

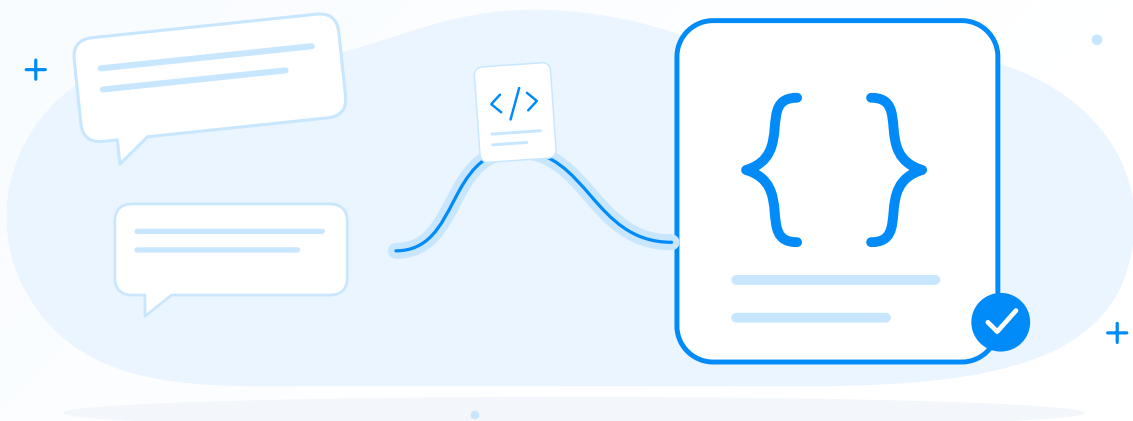
Tutoriel

LLM · Prompt engineering · Data

Extraire des données structurées avec un LLM

Transformer des tickets support en objets JSON validables avec un prompt contraint, puis mesurer les erreurs sur un petit jeu de cas.

Auteur **Youness Boutyour**

**DURÉE**

55 minutes

NIVEAU

Débutant avancé

FORMAT

Tutoriel pratique

LLM · Prompt engineering · Data · publié le 2026-05-23 · mock

Objectifs et livrables

Les tickets support arrivent sous forme de phrases, alors qu'un traitement Data attend des champs stables. Nous allons construire un extracteur qui transforme ces textes en objets contenant une catégorie, une priorité et une liste d'entités.

Nous partirons de quelques tickets pour définir précisément ce que chaque champ représente. Vous écrirez la consigne, récupérerez les réponses en mode mock ou avec un modèle, puis les passerez dans un validateur Python. Les essais feront apparaître les valeurs autorisées, les champs manquants et les types incorrects.

La lecture des erreurs permettra de comprendre ce qu'une réponse JSON garantit et ce qu'il reste à vérifier. Une structure peut être lisible par Python tout en contenant une catégorie inconnue ou une information mal extraite ; le contrat et les exemples serviront à distinguer ces situations.

Vous obtiendrez `outputs/tickets.jsonl` et un compteur d'erreurs pour suivre le traitement. Le mode `--mock` rend le parcours exécutable sans clé API, ce qui vous permettra de vérifier le validateur avant de comparer les réponses de différents modèles.

Faire respecter un contrat de sortie

Schéma de l'extraction JSON : ticket libre, contrat de champs, validation syntaxique et métier, puis fichier JSONL.



● forme valide ≠ valeur correcte · erreurs conservées

prompt-json-structure

Un JSON bien formé doit encore respecter les valeurs et le sens attendus par le métier.

Lire le flux avant de coder

Le schéma distingue trois contrôles successifs. Le prompt demande une forme, le parseur vérifie que la réponse est du JSON, puis la validation métier vérifie les catégories, priorités et entités réellement autorisées. Passer le premier contrôle ne garantit donc pas qu'un ticket est correctement compris.

Le mode mock permet de provoquer volontairement un champ manquant, une catégorie inconnue ou un type incorrect. Ces cas servent à vérifier que le programme conserve l'erreur et n'invente pas une valeur de remplacement avant d'utiliser le fichier JSONL.

1. Préparer l'environnement

Commencez dans un dossier réservé à cet exemple : les chemins `data/`, `src/` et `outputs/` sont relatifs à ce dossier. L'environnement virtuel permet d'installer `jsonschema` sans modifier les bibliothèques des autres projets. Les fichiers complets de l'archive constituent la version exécutable ; les extraits ci-dessous isolent les parties à examiner.

Le fichier JSONL contient un objet JSON par ligne, ce qui permet de traiter les tickets indépendamment. La création manuelle du fichier et le générateur Python de l'étape suivante sont deux façons de préparer les mêmes cinq entrées : utilisez l'une ou l'autre, sans les concaténer. Vérifiez d'abord que l'import et la lecture du fichier fonctionnent avant d'introduire un appel au modèle.

```
Terminal · Bash

mkdir -p prompt-json-structure/{data,src,outputs}
cd prompt-json-structure
python3 -m venv .venv && source .venv/bin/activate
python -m pip install jsonschema
```

Sortie obtenue

Sortie / exemple

Successfully installed jsonschema

Commande de vérification

```
Terminal · Bash

python -c "import jsonschema; print(jsonschema.__version__)"
```

Code de l'étape

Créez `data/tickets.jsonl` avec une ligne JSON par ticket :

● ● ● JSON Lines

```
{"text": "Panne intermittente de l'API du compte étudiant depuis 08:30"}
{"text": "Je demande un accès au laboratoire de vision pour mon projet"}
{"text": "La facture de formation du mois dernier est manquante"}
{"text": "Question générale sur les horaires du campus"}
{"text": "Panne réseau du campus, impossible de joindre le service"}
```

2. Préparer cinq tickets

Les cinq tickets constituent un petit jeu de contrôle, pas un échantillon représentatif de toutes les demandes support. Lisez-les avant de lancer l'extraction et notez, pour chacun, la catégorie, la priorité et les entités que vous considérez correctes. Cette lecture permettra de distinguer une erreur du modèle d'une ambiguïté dans la consigne.

Le champ `text` conserve le message original, tandis que les champs extraits seront écrits séparément dans les résultats. `ensure_ascii=False` rend les accents lisibles dans les fichiers, sans changer leur sens. Gardez le même ordre d'entrées lors des comparaisons : une prédiction associée au mauvais ticket peut être parfaitement valide en JSON et néanmoins inutilisable.

Créez `data/tickets.jsonl` avec cinq tickets réalistes : une panne d'API liée à un compte, une demande d'accès, une facture manquante, une question générale et une panne réseau. Cette diversité permet d'observer les catégories, les priorités et les entités extraites.

Code de l'étape

● ● ● Python

```
from pathlib import Path

tickets = [
    "Panne intermittente de l'API du compte étudiant depuis 08:30",
    "Je demande un accès au laboratoire de vision pour mon projet",
    "La facture de formation du mois dernier est manquante",
    "Question générale sur les horaires du campus",
    "Panne réseau du campus, impossible de joindre le service",
]

import json
Path("data/tickets.jsonl").write_text(
    "".join(json.dumps({"text": text}, ensure_ascii=False) + "\n" for text in tickets),
```

```
encoding="utf-8",
)
assert len(Path("data/tickets.jsonl").read_text().splitlines()) == 5
```

Sortie obtenue

Sortie / exemple

5 lignes dans data/tickets.jsonl

Commande de vérification

Terminal · Bash

```
wc -l data/tickets.jsonl
```

3. Contraindre et valider la sortie

La validation se déroule en plusieurs temps. `json.loads` vérifie que la réponse peut être décodée ; `jsonschema.validate` contrôle ensuite sa structure. Enfin, les règles locales vérifient les valeurs autorisées et les clés présentes. Le schéma montré ici n'impose pas à lui seul les catégories admises ni l'absence de clés supplémentaires : ces contraintes sont complétées par `validate_item`.

Une réponse acceptée respecte donc le contrat de données, mais sa classification peut encore être fausse. Commencez avec `--mock` pour vérifier ce contrat sans dépendre d'un service externe, puis gardez les mêmes tickets en mode réel. Lorsqu'une réponse est rejetée, examinez le champ en cause avant de modifier le schéma : l'assouplir pour accepter une erreur masquerait le problème.

Dans `src/extract.py`, définissez un schéma JSON et validez chaque réponse :

Python

```
import json
from jsonschema import validate

SCHEMA = {"type": "object", "required": ["category", "priority", "entities"],
          "properties": {"category": {"type": "string"},
                          "priority": {"enum": ["low", "medium", "high"]},
                          "entities": {"type": "array", "items": {"type": "string"}}}}

def accept(raw: str) → dict:
```

```
item = json.loads(raw)
validate(item, SCHEMA)
return item
```

Ajoutez l'appel à votre endpoint LLM ou le mode `--mock`, puis écrivez uniquement les objets acceptés dans `outputs/tickets.jsonl`.

Pour un appel réel compatible Ollama/OpenAI, configurez `GENAI_BASE_URL`, `GENAI_MODEL` et éventuellement `GENAI_API_KEY`, puis lancez `python src/extract.py --real --input data/tickets.jsonl`. Le mode mock reste disponible pour le test sans réseau.

Sortie obtenue

Chaque objet accepté possède exactement les trois champs utiles et une priorité autorisée.

■ Sortie / exemple

```
accepted=5 rejected=0 fields_ok=5 priority_values=low,medium,high
```

Commande de vérification

Terminal · Bash

```
python src/extract.py --mock
python -c "import json; [json.loads(line) for line in open('outputs/tickets.jsonl', encoding='utf-8')];
print('jsonl ok')"
```

Code de l'étape

Python

```
ALLOWED = {"category": {"access", "billing", "technical", "other"},
            "priority": {"low", "medium", "high"}}

def validate_item(item):
    if set(item) != {"category", "priority", "entities"}:
        raise ValueError("schéma incomplet ou champs supplémentaires")
    if item["category"] not in ALLOWED["category"]:
        raise ValueError("category invalide")
    if item["priority"] not in ALLOWED["priority"]:
        raise ValueError("priority invalide")
    if not isinstance(item["entities"], list):
        raise ValueError("entities doit être une liste")
    return item
```

4. Mesurer les erreurs

Mesurez séparément les échecs de structure et les erreurs de contenu. Un JSON illisible, une priorité hors liste et une mauvaise catégorie sont trois problèmes différents ; les regrouper dans un seul taux empêcherait de choisir la bonne correction. Le fichier des erreurs permet de revenir au ticket et à la sortie qui ont produit chaque échec.

Avant de calculer l'exactitude, vérifiez que les prédictions et les références ont la même longueur et correspondent aux mêmes tickets. L'extrait utilise `zip`, qui s'arrête à la liste la plus courte : des réponses manquantes pourraient ainsi disparaître du calcul. Une réussite sur cinq cas reste un contrôle de fonctionnement ; ajoutez ensuite des tickets ambigus, incomplets ou contenant plusieurs demandes.

Comparez les cinq sorties à une vérité terrain écrite dans `data/expected.json`. Comptez les catégories incorrectes et les réponses rejetées. Ne remplacez pas une erreur par une réponse inventée.

Sortie obtenue

Un rapport affiche le nombre de lignes traitées, acceptées et rejetées.

Sortie / exemple

```
processed=5 accepted=5 rejected=0 exact_match=1.00
```

Commande de vérification

Terminal · Bash

```
python src/evaluate.py --predictions outputs/tickets.jsonl --expected data/expected.json
```

Code de l'étape

Python

```
predictions = [json.loads(line) for line in open("outputs/tickets.jsonl")]
expected = json.load(open("data/expected.json", encoding="utf-8"))
matches = sum(prediction == reference
                for prediction, reference in zip(predictions, expected))
print({"processed": len(predictions), "expected": len(expected),
      "exact_matches": matches})
```

Exemples de sorties

En mode mock, cinq tickets sont lus et le fichier de sortie est valide. Les scores du vrai modèle dépendent du modèle, du prompt et de la température ; notez-les avec la version utilisée.

Dépannage et diagnostics

Une erreur JSON doit être conservée dans `outputs/errors.jsonl`. Vérifiez aussi que le prompt demande explicitement les valeurs autorisées et qu'aucune donnée personnelle réelle n'est utilisée.

Vérification finale

- ☐ cinq tickets traités ;
- ☐ schéma validé ;
- ☐ erreurs séparées des sorties acceptées ;
- ☐ modèle et prompt consignés.

Limites et prolongements

Le JSON valide ne garantit pas que le contenu est vrai. Ajoutez ensuite un échantillonnage humain et des tests de régression avant toute intégration.

Références

- [JSON Schema — documentation officielle](#)
- [OpenAI — Structured Outputs](#)