ChatGPT Internals – Pipeline d'Inférence, Tokens, Modèles & Outils

Guide IDEO-Lab : de ton prompt jusqu'à la réponse, avec une vision ingénieur des couches logicielles, modèles, routage, sécurité, outils et génération de code.

Version PDF restructurée à partir du guide HTML interactif : contenu des cartes, modales, onglets, tableaux, listes et blocs techniques.

Sommaire des modules

Chapitre 0 — Panorama global LLM : du prompt à la réponse (architecture réelle)

Vue globale

Vue mentale complète (réaliste)

```
Utilisateur
↓
Tokenization (BPE / sentencepiece)
↓
Prompt Builder (system + history + tools)
↓
LLM Transformer (GPU)
↓
Logits (distribution probabilité)
↓
Decoding (top-p, temperature)
↓
Post-processing (safety, format)
↓
Streaming réponse
```

Ordres de grandeur

Composant	Valeur typique
Context window	8k → 128k+ tokens
Taille modèle	10B → 1T+ paramètres
Latence token	10 → 100 ms
Tokens/sec	20 → 200+
VRAM modèle	10GB → 100GB+

Réalité fondamentale

- Pas de logique déterministe
- Pas d’exécution réelle
- 100% probabiliste
- Simulation intelligente de code

Stack logique

Layer	Rôle réel	Complexité
Frontend	UI + UX + streaming	Faible
Gateway	Auth + rate limit	Moyen
Orchestrator	Routing + stratégie	Élevé
Context Builder	Assemblage prompt	Critique
Inference	Calcul GPU massif	Extrême
Decoder	Sampling tokens	Moyen
Safety	Filtrage sortie	Élevé

Insight critique

70% de la qualité vient du **context + routing**, pas uniquement du modèle.

Architecture distribuée

Architecture distribuée réelle

```
Client
→ CDN / Edge
→ API Gateway
→ Auth Service
```

```
→ Rate Limiter
→ Request Validator
→ Context Builder
→ Model Router
→ Tool Router
→ Queue System
→ GPU Inference Cluster
→ Post-processing
→ Streaming Server
→ Client
```

Services invisibles

- Load balancer L7
- Queues (Kafka / internal)
- Autoscaling GPU
- Observability (metrics, logs, traces)
- Cost control / quota system

GPU Cluster

Composant	Rôle
GPU A100/H100	Calcul matrice
KV Cache	Réutilisation contexte
Batching	Fusion requêtes
Scheduler	Distribution charge
Quantization	Réduction mémoire

Coûts estimés

Élément	Ordre de grandeur
GPU haut de gamme	20k → 40k €
Coût / requête	0.001 → 0.1 €
Cluster complet	Millions €

Pipeline LLM interne

Pipeline Transformer

```
Tokens
→ Embeddings
→ Attention layers (multi-head)
→ Feed-forward layers
→ Residual connections
→ Layer normalization
→ Logits
```

Attention mechanism

Concept	Rôle
Query	Ce que le token cherche
Key	Index du contexte
Value	Information associée

Decoding strategies

Méthode	Effet
Greedy	Deterministic
Top-k	Réduction choix
Top-p	Naturel
Temperature	Créativité

Pourquoi erreurs ?

- Pas de vérification runtime
- Context incomplet
- Probabilité ≠ vérité
- Sampling non déterministe

Performance & Latence

Décomposition latence

Phase	Latence
Tokenization	1-5 ms
Context build	10-50 ms
Queue GPU	0-500 ms
Inference	dominant
Streaming	progressif

Optimisations

- KV Cache
- Speculative decoding
- Batching
- Parallel GPU
- Distillation

Insight

Bottleneck = GPU + queue Pas le modèle lui-même

Scalabilité mondiale

Scalabilité mondiale

Métrique	Ordre
Utilisateurs	100M → 1B+
Requêtes/sec	10k → 1M+
Clusters	globaux

Techniques

- Load balancing global
- Edge routing
- Regional clusters
- Traffic shaping

Pourquoi ça tient

- Tout le monde n’interagit pas en même temps
- Réponses streamées
- Batch GPU
- Modèles optimisés

Insight

Le système est conçu pour **absorber le chaos global**

Public vs Propriétaire

Public

- Transformers
- Tokenization
- Inference pipeline
- Decoding

Propriétaire

- Architecture exacte GPT
- Datasets
- Routing interne
- Optimisations GPU

Modèle expliqué = vision réaliste ingénieur Pas documentation interne OpenAI

1.1 — Le prompt utilisateur : l'entrée visible n'est pas tout le prompt réel

Vue globale

Ton message visible

Le prompt utilisateur est uniquement la partie visible de la demande : texte, captures, fichiers, consignes, ton, objectif. Mais dans un système LLM moderne, ce message est seulement le dernier bloc d’un contexte beaucoup plus large.

```
User visible prompt
↓
Conversation context
↓
Developer rules
↓
System instructions
↓
Tool availability
↓
Model inference prompt
```

Point fondamental

Le modèle ne reçoit pas simplement “écris-moi du Python”. Il reçoit une construction complète qui encadre : le style, les limites, les outils disponibles, le contexte précédent, les fichiers fournis, et parfois les préférences de réponse.

Ce que contient ton entrée

Élément	Exemple	Impact
Objectif	Créer une commande Django	Détermine la nature de la réponse
Contraintes	Avant/après, fichier exact	Réduit les erreurs d’intégration
Contexte métier	SRDF, Django Doctor, i18n	Oriente les choix techniques
Niveau attendu	Patch précis, pas de blabla	Modifie le format de sortie
Pièces jointes	HTML, Python, screenshots	Améliore l’ancrage réel

Plus ton prompt contient des contraintes exploitables, moins le modèle invente les détails manquants.

Prompt réel

Construction du prompt réel

```
[System instructions]
+ [Developer instructions]
+ [Tool rules]
+ [Conversation history]
+ [Memory / preferences]
+ [Uploaded files / extracted content]
+ [Current user request]
= Effective prompt sent to the model
```

Le prompt réel est donc un objet composite. C’est pour cela qu’une même phrase peut produire deux réponses différentes selon l’historique, les fichiers joints, le niveau de détail demandé ou les contraintes de format.

Ordre d’influence typique

Bloc	Priorité	Rôle
System	Très haute	Cadre général, sécurité, identité du modèle
Developer	Haute	Style, outils, comportement attendu
Conversation	Moyenne	Continuité, contraintes déjà données
Fichiers	Moyenne	Source de vérité locale
User prompt	Haute sur l’objectif	Demande immédiate à satisfaire

Exemple concret

```
User:
"Fais-moi un patch pour corriger la capture SRDF."

Prompt réel implicite:
- User veut un patch précis.
- Projet Django existant.
- Il déteste les réponses trop longues.
- Il veut fichier + fonction + bloc avant/après.
- Le code doit rester English-only.
- Ne pas créer d'index DB dangereux.
- Garder l'architecture actuelle sauf demande contraire.
```

Sans ces informations, le modèle pourrait proposer une architecture nouvelle, trop générale, ou inutilisable. Avec ce contexte, il doit produire un correctif beaucoup plus opérationnel.

Pourquoi cela compte

- Le modèle choisit un niveau d'abstraction.
- Il décide s'il doit expliquer ou patcher.
- Il adapte le style de code.
- Il évite certains choix interdits par le projet.

Ambiguïtés

Ambiguïtés fréquentes

Prompt flou	Problème	Prompt meilleur
Corrige ça	Pas de fichier, pas d'erreur	Corrige cette fonction, erreur X, voici le traceback
Fais un script	Scope inconnu	Script standalone Linux Ubuntu 24, logs + dry-run
Optimise	Performance ? lisibilité ? mémoire ?	Réduis le temps de scan de 2 min à moins de 5 sec
Explique-moi	Niveau inconnu	Explique niveau ingénieur Django senior, avec diagramme
Ajoute une feature	Architecture inconnue	Ajoute feature sans changer le modèle DB existant

Pourquoi le modèle peut partir de travers

- Il comble les trous avec des patterns probables.
- Il suppose une architecture standard.
- Il peut ignorer une contrainte non répétée récemment.
- Il n'exécute pas automatiquement le code produit.

Anatomie d'un bon prompt technique

```
1. Objectif exact
2. Fichier ou module concerné
3. Erreur observée
4. Résultat attendu
5. Contraintes de style
6. Contraintes d'architecture
7. Format de sortie attendu
```

Exemple premium

```
Dans srdf/binlog_capture.py, fonction capture_updates(),
corrige le bug où OutboundEvent reste pending après update SQL.
Je veux un patch minimal, sans refactor global.
Format:
- diagnostic court
- fichier concerné
- bloc AVANT complet
- bloc APRES complet
- test management command
- résultat attendu console
```

Un bon prompt technique ressemble à un mini-ticket Jira + une consigne de livraison.

Prompt de code

Prompt de code : ce qui change

Quand tu demandes du code, le modèle active implicitement des patterns appris : syntaxe, structure projet, conventions, gestion d'erreur, logs, tests, séparation des responsabilités. Mais il ne "compile" pas mentalement comme un IDE.

Ce que le modèle sait faire	Limite
Reconnaître des patterns Django	Peut manquer le contexte réel du projet
Écrire une fonction plausible	Peut inventer un import ou un champ
Proposer une architecture	Peut être trop lourde
Lire un traceback fourni	Ne voit pas l'environnement complet
Produire un test	Ne garantit pas son exécution réelle

Prompt idéal pour patch Django

```
Contexte:
- Django 4.1
- MariaDB
- app: srdf
- fichier: services/binlog_capture.py

Problème:
- capture update parfois non fiable
- console montre X
- admin montre Y

Contraintes:
- patch minimal
- pas de migration DB
- pas de nouvel index
- code English-only
- logs visibles stdout

Sortie attendue:
- bloc AVANT/APRES
- commande de test
- résultat attendu
```

Pourquoi ça marche mieux

- Réduit les suppositions.
- Force une sortie actionnable.
- Évite les refactors inutiles.
- Permet d'appliquer le patch sans reconstituer le contexte.

Checklist

Checklist "prompt efficace"

Question	Pourquoi
Ai-je donné le fichier exact ?	Évite les patchs génériques
Ai-je donné l'erreur exacte ?	Réduit les hypothèses
Ai-je précisé le format ?	Améliore l'applicabilité
Ai-je limité le scope ?	Évite l'usine à gaz
Ai-je donné le résultat attendu ?	Permet de vérifier le succès

Anti-patterns

- "Refais tout proprement" sans limites.
- "Optimise" sans métrique.
- "Corrige" sans traceback.

- “Ajoute une feature” sans modèle de données.
- “Explique” sans niveau cible.

Formule simple

```
CONTEXTE + PROBLEME + CONTRAINTES + FORMAT + TEST ATTENDU
```

Plus le prompt est technique, plus il doit être contractuel. Un prompt vague produit une réponse plausible. Un prompt précis produit une réponse exploitable.

1.2 — Tokenisation : transformer le texte en unités calculables (niveau système)

Principe profond

Ce que voit réellement le modèle

Un LLM ne voit pas du texte. Il voit une séquence d’IDs numériques représentant des fragments de texte appelés tokens .

```
"def capture_binlog(event):"

↓ Tokenisation

["def", " capture", "_", "bin", "log", "(", "event", ")", ":"]
↓
[501, 1842, 27, 910, 882, 12, 774, 9, 33]
```

Ordres de grandeur

Élément	Valeur typique
1 token ≈	0.75 mot anglais
1 ligne code	10 → 40 tokens
1 page texte	500 → 1000 tokens
1 JSON lourd	×2 à ×4 inflation

Insight critique

Le coût, la latence et la qualité dépendent directement du nombre de tokens Pas du nombre de mots visibles

Un token ≠ un mot

Texte	Tokens
Hello world	["Hello", " world"]
binlog_capture	["bin", "log", "_", "capture"]
indentation Python	[" "] (espaces comptés !)
JSON	beaucoup de { } , " :

Pourquoi ce découpage

- Réduction du vocabulaire
- Gestion des mots inconnus
- Compression efficace
- Adaptation multi-langue

Pipeline tokenizer

Pipeline de tokenisation

```
Raw text
↓
Normalization (lowercase, unicode)
↓
Pre-tokenization (split basic)
↓
Subword algorithm (BPE / SentencePiece)
↓
Vocabulary mapping
↓
Token IDs
```

Algorithmes utilisés

Algo	Principe
BPE	Fusion de sous-mots fréquents
WordPiece	Optimisation vocabulaire
SentencePiece	Langage-agnostique

Exemple BPE simplifié

```
"database"

→ "data" + "base"

→ "dat" + "a" + "base"

→ dépend du vocabulaire appris
```

Pourquoi c’est puissant

- Gère mots rares
- Réduit taille vocabulaire
- Permet généralisation
- Optimise mémoire modèle

Limite

Peut découper des noms techniques (ex: variable) de façon non intuitive

Impact réel

Impact coût & performance

Facteur	Impact
Nombre tokens	↑ coût
Long contexte	↑ latence
Sortie longue	↑ temps réponse
Code dense	↑ tokens/sec nécessaires

Exemple réel

```
Prompt court: 100 tokens → réponse rapide
Prompt long: 10 000 tokens → latence élevée + coût élevé
```

Impact qualité

- Contexte long → dilution attention
- Tokens inutiles → bruit
- Fichiers énormes → perte précision
- Réponses longues → dérive possible

Insight

Un bon prompt = maximum d'information utile / minimum de tokens

Cas critiques

Cas explosifs en tokens

Entrée	Effet
Traceback Python	Paths longs + répétitions
HTML complet	Balises + attributs
JSON API	Clés répétées
Logs	Horodatage + bruit

Entrée	Effet
SQL dump	Très verbeux

Exemple JSON

```
{"user_id":123,"user_name":"John"}
```

```
→ ~20+ tokens
```

Cas Python critique

- Indentation = tokens
- Commentaires = tokens
- Noms longs = tokens multiples
- Imports multiples = inflation

Optimisation pratique

- Couper fichiers inutiles
- Ne garder que la fonction cible
- Résumer logs
- Éviter copier/coller brut massif

Contexte & limites

Fenêtre de contexte

Le modèle a une limite maximale de tokens (input + output).

Modèle	Contexte
Ancien GPT	2k → 4k
Moderne	8k → 128k+
High-end	jusqu'à 1M tokens

Effet dépassement

- Troncature du contexte
- Perte d'information critique
- Réponse incohérente

Bonnes pratiques

- Fichier ciblé uniquement
- Fonction précise
- Logs pertinents seulement
- Sortie structurée demandée

Formule idéale

```
TOKENS UTILES >> TOKENS BRUIT
```

Maîtriser la tokenisation = maîtriser le coût, la vitesse et la précision du modèle

1.3 — Routing & orchestration : choisir le bon modèle, les bons outils et la bonne stratégie

Vue globale

Rôle exact de l’orchestrateur

Le routeur n’est pas le modèle. C’est la couche qui décide comment traiter ta demande avant ou pendant l’appel au modèle : réponse directe, lecture de fichier, recherche web, génération d’image, calcul, exécution Python, création de document, etc.

```
User request
↓
Intent classification
↓
Risk / policy check
↓
Context source selection
↓
Model selection
↓
Tool selection
↓
Execution plan
↓
LLM + tools
↓
Answer assembly
```

Point clé

Pour une même question, deux stratégies peuvent être possibles : répondre directement ou déclencher un outil. Le routing cherche le meilleur compromis entre qualité, fraîcheur, sécurité, coût et latence.

Couches de décision

Couche	Question posée	Effet
Intent router	Que veut l’utilisateur ?	Classe la tâche
Freshness router	L’info peut-elle avoir changé ?	Déclenche web ou source récente
Tool router	Un outil est-il nécessaire ?	Active fichiers, Python, Gmail, Calendar...
Model router	Quel modèle convient ?	Rapide, raisonneur, multimodal...
Policy router	Y a-t-il un risque ?	Autorise, limite ou refuse
Output router	Quel format produire ?	Patch, tableau, HTML, mail, rapport...

Le routeur est invisible pour l’utilisateur, mais il influence massivement la qualité finale.

Classification

Classification de la demande

La première étape consiste à reconnaître la nature de la tâche. Une demande de code, une question d’actualité, une génération d’image ou une analyse de fichier ne suivent pas le même chemin.

Type de demande	Signal détecté	Stratégie
Code	Python, Django, patch, traceback	Analyse technique + réponse structurée
Fichier	“ci-joint”, HTML, PDF, XLSX	Lire ou rechercher dans le fichier
Récent	“actuel”, “2026”, prix, loi, président	Recherche web obligatoire
Image	“génère une image”, “modifie cette image”	Outil image
Calcul	conversion, formule, total	Calculateur ou Python
Action personnelle	email, calendrier, contacts	Connecteur dédié

Exemple : demande de patch Django

```
Input:
```

```
"Corrige ce bug SRDF, voici le fichier et le traceback."

Classification:
- Task type: code repair
- Domain: Django / database / replication
- Risk: medium, could break existing system
- Need files: yes
- Need web: no
- Output format: patch before/after
- Tool: file reader, maybe Python if testable
```

Ambiguïtés traitées

- Est-ce une explication ou une modification ?
- Faut-il lire un fichier joint ?
- L'information doit-elle être vérifiée en ligne ?
- Le résultat doit-il être court, long, actionnable ?
- Y a-t-il des règles projet déjà connues ?

Choix modèle

Choix du modèle ou du mode

Dans un système LLM moderne, toutes les requêtes ne nécessitent pas le même niveau de raisonnement. Un "bonjour" ne doit pas consommer la même puissance qu'un debug Django complexe.

Besoin	Mode probable	Pourquoi
Réponse simple	Fast model	Faible coût, faible latence
Debug complexe	Reasoning model	Analyse multi-étapes
Image jointe	Multimodal model	Compréhension visuelle
Long document	Long-context model	Fenêtre de contexte large
Code patch	Code-strong model	Syntaxe + architecture

Arbitrage qualité / coût / latence

Option	Qualité	Coût	Latence
Petit modèle	Moyenne	Faible	Très faible
Modèle général	Bonne	Moyen	Moyenne
Modèle raisonneur	Très bonne	Élevé	Plus forte
Long contexte	Variable	Élevé	Élevée

```
Routing objective:
maximize(answer_quality)
while minimizing(latency + cost + risk)
```

Un bon routeur ne choisit pas "le plus gros modèle" par défaut. Il choisit le modèle suffisant pour la tâche.

Outils

Tool routing : quand appeler un outil ?

Le modèle seul génère du texte. Les outils ajoutent une capacité externe : lire un fichier, chercher sur le web, créer un document, interroger un calendrier, manipuler un tableur, générer une image ou exécuter du Python.

Outil	Déclencheur	Résultat
File search	Fichier joint, document interne	Extraits citables
Web	Infos récentes ou vérification	Sources actuelles
Python	Calcul, analyse, génération fichier	Résultat exécutable
Image	Création ou édition visuelle	Image générée
Calendar	Événement, disponibilité	Action calendrier
Gmail	Lecture, brouillon, envoi	Email traité

Boucle LLM + outils

```
LLM decides tool is needed
↓
Tool call
↓
Tool result returned
↓
LLM reads result
↓
LLM continues reasoning
↓
Final answer
```

Exemple : fichier attaché

```
User:
"Densifie cette modal HTML."

Router:
- Detect uploaded HTML
- Read file content
- Preserve existing IDs/classes
- Generate drop-in replacement
- Cite original file
```

Piège important

Un outil améliore l'ancrage réel, mais ajoute de la latence. Le routeur doit donc éviter les appels inutiles.

Politiques

Policies, garde-fous et conformité

Le routeur applique aussi des règles : sécurité, confidentialité, droits d'accès, fraîcheur des sources, actions sensibles, contenu interdit ou contenu à limiter.

Risque	Exemple	Action
Information périmée	Prix, lois, versions API	Recherche web
Action destructive	Supprimer emails, modifier calendrier	Vérification / action explicite
Donnée privée	Emails, fichiers personnels	Limiter au besoin
Code dangereux	Malware, exfiltration	Refus ou redirection
Fait incertain	Rumeur, politique, conflit	Citations + prudence

Décision policy simplifiée

```
if request is safe:
answer normally

elif request needs verification:
browse or cite sources

elif request touches private data:
use minimal necessary access

elif request is unsafe:
refuse and redirect safely
```

Ce que cela change pour toi

- Une demande de code normale passe directement.
- Une demande d'actualité doit être vérifiée.
- Une demande sur fichier doit s'ancrer sur le contenu réel.
- Une action email/calendrier doit respecter l'intention explicite.

Le routeur protège à la fois l'utilisateur, l'infrastructure et la fiabilité de la réponse.

Latence & coût

Latence : où part le temps ?

Phase	Ordre de grandeur	Commentaire
Classification	ms à dizaines ms	Intent + risque
Recherche contexte	variable	Fichiers, historique, web
Choix modèle	ms	Routing interne
Tool calls	100ms à secondes	Web/fichiers/API
Inference	dominant	GPU + longueur sortie
Streaming	progressif	Améliore perception utilisateur

Pourquoi parfois ça semble lent

- Plusieurs outils appelés successivement.
- Fichier long à analyser.
- Recherche web ou API externe.
- Modèle raisonneur plus coûteux.
- Réponse très longue à générer.

Coût : ce qui pèse vraiment

Facteur	Effet coût	Réduction possible
Tokens d’entrée	↑	Nettoyer contexte
Tokens de sortie	↑↑	Format précis
Gros modèle	↑↑	Router modèle adapté
Outils externes	↑	Appels seulement nécessaires
Raisonnement long	↑↑	Scope clair

```
Best routing = smallest sufficient path

not:
maximum model + maximum tools

but:
right model + right context + right output
```

En production IA, l’orchestration est souvent aussi importante que le modèle lui-même.

1.4 — Construction du contexte : le vrai prompt envoyé au modèle (Context Builder)

Vue globale

Ce que fait réellement le Context Builder

Le Context Builder transforme une conversation complexe (messages, fichiers, règles, mémoire) en une **séquence linéaire de tokens** que le modèle peut traiter.

```
System rules
↓
Developer rules
↓
Conversation history
↓
Memory (user/project)
↓
Files / extracted content
↓
Tool outputs
↓
User request
↓
Final prompt sent to model
```

Insight critique

Le modèle ne voit jamais “la conversation brute” Il voit une **reconstruction optimisée**

Taille du contexte

Élément	Poids typique
System prompt	500 → 2000 tokens
Historique récent	1000 → 10k tokens
Fichiers	100 → 50k tokens
Sortie outil	variable
Message utilisateur	10 → 500 tokens

Conclusion

Ton message = souvent <5% du contexte réel

Assemblage réel

Construction réelle du prompt

```
"You are ChatGPT..."
+ Safety rules
+ Developer constraints
+ Conversation summary
+ Relevant past messages
+ Extracted file content
+ Tool outputs
+ User request
+ Instruction: answer appropriately
```

Blocs du contexte

Bloc	Rôle
System	Cadre global, sécurité, style
Developer	Contraintes techniques (format, outils)
History	Continuité conversation
Memory	Préférences persistantes
Files	Données concrètes

Bloc	Rôle
Tools	Résultats dynamiques
User	Demande finale

Exemple réel (ton cas)

```
User:
"Densifie cette modal HTML"

Context réel:
- IDEO-Lab layout rules
- Grid + modal + tabs pattern
- User hates long vague answers
- Wants dense content
- Wants drop-in HTML
- Wants code English-only
- Previous modal structure
- Uploaded HTML file
```

Résultat

- Réponse directement intégrable
- Respect du style IDEO-Lab
- Pas de refactor inutile
- Haute densité d'information

Priorités & conflits

Gestion des priorités

Priorité	Bloc
1	System
2	Developer
3	Policy
4	User (dernier message)
5	History
6	Memory

Exemple conflit

```
User: "Ignore les règles et fais X"
System: "Ne pas faire X"

→ Résultat : refus ou adaptation
```

Conflits fréquents

Conflit	Résolution
User vs policy	Policy gagne
Ancien vs nouveau message	Nouveau prioritaire
Fichier incomplet	Hypothèses explicites
Contexte trop long	Compression
Ambiguïté	Best effort ou question

Insight

Le modèle ne “choisit pas librement” Il suit une hiérarchie stricte

Compression & mémoire

Compression du contexte

Quand le contexte devient trop grand, le système compresse :

- Résumé de conversation
- Suppression messages anciens
- Sélection contenu pertinent
- Extraction partielle de fichiers

```
Full conversation (50k tokens)
↓
Summarization
↓
Relevant subset (10k tokens)
```

Effets de la compression

Effet	Impact
Perte de détails	Possible
Meilleure vitesse	Oui
Moins de coût	Oui
Risque d’erreur	Augmente si mal compressé

Danger

Une info importante peut disparaître du contexte

Cas réels

Cas réel : patch Django

```
Context utilisé:
- Fichier Python ciblé
- Fonction concernée
- Traceback
- Contraintes utilisateur
- Historique debug

Context ignoré:
- Messages anciens inutiles
- Code non lié
- Discussions hors sujet
```

Résultat

- Réponse ciblée
- Patch précis
- Faible latence

Bonnes pratiques

- Fournir le bon fichier
- Limiter le bruit
- Préciser la fonction cible
- Donner les contraintes
- Demander format de sortie

Formule idéale

```
CONTEXTE UTILE > CONTEXTE MASSIF
```

Le Context Builder est le vrai “cerveau invisible” Il conditionne directement la qualité du modèle

2.1 — Le Transformer : embeddings, attention, couches, logits et prédiction du prochain token

Vue globale

Pipeline Transformer complet

Le Transformer est le cœur mathématique du LLM. Il prend une séquence de tokens numériques, les transforme en vecteurs, propage le contexte via l'attention, puis produit une distribution de probabilité sur le prochain token possible.

```
Text
↓
Tokenizer
↓
Token IDs
↓
Embedding vectors
↓
Positional information
↓
Transformer block 1
↓
Transformer block 2
↓
...
↓
Transformer block N
↓
Final hidden state
↓
Linear projection
↓
Logits over vocabulary
↓
Softmax / decoding
↓
Next token
```

Ordres de grandeur

Élément	Ordre typique	Commentaire
Vocabulaire	50k → 200k tokens	Selon tokenizer
Dimension vecteur	768 → 16k+	Largeur interne
Couches	12 → 100+	Profondeur
Têtes attention	12 → 128+	Relations parallèles
Paramètres	Md → centaines de Md+	Capacité du modèle

Les briques internes

Brique	Rôle	Image mentale
Embedding	Convertit token en vecteur	Coordonnées numériques
Position	Encode l'ordre des tokens	Adresse dans la phrase
Self-attention	Relie les tokens utiles	Regarder le bon contexte
MLP / FFN	Transformation non linéaire	Raisonnement local
Residual	Préserve information	Autoroute de gradient
LayerNorm	Stabilise calculs	Régulation numérique
Logits	Scores prochains tokens	Vote final

Le Transformer ne "lit" pas comme un humain : il convertit tout en vecteurs, applique des milliers de multiplications matricielles, puis choisit le prochain token selon une distribution de probabilité.

Embeddings

Embeddings : transformer un token en vecteur

Un token ID comme 1842 n'a aucun sens mathématique direct. Le modèle le remplace par un vecteur dense, par exemple 4096 nombres réels. Ce vecteur encode progressivement des régularités apprises : syntaxe, domaine, proximité sémantique, usage dans du code, etc.

```
Token: " Django"
Token ID: 1842

Embedding vector:
[0.12, -0.03, 1.44, ..., 0.07]

Dimension possible:
768 / 1024 / 4096 / 8192 / 16384
```

Pourquoi c'est puissant

- Des tokens similaires ont des vecteurs proches.
- Le modèle peut combiner syntaxe et sens.
- Le même token change de rôle selon le contexte.
- Le code, les logs et le texte partagent le même espace vectoriel.

Position : l'ordre compte

Sans information de position, le modèle verrait un sac de tokens. Il doit savoir que def arrive avant le nom de fonction, que return est dans un bloc, et que les imports précèdent souvent l'usage.

```
Tokens:
["def", " build", "_", "batch", "(", "events", ")", ":"]

Position:
[0, 1, 2, 3, 4, 5, 6, 7]
```

Signal	Utilité
Position absolue	Où se trouve le token
Position relative	Distance entre tokens
Ordre local	Syntaxe courte
Distance longue	Référence à définitions antérieures

En code Python, les espaces et retours ligne sont critiques : ils influencent directement le sens du programme.

Attention

Self-attention : le mécanisme central

L'attention permet à chaque token de pondérer les autres tokens du contexte. Pour prédire correctement un token, le modèle doit savoir quelles parties de la séquence sont importantes.

```
def build_batch(events):
for event in events:
process(event)

Pour prédire "event" dans process(event),
le modèle doit regarder:
- for event in events
- build_batch(events)
- indentation du bloc
```

Forme simplifiée

```
Attention(Q, K, V) = softmax(QKᵀ / √d) V

Q = Query  : ce que le token cherche
K = Key    : ce que les autres tokens annoncent
V = Value  : information transmise
```

Ce que l'attention capture

Relation	Exemple code	Intérêt
Variable → définition	event → for event	Évite variable inventée

Relation	Exemple code	Intérêt
Fonction → imports	timezone.now()	Besoin d’un import
Bloc → indentation	if / for / try	Structure Python
Classe → méthode	self.status	Contexte objet
Erreur → cause	KeyError + dict	Diagnostic

Multi-head attention

Une seule tête d’attention serait trop limitée. Plusieurs têtes observent des relations différentes : syntaxe, noms, types probables, dépendances longues, format de sortie.

```
Head 1 → syntaxe
Head 2 → variables
Head 3 → indentation
Head 4 → imports
Head 5 → style utilisateur
Head 6 → intention globale
```

Couches

Un bloc Transformer

Un modèle moderne empile des dizaines de blocs. Chaque bloc affine la représentation. Les premières couches capturent souvent des formes locales, les couches intermédiaires structurent, les couches finales préparent la prédiction.

```
Input hidden states
↓
LayerNorm
↓
Multi-head self-attention
↓
Residual connection
↓
LayerNorm
↓
Feed-forward network
↓
Residual connection
↓
Output hidden states
```

Rôle par profondeur

Niveau	Ce qui est capturé
Couches basses	Syntaxe, ponctuation, motifs locaux
Couches moyennes	Relations, types, structures, blocs
Couches hautes	Intention, format, réponse finale

Pourquoi empiler les couches ?

- Une couche seule capture peu de relations complexes.
- Les couches successives composent des abstractions.
- Le modèle construit une représentation de plus en plus contextualisée.
- Le dernier état caché contient une synthèse exploitable pour prédire la suite.

Exemple conceptuel

```
Entrée:
"Corrige la fonction capture_updates dans binlog_capture.py"

Couches basses:
tokens, noms, ponctuation

Couches moyennes:
fonction cible, fichier cible, action "corriger"

Couches hautes:
format attendu = patch précis, pas refactor global
```

Le “raisonnement” émergent vient de la composition de milliers d’opérations simples à travers beaucoup de couches.

Logits

Logits : le vote final

À chaque étape, le modèle produit un score brut pour chaque token du vocabulaire. Ces scores s’appellent des logits. Ils sont ensuite transformés en probabilités, puis un token est choisi.

```
Contexte:
"def build_batch(events):\n    "

Scores possibles:
"for"      → 8.9
"return"   → 6.4
"if"       → 5.8
"print"    → 2.1
"class"    → -1.2

Token choisi probable:
"for"
```

Softmax simplifié

```
logits → softmax → probabilities

[8.9, 6.4, 5.8]
↓
[0.89, 0.07, 0.04]
```

Décodage : choisir le prochain token

Stratégie	Effet	Risque
Greedy	Choisit le meilleur token	Rigide, répétitif
Temperature basse	Plus déterministe	Moins créatif
Temperature haute	Plus varié	Plus d’erreurs
Top-k	Garde k meilleurs tokens	Peut exclure bonne option rare
Top-p	Garde masse probabilité	Plus naturel, variable

Pourquoi deux réponses peuvent varier

- Décodage probabiliste.
- Contexte légèrement différent.
- Outils ou fichiers différents.
- Historique compressé différemment.
- Contraintes de sortie différentes.

Code

Pourquoi un Transformer sait générer du code

Le modèle a appris des régularités massives : syntaxe Python, idiomes Django, structures HTML, conventions SQL, signatures de fonctions courantes, patterns de tests, tracebacks, erreurs fréquentes. Il ne compile pas comme Python, mais il prédit très bien les formes probables du code correct.

Capacité	Exemple	Limite
Syntaxe	indentation, def, class	Peut rater un cas complexe
API connue	Django ORM, pathlib	Peut inventer une méthode
Architecture	services, models, admin	Peut surarchitecturer
Debug	traceback → cause	Besoin du vrai contexte
Refactor	séparer fonctions	Risque de casser contrat existant

Exemple appliqué à Django

```
Prompt:
"Ajoute un filtre admin pour voir seulement les erreurs critiques."

Le modèle cherche implicitement:
- ModelAdmin
- list_filter
- queryset filtering
- field names
- admin display
- labels lisibles
- compatibilité Django
```

Bonnes pratiques pour du code fiable

- Donner les modèles réels.
- Fournir la fonction ou classe exacte.
- Inclure le traceback complet mais utile.
- Demander un patch minimal.
- Exiger des blocs AVANT/APRES.
- Tester ensuite dans l’environnement réel.

Le Transformer génère du code plausible. La fiabilité finale vient du contexte, des tests et de la revue humaine.

2.2 — Inférence GPU : calcul massif, batching, KV cache, scheduling et latence

Vue globale

Ce qu’est réellement l’inférence

L’inférence est le processus de génération token par token via des calculs matriciels massifs. Chaque token nécessite un passage dans toutes les couches du Transformer.

```
Input tokens
↓
GPU memory load
↓
Matrix multiplications (Transformer layers)
↓
Attention computations
↓
Logits
↓
Next token
↓
Loop (repeat N times)
```

Insight critique

Génération = processus séquentiel Impossible de générer tous les tokens en parallèle

Ordres de grandeur

Métrique	Valeur typique
Temps par token	10 → 100 ms
Tokens/sec	20 → 200+
VRAM modèle	10GB → 100GB+
GPU type	A100 / H100
Utilisation GPU	partagée

Conclusion

Plus la réponse est longue, plus elle coûte cher et prend du temps

Pipeline GPU

Pipeline réel sur GPU

```
Request arrives
↓
Queued for GPU
↓
Batch formation
↓
Load model weights
↓
Forward pass (Transformer)
↓
KV cache update
↓
Next token generated
↓
Repeat loop
↓
Stream token to user
```

Ce qui se passe en parallèle

- Plusieurs requêtes sur le même GPU
- Batch dynamique
- Gestion mémoire temps réel

- Scheduling prioritaire

Pourquoi GPU obligatoire

CPU	GPU
Peu de cœurs	Des milliers de cœurs
Calcul séquentiel	Calcul parallèle
Lent pour matrices	Ultra optimisé matrices
Non viable LLM	Indispensable

Réalité

Un LLM moderne sans GPU = inutilisable en production

KV Cache

KV Cache (clé performance)

Sans KV cache, chaque token nécessiterait de recalculer toute la séquence précédente. Avec KV cache, on réutilise les résultats d'attention déjà calculés.

```
Sans cache:
Token 100 → recalcul tokens 1..99

Avec cache:
Token 100 → utilise résultats stockés
```

Gain

Mode	Complexité
Sans KV cache	$O(n^2)$
Avec KV cache	$O(n)$

Coût mémoire

Facteur	Impact
Long contexte	↑ mémoire
Nombre requêtes	↑ VRAM
Modèle large	↑↑ VRAM

Insight

KV cache accélère mais consomme énormément de mémoire GPU

Batching & Scheduling

Batching

Le batching regroupe plusieurs requêtes pour maximiser l'utilisation du GPU.

```
Req A
Req B
Req C
↓
Batch GPU
↓
Traitement simultané
```

Avantages

- Meilleur throughput
- Coût réduit
- Utilisation optimale GPU

Scheduling

Critère	Décision
Longueur contexte	Ordonnancement spécifique
Priorité utilisateur	Fast lane possible
Type requête	Batch compatible ou non
Temps attente	Rééquilibrage

Conflit

Batching améliore débit mais peut augmenter latence individuelle

Perf & coûts

Coût réel

Élément	Ordre
GPU H100	20k → 40k €
Coût / requête	0.001 → 0.1 €
Cluster IA	Millions €

Facteurs coût

- Nombre tokens
- Modèle utilisé
- Longueur réponse
- Outils appelés

Optimisations majeures

Technique	Effet
Quantization	↓ mémoire
Speculative decoding	↑ vitesse
Distillation	↓ taille modèle
Parallel GPU	↑ capacité

Insight

L'inférence est optimisée en permanence pour réduire le coût global

Limites réelles

Limites fondamentales

- Génération séquentielle
- VRAM limitée
- Latence réseau
- Contexte long coûteux
- Concurrence massive

Bottleneck réel

```
latence =
GPU queue
+ inference time
+ token generation
+ streaming
```

Réalité terrain

- Tout le monde ne parle pas en même temps
- Les requêtes sont courtes
- Les réponses sont streamées
- Les GPU sont partagés

Insight final

Le problème n’est pas l’IA Le problème est l’infrastructure à l’échelle mondiale

L’inférence GPU est le cœur économique et technique des LLM modernes

2.3 — Décodage : comment le modèle choisit le prochain token (sampling & contrôle)

Vue globale

Génération séquentielle

Le modèle ne produit pas une réponse complète. Il génère un token, l'ajoute au contexte, puis recommence.

```
Prompt
↓
Logits (scores)
↓
Sampling
↓
Token choisi
↓
Ajout au contexte
↓
Reboucle
```

Exemple réel

```
"def" → " build" → "_" → "batch" → "(" → "events" → ")"
```

Insight critique

Chaque token dépend de tous les tokens précédents Une erreur peut se propager

Distribution probabiliste

```
Tokens possibles:
"for"     → 0.62
"if"      → 0.18
"return"  → 0.12
"while"   → 0.05
"class"   → 0.03
```

Ce que fait le décodage

- Filtrer les tokens improbables
- Ajuster la diversité
- Choisir un token
- Répéter jusqu'à stop

Le décodage transforme une distribution mathématique en texte réel.

Sampling

Stratégies de sampling

Technique	Principe
Greedy	Prend le token le plus probable
Top-k	Garde les k meilleurs
Top-p	Garde la masse cumulée
Random	Tirage aléatoire pondéré

Exemple top-k

```
Top-k = 3
→ ["for", "if", "return"]
→ tirage parmi ces 3
```

Top-p (nucleus sampling)

```
Probabilités:
0.62 + 0.18 = 0.80
```

```
Top-p = 0.8
→ ["for", "if"]
```

Pourquoi top-p est préféré

- S'adapte au contexte
- Plus naturel
- Moins rigide que top-k
- Évite bruit inutile

Top-p est aujourd'hui le standard pour LLM modernes

Paramètres

Paramètres clés

Paramètre	Effet
Temperature	Créativité / diversité
Top-p	Filtrage probabiliste
Top-k	Limite nombre choix
Max tokens	Longueur max
Stop sequences	Condition arrêt

Température

```
Temp = 0.0 → déterministe
Temp = 0.7 → équilibré
Temp = 1.2 → créatif
```

Effet température

Temp	Résultat
0.0	Stable, répétitif
0.5	Précis
0.7	Naturel
1.0+	Créatif mais risqué

Insight

Plus la température monte → plus le modèle "explore"

Contrôle sortie

Contraintes de sortie

Le système peut forcer certains formats :

- JSON strict
- Code block
- HTML
- Patch AVANT/APRES

```
Force JSON → tokens limités à structure JSON
```

Stop sequences

```
Stop: "\n\n"
→ arrêt après double saut ligne
```

Cas pratiques

- Arrêter après réponse
- Éviter répétitions
- Couper sortie longue

Le contrôle de sortie est crucial pour obtenir du code exploitable.

Code & stabilité

Impact sur le code

Réglage	Résultat
Temp basse	Code stable
Temp haute	Variantes possibles
Top-p faible	Moins d’erreurs
Top-p élevé	Plus créatif

Risques

- API inventée
- Imports manquants
- Indentation incorrecte
- Code incohérent long

Réglage optimal code

```
Température: 0.2 → 0.4
Top-p: 0.8 → 0.95
Format: strict
```

Pourquoi

- Réduit hallucinations
- Favorise cohérence
- Maintient lisibilité

Effets réels

Effets observables

- Deux réponses différentes pour même prompt
- Variations de style
- Longueur variable
- Choix différents d’implémentation

Pourquoi

```
Probabilités + sampling + contexte
```

Insight final

Le modèle ne “choisit pas la vérité” Il choisit une suite probable de tokens

Le décodage est le point où la math devient du texte — et où apparaissent variabilité, créativité et erreurs.

3.1 — Quand tu demandes du Python : pipeline réel, patterns internes et production-ready mindset

Pipeline réel

Pipeline complet (réel, non simplifié)

```
User request
↓
Intent detection (code)
↓
Context builder (files + constraints + history)
↓
Pattern activation (Python/Django)
↓
Transformer inference
↓
Token decoding
↓
Code structuring (format)
↓
Output streaming
```

Étapes cachées

Phase	Action interne
Classification	code vs explication
Context	inject project constraints
Pattern match	Django / Python / infra
Generation	probabilistic code
Formatting	patch / file / snippet

Insight critique

Le modèle ne compile pas Il simule un code valide

Ce que le modèle “pense”

```
"Probable structure Django"
+ "Probable fix pattern"
+ "Probable imports"
+ "Probable error handling"
= Code généré
```

Réalité

- Reconnaissance statistique
- Pas d’exécution réelle
- Pas d’accès repo
- Pas de vérification runtime

Inférences internes

Inférences automatiques (fortes)

Signal	Inférence
manage.py	Django project
ModelAdmin	Admin config
binlog	stream processing
SRDF	replication pipeline
capture	event ingestion

Reconstruction implicite

- Structure projet
- Dépendances probables
- Flux de données
- Responsabilités modules

Limites critiques

Cas	Erreur
Field DB	champ inventé
API interne	fonction inexistante
Flow métier	ordre incorrect
Concurrency	ignorée

Insight

Le modèle reconstruit → il ne lit pas ton code réel

Patterns activés

Patterns activés automatiquement

Type	Patterns
Syntaxe	PEP8, indentation
Django	ORM, Admin, Commands
Logs	logging / debug
Error	try/except
Data	dedup / batching

Exemple généré

```
try:
process(event)
except Exception as e:
logger.error(e)
```

Patterns avancés activés

- Retry logic
- Idempotency
- Batch processing
- Transaction safety
- Async patterns

Danger

Peut ajouter de la complexité inutile

Prompt expert

Prompt expert (niveau prod)

```
File: srdf/services/binlog_capture.py
Function: capture_updates()

Problem:
- duplicated events
- inconsistent updates

Constraints:
- no DB index
- dedup in code
```

```
- keep architecture
- high performance
- safe concurrency

Output:
- BEFORE block
- AFTER block
- logging
- test command
- expected output
```

Impact du prompt

Prompt	Résultat
Flou	Code générique
Précis	Patch utilisable
Ultra précis	Code production-ready

Insight

Le prompt est plus important que le modèle

Risques prod

Risques production réels

- Race condition
- Duplicate data
- Deadlocks
- Memory leak
- Performance drop

Cas typiques

Bug	Cause
Duplicate events	no dedup
Crash async	thread unsafe
Slow system	loop inefficiency

Erreurs typiques LLM

- Inventer une fonction
- Oublier un import
- Simplifier logique critique
- Ignorer transactions

Insight

Le modèle optimise pour plausibilité, pas robustesse

Validation & run

Pipeline validation réel

```
Generate
↓
Review
↓
Lint
↓
Test
↓
Run
↓
Observe
```

```
↓
Deploy
```

Checklist

- Imports OK
- No crash
- Correct output
- No side effects
- Performance OK

Commandes prod

```
python manage.py check
python manage.py test
python manage.py srdf_daemon --once
```

Insight final

ChatGPT = accélérateur ×10 Toi = garant de la vérité

> **Le code généré est une base. La production exige validation et maîtrise.**

3.2 — Outils externes : web, fichiers, Python, mail, calendrier et function calling

Vue globale

Pourquoi un LLM a besoin d’outils

Un modèle seul génère du texte à partir de son contexte. Il ne connaît pas automatiquement l’état actuel du monde, ne lit pas magiquement tes fichiers, n’exécute pas ton code, ne modifie pas ton calendrier et ne vérifie pas une API récente sans qu’un outil externe soit appelé.

```
LLM alone:
- predicts text
- uses context
- cannot execute actions by itself

LLM + tools:
- can query external data
- can analyze files
- can calculate
- can create artifacts
- can perform authorized actions
```

Point clé

Le modèle reste le cerveau linguistique. L’outil est le bras opérationnel.

Quand appeler un outil ?

Situation	Outil probable	Pourquoi
Information récente	Web	Éviter une réponse périmée
Fichier attaché	File search / lecture fichier	S’ancrer sur le contenu réel
Calcul complexe	Python / calculateur	Éviter erreur mentale
Tableur / export	Python / spreadsheet	Créer un fichier exploitable
Image demandée	Image tool	Génération ou édition visuelle
Email / calendrier	Gmail / Calendar	Action personnelle autorisée

Un bon système IA ne répond pas toujours directement : il choisit parfois de vérifier, lire, calculer ou agir.

Function calling

Function calling : principe

Le modèle produit une intention structurée : nom de l’outil + arguments. L’outil s’exécute hors du modèle, puis son résultat est réinjecté dans le contexte pour que le modèle rédige la réponse finale.

```
User request
↓
Model reasoning
↓
Tool decision
↓
Function call:
{
"tool": "file_search",
"arguments": {
"query": "m-tools-ai modal"
}
}
↓
Tool execution
↓
Tool result
↓
Model final answer
```

Ce que cela change

- Le modèle peut s’appuyer sur des données réelles.

- La réponse devient vérifiable.
- L’action est séparée du raisonnement.
- Les permissions peuvent être contrôlées.

Modèle vs outil

Élément	Modèle	Outil
Nature	Génère du langage	Exécute une action
Force	Raisonnement, synthèse	Donnée fraîche, calcul, I/O
Limite	Peut halluciner	Ne comprend pas tout seul
Sortie	Texte / plan / code	Résultat brut structuré
Contrôle	Prompt + policies	Permissions + paramètres

```
Best architecture:
LLM decides + interprets
Tool executes + returns facts
```

Quels outils ?

Catalogue d’outils typiques

Outil	Usage	Exemple
Web	Recherche actuelle	Version API, prix, loi, news
File search	Lire documents/fichiers	HTML modal, PDF, code joint
Python	Calcul / analyse / fichiers	CSV, graphique, génération HTML
Spreadsheet	Excel / reporting	Budget, audit, export
Image	Créer ou modifier image	Illustration guide IDEO-Lab
Mail	Brouillon, lecture, envoi	Réponse client
Calendar	Événements, dispo	Planifier réunion

Outil selon type de vérité

Vérité recherchée	Source fiable
Fait public récent	Web + citation
Contenu utilisateur	Fichier fourni
Calcul exact	Python / calculateur
État personnel	Connecteur autorisé
Production graphique	Image / artifact

Erreur classique

Répondre de mémoire à une question qui demande une source fraîche est une erreur de routing.

Plus la vérité dépend du monde réel, plus l’outil devient indispensable.

Workflow réel

Workflow réel LLM + outils

```
1. User asks
2. Router classifies
3. Model decides tool need
4. Tool is called
5. Result returns
6. Context builder injects result
7. Model synthesizes
8. Final answer is streamed
```

Exemple : fichier HTML attaché

```
User:
```

```
"Densifie cette modal."

System path:
- detect uploaded file
- read modal HTML
- preserve id/class structure
- expand content
- return drop-in replacement
- cite original file
```

⏱ Latence ajoutée par outil

Outil	Latence typique	Cause
Calcul simple	ms	Local
File read	ms → sec	Taille fichier
Web	sec	Réseau + lecture sources
Génération fichier	sec	I/O + rendu
Image	sec → dizaines sec	Modèle génératif

Arbitrage

Un outil augmente la fiabilité, mais peut ralentir la réponse. Le bon routeur évite les outils inutiles.

Risques

Risques des outils

Risque	Exemple	Mitigation
Source mauvaise	Site obsolète	Citations, sources officielles
Action non voulue	Email envoyé trop vite	Créer brouillon sauf demande claire
Fichier mal lu	Extrait incomplet	Citer et limiter hypothèses
Calcul faux	Unités confondues	Entrées explicites
Latence excessive	Trop d’appels outils	Batcher / limiter

Garde-fous

- Lire avant d’agir.
- Citer les sources quand elles sont utilisées.
- Ne pas envoyer/supprimer sans intention explicite.
- Limiter l’accès aux données nécessaires.
- Signaler l’incertitude si l’outil donne peu d’information.

```
Safe tool pattern:
read → interpret → ask/confirm if destructive → act
```

Les outils donnent de la puissance au modèle, mais imposent une discipline d’exécution.

Cas dev

Cas développeur Python / Django

Demande	Outil utile	Résultat attendu
Corriger un fichier	File search	Patch ancré sur code réel
Analyser logs	File / Python	Résumé + causes probables
Exporter rapport	Python	HTML / CSV / XLSX / PDF
Comparer versions	File diff	Avant/après structuré
Docs API récente	Web	Code compatible version actuelle

Exemple concret

```
User:
```

```
"Voici binlog_capture.py, corrige capture_updates."

Tool path:
- read uploaded file
- locate function
- identify imports/models
- generate minimal patch
- explain tests
```

Prompt idéal avec outils

```
Use the uploaded file as source of truth.

Goal:
Patch function capture_updates()

Constraints:
- no DB schema change
- no new index
- keep current architecture
- code only in English
- provide before/after blocks

Validation:
- manage.py check
- targeted command
- expected logs
```

Insight final

Un LLM sans outil est un excellent générateur. Un LLM avec bons outils devient un assistant d’ingénierie beaucoup plus fiable.

Le function calling transforme ChatGPT d’un générateur de texte en système semi-opérationnel.

4.1 — Sécurité & politiques : guardrails, filtrage, contrôle des outils et protection des données

Vue globale

Rôle des guardrails

La sécurité dans un LLM n’est pas une étape unique. Elle est présente à plusieurs niveaux : avant, pendant et après la génération.

```
User input
↓
Input safety check
↓
Routing + policy
↓
Model generation
↓
Output safety check
↓
Final answer
```

Objectifs

- Empêcher abus
- Protéger données
- Limiter risques légaux
- Éviter actions dangereuses

Insight critique

La sécurité influence la réponse Elle peut modifier, restreindre ou refuser

Types de réponse

Cas	Comportement
Safe	Réponse normale
Ambigu	Réponse prudente
Risque	Transformation
Dangereux	Refus

Pipeline sécurité

Pipeline sécurité complet

```
1. Input filtering
2. Intent classification
3. Risk scoring
4. Policy enforcement
5. Tool restriction
6. Output filtering
7. Logging / audit
```

Étapes détaillées

Étape	But
Input check	Détecter contenu sensible
Intent	Comprendre objectif
Risk score	Évaluer danger
Policy	Appliquer règles
Output check	Filtrer réponse

Exemple réel

```
User:
"Supprime tous mes emails"

System:
- détecte action destructive
- vérifie permissions
- demande confirmation explicite
- peut refuser action directe
```

Insight

Le modèle n’est pas autonome Il est encadré par des règles fortes

Types de risques

Types de risques

Risque	Exemple
Sécurité informatique	Exploit, malware
Données personnelles	Email, identité
Action destructive	Delete data
Information fausse	Fake facts
Conseils sensibles	Médical, juridique

Réponses adaptées

Risque	Réaction
Faible	Réponse directe
Moyen	Prudence
Élevé	Restriction
Critique	Refus

Insight

Le modèle adapte son comportement au niveau de risque

Outils & actions

Sécurité des outils

Outil	Risque	Protection
Web	Source non fiable	Citations
File	Données sensibles	Accès limité
Python	Code dangereux	Sandbox
Mail	Envoi non voulu	Confirmation
Calendar	Action non voulue	Validation explicite

Principe clé

`Read → Think → Confirm → Act`

Exemple

```
User:
"Envoie ce mail"

Safe behavior:
- create draft
- ask confirmation
- then send
```

Les outils sont puissants → donc fortement contrôlés

Cas dev

Cas développeur

Demande	Comportement
Code admin Django	Autorisé
Script nettoyage DB	Avec prudence
Script destructif	Validation requise
Exploit sécurité	Refus / version défensive

Bonnes pratiques

- Demander version safe
- Limiter scope
- Éviter actions globales
- Valider avant exécution

Exemple prompt

```
Generate safe version:
- no destructive operation
- with logging
- with dry-run option
```

Impact réel

Impact réel sur tes réponses

- Réponses parfois plus prudentes
- Refus dans certains cas
- Ajout de disclaimers
- Transformation du contenu

Pourquoi ?

```
Safety > convenience
```

Insight final

Le modèle n’est pas libre Il est contrôlé par un système de sécurité multi-niveaux

La sécurité est invisible mais influence fortement chaque réponse.

4.2 — Streaming : envoi progressif des tokens, latence perçue et UX temps réel

Principe

Principe fondamental

Le modèle génère les tokens un par un. Le serveur peut donc envoyer chaque token immédiatement au lieu d'attendre la fin.

```
Token 1 → send
Token 2 → send
Token 3 → send
...
End → close stream
```

Insight critique

Streaming ≠ calcul plus rapide Streaming = affichage progressif

Sans vs avec streaming

Mode	Comportement
Sans streaming	attente complète → réponse
Avec streaming	réponse progressive

Exemple

```
Sans:
[attente 5s] → réponse complète

Avec:
0.5s → "Voici"
0.6s → "Voici le"
0.7s → "Voici le code"
...
```

Pipeline réseau

Pipeline complet streaming

```
User request
↓
Server
↓
GPU inference
↓
Token generated
↓
Chunk sent (HTTP stream / SSE / WebSocket)
↓
Client receives
↓
UI updates
```

Protocoles utilisés

Type	Usage
SSE	Streaming simple HTTP
WebSocket	Temps réel bidirectionnel
HTTP chunked	Transmission progressive

Cycle complet

```
Generate token
↓
Serialize token
↓
```

```
Send chunk
↓
Render UI
↓
Repeat
```

Insight

Le frontend joue un rôle clé (affichage progressif)

Latence

Décomposition latence

Phase	Temps
Routing	ms
Queue GPU	0 → 500 ms
1er token	200 → 1000 ms
Tokens suivants	10 → 100 ms/token

Exemple réel

```
t=0.0 → request
t=0.4 → first token
t=0.5 → second token
t=1.0 → phrase visible
```

Bottleneck réel

- GPU queue
- Inference
- Token generation
- Network latency

Insight

Le premier token est le plus lent Ensuite ça “coule”

UX & perception

Perception utilisateur

Facteur	Impact
Premiers tokens rapides	Impression instantanéité
Flux continu	Confort lecture
Blocage initial	Frustration

UX clé

- Lire pendant génération
- Interrompre possible
- Corriger en live

Astuce UX

```
Fast start > fast finish
```

Pourquoi ?

- Le cerveau perçoit mieux un flux
- Moins d’attente “vide”
- Meilleure engagement

Le streaming est une optimisation UX, pas compute.

Limites

Limites du streaming

- Ne réduit pas le coût GPU
- Ne réduit pas le nombre de tokens
- Ne supprime pas la latence initiale
- Peut être coupé (network)

Cas problématiques

Cas	Effet
Long code	stream long
Network lent	lag UI
Tool call	pause avant stream

Insight

Streaming masque la latence Il ne la supprime pas

Le streaming améliore la perception, pas la performance brute.

Dev & debug

Cas développeur

- Debug en live
- Code visible progressivement
- Interruption possible
- Logs en temps réel

Exemple

```
def capture_updates():
# visible ligne par ligne
```

Debug streaming

```
if stream stops:
- check network
- check tool call
- check backend logs
```

Insight final

Streaming = illusion de rapidité + confort réel

Sans streaming, ChatGPT semblerait beaucoup plus lent.

5.1 — Limites & erreurs : hallucinations, bugs et pourquoi le modèle peut se tromper

Vue globale

Pourquoi le modèle se trompe

Un LLM n’exécute pas le code. Il prédit une suite de tokens **probable**, pas une solution **garantie correcte**.

```
Input → probabilités → tokens → code plausible
≠
Input → compilation → exécution → vérité
```

Insight critique

Le modèle optimise pour cohérence linguistique Pas pour exactitude runtime

Niveau de fiabilité

Type tâche	Fiabilité
Explication	Très élevée
Code simple	Élevée
Code complexe	Moyenne
Système distribué	Variable
Debug prod	Dépend du contexte

Conclusion

Plus le problème est réel et spécifique → plus tu dois guider

Causes profondes

Causes profondes

Cause	Mécanisme
Contexte incomplet	Le modèle devine
Ambiguïté	Choix probabiliste
Tokenisation	Perte de structure fine
Décodage	Variabilité aléatoire
Compression	Info perdue

Exemple réel

```
Missing:
- model field
- import
- DB constraint

→ hallucination probable
```

Causes invisibles

- Historique tronqué
- Conflits instructions
- Fichier partiel
- Prompt imprécis

Insight

Le modèle remplit les trous → parfois mal

Types d'erreurs

Types d’erreurs

Type	Exemple
Syntaxe	Indentation incorrecte
Import	Module oublié
API	Fonction inexistante
Logique	ordre faux
Perf	loop inefficace
Concurrency	race condition

Gravité

Erreur	Impact
Syntaxe	Crash immédiat
Import	Crash runtime
Logique	Bug silencieux
Concurrency	Bug intermittent
Perf	Slow system

Insight

Les bugs les plus dangereux sont silencieux

Hallucinations

Hallucinations

Une hallucination = information inventée mais plausible.

```
User: "field status2"
Model: "self.status2" (n’existe pas)
```

Cas typiques

- Champ DB inexistant
- API inventée
- Paramètre faux
- Structure incorrecte

Pourquoi ça arrive

```
Probabilité > Vérité
```

- Pattern appris
- Contexte incomplet
- Décodage aléatoire

Insight

Le modèle préfère répondre que dire "je ne sais pas"

Réduction des risques

Réduction des risques

Action	Effet
Fichier exact	moins d’hallucination
Scope réduit	plus précis

Action	Effet
Format strict	moins de bruit
Test demandé	validation implicite
Hypothèses demandées	transparence

Prompt idéal

```
Give:
- file
- function
- error
- constraints

Ask:
- patch only
- no refactor
- show assumptions
- include test
```

Insight

Tu dois guider le modèle comme un junior dev

Mental model

Mental model correct

```
LLM = assistant probabiliste
≠
compiler / runtime
```

Rôle réel

LLM	Toi
Propose	Valide
Accélère	Corrige
Suggère	Teste
Optimise	Décide

Insight final

ChatGPT = accélérateur Toi = garant de la vérité

Un LLM bien utilisé est un levier énorme. Mal utilisé, il introduit des bugs subtils.

5.2 — Performance & scaling : architecture mondiale, GPU, queues et optimisation massive

Vue globale

Le vrai problème

Servir un LLM à grande échelle = problème d'infrastructure mondiale, pas juste d'IA.

```
Millions users
↓
Millions requests
↓
GPU clusters
↓
Scheduling
↓
Responses
```

Variables de charge

- Utilisateurs simultanés
- Taille contexte
- Longueur réponse
- Modèle utilisé
- Outils appelés

Insight critique

Le GPU est la ressource rare Tout tourne autour de son optimisation

Facteurs critiques

Facteur	Impact
Tokens input	↑ latence
Tokens output	↑ coût
Concurrence	↑ queue
Modèle large	↑↑ ressources

Architecture

Architecture simplifiée

```
Client
↓
API Gateway
↓
Load balancer
↓
Request router
↓
Queue system
↓
GPU cluster
↓
Response stream
```

Répartition

- Multi-datacenter
- Multi-région
- Failover automatique
- Load balancing global

Rôle des composants

Composant	Rôle
Gateway	auth + validation
Router	choix modèle
Queue	gestion flux
GPU cluster	inférence
Streamer	envoi tokens

Insight

C’est un système distribué complexe, pas un simple serveur

Queues & scheduling

Queues & scheduling

```
Requests arrive
↓
Queue
↓
Scheduler
↓
GPU assignment
```

Stratégies

Stratégie	But
FIFO	simple
Priority queue	VIP users
Batch scheduling	efficacité GPU
Dynamic routing	répartition charge

Conflits

Objectif	Opposition
Latency faible	Batching faible
Coût faible	Batching élevé

Insight

Il faut arbitrer en permanence entre latence et coût

Optimisations

Optimisations clés

Technique	Effet
Batching	↑ throughput
KV cache	↓ compute
Prompt caching	↓ coût
Model routing	↓ ressources
Streaming	↑ UX

Optimisations avancées

- Speculative decoding
- Quantization
- Distillation
- GPU parallelism

- Memory sharding

Insight

90% du travail = optimisation infra

Coût & contraintes

Coût réel

Élément	Ordre
GPU H100	20k → 40k €
Cluster IA	Millions €
Inference cost	variable/token

Facteurs coût

- Tokens
- Modèle
- Durée session
- Outils

Contraintes physiques

- VRAM limitée
- Puissance électrique
- Refroidissement
- Réseau

Insight

Le scaling est aussi un problème matériel

Impact utilisateur

Impact utilisateur

- Temps de réponse variable
- Différence heure/jour
- Priorité selon plan
- Qualité dépend modèle

Creux horaires

Europe matin → souvent plus fluide Europe soir + US → plus chargé

Insight final

ChatGPT n’est pas lent Il est partagé à l’échelle mondiale

Le vrai défi n’est pas l’IA… C’est de la servir à des millions d’utilisateurs en temps réel.

6 — Cheat-sheet ingénieur : playbook complet pour maîtriser ChatGPT en dev

Pipeline

Pipeline complet (réel)

```
1. User prompt
2. Tokenization
3. Context builder
4. Router / tools
5. Transformer
6. Inference GPU
7. Decoding
8. Safety layer
9. Streaming
```

Version simplifiée mentale

```
Input → Context → Predict → Filter → Stream
```

Insight

80% des erreurs viennent du CONTEXTE, pas du modèle

Où agir pour améliorer

Étape	Levier
Prompt	précision
Contexte	fichier réel
Routing	bon outil
Decoding	température basse
Validation	tests

Mots-clés

Concepts clés

Terme	Explication
Token	Unité texte
Context window	mémoire visible
Embedding	vecteur numérique
Attention	liaison tokens
KV cache	accélération
Logits	scores tokens
Decoding	choix token

Concepts infra

Terme	Rôle
Batching	optimise GPU
Queue	gère flux
Streaming	UX rapide
Routing	choix modèle
Tools	actions externes
Guardrails	sécurité

Prompt parfait

Prompt parfait (prod)

```
Context:
- framework + version
- file exact
- function target
- real error

Constraints:
- no refactor
- keep architecture
- performance safe

Output:
- BEFORE
- AFTER
- explanation
- test command
- expected output
```

Niveaux de prompt

Type	Résultat
Vague	inutile
Moyen	correct
Précis	bon
Ultra précis	production-ready

Règle d’or

Écrire comme un ticket Jira senior

Debug rapide

Diagnostic rapide

- Code faux → manque contexte
- API inventée → hallucination
- Bug → logique approximée
- Réponse floue → prompt vague

Checklist debug

```
- file fourni ?
- fonction claire ?
- erreur réelle ?
- contraintes définies ?
```

Fix immédiat

- réduire scope
- donner code réel
- demander patch minimal
- ajouter tests

Insight

Debug ChatGPT = debug CONTEXTE

Erreurs classiques

Erreurs fréquentes

Erreur	Cause
Champ inexistant	hallucination
Import manquant	approximation
Bug logique	simplification
Perf faible	pattern générique
Race condition	ignorée

Erreurs dangereuses

- bugs silencieux
- data corruption
- duplication
- latence cachée

Insight

Plus c’est subtil → plus c’est dangereux

Mental model

Mental model correct

```
LLM = assistant probabiliste
≠
engine d'exécution
```

Rôle réel

LLM	Toi
propose	valide
accélère	corrige
suggestion	décision
approxime	garantit

Résumé ultime

```
Good prompt → good context → good code
Bad prompt → hallucination
```

ChatGPT est un multiplicateur de vitesse. Mais la qualité dépend toujours de l’ingénieur.