1 What Business Problems Do LLMs Solve?

Almost every business LLM application is a variation on a small set of task types. Learning to recognize which type you have is the first design skill.

Task type	**What it means**	**Business example**
Classification	Assign an input to one of a fixed set of labels	Route a support ticket to Billing / Technical / Sales; flag spam; sentiment of reviews
Extraction	Pull structured fields out of unstructured text	Invoice → vendor, date, total; resume → skills list; contract → renewal date
Summarization	Compress long content, preserving what matters	Meeting transcript → action items; 50-page report → executive brief
Generation	Produce new text from instructions	Marketing copy, email drafts, product descriptions, first-draft reports
Transformation	Rewrite, translate, reformat, normalize	Translate to Spanish; convert notes to a formal memo; normalize addresses
Question answering	Answer questions over a body of knowledge	Employee HR chatbot; "what does our refund policy say about X?"
Reasoning & analysis	Compare, diagnose, weigh tradeoffs	Analyze why churn rose; review a contract clause for risk
Code generation	Write, explain, or fix code, SQL, formulas	Natural-language question → SQL query against the data warehouse
Search & matching	Semantic similarity, ranking, deduplication	Find similar past cases; match job candidates to openings; detect duplicate records
Evaluation ("LLM as judge")	Grade or review other text against a rubric	Score support replies for tone and accuracy; check drafts against compliance rules
Planning & orchestration	Decide the next step toward a goal	Agent that books travel, files an expense report, or triages an incident

Table 1: A working taxonomy of LLM business tasks. Real systems usually combine several.

Practical observation

Many valuable applications are *pipelines*: classify → extract → look something up → generate. A support-ticket system might classify urgency, extract the order number, fetch the order record, and draft a reply. The rest of today's lecture is about how to build that kind of pipeline.

Simple example (classification + extraction in one prompt).

```
Prompt:
  You are a support triage assistant. Read the message below.
  Return JSON with fields: category (billing | technical | sales),
  urgency (low | medium | high), order_id (string or null).

  Message: "I was charged twice for order 88213 and need this
  fixed before my card statement closes Friday!"

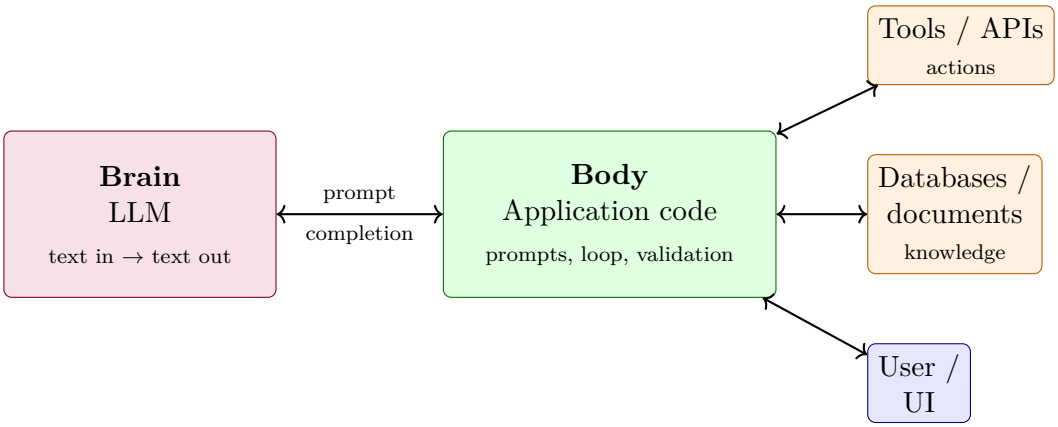
Model output:
  {"category": "billing", "urgency": "high", "order_id": "88213"}
```

When are LLMs a good fit? They shine when the input is messy natural language, the rules are too numerous or fuzzy to code by hand, and an occasional error is tolerable or can be caught by a human or a check. They are a poor fit when you need guaranteed exactness (arithmetic, ledgers), hard real-time latency, or where a simple rule or database query already solves the problem more cheaply.

2 The Brain and the Body

A useful mental picture: the language model is a *brain in a jar*. It can read text and produce text. That is all it does. By itself it cannot browse the web, check a database, send an email, remember yesterday's conversation, or even know today's date.

Everything else is the *body*: the surrounding software that gives the brain senses, memory, and hands.



Design principle

Let the brain do what only it can do (interpret language, make fuzzy judgments) and let ordinary software do everything else. A good LLM system is mostly good software engineering around a small, well-defined model call.

	Brain (the model)	**Body (your system)**
Does	Understands, reasons, writes	Stores, fetches, acts, enforces rules
Knows	Patterns from training data (frozen at a cutoff)	Live data: your customers, today's prices
Memory	None between calls	Databases, conversation logs
Side effects	None; it only emits text	Sends email, updates records, charges cards
Trustworthy for	Language, judgment calls	Exact facts, permissions, audit trails

3 Prompting and the Stateless Interaction Loop

3.1 The basic interaction

At the lowest level an LLM call is a function:

$$\text{completion} = \text{LLM}(\text{prompt})$$

The *prompt* is everything the model sees: instructions, examples, data, and the conversation so far. The *completion* is the text it generates in response. Prompting is the craft of writing the first argument well.

A typical prompt has several parts:

- **System instructions:** role, rules, tone, output format ("You are a support agent. Answer in two sentences.").
- **Context:** background material the model needs (a policy document, a customer record).
- **Examples** ("few-shot"): sample inputs with desired outputs.
- **The task or question:** what to do right now.

3.2 Context: all the model knows is what is in the window

The **context window** is the maximum amount of text (measured in *tokens*, roughly $\frac{3}{4}$ of a word each) that the model can consider in one call. Both the prompt and the generated completion count against it.

Key idea: context is everything

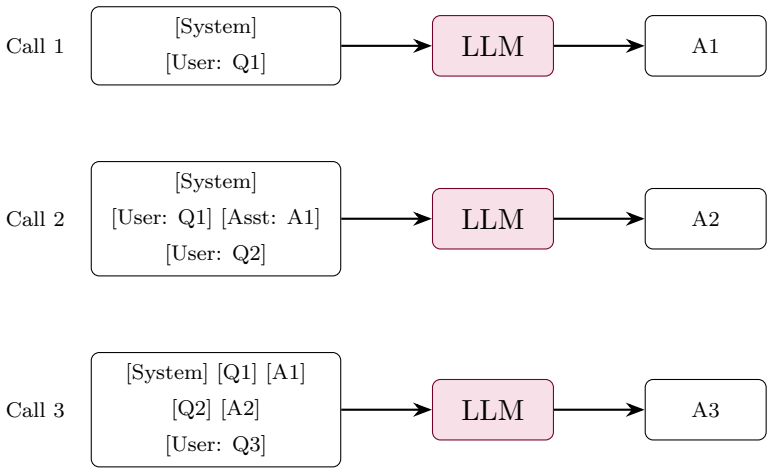
The model has no access to your business, your customer, or your last conversation *unless that information is placed in the prompt.* Weights hold general knowledge; the context window holds the specifics of this task. If it is not in the context, the model does not know it (and may confidently guess).

3.3 Statelessness

LLM APIs are **stateless**: each call is independent. The model does not remember previous calls. There is no hidden session on the model side.

Then how does a chatbot "remember" what you said two messages ago? Because the application *re-sends the entire conversation history* with every new request. The memory lives in the body, not the brain.

Analogy. Imagine consulting an expert with total amnesia. Every time you speak to them, you hand over a folder containing the complete transcript of everything said so far, plus your new question. They read the whole folder, write one reply, and forget everything again. The "conversation" exists only in the folder you keep.



Note that the prompt grows with every turn: the application appends the model's answer and the user's next message, then sends the whole thing again.

3.4 The loop in pseudo-code

```
history = [ {role: "system", content: "You are a helpful support agent."} ]

loop:
    user_text = read_user_input()
    history.append( {role: "user", content: user_text} )

    reply = LLM(history)                    # stateless call: send EVERYTHING
    history.append( {role: "assistant", content: reply} )

    show(reply)
```

The variable `history` is the entire "memory" of the conversation. If the program crashes, or you forget to append a message, the model has no idea it ever happened.

3.5 Consequences you must design for

- **Cost and latency grow with history.** You pay for input tokens on every call, so a long conversation re-bills the early messages again and again. (Providers offer *prompt caching* to reduce this for repeated prefixes.)
- **The window is finite.** Long conversations eventually must be *truncated* (drop oldest turns) or *summarized* (replace old turns with a short summary). Both lose information, so decide deliberately what to keep.
- **Memory is your job.** "Long-term memory" features are just the application storing facts in a database and inserting them into later prompts.
- **Position and clutter matter.** Models can overlook details buried in very long contexts. Irrelevant material is not free: it dilutes attention and costs money.
- **Reproducibility.** Because each call is a pure function of its prompt (plus sampling randomness), you can log prompts, replay them, and test changes. Treat prompts like code.
- **Privacy.** Anything placed in the context is sent to the model provider. Decide what data is allowed in a prompt.

Context engineering

Prompting at scale is less about clever phrasing and more about *deciding what goes into the context window on each call*: which instructions, which retrieved facts, which history, which tool results. This discipline is increasingly called *context engineering*. Everything in the rest of today's lecture (functions, RAG, MCP) is a mechanism for getting the right information into the context, or for letting the model act on the world.

4 Functions (Tool Calling)

4.1 The limitation

So far the model can only emit text. But a business needs it to *do* things and *look things up*: "What is the status of order 88213?" requires data the model has never seen. We could paste all possible data into the prompt, but that is impractical and often impossible.

4.2 The idea

We give the model a menu of functions it may *request*. Crucially:

Models do not execute functions

The model never runs code. It only *emits a structured request* ("please call `get_order_status` with `order_id="88213"`"). Your application decides whether to run it, runs it, and puts the result back into the context. The brain asks; the body acts.

4.3 Step 1: describe the function to the model

A function is described by a name, a plain-language description, and a schema for its parameters. The description is effectively a prompt: it tells the model when to use the tool.

```
tools = [
  {
    name: "get_order_status",
    description: "Look up the shipping status of a customer order.",
    parameters: {
      order_id: { type: "string", description: "The order number" }
    }
  }
]
```

4.4 Step 2: the application implements it

```
function get_order_status(order_id):
    row = database.query("SELECT status, eta FROM orders WHERE id = ?", order_id)
    return { status: row.status, eta: row.eta }
```

4.5 Step 3: the loop with tools

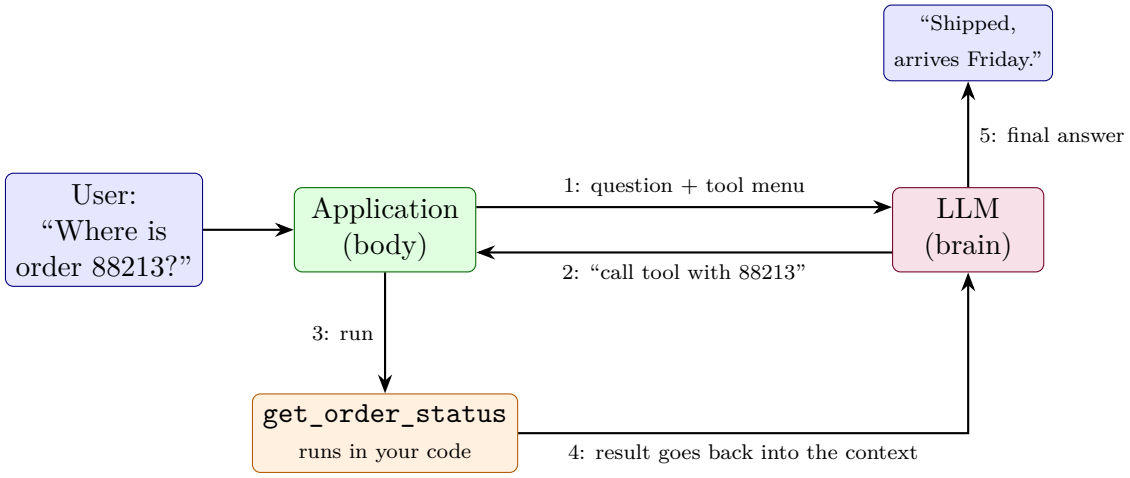
The interaction loop from Section 3 gains an inner loop. The model's reply is now either *final text* or *a request to call a tool*.

```
history.append( user_message )

while true:
    reply = LLM(history, tools)            # tools are sent with every call

    if reply.is_tool_request:
        history.append(reply)              # the request becomes part of context
        result = run_tool(reply.name, reply.arguments)   # YOUR code runs it
        history.append( {role: "tool", name: reply.name, content: result} )
        continue                           # call the model again with the result
```

```
    else:                                    # plain text: we are done
        history.append(reply)
        show(reply.text)
        break
```



4.6 A complete trace

```
[user]       Where is order 88213?
[assistant]  (tool request) get_order_status(order_id="88213")
[tool]       {"status": "shipped", "eta": "Friday"}
[assistant]  Your order 88213 has shipped and should arrive on Friday.
```

Notice that every line above is part of the *same growing context*. The tool result is not magically known to the model; it is just more text in the history, consumed on the next stateless call.

4.7 What functions give us

- **Fresh knowledge:** look up live data (inventory, prices, account status).
- **Actions:** create a ticket, send an email, issue a refund.
- **Exact computation:** call a calculator or run code instead of "mental" arithmetic.
- **Structured output:** even with no real action, a function schema is a reliable way to make the model return well-formed JSON (the extraction use case from Section 1).

4.8 Agents

When the loop above is allowed to run for many iterations, with the model choosing which tool to call next to reach a goal, we call the system an **agent**. Nothing new is added mechanically: it is the same stateless loop, repeated, with a growing context of tool results.

Engineering cautions

- **Validate arguments.** The model can produce wrong or malicious arguments. Treat them as untrusted user input.
- **Least privilege.** Expose read-only tools where possible; require human approval for irreversible actions (payments, deletions).
- **Cap the loop.** Set a maximum number of iterations and handle tool errors gracefully (return the error text so the model can adapt).
- **Prompt injection.** Text returned by tools (web pages, emails) may contain instructions that try to hijack the model. Data should never be treated as commands.

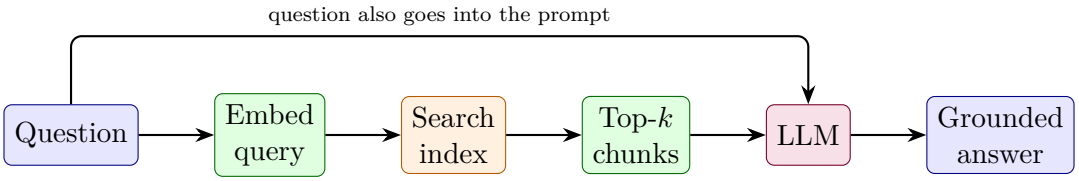
5 Retrieval-Augmented Generation (RAG)

5.1 The problem

Businesses have large private knowledge bases: policies, manuals, wikis, contracts, support history. The model has never seen them, and they will not fit in one context window. Fine-tuning a model on them is costly, slow to update, and does not give reliable citations.

5.2 The idea

RAG = *retrieve* the few most relevant pieces of text for the current question, *insert them into the prompt*, and let the model answer from them. It is the stateless context principle applied deliberately: we cannot show the model everything, so we show it the right small slice.



5.3 How it works

Offline (indexing):

1. Split documents into *chunks* (e.g. a few hundred words, often with some overlap).

2. Convert each chunk to an *embedding*: a vector of numbers where similar meanings land near each other.
3. Store the vectors (and the original text) in a vector index or database.

Online (each question):

1. Embed the user's question with the same embedding model.
2. Find the k chunks whose vectors are most similar, commonly by cosine similarity:

$$\text{sim}(\mathbf{q}, \mathbf{d}) = \frac{\mathbf{q} \cdot \mathbf{d}}{\|\mathbf{q}\| \, \|\mathbf{d}\|}$$

3. Build a prompt containing the instructions, the retrieved chunks, and the question.
4. The model answers using that material, ideally citing which chunk supported each claim.

```
function answer_with_rag(question):
    q_vec  = embed(question)
    chunks = vector_index.search(q_vec, top_k=4)

    prompt = "Answer ONLY from the sources below. If the answer is not
              there, say you don't know. Cite source ids.\n\n"
    for c in chunks:
        prompt += "[" + c.id + "] " + c.text + "\n"
    prompt += "\nQuestion: " + question

    return LLM(prompt)
```

5.4 Where RAG fits

RAG is not a competitor to functions; it is a *pattern* that can be implemented in two ways:

- **Fixed pipeline:** the application always retrieves before calling the model (as in the code above).
- **As a tool:** expose `search_documents(query)` as a function and let the model decide when and how often to search, possibly refining its query. This is often called *agentic RAG*.

5.5 Why teams choose RAG

- Knowledge stays current: update the index, not the model.
- Answers can cite sources, which supports auditing and trust.

- Access control can be enforced at retrieval time (only fetch what this user may see).
- Reduces (but does not eliminate) hallucination.

5.6 Common failure modes

- **Retrieval misses:** the right chunk is never fetched, so the model answers from nothing or guesses. Most RAG bugs are retrieval bugs, not model bugs.
- **Poor chunking:** an answer split across two chunks, or a chunk stripped of the context that gives it meaning.
- **Too much or irrelevant context:** stuffing in 50 chunks hurts quality and cost.
- **Stale or conflicting documents** in the index.

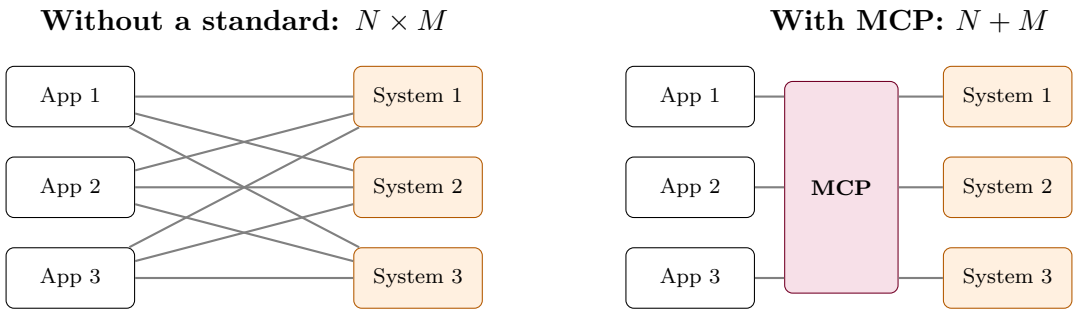
6 The Model Context Protocol (MCP)

6.1 The problem: integration sprawl

By now we have a pattern: connect the model to tools and data. But every application wires this up differently. Suppose we have N AI applications (a chat assistant, an IDE, an internal agent) and M systems to connect (Slack, GitHub, a CRM, a document store). Without a standard, each pairing needs its own custom integration:

$$\text{custom integrations} = N \times M.$$

Each one has its own tool-description format, authentication, error handling, and maintenance burden. When a system changes its API, every integration that touches it breaks.



6.2 The idea

MCP is an open protocol that standardizes how AI applications connect to external tools and data. Each system is wrapped *once* as an MCP server; any MCP-capable application

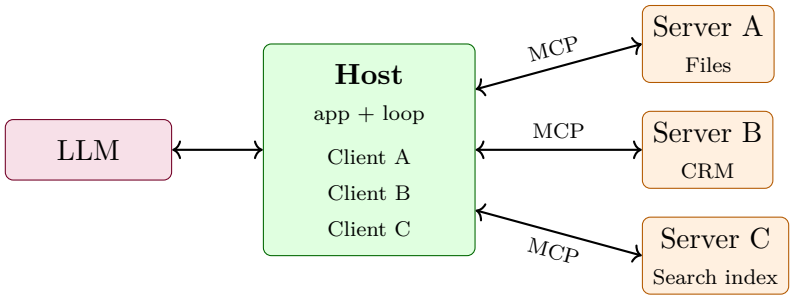
can then use it. The common analogy is **USB-C for AI applications**: one plug shape, many devices.

What problem MCP solves

It separates *what a capability is* (name, description, schema, how to invoke it) from *which application is using it*. Tool authors write one server. Application authors implement one client. Neither needs to know about the other in advance.

6.3 Architecture

- **Host:** the AI application the user interacts with (a desktop assistant, an IDE, your agent). It contains the LLM loop from Section 3.
- **Client:** a connector inside the host that maintains a one-to-one connection with a single server. A host may run many clients.
- **Server:** a program that exposes capabilities from some system (a database, a file system, an SaaS API).



6.4 What a server can offer

MCP defines three main kinds of capability ("primitives") a server exposes:

Primitive	Meaning	Controlled by	Example
Tools	Functions the model can ask to invoke (can have side effects)	The model, with host approval	`create_ticket`, `run_query`
Resources	Read-only data the host can load into context	The application	A file, a database row, a document
Prompts	Reusable prompt templates	The user	"Summarize this PR" slash-command

Tools are the part that corresponds directly to Section 4. The difference is that the tool descriptions now come from the server *at runtime*, rather than being hard-coded in your application.

6.5 How it works on the wire

MCP messages are JSON-RPC 2.0 requests and responses. A connection goes through three stages:

1. **Initialize.** Client and server exchange protocol versions and declare their capabilities.
2. **Discover.** The client asks what the server offers (`tools/list`, `resources/list`, `prompts/list`). The host converts the tool list into the tool menu it sends to the LLM.
3. **Use.** When the model requests a tool, the host's client sends `tools/call` to the server and returns the result into the context.

Messages travel over a *transport*. Two are standard: **stdio** (the host launches the server as a local subprocess and talks over standard input/output) and **streamable HTTP** (a remote server reached over the network, typically with authorization).

Example exchange (simplified).

```
# Client -> Server: what tools do you have?
{"jsonrpc": "2.0", "id": 1, "method": "tools/list"}

# Server -> Client
{"jsonrpc": "2.0", "id": 1, "result": {"tools": [
   {"name": "get_order_status",
    "description": "Look up the shipping status of a customer order.",
    "inputSchema": {"type": "object",
                    "properties": {"order_id": {"type": "string"}},
                    "required": ["order_id"]}}
]}}

# Later, after the LLM asks for the tool:
{"jsonrpc": "2.0", "id": 2, "method": "tools/call",
 "params": {"name": "get_order_status", "arguments": {"order_id": "88213"}}}

{"jsonrpc": "2.0", "id": 2, "result": {"content": [
   {"type": "text", "text": "{\"status\": \"shipped\", \"eta\": \"Friday\"}"}]}}
```

How the host ties it to the loop:

```
# at startup
clients = [connect(server) for server in configured_servers]
tools   = flatten( c.list_tools() for c in clients )

# the same loop as Section 4, with one change: run_tool goes through MCP
while true:
    reply = LLM(history, tools)
    if reply.is_tool_request:
```

```
        client = find_client_that_owns(reply.name)
        result = client.call_tool(reply.name, reply.arguments)   # MCP tools/call
        history.append( tool_result(result) )
        continue
    ...
```

6.6 MCP and RAG together

A retrieval system fits naturally behind MCP. Wrap your vector search as an MCP server that exposes a `search_docs` tool (or serves documents as resources). Any MCP-capable assistant can then do RAG over your knowledge base without custom glue code, and the same server can be reused by every application in the company.

6.7 Benefits and caveats

- **Reuse:** build a connector once, use it everywhere. A growing ecosystem of ready-made servers exists.
- **Dynamic:** capabilities are discovered at runtime; servers can be added without changing application code.
- **Separation of concerns:** the team that owns the CRM maintains the CRM server.
- **Security is still your job.** A protocol standardizes the plumbing; it does not make a tool safe. Only connect servers you trust, grant least privilege, authenticate remote servers, keep a human in the loop for consequential actions, and remember that tool descriptions and tool results are text in the model's context and can carry prompt injection.
- **Context cost.** Every connected tool's description occupies context tokens on every call. Dozens of servers can bloat the prompt, so expose only what each task needs.

7 Putting It All Together

7.1 One end-to-end example

Scenario: an HR assistant answers employee questions and can file requests.

1. The **host application** (body) keeps conversation history and loops (Section 3).
2. It connects via **MCP** to two servers: a `policy-search` server and an `hr-system` server (Section 6).
3. Employee asks: "How many vacation days do I have left, and what is the carryover policy?"

4. The model requests `get_leave_balance(employee_id)` (a **function**, Section 4). The body runs it via MCP and returns `{"remaining": 7}`.
5. The model requests `search_docs("vacation carryover policy")` (**RAG as a tool**, Section 5). The server returns the three most relevant policy chunks.
6. With all of that now in its **context** (Section 3), the model writes the answer, citing the policy section.
7. Next turn, the application re-sends the whole history, because the model is **stateless**.

7.2 Summary table

Concept	Problem it addresses	Core mechanism
Prompting	How to direct a model at all	Instructions, examples, and data placed in the input text
Context window	Model knows only what it is shown	Everything needed must be in the prompt
Stateless loop	Model has no memory between calls	Application re-sends the full history each time
Functions	Model cannot act or fetch live data	Model emits a structured request; app executes; result returns as text
RAG	Knowledge is too large or private for the prompt	Retrieve relevant chunks, insert into context
MCP	$N \times M$ custom integrations	Standard client/server protocol for tools, resources, prompts

7.3 Design checklist for a new LLM application

1. Which task type is this (classification, extraction, ...)? Could a simple rule do it?
2. What information must be in the context for the model to succeed? Where does it come from?
3. Does it need live data or actions? Define the tools, with least privilege.
4. Is the knowledge too big to include? Use retrieval; measure retrieval quality separately.
5. How will conversation state be stored, trimmed, or summarized?
6. What must be validated or approved by code or a human?
7. How will we evaluate quality (test set, LLM-as-judge, human review) and monitor cost and latency?
8. What data is leaving the organization in prompts?