Plateforme d'Entreprise Agentique : Spécifications Techniques Détaillées

(Version 1.0 - Projet Pilote : Souscription d'Assurance Automatisée)

1. Introduction

1.1. Objet du Document et Public Cible

Ce document constitue le plan d'exécution technique définitif pour la mise en œuvre du projet pilote de la Plateforme d'Entreprise Agentique. Il a pour objet de traduire les exigences fonctionnelles, telles que décrites dans le document *Spécifications Fonctionnelles Détaillées v1.0*, en spécifications et tâches d'ingénierie concrètes, précises et actionnables. Ce document établit le "comment" technique en réponse au "quoi" fonctionnel.

Il est la source unique de vérité pour l'implémentation technique du système. Toute déviation par rapport aux spécifications contenues dans ce document doit faire l'objet d'un processus formel de revue d'architecture et de gestion du changement.

Le public cible de ce document est exclusivement technique et se compose des équipes suivantes :

- **Équipes de Développement Logiciel :** Ingénieurs responsables du développement des Workers Camunda et des Agents Autonomes.
- **Équipes DevOps / Ingénierie de Plateforme :** Spécialistes responsables du provisionnement de l'infrastructure, de la configuration des environnements, et de la mise en place des pipelines d'intégration et de déploiement continus (CI/CD).
- **Équipes d'Opérations / SRE (Site Reliability Engineering) :** Ingénieurs chargés de la surveillance, de la maintenance, de la réponse aux incidents et de la garantie de la fiabilité de la plateforme en production.

1.2. Portée des Spécifications Techniques

La portée de ce document est rigoureusement limitée à l'implémentation technique du projet pilote "Souscription d'Assurance Automatisée". Les spécifications couvrent les domaines suivants :

- **Infrastructure Cloud :** Architecture d'hébergement, topologie réseau, et provisionnement via Infrastructure-as-Code.
- **Plateformes Technologiques :** Configuration détaillée des services managés (Confluent Cloud) et des logiciels auto-gérés (Camunda 8).
- **Conception Logicielle :** Piles technologiques, patrons de conception, et standards de développement pour tous les composants applicatifs (Workers et Agents).
- **Communication :** Spécifications pour la couche de communication événementielle (Kafka) et la sérialisation des données.
- **Automatisation du Cycle de Vie :** Définition des pipelines CI/CD, des stratégies de scan de sécurité et de la gestion des versions.
- **Sécurité Technique :** Protocoles d'authentification, de chiffrement et d'analyse de la sécurité du code.
- **Observabilité et Opérations :** Standards pour la journalisation, les métriques et le traçage distribué.
- **Stratégies de Test :** Approches pour les tests d'intégration, de contrat, de performance et de chaos.

Sont explicitement hors de portée de cette version 1.0 :

- La conception de l'interface utilisateur (UI) et de l'expérience utilisateur (UX).
- La logique métier détaillée, qui est encapsulée et référencée dans les modèles DMN (Decision Model and Notation).
- La feuille de route stratégique de la plateforme au-delà de la phase pilote.

1.3. Références Documentaires

Ce document s'inscrit dans une hiérarchie documentaire et dépend des documents de référence suivants :

1.3.1. Architecture de Référence v1.0

Ce document fondateur établit les principes architecturaux non négociables de la plateforme (par exemple, architecture événementielle, microservices, cloud-native). Toutes les décisions techniques et spécifications contenues dans le présent document sont contraintes et guidées par ces principes.

1.3.2. Spécifications Fonctionnelles Détaillées v1.0

Ce document décrit les exigences fonctionnelles et les cas d'usage du projet pilote. Chaque composant technique et chaque spécification décrits ci-après doivent pouvoir être tracés jusqu'à une ou plusieurs exigences fonctionnelles du SFD, assurant ainsi un alignement complet entre les besoins métier et la solution technique.

2. Architecture d'Infrastructure et Environnements

Cette section définit la fondation cloud-native sur laquelle l'ensemble de la plateforme opérera. Les spécifications sont conçues pour garantir la résilience, la scalabilité, la sécurité et l'efficacité opérationnelle dès le premier jour.

2.1. Architecture d'Hébergement Cible (ex: Kubernetes sur AWS/Azure/GCP)

La plateforme sera déployée sur **Amazon Elastic Kubernetes Service (EKS)**. Ce choix stratégique permet de déléguer la gestion complexe du plan de contrôle Kubernetes à AWS, réduisant ainsi la charge opérationnelle, tout en bénéficiant d'une intégration profonde avec l'écosystème AWS pour le réseau, la sécurité, le stockage et l'observabilité.[1]

L'architecture du cluster EKS se conformera rigoureusement aux meilleures pratiques de l'industrie pour assurer la sécurité, la résilience et la performance :

- **Déploiement Multi-AZ :** Les nœuds de travail (worker nodes) seront répartis sur un minimum de trois Zones de Disponibilité (Availability Zones - AZs). Cette répartition sera gérée par des **EKS Managed Node Groups**, ce qui élimine les points de défaillance uniques au niveau de l'infrastructure et garantit une haute disponibilité.[3]
- **Isolation par Namespaces :** Les environnements (dev, staging, prod) seront

logiquement isolés au sein d'un même cluster (pour les environnements de non-production) ou sur des clusters dédiés (pour la production) à l'aide de namespaces Kubernetes. Cette pratique est fondamentale pour la sécurité, car elle permet d'appliquer des politiques de contrôle d'accès basées sur les rôles (RBAC) de manière granulaire. Elle facilite également la gestion des ressources par environnement en utilisant des objets ResourceQuotas et LimitRanges pour prévenir la consommation excessive de ressources par une équipe ou une application.[5]

- **Autoscaling :** Une stratégie d'autoscaling à deux niveaux sera mise en œuvre :
  - **Horizontal Pod Autoscaler (HPA) :** Pour les services sans état (Agents, Workers), le HPA augmentera ou diminuera automatiquement le nombre de réplicas de pods en fonction de l'utilisation du CPU ou de la mémoire.
  - **Karpenter :** Pour le dimensionnement au niveau des nœuds, Karpenter sera utilisé de préférence au Cluster Autoscaler standard. Karpenter provisionne de nouveaux nœuds de manière plus réactive et optimisée en fonction des besoins réels des pods en attente, ce qui améliore l'efficacité des coûts et la vitesse de mise à l'échelle.[2]
- **Sondes de Santé (Probes) :** Il est **obligatoire** que chaque conteneur déployé (Agents, Workers, composants Camunda) définisse à la fois une livenessProbe et une readinessProbe.
  - La livenessProbe permet à Kubernetes de détecter si une application est bloquée ou en état d'échec, et de la redémarrer automatiquement.
  - La readinessProbe indique à Kubernetes quand une application est prête à accepter du trafic. Cela empêche que des requêtes soient envoyées à un pod qui est encore en cours de démarrage ou temporairement incapable de traiter des requêtes, garantissant ainsi l'absence d'erreurs pour l'utilisateur final.[3]

2.2. Topologie Réseau (VPC, Sous-réseaux, Groupes de Sécurité, Accès Privé)

Une topologie réseau robuste et sécurisée est fondamentale. Une approche multi-VPC sera adoptée pour une isolation maximale entre les environnements.

- **Conception des VPC :** Un Virtual Private Cloud (VPC) dédié sera provisionné pour chaque environnement principal (dev, staging, prod). Cette approche multi-VPC est privilégiée par rapport à un VPC partagé pour maximiser la sécurité, simplifier la gestion des plages d'adresses IP et minimiser le "rayon d'explosion" (blast radius) en cas de mauvaise configuration réseau ou d'incident de sécurité.[8]
- **Structure des Sous-réseaux :** Au sein de chaque VPC, une stratégie de sous-réseaux multi-niveaux sera mise en œuvre pour segmenter le trafic et renforcer la sécurité :
  - **Sous-réseaux Publics :** Contiendront les ressources exposées à Internet, telles que

les Application Load Balancers (ALB) et les NAT Gateways. Aucune charge de travail applicative ne sera déployée dans ces sous-réseaux.
    - **Sous-réseaux Privés :** Hébergeront toutes les charges de travail applicatives, y compris les nœuds de travail EKS, les pods des agents, les workers Camunda et les autres services de soutien. Ces ressources n'auront pas d'adresse IP publique et ne seront pas directement accessibles depuis Internet.[4]
- **Connectivité :**
    - **Internet Gateway (IGW) :** Attaché à chaque VPC pour permettre le trafic sortant des ressources dans les sous-réseaux publics (par exemple, les ALBs).
    - **NAT Gateways :** Provisionnés dans chaque sous-réseau public (un par AZ pour la haute disponibilité) pour permettre aux ressources des sous-réseaux privés (par exemple, un worker Camunda devant appeler une API externe) d'accéder à Internet de manière contrôlée et sécurisée, sans être exposées aux connexions entrantes.[11]
    - **VPC Endpoints :** Des points de terminaison VPC (Gateway pour S3/DynamoDB, Interface pour les autres services) seront systématiquement utilisés pour permettre un accès privé et sécurisé aux services AWS (ECR, Secrets Manager, S3, etc.) depuis les sous-réseaux privés. Cela garantit que le trafic ne transite pas par l'Internet public, ce qui améliore la sécurité, réduit la latence et diminue les coûts de transfert de données.[8]
    - **Communication Inter-VPC :** Pour les communications contrôlées entre les VPC des différents environnements (par exemple, pour permettre à un pipeline CI/CD de se connecter à plusieurs clusters), **AWS Transit Gateway** sera utilisé. Il agira comme un hub réseau centralisé, offrant une gestion simplifiée et une meilleure évolutivité par rapport à une multitude de connexions de peering VPC.[8]
- **Groupes de Sécurité (Security Groups) :** Ils agiront comme des pare-feu avec état (stateful) au niveau des instances EC2 et des pods. Le principe du moindre privilège sera strictement appliqué : par défaut, tout trafic est refusé, et seules les règles explicitement nécessaires (port, protocole, source) seront autorisées. La fonctionnalité **Security Groups for Pods** sera activée sur EKS pour attribuer des groupes de sécurité directement aux pods, permettant ainsi de définir des politiques réseau micro-segmentées pour la communication inter-services au sein du cluster.[2]

Le choix d'une topologie multi-VPC, bien que très sécurisée, introduit une complexité pour les processus qui doivent traverser les frontières des environnements, comme le déploiement d'artefacts. Si chaque environnement possédait son propre registre de conteneurs (ECR), la promotion d'une image de l'ECR de staging à celui de prod nécessiterait des configurations réseau et IAM complexes. Pour contourner ce problème, une stratégie alternative est adoptée : un **registre ECR unique et centralisé** sera utilisé pour tous les environnements. La séparation sera assurée par des politiques IAM et des conventions de nommage des images. Le pipeline CI/CD (décrit à la section 4) sera responsable de la promotion d'une image en la re-taguant, par exemple de my-agent:1.2.0-staging à my-agent:1.2.0-prod, après validation. Cette approche simplifie radicalement l'architecture réseau tout en maintenant une

séparation logique et sécurisée des artefacts.

Table 2.1: Allocation Réseau par Environnement

Environnement	Région AWS	CIDR du VPC	CIDR des Sous-réseaux Publics	CIDR des Sous-réseaux Privés
dev	eu-west-3	10.10.0.0/16	10.10.1.0/24, 10.10.2.0/24, 10.10.3.0/24	10.10.10.0/24, 10.10.11.0/24, 10.10.12.0/24
staging	eu-west-3	10.20.0.0/16	10.20.1.0/24, 10.20.2.0/24, 10.20.3.0/24	10.20.10.0/24, 10.20.11.0/24, 10.20.12.0/24
prod	eu-west-3	10.30.0.0/16	10.30.1.0/24, 10.30.2.0/24, 10.30.3.0/24	10.30.10.0/24, 10.30.11.0/24, 10.30.12.0/24

2.3. Provisionnement de l'Infrastructure (Infrastructure-as-Code avec Terraform/OpenTofu)

L'automatisation et la reproductibilité de l'infrastructure sont des piliers de notre stratégie opérationnelle.

- **Obligation d'IaC :** L'ensemble des composants d'infrastructure (VPC, clusters EKS, rôles IAM, groupes de sécurité, endpoints VPC, etc.) **doit** être défini, provisionné et géré en tant qu'Infrastructure-as-Code (IaC) à l'aide de **Terraform**. Cette approche déclarative garantit la reproductibilité, le versionnement, l'auditabilité et la collaboration sur les changements d'infrastructure, éliminant ainsi les configurations manuelles sujettes à erreur.[3]
- **Structure Modulaire :** Le code Terraform sera organisé en modules réutilisables et bien définis (par exemple, un module vpc, un module eks-cluster, un module iam-role). Cette modularité favorise la cohérence entre les environnements, réduit la duplication de code

et simplifie la maintenance. Le module officiel et largement adopté terraform-aws-modules/eks/aws sera utilisé comme base pour la création de nos clusters EKS, en bénéficiant de l'expertise de la communauté et des meilleures pratiques intégrées.[15]

- **Gestion de l'État :** L'état de Terraform (terraform.tfstate), qui est une représentation de l'infrastructure gérée, sera stocké à distance dans un backend centralisé et sécurisé. Spécifiquement, un **bucket Amazon S3** sera utilisé pour le stockage, avec le **verrouillage d'état (state locking)** activé via une **table DynamoDB**. Ce mécanisme est critique pour le travail en équipe, car il empêche les modifications concurrentes de l'état, qui pourraient entraîner une corruption et des incohérences dans l'infrastructure.[15]
- **Stratégie Multi-Environnement :** Les **workspaces Terraform** seront utilisés pour gérer des états distincts pour chaque environnement (dev, staging, prod) à partir de la même base de code. Cela permet de déployer des variations de l'infrastructure (par exemple, avec des tailles d'instance différentes) tout en utilisant les mêmes modules. Les configurations spécifiques à chaque environnement seront externalisées dans des fichiers de variables (.tfvars), séparant ainsi la configuration de la logique.[18]

2.4. Spécifications de Configuration de Confluent Cloud

La plateforme d'événements est l'épine dorsale de notre architecture. Sa configuration doit garantir la durabilité des données et la performance.

2.4.1. Type de Cluster

Pour le projet pilote, un cluster Confluent Cloud de type **Standard** sera provisionné. Ce type de cluster offre un déploiement multi-zone par défaut, fournissant un excellent équilibre entre le coût et la haute disponibilité requis pour une première mise en production. Un cluster de type **Enterprise** pourra être envisagé dans les phases ultérieures si des fonctionnalités avancées comme le peering VPC privé avec notre infrastructure AWS deviennent nécessaires.[20]

2.4.2. Configuration des Topics Kafka

Une configuration standardisée et rigoureuse des topics est impérative pour la fiabilité et la performance de la plateforme. Les paramètres suivants sont obligatoires pour tous les topics en production.

- **Facteur de Réplication (replication.factor) :** Ce paramètre **doit** être fixé à **3**. Cette valeur est non négociable en production. Elle s'aligne sur l'architecture multi-AZ des fournisseurs de cloud et garantit la durabilité des données en permettant au système de survivre à la défaillance de jusqu'à deux brokers (N-1) sans perte de données.[21]
- **Partitions (num.partitions) :** Le nombre de partitions est le principal levier de parallélisme pour les consommateurs. Un nombre insuffisant de partitions peut créer un goulot d'étranglement. Le nombre initial de partitions pour un topic sera déterminé en fonction du débit attendu et du parallélisme de consommation souhaité. En règle générale, un minimum de **6 à 12 partitions** sera utilisé pour les topics métiers principaux, ce qui offre une marge de manœuvre pour augmenter le nombre de consommateurs à l'avenir.[21]
- **Répliques Synchronisées Minimales (min.insync.replicas) :** Ce paramètre **doit** être fixé à **2**. Lorsqu'il est utilisé en conjonction avec la configuration acks=all du producteur (voir section 3.4.1), il garantit qu'une écriture n'est confirmée qu'après avoir été répliquée de manière durable sur au moins deux brokers. Ce mécanisme est la clé pour garantir une tolérance aux pannes de type "zéro perte de données" en cas de défaillance d'un seul broker.[20]

Table 2.2: Standards de Configuration des Topics Kafka

Usage du Topic	Convention de Nommage	Partitions (Défaut)	replication.factor	min.insync.replicas	Rétention (Jours)	Compression
Événements Métier	business.<domaine>.<evenement>.v<N>	12	**3** (Obligatoire)	**2** (Obligatoire)	7	zstd
Commandes d'Agent	agent.<type>.<commande>.v	6	**3** (Obligatoire)	**2** (Obligatoire)	1	lz4

	<N>					
Files de Lettres Mortes (DLQ)	<topic-source>.dlq	1	**3** (Obligatoire)	**2** (Obligatoire)	30	gzip

2.5. Spécifications de Déploiement de Camunda 8 (SaaS/Auto-géré)

La décision est prise d'opter pour un déploiement **Auto-géré (Self-Managed)** de Camunda 8 sur le cluster EKS cible. Cette orientation stratégique privilégie la cohérence architecturale, le contrôle total sur la résidence des données et une intégration plus profonde avec notre pile d'observabilité et de sécurité personnalisée. Bien que l'option SaaS offre une plus grande commodité opérationnelle, l'approche auto-gérée aligne le moteur de workflow avec le reste de notre pile applicative, créant un plan opérationnel unifié.[27] Cette décision transfère la responsabilité de la disponibilité, de la mise à l'échelle et des mises à jour du moteur de Camunda vers l'équipe de plateforme interne, ce qui nécessite une expertise Kubernetes et Camunda 8 avérée.[29]

- **Méthode de Déploiement :** Le déploiement sera entièrement géré via les **charts Helm** officiels fournis par Camunda. Cette approche standardise la configuration, simplifie les mises à jour et s'intègre naturellement dans nos pipelines CI/CD.[30]

2.5.1. Configuration du Cluster Zeebe

Le cluster Zeebe est le cœur du moteur de processus et sa configuration est critique pour la performance et la résilience.

- **Taille et Réplication :** Le cluster sera déployé avec **3 brokers**. Les paramètres Helm clusterSize et replicationFactor seront tous deux fixés à **3**. Cette configuration est le minimum requis pour un environnement de production tolérant aux pannes, capable de survivre à la perte d'un broker sans interruption de service.[30]
- **Partitions :** Le nombre de partitions (partitionCount) sera fixé à **6**, un multiple du facteur de réplication. Cela permet une distribution équilibrée des leaders de partition sur les trois brokers, optimisant ainsi la répartition de la charge.[31]
- **Allocation des Ressources :** Sur la base des benchmarks de performance, des requêtes

et limites de ressources (CPU et mémoire) spécifiques seront allouées à chaque pod de broker Zeebe pour garantir une performance stable et prévisible (par exemple, requests: {cpu: "2", memory: "4Gi"}, limits: {cpu: "4", memory: "8Gi"}). Il est impératif d'utiliser une classe de stockage (StorageClass) Kubernetes qui provisionne des volumes persistants sur des disques SSD à haute performance (type gp3 sur AWS avec des IOPS provisionnés) pour minimiser la latence de traitement de Zeebe, qui est fortement dépendante des performances d'écriture sur disque.[32]

2.5.2. Configuration des Composants (Operate, Tasklist, Connectors)

- Tous les composants auxiliaires (Operate, Tasklist, Identity, Connectors) seront déployés via le même chart Helm parent que Zeebe, garantissant la cohérence des versions.
- **Ingress :** Un contrôleur d'ingress NGINX sera déployé dans le cluster pour exposer les applications web (Operate, Tasklist) ainsi que le point de terminaison gRPC du Gateway Zeebe. Un domaine unique avec un routage basé sur le chemin sera configuré (par exemple, camunda.pilot.mycompany.com/operate, camunda.pilot.mycompany.com/tasklist).[33] Le trafic gRPC vers le Gateway Zeebe sera géré via un Ingress de type grpc.
- **Datastore (Elasticsearch/OpenSearch) :** Pour stocker les données exportées par Zeebe, le chart Helm sera configuré pour se connecter à un service managé **Amazon OpenSearch** externe, plutôt que de déployer une instance Elasticsearch au sein du cluster. Cette décision stratégique décharge l'équipe de la charge opérationnelle de la gestion d'un cluster de recherche avec état, complexe et gourmand en ressources, tout en bénéficiant de la scalabilité et de la résilience d'un service managé AWS.[34]

2.6. Spécifications d'Exécution des Agents

2.6.1. Conteneurisation

Tous les agents autonomes seront packagés en tant qu'images Docker. Un modèle de Dockerfile multi-étapes standardisé sera fourni pour garantir la production d'images finales légères et sécurisées. L'image de base sera une image minimale et durcie, telle que

python:3.11-slim ou une image Distroless, afin de réduire la surface d'attaque.[6]

2.6.2. Orchestration de Conteneurs

Les agents seront déployés sur le cluster EKS en tant que Deployments Kubernetes. La gestion du cycle de vie et de la configuration de ces déploiements sera standardisée à l'aide de **charts Helm**. Un chart Helm modèle sera créé et servira de base pour tous les agents. Il paramétrisera les éléments variables tels que le tag de l'image, les requêtes et limites de ressources, les variables d'environnement et les configurations spécifiques à chaque environnement (dev, staging, prod).[35]

3. Conception et Développement des Composants

Cette section établit les normes d'ingénierie logicielle, les patrons de conception et les standards que les équipes de développement sont tenues de respecter pour garantir la qualité, la maintenabilité et la cohérence de la plateforme.

3.1. Pile Technologique (Tech Stack)

3.1.1. Langages et Cadriciels

- **Workers Camunda :** Le développement des workers s'effectuera en **Java 17** avec le framework **Spring Boot 3.x**. Ce choix s'appuie sur la maturité et la richesse fonctionnelle de la bibliothèque cliente spring-zeebe. Cette dernière offre une intégration transparente avec l'écosystème Spring, incluant des mécanismes robustes de gestion des erreurs, des stratégies de relance (retry) configurables et une gestion simplifiée du cycle de vie des workers.[32]
- **Agents Autonomes (Intégration LLM) :** Le développement des agents se fera en **Python 3.11** avec le framework **FastAPI**. Cette pile technologique est sélectionnée pour ses hautes performances dans les opérations asynchrones liées aux entrées/sorties

(I/O-bound), ce qui est idéal pour les appels API vers les modèles de langage (LLM). De plus, l'écosystème Python est dominant dans le domaine de l'intelligence artificielle et du machine learning, offrant un accès inégalé aux bibliothèques et outils pertinents.

3.1.2. Bibliothèques Logicielles Clés

- **Pour l'écosystème Java/Spring Boot :**
  - io.camunda:spring-zeebe-client : Pour l'intégration avec le moteur Zeebe.
  - org.springframework.cloud:spring-cloud-starter-vault-config : Pour l'intégration native avec HashiCorp Vault.
  - io.confluent:kafka-avro-serializer : Pour la sérialisation des événements Kafka au format Avro.
  - org.springframework.kafka:spring-kafka : Pour l'intégration avec Kafka.
- **Pour l'écosystème Python/FastAPI :**
  - pyzeebe : Client Python pour interagir avec le moteur Zeebe.
  - confluent-kafka[avro] : Client Kafka de Confluent avec support Avro et intégration au Schema Registry.
  - fastapi et uvicorn : Pour la création de services web asynchrones.
  - httpx : Pour effectuer des appels API asynchrones vers les LLM et autres services externes.
  - opentelemetry-distro et opentelemetry-instrumentation-fastapi : Pour l'instrumentation automatique et le traçage distribué.

3.2. Spécifications des Workers Camunda

3.2.1. Patron de Conception du Client "External Task"

Toutes les tâches de service (serviceTask) dans les modèles BPMN qui nécessitent l'exécution de code personnalisé **doivent** être implémentées en utilisant le patron de conception "External Task".[38] Ce modèle découple le moteur de processus de l'implémentation de la logique métier. Le moteur publie une tâche dans une file d'attente (un "topic" de jobs) et attend qu'un worker externe la récupère, l'exécute et la complète.

Ce découplage offre des avantages architecturaux majeurs :

- **Scalabilité indépendante :** Les workers peuvent être mis à l'échelle horizontalement (en ajoutant plus d'instances) indépendamment du moteur de processus, en fonction de la charge de travail spécifique à leur fonction.
- **Résilience :** Une défaillance dans un worker n'impacte pas directement le moteur de processus. La tâche restera simplement dans la file d'attente et pourra être reprise par une autre instance du worker ou après un redémarrage.
- **Polyglottisme :** Bien que nous standardisions sur Java pour les workers, ce patron permettrait à l'avenir d'écrire des workers dans n'importe quel langage disposant d'un client gRPC pour Zeebe.[38]

Règles d'implémentation :

- Chaque fonction métier distincte doit avoir son propre jobType (topic) dédié. La réutilisation d'un même jobType pour des tâches différentes est **strictement interdite** afin de garantir la clarté, la maintenabilité et d'éviter les conflits.[41]
- Les workers doivent être conçus pour être **sans état (stateless)**. Tout état nécessaire à l'exécution d'une tâche doit être fourni via les variables du processus. Les résultats de l'exécution doivent être retournés en tant que nouvelles variables ou mises à jour de variables à la complétion de la tâche.

3.2.2. Gestion de la Configuration et des Secrets

La sécurité des informations sensibles est une priorité absolue.

- **Obligation :** Aucun secret (clé d'API, mot de passe, token, certificat) ne doit être stocké en clair dans le code source, les fichiers de configuration, les variables d'environnement ou les images de conteneur. La gestion de tous les secrets **doit** être centralisée dans **HashiCorp Vault**.[42]
- **Intégration avec Spring Boot :** Les workers Spring Boot s'intégreront à Vault en utilisant la bibliothèque spring-cloud-starter-vault-config. L'authentification de l'application auprès de Vault se fera via la **méthode d'authentification Kubernetes**. Dans ce flux, le pod de l'application utilise son jeton de compte de service (Service Account JWT) pour s'authentifier auprès de Vault et obtenir un token Vault temporaire. Cette approche, connue sous le nom de "zero-secret bootstrap", élimine la nécessité de gérer un secret initial pour l'application elle-même.[44]
- **Injection et Rafraîchissement Dynamique :** Les secrets seront injectés dynamiquement dans le contexte de l'application Spring au démarrage. Pour les secrets qui peuvent être renouvelés (par exemple, les identifiants de base de données dynamiques), les beans de configuration correspondants (comme DataSource) devront être annotés avec @RefreshScope. Cela permettra à Spring Cloud de rafraîchir ces beans et de recharger les secrets depuis Vault sans nécessiter un redémarrage complet de l'application, ce qui

est crucial pour la haute disponibilité.[43]

3.2.3. Implémentation de la Logique de Compensation (Saga)

Pour maintenir la cohérence des données à travers plusieurs services distribués dans des transactions métier de longue durée (comme le processus complet de souscription d'assurance), le patron de conception **Saga** sera utilisé.[46]

- **Style d'Implémentation :** Une **Saga de type Orchestration** sera mise en œuvre. Le modèle de processus BPMN lui-même agira en tant qu'orchestrateur de la saga.[47] Cette approche a été choisie car elle rend la logique de gestion des échecs explicite et visible, ce qui est un avantage considérable pour la compréhension, la maintenance et l'audit du processus. Le flux de compensation n'est pas caché dans le code des microservices, mais modélisé graphiquement.
- **Mécanisme de Compensation :**
  - Pour chaque action transactionnelle (par exemple, "Réserver un créneau de souscription"), une **action de compensation** correspondante doit être définie (par exemple, "Libérer le créneau de souscription").
  - Dans le modèle BPMN, les tâches qui peuvent échouer seront entourées d'un **événement de bordure d'erreur (error boundary event)**.
  - En cas d'échec d'une tâche, cet événement interceptera l'erreur et redirigera le flux du processus vers une séquence de tâches de compensation. Ces tâches appelleront les actions de compensation nécessaires, généralement dans l'ordre inverse de l'exécution des actions originales, pour annuler la transaction et ramener le système à un état cohérent.[46]
- Ce choix architectural privilégie la visibilité et le contrôle explicite du processus au détriment de l'indépendance totale des services. Par conséquent, tout service participant à une saga doit exposer une action de compensation discrète (via une API REST ou un message Kafka dédié) que le processus BPMN orchestrateur peut invoquer.

3.3. Spécifications des Agents Autonomes

3.3.1. Structure de Projet Type

Un modèle de dépôt Git standardisé (cookiecutter ou un template de dépôt GitLab) sera créé pour tous les agents basés sur Python. Ce modèle inclura :

- Une structure de répertoires prédéfinie (/app, /tests, /schemas).
- Un Dockerfile optimisé et multi-étapes.
- Un modèle de chart Helm.
- Un fichier pyproject.toml avec les dépendances de base (FastAPI, confluent-kafka, opentelemetry-distro, etc.).
- Du code de démarrage (boilerplate) pour initialiser les producteurs et consommateurs Kafka, la configuration de l'observabilité et l'intégration avec Vault.

3.3.2. Intégration Sécurisée avec les API des LLM

Les clés d'API des fournisseurs de LLM sont des secrets extrêmement sensibles et leur gestion doit être particulièrement rigoureuse.

- Les clés d'API des LLM **doivent** être stockées dans HashiCorp Vault. Elles ne doivent **jamais** être présentes dans le code source, les fichiers de configuration, les variables d'environnement en clair, ou être intégrées dans les images de conteneur.[51]
- L'agent Python récupérera la clé d'API du LLM depuis Vault à son démarrage, en utilisant la même méthode d'authentification Kubernetes que les workers Java.
- Toutes les communications avec les API des fournisseurs de LLM **doivent** transiter par une **passerelle d'API (API Gateway) interne et centralisée**. L'agent n'appellera jamais directement le point de terminaison public du LLM. Cette passerelle aura les responsabilités suivantes :
    - Injecter la clé d'API (récupérée de Vault) dans la requête sortante.
    - Appliquer des politiques de limitation de débit (rate limiting) et de limitation de la simultanéité (throttling) pour contrôler les coûts et prévenir les abus.
    - Journaliser de manière centralisée toutes les requêtes et réponses pour l'audit et l'analyse des coûts.
    - Effectuer la validation et l'assainissement (sanitization) des entrées et des sorties pour se prémunir contre les attaques par injection de prompt (prompt injection).[53]

      Cette passerelle constitue un point de contrôle et d'observation critique pour toutes les interactions avec les IA.

3.4. Spécifications de la Couche de Communication

3.4.1. Configuration des Producteurs et Consommateurs Kafka

La fiabilité de la communication événementielle est la pierre angulaire de la plateforme.

- **Producteurs (Producers) :**
  - acks : **Doit** être configuré sur all (ou -1). Cette configuration est non négociable pour toutes les données critiques. Elle garantit que le producteur attend la confirmation de l'écriture du message par le leader et toutes les répliques synchronisées (in-sync replicas), assurant ainsi la plus haute durabilité.[26]
  - enable.idempotence : **Doit** être configuré sur true. Cette option cruciale garantit que les tentatives de renvoi (retries) du producteur en cas d'erreur réseau ne créent pas de messages en double dans le topic. Elle active un mécanisme de déduplication côté broker basé sur un ID de producteur et un numéro de séquence, assurant une sémantique de livraison "exactement une fois" (exactly-once) par partition.[56]
  - retries : Doit être configuré sur une valeur élevée (par exemple, Integer.MAX_VALUE). L'activation de l'idempotence rend les relances sûres, il est donc préférable de laisser le client Kafka gérer les erreurs réseau transitoires de manière persistante.[57]
- **Consommateurs (Consumers) :**
  - enable.auto.commit : **Doit** être configuré sur false. La validation des offsets (commit) sera gérée manuellement dans le code de l'application. L'offset d'un message ne sera validé qu'après que son traitement a été entièrement et avec succès terminé. Cette approche garantit une sémantique de traitement "au moins une fois" (at-least-once) et prévient la perte de messages si le consommateur s'arrête après avoir reçu un message mais avant de l'avoir traité.[56]
  - **Idempotence du Consommateur :** La logique applicative du consommateur **doit** être conçue pour être **idempotente**. Étant donné que la sémantique "au moins une fois" peut entraîner la re-livraison d'un même message en cas d'échec après le traitement mais avant la validation de l'offset, le code de traitement doit être capable de gérer cette situation sans créer d'effets de bord indésirables (par exemple, en vérifiant si une opération a déjà été effectuée avant de la ré-exécuter).

3.4.2. Implémentation de la Sérialisation/Désérialisation (Avro et classes générées)

L'utilisation d'un format de données structuré et d'un schéma est essentielle pour la gouvernance des données et l'évolutivité.

- **Obligation d'Avro :** Tous les messages échangés entre les services via Kafka **doivent** être sérialisés en utilisant le format **Apache Avro**. L'utilisation de formats non structurés comme JSON brut ou des chaînes de caractères est **interdite** pour la communication inter-services afin d'imposer un contrat de données strict et de bénéficier des performances de la sérialisation binaire.[60]
- **Gestion des Schémas :** Tous les schémas Avro (fichiers .avsc) seront versionnés dans un dépôt Git dédié. Ils **doivent** être enregistrés et gérés de manière centralisée dans **Confluent Schema Registry**. Le Schema Registry agit comme un catalogue de schémas et permet d'appliquer des règles de compatibilité (par exemple, BACKWARD_COMPATIBLE) lors de l'évolution des schémas, empêchant ainsi les producteurs de déployer des changements qui casseraient les consommateurs existants.[60]
- L'instauration d'un Schema Registry n'est pas seulement une décision technique, mais un acte de gouvernance fondamental. Elle force la collaboration inter-équipes sur les contrats de données. Une équipe ne peut plus modifier un format de message unilatéralement. Le processus de modification d'un schéma devient un acte formel, nécessitant une validation de compatibilité et potentiellement l'approbation des équipes consommatrices, ce qui est intégré directement dans le pipeline CI/CD (voir section 4).
- **Génération de Code :**
    - **Python (Agents) :** Des classes de données Python (dataclasses) seront générées automatiquement à partir des schémas Avro via un script intégré au pipeline CI/CD. Cela offre les avantages du typage statique, de l'auto-complétion dans les IDE et de la validation au moment de la compilation, ce qui améliore considérablement la productivité des développeurs et réduit les erreurs d'exécution.[62] Les bibliothèques AvroSerializer et AvroDeserializer de confluent-kafka-python seront utilisées pour une intégration transparente avec le Schema Registry.[63]
    - **Java (Workers) :** Des classes Java spécifiques seront générées à partir des schémas Avro en utilisant le plugin avro-maven-plugin dans le processus de build. Ces classes seront utilisées avec les KafkaAvroSerializer et KafkaAvroDeserializer de Confluent pour la communication avec Kafka.[61]

4. Intégration et Déploiement Continus (CI/CD)

Cette section définit les pipelines automatisés qui constituent l'épine dorsale de notre processus de développement et de livraison, garantissant la rapidité, la fiabilité et la sécurité de nos déploiements.

4.1. Description des Outils

- **Plateforme CI/CD : GitLab CI** est la plateforme obligatoire pour toute l'intégration et le déploiement continus. Tous les pipelines doivent être définis dans des fichiers .gitlab-ci.yml situés à la racine de chaque dépôt de projet.[35]
- **Exécuteurs (Runners) :** Les jobs des pipelines seront exécutés par des **GitLab Runners** configurés sur notre cluster EKS. L'exécuteur Kubernetes sera utilisé, ce qui permet de lancer des pods éphémères pour chaque job, garantissant un environnement d'exécution propre et isolé.

4.2. Pipeline de Déploiement pour un Modèle BPMN/DMN

Ce pipeline est conçu pour les dépôts contenant exclusivement des modèles de processus (BPMN) et de décision (DMN).

- **Déclencheur :** Le pipeline est déclenché à chaque fusion (merge) sur la branche main.
- **Étapes (Stages) et Tâches (Jobs) :**
  1. **validate :** Un job qui utilise un outil de validation statique (linter), tel que bpmn-js-cli, pour vérifier la syntaxe des modèles et leur conformité avec les directives de modélisation internes. Un échec à cette étape bloque le pipeline.
  2. **deploy-to-staging :** Un job qui s'exécute uniquement sur la branche main. Il utilise l'outil en ligne de commande zbctl (le CLI de Zeebe) pour déployer les modèles sur le cluster Camunda de l'environnement staging. Cela permet une validation rapide des modèles dans un environnement intégré.[66]
  3. **deploy-to-prod :** Un job avec une approbation manuelle (when: manual). Lorsqu'il est déclenché manuellement par un responsable de version depuis l'interface GitLab, ce job déploie les mêmes modèles sur le cluster de production. Cette étape manuelle constitue une porte de contrôle humaine essentielle avant toute modification en production.[66]

4.3. Pipeline de Déploiement pour un Agent

Ce pipeline est le standard pour toutes les applications conteneurisées (Agents Python et Workers Java).

- **Déclencheur :** Le pipeline s'exécute à chaque push sur n'importe quelle branche, ainsi

que lors de la création ou de la mise à jour d'une demande de fusion (Merge Request).

4.3.1. Étapes (Build, Test, Scan de Sécurité, Push vers le Registre de Conteneurs)

Le pipeline est structuré en plusieurs étapes séquentielles :

1. **build :**
   - Compile le code source (pour les projets Java).
   - Construit l'image Docker en utilisant le Dockerfile standardisé du projet.
   - Tag l'image avec le hash du commit Git ($CI_COMMIT_SHA) pour une traçabilité parfaite.
2. **test :**
   - Exécute les tests unitaires.
   - Exécute les tests d'intégration. Pour les tests nécessitant des dépendances externes (comme Kafka, une base de données ou Vault), **Testcontainers** sera utilisé pour démarrer des conteneurs éphémères de ces services, garantissant des tests fiables et isolés.
3. **security-scan :** Cette étape est une porte de qualité de sécurité non négociable.
   - **SAST (Static Application Security Testing) :** Un job intègre un outil d'analyse statique de la sécurité du code (par exemple, **SonarQube** ou **Semgrep**, intégrés nativement dans GitLab Security). Il analyse le code source à la recherche de vulnérabilités connues (injections SQL, XSS, etc.). Le pipeline échouera si des vulnérabilités de sévérité critique sont détectées.[68]
   - **Analyse des Dépendances :** Utilise la fonctionnalité intégrée de GitLab pour scanner les bibliothèques tierces (pom.xml, pyproject.toml) à la recherche de vulnérabilités connues (CVEs).
   - **Analyse de Conteneur :** Scanne l'image Docker construite à l'étape build pour détecter des vulnérabilités au niveau du système d'exploitation de base et de ses paquets.
4. **push-to-registry :**
   - Ce job s'exécute uniquement sur la branche main.
   - Si toutes les étapes précédentes ont réussi, l'image Docker, validée et scannée, est poussée vers le registre central **Amazon ECR**.
   - L'image est taguée avec la version sémantique extraite du tag Git (par exemple, 1.2.3).
5. **deploy-to-staging :**
   - Déploie l'application sur l'environnement de staging en utilisant la commande helm upgrade --install. Le chart Helm est configuré pour utiliser le tag de l'image qui vient d'être poussée.[35]
6. **run-e2e-tests :**

    - Après le déploiement réussi sur staging, un job déclenche une suite de tests de bout en bout (end-to-end) qui valident le comportement de l'application dans un environnement entièrement intégré.
7. **deploy-to-prod :**
    - Ce job est configuré avec une approbation manuelle (when: manual).
    - Lorsqu'il est déclenché, il exécute la même commande helm upgrade --install que pour staging, mais en ciblant le namespace de production. Cela garantit que l'artefact exact qui a été testé sur staging est celui qui est déployé en production, respectant ainsi le principe d'immutabilité des artefacts de déploiement.[36]

4.4. Stratégie de Gestion des Versions (Semantic Versioning) pour les Processus, Agents et Schémas

Une stratégie de versionnement cohérente est essentielle pour gérer les dépendances et la compatibilité dans un écosystème de microservices.

- **Obligation :** Tous les artefacts versionnés — Agents/Workers (applications), Schémas Avro, et Définitions de Processus BPMN/DMN — **doivent** adhérer à la spécification **Semantic Versioning 2.0.0 (SemVer)**.[72]
- **Format de Version :** La version doit suivre le format MAJEUR.MINEUR.PATCH (par exemple, 1.2.5).
- **Application aux Artefacts :**
    - **Agents / Workers (Applications) :**
        - MAJEUR : Incrémenté pour tout changement non rétrocompatible (breaking change) dans l'API ou le comportement du service.
        - MINEUR : Incrémenté pour l'ajout de nouvelles fonctionnalités de manière rétrocompatible.
        - PATCH : Incrémenté pour des corrections de bugs rétrocompatibles.
    - **Schémas Avro :** Le versionnement est critique pour la gouvernance des données et la gestion de l'évolution des contrats.[73]
        - MAJEUR : Incrémenté pour tout changement non rétrocompatible (par exemple, suppression d'un champ, changement du type d'un champ, renommage d'un champ).
        - MINEUR : Incrémenté pour des changements rétrocompatibles (par exemple, ajout d'un nouveau champ optionnel avec une valeur par défaut).
        - PATCH : Généralement non utilisé pour les schémas ; les changements de type patch sont couverts par l'incrément MINEUR.
    - **Processus BPMN / DMN :**
        - MAJEUR : Incrémenté si un changement casse les instances de processus en cours ou nécessite une modification des workers/agents qui interagissent avec lui

(par exemple, changement du jobType d'une tâche, suppression d'une variable attendue, modification fondamentale du flux).
- MINEUR : Incrémenté pour l'ajout de nouveaux chemins optionnels, de nouvelles tâches qui n'affectent pas le flux principal, ou l'ajout de variables optionnelles.
- PATCH : Incrémenté pour des changements non exécutables comme la documentation, le repositionnement d'éléments graphiques ou des corrections de libellés.

L'application rigoureuse de SemVer aux schémas Avro est le mécanisme technique qui permet d'automatiser la gouvernance des données. Le pipeline CI/CD du dépôt de schémas sera configuré pour analyser la nouvelle version proposée dans une demande de fusion. Si une modification entraîne une incrémentation de la version MAJEUR, le pipeline peut automatiquement appliquer une politique d'approbation plus stricte, par exemple en exigeant l'approbation explicite des responsables de tous les services consommateurs connus. De cette manière, le pipeline devient le gardien automatisé des contrats de données, transformant un processus de gouvernance humain en une vérification automatisée et fiable, réduisant ainsi drastiquement le risque de défaillances en production dues à des formats de données incompatibles.

5. Sécurité Technique

La sécurité est intégrée à chaque niveau de la plateforme, suivant une approche de "défense en profondeur".

5.1. Authentification et Autorisation des Agents (OAuth 2.0 - Client Credentials Flow)

Pour la communication de service à service (par exemple, un agent appelant une API interne), le flux **OAuth 2.0 Client Credentials** est la méthode d'authentification et d'autorisation standardisée.[74] Ce flux est spécifiquement conçu pour les interactions machine-à-machine (M2M) où il n'y a pas d'utilisateur final impliqué.[75]

Le processus fonctionne comme suit :

1. Chaque service client (par exemple, un agent) est enregistré auprès d'un serveur d'autorisation central (par exemple, Keycloak ou Auth0, géré par notre plateforme d'identité) et reçoit un client_id et un client_secret.

2. Pour appeler un autre service, le client s'authentifie auprès du serveur d'autorisation en utilisant ses propres identifiants (client_id et client_secret).
3. Si les identifiants sont valides, le serveur d'autorisation émet un jeton d'accès (Access Token) au format JWT (JSON Web Token). Ce jeton contient des informations sur le client (le sub ou client_id) et les permissions qui lui sont accordées (les scopes).[75]
4. Le client inclut ce jeton d'accès dans l'en-tête Authorization de sa requête HTTP (Authorization: Bearer <token>) lorsqu'il appelle le service cible.
5. Le service cible valide le jeton (vérifie sa signature, son expiration et son audience) et vérifie que les scopes qu'il contient autorisent l'opération demandée.

Cette approche centralise la gestion des identités des services et permet un contrôle d'accès granulaire basé sur les permissions, plutôt que de se fier à une simple confiance réseau.

5.2. Chiffrement des Données en Transit (TLS) et au Repos

Toutes les données, qu'elles soient en mouvement ou stockées, doivent être chiffrées.

- **Données en Transit :** Tout le trafic réseau, sans exception, doit être chiffré à l'aide de **TLS (Transport Layer Security) version 1.2 ou supérieure**. Cela s'applique à :
  - Les communications externes (utilisateurs accédant à Operate/Tasklist).
  - Les communications internes entre les microservices au sein du cluster Kubernetes (via un service mesh comme Istio ou Linkerd, qui applique le mTLS par défaut).
  - Les connexions aux services AWS (VPC Endpoints utilisant HTTPS).
  - Les connexions à Confluent Cloud et HashiCorp Vault.
    L'utilisation de protocoles non chiffrés (comme HTTP ou telnet) est strictement interdite.77
- **Données au Repos :** Toutes les données stockées de manière persistante doivent être chiffrées.
  - Les volumes persistants EBS attachés aux nœuds EKS (utilisés par Zeebe) doivent être chiffrés à l'aide d'une clé gérée par **AWS Key Management Service (KMS)**.
  - Les objets stockés dans les buckets S3 (par exemple, pour les sauvegardes) doivent être chiffrés côté serveur (SSE-S3 ou SSE-KMS).
  - Les données dans le service Amazon OpenSearch doivent être chiffrées au repos.
  - Les secrets stockés dans HashiCorp Vault sont chiffrés par défaut.[80]

5.3. Analyse Statique de la Sécurité du Code (SAST) et des Dépendances

La sécurité doit être intégrée dès les premières étapes du cycle de développement ("shift-left security").

- **SAST :** L'intégration d'un outil SAST dans le pipeline CI/CD, comme spécifié à la section 4.3.1, est obligatoire. Cet outil analyse le code source à chaque commit pour identifier les modèles de code vulnérables avant même que l'application ne soit construite. Les résultats de l'analyse doivent être traités comme des bugs, avec une politique de "zero tolerance" pour les vulnérabilités critiques, qui doivent bloquer la fusion du code.[68]
- **Analyse des Dépendances (SCA - Software Composition Analysis) :** Le pipeline CI/CD doit également intégrer une analyse automatisée des dépendances tierces. Cet outil scanne les manifestes de dépendances (par exemple, pom.xml, pyproject.toml) et identifie les bibliothèques contenant des vulnérabilités de sécurité connues (CVEs). La présence de vulnérabilités critiques ou élevées dans une dépendance doit également bloquer le pipeline, forçant les développeurs à mettre à jour la bibliothèque ou à trouver une alternative.[84]

6. Observabilité et Opérations

Une observabilité complète est essentielle pour opérer, déboguer et optimiser un système distribué complexe. Les trois piliers de l'observabilité — la journalisation, les métriques et le traçage — seront mis en œuvre de manière standardisée sur toute la plateforme.

6.1. Journalisation (Logging)

6.1.1. Format Standard des Journaux (JSON structuré)

Tous les composants applicatifs (Workers, Agents) et les composants d'infrastructure qui le permettent **doivent** produire des journaux (logs) au format **JSON structuré**. L'utilisation de logs en texte brut non structuré est proscrite.[85]

Le format JSON permet aux systèmes de collecte et d'analyse de logs (comme OpenSearch, Loki ou Datadog) de parser, d'indexer et de requêter les logs de manière efficace et puissante.

Chaque entrée de log est un objet JSON avec des paires clé-valeur bien définies.

Un schéma de log standard de base sera adopté, incluant les champs suivants pour chaque entrée de log :

- timestamp : Horodatage précis au format ISO 8601 (par exemple, 2024-10-27T10:00:00.123Z).
- level : Niveau de sévérité du log (par exemple, INFO, WARN, ERROR, DEBUG).
- service_name : Nom du service émettant le log (par exemple, insurance-agent, credit-check-worker).
- message : Le message de log principal, lisible par un humain.
- trace_id : L'identifiant de trace distribuée (voir section 6.3).
- span_id : L'identifiant du span de la trace distribuée.

Les traces d'appels (stack traces) des exceptions doivent également être structurées en JSON, en décomposant la trace en un tableau d'objets, chacun contenant le fichier, le numéro de ligne et le nom de la méthode.[85]

6.1.2. Stratégie de Corrélation des Identifiants de Trace

Pour permettre une analyse transversale des problèmes, il est impératif de pouvoir corréler les logs, les traces et les métriques. L'identifiant de trace (trace_id) et l'identifiant de span (span_id) issus du système de traçage distribué (OpenTelemetry) **doivent** être injectés dans chaque entrée de log.

Cela permet, à partir d'une seule trace d'erreur dans un système comme Jaeger ou Uptrace, de pivoter directement vers tous les logs générés par tous les services impliqués dans cette transaction spécifique, simplifiant radicalement le débogage.[88]

6.2. Métriques (Metrics)

6.2.1. Métriques Techniques à Exposer

Chaque service doit exposer un ensemble standard de métriques techniques pour surveiller

sa santé et ses performances. Les métriques clés à surveiller incluent :

- **Latence des agents :**
  - **Latence de traitement de l'action :** Temps total entre la réception d'une commande et l'envoi d'un événement de résultat.[90]
  - **Latence des appels externes (LLM) :** Temps de réponse des API externes, incluant le temps de premier octet (Time To First Byte - TTFB) pour les réponses en streaming.[91]
- **Lag des consommateurs Kafka :** Il s'agit de la métrique la plus critique pour une architecture événementielle. Elle mesure le nombre de messages de retard entre le dernier message produit dans une partition et le dernier message consommé par un groupe de consommateurs. Un lag qui augmente de manière continue indique qu'un consommateur n'est pas capable de traiter les messages assez rapidement, ce qui peut entraîner des retards dans le traitement métier.[92]
- **Débit (Throughput) :** Nombre de messages traités par seconde, nombre de requêtes API par seconde.
- **Taux d'erreur :** Pourcentage de traitements de messages ou de requêtes API qui se terminent en erreur.
- **Métriques de la JVM (pour les workers Java) :** Utilisation du tas (heap), activité du garbage collector, nombre de threads.
- **Métriques Zeebe/Camunda :**
  - zeebe_stream_processor_records_total : Pour mesurer le débit global de traitement du moteur.
  - zeebe_job_events_total : Pour suivre le cycle de vie des jobs (créés, activés, complétés, échoués).
  - Métriques de contre-pression (backpressure) : Pour détecter si le cluster Zeebe est surchargé.[95]

6.2.2. Outils de Supervision

- **Collecte : Prometheus** sera utilisé comme standard pour la collecte et le stockage des métriques de séries temporelles. Tous les services devront exposer leurs métriques via un endpoint /metrics au format Prometheus.
- **Visualisation : Grafana** sera utilisé pour créer des tableaux de bord (dashboards) de visualisation des métriques collectées par Prometheus. Des tableaux de bord standardisés seront créés pour chaque type de service (agent, worker) et pour la plateforme globale (Zeebe, Kafka).
- **Alerting : Alertmanager** (composant de l'écosystème Prometheus) sera utilisé pour définir des règles d'alerte sur les métriques critiques (par exemple, "alerter si le lag Kafka dépasse 1000 messages pendant plus de 5 minutes").

6.3. Traçage (Tracing)

6.3.1. Implémentation du Traçage Distribué (OpenTelemetry)

Pour comprendre le parcours d'une requête à travers notre système distribué polyglotte (Java et Python), le traçage distribué est indispensable.

- **Standard : OpenTelemetry (OTel)** est adopté comme le standard unique pour l'instrumentation, la collecte et l'exportation des données de traçage.[98] Son approche neutre vis-à-vis des fournisseurs permet de changer de backend d'analyse sans modifier l'instrumentation du code.
- **Instrumentation :**
  - **Java/Spring Boot :** L'agent Java OpenTelemetry sera utilisé pour l'**instrumentation automatique**. En attachant l'agent à la JVM au démarrage, il instrumentera automatiquement les frameworks et bibliothèques courants (Spring Web, clients HTTP, clients Kafka, JDBC) sans modification du code.[101]
  - **Python/FastAPI :** L'instrumentation automatique de Python via opentelemetry-instrument sera utilisée. Elle instrumente dynamiquement les bibliothèques populaires comme FastAPI, httpx, et confluent-kafka.[103]
- **Propagation de Contexte :** Le cœur du traçage distribué est la propagation du contexte de trace (contenant trace_id et parent_span_id) d'un service à l'autre. OpenTelemetry gère cela automatiquement pour les protocoles standards :
  - **Appels HTTP/gRPC :** Le contexte est propagé via les en-têtes HTTP (en utilisant la norme W3C Trace Context).[106]
  - **Messages Kafka :** Le contexte de trace sera injecté dans les en-têtes des messages Kafka par les bibliothèques d'instrumentation OTel. Le consommateur extraira automatiquement ce contexte pour continuer la trace.[89]
- **Collecte et Visualisation :** Un **OpenTelemetry Collector** sera déployé en tant qu'agent sur chaque nœud ou en tant que déploiement centralisé pour recevoir les données de télémétrie des applications, les traiter (par exemple, échantillonnage) et les exporter vers un backend de traçage comme **Jaeger** ou **Uptrace**.

7. Stratégies de Tests Techniques

Une stratégie de test multi-niveaux est requise pour garantir la robustesse et la fiabilité d'un système distribué.

7.1. Tests d'Intégration entre les Agents et Camunda

Ces tests valident que les composants (workers, agents) interagissent correctement avec le moteur de processus Camunda et les autres services.

- **Objectif :** Vérifier que les workers s'abonnent correctement aux bons jobType, qu'ils peuvent traiter les variables d'entrée et retourner les variables de sortie attendues, et que les agents produisent et consomment correctement les messages Kafka qui déclenchent ou poursuivent les processus.[107]
- **Environnement de Test :** Ces tests seront exécutés dans le pipeline CI/CD. L'environnement de test sera orchestré à l'aide de **Testcontainers**. Un ensemble de conteneurs Docker (un moteur Zeebe, un broker Kafka, un Schema Registry) sera démarré dynamiquement pour chaque suite de tests, fournissant un environnement d'intégration éphémère et entièrement isolé.[109]
- **Scénarios :** Les tests simuleront des flux de processus partiels. Par exemple, un test pourrait démarrer une instance de processus, vérifier qu'un job est créé, puis simuler un worker qui complète ce job et vérifier que le processus avance à l'état suivant attendu.[110]

7.2. Tests de Contrat pour les Schémas d'Événements Kafka

Pour éviter que des modifications dans un service producteur ne cassent les services consommateurs dans une architecture événementielle, des tests de contrat sont obligatoires.

- **Outil : Pact** sera utilisé pour les tests de contrat sur les messages Kafka.[111] Pact adopte une approche "consumer-driven", où le consommateur dicte le contrat.
- **Flux de Travail :**
  1. **Côté Consommateur :** Dans la suite de tests du service consommateur, un test est écrit pour définir la structure et le contenu du message qu'il s'attend à recevoir d'un topic Kafka. L'exécution de ce test génère un fichier de contrat (le "pacte") qui décrit ces attentes.[112]
  2. **Publication du Contrat :** Ce fichier de pacte est publié sur un **Pact Broker** centralisé.

3. **Côté Producteur :** Dans le pipeline CI/CD du service producteur, un test de vérification est exécuté. Ce test télécharge le contrat depuis le Pact Broker et vérifie que le message réellement généré par le producteur est conforme aux attentes du consommateur. Si le producteur a introduit un changement non compatible, ce test échouera, empêchant le déploiement du changement et signalant la rupture du contrat avant qu'elle n'atteigne la production.[114]

- Cette approche élimine le besoin de tests de bout en bout fragiles et coûteux pour valider les intégrations de messages, tout en permettant aux équipes de faire évoluer leurs services de manière indépendante et sûre.[111]

7.3. Tests de Performance et de Charge sur le Processus Pilote

L'objectif de ces tests est de valider que la plateforme peut supporter la charge de production attendue tout en respectant les exigences de performance (latence).

- **Environnement :** Les tests de performance seront exécutés sur un environnement de staging dédié et dimensionné de manière identique à l'environnement de production. Les résultats obtenus sur un environnement sous-dimensionné ne sont pas pertinents.[116]
- **Scénarios :** Des scénarios de test réalistes seront définis, basés sur les cas d'usage du projet pilote. Ils doivent simuler non seulement le volume (par exemple, nombre d'instances de processus démarrées par seconde) mais aussi la complexité des processus (nombre de tâches, taille des variables).[32]
- **Outils :** Un outil de génération de charge comme **k6**, **Gatling** ou **JMeter** sera utilisé pour simuler l'activité des utilisateurs ou des systèmes externes qui initient les processus. Un générateur de charge spécifique à Camunda 8, comme le projet camunda-8-benchmark, pourra être utilisé pour simuler l'activité des workers et générer une charge directement sur le moteur Zeebe.[118]
- **Objectifs (SLOs) :** Des objectifs de niveau de service clairs doivent être définis avant les tests. Par exemple :
  - Soutenir un débit de 50 nouvelles souscriptions par seconde.
  - Maintenir une latence de complétion de processus (P95) inférieure à 2 secondes.
- **Analyse :** Les résultats seront analysés en corrélant les métriques de l'outil de charge avec les métriques d'observabilité de la plateforme (CPU/mémoire des pods, lag Kafka, latence de traitement de Zeebe) pour identifier les goulots d'étranglement.[116]

7.4. Scénarios de Test de Chaos (ex: simulation de panne d'un agent, latence réseau)

Le Chaos Engineering est la pratique consistant à injecter des défaillances de manière contrôlée dans un système pour tester sa résilience et découvrir les faiblesses avant qu'elles ne se manifestent en production.[120]

- **Outils :** Des outils natifs à Kubernetes comme **Chaos Mesh** ou **LitmusChaos** seront utilisés pour orchestrer les expériences de chaos.[122]
- **Scénarios pour le Pilote :** Les expériences de chaos suivantes seront menées dans l'environnement de staging :
    1. **Panne d'un Agent/Worker :** Suppression aléatoire de pods d'agents ou de workers. **Hypothèse :** Le Deployment Kubernetes doit automatiquement recréer les pods manquants. Pour les workers Camunda, les jobs non acquittés doivent redevenir disponibles et être traités par d'autres instances. Pour les consommateurs Kafka, une rééquilibration du groupe de consommateurs doit se produire et le traitement doit reprendre sur les instances restantes.[125]
    2. **Latence Réseau :** Injection de latence (par exemple, 100 ms) sur les appels réseau entre un worker et une dépendance externe, ou entre deux services internes. **Hypothèse :** Les mécanismes de timeout et de retry du client HTTP doivent se comporter comme prévu. Les métriques de latence doivent refléter l'augmentation et des alertes doivent être déclenchées.[126]
    3. **Surcharge des Ressources :** Injection d'une charge CPU ou d'une consommation mémoire élevée sur un pod. **Hypothèse :** Les ResourceQuotas et LimitRanges de Kubernetes doivent contenir l'impact. L'Horizontal Pod Autoscaler (HPA) devrait, si configuré, augmenter le nombre de réplicas pour gérer la charge.[125]
    4. **Indisponibilité d'une Dépendance :** Simulation d'une panne du service de base de données ou d'une API externe critique. **Hypothèse :** Les patrons de résilience comme les disjoncteurs (circuit breakers) doivent s'ouvrir, empêchant les appels en cascade vers un service défaillant. Le processus BPMN doit gérer l'erreur via un événement de bordure et potentiellement déclencher une logique de compensation (Saga).[121]

8. Annexes

8.1. Exemples de Fichiers de Configuration

values.yaml pour le Chart Helm d'un Agent Python

YAML

```yaml
# values.yaml - Exemple pour un agent de souscription
replicaCount: 2

image:
  repository: 123456789012.dkr.ecr.eu-west-3.amazonaws.com/agent-insurance-subscription
  pullPolicy: IfNotPresent
  # Le tag est généralement surchargé par le pipeline CI/CD
  tag: "1.0.0"

# Spécifications des ressources pour le pod
resources:
  requests:
    cpu: "250m"
    memory: "512Mi"
  limits:
    cpu: "1"
    memory: "1Gi"

# Configuration de l'application via les variables d'environnement
# Les secrets ne sont PAS définis ici, mais montés via le sidecar de l'agent Vault
env:
  - name: KAFKA_BOOTSTRAP_SERVERS
    value: "pkc-xxxx.europe-west3.gcp.confluent.cloud:9092"
  - name: SCHEMA_REGISTRY_URL
    value: "https://psrc-xxxx.eu-central-1.aws.confluent.cloud"
  - name: CAMUNDA_ZEEBE_ADDRESS
    value: "camunda-zeebe-gateway.camunda.svc.cluster.local:26500"
  - name: OTEL_EXPORTER_OTLP_ENDPOINT
    value: "http://opentelemetry-collector.observability.svc.cluster.local:4317"
  - name: OTEL_SERVICE_NAME
    value: "agent-insurance-subscription"

# Annotations pour l'injection de l'agent Vault
```

```
podAnnotations:
  vault.hashicorp.com/agent-inject: "true"
  vault.hashicorp.com/role: "agent-insurance-subscription-role"
  vault.hashicorp.com/agent-inject-secret-llm-key: "secret/data/llm/openai"
  vault.hashicorp.com/agent-inject-template-llm-key: |
    {{- with secret "secret/data/llm/openai" -}}
    export OPENAI_API_KEY="{{.Data.data.api_key }}"
    {{- end -}}
```

docker-compose.yml pour l'Environnement de Développement Local

YAML

```
# docker-compose.yml pour un environnement de développement local simplifié
version: '3.8'

services:
  zeebe:
    image: camunda/zeebe:8.5.0
    ports:
      - "26500:26500"
      - "9600:9600"
    environment:
      -
ZEEBE_BROKER_EXPORTERS_ELASTICSEARCH_CLASSNAME=io.camunda.zeebe.exporter.Elasticsearch
Exporter
      - ZEEBE_BROKER_EXPORTERS_ELASTICSEARCH_ARGS_URL=http://elasticsearch:9200
      # Configuration pour un seul broker en local
      - ZEEBE_BROKER_CLUSTER_CLUSTERSIZE=1
      - ZEEBE_BROKER_CLUSTER_PARTITIONCOUNT=1
      - ZEEBE_BROKER_CLUSTER_REPLICATIONFACTOR=1
      - ZEEBE_BROKER_CLUSTER_INITIALCONTACTPOINTS=zeebe:26502
    networks:
      - camunda-net

  elasticsearch:
    image: docker.elastic.co/elasticsearch/elasticsearch:8.11.3
```

```
    ports:
      - "9200:9200"
    environment:
      - discovery.type=single-node
      - xpack.security.enabled=false
    networks:
      - camunda-net

  operate:
    image: camunda/operate:8.5.0
    ports:
      - "8080:8080"
    environment:
      - CAMUNDA_OPERATE_ZEEBE_GATEWAYADDRESS=zeebe:26500
      - CAMUNDA_OPERATE_ELASTICSEARCH_URL=http://elasticsearch:9200
    depends_on:
      - zeebe
      - elasticsearch
    networks:
      - camunda-net

networks:
  camunda-net:
    driver: bridge
```

8.2. Spécifications des API REST exposées (le cas échéant)

Cette section sera complétée si des agents ou des workers exposent des points de terminaison REST pour des raisons administratives ou de rappel (callbacks). La spécification sera fournie au format **OpenAPI 3.0**. Pour le projet pilote actuel, aucune API REST publique n'est exposée par les composants développés. Les interactions se font principalement via le moteur Camunda et la plateforme d'événements Kafka.

Ouvrages cités

1. Reference Architecture Examples and Best Practices - AWS, dernier accès : octobre 1, 2025, https://aws.amazon.com/architecture/
2. Amazon EKS Best Practices Guide, dernier accès : octobre 1, 2025, https://docs.aws.amazon.com/eks/latest/best-practices/introduction.html
3. Building Resilient Infrastructure with AWS and Kubernetes - CloudJournee,

dernier accès : octobre 1, 2025, https://www.cloudjournee.com/blog/aws-kubernetes-resilient-infrastructure-guide/
4. AWS VPC Design Best Practices - GeeksforGeeks, dernier accès : octobre 1, 2025, https://www.geeksforgeeks.org/devops/aws-vpc-design-best-practices/
5. 17 Kubernetes Best Practices Every Developer Should Know - Spacelift, dernier accès : octobre 1, 2025, https://spacelift.io/blog/kubernetes-best-practices
6. 10+ Kubernetes Architecture Best Practices For Performance, Security, And Cost Efficiency, dernier accès : octobre 1, 2025, https://www.cloudzero.com/blog/kubernetes-architecture/
7. Kubernetes concepts - Amazon EKS - AWS Documentation, dernier accès : octobre 1, 2025, https://docs.aws.amazon.com/eks/latest/userguide/kubernetes-concepts.html
8. Using VPC Sharing for a Cost-Effective Multi-Account Microservice Architecture - AWS, dernier accès : octobre 1, 2025, https://aws.amazon.com/blogs/architecture/using-vpc-sharing-for-a-cost-effective-multi-account-microservice-architecture/
9. Networking | Cloud Architecture Center, dernier accès : octobre 1, 2025, https://cloud.google.com/architecture/blueprints/security-foundations/networking
10. Proposed Infrastructure Setup on AWS for a Microservices Architecture: Overview of the Infrastructure and Components. | Codementor, dernier accès : octobre 1, 2025, https://www.codementor.io/@nicolaskh4/proposed-infrastructure-setup-on-aws-for-a-microservices-architecture-overview-of-the-infrastructure-and-components-1aijok5pcx
11. Example: VPC with servers in private subnets and NAT - AWS Documentation, dernier accès : octobre 1, 2025, https://docs.aws.amazon.com/vpc/latest/userguide/vpc-example-private-subnets-nat.html
12. Best practices and reference architectures for VPC design - Google Cloud, dernier accès : octobre 1, 2025, https://cloud.google.com/architecture/best-practices-vpc-design
13. Terraform Infrastructure as Code (IaC) Guide With Examples - Spacelift, dernier accès : octobre 1, 2025, https://spacelift.io/blog/terraform-infrastructure-as-code
14. Kubernetes Infrastructure as Code (IaC): Best Practices and Guide - Mirantis, dernier accès : octobre 1, 2025, https://www.mirantis.com/blog/kubernetes-infrastructure-as-code-iac-best-practices-and-guide/
15. Top 10 Terraform Best Practices for Infrastructure Automation | Atmosly - Medium, dernier accès : octobre 1, 2025, https://medium.com/atmosly/top-10-terraform-best-practices-for-infrastructure-automation-aa192c57b7aa
16. Mastering Kubernetes with Terraform: A Provider Deep Dive - Scalr, dernier accès : octobre 1, 2025,

https://scalr.com/learning-center/mastering-kubernetes-with-terraform-a-provider-deep-dive/
17. terraform-aws-modules/terraform-aws-eks: Terraform module to create Amazon Elastic Kubernetes (EKS) resources - GitHub, dernier accès : octobre 1, 2025, https://github.com/terraform-aws-modules/terraform-aws-eks
18. Deploying Multiple Environments with Terraform | by Chris Pisano | Capital One Tech, dernier accès : octobre 1, 2025, https://medium.com/capital-one-tech/deploying-multiple-environments-with-terraform-kubernetes-7b7f389e622
19. Basic repo layout for multi-environment AWS deployments : r/Terraform - Reddit, dernier accès : octobre 1, 2025, https://www.reddit.com/r/Terraform/comments/cbx0d2/basic_repo_layout_for_multienvironment_aws/
20. Configuration Reference for Topics in Confluent Cloud | Confluent ..., dernier accès : octobre 1, 2025, https://docs.confluent.io/cloud/current/topics/manage.html
21. Kafka Topics Choosing the Replication Factor and Partitions Count - Conduktor, dernier accès : octobre 1, 2025, https://learn.conduktor.io/kafka/kafka-topics-choosing-the-replication-factor-and-partitions-count/
22. Documentation - Apache Kafka, dernier accès : octobre 1, 2025, https://kafka.apache.org/documentation/
23. Kafka Replication and Committed Messages - Confluent Documentation, dernier accès : octobre 1, 2025, https://docs.confluent.io/kafka/design/replication.html
24. Kafka topic partitioning strategies and best practices - New Relic, dernier accès : octobre 1, 2025, https://newrelic.com/blog/best-practices/effective-strategies-kafka-topic-partitioning
25. Best Practices for Kafka Production Deployments in Confluent Platform, dernier accès : octobre 1, 2025, https://docs.confluent.io/platform/current/kafka/post-deployment.html
26. Kafka Producer Best Practices: Enabling Reliable Data Streaming - LimePoint, dernier accès : octobre 1, 2025, https://www.limepoint.com/blog/kafka-producer-best-practices-enabling-reliable-data-streaming
27. Camunda 8 Self-Managed, dernier accès : octobre 1, 2025, https://docs.camunda.io/docs/self-managed/about-self-managed/
28. Camunda8 saas or self managed - Discussion & Questions - Camunda Forum, dernier accès : octobre 1, 2025, https://forum.camunda.io/t/camunda8-saas-or-self-managed/48168
29. Camunda 8 - SaaS vs Self managed - Discussion & Questions, dernier accès : octobre 1, 2025, https://forum.camunda.io/t/camunda-8-saas-vs-self-managed/39621
30. Using Helm and Kubernetes to deploy Camunda 8, dernier accès : octobre 1, 2025,

https://camunda.com/blog/2022/05/using-helm-and-kubernetes-to-deploy-camunda-8/
31. Setting up a Zeebe cluster | Camunda 8 Docs, dernier accès : octobre 1, 2025, https://docs.camunda.io/docs/self-managed/zeebe-deployment/operations/setting-up-a-cluster/
32. Performance Tuning in Camunda 8, dernier accès : octobre 1, 2025, https://camunda.com/blog/2025/01/performance-tuning-camunda-8/
33. Camunda Self-Managed for Absolute Beginners, Part 2—Ingress and TLS SSL, dernier accès : octobre 1, 2025, https://camunda.com/blog/2024/01/camunda-self-managed-absolute-beginners-part-2-ingress-tls-ssl/
34. Unified Configuration for Orchestration Cluster - Camunda Product Roadmap, dernier accès : octobre 1, 2025, https://roadmap.camunda.com/c/169-unified-configuration-for-orchestration-cluster
35. Building a Kubernetes CI/CD Pipeline with GitLab and Helm | NextLink Labs, dernier accès : octobre 1, 2025, https://nextlinklabs.com/resources/insights/kubernetes-ci-cd-gitlab-with-helm
36. Kubernetes CI/CD Pipelines – 8 Best Practices and Tools - Spacelift, dernier accès : octobre 1, 2025, https://spacelift.io/blog/kubernetes-ci-cd
37. Helm deployments to a Kubernetes cluster with CI/CD - CircleCI, dernier accès : octobre 1, 2025, https://circleci.com/blog/helm-deployment-to-kubernetes/
38. Maximizing Camunda's Potential with External Task Pattern - Content Services, dernier accès : octobre 1, 2025, https://contentservices.asee.io/maximizing-camundas-potential-with-external-task-pattern/
39. Camunda External Task Worker - The Basics - viadee Blog, dernier accès : octobre 1, 2025, https://blog.viadee.de/en/camunda-external-task-worker
40. Camunda external tasks — a powerful tool for creating applications with a fault-tolerant and scalable architecture | by Alexandr Kazachenko | IT-компании Тинькофф | Medium, dernier accès : octobre 1, 2025, https://medium.com/its-tinkoff/camunda-external-tasks-a-powerful-tool-for-creating-applications-with-a-fault-tolerant-and-329b5ac3e1a6
41. Questions about External Task pattern - Camunda Forum, dernier accès : octobre 1, 2025, https://forum.camunda.io/t/questions-about-external-task-pattern/30341
42. Comprehensive Guide to Using HashiCorp Vault in a Spring Boot Environment - Medium, dernier accès : octobre 1, 2025, https://medium.com/@lgianlucaster/comprehensive-guide-to-using-hashicorp-vault-in-a-spring-boot-environment-c9cd2d27e6ce
43. Dynamic secrets management in spring boot application with vault | by Sudhakar Bhat, dernier accès : octobre 1, 2025, https://medium.com/@sudhakar.bhat247/dynamic-secrets-management-in-spring-boot-application-with-vault-6ae9cec80242
44. Vault: Spring Boot web app using Spring Cloud Vault to fetch secrets | Fabian Lee, dernier accès : octobre 1, 2025,

https://fabianlee.org/2023/11/12/vault-spring-boot-web-app-using-spring-cloud-vault-to-fetch-secrets/
45. Reload HashiCorp Vault secrets in Spring applications, dernier accès : octobre 1, 2025, https://developer.hashicorp.com/vault/tutorials/app-integration/spring-reload-secrets
46. Saga Design Pattern - Azure Architecture Center | Microsoft Learn, dernier accès : octobre 1, 2025, https://learn.microsoft.com/en-us/azure/architecture/patterns/saga
47. Pattern: Saga - Microservices.io, dernier accès : octobre 1, 2025, https://microservices.io/patterns/data/saga.html
48. Saga Pattern in Microservices | Baeldung on Computer Science, dernier accès : octobre 1, 2025, https://www.baeldung.com/cs/saga-pattern-microservices
49. Saga orchestration pattern - AWS Prescriptive Guidance, dernier accès : octobre 1, 2025, https://docs.aws.amazon.com/prescriptive-guidance/latest/cloud-design-patterns/saga-orchestration.html
50. The Saga Design Pattern: Coordinating Long-Running Transactions in Distributed Systems, dernier accès : octobre 1, 2025, https://medium.com/@CodeWithTech/the-saga-design-pattern-coordinating-long-running-transactions-in-distributed-systems-edbc9b9a9116
51. Best Practices for API Key Safety | OpenAI Help Center, dernier accès : octobre 1, 2025, https://help.openai.com/en/articles/5112595-best-practices-for-api-key-safety
52. Using Gemini API keys | Google AI for Developers, dernier accès : octobre 1, 2025, https://ai.google.dev/gemini-api/docs/api-key
53. Manage API keys - kgateway, dernier accès : octobre 1, 2025, https://kgateway.dev/docs/main/agentgateway/llm/api-keys/
54. How to expose APIs to LLMs without breaking security - Digital API, dernier accès : octobre 1, 2025, https://www.digitalapi.ai/blogs/expose-apis-to-llms
55. Secure API Management For LLM-Based Services - Protecto AI, dernier accès : octobre 1, 2025, https://www.protecto.ai/blog/secure-api-management-llm-based-services
56. Kafka: Producer & Consumer Best Practices | ActiveWizards: AI & Agent Engineering | Data Platforms, dernier accès : octobre 1, 2025, https://activewizards.com/blog/kafka-producer-and-consumer-best-practices
57. Chapter 5. Kafka producer configuration tuning - Red Hat Documentation, dernier accès : octobre 1, 2025, https://docs.redhat.com/en/documentation/red_hat_streams_for_apache_kafka/2.2/html/kafka_configuration_tuning/con-producer-config-properties-str
58. Idempotent Kafka Producer | Learn Apache Kafka with Conduktor, dernier accès : octobre 1, 2025, https://learn.conduktor.io/kafka/idempotent-kafka-producer/
59. Best practices for Apache Kafka clients - AWS Documentation, dernier accès : octobre 1, 2025, https://docs.aws.amazon.com/msk/latest/developerguide/bestpractices-kafka-cli

ent.html
60. Using Kafka with AVRO in Python | Towards Data Science, dernier accès : octobre 1, 2025, https://towardsdatascience.com/using-kafka-with-avro-in-python-da85b3e0f966/
61. Avro Schema Serializer and Deserializer for Schema Registry on Confluent Platform, dernier accès : octobre 1, 2025, https://docs.confluent.io/platform/current/schema-registry/fundamentals/serdes-develop/serdes-avro.html
62. Harmonizing Avro and Python: A Dance of Data Classes - Vortexa, dernier accès : octobre 1, 2025, https://www.vortexa.com/blog/harmonizing-avro-and-python-a-dance-of
63. confluent_kafka API — confluent-kafka 2.11.0 documentation, dernier accès : octobre 1, 2025, https://docs.confluent.io/platform/current/clients/confluent-kafka-python/html/index.html
64. Install confluent-kafka avro with pip - Codemia.io, dernier accès : octobre 1, 2025, https://codemia.io/knowledge-hub/path/install_confluent-kafka_avro_with_pip
65. CI/CD pipelines - GitLab Docs, dernier accès : octobre 1, 2025, https://docs.gitlab.com/ci/pipelines/
66. Deployments - GitLab Docs, dernier accès : octobre 1, 2025, https://docs.gitlab.com/ci/environments/deployments/
67. Deploy bpmn and dmn to production in camunda8 - Camunda Forum, dernier accès : octobre 1, 2025, https://forum.camunda.io/t/deploy-bpmn-and-dmn-to-production-in-camunda8/58624
68. Integrating SAST Into Your CI/CD Pipeline: A Step-by-Step Guide - Jit.io, dernier accès : octobre 1, 2025, https://www.jit.io/resources/app-security/integrating-sast-into-your-cicd-pipeline-a-step-by-step-guide
69. Static Application Security Testing (SAST) - GitLab Docs, dernier accès : octobre 1, 2025, https://docs.gitlab.com/user/application_security/sast/
70. Adding SAST to CI/CD Pipeline - GeeksforGeeks, dernier accès : octobre 1, 2025, https://www.geeksforgeeks.org/devops/adding-sast-to-ci-cd-pipeline/
71. Build a CI/CD pipeline for microservices on Kubernetes with Azure DevOps and Helm, dernier accès : octobre 1, 2025, https://learn.microsoft.com/en-us/azure/architecture/microservices/ci-cd-kubernetes
72. Semantic Versioning 2.0.0 | Semantic Versioning, dernier accès : octobre 1, 2025, https://semver.org/
73. Semantic Versioning - Microservice API Patterns, dernier accès : octobre 1, 2025, https://microservice-api-patterns.org/patterns/evolution/SemanticVersioning
74. OAuth 2.0 client credentials flow on the Microsoft identity platform, dernier accès : octobre 1, 2025, https://learn.microsoft.com/en-us/entra/identity-platform/v2-oauth2-client-creds

-grant-flow
75. Understanding the OAuth 2.0 Client Credentials flow - WorkOS, dernier accès : octobre 1, 2025, https://workos.com/blog/client-credentials
76. Client credentials flow in OAuth 2.0: How it works - Scalekit, dernier accès : octobre 1, 2025, https://www.scalekit.com/blog/client-credentials-flow-oauth
77. Data Encryption Best Practices - LMG Security, dernier accès : octobre 1, 2025, https://www.lmgsecurity.com/data-encryption-best-practices/
78. Encryption in transit for Google Cloud | Security, dernier accès : octobre 1, 2025, https://cloud.google.com/docs/security/encryption-in-transit
79. Protecting Data in Transit: Key Security Strategies - Oneleet, dernier accès : octobre 1, 2025, https://www.oneleet.com/blog/data-security-in-transit
80. What is Data at Rest | Security & Encryption Explained - Imperva, dernier accès : octobre 1, 2025, https://www.imperva.com/learn/data-security/data-at-rest/
81. 3. Protect data at rest and in transit - NCSC.GOV.UK, dernier accès : octobre 1, 2025, https://www.ncsc.gov.uk/collection/device-security-principles-for-manufacturers/protect-data-at-rest-and-in-transit
82. Azure data security and encryption best practices - Microsoft Learn, dernier accès : octobre 1, 2025, https://learn.microsoft.com/en-us/azure/security/fundamentals/data-encryption-best-practices
83. DevSecOps - Top Four OpenSource SAST tools for your CI/CD pipeline - GitHub Gist, dernier accès : octobre 1, 2025, https://gist.github.com/sttor/ba398d157a7ac3b3e42e18b4734bd1de
84. SAST: A guide to static application security testing | CircleCI, dernier accès : octobre 1, 2025, https://circleci.com/blog/static-application-security-testing-sast/
85. A Beginner's Guide to JSON Logging | Better Stack Community, dernier accès : octobre 1, 2025, https://betterstack.com/community/guides/logging/json-logging/
86. Structured logging: What it is and why you need it - New Relic, dernier accès : octobre 1, 2025, https://newrelic.com/blog/how-to-relic/structured-logging
87. JSON Logging: A Quick Guide for Engineers - Dash0, dernier accès : octobre 1, 2025, https://www.dash0.com/guides/json-logging
88. OpenTelemetry Distributed Tracing Implementation Guide - Logit.io, dernier accès : octobre 1, 2025, https://logit.io/blog/post/opentelemetry-distributed-tracing-implementation/
89. Implementing Distributed Tracing in a Polyglot Microservices Environment - Endowus Tech, dernier accès : octobre 1, 2025, https://tech.endowus.com/distributed-tracing-log-correlation/
90. Agent Action Latency: The Critical Performance Metric Every Enterprise Needs to Monitor, dernier accès : octobre 1, 2025, https://www.adopt.ai/glossary/agent-action-latency
91. Logs, metrics, and telemetry | LiveKit Docs, dernier accès : octobre 1, 2025, https://docs.livekit.io/agents/build/metrics/
92. Monitor Kafka Consumer Lag in Confluent Cloud, dernier accès : octobre 1, 2025, https://docs.confluent.io/cloud/current/monitoring/monitor-lag.html

93. A Guide to Fixing Kafka Consumer Lag [Without Jargon] - Last9, dernier accès : octobre 1, 2025, https://last9.io/blog/fixing-kafka-consumer-lag/
94. Apache Kafka Consumer Lag Monitoring: How to Check and Fix It to Stream Data Smoothly, dernier accès : octobre 1, 2025, https://sematext.com/blog/kafka-consumer-lag-offsets-monitoring/
95. Metrics - Camunda 8 Docs, dernier accès : octobre 1, 2025, https://docs.camunda.io/docs/self-managed/operational-guides/monitoring/metrics/
96. Metrics - Camunda 8 Docs, dernier accès : octobre 1, 2025, https://docs.camunda.io/docs/8.5/self-managed/zeebe-deployment/operations/metrics/
97. Metrics | Camunda 8 Docs, dernier accès : octobre 1, 2025, https://unsupported.docs.camunda.io/1.3/docs/self-managed/zeebe-deployment/operations/metrics/
98. Distributed Tracing with OpenTelemetry - Part I | SigNoz, dernier accès : octobre 1, 2025, https://signoz.io/blog/opentelemetry-distributed-tracing-part-1/
99. OpenTelemetry Implementation Guide: Distributed Tracing Mastery - Logit.io, dernier accès : octobre 1, 2025, https://logit.io/blog/post/opentelemetry-distributed-tracing-implementation-guide/
100. What is OpenTelemetry?, dernier accès : octobre 1, 2025, https://opentelemetry.io/docs/what-is-opentelemetry/
101. Getting Started by Example - OpenTelemetry, dernier accès : octobre 1, 2025, https://opentelemetry.io/docs/languages/java/getting-started/
102. OpenTelemetry for Java Applications: A Practical Guide - Lumigo, dernier accès : octobre 1, 2025, https://lumigo.io/opentelemetry/opentelemetry-for-java-applications-a-practical-guide/
103. Auto-Instrumentation Example | OpenTelemetry, dernier accès : octobre 1, 2025, https://opentelemetry.io/docs/zero-code/python/example/
104. Package opentelemetry-apm/fastapi - GitHub, dernier accès : octobre 1, 2025, https://github.com/users/blueswen/packages/container/package/opentelemetry-apm%2Ffastapi
105. Implementing OpenTelemetry in FastAPI - A Practical Guide - SigNoz, dernier accès : octobre 1, 2025, https://signoz.io/blog/opentelemetry-fastapi/
106. What is Distributed Tracing? Concepts & OpenTelemetry Implementation | Uptrace, dernier accès : octobre 1, 2025, https://uptrace.dev/opentelemetry/distributed-tracing
107. System Integration Testing (SIT) – A Detailed Overview - TestingXperts, dernier accès : octobre 1, 2025, https://www.testingxperts.com/blog/system-integration-testing
108. Integration Testing: A Comprehensive Guide with Code Examples | by Vansh Khandelwal, dernier accès : octobre 1, 2025, https://medium.com/@vansh.khandelwal06/integration-testing-a-comprehensive-guide-with-code-examples-35810ba3048d

109. Distributed systems testing - Codemia, dernier accès : octobre 1, 2025, https://codemia.io/knowledge-hub/path/distributed_systems_testing
110. Testing process definitions | Camunda 8 Docs, dernier accès : octobre 1, 2025, https://docs.camunda.io/docs/components/best-practices/development/testing-process-definitions/
111. Consumer Driven Contract Testing with Pact, Kafka and Spring Boot, dernier accès : octobre 1, 2025, https://capgemini.github.io/development/pact-contract-testing-with-kafka/
112. pact-foundation/pact-workshop-message: collection of pact message workshops in various languages - GitHub, dernier accès : octobre 1, 2025, https://github.com/pact-foundation/pact-workshop-message
113. Contract testing of the event-driven system with Kafka and Pact - SoftwareMill, dernier accès : octobre 1, 2025, https://softwaremill.com/contract-testing-of-the-event-driven-system-with-kafka-and-pact/
114. Event driven-systems | Pact Docs, dernier accès : octobre 1, 2025, https://docs.pact.io/implementation_guides/javascript/docs/messages
115. KAFKA CONTRACT TEST WITH PACT. Hello! In this article, we will talk... | by Samet Koç | Trendyol Tech | Medium, dernier accès : octobre 1, 2025, https://medium.com/trendyol-tech/kafka-contract-test-with-pact-abed78551382
116. Performance tuning Camunda 7, dernier accès : octobre 1, 2025, https://docs.camunda.io/docs/components/best-practices/operations/performance-tuning-camunda-c7/
117. How to Benchmark Your Camunda 8 Cluster | by Bernd Rücker - berndruecker, dernier accès : octobre 1, 2025, https://blog.bernd-ruecker.com/how-to-benchmark-your-camunda-8-cluster-48ada4b047b6
118. State of Zeebe Performance: Benchmarking Camunda's Workflow Engine for Scalability, dernier accès : octobre 1, 2025, https://camunda.com/blog/2025/02/state-of-zeebe-performance/
119. Helper to create benchmarks for Camunda Platform 8 and Zeebe - GitHub, dernier accès : octobre 1, 2025, https://github.com/camunda-community-hub/camunda-8-benchmark
120. What is chaos engineering? - Dynatrace, dernier accès : octobre 1, 2025, https://www.dynatrace.com/news/blog/what-is-chaos-engineering/
121. Chaos Engineering - Gremlin, dernier accès : octobre 1, 2025, https://www.gremlin.com/chaos-engineering
122. Chaos Mesh: A Powerful Chaos Engineering Platform for Kubernetes, dernier accès : octobre 1, 2025, https://chaos-mesh.org/
123. Chaos engineering kubernetes | by Alexandr Ivenin - Medium, dernier accès : octobre 1, 2025, https://medium.com/@alex.ivenin/chaos-engineering-in-kubernetes-4de425132ba1
124. Chaos Engineering on Kubernetes: A Beginner's Guide: Revel In Chaos - Blog, dernier accès : octobre 1, 2025,

https://seifrajhi.github.io/blog/chaos-engineering-kubernetes/
125. Getting started with Chaos Engineering on Kubernetes - Gremlin, dernier accès : octobre 1, 2025, https://www.gremlin.com/kubernetes-chaos-engineering
126. 6 Proven Chaos Testing Techniques for More Resilient APIs, dernier accès : octobre 1, 2025, https://www.gravitee.io/blog/chaos-testing-techniques
127. Understanding Chaos Engineering. Enterprise software systems have become... | by Miah Md Shahjahan | Medium, dernier accès : octobre 1, 2025, https://medium.com/@hasanshahjahan/understanding-chaos-engineering-3e853d757535