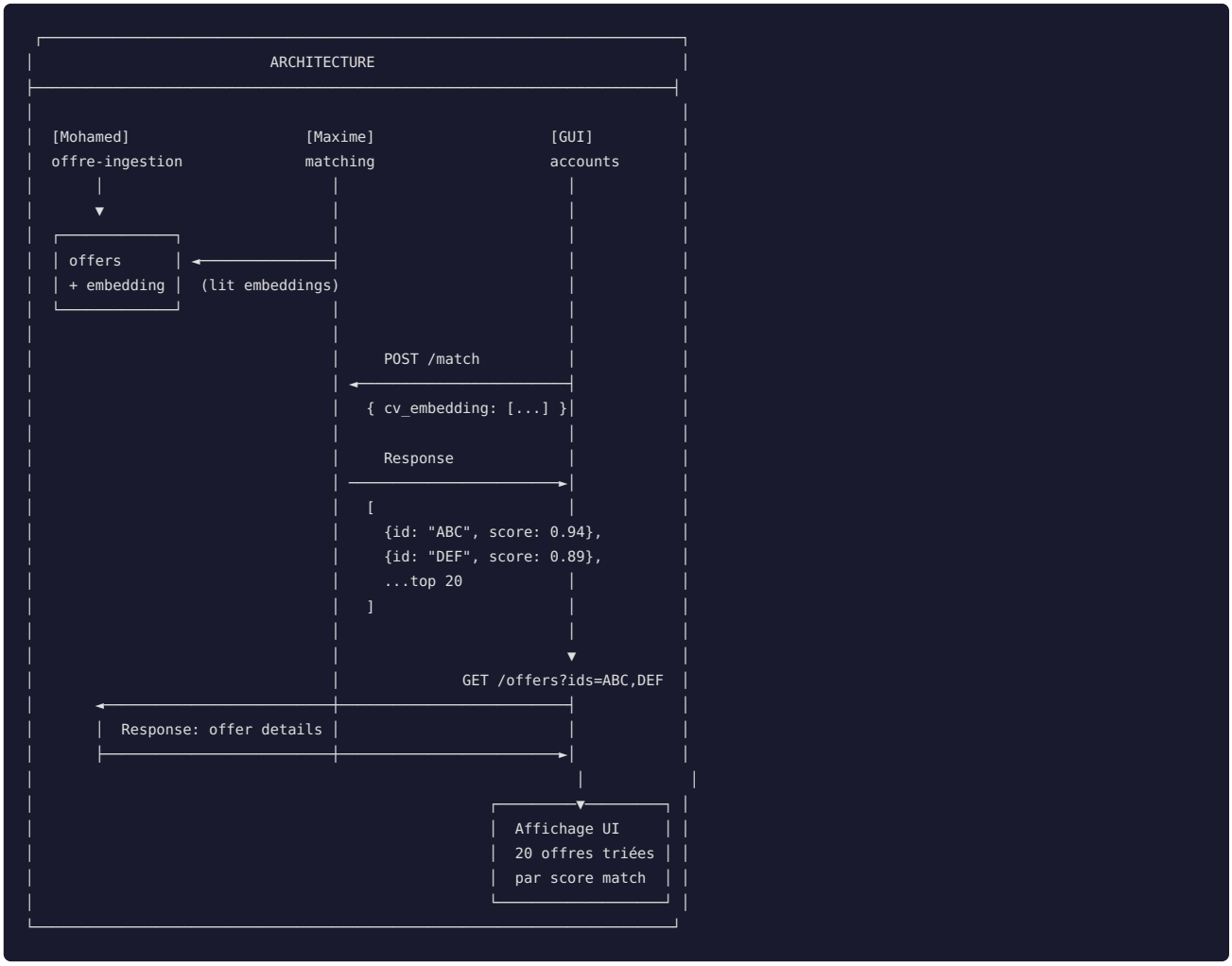
Interface GUI - Offres & Matching

Vue d'ensemble



Flow de données

1. Génération embedding CV

- **Responsable** : Maxime (dossier `shared/`)
- **Stockage** : Base GUI (champ sur CandidateProfile ou table dédiée)

2. Matching CV ↔ Offres

- **GUI → Maxime** : `POST /match` avec `cv_embedding` (vecteur)
- **Maxime** : Compare avec embeddings offres (fournis par Mohamed)
- **Maxime → GUI** : Liste `[{id, score}, ...]` (top 20)

3. Récupération détails offres

- **GUI → Mohamed** : `GET /offers?ids=ABC,DEF,...`
- **Mohamed → GUI** : Détails complets des offres

4. Affichage

- GUI affiche les offres triées par score de matching

APIs attendues

API Matching (Maxime)

```
 POST /match
Content-Type: application/json

{
  "cv_embedding": [0.123, -0.456, 0.789, ...]  // vecteur 768/1536 dimensions
}
```

Response:

```
 [
  {"id": "201VPGR", "score": 0.94},
  {"id": "201VPGQ", "score": 0.89},
  {"id": "201VPGN", "score": 0.87},
  ...
]
```

API Offres (Mohamed)

```
 GET /offers?ids=201VPGR,201VPGQ,201VPGN
```

Response:

```
 [
  {
    "id": "201VPGR",
    "intitule": "Développeur Python",
    "description": "...",
    "entreprise": "ACME Corp",
    "lieu": "Paris 75001",
    "typeContrat": "CDI",
    "salaire": "45-55K€",
    "competences": [
      {"libelle": "Python", "exigence": "E"},
      {"libelle": "Django", "exigence": "S"}
    ]
  },
  ...
]
```

Dossier shared/

Contenu actuel (piloté par Maxime)

```
 shared/
└── embedding/
```

```
    └── model.py  # Code génération embedding
```

Bonnes pratiques

Contenu	OK ?	Raison
Code partagé (fonctions, classes)		Évite duplication entre services
Modèle d'embedding (fichiers .bin)	⚠	Mieux dans volume Docker ou S3
Données (embeddings calculés)	×	Devrait être en base (PostgreSQL + pgvector)

Recommandation : Le modèle d'embedding devrait être : - Téléchargé au démarrage du container (cache Docker), ou - Stocké dans un bucket S3/MinIO partagé

Cache des résultats matching (GUI)

Option retenue : Cache en base avec lazy refresh

```
class MatchResult(models.Model):
   """Cache local des résultats de matching."""
   user = models.ForeignKey(User, on_delete=models.CASCADE)
   profile = models.ForeignKey(CandidateProfile, on_delete=models.CASCADE)
   offer_external_id = models.CharField(max_length=50)
   score = models.FloatField()
   computed_at = models.DateTimeField(auto_now=True)

   class Meta:
       unique_together = ['profile', 'offer_external_id']
       indexes = [
           models.Index(fields=['profile', '-score']),
       ]
```

Stratégie d'invalidation

Événement	Action
TTL > 24h	Re-match au prochain accès
CV modifié	Invalider cache immédiatement
Profil modifié (expériences, compétences)	Invalider cache
Nouvelles offres (Mohamed)	Lazy refresh au prochain accès

Évolution future (optionnelle)

Background job Celery pour refresh nocturne + notifications si nouveau match > 0.9

Modèle JobOffer (GUI)

Le GUI définit son propre modèle, indépendant du schéma de Mohamed :

```
class JobOffer(models.Model):
    """Local cache of job offers from offre-ingestion API."""

    # ID externe (celui de Mohamed)
    external_id = models.CharField(max_length=50, unique=True)

    # Champs principaux
    title = models.TextField()
    description = models.TextField()
    company_name = models.CharField(max_length=255, blank=True)

    # Localisation
    location = models.CharField(max_length=255)
    postal_code = models.CharField(max_length=10, blank=True)
    latitude = models.FloatField(null=True, blank=True)
    longitude = models.FloatField(null=True, blank=True)

    # Contrat
    contract_type = models.CharField(max_length=100)
    experience_required = models.CharField(max_length=100, blank=True)

    # Classification
    rome_code = models.CharField(max_length=10, blank=True)
    rome_label = models.CharField(max_length=255, blank=True)
    sector = models.CharField(max_length=255, blank=True)

    # Dates
    published_at = models.DateTimeField()
    synced_at = models.DateTimeField(auto_now=True)

    class Meta:
        ordering = ['-published_at']


class JobOfferSkill(models.Model):
    """Skills required for a job offer."""
    offer = models.ForeignKey(JobOffer, on_delete=models.CASCADE, related_name='skills')
    label = models.TextField()
    is_required = models.BooleanField(default=False)  # E=True, S=False
```

Avantages : - Nommage anglais cohérent avec le codebase GUI - Structure adaptée aux besoins UI (pas les 13 tables de Mohamed) - Cache local = moins d'appels API, UI rapide - Indépendance totale du schéma de Mohamed

---

Questions en attente

1. **Mohamed** : Quelle URL pour l'API offres ? Format de réponse ?
2. **Maxime** : Quelle URL pour l'API matching ? Dimension du vecteur embedding ?
3. **Équipe** : Notification quand nouveau match > seuil ?

Analyse critique de l'architecture actuelle

Ce qui est bien fait

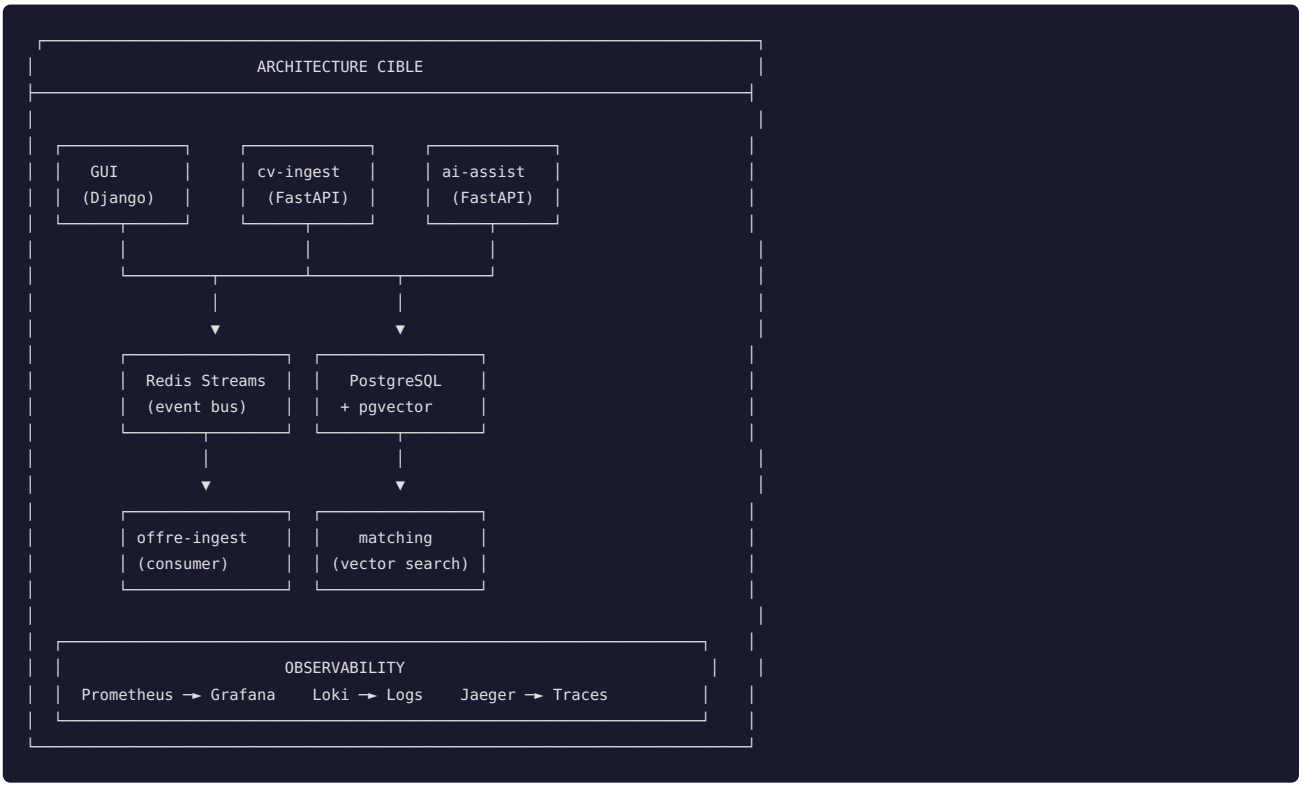
Aspect	Analyse
Microservices	Bonne séparation : `gui`, `cv-ingestion`, `ai-assistant`, `offre-ingestion`, `matching`
Docker Compose	Network isolé, variables d'environnement externalisées

shared/	Package Python installable (`pip install -e .`), interfaces typées (Pydantic)
Embeddings	Factory pattern avec providers interchangeables (sentence-transformers, vec2vec)

Points d'amélioration (niveau Data Engineer senior)

Problème	Impact	Solution
Pas de pgvector	Matching en Python = lent sur 100K+ offres	PostgreSQL + pgvector pour recherche vectorielle native
SQLite pour offres	Pas scalable, pas de concurrence	Migration PostgreSQL obligatoire
Embeddings en mémoire	Recalcul à chaque restart	Stocker en base avec index HNSW
Pas de message queue	Couplage synchrone entre services	Redis Streams ou RabbitMQ pour découplage
Pas de monitoring	Aveugle en production	Prometheus + Grafana
Pas de CI/CD	Déploiements manuels risqués	GitHub Actions + ArgoCD
Secrets en .env	Risque de commit accidentel	Vault ou cloud secrets manager
Pas de rate limiting	Vulnérable aux abus	nginx rate limit ou API Gateway

Architecture cible (niveau production)



Infrastructure Cloud recommandée

Comparatif des 3 clouds

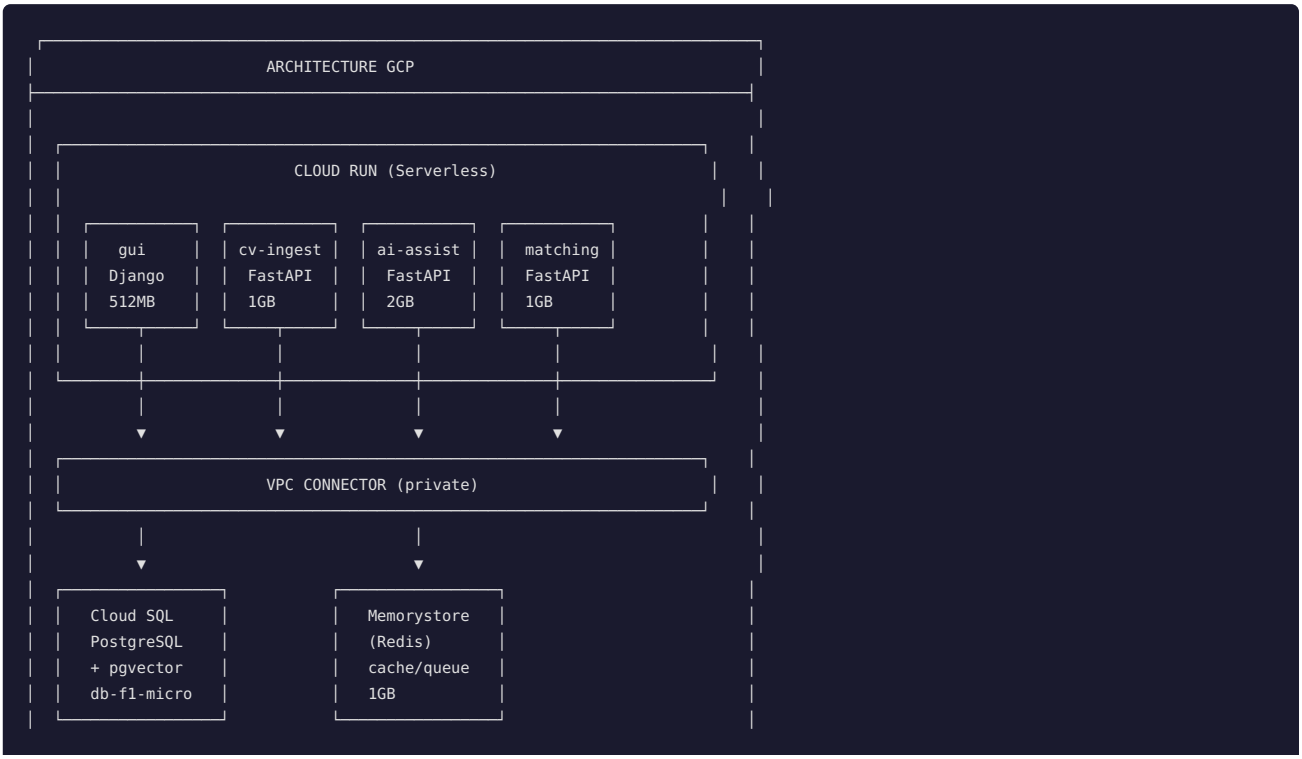
Critère	GCP	AWS	Azure
Coût startup	300$ crédits + Always Free	⚠ 12 mois Free Tier	⚠ 200$ crédits
PostgreSQL + pgvector	Cloud SQL (pgvector natif)	RDS + pgvector	Azure DB
Vector Search managé	Vertex AI Matching Engine	⚠ OpenSearch (cher)	⚠ Cognitive Search
Kubernetes	GKE Autopilot (simple)	⚠ EKS (complexe)	⚠ AKS
Serverless containers	Cloud Run (excellent)	Fargate	⚠ Container Apps
ML/Embeddings	Vertex AI	SageMaker	⚠ Azure ML
Object Storage	GCS	S3	Blob Storage

Recommandation : Google Cloud Platform (GCP)

Pourquoi GCP ? 1. **Cloud Run** : Containers serverless avec scale-to-zero (0$ quand pas de trafic) 2. **Cloud SQL** : PostgreSQL managé avec pgvector inclus 3. **Vertex AI** : Embeddings API (text-embedding-004) sans gérer de GPU 4. **BigQuery** : Analytics sur les offres (gratuit jusqu'à 1TB/mois) 5. **Crédits startup** : 300$ gratuits + programme Google for Startups

---

Architecture GCP proposée



Estimation des coûts GCP (mensuel)

Scénario 1 : MVP (< 1000 users/mois)

Service	Config	Coût/mois
Cloud Run (4 services)	512MB-2GB, scale-to-zero	~5-15$
Cloud SQL	db-f1-micro (shared CPU)	~10$
Memorystore Redis	1GB Basic	~35$
Cloud Storage	10GB	~0.25$
Vertex AI Embeddings	10K requêtes	~2$
TOTAL MVP		**~50-60$/mois**

Scénario 2 : Growth (10K+ users/mois)

Service	Config	Coût/mois
Cloud Run (4 services)	1-4GB, min instances=1	~80-150$
Cloud SQL	db-custom-2-4096 (2 vCPU)	~70$
Memorystore Redis	5GB Standard	~150$
Cloud Storage	100GB	~2.5$
Vertex AI Embeddings	100K requêtes	~20$
BigQuery	100GB stockage	~2.5$
TOTAL Growth		**~300-400$/mois**

Optimisations coût

1. **Committed Use Discounts** : -30% sur Cloud SQL si engagement 1 an
2. **Cloud Run scale-to-zero** : 0$ la nuit/weekend si pas de trafic

3. **Preemptible/Spot VMs** : -60% pour les jobs batch (embeddings)
4. **AlloyDB** : Alternative Cloud SQL si besoin de perf vectorielle extrême

---

Migration vers GCP - Roadmap

Phase 1 : Foundation (Semaine 1-2)

```
 # 1. Créer projet GCP
gcloud projects create jobmatch-prod --name="JobMatch Production"

# 2. Activer APIs
gcloud services enable \
  run.googleapis.com \
  sqladmin.googleapis.com \
  redis.googleapis.com \
  storage.googleapis.com \
  aiplatform.googleapis.com

# 3. Créer Cloud SQL avec pgvector
gcloud sql instances create jobmatch-db \
  --database-version=POSTGRES_15 \
  --tier=db-f1-micro \
  --region=europe-west1 \
  --database-flags=cloudsql.enable_pgvector=on
```

Phase 2 : Deploy services (Semaine 3-4)

```
 # cloudbuild.yaml
steps:
  - name: 'gcr.io/cloud-builders/docker'
    args: ['build', '-t', 'gcr.io/$PROJECT_ID/gui', '-f', 'app/gui/Dockerfile', '.']
  - name: 'gcr.io/cloud-builders/docker'
    args: ['push', 'gcr.io/$PROJECT_ID/gui']
  - name: 'gcr.io/google.com/cloudsdktool/cloud-sdk'
    entrypoint: gcloud
    args:
      - 'run'
      - 'deploy'
      - 'gui'
      - '--image=gcr.io/$PROJECT_ID/gui'
      - '--region=europe-west1'
      - '--allow-unauthenticated'
      - '--set-env-vars=DATABASE_URL=$$DATABASE_URL'
```

Phase 3 : pgvector migration (Semaine 5)

```
 -- Activer pgvector
CREATE EXTENSION IF NOT EXISTS vector;

-- Table offres avec embedding
CREATE TABLE offers (
    id VARCHAR(50) PRIMARY KEY,
    title TEXT,
    description TEXT,
    embedding vector(768),  -- dimension du modèle
    -- autres champs...
);

-- Index HNSW pour recherche rapide
CREATE INDEX ON offers USING hnsw (embedding vector_cosine_ops);

-- Recherche des 20 meilleures offres
SELECT id, title, 1 - (embedding <=> $1) as score
FROM offers
ORDER BY embedding <=> $1
```

```
LIMIT 20;
```

Phase 4 : Vertex AI embeddings (Semaine 6)

```
 # Remplacer sentence-transformers local par Vertex AI
from google.cloud import aiplatform

def get_embedding(text: str) -> list[float]:
    """Generate embedding using Vertex AI."""
    model = aiplatform.TextEmbeddingModel.from_pretrained("text-embedding-004")
    embeddings = model.get_embeddings([text])
    return embeddings[0].values  # 768 dimensions
```

Alternatives selon le budget

Budget serré (< 30$/mois)

```
┌─────────────────────────────────────┐
│  Railway.app ou Render.com          │
│  ─────────────────────────────────  │
│  • PostgreSQL inclus (pgvector)     │
│  • Docker containers                │
│  • Auto-deploy from GitHub          │
│  • ~20-30$/mois pour 4 services     │
└─────────────────────────────────────┘
```

Budget moyen (50-100$/mois) - Recommandé

```
┌─────────────────────────────────────┐
│  GCP Cloud Run + Cloud SQL          │
│  ─────────────────────────────────  │
│  • Scale-to-zero                    │
│  • PostgreSQL + pgvector            │
│  • Vertex AI pour embeddings        │
│  • ~50-80$/mois                     │
└─────────────────────────────────────┘
```

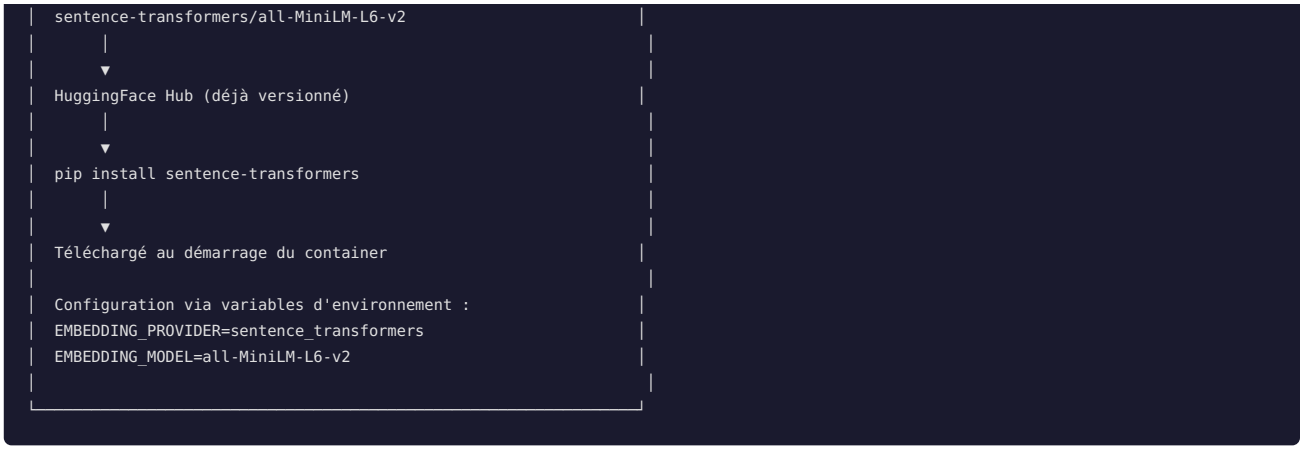
Budget confortable (200-500$/mois)

```
┌─────────────────────────────────────┐
│  GCP GKE Autopilot                  │
│  ─────────────────────────────────  │
│  • Kubernetes managé                │
│  • Scaling horizontal auto          │
│  • AlloyDB pour vector search       │
│  • Monitoring complet               │
│  • ~300-500$/mois                   │
└─────────────────────────────────────┘
```

Stratégie ML & Embeddings

Phase actuelle : Modèle pré-entraîné (MVP)

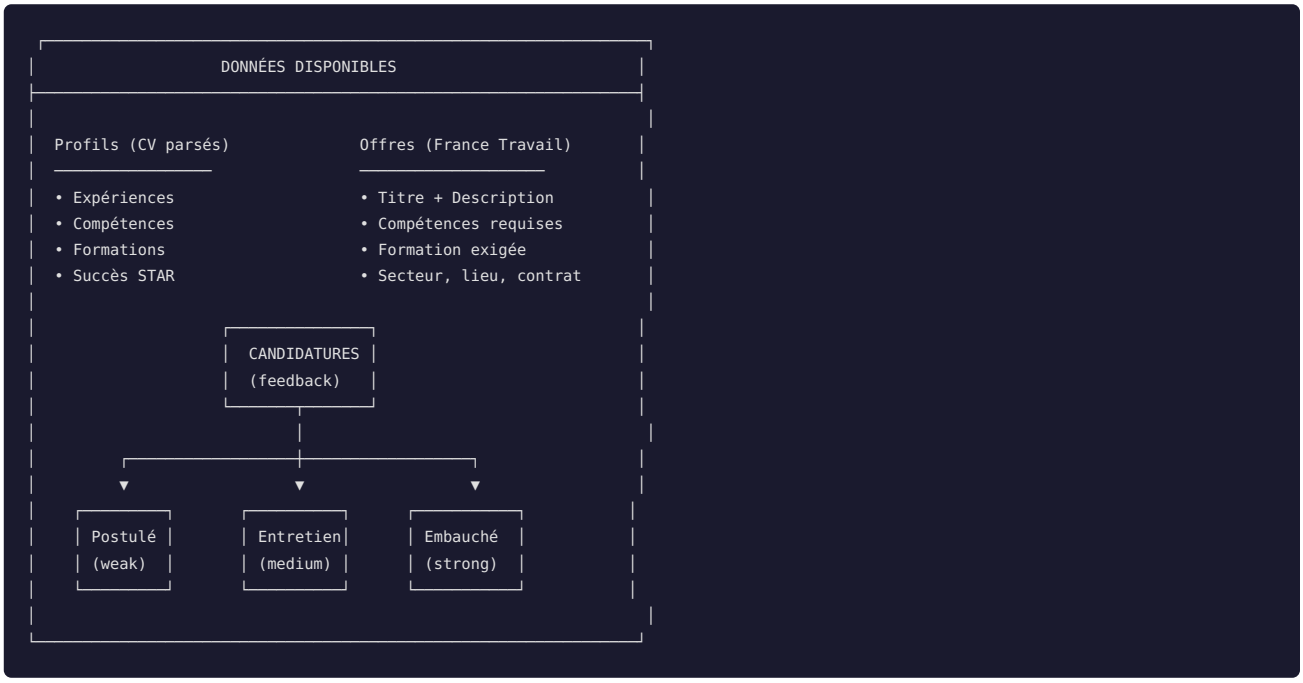
```
┌─────────────────────────────────────────────────────┐
│  APPROCHE MVP - Pas de MLflow nécessaire            │
├─────────────────────────────────────────────────────┤
│                                                     │
```



Pourquoi pas MLflow maintenant ? - Modèles pré-entraînés = pas d'expériences à tracker - Pas de fine-tuning = pas de versions custom à gérer - HuggingFace Hub versionne déjà les modèles

Phase future : Fine-tuning sur données de candidature

Dataset potentiel



Types de fine-tuning possibles

Approche	Description	Dataset format
Embedding fine-tuning	Contrastive learning (triplet loss)	`(cv, offre_positive, offre_negative)`
Cross-encoder	Reranker après vector search	`(cv, offre, score: 0-1)`
Learning to Rank	Ranking global par utilisateur	`{user, [(offre, relevance), ...]}`

Exemple : Fine-tuning embedding (Contrastive Learning)

```
 # Dataset format: (anchor, positive, negative)
training_data = [
    {
        "anchor": "Développeur Python 5 ans, Django, API REST",  # CV
```

```
        "positive": "Dev Python Senior - Django - Paris",        # Offre où postulé
        "negative": "Comptable junior - Excel - Lyon"            # Offre non pertinente
    },
    # ...
]

# Modèle: sentence-transformers fine-tuné
# Loss: MultipleNegativesRankingLoss ou TripletLoss
from sentence_transformers import SentenceTransformer, losses

model = SentenceTransformer('all-MiniLM-L6-v2')
train_loss = losses.MultipleNegativesRankingLoss(model)
# ... training loop
```

Volume de données nécessaire

Approche	Minimum viable	Idéal
Embedding fine-tuning	~5K paires	~50K paires
Cross-encoder	~10K exemples	~100K exemples
Learning to Rank	~1K users × 10 offres	~10K users × 50 offres

Quand MLflow devient pertinent

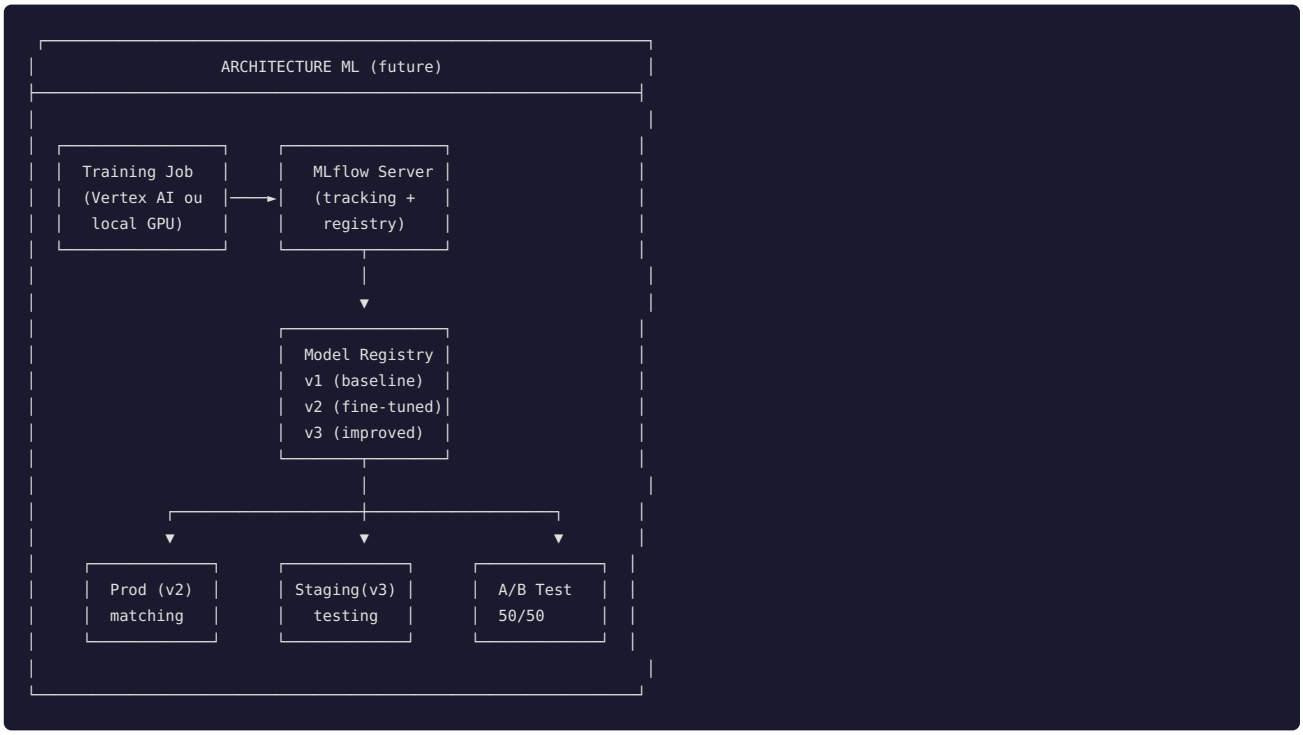
```
┌─────────────────────────────────────────────────────────┐
│  AVEC fine-tuning sur vos données → MLflow fait sens    │
├─────────────────────────────────────────────────────────┤
│                                                         │
│  1. EXPERIMENT TRACKING                                 │
│     ──────────────────                                  │
│     • Comparer base model vs fine-tuned                 │
│     • Hyperparams: learning rate, epochs, batch size    │
│     • Métriques: MRR, NDCG, Recall@20                   │
│                                                         │
│  2. MODEL REGISTRY                                      │
│     ──────────────                                      │
│     • v1: all-MiniLM-L6-v2 (baseline)                   │
│     • v2: fine-tuned sur 1K candidatures                │
│     • v3: fine-tuned sur 10K candidatures + feedback    │
│     • Staging → Production workflow                     │
│                                                         │
│  3. A/B TESTING                                         │
│     ──────────                                          │
│     • 50% users: model v2                               │
│     • 50% users: model v3                               │
│     • Mesurer: taux de clic, taux de candidature        │
│                                                         │
└─────────────────────────────────────────────────────────┘
```

Roadmap ML

Phase	Action	Données requises	MLflow ?
MVP	Modèle pré-entraîné (MiniLM, Vertex AI)	Aucune	× Non
V1	Collecter candidatures	6 mois d'usage	× Non
V2	Fine-tune embedding sur candidatures	~5K paires	Oui
V3	A/B test base vs fine-tuned	Métriques prod	Oui
V4	Cross-encoder reranker	~10K exemples	Oui

v4	Cross-encoder reranker	~10K exemples	Oui

Architecture ML avec MLflow (Phase V2+)



Métriques à collecter (dès maintenant)

Pour préparer le fine-tuning futur, commencer à logger :

```
 # Dans le GUI, tracker les interactions utilisateur
class OfferInteraction(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    offer_external_id = models.CharField(max_length=50)
    match_score = models.FloatField()  # Score du matching initial

    # Interactions (feedback implicite)
    viewed = models.BooleanField(default=False)
    viewed_at = models.DateTimeField(null=True)
    time_spent_seconds = models.IntegerField(default=0)

    # Actions (feedback explicite)
    saved = models.BooleanField(default=False)
    applied = models.BooleanField(default=False)
    applied_at = models.DateTimeField(null=True)

    # Outcome (feedback fort - si disponible)
    got_interview = models.BooleanField(null=True)
    got_hired = models.BooleanField(null=True)

    class Meta:
        unique_together = ['user', 'offer_external_id']
```

Ces données permettront de construire le dataset de fine-tuning quand le volume sera suffisant.

Checklist "Top niveau Data Engineer"

Code & Architecture

- [ ] **pgvector** : Recherche vectorielle en SQL, pas en Python

- [ ] **Pydantic v2** : Validation des schémas d'API
- [ ] **OpenAPI/Swagger** : Documentation auto des APIs
- [ ] **Async everywhere** : FastAPI async, asyncpg pour PostgreSQL
- [ ] **Dependency injection** : Facilite les tests

Data Pipeline

- [ ] **Medallion architecture** : Bronze → Silver → Gold (déjà fait pour offres)
- [ ] **Idempotent jobs** : Re-run sans effets de bord
- [ ] **Data lineage** : Traçabilité des transformations
- [ ] **Schema registry** : Versioning des schémas de données

DevOps & Observability

- [ ] **IaC (Terraform/Pulumi)** : Infrastructure as Code
- [ ] **GitOps (ArgoCD)** : Déploiements déclaratifs
- [ ] **Prometheus + Grafana** : Métriques custom
- [ ] **Structured logging** : JSON logs avec correlation IDs
- [ ] **Distributed tracing** : Jaeger/OpenTelemetry

Sécurité

- [ ] **Secrets management** : GCP Secret Manager, pas de .env
- [ ] **Service accounts** : Principe du moindre privilège
- [ ] **VPC + Private IPs** : Pas d'exposition publique des DBs
- [ ] **IAM policies** : Accès granulaires par service

References

- interface_gui_offers.md - Analyse base SQLite offers.db
- GCP Cloud Run Documentation
- pgvector GitHub
- Vertex AI Embeddings
- POSTMORTEM.md - Historique des décisions techniques