft-graph-api-read-mail-with-python

16 Why am I receiving "reauthorizationRequired" lifecycle notifications ...
https://learn.microsoft.com/en-us/answers/questions/737296/why-am-i-receiving-reauthorizationrequired-lifecyc