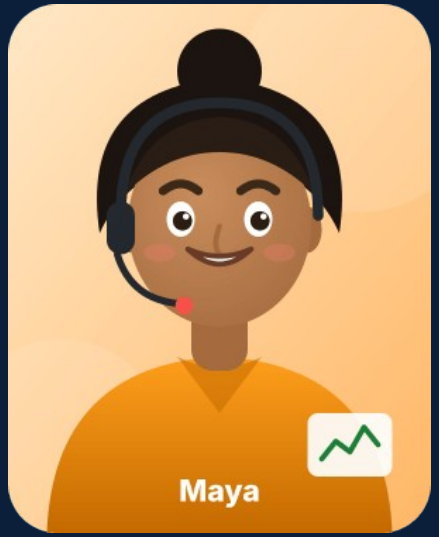
HEF-160 · MISSION 02 / 10

Prompt engineering orienté tâche

Un bon prompt est une spécification : il dit quoi faire, avec quoi, sous quelles contraintes, dans quel format — et comment on saura que c'est réussi.



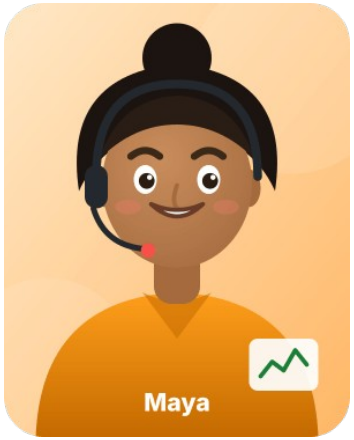


Badge : Rédacteur de consignes

105 min · avancé

L'histoire

Maya demande à un assistant : « Fais-moi un résumé de nos retours clients. » Elle obtient dix lignes sympathiques, vagues, inutilisables. Elle reformule : « Voici 40 retours clients [collés]. Rédige un résumé pour la direction (5 lignes max) : les 3 principaux problèmes, avec le nombre de retours concernés, une citation exacte par problème, et une recommandation par problème. N'invente aucun chiffre : si une donnée manque, écris "non disponible". Réponds en JSON avec les clés `problemes`, `citation`, `recommandation`. » Résultat : exploitable, vérifiable, réutilisable.



> *« La différence, ce n'est pas la politesse ni un mot magique, dit Sam. C'est que la deuxième demande contient une spécification : tâche, contexte, contraintes, format, critère de réussite. Un prompt, c'est une interface. Et comme toute interface, ça se conçoit, ça se teste, ça se versionne. »*

L'histoire (suite)

Tu apprends à structurer un prompt orienté tâche, à utiliser des exemples (few-shot), à demander un format de sortie strict qu'on peut valider, à gérer les échecs par réparation contrôlée, et à tester tes prompts comme du code.

Ta mission

À la fin, je serai capable de…

1. structurer un prompt (rôle, tâche, contexte, contraintes, format, exemples, critères)
2. exiger une sortie structurée et la valider par un schéma, avec une boucle de réparation bornée
3. choisir des exemples (few-shot) pertinents et mesurer leur effet
4. décomposer une tâche complexe en étapes vérifiables
5. tester et versionner des prompts avec des cas de test et un modèle simulé

Défi d'échauffement

Défi d'échauffement

Tu confies une tâche à un nouveau collègue très compétent mais qui ne connaît ni l'entreprise, ni ton projet, ni tes habitudes. Version A : « Occupe-toi du rapport. » Version B : un message qui contient l'objectif, le public, la date limite, les données à utiliser, ce qu'il ne faut pas faire, le format attendu et un exemple de rapport réussi. Dans quel cas tu obtiens le bon résultat du premier coup ? Qu'est-ce que B contient que A n'a pas ?

Le concept

Un prompt est une spécification

Un modèle n'a **que** ce que contient son contexte. Tout ce que tu sais et qu'il ne sait pas doit y figurer. Les ingrédients d'un bon prompt :

Élément	Question à laquelle il répond	Exemple
Rôle / point de vue	qui le modèle doit-il être ?	« Tu es un relecteur de code Python expérimenté. »
Tâche	que faut-il produire, précisément ?	« Identifie les failles d'injection SQL dans ce code. »
Contexte	quelles informations utiliser ?	le code, la politique, les données
Contraintes	quelles règles ?	« N'invente rien. Cite la ligne. Pas de suggestions de style. »

Le concept

Un prompt est une spécification (suite)

Élément	Question à laquelle il répond	Exemple
Format de sortie	comment présenter ?	tableau, JSON avec clés données
Exemples	à quoi ressemble une bonne réponse ?	1 à 3 paires entrée → sortie
Critères de réussite	comment saura-t-on que c'est bon ?	« chaque faille a un numéro de ligne et une correction »
Cas limites	que faire quand il manque quelque chose ?	« Si aucune faille n'existe, réponds `[]`. »

Principes

1. **Spécificité** : remplace « un bon résumé » par ses critères.
2. **Séparer instructions et données** : délimiteurs clairs (`<donnees>...</donnees>`), surtout pour des contenus non fiables (mission 9).

Le concept

Principes (suite)

3. **Dire quoi faire** plutôt que quoi ne pas faire (et ajouter les interdits essentiels).
4. **Autoriser « je ne sais pas »** : prévoir la valeur de repli (`null`, « non disponible ») pour réduire l'invention.
5. **Demander de citer** les passages sources quand l'exactitude compte.
6. **Une tâche par requête** quand c'est possible ; **décomposer** les tâches complexes en étapes enchaînées avec vérification entre elles.
7. **Itérer** : un prompt se révise comme du code, à partir des échecs observés.

Les exemples (few-shot)

Quelques paires entrée → sortie **montrent** le format et le ton mieux qu'une longue description.

- choisir des exemples **représentatifs et variés**, dont un **cas limite** ;

Le concept

Les exemples (few-shot) (suite)

- les **exemples proches** de la requête (par similarité) aident davantage que des exemples quelconques ;
- l'ordre peut influencer ; **tester**.

Sorties structurées

Pour qu'un programme exploite la sortie, exiger un **format strict** (JSON) et le **valider** :

1. définir un **schéma** (clés obligatoires, types, valeurs permises) ;
2. **analyser** la sortie et **valider** ;
3. si invalide : **réparer** — renvoyer au modèle l'erreur précise (« la clé `severite` doit valoir `basse`, `moyenne` ou `haute` ») ;
4. **borner** le nombre de tentatives (2 ou 3) puis échouer proprement ;
5. ne **jamais** exécuter ni interpréter sans validation une sortie de modèle.

Le concept

Sorties structurées (suite)

Beaucoup de fournisseurs offrent un mode « sortie structurée » contraint par un schéma, qui réduit (sans éliminer) les erreurs : la validation reste nécessaire.

Décomposer une tâche

```
Étape 1 : extraire les faits du texte (liste vérifiable)
Étape 2 : classer chaque fait selon la grille
Étape 3 : rédiger le résumé en n'utilisant que la liste de l'étape 2
```

Chaque étape produit une sortie **vérifiable**, ce qui limite la propagation d'erreurs. Le coût : plus d'appels.

Demander un raisonnement

Le concept

Demander un raisonnement (suite)

Demander au modèle d'**expliciter ses étapes** améliore souvent les résultats sur des problèmes à plusieurs étapes, et permet de **relire** le raisonnement. Mais le raisonnement affiché n'est **pas une preuve** : il peut être plausible et faux. On vérifie la **conclusion**, par un calcul ou un test.

Prompts comme du code

Pratique	Raison
Versionner les prompts (dépôt)	traçabilité, revue
Cas de test (entrée → critères vérifiables)	détecter les régressions quand on modifie le prompt ou change de modèle
Modèles de prompts avec variables validées	pas d'injection accidentelle, pas de variable oubliée
Mesurer (taux de réussite sur un jeu)	décider avec des chiffres (mission 7)
Linter de prompts	rappeler les éléments manquants

Le concept

Prompts comme du code (suite)

🧠 ***Astuce*** *: relis ton prompt en te mettant à la place d'un lecteur compétent qui n'a aucun autre contexte. S'il lui manque une information pour réussir, il manque aussi au modèle.*

Exemple pas à pas

Problème : analyser la qualité d'un prompt avec un « linter », assembler un modèle de prompt avec variables validées, exiger une sortie JSON, la valider contre un schéma avec une boucle de réparation bornée (avec un modèle **simulé**), choisir des exemples few-shot par similarité, et tester un prompt comme du code.

```
import json, math, re
from collections import Counter

# ---------------- 1. un « linter » de prompt ----------------
ELEMENTS = {
    "tâche": re.compile(r"\b(rédige|résume|identifie|classe|extrais|écris|génère|t
   raduis|corrige|compare|liste)\b", re.I),
    "contexte": re.compile(r"<donnees>|\bvoici\b|\bci-dessous\b|\bcontexte\b", re.
   I),
    "contraintes": re.compile(r"\b(n'invente|ne pas|sans|maximum|au plus|uniquemen
   t|seulement|exactement)\b", re.I),
    "format": re.compile(r"\b(json|tableau|liste à puces|clés?|format|colonnes?)\b
   ", re.I),
    "repli": re.compile(r"\b(si .* (manque|absent|inconnu)|non disponible|null|je
```

Exemple pas à pas

Exemple pas à pas (suite)

```
   ne sais pas)\b", re.I),
    "critère de réussite": re.compile(r"\b(chaque|doit|vérifiable|cite|citation|av
   ec le nombre)\b", re.I),
}
def linter(prompt):
    presents = {nom for nom, motif in ELEMENTS.items() if motif.search(prompt)}
    manquants = [n for n in ELEMENTS if n not in presents]
    return len(presents) / len(ELEMENTS), manquants
VAGUE = "Fais-moi un résumé de nos retours clients."
PRECIS = ("Voici des retours clients dans <donnees>. Résume-les pour la direction
   : les 3 principaux problèmes avec le nombre de retours concernés, "
          "une citation exacte par problème. N'invente aucun chiffre : si une donn
   ée manque, écris non disponible. Réponds en JSON avec les clés problemes, citat
   ion.")
for nom, p in (("vague", VAGUE), ("précis", PRECIS)):
    score, manquants = linter(p); print(f"prompt {nom:7s}: score {score:.0%} | man
   que : {manquants or 'rien'}")
assert linter(VAGUE)[0] < 0.5 < linter(PRECIS)[0]
```

Exemple pas à pas

Exemple pas à pas (suite)

```
# ---------------- 2. modèle de prompt à variables validées ----------------
class ModelePrompt:
    def __init__(self, texte, variables): self.texte, self.variables = texte, vari
   ables
    def rendre(self, **valeurs):
        manquantes = set(self.variables) - set(valeurs)
        inconnues = set(valeurs) - set(self.variables)
        if manquantes or inconnues: raise ValueError(f"variables manquantes {sorte
   d(manquantes)} / inconnues {sorted(inconnues)}")
        rendu = self.texte
        for k, v in valeurs.items(): rendu = rendu.replace("{" + k + "}", str(v))
        return rendu
MODELE = ModelePrompt("Tu es un relecteur de code Python. Identifie les failles d'
   injection SQL dans le code ci-dessous. Réponds en JSON : une liste d'objets ave
   c les clés ligne (entier), "
                      "severite (basse|moyenne|haute), correction (texte). Si aucu
   ne faille, réponds []. N'invente aucune ligne.\n<donnees>\n{code}\n</donnees>",
    ["code"])
prompt = MODELE.rendre(code="bd.execute(f'SELECT * FROM t WHERE id = {x}')")
```

Exemple pas à pas (suite)

```
assert "<donnees>" in prompt
try: MODELE.rendre(code="x", langue="fr"); raise AssertionError
except ValueError as e: print("erreur de variable détectée :", e)

# ---------------- 3. sortie structurée : validation et réparation bornée --------
   --------
SCHEMA = {"ligne": int, "severite": ("basse", "moyenne", "haute"), "correction": s
   tr}
def valider(texte):
    try: donnees = json.loads(texte)
    except json.JSONDecodeError as e: return None, [f"JSON invalide : {e.msg}"]
    if not isinstance(donnees, list): return None, ["la racine doit être une liste
   "]
    erreurs = []
    for i, obj in enumerate(donnees):
        if not isinstance(obj, dict): erreurs.append(f"élément {i} : objet attendu
   "); continue
        for cle, attendu in SCHEMA.items():
            if cle not in obj: erreurs.append(f"élément {i} : clé « {cle} » manqua
```

Exemple pas à pas (suite)

```
   nte")
            elif isinstance(attendu, tuple) and obj[cle] not in attendu: erreurs.a
   ppend(f"élément {i} : « {cle} » doit valoir {'|'.join(attendu)}")
            elif isinstance(attendu, type) and (not isinstance(obj[cle], attendu)
   or isinstance(obj[cle], bool)): erreurs.append(f"élément {i} : « {cle} » doit ê
   tre de type {attendu.__name__}")
        if set(obj) - set(SCHEMA): erreurs.append(f"élément {i} : clés inattendues
    {sorted(set(obj) - set(SCHEMA))}")
    return (donnees if not erreurs else None), erreurs

class ModeleSimule:                          # un « modèle » scripté : il se trompe d
   'abord, puis corrige quand on lui donne l'erreur précise
    def __init__(self, reponses): self.reponses, self.appels = list(reponses), []
    def repondre(self, messages):
        self.appels.append(messages[-1]); return self.reponses.pop(0)

def appeler_avec_reparation(modele, prompt, max_tentatives=3):
    messages = [prompt]
    for tentative in range(1, max_tentatives + 1):
```

Exemple pas à pas (suite)

```
        sortie = modele.repondre(messages)
        donnees, erreurs = valider(sortie)
        if donnees is not None: return donnees, tentative
        messages.append("Ta réponse est invalide : " + "; ".join(erreurs) + ". Cor
   rige et réponds UNIQUEMENT avec le JSON valide.")
    raise RuntimeError(f"échec après {max_tentatives} tentatives : {erreurs}")
m = ModeleSimule(['Voici les failles : [{"ligne": 1}]', '[{"ligne": 1, "severite":
    "critique", "correction": "paramètres"}]', '[{"ligne": 1, "severite": "haute",
    "correction": "utiliser des paramètres"}]'])
donnees, essais = appeler_avec_reparation(m, prompt)
print(f"résultat valide après {essais} tentative(s) :", donnees)
print("dernier message de réparation envoyé :", m.appels[-1])
assert essais == 3 and donnees[0]["severite"] == "haute"
try: appeler_avec_reparation(ModeleSimule(["pas du json"] * 3), prompt); raise Ass
   ertionError
except RuntimeError as e: print("échec borné :", str(e)[:60], "…")

# ---------------- 4. exemples few-shot choisis par similarité ----------------
def sac(texte): return Counter(re.findall(r"\w+", texte.lower()))
```

Exemple pas à pas

Exemple pas à pas (suite)

```
def cosinus(a, b):
    num = sum(a[k] * b[k] for k in a)
    return num / ((math.sqrt(sum(v * v for v in a.values())) * math.sqrt(sum(v * v
    for v in b.values()))) or 1)
BANQUE = [("le paiement a échoué deux fois", "paiement"), ("je ne retrouve plus mo
   n mot de passe", "compte"), ("la facture est incorrecte", "paiement"),
          ("impossible de me connecter", "compte"), ("la vidéo du cours ne se char
   ge pas", "technique"), ("l'application plante au démarrage", "technique")]
def choisir_exemples(requete, k=2):
    q = sac(requete); return sorted(BANQUE, key=lambda ex: -cosinus(q, sac(ex[0]))
   )[:k]
choisis = choisir_exemples("mon paiement a été refusé")
print("exemples choisis pour « mon paiement a été refusé » :", [e[1] for e in choi
   sis])
assert choisis[0][1] == "paiement"

# ---------------- 5. tester un prompt comme du code ----------------
CAS_DE_TEST = [{"entree": "bd.execute('SELECT 1')", "attendu": lambda r: r == []},
    {"entree": "bd.execute(f'... {x}')", "attendu": lambda r: len(r) == 1 and r[0]
```

Exemple pas à pas (suite)

```
   ["severite"] == "haute"}]
def executer_tests(modele_factory):
    return sum(1 for cas in CAS_DE_TEST if (lambda r: cas["attendu"](r))(appeler_a
   vec_reparation(modele_factory(cas["entree"]), MODELE.rendre(code=cas["entree"])
   )[0]))
bon = lambda entree: ModeleSimule(["[]" if "{" not in entree else '[{"ligne": 1, "
   severite": "haute", "correction": "paramètres"}]'])
mauvais = lambda entree: ModeleSimule(['[{"ligne": 1, "severite": "basse", "correc
   tion": "x"}]'])
print("tests réussis — bon prompt :", executer_tests(bon), "/ 2 | prompt qui hallu
   cine une faille : ", executer_tests(mauvais), "/ 2")
assert executer_tests(bon) == 2 and executer_tests(mauvais) == 0
```

Lecture :

Exemple pas à pas (suite)

1. Le **linter** repère les éléments absents : le prompt vague n'a pas de contexte, de contraintes, de format ni de critère ; le prompt précis les contient presque tous. C'est une aide à la relecture, pas une garantie de qualité.
2. Le **modèle de prompt** refuse les variables manquantes ou inconnues : on évite d'envoyer « {code} » littéral ou d'oublier un champ.
3. La **sortie structurée** est validée contre un schéma. Les erreurs précises (« clé `correction` manquante », « `severite` doit valoir basse|moyenne|haute ») sont renvoyées au modèle ; au bout de trois tentatives, on **échoue proprement** plutôt que de continuer avec une sortie invalide.
4. Les **exemples few-shot** sont choisis par similarité avec la requête.
5. Les **prompts se testent** : un jeu de cas avec critères vérifiables détecte un prompt qui invente une faille là où il n'y en a pas.

Le « modèle » ici est un simulateur scripté : il illustre le **mécanisme** (validation et réparation), pas les performances d'un vrai modèle.

À toi, avec aide

Voici un prompt faible : « Écris-moi des tests pour cette fonction. »

1. Passe-le au linter et note les éléments manquants.
2. Réécris-le en un prompt qui précise : le langage et le framework de test, le comportement à couvrir (valeurs limites, erreurs), le format (code seul, noms de tests en français), les contraintes (pas de dépendance au réseau, pas de hasard), un critère de réussite (chaque cas d'erreur documenté testé), et le cas « si le comportement attendu n'est pas clair, pose une question au lieu de deviner ».
3. Vérifie que ton nouveau prompt passe le linter à au moins 80 %.
4. Écris les trois critères **vérifiables par un programme** que tu appliquerais à la sortie (par exemple : le code s'exécute, tous les noms de tests commencent par `test_`, aucun import réseau).

À toi, avec aide · réponse

Indices progressifs

Indice 1

Utilise la fonction `linter` de l'exemple.

Indice 2

Un critère vérifiable par programme : `ast.parse` réussit, nombre de fonctions `test_...`, absence de `requests`/`socket`.

Indice 3

Prévois une valeur de repli.

À toi, avec aide · réponse

Réponse

```
import ast, re
ELEMENTS = {
    "tâche": re.compile(r"\b(rédige|résume|identifie|classe|extrais|écris|génère|t
   raduis|corrige|compare|liste)\b", re.I),
    "contexte": re.compile(r"<donnees>|\bvoici\b|\bci-dessous\b|\bcontexte\b", re.
   I),
    "contraintes": re.compile(r"\b(n'invente|ne pas|sans|maximum|au plus|uniquemen
   t|seulement|exactement)\b", re.I),
    "format": re.compile(r"\b(json|tableau|liste à puces|clés?|format|colonnes?)\b
   ", re.I),
    "repli": re.compile(r"\b(si .* (manque|absent|inconnu|pas clair)|non disponibl
   e|null|je ne sais pas|pose une question)\b", re.I),
    "critère de réussite": re.compile(r"\b(chaque|doit|vérifiable|cite|citation|av
   ec le nombre)\b", re.I),
}
def linter(p):
    presents = {n for n, m in ELEMENTS.items() if m.search(p)}; return len(present
   s) / len(ELEMENTS), [n for n in ELEMENTS if n not in presents]
FAIBLE = "Écris-moi des tests pour cette fonction."
```

Réponse (suite)

```
BON = ("Écris des tests unitaires pytest pour la fonction ci-dessous, fournie dans
    <donnees>. Couvre le comportement nominal, les valeurs limites et chaque cas d
   'erreur documenté. "
       "Format : uniquement du code Python, noms de tests en français commençant p
   ar test_. Contraintes : sans accès réseau, sans hasard non fixé. "
       "Chaque cas d'erreur documenté doit avoir un test. Si le comportement atten
   du n'est pas clair, pose une question au lieu de deviner.\n<donnees>\n{code}\n<
   /donnees>")
print("faible :", linter(FAIBLE)); print("amélioré :", linter(BON))
assert linter(FAIBLE)[0] < 0.4 and linter(BON)[0] >= 0.8

def verifier_sortie(code):
    erreurs = []
    try: arbre = ast.parse(code)
    except SyntaxError as e: return [f"syntaxe invalide : {e.msg}"]
    tests = [n.name for n in ast.walk(arbre) if isinstance(n, ast.FunctionDef) and
    n.name.startswith("test_")]
    if not tests: erreurs.append("aucune fonction test_")
    imports = {a.name.split(".")[0] for n in ast.walk(arbre) if isinstance(n, ast.
```

À toi, avec aide · réponse

Réponse (suite)

```
    Import) for a in n.names} | {n.module.split(".")[0] for n in ast.walk(arbre) if
     isinstance(n, ast.ImportFrom) and n.module}
     if imports & {"requests", "socket", "urllib", "http"}: erreurs.append("import
    réseau interdit")
     if not all(re.fullmatch(r"test_[a-zàâçéèêëîïôûùüÿœ_0-9]+", t) for t in tests):
     erreurs.append("nom de test non conforme")
     return erreurs
assert verifier_sortie("def test_somme():\n    assert 1 + 1 == 2\n") == []
assert "import réseau interdit" in verifier_sortie("import requests\ndef test_x():
    \n    pass\n") and "aucune fonction test_" in verifier_sortie("x = 1\n")
print("trois contrôles automatiques : syntaxe valide, noms test_*, pas d'import ré
    seau")
```

Un bon prompt **appelle des contrôles automatiques** : si tu n'arrives pas à écrire comment vérifier la sortie, c'est que la tâche n'est pas encore assez spécifiée.

À toi, en solo

Banc de régression de prompts. Construis un petit outil pour qualifier un prompt avant de le déployer :

1. un jeu de 12 cas de test (entrée, propriétés attendues vérifiables par code) pour une tâche de classification de tickets (« paiement », « compte », « technique », « autre ») ;
2. un « modèle simulé » configurable dont le taux d'erreur dépend de trois caractéristiques du prompt : présence d'exemples, présence d'un format strict, présence d'une valeur de repli (tu choisis les effets, par exemple 30 % d'erreurs sans rien, 12 % avec tout), avec une graine fixée ;
3. une fonction `evaluer(prompt, n_repetitions=20)` qui exécute chaque cas plusieurs fois, valide la sortie par schéma, calcule le **taux de réussite** et un intervalle de confiance (Wilson) ;
4. compare 4 variantes du prompt (nu, + exemples, + format, + exemples + format + repli) et décide laquelle déployer avec une règle écrite d'avance (la meilleure dont l'intervalle bas dépasse 85 %) ;
5. montre qu'un changement de « modèle » (taux d'erreur différent) détecte une régression avec le même banc.

À toi, en solo · réponse

Pistes de correction

```
import json, math, random

CAS = [("le paiement a échoué", "paiement"), ("facture incorrecte", "paiement"), (
   "remboursement refusé", "paiement"), ("mot de passe oublié", "compte"), ("impos
   sible de me connecter", "compte"),
       ("changer mon adresse e-mail", "compte"), ("la vidéo ne charge pas", "techn
   ique"), ("l'application plante", "technique"), ("page blanche au démarrage", "t
   echnique"),
       ("bonjour, merci pour tout", "autre"), ("quel temps fait-il ?", "autre"), (
   "je voudrais un conseil de carrière", "autre")]
CATEGORIES = ("paiement", "compte", "technique", "autre")

def taux_erreur(prompt):
    e = 0.25
    if "exemple" in prompt.lower(): e -= 0.08
    if "json" in prompt.lower() and "categorie" in prompt.lower(): e -= 0.06
    if "autre" in prompt.lower() and "si aucune" in prompt.lower(): e -= 0.04
    return max(e, 0.05)
```

À toi, en solo · réponse

Pistes de correction (suite)

```
def modele_simule(prompt, entree, verite, rng):
    if rng.random() < taux_erreur(prompt):
        faux = rng.choice([c for c in CATEGORIES if c != verite])
        return json.dumps({"categorie": faux}) if "json" in prompt.lower() and rng
   .random() < 0.9 else faux          # sans format strict, il bavarde parfois
    return json.dumps({"categorie": verite}) if "json" in prompt.lower() else veri
   te

def valide(sortie, verite):
    try:
        d = json.loads(sortie); return d.get("categorie") == verite
    except (json.JSONDecodeError, AttributeError):
        return sortie.strip() == verite

def wilson(s, n, z=1.96):
    p = s / n; c = (p + z * z / (2 * n)) / (1 + z * z / n); d = z * math.sqrt(p *
   (1 - p) / n + z * z / (4 * n * n)) / (1 + z * z / n); return c - d, c + d

def evaluer(prompt, n_rep=20, graine=1):
```

À toi, en solo · réponse

Pistes de correction (suite)

```
    rng = random.Random(graine); ok = total = 0
    for _ in range(n_rep):
        for entree, verite in CAS:
            ok += valide(modele_simule(prompt, entree, verite, rng), verite); tota
   l += 1
    return ok / total, wilson(ok, total)

VARIANTES = {"nu": "Classe ce ticket.", "+ exemples": "Classe ce ticket. Exemple :
    « facture incorrecte » → paiement.",
             "+ format": "Classe ce ticket. Réponds en JSON avec la clé categorie.
   ",
             "complet": "Classe ce ticket parmi paiement, compte, technique, autre
   . Exemple : « facture incorrecte » → paiement. Réponds en JSON avec la clé cate
   gorie. Si aucune catégorie ne convient, réponds autre."}
print("variante    | réussite | IC95 Wilson")
res = {}
for nom, p in VARIANTES.items():
    taux, (bas, haut) = evaluer(p); res[nom] = (taux, bas, haut); print(f"{nom:11s
   } |  {taux:5.1%}  | [{bas:5.1%} ; {haut:5.1%}]")
```

À toi, en solo · réponse

Pistes de correction (suite)

```
eligibles = [n for n, (_, bas, _) in res.items() if bas >= 0.85]
choix = max(eligibles, key=lambda n: res[n][0]) if eligibles else None
print("variante déployée selon la règle (IC bas ≥ 85 %) :", choix)
assert res["nu"][0] < res["complet"][0] and res["complet"][1] >= 0.85
# régression : on « change de modèle » (erreurs ×2) et le banc le détecte avec le
   même prompt
taux_avant = evaluer(VARIANTES["complet"])[0]
_t = taux_erreur
taux_erreur = lambda p: _t(p) * 2.2
taux_apres, (bas_apres, _) = evaluer(VARIANTES["complet"])
taux_erreur = _t
print(f"prompt complet : avant {taux_avant:.1%} → après changement de modèle {taux
   _apres:.1%} (IC bas {bas_apres:.1%}) → régression {'DÉTECTÉE' if bas_apres < 0.
   85 else 'non détectée'}")
assert bas_apres < 0.85
```

À toi, en solo · réponse

Pistes de correction (suite)

La règle de déploiement est **écrite avant** d'exécuter les variantes, et le même banc détecte la régression quand le modèle change. Un prompt n'est jamais « bon » dans l'absolu : il est bon **pour un modèle, une tâche et un jeu de cas** — d'où l'importance de pouvoir rejouer le banc à chaque changement.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Prompt vague (« fais un résumé »)	Sortie vague et inutilisable	Spécifier tâche, contexte, contraintes, format, critères
Utiliser la sortie d'un modèle sans la valider	Données mal formées, injections, plantages	Schéma, validation, réparation bornée
Prompt jamais testé ni versionné	Régressions silencieuses au moindre changement	Cas de test, banc d'évaluation, dépôt
Tout demander en une fois	Erreurs qui se propagent sans contrôle	Décomposer en étapes vérifiables

Mini-boss

Mini-boss

Bibliothèque de prompts de HashBox. Crée une petite bibliothèque de 4 prompts pour des tâches réelles de l'équipe (résumé de retours clients, classification de tickets, extraction d'informations d'un e-mail, revue d'un extrait de code).

1. un modèle de prompt par tâche, avec variables validées et délimiteurs pour les données ;
2. un schéma de sortie par tâche et une boucle de validation/réparation bornée ;
3. 8 cas de test par tâche avec propriétés vérifiables (dont un cas limite et un cas « information absente » où la bonne réponse est le repli) ;
4. le linter de prompts appliqué à chacun (score ≥ 80 %) ;
5. un banc d'évaluation avec un modèle simulé : taux de réussite, intervalle de Wilson, règle de déploiement écrite d'avance ;
6. un journal de versions : pour chaque modification, la raison et l'effet mesuré ;
7. un fichier `README` expliquant comment ajouter un prompt et comment le qualifier.

Mini-boss · réponse

Pistes de correction

Un bon travail : 4 modèles avec variables, 4 schémas, boucle de réparation bornée, 32 cas de test, rapport d'évaluation par tâche avec intervalles, règle de déploiement écrite, journal de versions montrant l'effet mesuré de chaque modification (par exemple « ajout d'une valeur de repli : −40 % d'inventions sur les cas limites »), et un README qui impose qu'un prompt sans cas de test ne soit pas intégré.

Quiz éclair

Quiz éclair

1. Un bon prompt orienté tâche contient :

A. uniquement un mot poli
B. le plus de mots possible
C. tâche, contexte, contraintes, format de sortie et critères de réussite

2. Avant d'utiliser la sortie JSON d'un modèle dans un programme, il faut :

A. la valider contre un schéma et gérer l'échec
B. l'exécuter directement
C. supposer qu'elle est correcte

3. Pour qualifier un prompt avant déploiement, on :

A. le relit une fois

Quiz éclair

Quiz éclair (suite)

B. l'évalue sur un jeu de cas de test avec un intervalle de confiance
C. demande l'avis du modèle

4. 🔁 *Révision mission 01* : Un modèle de langage produit un texte en :

A. consultant une base de faits garantie exacte
B. copiant des pages web en direct
C. tirant successivement le jeton suivant selon une distribution de probabilités

5. 🔁 *Révision mission 01* : Une hallucination est :

A. une sortie plausible et fluide mais fausse ou non fondée
B. une erreur de syntaxe
C. un message de refus

Quiz éclair · réponse

Réponses

1. **C**.
2. **A**.
3. **B**.
4. **C**.
5. **A**.

Je réfléchis

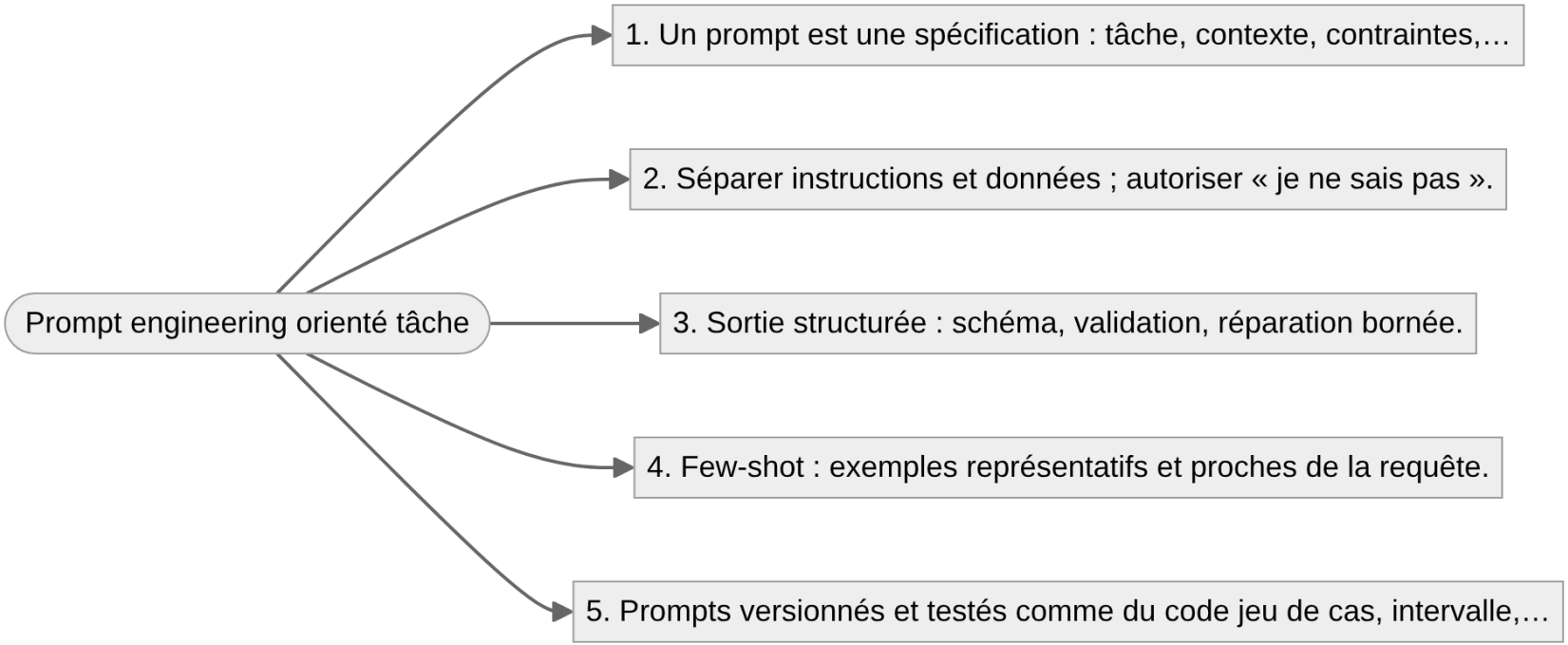
1. Quel prompt que tu utilises souvent n'a jamais été testé sur plus de deux exemples ?
2. Que perdrais-tu si le modèle changeait demain ? Pourrais-tu le détecter ?

À retenir

1. Un prompt est une spécification : tâche, contexte, contraintes, format, exemples, critères, repli.
2. Séparer instructions et données ; autoriser « je ne sais pas ».
3. Sortie structurée : schéma, validation, réparation bornée.
4. Few-shot : exemples représentatifs et proches de la requête.
5. Prompts versionnés et testés comme du code (jeu de cas, intervalle, règle de déploiement).

À retenir

Le schéma de la mission



MISSION ACCOMPLIE

+100 XP

Badge débloqué : Rédacteur de consignes

Prochaine mission : Context engineering

