AI INFRASTRUCTURE ACADEMY EUROPE

Starting Guide

for all who want to join this course series!

Understand AI applications.
Then understand what it takes to host them.

A plain-English companion to the ten AI Engineering visuals.
For datacenter operators, infrastructure teams, facility managers, CTOs, CIOs and everyone starting from the beginning.

10	illustrated chapters
22	pages of explanation
0	AI prerequisites

Start with the picture. Then read the explanation.

CONTENTS

01	The complete AI application	3
02	LLMs and inference	6
03	Prompt engineering	9
04	Fine-tuning	12
05	RAG and enterprise information	15
06	Context engineering	18
07	AI agents and MCP	21
08	LLM optimisation	25
09	Evaluation and deployment	28
10	Observability	32
	Plain-English glossary	35
	Before your first live class	37
	Sources and further reading	38

No AI background is assumed

Read the explanations in order for a complete introduction, or use the contents and PDF bookmarks to revisit a term. Technical words are defined when they first appear; no programming or model mathematics is required.

One repeated example

A fictional datacenter assistant helps an operator investigate an alert on rack R17. It gives the concepts a concrete setting. The examples are not live equipment instructions or claims about a deployed product.

What comes from the original material

The ten supplied pictures are reproduced unchanged and in their original order. The prose is a newly written teaching companion. Captions identify simplified diagrams; added technical detail is supported by the numbered references.

Read illustrative numbers correctly

The token-budget display, scorecard and dashboard contain examples—not measured Academy, Jurilo or vendor results. Calculations in the text are explicitly hypothetical. References are optional reading, not required preparation.

AI Engineering

A practical introduction for IT operators

What each layer does, how the pieces fit together, and what matters when hosting AI applications.

Agenda / Topics

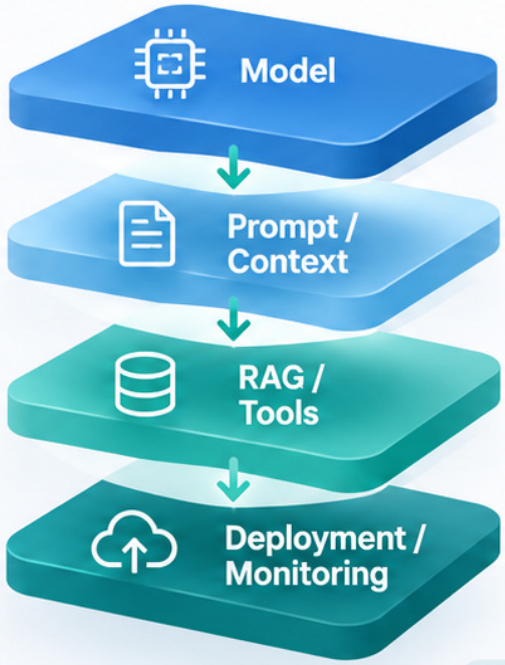
Foundations
- LLMs
- Prompt Engineering

Building AI Apps
- Finetuning
- RAG
- Context Engineering

Operating & Improving
- AI Agents
- MCP
- LLM Optimization

Running in Production
- LLM Evaluation
- LLM Deployment
- LLM Observability

End-to-End AI Application Stack



Slide 1

How to read this picture. Start with the four topic groups. Then inspect the stack on the right: it names the layers we need to understand, rather than prescribing the order of a live request.

What are we actually building?

An AI model is not the whole application

An **AI application** is software that uses an artificial-intelligence model to help complete a task. The task might be summarising an incident, finding an operating procedure or drafting a service response. The person sees an application. Behind that application are several cooperating services, and only one of them is the model.

A **model** is a trained computational system. In this guide, we mainly discuss a large language model, or LLM, which processes and generates language. An application supplies input to it, receives output and decides what to do next. The model does not automatically gain access to your documents, monitoring systems or equipment.

What "engineering" adds

AI engineering is the work of making that capability useful, testable and dependable. It includes choosing the model, preparing its input, connecting authorised data, checking results and running the whole service. That is why the picture contains many topics beyond LLMs.

The difference from a demonstration is responsibility. A demonstration can produce one impressive answer. A production service must also handle missing information, busy periods, access restrictions, software changes and failure. The Academy's central question is therefore not only "Can it answer?" but "Can we operate it responsibly?"

Our running example: an operations assistant

Imagine a deliberately limited assistant for a datacenter operations team. An operator asks: "Rack R17 has an inlet-temperature alert. Summarise what we know and identify the approved next checks." Throughout this guide, R17 and its documents are fictional teaching examples, not operating instructions.

The assistant may need the current alert, the correct site procedure, recent maintenance records and the operator's access rights. It should distinguish observed facts from possible explanations. It should not invent a temperature, assume a diagnosis or change equipment settings merely because its answer sounds confident.

What the picture's four groups mean

Foundations explain the model and its instructions. **Building AI apps** explains how to adapt behaviour and provide relevant information. **Operating and improving** covers tool use and efficiency. **Running in production** covers testing, deployment and visibility into what happens.

These groups overlap; they are a learning map, not separate departments or a compulsory project sequence. Likewise, the stack on the right is conceptual. In a live request, the application normally assembles input *before* calling the model. The next page follows that practical request path.

CHECK YOUR UNDERSTANDING If you install an LLM, have you installed a complete operations assistant?

Answer: No. The application, authorised information sources, access controls and operational processes still have to be provided.

Follow one request through the system

From a person to a controlled model call

The user begins in a web application or another interface. The application identifies the user and receives the question. An **API**, or application programming interface, is a defined way for one software service to ask another to do something. The model is often reached through such an API.

A **gateway** can sit in front of model services. It can check access, apply usage limits and direct requests to the selected service. The application then prepares the instructions and supporting information. Retrieval might find a procedure; a permitted tool might read the alert. These are distinct operations, not unexplained abilities inside the model.

From the model back to the user

The **inference runtime** is the software that runs the trained model to produce an answer. It loads the model, schedules work and uses available processors and memory. After generation, the application can validate the output, add source links and present it to the user.

That answer may finish the task. Alternatively, the application may make another model call or ask a human to approve a proposed action. A conversation that appears simple on screen can therefore involve several internal requests. We examine this explicitly in the agents chapter.

What sits underneath

A **CPU** is a general-purpose processor. A **GPU** is a processor suited to large amounts of parallel numerical work. Both can participate in AI systems. **Memory** holds data actively being used; **storage** retains files and records. The network connects the services and, in some deployments, multiple accelerators. [1,3]

For our example, the web application, document store, retrieval service and model service can be separate components. A powerful GPU will not fix an unavailable document service or an incorrect permission check. Hosting decisions must reflect the complete service, not just its most expensive component.

What an operator needs to find out

Begin with the job the application must do, who will use it and which information it may access. Then establish acceptable response time, expected simultaneous demand and the consequences of an incorrect answer. Those answers guide capacity, resilience and security choices.

Power and cooling remain essential, but there is no universal “AI rack” specification hidden in this picture. Requirements depend on the chosen equipment and workload. A small local model, a remote model API and a large multi-GPU service create very different hosting obligations. Learning the application path is the first step towards asking useful infrastructure questions.

CHECK YOUR UNDERSTANDING Which should come first: choosing the GPU or understanding the application’s workload?

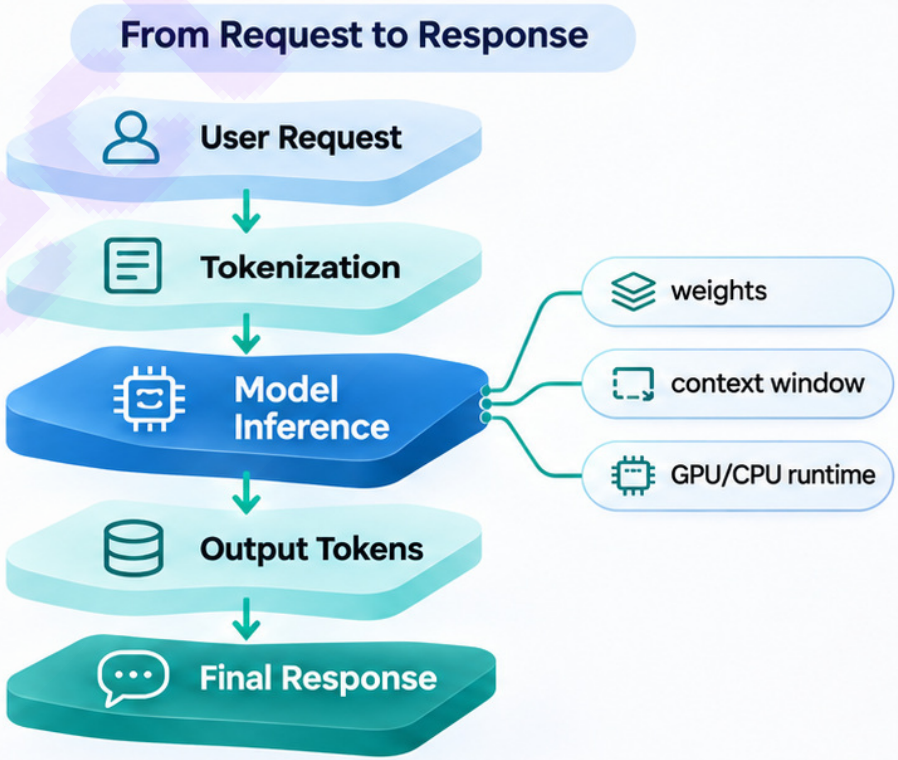
Answer: Understand the workload, service expectations and constraints first; then evaluate suitable infrastructure.

LLMs

What the model does — and what it does not do

Key Takeaways

- An LLM predicts the next token based on patterns learned during training.
- It is excellent at language tasks: summarize, explain, classify, extract, generate.
- The model brings general knowledge and reasoning style, but no live access to your systems unless connected to tools.
- For operators, the main concerns are model size, context window, latency, throughput, cost, and deployment location.



Hosting view: the model is only one layer of the application stack.

Slide 2

How to read this picture. Follow the request from tokenisation to model inference and output. The side labels identify three different things: learned weights, the available context and the runtime executing the model.

How a language model produces an answer

Language becomes tokens

A **large language model** is trained to work with language. Before it can process text, a **tokeniser** converts the text into small units called tokens and represents them as numbers. A token may be a word, part of a word or punctuation. The exact split depends on the tokeniser; there is no universal words-to-tokens conversion. [2,4]

For the R17 example, the operator's question, the instructions and any supplied procedure all become input tokens. Technical identifiers and unusual terms can split into several pieces. Counting tokens matters because model limits and much of the processing work relate to tokens rather than to the number of sentences.

The model generates a continuation

For the text-generating models discussed here, generation is typically autoregressive: the model produces a token, incorporates it into the sequence and continues. The model evaluates possible next tokens using the preceding context and its learned parameters. The selected tokens are converted back into readable text. [2]

This explains why an answer can arrive progressively rather than appearing all at once. It also explains why a fluent answer is not proof of factual correctness. The system is generating a response, not automatically checking every statement against an authoritative database. An unsupported but convincing statement is often called a **hallucination**.

What the weights contain

Weights, also called parameters, are numerical values learned during training. They determine how the model transforms input into predictions. The model can acquire language patterns and broad knowledge during training, but its weights are not a searchable copy of your current operations database.

An ordinary question does not normally rewrite those weights. Supplying a document with the question can affect the answer for that request without teaching the model permanently. Updating the weights is a different process, discussed in the fine-tuning chapter. Keeping these two mechanisms separate prevents much confusion.

What "Transformer" means here

Many language models use a neural-network architecture called a **Transformer**. Its attention mechanisms help the calculation take account of relationships between tokens. You do not need the underlying mathematics to follow this guide; the important point is that a model repeatedly performs numerical operations over representations of the input. [5]

Training creates or adjusts the model. **Inference** uses the resulting model to handle requests. An organisation can host inference without training a model from scratch. For the R17 assistant, that could mean selecting an existing model and connecting it to approved information through the application.

CHECK YOUR UNDERSTANDING Does a polished answer establish that the model has checked the current status of R17?

Answer: No. Current status must be supplied or retrieved through an authorised connection, and important claims still need verification.

What the model requires from its host

Context is the input available now

The **context window** is the amount of token-based information a model can handle in a request under its configured limits. It can include instructions, the question, selected conversation history and supplied evidence. In common autoregressive serving setups, you must also reserve room for generated output; exact limits depend on the model and service. [1,4]

A conversation can appear continuous because the application sends relevant earlier material again or maintains state. This is different from the model learning permanently. When older material is omitted, the current model call cannot reliably use it. The application therefore has to decide what remains relevant.

Reading input and producing output

Inference is commonly described in two stages. **Prefill** processes the supplied input. **Decode** generates the continuation. A **KV cache** retains intermediate attention data so the model need not redo all the same calculations for every new token. This is working data, not a store of approved final answers. [1,6]

Longer input, longer output and more simultaneous requests can change memory demand and response time. The exact effect depends on the model, runtime and hardware. This is why an isolated test with a short question is not enough to size a shared production service.

Measure two different kinds of speed

Time to first token measures how long the user waits before generation starts appearing. **Tokens per second** describes a generation rate; always specify whether it is for one request or the entire service. Total response time also includes retrieval, queuing and other application work. [7]

An assistant that starts quickly but produces a very long answer may still feel slow. A service with high aggregate throughput can also have a poor experience for an individual user. Start with the user's task: our operator needs a short, timely, supported summary, not the longest possible response.

Model size is only part of capacity planning

A larger parameter count usually means more weight data to hold, but it is not the complete memory requirement or a guarantee of better results. Runtime memory, cached intermediate data and the chosen numerical representation also matter. Hosting on CPUs can be appropriate for some models and service requirements; not every application needs a large GPU installation. [3]

For an initial assessment, ask for the exact model and runtime, expected input and output sizes, simultaneous demand and a representative test. Also ask what happens when capacity is exhausted. A controlled queue or a clear refusal is preferable to silently losing requests.

CHECK YOUR UNDERSTANDING Is the KV cache the same as a stored chat history or an answer cache?

Answer: No. It holds intermediate inference data; conversation records and reusable final responses are separate application concerns.

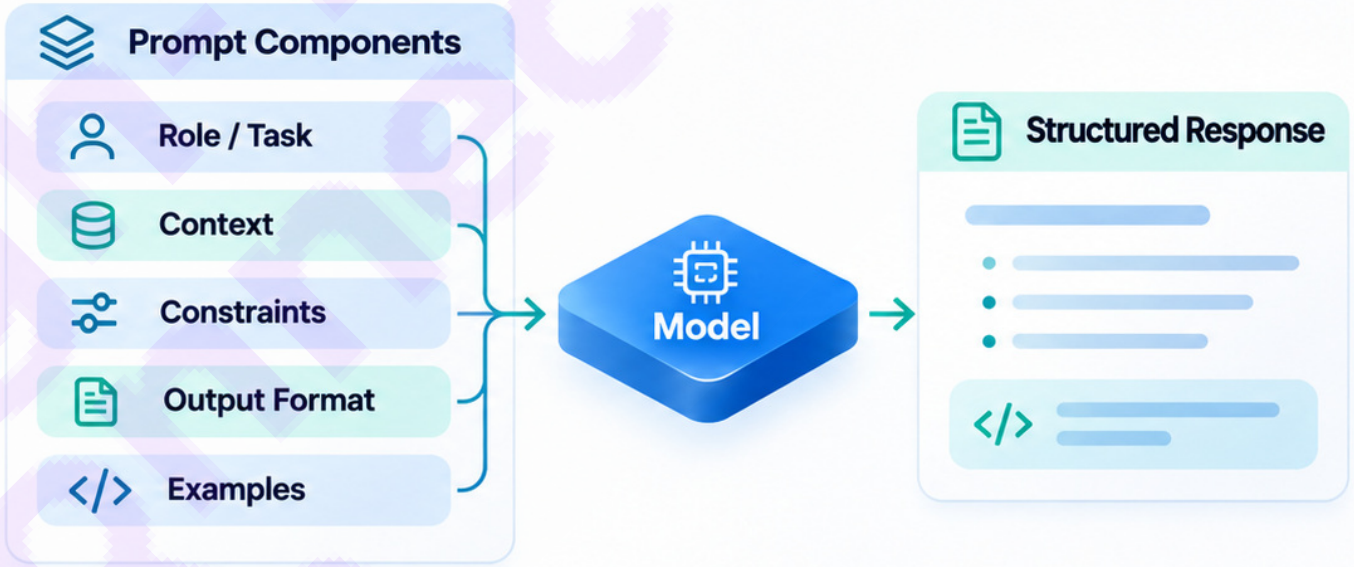
Prompt Engineering

How we guide the model to produce useful answers

Key Takeaways

- A prompt is the instruction package sent with the user request.
- Good prompts define the task, the desired output format, relevant constraints, and important context.
- Prompting improves reliability, but it does not replace missing knowledge or poor source data.
- For production systems, prompts should be versioned, tested, and treated like application logic.

Typical pattern: system prompt + user input + retrieved context



What to manage: templates, guardrails, structured outputs, prompt version control.

Slide 3

How to read this picture. Read the five prompt components as a task specification: role, context, constraints, output format and examples. The model receives the assembled request; software must still validate and enforce the result.

Give the model a clear job

A prompt is the request package

A **prompt** is the input used to guide a model's response. In an application it usually includes more than the sentence a user typed. Application instructions, selected evidence and formatting requirements can all be part of the package. Prompt engineering means making that package clear, appropriate and testable. [8]

The role or task tells the model what kind of work to perform. Context provides information it needs. Constraints establish boundaries. The output format specifies how the result should be presented. Examples demonstrate what an acceptable response looks like. These are the five inputs shown in the picture.

A vague request leaves important decisions open

Consider "Explain this alert." The model does not know how much detail the reader needs, what evidence is trusted or whether it may suggest actions. It could produce a generic explanation that sounds helpful but fails to address the actual situation.

A more precise task would be: "Summarise the supplied R17 alert for the on-duty operator. Separate observed facts from possible explanations. Use only the supplied current records for factual claims. Identify missing information. Give the document IDs supporting the proposed checks. Do not execute changes." This is an illustrative teaching prompt, not a production safety policy.

Format makes the response easier to use

An application can request sections such as **Observed facts**, **Missing information**, **Approved checks** and **Sources**. The result is easier for a person to review because the answer has a predictable structure. For software integration, an application might request JSON: a structured text format with named fields.

For example, a JSON object might have fields called "asset_id", "summary" and "source_ids". A well-formed object is easier to process than unrestricted prose, but the content can still be wrong. Format validation asks whether required fields exist. Semantic validation asks whether their values make sense.

Examples clarify expectations

A small set of input-and-output examples can show how to handle uncertainty or use the required format. This is often called **few-shot prompting**. The examples are included in the request; they are not, by themselves, a training run that updates model weights. [8]

For our assistant, one example could show a useful refusal: "The procedure supplied is for another site; request the correct version." Another could show an answer with clear source references. Examples should reflect the behaviour actually wanted, including difficult cases, rather than only polished successes. The purpose is a clearer specification, not a special phrase that guarantees correctness.

CHECK YOUR UNDERSTANDING If examples are included in a prompt, has the model been fine-tuned?

Answer: No. Examples in the request influence that request; fine-tuning updates trainable model parameters through a training process.

Treat prompts as part of the application

Separate rules from supplied material

Applications commonly distinguish higher-priority instructions from user messages and retrieved content. The exact message roles differ between model interfaces. The useful principle is consistent: operating rules and untrusted source material should not be treated as the same kind of input.

A retrieved document might contain instructions aimed at manipulating the assistant rather than information relevant to the task. This is a form of **prompt injection**. Labelling a document as evidence helps define intended behaviour, but a prompt is not an access-control boundary. Permissions must be enforced by the application and connected services. [9]

Make prompt changes reviewable

Imagine changing “suggest the next check” to “resolve the incident”. That small wording change could alter the actions the system attempts. Store prompt templates with identifiers and versions, and keep a record of which version was used for each request.

Before release, compare the old and new versions on the same representative cases. Include missing records, conflicting procedures and requests outside the assistant’s remit. Test not only whether the response sounds good, but whether it obeys the intended limits and supplies enough evidence. Keep a known working version available for rollback.

Validate before anything important happens

A **guardrail** is a control intended to keep behaviour within a defined boundary. Some controls inspect generated text; others are deterministic software checks. For the example assistant, a deterministic check can verify that an asset exists or that the user is allowed to access a source. The model should not be the sole judge of its own permissions.

Where structured output is supported, check its schema: the expected fields, types and permitted values. Then check business meaning. An invented source ID can be perfectly valid JSON while still making the answer unusable. A failed validation should trigger a controlled response, not be silently accepted.

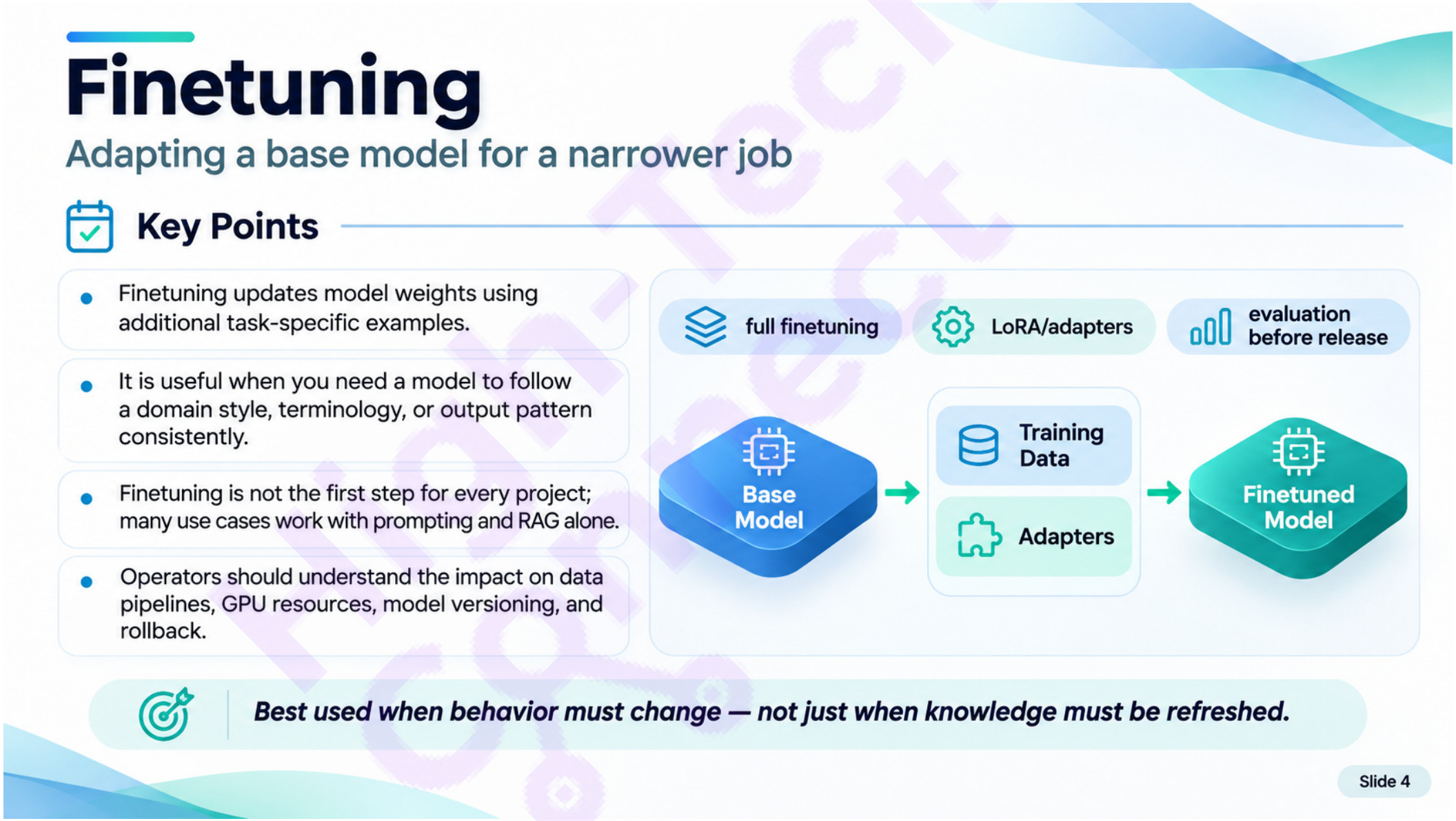
Know which problems prompting cannot solve

A clearer prompt cannot supply a missing monitoring connection, repair an outdated procedure or create more accelerator memory. It can ask the model to admit missing information, but the application must provide the actual data path and resources.

Use prompting to specify the job. Use retrieval or tools when the job needs external information. Consider fine-tuning when a measured behaviour problem persists across many examples. Then evaluate the complete change. This division of responsibilities is more useful than assuming every failure can be fixed by making the instructions longer.

CHECK YOUR UNDERSTANDING Why is “Never access unauthorised records” insufficient as a security control?

Answer: It is an instruction, not enforcement. The data service must independently check the user’s permissions before releasing records.



How to read this picture. The picture shows a base model being adapted with training examples. Full fine-tuning and adapters are different approaches; the output may be a complete model or a base model used together with an adapter.

What changes during fine-tuning?

Start with an already trained model

Fine-tuning means continuing training on an existing model using data selected for a narrower purpose. The starting model is the **base model**. Instead of merely supplying instructions for one request, a training process adjusts trainable parameters so later behaviour can change. [5]

For our operations assistant, a possible task is converting incident notes into a consistent, approved handover structure. Training examples could pair messy notes with reviewed summaries. The objective would be a more reliable transformation pattern, not teaching the model today's live R17 temperature.

Full fine-tuning and smaller adaptations

In **full fine-tuning**, many or all model weights can be updated. This creates training work and associated memory requirements. In **parameter-efficient fine-tuning**, only a smaller set of parameters is trained. The goal is to adapt the model without updating every original weight.

LoRA, short for low-rank adaptation, is one such method. It trains smaller update matrices while keeping the original weights frozen. Depending on the deployment, the adaptation can remain separate or be merged into a compatible model. The picture's "adapters" box therefore does not necessarily mean replacing the entire base model. [10]

Examples must teach the intended behaviour

The value of the training data comes from its relevance and quality, not simply its size. A summary that omits important uncertainty can teach the wrong lesson. A historical ticket that contains an incorrect diagnosis should not become an unquestioned target answer.

For an illustrative dataset, reviewers could identify the observed facts, the approved vocabulary and the proper handling of absent evidence. Records should be authorised for this purpose and appropriately prepared. Remove unnecessary sensitive details and track where examples came from. This is a data-governance task as well as a model task.

Training and testing must be separated

Use some examples to adapt the model, and reserve other examples to test whether it generalises. A model that reproduces training examples well may still fail on unfamiliar incidents. **Overfitting** means fitting the training cases too closely at the expense of useful behaviour elsewhere.

Also test capabilities that must not deteriorate. Better formatting is not a success if factual care or uncertainty handling gets worse. Evaluation should compare the adapted model against the original model on the same task set. A new model artifact is a candidate for release, not evidence that the application has improved.

CHECK YOUR UNDERSTANDING What stays frozen in a typical LoRA adaptation?

Answer: The original model weights; the smaller added adaptation parameters are trained.

Choose fine-tuning for the right reason

Behaviour and knowledge are different problems

The slide's closing sentence is a useful rule of thumb: fine-tuning is often considered when behaviour must become more consistent, while retrieval is usually easier to update when the application needs changing documents. It is not an absolute law: training can affect knowledge too, but it is not a dependable substitute for a current, permission-aware record system.

For R17, the correct procedure can change tomorrow. Keeping that procedure in a controlled repository lets the application retrieve the current version. Re-training a model every time a document changes would introduce another release process and would not by itself provide source-level access control.

Ask what problem remains after a simpler baseline

Suppose a clear prompt and suitable examples already produce acceptable summaries. Fine-tuning may add cost without solving a real problem. Suppose, instead, the output repeatedly violates an important formatting convention despite careful prompting. That is a more specific behaviour gap to investigate.

The decision should follow observed failures. Define the required improvement, choose an evaluation set and compare alternatives. Prompting, retrieval and fine-tuning are not competing products; they are different mechanisms that can be combined. The right combination depends on the task and the evidence from testing.

What an infrastructure team must plan

Training is a separate workload from serving user requests. It may need accelerators, temporary storage, checkpoints and a schedule that does not disrupt production. A **checkpoint** is a saved training state or model snapshot used to preserve progress or produce a release artifact.

After adaptation, record the base-model version, adapter version, tokenizer, runtime and configuration. A small adapter is not automatically compatible with any model carrying a similar name. Test the actual combination that will be deployed and retain the information needed to reproduce it. [10]

Plan a reversible release

Keep the old and new versions clearly identifiable. Release to a limited audience first, monitor both quality and operational behaviour and retain a rollback route. A rollback must restore the working combination, not just one file while leaving an incompatible runtime or prompt in place.

For our example, the release decision might be: the adapted model produces clearer handovers, maintains source discipline and meets the agreed response-time target under realistic load. Without that evidence, "we fine-tuned it" describes an activity, not an outcome. The operator's interest is the service that results.

CHECK YOUR UNDERSTANDING Your procedures change weekly. Is that alone a reason to fine-tune every week?

Answer: No. Start by considering a maintained retrieval or tool-based data path; evaluate fine-tuning against a specific behaviour need.

RAG

Retrieval-Augmented Generation adds external knowledge at runtime

Key Points

- RAG retrieves relevant documents or records and supplies them to the model as context.
- This is how an AI app answers with company-specific, current, or regulated information.
- RAG usually includes ingestion, chunking, embeddings, vector search, reranking, and answer generation.
- Quality depends on the source documents, retrieval accuracy, and context assembly — not just the model.
- For hosting, pay attention to storage, indexing jobs, refresh cycles, latency, access control, and data privacy.

RAG Pipeline



RAG changes the application architecture, not the model weights.

Slide 5

How to read this picture. The left part of the pipeline prepares documents; the right part uses selected material to answer a question. The picture joins these stages for simplicity, but document preparation does not normally repeat in full for every user question.

Make documents available to the application

What RAG adds

Retrieval-augmented generation, or RAG, combines information retrieval with model generation. The application searches an external source, selects relevant material and includes that material in the input to the model. The model can then produce an answer informed by those sources without changing its weights. [11]

For the R17 assistant, the relevant source might be an approved operating procedure. RAG helps put the right part of that procedure in front of the model. It does not automatically establish that the procedure is current, appropriate for this site or correctly interpreted.

Ingestion prepares the source material

Ingestion means bringing documents into a processing pipeline. Text may need to be extracted from files, cleaned and associated with metadata such as document title, site, version, owner and access rights. Metadata is information describing a record; it is not the record's main text.

Long documents are often divided into smaller sections called **chunks**. Retrieval can then return a relevant section rather than an entire manual. Chunking must preserve enough context to remain meaningful. A paragraph stating an exception should not be separated from the rule it qualifies without a way to reconstruct the relationship. [12]

Embeddings support similarity search

An **embedding** is a numerical representation of content. An embedding model maps text into a vector: an ordered list of numbers. Comparing vectors can help find passages with related meaning even when they do not contain exactly the same words. No vector mathematics is required to understand the purpose. [11,12]

The embedding is not the original document. The application still needs the original text, identifiers and permission information. A similarity result is also not proof of relevance or truth. "Rack inlet alarm" and "room temperature warning" may be related, yet belong to different procedures.

A vector database is one possible component

A **vector database** stores and searches vector representations together with associated information. Some search systems combine this with ordinary keyword search, which can be particularly helpful for exact asset identifiers or error codes. RAG does not require a particular database product or even vector search in every implementation. [11]

As an operator, distinguish the document store from the search index. The store preserves authoritative material; the index helps find it. Both need ownership and update procedures. A working index over obsolete manuals is still an obsolete information service. Deletions and permission changes must propagate as well as additions.

CHECK YOUR UNDERSTANDING If an embedding is stored, can the application discard the document and its permissions?

Answer: No. It still needs the source content, provenance and access controls to retrieve and present information responsibly.

Retrieve evidence, then generate an answer

The online request path

When the operator asks about R17, the retrieval service searches for potentially useful material. If vector search is used, the question is represented in a compatible embedding space. Filters can restrict results to the authorised site, asset type or document version. The important security requirement is that inaccessible records do not reach the model.

A **reranker** can then examine candidate passages more closely and reorder them by relevance. This is a second selection stage, not a guarantee that the best document exists. The selected passages, source identifiers and question are assembled into the model's context. The model generates an answer from that package. [11,12]

A source link is useful, but not proof

An answer may cite a procedure while misunderstanding it. Checking **grounding** means checking whether the supplied evidence actually supports the claims made. A generated reference that merely looks like a real document ID is not sufficient.

For the example assistant, reviewers should be able to open the cited version and see which passage supports a recommended check. If no suitable source is found, the application should communicate that limitation rather than fill the gap with a plausible operating procedure.

Troubleshoot the chain, not just the model

When an answer is wrong, ask whether the right document existed, whether ingestion preserved its meaning, whether retrieval found it, whether the context included it and whether generation used it correctly. Those are different failure points with different fixes.

An index-refresh failure can hide yesterday's approved procedure. Bad chunking can separate a warning from its conditions. Retrieval can select the wrong site. A prompt can ask for certainty when the source is ambiguous. Buying a larger model will not reliably correct those upstream problems.

What reliable RAG hosting involves

Plan for ingestion jobs, searchable indexes, document storage, embedding or reranking services and the generation service. These can have different performance and availability needs. Monitor document freshness and retrieval failures as well as GPU utilisation.

Keep user and tenant permissions attached to the retrieval path, and review what may leave the organisation when a remote model is used. RAG does not remove the need for privacy controls or evaluation. The practical objective is a traceable answer built from appropriate evidence, with a clear failure mode when the evidence is unavailable. That is a service requirement, not just a search feature.

CHECK YOUR UNDERSTANDING Why can a RAG answer be wrong even when the language model is functioning normally?

Answer: The source may be stale, unauthorised, missing or incorrectly retrieved; the answer may also misinterpret the retrieved evidence.

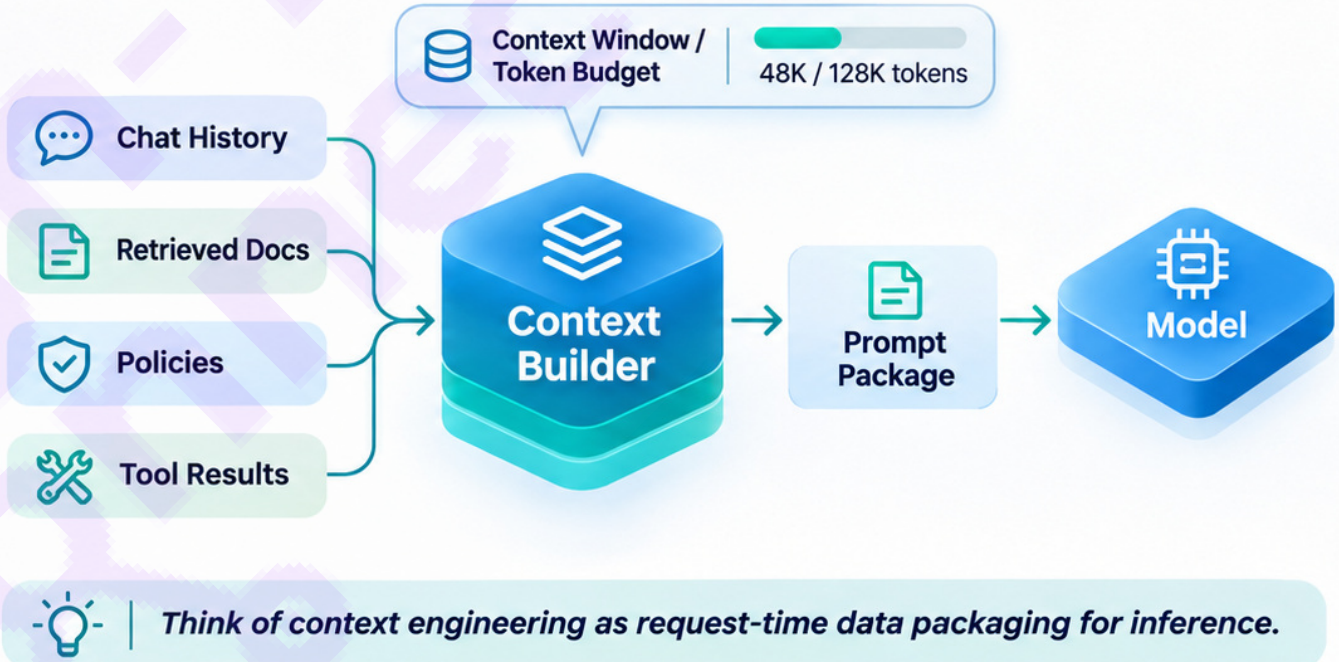
Context Engineering

Delivering the right information to the model at the right time

Key Takeaways

- Context engineering is broader than prompting: it decides what information, memory, history, and tool results reach the model.
- The goal is relevance, not volume — too much context increases cost, latency, and confusion.
- Typical inputs include user history, retrieved documents, structured data, policies, and recent tool outputs.
- For operators, key concerns are context window limits, caching, session state, security boundaries, and orchestration logic.

Context Assembly Flow



Think of context engineering as request-time data packaging for inference.

Slide 6

How to read this picture. Follow the inputs into the context builder. It selects and packages information for a particular model call. The 48K / 128K display is an illustration of a token budget, not a specification for the Academy or any named model.

Give the model the right information now

Prompting and context are related, not identical

Prompt engineering specifies the task and desired response. **Context engineering** is the broader work of deciding which information the model receives when it performs that task. It includes instructions, selected conversation history, retrieved documents, tool descriptions and tool results. [13]

For the R17 assistant, a well-written instruction is not enough. The request also needs the right alert, the right procedure and enough recent history to interpret the question. Context engineering decides what to include, what to leave out and how to identify the material clearly.

Read the four inputs in the picture

Chat history preserves relevant earlier exchanges. If the operator said "I mean the north hall", the next request may need that clarification. **Retrieved documents** provide selected evidence. **Policies** provide applicable instructions or constraints. **Tool results** provide information returned by connected systems.

These inputs have different authority and freshness. A live alert and an old discussion about a similar alert should not be presented as interchangeable facts. Clear labels can preserve the distinction: source, time, site, version and whether the information is observed or inferred.

The context builder is application logic

The picture's **context builder** need not be a special commercial product. It represents the software and rules that assemble the request. The builder might keep the latest question, select a few relevant passages, include recent tool results and omit unrelated conversation.

This process happens for a particular model call. When the task changes or a tool returns new data, the package may change too. The model does not independently know every record the application has available. Only the selected material, together with the model's learned capabilities, can influence that call.

More information is not always better

A large context window is a capacity limit, not a promise that every included passage will be used correctly. Irrelevant text increases processing and can make it harder to identify the evidence that matters. Context design therefore aims for useful information, not maximum occupancy. [13]

For example, five current, asset-specific records may be more useful than a thousand pages of loosely related manuals. Removing irrelevant material is not hiding information from the user; it is making a deliberate selection for a constrained computation. The selection itself should be testable, especially when a missed exception could change the answer.

CHECK YOUR UNDERSTANDING What is the difference between a good prompt and a good context package?

Answer: The prompt specifies the job; the full context package also supplies the relevant evidence, history and tool information for that call.

Manage the budget, state and trust boundaries

A token budget is a practical allocation

Suppose a fictional deployment permits a combined 32,000-token sequence. Reserving 2,000 tokens for the answer leaves at most 30,000 for its input under that assumed configuration. Instructions, history, retrieved passages and tool information must fit within that remainder. This is a teaching calculation, not a recommended model setting.

The picture's "48K / 128K" has the same illustrative purpose. It does not show a measured optimum. In a real deployment, check the model's limit, the serving configuration and any separate output limit. Do not assume that buying a service with a large advertised context window makes unrestricted input economical.

Conversation state has to live somewhere

State is information retained between steps or requests. It can include conversation messages, task progress or approved user preferences. A database may store that state even when the model process itself is restarted. The application decides which parts to retrieve for the next call.

Long conversations may be shortened through selection or summarisation. A summary can omit an important detail, so retain authoritative records separately and retrieve them when needed. Persistent memory is a designed application feature; it is not evidence that the model's weights changed during the conversation.

Caching needs a precise meaning

The word **cache** can refer to different things. A response cache stores a previous answer. A retrieval cache stores search results. A prefix cache reuses inference calculations for an identical starting sequence. These caches have different purposes and invalidation requirements. Prefix caching does not simply return a previous answer. [14]

For R17, a previous summary may no longer be valid after the alert changes. Cache reuse must also respect permissions: one user's authorised view must not become another user's shortcut around access control. A performance improvement that leaks data is not an acceptable optimisation.

Preserve boundaries as context moves

The builder should not mix evidence from different customers or sites merely because the passages look similar. Keep credentials out of natural-language context where possible, and ensure connected services enforce their own authorisation. Tool results and external documents remain potentially untrusted input.

A useful review asks: What information entered this request? Who was allowed to see it? Which version was used? What was omitted? That review can explain both quality problems and unexpected token growth. Context engineering is therefore closely connected to security, observability and capacity planning, not just to how a sentence is worded.

CHECK YOUR UNDERSTANDING Does a 128K context limit mean every request should contain 128K tokens?

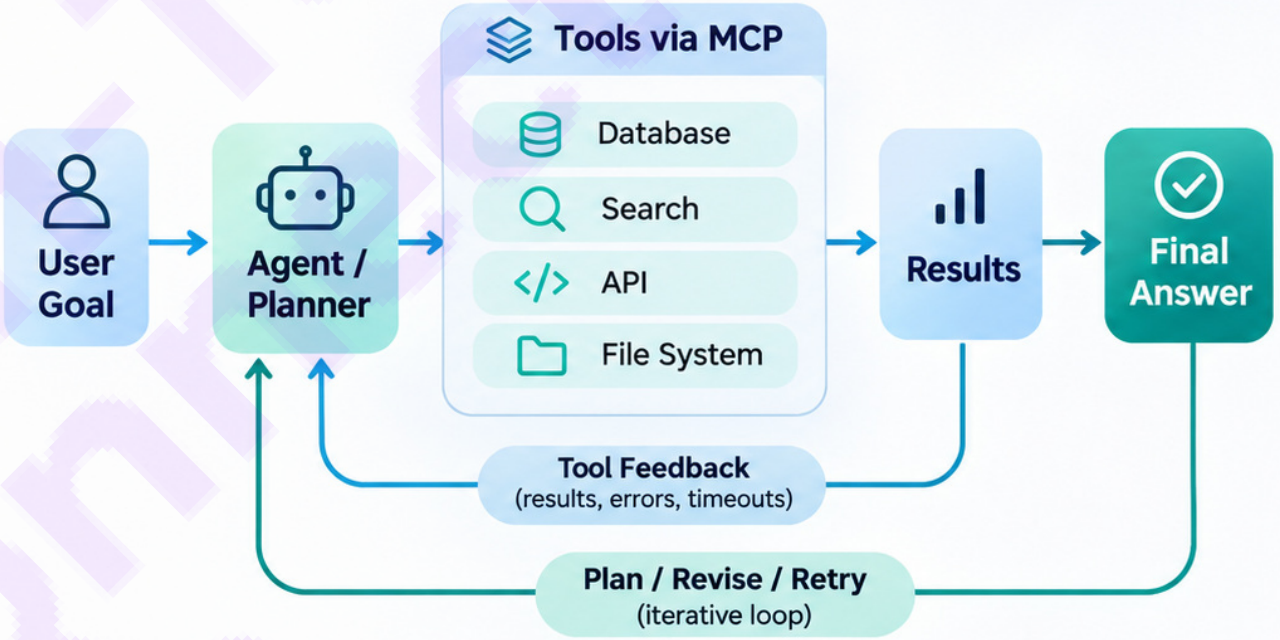
Answer: No. Include what the task needs, reserve output capacity and test quality, latency and cost with representative requests.

AI Agents and MCP

When the model can plan steps and use tools

Key Takeaways

- An **AI agent** is an application pattern where the model decides which steps to take to complete a task.
- Agents often call tools such as search, databases, APIs, ticketing systems, or scripts.
- **MCP (Model Context Protocol)** is a standard way to connect models to tools and structured context providers.
- This enables reusable integrations, but it also raises security, permission, timeout, and audit concerns.
- Operators should monitor tool calls, failure handling, and the boundaries between autonomous action and human approval.



The model is not only answering — it is orchestrating actions.

Slide 7

How to read this picture. The diagram combines two topics: an agent chooses steps, while MCP can connect the application to tools. Tool results may feed another decision. A final answer normally ends the run; the return arrow is not a requirement to continue forever.

From a fixed workflow to an agent

A workflow follows a designed sequence

A **workflow** is a set of steps coordinated by software. For example: receive an alert, retrieve its procedure, ask a model to summarise the evidence and return the result. The steps can include conditions and error handling, but the developer specifies the permitted paths.

An **AI agent** gives a model some responsibility for choosing the next step or tool within a controlled process. It might decide that the available evidence is insufficient, request another record and then reassess. This is the distinction between a predefined workflow and model-directed tool use described by Anthropic. [15]

A tool is an exposed software capability

A **tool** is a function the application makes available. Examples might be searching a document collection, reading an alert or drafting a ticket. Each tool has an interface: its name, expected input and returned result. The model can propose a call using that interface.

The model does not need direct credentials to every underlying system. Application code validates the requested operation and invokes the tool through the authorised connection. This separation matters: generating text that resembles a command is not the same as having permission to execute it.

Read the decision loop

The run begins with a user goal. The application supplies the goal and permitted tool descriptions to the model. The model may produce a proposed tool call. The application checks it, executes it when allowed and returns the result as new context. Another model call can then decide whether to continue or finish.

A tool can return useful data, no match, an error or a timeout. Each outcome needs handling. The result should not be invented when the tool fails. The diagram's feedback arrows express that iterative process, while the final-answer box represents a normal stopping point.

Use autonomy only where it earns its cost

If the required steps are already known and stable, a fixed workflow may be easier to test and operate. Agentic behaviour adds flexibility but also more possible execution paths. The label "agent" does not automatically make an application more capable, reliable or efficient. [15]

For our running example, the system might select which approved document search to perform. It need not be allowed to change cooling settings. A narrowly scoped assistant can be useful while retaining human responsibility for consequential actions. Define the task boundary before expanding the available tools.

CHECK YOUR UNDERSTANDING When does a tool call actually affect another system?

Answer: When the application authorises and executes it—not merely when the model generates a proposed call.

One user request can become many operations

A worked run for the operations assistant

Consider a fictional run. First, the model requests the current alert details. Second, after receiving them, it asks for the applicable procedure. Third, it compares the supplied facts and drafts a summary. Fourth, after a requested review step, it produces the final response.

That example uses four model calls and two tool calls for one user request. A different design might use fewer or more. Some tool calls can happen in parallel; others must wait for earlier results. Do not treat the illustrative sequence as a universal agent architecture.

Capacity follows internal work

If ten user requests arrive each second and each needs four model calls, the application creates an average of forty model calls per second before retries. This simple multiplication is an illustration, not a capacity estimate. Calls can differ greatly in input length, output length and computational cost.

Tool services also receive load. A document search bottleneck can delay the entire run even when the model service has spare capacity. Capacity tests should replay representative end-to-end work rather than count only the messages visible in the user interface.

Bound the run before it starts

Define a maximum number of steps, a time budget and limits on token use or spending. Provide a clear outcome when those limits are reached: return partial evidence, request clarification or hand over to a person. An unbounded “keep trying” loop is not an operational strategy.

Retries deserve particular care. Repeating a read request may be harmless; repeating an action can create duplicate tickets or changes. An **idempotent** operation is designed so repeating the same request does not create additional unintended effects. Use transaction identifiers or application checks where appropriate.

Separate proposing from approving

A proposed action can be shown to an authorised person before execution. The approval should identify the exact action and relevant parameters, not simply grant vague permission to continue. High-impact operations should remain under established operational controls.

Also record the actual external operations, errors and approvals. You do not need access to a model’s hidden internal reasoning to maintain a useful audit trail. The observable sequence—request, permitted call, result, approval and action—is what lets an operator investigate the run. A final answer should report unresolved issues rather than conceal failed steps.

CHECK YOUR UNDERSTANDING Why can ten user requests per second create much more than ten model calls per second?

Answer: Each request may involve several model turns, checks and retries. Measure the full workflow and its token demand.

What MCP standardises—and what it does not

MCP connects applications to capabilities

MCP stands for **Model Context Protocol**. It defines a common way for an AI application to discover and interact with capabilities exposed by another service. It is not a language model, an agent by itself or a replacement for the underlying business systems. [16]

The **host** is the AI application. An MCP **client** inside that application communicates with an MCP **server**. The server exposes capabilities such as tools or resources and can connect onward to an existing service. The model generally interacts through the application; it does not itself establish network connections.

The same business operation remains underneath

An example MCP server could expose a read-only “get_alert” tool for the monitoring system. The application can discover the tool’s expected fields, request a permitted call and receive structured results. A conventional monitoring API may still perform the actual lookup behind the MCP server.

MCP can reduce the need for a completely different integration pattern for every AI application. It does not eliminate the integration work inside each server. The server still needs accurate tool definitions, a reliable connection to the underlying system and sensible failure handling.

Security is not automatic

An MCP connection must be protected like any other interface with access to valuable data or actions. **Authentication** establishes who is calling. **Authorisation** determines what that caller may do. Permissions should be enforced at the server and underlying service, not inferred from the model’s request. [17]

Use narrowly scoped credentials and approved servers. Avoid treating arbitrary tool descriptions or returned text as trusted operating instructions. For the R17 assistant, a read-only monitoring integration should not accidentally become an administrative interface to every building system.

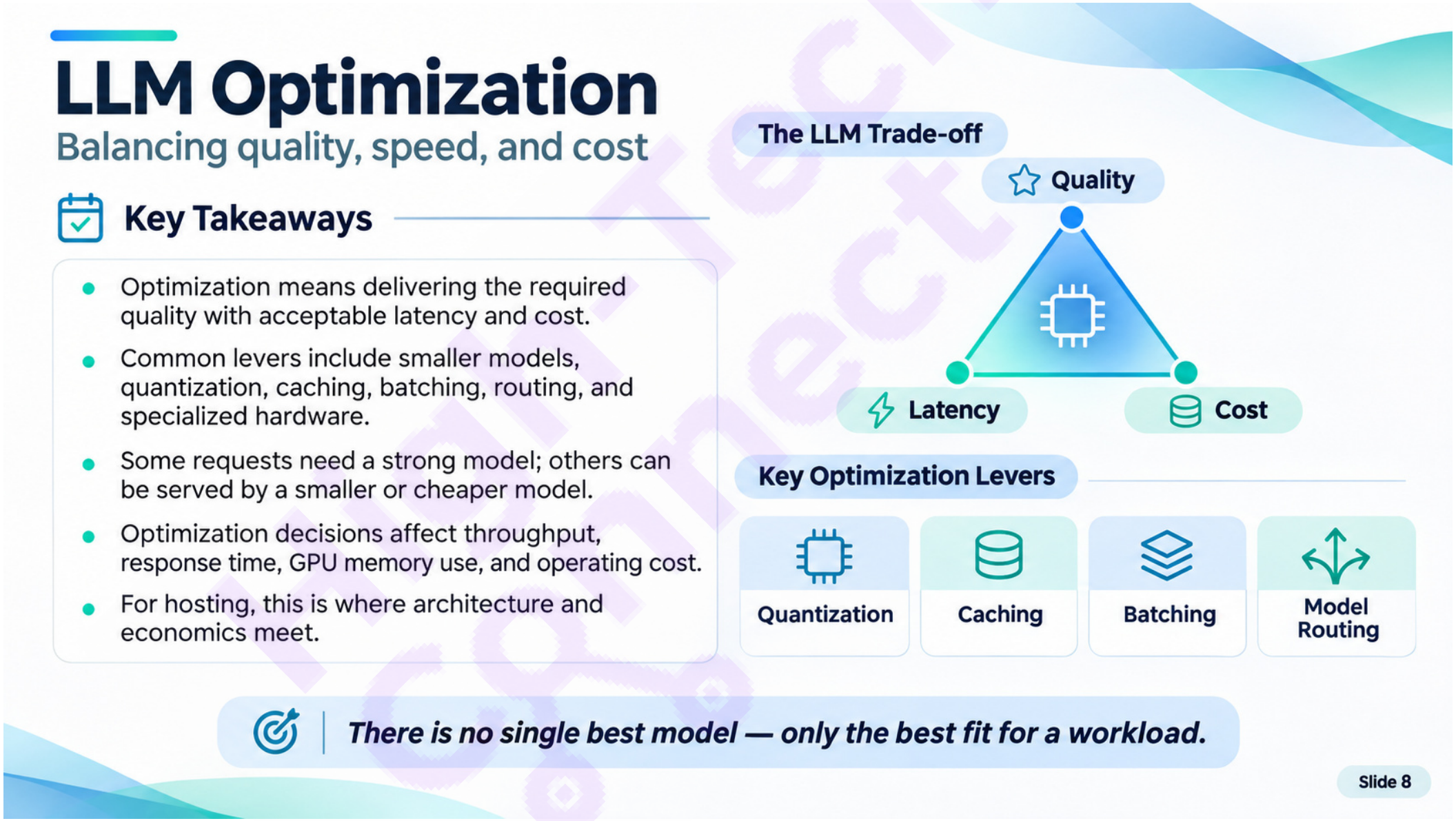
What the operator must own

Maintain an inventory of connected servers, their owners, permitted tools and data paths. Establish timeouts, availability expectations, logging and an update process. A server can be local to an application or remote; these choices create different deployment and network responsibilities. [16]

The key distinction is simple: the **agent** chooses among allowed steps, the **application** enforces the process, and **MCP** can provide the connection format. None of these removes accountability for the action. Start with a small, read-only capability and expand only when testing and operational controls justify it.

CHECK YOUR UNDERSTANDING Does using MCP mean an agent may safely run every tool a server advertises?

Answer: No. Tool availability is not permission; access, policy, validation and appropriate approvals still need enforcement.



How to read this picture. The triangle shows objectives that must be considered together. The four lower boxes name different optimisation levers. None is a universal improvement: test each change against the task and the complete service.

Define a useful result before optimising

Quality comes from the task

Optimisation means improving a system against chosen objectives. For the R17 assistant, a useful result is a concise, supported summary that arrives soon enough to help the operator. A rapid but incorrect answer is not an improvement. Neither is a perfect-looking answer that takes too long to be useful.

The triangle therefore puts quality, latency and cost together. **Latency** is elapsed time for an operation. **Throughput** is the amount of work completed in a period. They are related but not interchangeable: processing more work overall does not guarantee a shorter wait for each request.

First remove unnecessary work

The simplest savings may come from a shorter relevant context, a shorter requested answer or fewer model calls. Avoid asking the model to repeat information the application already has. Check whether a fixed software operation can perform a step more reliably than another generated response.

Then compare candidate models on the same task set. A smaller model may meet the required quality for classification or formatting, while a more demanding task may need a stronger model. These are hypotheses to test, not promises based on parameter count or marketing categories.

Quantisation changes numerical precision

Quantisation stores or processes some model values using fewer bits. A **bit** is a binary unit of information. Fewer bits per stored weight can reduce weight-memory requirements, but the actual effect on quality and speed depends on the method, model, runtime and hardware. Lower precision is not automatically faster on every system. [18]

For an illustrative model with eight billion parameters, storing every weight in 16 bits would require about 16 billion bytes, or 16 GB in decimal units, for weights alone. An idealised 4-bit representation would use about 4 GB. These calculations exclude quantisation metadata, working memory and the KV cache.

Measure the whole service after a change

Actual memory demand is greater than the simple weight calculation, and the service also needs room for concurrent work. A model that loads successfully is not necessarily one that serves the target workload with acceptable latency. [3]

Test representative input lengths, output lengths and demand patterns. Compare answer quality as well as resource use. A compression that saves memory but loses crucial distinctions in an operating procedure is not acceptable for that task. Keep the baseline and the evaluation results so the decision is reproducible.

CHECK YOUR UNDERSTANDING Does the 4 GB weight calculation prove that an eight-billion-parameter service fits on a 4 GB device?

Answer: No. It omits other memory requirements and implementation overhead; serving capacity must be measured.

Use the right lever for the bottleneck

Batching and caching solve different problems

Batching groups work so hardware can process it more efficiently. In continuous batching, a serving system can admit new requests as others finish rather than wait for a whole fixed group. This can improve utilisation, but scheduling and memory limits still affect individual waiting time. [19]

Caching avoids repeating eligible work. Reusing an identical input prefix can reduce repeated input processing; reusing a final answer is a different application-level choice. A cache needs appropriate validity and isolation rules. A cached answer about a changing alert should not be treated as current merely because it is fast. [14]

Routing chooses a service for a request

Model routing selects which model or endpoint should handle a task. A simple classification could go to one service and a difficult document comparison to another. Routing may also respond to availability or capacity, within approved data-location and security constraints.

The routing rule itself needs evaluation. A cheaper path that mishandles important cases may increase review work and total cost. A fallback can also change behaviour or send data to an unapproved location. Define what a fallback is allowed to change before an incident occurs.

Diagnose before buying more hardware

Slow performance could come from a long queue, excessive input, slow retrieval, model computation or a remote dependency. More GPU capacity helps only when it addresses the limiting part of the system. A trace across the request can reveal where time is actually spent.

Also distinguish memory *capacity* from memory *bandwidth*. Capacity is how much data can be held. Bandwidth is how quickly it can be moved. A model can fit in memory yet still be limited by data movement. Runtime and workload characteristics determine which resource matters most. [3]

Calculate cost per useful completed task

For a hosted model API, charges may depend on usage and service terms. For a self-hosted service, consider equipment, power, cooling, software, support and staff, as well as utilisation. Neither location is automatically cheaper for every demand pattern.

A practical comparison can divide an agreed total service cost by the number of tasks completed to the required quality. Include retries and failures in the accounting. This is a proposed decision method, not a benchmark from the slide. It links infrastructure efficiency to a useful business outcome instead of celebrating token output that nobody needed.

CHECK YOUR UNDERSTANDING Why can a cheaper model call produce a more expensive application?

Answer: Lower quality can create extra calls, retries, human review or failed tasks. Compare complete outcomes, not only the price of one call.

LLM Evaluation and Deployment

Prove it works — then run it safely

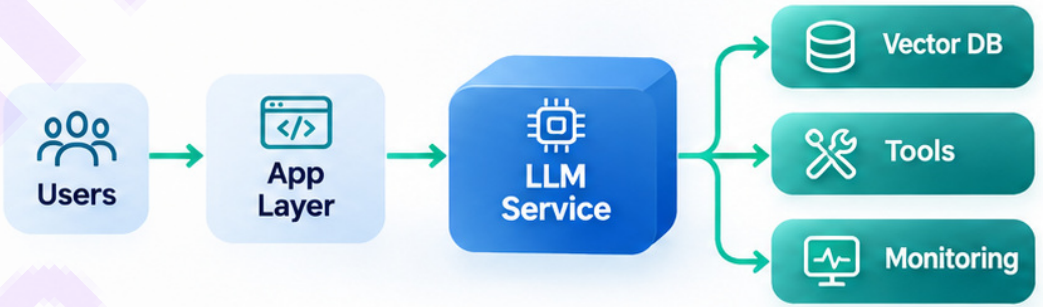
Key Takeaways

- Evaluation checks whether the application is accurate, useful, safe, and consistent on real tasks.
- Typical evaluation dimensions are answer quality, grounding, format compliance, latency, and failure rate.
- Deployment means packaging the model and its application services for reliable production use.
- Common deployment options include cloud APIs, managed platforms, self-hosted inference, and hybrid designs.
- Operators care about scaling, networking, secrets, rollback, HA, capacity planning, and environment separation.

Evaluation Scorecard (Example)

Dimension	Metric (Example)	Score
Accuracy	Task correctness (%)	92%
Grounding	Supported by sources (%)	95%
Latency	P95 response time (s)	1.3 s
Safety	Policy violation rate (%)	0.2%

Deployment Architecture



Evaluation prevents silent failure; deployment turns prototypes into services.

Slide 9

How to read this picture. The left-hand scorecard is illustrative, not evidence of measured performance. The right-hand drawing is a simplified dependency view; in practice, application or orchestration software normally coordinates retrieval and tools around the model service.

Evaluate the application on real tasks

Define success before looking at scores

Evaluation is systematic testing against stated criteria. For a language application, correctness may involve several dimensions: answering the question, supporting claims with evidence, respecting permissions, using the required format and responding within an acceptable time. A model leaderboard alone cannot establish all of these for your application. [20]

For the R17 assistant, success might mean accurately describing the supplied alert, identifying the applicable procedure and clearly stating missing information. It does not mean sounding confident. The operational team should help define the criteria because it understands which errors matter.

Build a representative test collection

A test case includes an input and a way to assess the expected result. Include normal cases and difficult ones: the wrong site, missing records, obsolete procedures, conflicting evidence and attempts to access another customer's material. Set aside cases that were not used to tune prompts or train the model.

Do not treat one successful demonstration as representative. Also check repeated runs where variation matters. A system that changes its recommendation unpredictably on the same important evidence may need further controls even if one answer looked excellent.

Use several kinds of checks

Software can verify required fields, permitted values or whether a cited document ID exists. A domain expert may be needed to judge whether a proposed conclusion follows from the evidence. Another model can help grade responses, but its scores should be checked against human judgement and known cases. [20]

For retrieval, test whether useful evidence was found. For generation, test whether the answer used that evidence correctly. For agents, test the selected actions and stopping behaviour. Keeping these checks separate makes a failing result easier to diagnose.

Compare versions under the same conditions

Run the old and new configurations against the same test set. Record the model, prompt, retrieval settings and other relevant versions. An improvement in one dimension may be accompanied by a regression in another, such as better wording but worse citation accuracy.

The purpose is a release decision, not a single attractive percentage. Document which errors remain, who accepts the residual risk and what happens when the application cannot meet the criteria. This is especially important for an assistant operating near business-critical processes, even when it has no direct control over equipment.

CHECK YOUR UNDERSTANDING Is a high score on a general model benchmark enough to approve the R17 assistant?

Answer: No. The complete application needs task-specific tests covering evidence, permissions, failure handling and operational performance.

Read the scorecard without being misled

Every percentage needs a definition

The picture shows 92% accuracy and 95% grounding as examples. Before using such values in a decision, ask what was measured, on which cases, by whom and using what rules. A percentage without its denominator and test conditions is incomplete information.

In a fictional collection of 200 cases, 184 passing cases would produce 92%. That arithmetic does not explain the severity of the sixteen failures or whether the cases resemble production. Rare but serious failures can be obscured by an impressive average. Break results down by task type and important risk categories.

Grounding and correctness are not identical

A grounded answer is supported by its supplied sources. The sources themselves can still be outdated or unsuitable. Conversely, a statement can happen to be true while not being supported by the cited evidence. A source-backed application should test both appropriate sourcing and correct interpretation.

The same caution applies to safety scores. “0.2% policy violations” is not meaningful without a definition of violation, a test design and an explanation of severity. The number in this picture is a fictional teaching value, not an accepted threshold or a promise about any product.

Understand p95 latency

The **95th percentile**, usually written p95, is a value at or below which approximately 95% of observations fall. A p95 response time of 1.3 seconds would still allow roughly 5% of measured requests to take longer. The picture’s 1.3 seconds is illustrative, not a target for this Academy.

Also define the measurement boundary. Time to first token, model generation time and the complete user-visible response time are different quantities. A model-only figure can exclude retrieval and application delays. Compare like with like, at a stated input size and concurrency level. [7]

Convert tests into release gates

A **release gate** is an explicit condition that must be met before a change reaches users. Example categories include acceptable task quality, no detected cross-tenant disclosure in the security tests, bounded latency under representative load and a verified rollback procedure. The exact acceptance levels belong to the service owner.

Passing a test suite does not prove that every future request will succeed. Tests sample behaviour. Production monitoring, incident handling and periodic re-evaluation are still required. The final sentence of the picture should therefore be read as an objective: evaluation helps reveal silent failures, rather than guaranteeing their elimination.

CHECK YOUR UNDERSTANDING Does p95 latency mean every request will complete within that time?

Answer: No. It describes a percentile of measured observations; slower requests remain possible.

Deploy the complete service, not just a model

Choose where responsibilities sit

Deployment makes the application available in an operational environment. With a hosted model API, another provider runs the model service, while your team still owns the application and its data paths. With self-hosted inference, your team also owns more of the runtime and capacity. A managed platform can divide responsibilities differently.

A **hybrid** design combines environments. That may be useful, but it introduces additional connections, security boundaries and failure modes. State where prompts, retrieved data, model calls and logs go. “Runs locally” is not sufficient evidence that every component and every record stays local.

Package a reproducible release

A release includes application code, model identifiers or artifacts, tokeniser, prompts, dependencies and configuration. A **container** packages software with its runtime dependencies. It helps make deployment repeatable but does not by itself guarantee compatible drivers, sufficient memory or secure operation.

Separate development, testing and production. Check access to required data and tools. Load the model and perform representative readiness tests before directing user traffic to it. A process that responds to a basic health check may still be unable to serve a real generation request.

Resilience involves more than copying a server

A **replica** is another instance of a service. Multiple replicas can provide capacity or resilience, but the design must account for shared dependencies. If all replicas depend on one unavailable search database, the user-facing service can still fail.

A model may be split across several accelerators because of its size or desired performance. That is different from running independent replicas, and it creates communication requirements. The serving design and hardware determine the appropriate arrangement; distributed inference is not automatically necessary for every application. [22]

Own the full production path

Define load limits, timeouts, queues, credential handling, backups and recovery procedures. Check power, cooling and network requirements against the actual equipment. Deploy a limited release, watch quality and service behaviour and make rollback practical.

The picture places the vector database and tools beside the LLM service. That is a simplified dependency view, not a statement that the model itself connects directly to everything. Application orchestration normally manages those interactions. An operator should be able to name the owner and failure response for each component in that real architecture.

CHECK YOUR UNDERSTANDING What is missing when “deployment complete” means only that the model process starts?

Answer: Validated application behaviour, dependencies, security, realistic capacity, recovery and ownership of the complete user-facing service.



How to read this picture. Read the dashboard as a set of questions, not as a real performance report. All its numbers and trend arrows are illustrative. The trace view is a simplified sketch; it does not establish a measured end-to-end timeline.

See the request beyond the server metrics

Observability explains what happened

Observability is the ability to understand a system's behaviour from information it emits. Traditional infrastructure signals remain necessary: CPU load, memory, network errors and service availability. AI applications also need visibility into model calls, retrieval, tool use, response quality and cost.

An HTTP success response only establishes that a request completed at that interface. It does not establish that the answer was correct or the requested task was successfully completed. Likewise, a quiet GPU does not prove the application is healthy; it may be waiting on a failed upstream dependency.

Logs, metrics and traces answer different questions

A **log** records an event, such as a tool error or a failed validation. A **metric** summarises a measured quantity over time, such as request count or latency. A **trace** follows work through connected operations, showing where time was spent and what depended on what. [23]

A **span** is one timed operation within a trace. For the R17 assistant, spans might cover permission checks, retrieval, one model call and a tool request. A shared trace identifier lets the operations team connect these events without guessing which entries belong to the same user request.

Make the model call identifiable

Record which model and relevant configuration handled the request, along with prompt or application version identifiers. Capture token usage and completion status where available. For multi-step tasks, also record actual tool calls and their outcomes. These details make comparison and incident analysis possible. [24]

The record should distinguish user-visible tasks from internal calls. One failed tool call might be handled successfully by the application; conversely, several technically successful model calls can produce an unusable outcome. Count both the component behaviour and the final task result.

Add quality signals, not only speed

Quality can be monitored through sampled review, automated checks and user feedback. An increase in unsupported citations may matter even when response times improve. A thumbs-up rate is useful context, but it is not a comprehensive accuracy measurement.

Treat evaluation and observability as complementary. Evaluation tests a controlled collection before and after changes. Observability shows what real traffic is doing. Findings from production can become new evaluation cases, so the same failure is less likely to return unnoticed. That feedback loop is part of operating the application, not an optional reporting feature.

CHECK YOUR UNDERSTANDING Can an application return HTTP 200 and still fail the user's task?

Answer: Yes. Technical completion does not establish answer quality, factual support or successful task completion.

Use the evidence to operate and improve

Investigate latency at the right boundary

Suppose users report a slow R17 summary. A trace might show that document retrieval consumed most of the elapsed time while model generation remained normal. That observation suggests investigating retrieval rather than immediately expanding the GPU pool. This is a fictional diagnostic example.

Also inspect queues, input and output lengths, time to first token and inter-token timing. Definitions differ between instruments, so record what each measurement includes. An aggregate average can hide a small group of badly affected requests. View distributions and meaningful request categories, not just one headline number. [7]

Connect unusual cost to actual work

If token use rises, check whether a new prompt is longer, a context-selection change includes more material or an agent has started making extra calls. If cost per completed task rises, investigate retries and failures as well as model charges.

The dashboard's $0.012, 3,210 tokens, percentages and trend arrows are not real observations. In an actual dashboard, each needs a time range, unit, aggregation rule and comparison baseline. Without those definitions, even a visually impressive dashboard can encourage the wrong decision.

Protect the monitoring data too

Prompts, retrieved passages, outputs and tool arguments can contain sensitive information. Do not log complete content indiscriminately. Use redaction, restricted access, retention limits and approved sampling. Keep credentials and secrets out of logs. Metadata-only traces can often support diagnosis without retaining every document. [24]

Audit requirements and debugging convenience are not the same thing. Decide deliberately which records are needed, why they are retained and who may read them. Protecting the production database while copying its contents into an unrestricted logging platform would defeat the original boundary.

Finish with an operational feedback loop

Define alerts that lead to clear actions: rising error rates, growing queues, missing source freshness or an unusual increase in tool retries. Give each alert an owner and a diagnostic path. A dashboard without an owner does not resolve an incident.

When a problem is understood, fix the appropriate layer, test the change and compare the result after release. The central lesson across all ten pictures is now complete: an AI service depends on instructions, evidence, controlled execution, resources and operational discipline. You do not need to design a model to ask whether those parts form a service you can trust and support.

CHECK YOUR UNDERSTANDING What should accompany every important dashboard number?

Answer: A clear definition, unit, measurement boundary, time range and an owner who knows what action to take.

The terms, without the mystery

Part 1 of 2 / Definitions recap the preceding chapters. Use the chapter links to revisit the full explanation.

Access token A credential used to access a service. It is not the same as an LLM text token.

Agent An application pattern in which a model chooses some next steps or tools within imposed limits.

API Application programming interface: a defined way for one software component to request work from another.

Authentication / authorisation Establishing who a caller is / deciding what that caller is permitted to do.

Base model The model used as the starting point for an adaptation or application.

Batching Grouping computation from requests to use processing resources more efficiently.

Checkpoint A saved model or training-state snapshot.

Chunk A section of source content prepared as a unit for retrieval or processing.

Container A package containing software and its runtime dependencies.

Context The instructions and information available to a model for a particular call.

Context window The model/service limit on the token sequence it can handle; configuration and output limits also matter.

CPU / GPU A general-purpose processor / a processor suited to large amounts of parallel numerical computation.

Decode The inference stage that generates output tokens.

Deployment Releasing the complete application and its dependencies into an operational environment.

Embedding A numerical vector representing content, often used to support similarity search.

Evaluation Testing behaviour against stated criteria using representative cases.

Fine-tuning Further training of an existing model to adapt its behaviour or capabilities.

Gateway A service boundary that can enforce access, usage limits and routing.

Grounding Support for generated claims in the supplied evidence; it does not prove the evidence is current.

Guardrail A control intended to constrain behaviour; important boundaries require software enforcement.

Hallucination Generated content that is false or unsupported, even when it is expressed fluently.

Idempotency Designing an operation so that repeating the same request avoids unintended additional effects.

Inference Using a trained model to process input and generate a result.

JSON A structured text format containing named fields and values; valid structure does not prove valid content.

KV cache Stored intermediate attention data reused during model inference.

The terms, without the mystery

Part 2 of 2 / Definitions recap the preceding chapters. Use the chapter links to revisit the full explanation.

Latency Elapsed time for a request or stage; always specify the start and end points.

LLM Large language model: a trained system that processes and generates language.

LoRA / adapter A smaller trainable adaptation used with a base model; LoRA leaves the original weights frozen during adaptation.

MCP Model Context Protocol: a common interface for connecting AI applications to exposed capabilities.

Metadata Information describing a record, such as its owner, version, timestamp or access rights.

Observability Understanding system behaviour using emitted information, including logs, metrics and traces.

Orchestration Application logic that coordinates services, model calls, tools and state.

p95 The 95th percentile: approximately 95% of observations lie at or below it.

Parameters / weights Learned numerical values that govern a model's computations.

Prefill The inference stage that processes the input sequence before output generation.

Prompt The instructions and other input used to guide a model's response.

Prompt injection Input that attempts to redirect a model away from its intended behaviour or boundaries.

Quantisation Representing some values with lower numerical precision, subject to quality and hardware trade-offs.

RAG Retrieval-augmented generation: retrieve external information and supply it to the model at request time.

Replica Another running instance of a service; it is different from splitting one model across devices.

Reranking A second selection stage that reorders retrieved candidates by estimated relevance.

Rollback Returning to a known working version and compatible configuration.

State Information maintained across steps or requests, such as task progress or conversation records.

Throughput The amount of completed work per unit of time; distinct from one request's latency.

Token A model-specific unit of input or output, such as a word fragment or punctuation.

Tool An exposed software capability the application can invoke after checking the proposed call.

Trace / span A record following a request across operations / one timed operation within that record.

Vector database A store and search system for vector representations and their associated information.

Workflow A process coordinated through predefined software paths, potentially including model calls.

You do not need to know everything to begin.

Use this final check to turn the reading into a useful conversation about your own environment.

Explain the basic distinctions

A **model** is one component of an application. **Inference** uses the model; **training** changes trainable parameters. A **prompt** specifies the task, **RAG** supplies retrieved information and **fine-tuning** adapts model behaviour.

An **agent** selects some steps within a controlled process. **MCP** can connect the application to tools. **Evaluation** tests whether the service meets its criteria; **observability** helps reveal what it does in production.

Describe one real use case

Choose a task from your organisation. Identify the intended user, the information required and what a useful result would look like. Avoid starting with a particular GPU or model name.

Write down what must never happen: an unauthorised disclosure, an unsupported recommendation, an unapproved action or an unacceptable delay. These constraints help define the architecture and the evaluation.

Map the operational responsibilities

Identify who would own the application, source documents, access controls, model service and production support. Establish which services could run locally, which could be external and which data would cross those boundaries.

Then ask for a representative workload test: realistic inputs, output lengths, simultaneous users and failure cases. Connect the result to compute, memory, network, storage, power and cooling requirements.

Bring an uncertainty, not just an opinion

A productive class question might be: “We have a changing collection of procedures and need answers with sources. How should we keep retrieval current?” Another might be: “Our requests may involve several tool calls. How should we measure capacity?”

You are ready when you can describe the desired service and identify what still needs to be tested. The purpose of the Academy is to build that judgement across the application and infrastructure layers.

Continue with AI Infrastructure Academy Europe
Programme and registration: hitechconnect.org/ai-infra-academy

Explore further, when you are ready.

Primary documentation consulted 12 November 2026. Titles are clickable; technical details and interfaces can change.

[01] **Microsoft Learn**
LLM Fundamentals
Models, tokens, context and inference.

[02] **Hugging Face**
What are LLMs?
Language models and token-by-token generation.

[03] **Hugging Face**
Optimizing LLMs for Speed and Memory
Memory, precision and inference trade-offs.

[04] **Hugging Face**
Tokenization algorithms
How text is split into model-specific tokens.

[05] **Hugging Face**
How do Transformers work?
Training, fine-tuning and model architecture.

[06] **Hugging Face**
Caching
The purpose of the inference KV cache.

[07] **vLLM**
Production Metrics
Serving metrics and measurement boundaries.

[08] **Anthropic**
Prompt engineering overview
Task specification, clear prompts and testing.

[09] **OWASP Gen AI Security Project**
LLM01:2025 Prompt Injection
Untrusted content and defence in depth.

[10] **Hugging Face PEFT**
LoRA
Frozen base weights and trainable adapters.

[11] **NVIDIA**
What is Retrieval-Augmented Generation?
External information and retrieval-based generation.

[12] **NVIDIA Technical Blog**
RAG 101: Demystifying RAG Pipelines
Offline ingestion and online retrieval/generation.

The ten illustrations are the supplied Academy visuals, not figures reproduced from these external sources. The detailed text, R17 scenario, self-checks and hypothetical calculations were written for this companion.

Explore further, when you are ready.

Primary documentation consulted 12 November 2026. Titles are clickable; technical details and interfaces can change.

[13] **Anthropic**
Effective context engineering for AI agents
Selecting and maintaining useful context.

[14] **vLLM**
Automatic Prefix Caching
Reusing eligible input-prefix computations.

[15] **Anthropic**
Building effective agents
Conceptual distinction between workflows and agents.

[16] **Model Context Protocol**
Architecture overview
Host, client, server and protocol scope.

[17] **Model Context Protocol**
Security Best Practices
Permissions, trust boundaries and integration risks.

[18] **Hugging Face**
Quantization: Overview
Lower-precision representation and trade-offs.

[19] **Hugging Face**
Continuous batching
Scheduling requests as earlier requests finish.

[20] **Anthropic**
Define success criteria and build evaluations
Task-specific criteria and testing methods.

[21] **Anthropic**
Demystifying evals for AI agents
Repeated trials, graders and real task outcomes.

[22] **vLLM**
Parallelism and Scaling
Serving designs, replicas and distributed models.

[23] **OpenTelemetry**
Signals
The roles of logs, metrics and traces.

[24] **OpenTelemetry**
GenAI Semantic Conventions
Current GenAI telemetry reference. Content sensitivity is also discussed in the linked attribute registry.

Source notes support the added technical explanations. The original visual sequence is retained; captions and text explicitly qualify simplified arrows, illustrative metrics and implementation-dependent choices. This is an introductory guide, not a production specification.