

Ajouter un nouveau provider IA

1. Déclarer le provider et ses modèles

Dans `constantsVariable.js` :

```
const AI_PROVIDERS = {
  GEMINI: "gemini",
  OPENAI: "openai",
};

const AI_MODELS = {
  GEMINI_GENERATION: "gemini-3.5-flash-lite",
  GEMINI_EMBEDDING: "gemini-embedding-001",

  OPENAI_GENERATION: "gpt-example",
  OPENAI_EMBEDDING: "text-embedding-example",
};

const AI_PRICE_VERSIONS = {
  GEMINI: "vertex-global-standard-2026-09-29",
  OPENAI: "openai-standard-YYYY-MM-DD",
};
```

Configurer ensuite le provider actif dans `AI_CONFIG` :

```
const AI_CONFIG = {
  AI_PROVIDER: AI_PROVIDERS.OPENAI,
  AI_EMBEDDING_PROVIDER: AI_PROVIDERS.OPENAI,
```

```
AI_MODEL: AI_MODELS.OPENAI_GENERATION,  
AI_EMBEDDING_MODEL: AI_MODELS.OPENAI_EMBEDDING,  
  
OPENAI_API_KEY: process.env.OPENAI_API_KEY,  
};
```

Ne pas redéclarer les noms de modèles dans les fichiers du provider. Toujours importer `AI_MODELS` et `AI_PRICE_VERSIONS`.

“ **Note sur les embeddings** : si on conserve toujours le provider et le modèle d'embedding déjà existants, toute la partie « nouveau provider d'embedding » peut être ignorée. Le nouveau provider peut servir uniquement à la génération, tandis que `AI_EMBEDDING_PROVIDER` et `AI_EMBEDDING_MODEL` restent configurés sur l'existant.

2. Créer le dossier du provider

Créer la structure suivante :

```
server/ai/provider/openai/  
├─ generation_provider.js  
├─ embedding_provider.js  
├─ tokenizer.js  
├─ pricing.js  
└─ context_cache.js
```

`context_cache.js` est facultatif si le provider ne supporte pas le cache. `embedding_provider.js` est également facultatif si on garde le provider d'embedding existant.

3. Implémenter le provider de génération

Le contrat se trouve dans :

```
server/ai/contract/generation_provider_contract.js
```

Le provider doit exposer les méthodes suivantes :

```
class OpenAIProvider {
  constructor(config, dependencies = {}) {
    this.name = AI_PROVIDERS.OPENAI;
    this.config = config;
    this.client = null;
  }

  async initialize() {}

  isReady() {}

  async countTokens(messages, options) {}

  async generate(request) {}
}
```

Format normalisé des messages

```
{
  role: "user" | "assistant",
  content: [
    {
      type: "text",
      text: "Bonjour"
    },
    {
      type: "image",
      mimeType: "image/png",
      data: "base64..."
    }
  ]
}
```

Le provider doit convertir ce format vers celui de son SDK.

Requête reçue par `generate()`

```
{
  operation,
  scope,
  prefix,
  messages,
  temperature,
  maxOutputTokens,
  thinkingLevel,
  inputTokenLimit,
  tokenCountTimeoutMs,
  timeoutMs,
  usage
}
```

Réponse normalisée attendue

```
{
  text: "Réponse du modèle"
}
```

Aucun objet propre au SDK ne doit sortir du provider.

4. Implémenter le tokenizer

Créer `provider/openai/tokenizer.js` :

```
class OpenAITokenizer {
  constructor(client) {
    this.client = client;
  }

  async count(messages, options) {
    // Retourner un nombre entier de tokens.
  }
}
```

```

normalizeUsage(providerUsage) {
  return {
    inputTokens: 0,
    cachedInputTokens: 0,
    outputTokens: 0,
    reasoningTokens: 0,
  };
}
}

```

Les champs propres au SDK doivent rester dans ce fichier.

Exemples de champs qui ne doivent pas sortir du tokenizer :

```

promptTokenCount
candidatesTokenCount
input_tokens
output_tokens
cached_tokens

```

Le reste de l'application doit uniquement utiliser :

```

{
  inputTokens,
  cachedInputTokens,
  outputTokens,
  reasoningTokens
}

```

5. Implémenter le pricing

Créer `provider/openai/pricing.js`.

Importer les constantes partagées :

```

const {
  AI_PROVIDERS,
  AI_MODELS,

```

```
AI_PRICE_VERSIONS,  
} = require('../../../../constantsVariable');
```

Implémenter le pricing :

```
class OpenAIPricing {  
  constructor(config) {  
    this.config = config;  
    this.priceVersion = AI_PRICE_VERSIONS.OPENAI;  
  }  
  
  reservationCents(operation) {  
    // Calculer le coût maximal pouvant être facturé.  
  }  
}
```

Le fichier peut aussi exposer :

```
function generationCost(usage) {}  
  
function generationCeiling(inputTokens, outputTokens) {}  
  
// Ajouter des opérations tarifées uniquement si le provider les utilise.
```

Les calculs doivent recevoir les statistiques normalisées, jamais les données brutes du SDK.

Un modèle sans pricing explicitement validé doit être rejeté.

6. Implémenter les embeddings (facultatif)

“ Cette étape peut être ignorée si on utilise toujours le même embedding déjà existant. Dans ce cas, il n'est pas nécessaire d'implémenter un nouveau provider d'embedding ni de modifier la factory d'embedding.

Si un nouveau provider d'embedding est réellement nécessaire, le contrat se trouve dans :

```
server/ai/contract/embedding_provider_contract.js
```

Le provider doit exposer :

```
class OpenAIEmbeddingProvider {  
    constructor(config, dependencies = {}) {}  
  
    async initialize() {}  
  
    isReady() {}  
  
    assertInput(text) {}  
  
    async embed(text, usage) {}  
}
```

`embed()` doit retourner uniquement le vecteur :

```
[  
  0.012,  
  -0.034,  
  0.056  
]
```

`assertInput()` doit vérifier :

- Le modèle autorisé.
- La taille maximale du texte.
- La limite de tokens ou d'octets.
- La politique de coût applicable.

“ Changer de modèle d'embedding nécessite généralement de réindexer toute la collection Chroma.

7. Enregistrer le provider de génération

Modifier :

```
server/ai/factory/generation_provider_factory.js
```

Importer le provider :

```
const {  
  OpenAIProvider,  
} = require("../provider/openai/generation_provider");
```

Ajouter le cas dans la factory :

```
switch (config.AI_PROVIDER) {  
  case AI_PROVIDERS.GEMINI:  
    provider = new GeminiProvider(config, dependencies);  
    break;  
  
  case AI_PROVIDERS.OPENAI:  
    provider = new OpenAIProvider(config, dependencies);  
    break;  
  
  default:  
    throw new Error(  
      `Unsupported AI provider: ${config.AI_PROVIDER}`  
    );  
}
```

La factory valide automatiquement le contrat avant de retourner le provider.

8. Enregistrer le provider d'embedding (facultatif)

Ignorer cette étape si on garde le provider d'embedding existant. Le provider de génération et celui d'embedding peuvent être différents.

Modifier :

```
server/ai/factory/embedding_provider_factory.js
```

Ajouter :

```
case AI_PROVIDERS.OPENAI:
    return assertEmbeddingProviderContract(
        new OpenAIEmbeddingProvider(config, dependencies)
    );
```

Exemple si OpenAI sert à la génération mais Gemini reste utilisé pour les embeddings :

```
AI_PROVIDER: AI_PROVIDERS.OPENAI,
AI_EMBEDDING_PROVIDER: AI_PROVIDERS.GEMINI,
```

9. Enregistrer le pricing

Modifier :

```
server/ai/factory/pricing_factory.js
```

Ajouter :

```
case AI_PROVIDERS.OPENAI:
    return assertPricingContract(
        new OpenAIPricing(config)
    );
```

10. Ajouter les tests

Vérifier au minimum :

- L'initialisation du SDK.
- Le contrat de génération.
- Le contrat d'embedding, uniquement si un nouveau provider d'embedding est ajouté.
- La conversion des messages normalisés.
- La conversion des images.
- Le comptage des tokens.
- La normalisation des statistiques d'utilisation.
- Le calcul des coûts.
- Les limites d'entrée.
- Les timeouts.
- Les réponses sans métadonnées.
- Le rejet des modèles sans pricing.
- Le rejet des providers inconnus.

Exécuter les tests AI :

```
npm run test:ai
```

Checklist finale

- Provider ajouté dans `AI_PROVIDERS`.
- Modèles ajoutés dans `AI_MODELS`.
- Version tarifaire ajoutée dans `AI_PRICE_VERSIONS`.
- Dossier créé sous `ai/provider/<provider>/`.
- Contrat de génération respecté.
- Tokenizer spécifique isolé.

- Statistiques d'utilisation normalisées.
- Pricing explicitement validé.
- Contrat d'embedding respecté uniquement si un nouveau provider d'embedding est nécessaire.
- Provider enregistré dans les factories nécessaires.
- Aucun objet SDK ne sort du provider.
- Aucun champ propre au SDK n'est utilisé par le serveur générique.
- Tests AI réussis.
- Collection vectorielle réindexée uniquement si le modèle d'embedding change.

Provider actuellement implémenté et règles de coût

Seul Gemini est enregistré dans les factories. Les exemples OpenAI ci-dessus décrivent un ajout possible, pas un provider déjà disponible. Le modèle de génération configuré est `gemini-3.5-flash-lite`, l'embedding reste `gemini-embedding-001`. Gemini utilise `@google/genai` via Vertex AI et `vertex_client.js`; `GOOGLE_CLOUD_PROJECT` est requis et la politique tarifaire actuelle impose `GOOGLE_CLOUD_LOCATION=global`.

`generation_provider.js` adapte les messages normalisés et le SDK; `generator.js` compte l'entrée, applique la limite et enregistre l'usage; `tokenizer.js` normalise `input/cachedInput/output/reasoning`. Le cache actuel est implicite : il n'y a plus de gestionnaire de cache de contexte explicite ni de coût de stockage associé dans ce provider.

Une nouvelle intégration doit enregistrer chaque opération payante avec `UsageMeter.begin()`, puis `complete()` lorsque l'usage est exploitable ou `failed()` sur erreur. `embed(text, usage)` accepte le meter à la recherche; l'ingestion peut appeler `embed(text)`. Les plafonds doivent inclure les tokens de raisonnement facturables. L'usage absent ou non pris en charge reste estimé au plafond réservé; les coûts ne doivent jamais être inventés à partir d'un usage manquant.

Les tests sont maintenant sous `test/ai/`. Depuis le backend : `npm run test:ai`, `npm run test:ai:coverage` ou `npm test -- --runInBand` pour toutes les suites.

Revision #4

Created 2026-09-17 20:31:12 UTC by Mathis

Updated 2026-09-30 14:41:53 UTC by Mathis