ChatGPT Internals - Inference Pipeline, Tokens, Models and Tools

IDEO-Lab illustrated English edition

A practical engineer-oriented guide explaining what happens between a user prompt and a generated answer: context building, routing, tokenization, transformer inference, GPU execution, decoding, tools, safety, streaming, limits, and scaling.

Figure - End-to-end LLM pipeline overview.

This regenerated PDF is an English adaptation of the uploaded HTML guide. It keeps the educational structure while adding explanatory diagrams for easier reading, printing, and reuse in documentation.

Table of contents

Chapter 0 - Global LLM overview: from prompt to answer

Explanatory diagram added for this section.

Mental model

A modern LLM service is not just a model running in isolation. It is a distributed pipeline: client, gateway, authentication, rate limits, context builder, model router, tool router, GPU inference, decoding, safety, and streaming.

The transformer is the core statistical engine, but a large part of final quality comes from the context that is assembled and the routing decisions made around it.

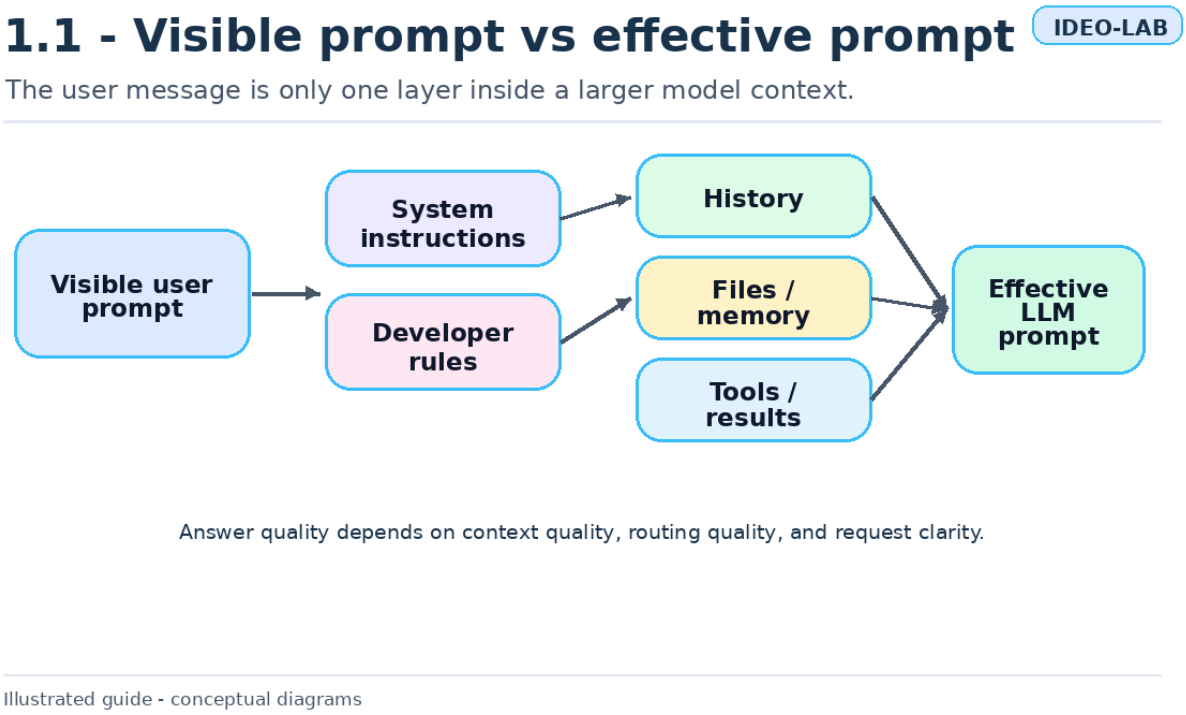
Typical layers

Layer	Role	Why it matters
Frontend	UI and streaming	Controls perceived latency
Gateway	Auth and quotas	Protects infrastructure
Context builder	Assembles prompt	Determines what the model can use
Inference cluster	GPU computation	Dominates cost and latency
Decoder	Chooses tokens	Controls style and variability
Safety layer	Policies and checks	Prevents harmful or unsafe outputs

Key insight

The service behaves like an orchestration platform around a probabilistic model. The answer is generated token by token, but the visible user experience hides many system layers.

1.1 - User prompt: the visible input is not the whole prompt



Explanatory diagram added for this section.

Visible prompt vs effective prompt

Your message is the visible request, but the model receives a larger effective prompt composed of system rules, developer rules, conversation history, memory, file excerpts, tool results, and the final user instruction.

This explains why the same sentence can lead to different responses depending on the conversation context and available files.

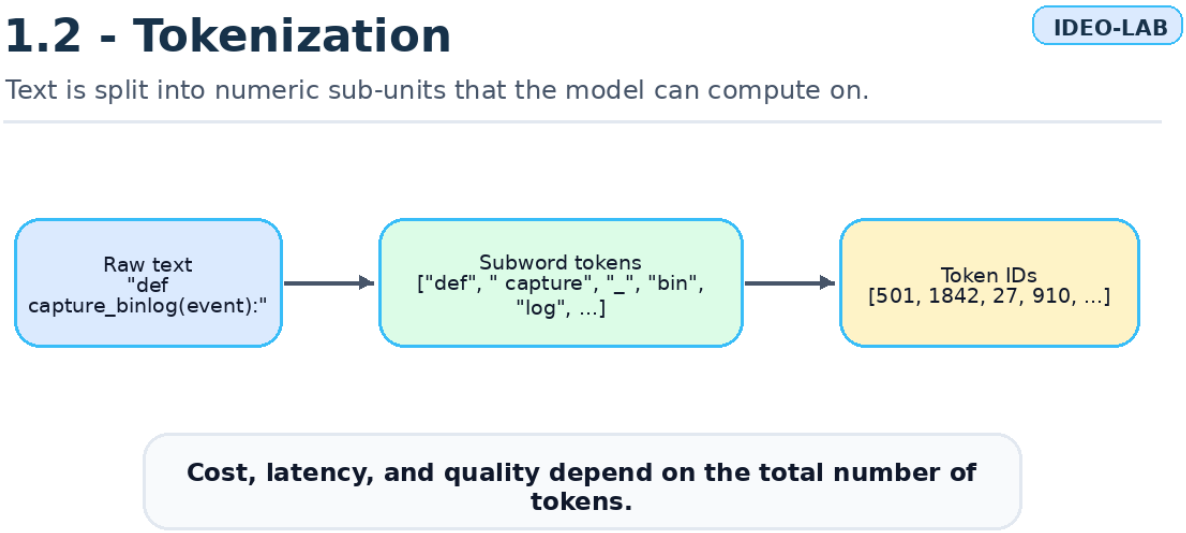
What improves technical prompts

Element	Example	Effect
Goal	Fix this Django command	Sets the task
Target file	services/binlog_capture.py	Reduces guessing
Observed error	Traceback or console log	Anchors diagnosis
Constraints	No DB migration, minimal patch	Prevents unwanted refactors
Expected output	Before/after block + test command	Makes the answer directly usable

Practical formula

Context + problem + constraints + expected format + validation test. A technical prompt should behave like a small engineering ticket.

1.2 - Tokenization: converting text into computable units



Explanatory diagram added for this section.

What the model really sees

The model does not see words directly. It sees a sequence of token IDs. Tokens may represent whole words, subwords, punctuation, spaces, indentation, or fragments of code identifiers.

For code and JSON, token count can grow quickly because punctuation, indentation, names, and repeated keys all become tokens.

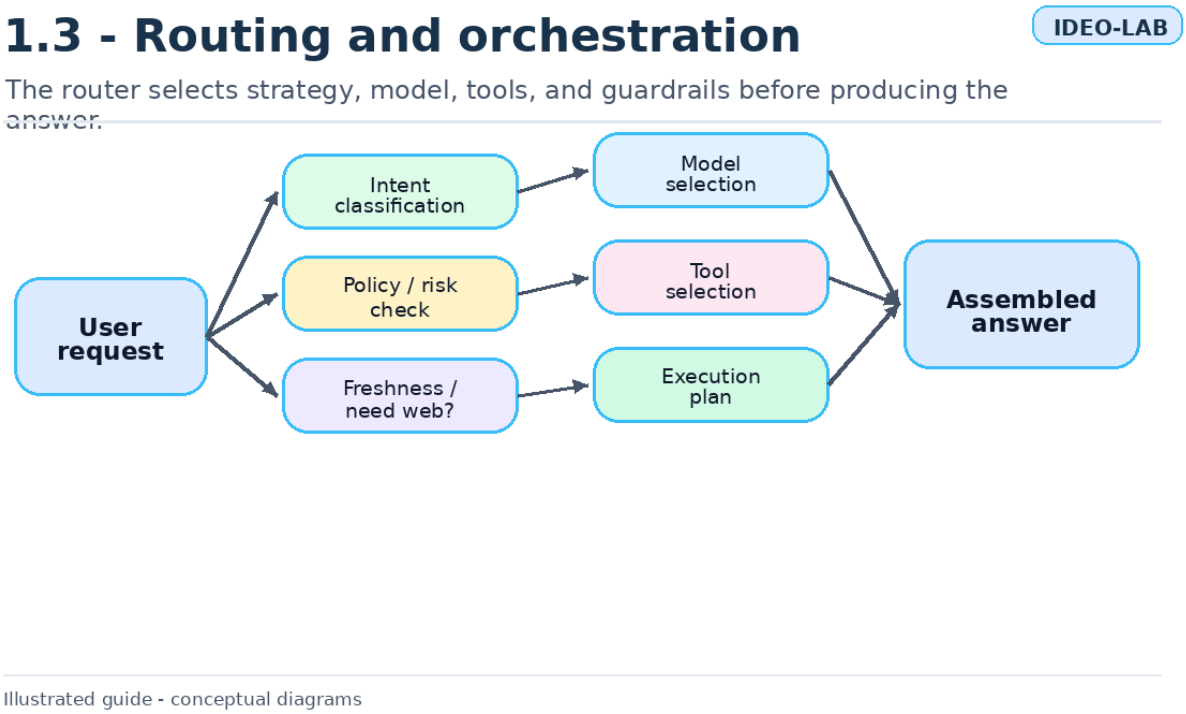
Operational impact

Factor	Impact
Input tokens	Increase cost and context processing time
Output tokens	Increase generation time
Long context	Can dilute attention and increase latency
Dense code/logs	Can become expensive and noisy

Best practice

Give the smallest useful context: the target function, relevant traceback, exact expected behavior, and only the logs needed for diagnosis.

1.3 - Routing and orchestration: model, tools, policies, strategy



Explanatory diagram added for this section.

Role of the router

The router classifies the request, decides whether freshness is required, selects a model, decides whether tools are needed, applies safety constraints, and shapes the final answer format.

A simple greeting should not consume the same path as a complex Django debugging request.

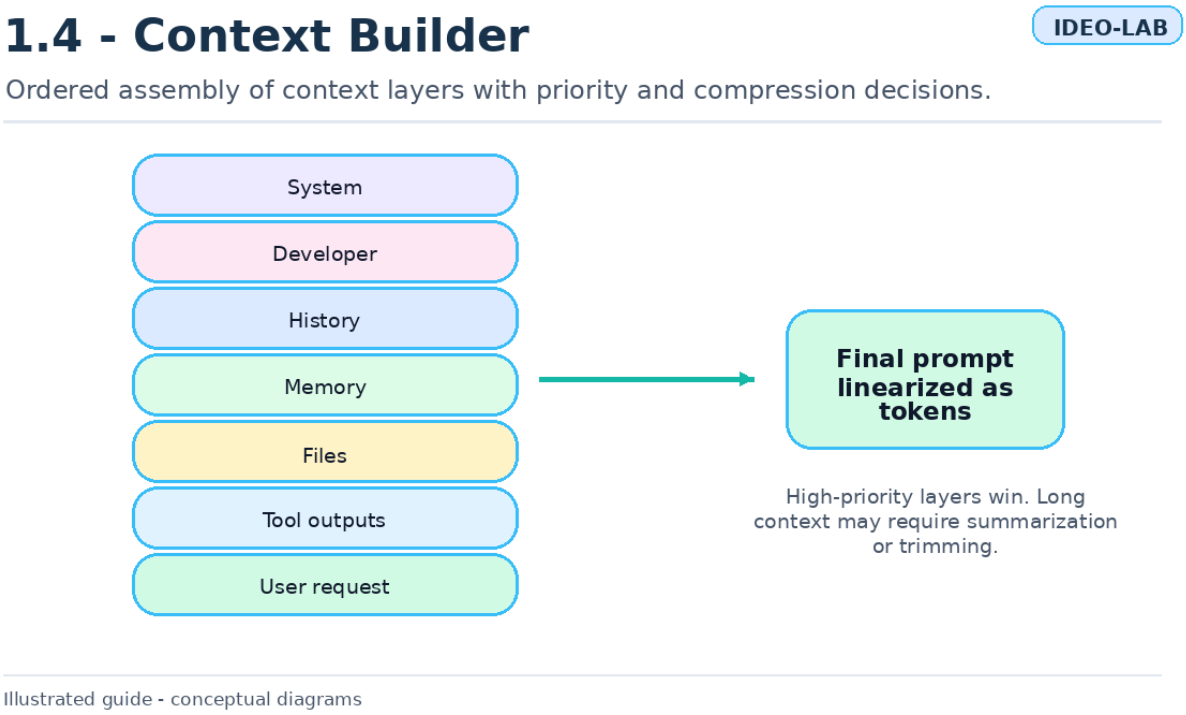
Decision layers

Router layer	Question answered	Possible effect
Intent router	What does the user want?	Code, document, image, mail, calendar, search
Freshness router	Could this be outdated?	Triggers web verification
Tool router	Is external data needed?	Reads files, runs Python, searches web
Policy router	Is there a risk?	Allows, limits, or refuses
Output router	What format is useful?	Patch, table, guide, PDF, email

Quality/cost tradeoff

Good orchestration is not about always using the largest model. It is about choosing the smallest sufficient path: right model, right context, right tools, right answer format.

1.4 - Context Builder: the real prompt sent to the model



Explanatory diagram added for this section.

Context assembly

The context builder transforms a complex conversation into a linear sequence of tokens. It must combine high-priority instructions, previous messages, memory, uploaded file excerpts, tool outputs, and the current user request.

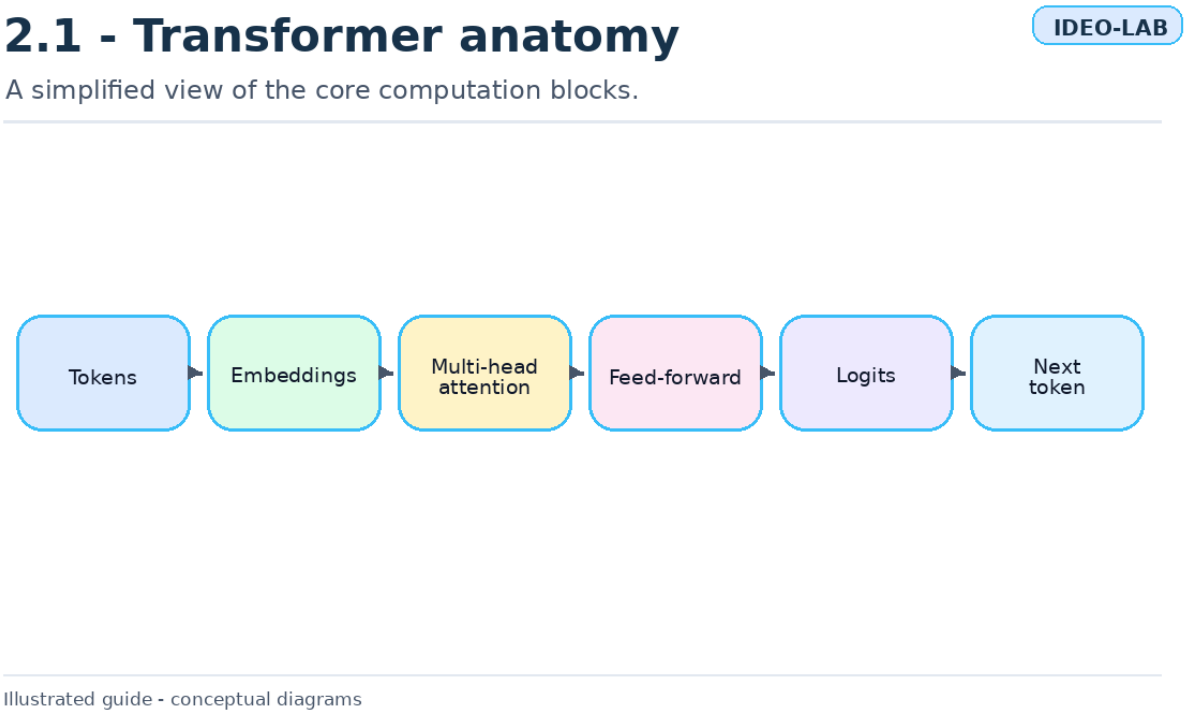
Priority and conflicts

Block	Typical priority	Purpose
System instructions	Very high	Global behavior and safety
Developer rules	High	Tool and style constraints
Conversation history	Medium	Continuity and previous requirements
Files and tool outputs	Medium/high	Grounding in real data
Current user request	High for task goal	Defines the immediate objective

Compression and trimming

When the context is too large, older or less relevant information may be summarized, selected, or dropped. This is why exact files and concise logs are so valuable in technical work.

2.1 - Transformer: embeddings, attention, layers, logits



Explanatory diagram added for this section.

Core computation

The transformer turns token IDs into embeddings, applies attention layers to relate tokens to one another, passes data through feed-forward blocks, then produces logits: scores for possible next tokens.

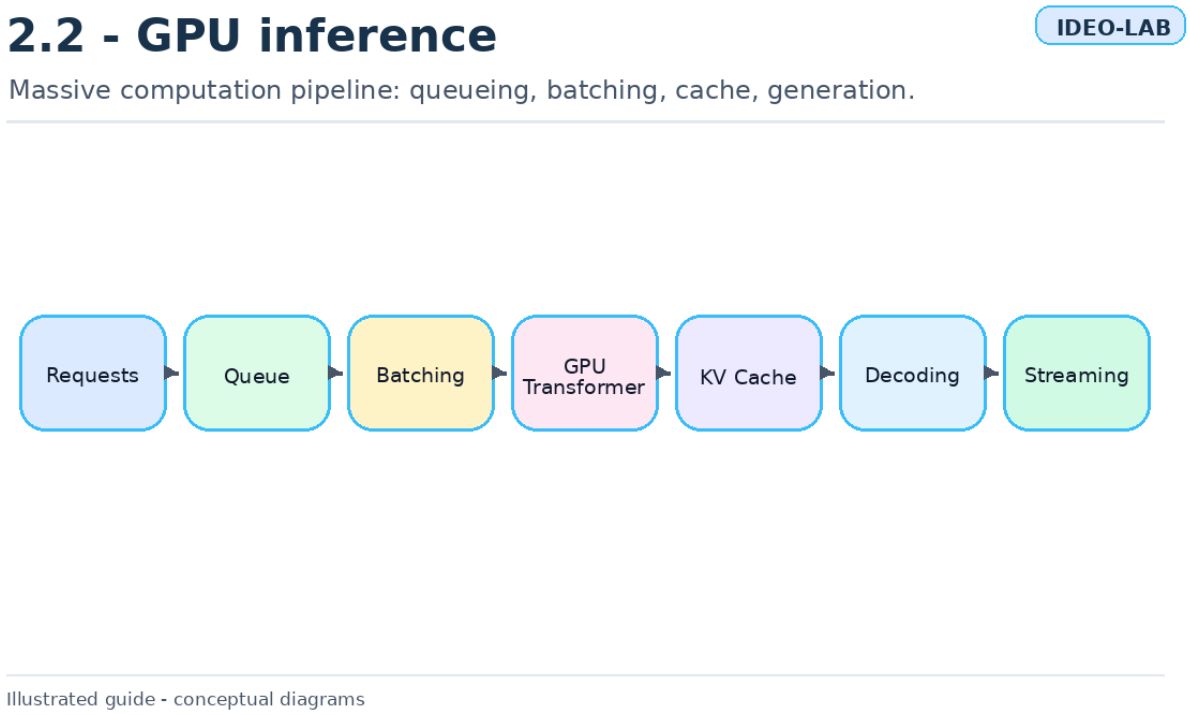
Attention in one table

Concept	Role
Query	What a token is looking for
Key	How context positions can be matched
Value	Information retrieved from matched context
Multi-head attention	Multiple relation types processed in parallel

Important limit

The model predicts plausible continuations. It does not automatically execute code or verify truth unless tools or external checks are used.

2.2 - GPU inference: batching, KV cache, scheduling, latency



Explanatory diagram added for this section.

GPU path

Inference is the runtime process of generating tokens. It uses GPU clusters, request queues, batching, memory management, and KV cache to reuse previous context computations.

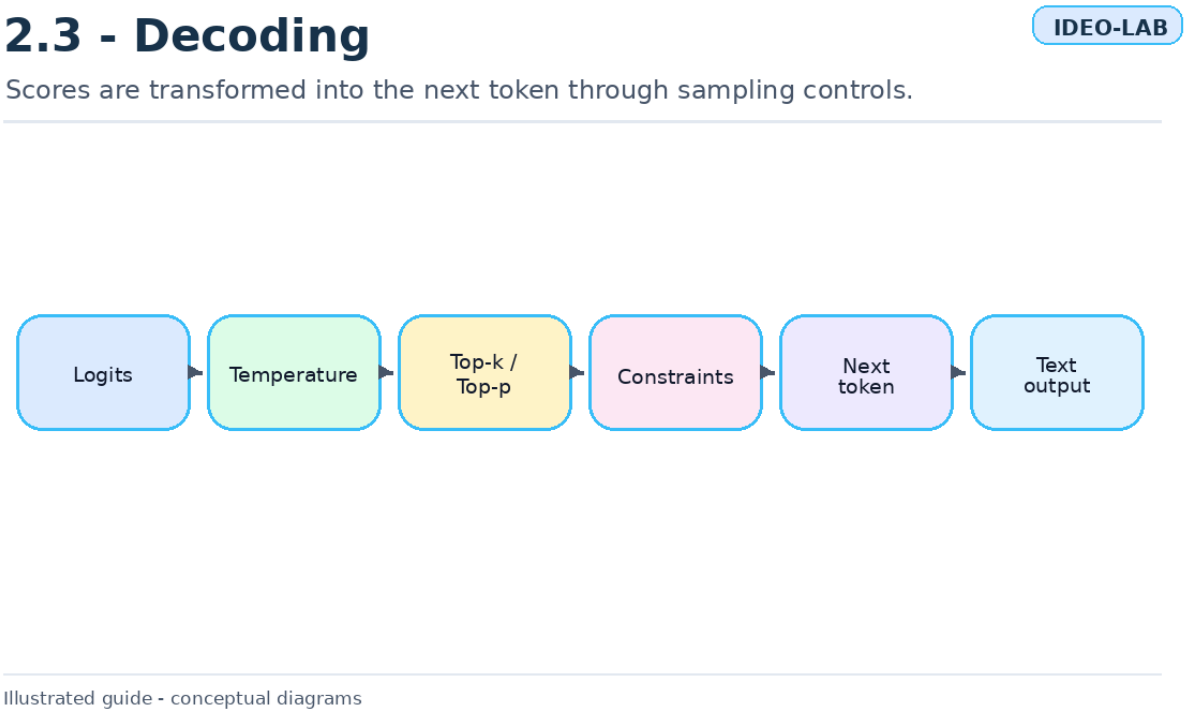
Performance factors

Factor	Effect
Context length	More tokens to process before generation
Output length	More sequential generation steps
Queue pressure	Can increase waiting time
Batching	Improves throughput but may affect latency
KV cache	Avoids recomputing previous attention states

Bottleneck

For large-scale services, the bottleneck is often GPU availability, memory bandwidth, queueing, and the number of tokens generated.

2.3 - Decoding: how the next token is selected



Explanatory diagram added for this section.

From logits to text

After the transformer computes logits, a decoding strategy selects the next token. The process repeats until a stop condition is reached.

Controls

Control	Effect
Greedy decoding	Chooses the highest-score token
Temperature	Changes randomness and creativity
Top-k	Limits choices to k candidates
Top-p	Keeps candidates within a probability mass
Stop conditions	Controls when generation ends

Practical impact

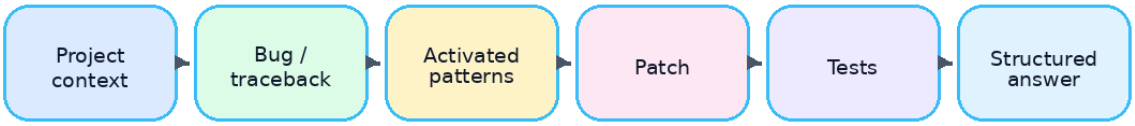
Lower randomness is useful for deterministic code-like answers. Higher randomness may help ideation, but it can increase inconsistency.

3.1 - When you ask for Python: patterns, architecture, tests

3.1 - When you ask for Python

IDEO-LAB

Typical path for a code patch or technical answer.



Illustrated guide - conceptual diagrams

Explanatory diagram added for this section.

What changes for code

For code requests, the model activates learned programming patterns: syntax, imports, project conventions, error handling, logging, and test structure. It can produce plausible code, but plausible does not guarantee executable in your exact environment.

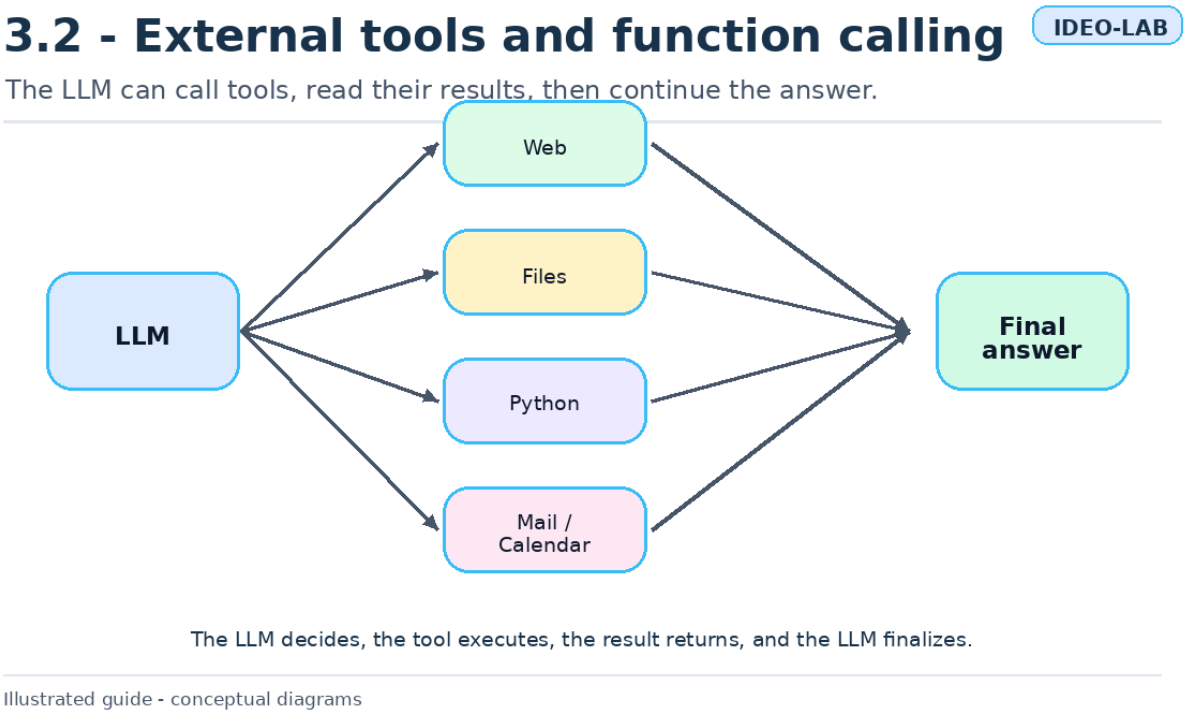
A strong patch prompt

Required element	Why it matters
Exact file and function	Targets the patch
Observed error	Prevents abstract diagnosis
Current code block	Avoids invented fields/imports
Constraints	Avoids unwanted architecture changes
Validation command	Defines success

Production mindset

For Django work, ask for file name, function name, before/after block, migration impact, admin impact, command to test, and expected console output.

3.2 - External tools: web, files, Python, mail, calendar



Explanatory diagram added for this section.

Why tools exist

The language model generates text. Tools add external capabilities: reading files, searching recent information, calculating, generating artifacts, sending email, checking calendars, or creating images.

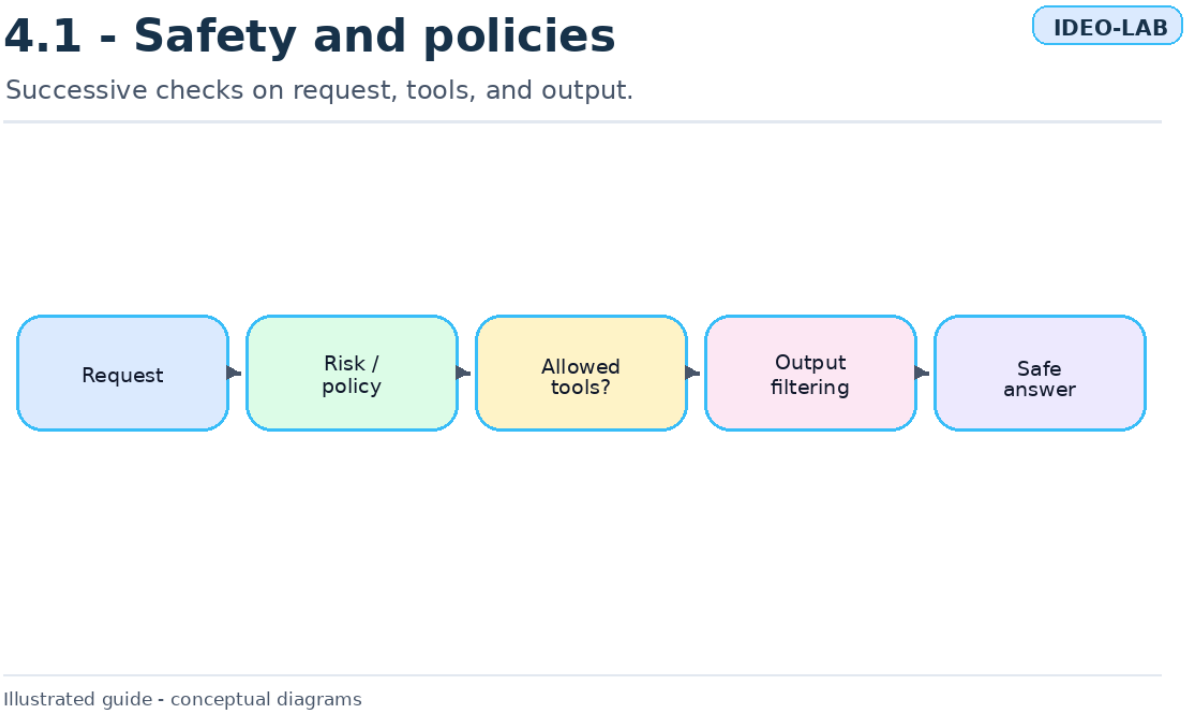
Common tool triggers

Tool	Typical trigger	Result
File search	Uploaded file or internal document	Grounded excerpts
Web	Fresh or uncertain facts	Current sources
Python	Computation or file generation	Executable result
Image generation	Create or edit visuals	Generated image
Gmail/Calendar	Personal action	Email or event operation

Loop

The LLM decides a tool is needed, calls it, receives the result, reads it, and then continues the final answer.

4.1 - Safety and policies: guardrails and controlled actions



Explanatory diagram added for this section.

Purpose

Safety layers reduce harmful instructions, unsafe code, privacy risks, destructive actions, and unsupported claims. They also decide when verification is required.

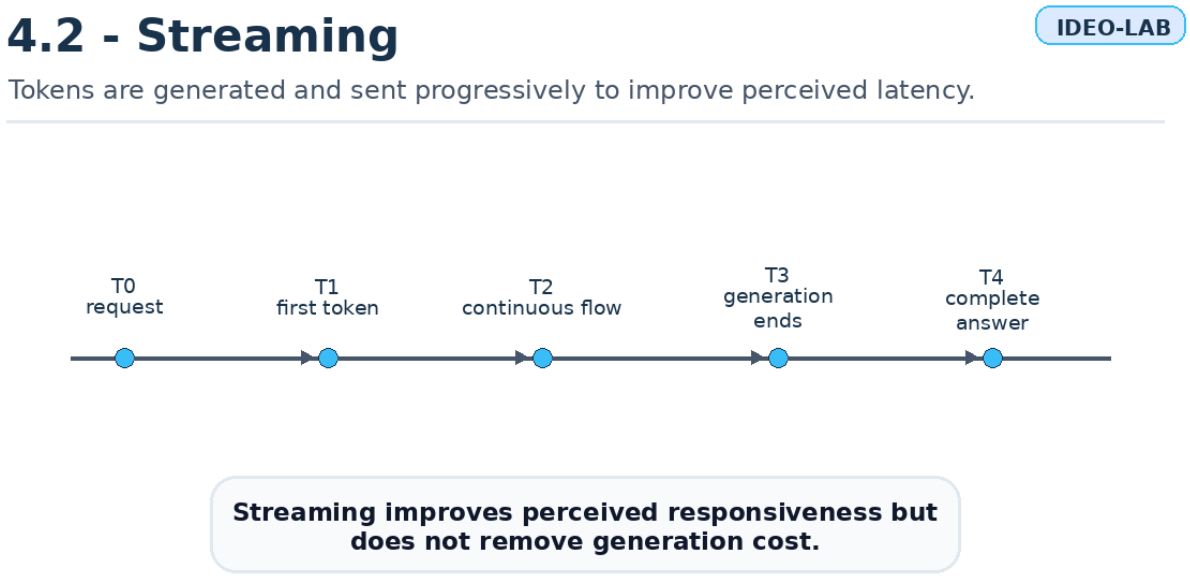
Risk examples

Risk	Example	Typical action
Outdated information	Prices, laws, versions	Search or cite sources
Private data	Emails, files	Use minimal access
Destructive action	Delete emails or events	Require explicit user intent
Unsafe code	Malware or exfiltration	Refuse or redirect
Uncertain claims	Politics, conflict, rumor	Use citations and caution

Developer impact

For normal debugging, safety should not block useful code. It mainly shapes how tools and risky operations are handled.

4.2 - Streaming: progressive token delivery and perceived latency



Explanatory diagram added for this section.

How it works

The server can send generated tokens progressively instead of waiting for the full answer. This makes the interface feel responsive while generation continues in the background of the same request.

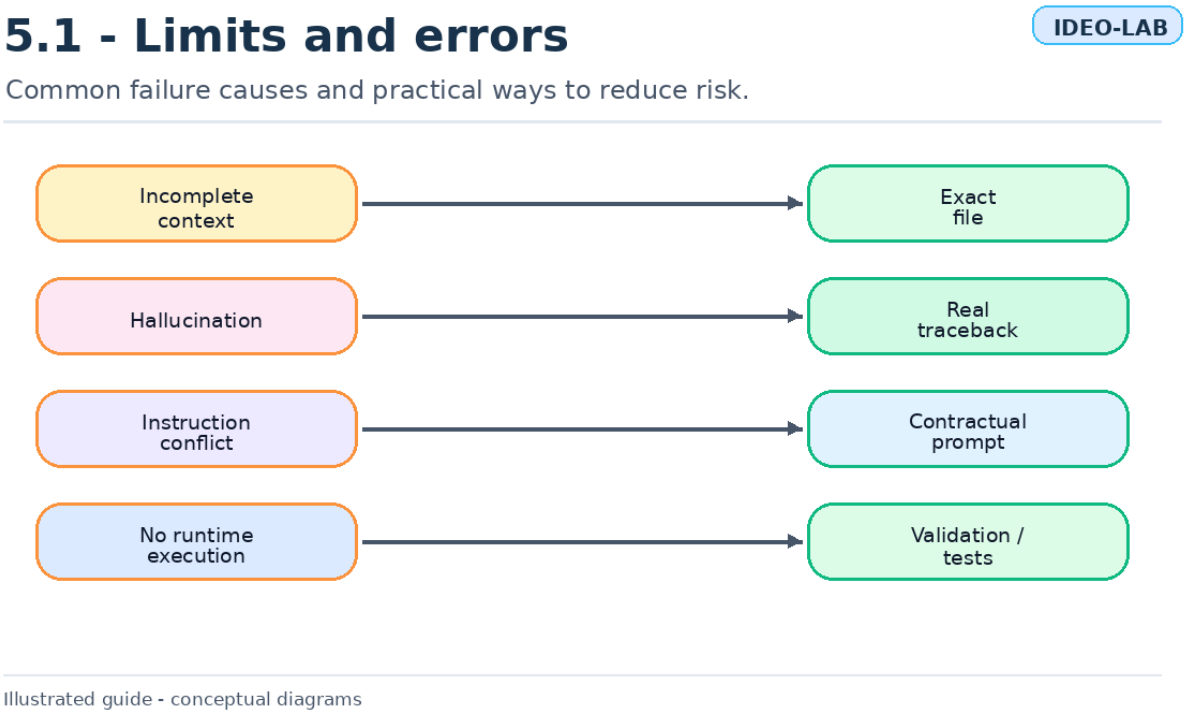
Latency components

Phase	Typical role
Request routing	Classify task and prepare path
Context build	Assemble and tokenize input
First-token latency	Time until generation starts
Token streaming	Progressive delivery
Post-processing	Final formatting or checks

UX insight

Streaming improves perceived latency. It does not make the underlying model computation free or instantaneous.

5.1 - Limits and errors: hallucinations, bugs, incomplete context



Explanatory diagram added for this section.

Why errors happen

The model predicts likely text. It may lack the exact project state, miss a hidden constraint, infer a field that does not exist, or produce a patch that looks correct but was not executed.

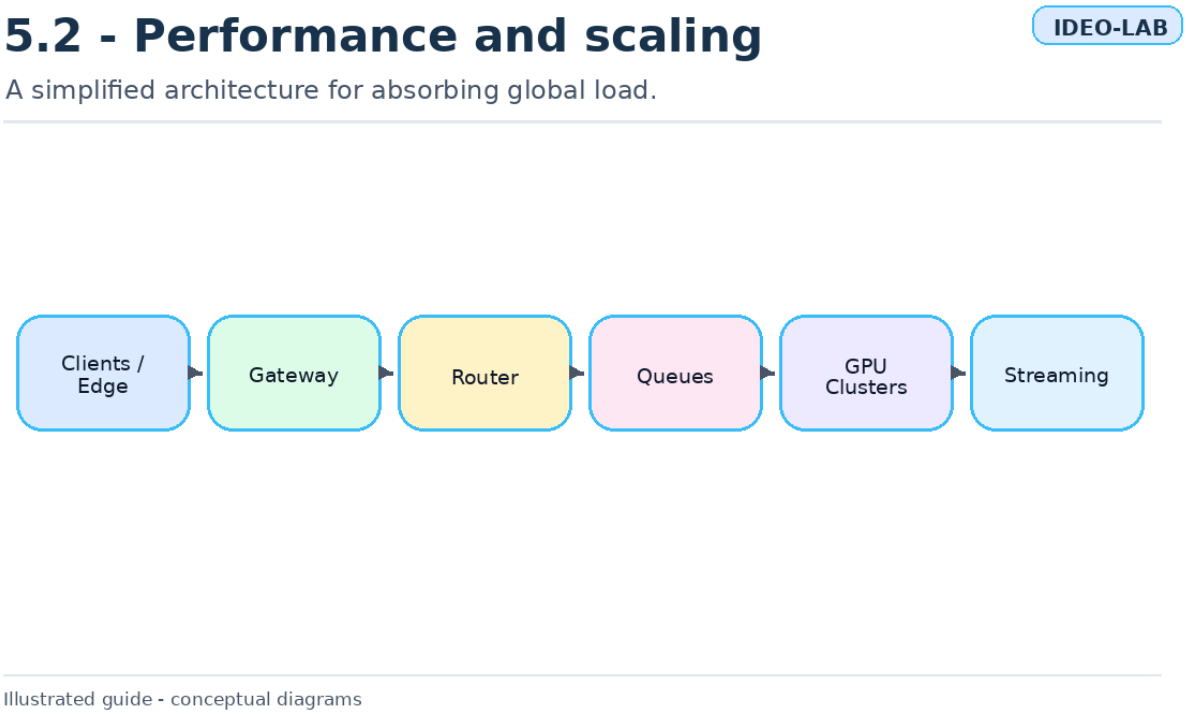
Common failures

Failure	Cause	Mitigation
Invented import or field	Missing project context	Provide current code
Wrong architecture	Prompt too broad	Request minimal patch
Outdated fact	Knowledge may be stale	Use web/source verification
Buggy code	No runtime execution	Add tests and validation
Overlong answer	Scope unclear	Specify output format

Mental model

Treat generated code as a senior-level draft that needs grounding, compilation, and verification in the real project.

5.2 - Performance and scaling: global architecture, queues, GPUs



Explanatory diagram added for this section.

Scaling view

Global LLM services rely on edge routing, API gateways, model routers, queues, GPU clusters, caching, batching, regional capacity planning, observability, and cost controls.

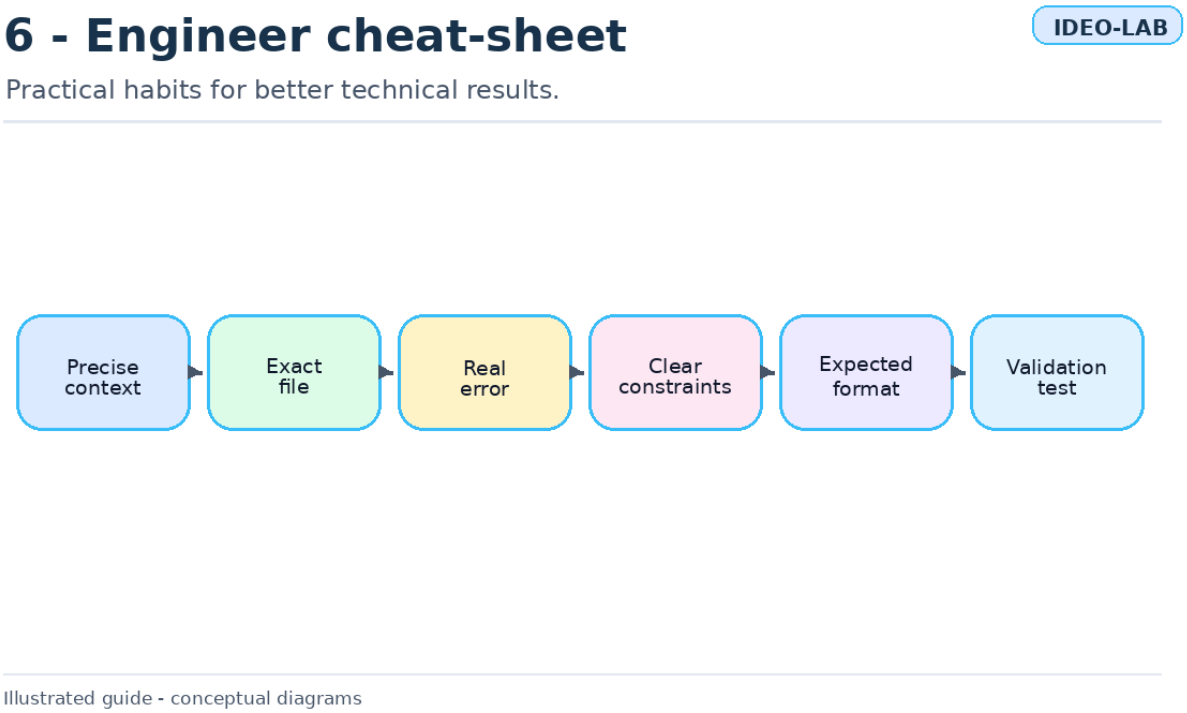
Optimization techniques

Technique	Benefit
Batching	Higher GPU throughput
KV cache	Avoids recomputing context
Speculative decoding	Can improve generation speed
Quantization	Reduces memory footprint
Traffic shaping	Protects service during load spikes
Model routing	Uses only the necessary model capacity

Constraint

Scaling is not only a machine learning problem. It is also distributed systems engineering, capacity management, and economics.

6 - Engineer cheat-sheet: practical playbook



Explanatory diagram added for this section.

Best prompt pattern

State the project context, exact target file, observed error, expected behavior, constraints, required output format, and validation command.

Fast diagnostics

Symptom	What to add to the prompt
Generic answer	Exact file and function
Wrong code	Current code block and model fields
Too much refactor	Minimal patch constraint
Unclear test	Expected command and expected output
Hallucinated fact	Ask for source or verification

Core rule

Useful LLM work is a loop: provide grounded context, receive a structured proposal, apply a small patch, run the test, feed back the real result.