Livrable 1 — Analyse et architecture

Benchmark express des architectures

Objectif du benchmark

L'objectif de ce benchmark est de comparer trois styles d'architecture applicative — **monolithique**, **microservices** et **hexagonale** — afin d'identifier la solution la plus adaptée au projet **PeopleFirst**, en tenant compte :

- de la complexité fonctionnelle,
- des contraintes de performance (pics de charge fin de mois),
- de la maintenabilité,
- des compétences de l'équipe,
- et des exigences de disponibilité.

---

1. Architecture monolithique

Description

L'architecture monolithique repose sur une **application unique** regroupant l'ensemble des fonctionnalités métier (authentification, congés, documents, paie, etc.) au sein d'un même codebase, généralement déployée comme un seul artefact.

Avantages

- Simplicité de mise en place et de déploiement
- Rapidité de développement initial
- Faible complexité opérationnelle
- Débogage plus simple au début du projet

Inconvénients

- Scalabilité limitée (scaling global, pas ciblé)
- Couplage fort entre les modules
- Risque élevé lors des montées en charge (ex : calcul de la paie)
- Difficulté d'évolution sur le long terme

Cas d'usage typique

- Applications simples
- Équipes réduites
- Peu ou pas de pics de charge différenciés

---

2. Architecture microservices

Description

L'architecture microservices consiste à découper l'application en **services indépendants**, chacun responsable d'un périmètre fonctionnel précis, avec ses propres données et son cycle de vie.

Avantages

- Scalabilité fine par service
- Résilience accrue (isolation des pannes)
- Liberté technologique par service
- Déploiement indépendant

Inconvénients

- Complexité technique élevée
- Coût important en infrastructure
- Besoin fort en DevOps et observabilité
- Temps de mise en œuvre long
- Surdimensionné pour un projet en démarrage

Cas d'usage typique

- Très grandes plateformes
- Forte volumétrie et trafic constant
- Équipes DevOps matures

---

3. Architecture hexagonale (Ports & Adapters)

Description

L'architecture hexagonale est une **approche de conception logicielle**, pas un mode de déploiement.
Elle vise à isoler le **cœur métier** des détails techniques (base de données, API, services externes) via des **ports (interfaces)** et des **adaptateurs**.

Elle peut être implémentée :

- dans un monolithe,
- ou dans des services indépendants.

Avantages

- Forte maintenabilité
- Code métier indépendant de la technique
- Testabilité élevée
- Facilite l'évolution vers des services séparés

- Réduction du couplage

Inconvénients

- Courbe d'apprentissage
- Mise en place plus structurée dès le départ
- Peut sembler "overkill" pour des projets très simples

Cas d'usage typique

- Applications métier complexes
- Besoin d'évolutivité maîtrisée
- Projets long terme

---

4. Recommandation pour PeopleFirst (1 page max)

Contexte du projet

PeopleFirst est une application métier RH intégrant :

- gestion des congés et absences,
- gestion documentaire,
- génération de fiches de paie,
- avec des **pics de charge mensuels liés à la paie**.

Le projet s'inscrit dans une **trajectoire d'évolution progressive**, avec une équipe maîtrisant principalement **JavaScript / TypeScript** et **Python**.

Choix recommandé

Architecture hexagonale avec un monolithe applicatif TypeScript et un service Paie séparé en Python.

Justification

- Le **monolithe hexagonal** permet :
    - un développement rapide,
    - une forte cohérence métier,
    - une excellente testabilité,
    - une maintenance facilitée.

- Le **service Paie**, isolé :
    - absorbe les pics de charge fin de mois,
    - limite les risques sur le cœur applicatif,
    - peut être scalé indépendamment.
- L'architecture hexagonale :
    - réduit le couplage,
    - prépare une éventuelle évolution vers plus de services,

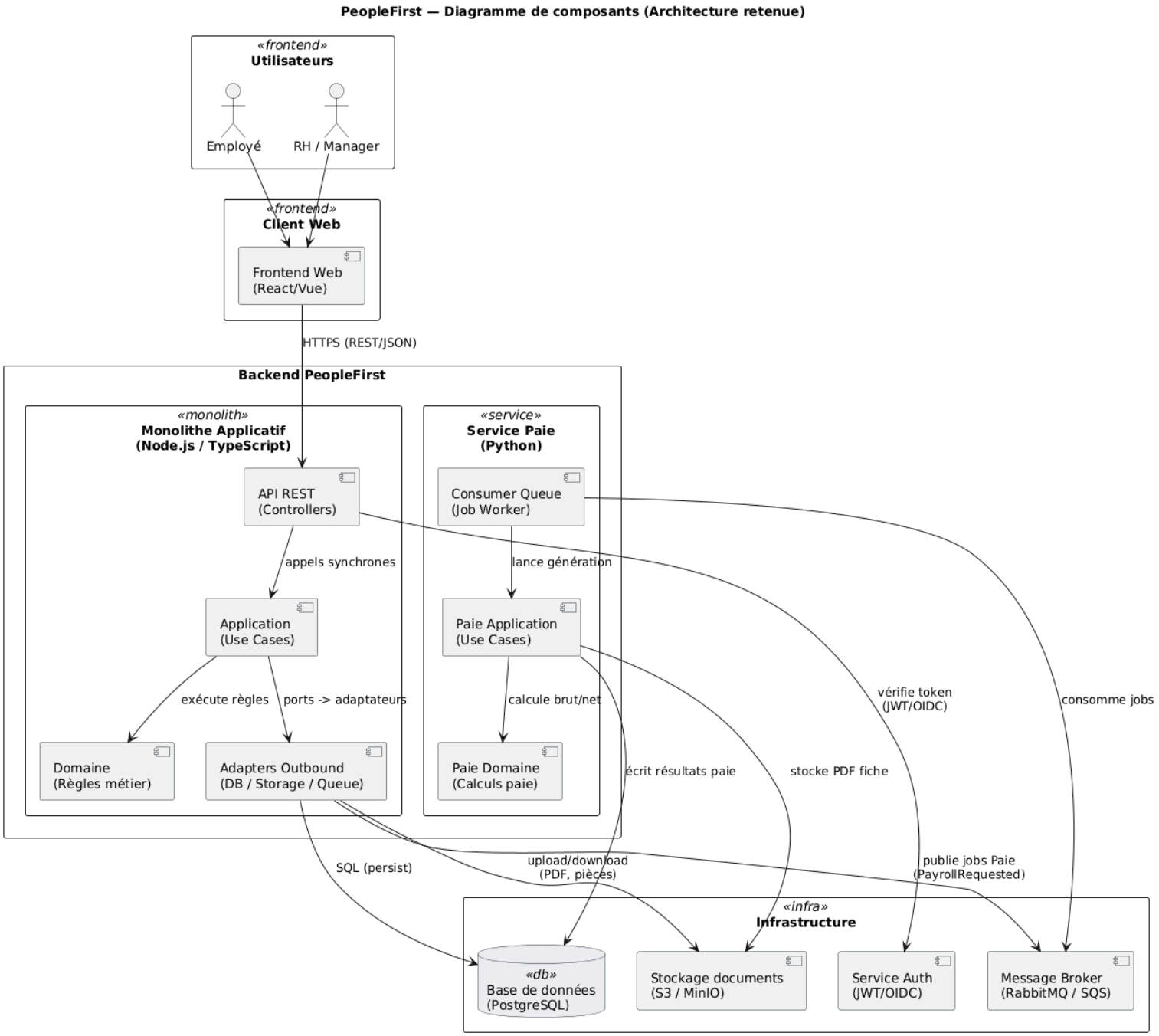
- évite la complexité prématurée des microservices.

Conclusion

Cette approche offre le **meilleur compromis** entre simplicité, robustesse et évolutivité. Elle est parfaitement adaptée aux contraintes fonctionnelles et techniques de PeopleFirst, tout en restant compatible avec une montée en charge future.

Architecture retenue

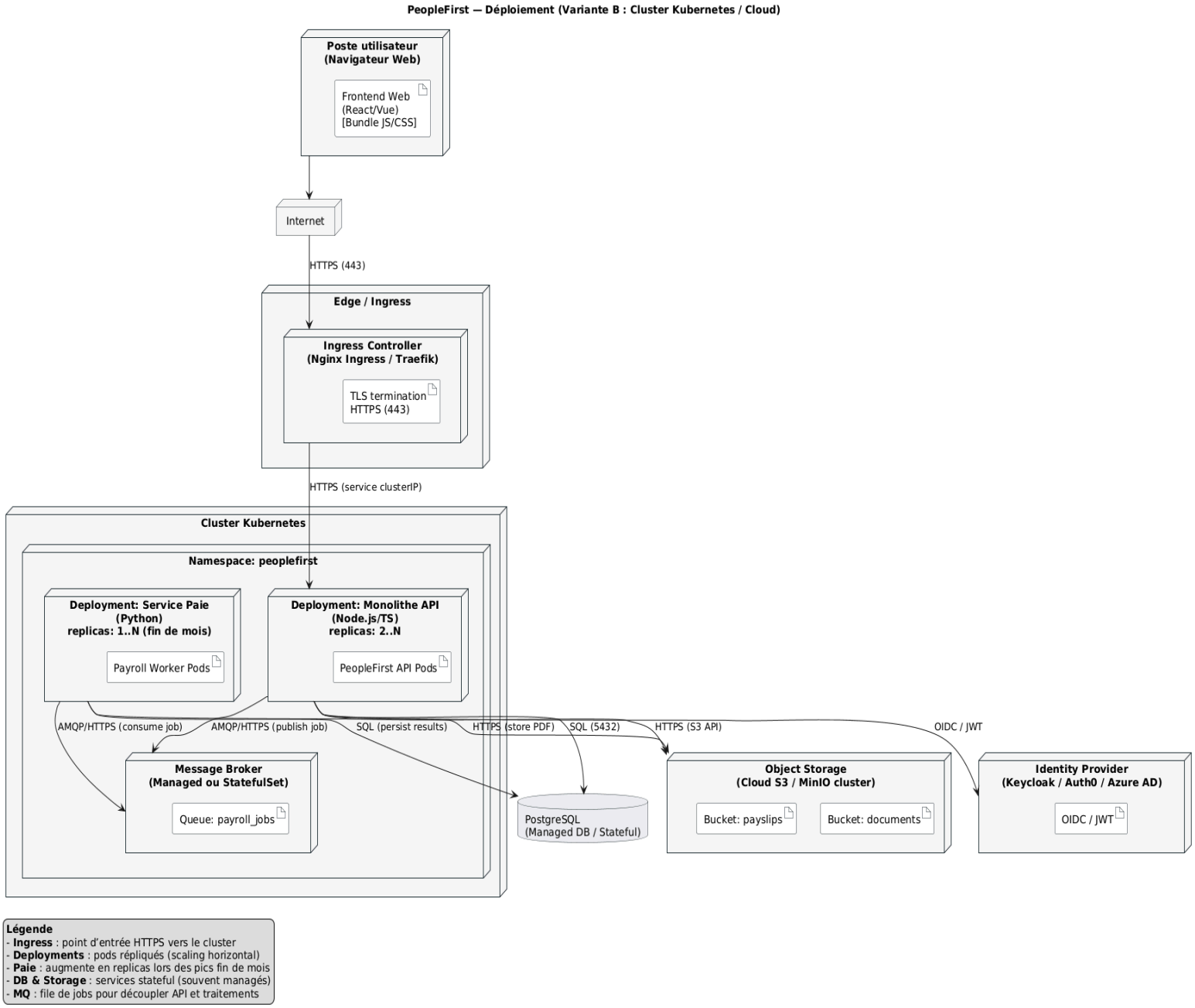
- ## Diagramme de composants avec légende



Légende
- **Frontend** : interface utilisateur (navigateur)
- **Monolithe TS** : cœur applicatif (hexa)
  * API = entrée (inbound adapters)
  * Application = cas d'usage
  * Domaine = règles métier
  * Outbound = accès DB/Queue/Storage
- **Service Paie Python** : génération paie isolée
- **Infra** :
  * DB = persistance
  * S3 = stockage documents
  * MQ = communication asynchrone
  * Auth = sécurité (JWT/OIDC)

Types de flux
- HTTPS/REST : synchro
- MQ : asynchrone (jobs/événements)

- # Diagramme de déploiement



Justification des choix structurants

Cette section présente les principaux choix techniques retenus pour l'architecture de PeopleFirst, en cohérence avec l'architecture cible basée sur un **monolithe hexagonal** complété par un **service Paie découplé**, déployé sur un **cluster**.

---

1. Choix de persistance

Solutions retenues

- **Base de données relationnelle PostgreSQL** pour les données métier structurées.
- **Stockage objet S3 compatible** (ex. MinIO ou service cloud) pour les documents et fichiers volumineux.
- **Séparation stricte entre données applicatives et fichiers**.

Justification

Le domaine RH et paie repose sur des données fortement structurées, nécessitant des **contraintes d'intégrité**, des **transactions** et une **cohérence forte**. Une base relationnelle comme PostgreSQL est donc particulièrement adaptée pour stocker les informations critiques (utilisateurs, congés, états de paie, historiques).

Les fichiers volumineux (fiches de paie PDF, documents RH) ne sont pas adaptés à un stockage en base relationnelle. L'utilisation d'un stockage objet permet :

- une **meilleure performance**,
- une **scalabilité horizontale**,
- une **réduction des coûts**,
- une meilleure compatibilité avec des environnements cloud ou conteneurisés.

Dans un contexte Kubernetes, les pods étant éphémères, toute persistance locale est proscrite. La base de données et le stockage objet constituent donc les **sources de vérité** du système.

Conséquences architecturales

- Le monolithe applicatif et le service Paie accèdent tous deux à PostgreSQL.
- Les documents générés sont déposés dans le stockage objet, indépendamment du cycle de vie des conteneurs.

2. Choix de sécurité

Solutions retenues

- **Authentification via OpenID Connect (OIDC)** avec émission de **JWT**.
- **Gestion des rôles et permissions (RBAC)** au niveau applicatif.
- **Chiffrement TLS** pour l'ensemble des communications.
- **Gestion sécurisée des secrets** (variables d'environnement, secrets Kubernetes).

Justification

La centralisation de l'authentification via un fournisseur d'identité (Keycloak, Auth0, Azure AD, etc.) permet :

- une gestion unifiée des utilisateurs,
- l'intégration possible de mécanismes avancés (SSO, MFA),
- une meilleure conformité aux bonnes pratiques de sécurité.

L'utilisation de JWT est particulièrement adaptée aux architectures distribuées, car elle permet une validation rapide et sans état côté serveur. Les règles d'autorisation (RBAC) sont appliquées dans le monolithe, garantissant une séparation stricte des droits entre employés, managers et RH, indispensable dans un contexte RH et paie.

Toutes les communications sont chiffrées via TLS afin de protéger les données sensibles en transit. Les secrets techniques (mots de passe, clés d'accès, secrets JWT) ne sont jamais stockés dans le code source et sont injectés dynamiquement par l'infrastructure.

Conséquences architecturales

- Le frontend communique avec le monolithe via HTTPS en transmettant un token JWT.
- Le service Paie n'est pas exposé publiquement et reste accessible uniquement via des mécanismes internes sécurisés.

---

3. Choix de communication

Solutions retenues

- **Communication synchrone REST/HTTPS** entre le frontend et le monolithe.

- **Communication asynchrone via un broker de messages** (file de messages) pour la paie.
- **Pattern Job / Worker** pour les traitements lourds.

Justification

Les interactions utilisateur (consultation, saisie, validation) nécessitent des réponses immédiates, ce qui justifie l'usage de communications synchrones via des API REST.

À l'inverse, la génération de la paie constitue un traitement :

- long,
- consommateur de ressources,
- sujet à des pics de charge prévisibles (fin de mois).

L'utilisation d'une file de messages permet de découpler l'API principale du traitement de paie. Les demandes sont mises en attente dans une queue et consommées par le service Paie de manière asynchrone. Cette approche améliore la **résilience**, la **scalabilité** et la **disponibilité globale** du système.

Grâce à cette architecture, le service Paie peut être répliqué horizontalement afin de traiter plusieurs jobs en parallèle, sans impacter le reste de l'application.

Conséquences architecturales

- Le monolithe publie des jobs de paie dans la file de messages.
- Le service Paie consomme ces jobs, exécute les calculs, puis stocke les résultats.
- L'utilisateur peut consulter l'état d'avancement sans bloquer l'application.

Conclusion

Les choix structurants retenus permettent de concilier **simplicité**, **robustesse** et **évolutivité**.
L'architecture repose sur des technologies éprouvées et des principes adaptés aux contraintes métier de PeopleFirst, notamment en matière de sécurité, de performance et de montée en charge liée à la paie.

Risques identifiés et mesures de mitigation

Cette section identifie les principaux risques techniques liés à l'architecture retenue et les mesures mises en place pour en limiter l'impact.

---

1. Risque de surcharge lors des pics de paie

Description
La génération des fiches de paie entraîne des pics de charge importants, concentrés sur une courte période (fin de mois), pouvant impacter la performance globale du système.

Mesures de mitigation

- Découplage du service Paie via une communication asynchrone (file de messages).
- Scalabilité horizontale du service Paie dans le cluster (réplication des pods).
- Mise en place de quotas et de limites de ressources par pod pour éviter l'effet domino.

---

2. Risque de couplage excessif entre les composants

Description
Un couplage fort entre les couches applicatives ou les services pourrait nuire à la maintenabilité et à l'évolution du système.

Mesures de mitigation

- Adoption de l'architecture hexagonale au sein du monolithe.
- Utilisation de ports et d'adaptateurs pour isoler le domaine métier.
- Interfaces contractuelles claires entre le monolithe et le service Paie.

---

3. Risque lié à la sécurité des données RH et paie

Description
Les données traitées sont sensibles (informations personnelles, données salariales), exposant le système à des risques de fuite ou d'accès non autorisé.

Mesures de mitigation

- Centralisation de l'authentification via un fournisseur d'identité (OIDC).
- Gestion fine des droits (RBAC) au niveau applicatif.

- Chiffrement systématique des échanges (TLS).
- Stockage sécurisé des secrets (pas de secrets en dur dans le code).

---

4. Risque de perte de données ou d’incohérence

Description
Une défaillance technique (panne, crash de pod) pourrait entraîner une perte ou une incohérence des données.

Mesures de mitigation

- Externalisation complète de la persistance (PostgreSQL, stockage objet).
- Sauvegardes régulières de la base de données.
- Idempotence des jobs de paie pour éviter les doublons.
- Journaux d’audit pour tracer les opérations critiques.

Conclusion globale — Architecture PeopleFirst

L’architecture retenue pour PeopleFirst repose sur un **monolithe applicatif structuré selon une architecture hexagonale**, complété par un **service Paie asynchrone** déployé indépendamment au sein d’un cluster.

Ce choix permet :

- de conserver une **forte cohérence métier**,
- de limiter la **complexité inutile** des microservices,
- d’assurer une **scalabilité ciblée** sur les traitements les plus coûteux,
- et de garantir un **haut niveau de sécurité** pour les données RH et salariales.

Grâce à une séparation claire des responsabilités, à l’utilisation de mécanismes asynchrones et à des solutions de persistance adaptées, PeopleFirst dispose d’une architecture **robuste, évolutive et maintenable**, capable d’accompagner la croissance fonctionnelle et la montée en charge du système.