AI Building Blocks: LLM, RAG, MCP, Skills, Agents

Version Nov 2026 · MCP spec 2026-07-28 · verified 2026-11-16

LLMs think, RAG grounds, MCP connects, skills remember, agents act — the five AI building blocks, explained for web developers.

Contents

Mental model 2

LLM — the brain 2

Context — the brain's desk 3

RAG — the brain plus a library 4

MCP — the nervous system 6

Skills — muscle memory 7

AI agent — the brain plus hands 8

Agent risks — permissions and goals 10

Designing all five together 11

Further reading 12

Mental model

An LLM is a brilliant function with no I/O: text in, text out, and everything it knows is either baked in at training time or sent in the request. The other four parts exist to fix that. RAG feeds it the right facts, MCP gives it a standard socket to your systems, skills hand it a reusable playbook, and an agent loop lets it act on what it decides. LLMs think, RAG grounds, MCP connects, skills remember, agents act — and the system works only when all five are designed together.

LLM — the brain

It can	It cannot
Read, write, summarise, translate	See data newer than its training
Write and review code	Read your DB, inbox or calendar
Reason through multi-step problems	Remember your previous request
Ask to call a tool you defined	Run that tool itself

Think of the model as a senior engineer on day one: sharp, widely read, and with zero access to your company. It has long-term memory (the training data, frozen at a cutoff date) and working memory (the **context window**: everything in the current request, including its own reply). Like a stateless HTTP endpoint, it keeps nothing between calls — a chat "remembers" only because your app re-sends the whole conversation every turn.

```ts
type Msg = { role: "user" | "assistant";
  text: string };
declare function llm(m: Msg[]): Promise<string>;


const history: Msg[] = [];
async function chat(text: string) {
  history.push({ role: "user", text });
  // the full history goes out on every call
  const reply = await llm(history);
  history.push({ role: "assistant",
    text: reply });
  return reply;
}
```

> **Gotcha:** Asked about something it cannot know — this week's meetings, yesterday's price — a model often answers anyway, fluently and wrongly. Missing context does not produce an error, it produces a plausible guess.

Context — the brain's desk

Goes into context	Cost
System prompt and instructions	Sent on every call
Conversation history	Grows each turn
Retrieved documents (RAG)	Pay per token

Goes into context	Cost
Tool definitions and results	Grows with each tool step

The context window is the model's desk: only what is on it right now can influence the answer. It is like RAM next to the training data's disk — fast and directly usable, but limited, and you pay for every token per request. A bigger desk is not automatically better: as it fills, recall and accuracy drop (Anthropic calls this *context rot*). Every building block below is, at heart, a strategy for putting the right things on the desk at the right moment.

> **Tip:** Curate instead of stuffing. Five relevant paragraphs beat fifty loosely related pages, both in answer quality and in cost.

RAG — the brain plus a library

Step	What happens
Index	Split docs into chunks, embed, store
Retrieve	Search for chunks matching the question
Augment	Paste the best chunks into the prompt
Generate	Model answers from those chunks

Retrieval-augmented generation turns a closed-book exam into an open-book one: before answering, your code looks things up in your docs, tickets or live data and puts the findings in the prompt. It is the same pattern as a server-side `loader` that fetches data before a page renders —

the template (the model) stays generic, the data makes it yours. Instruct the model to answer only from the supplied sources and to cite them, so an answer can be checked.

```
type Chunk = { id: string; text: string };
declare function search(
  q: string, k: number): Promise<Chunk[]>;
declare function llm(
  p: string): Promise<string>;

async function ask(q: string) {
  const docs = await search(q, 5);
  const ctx = docs
    .map((d) => `[${d.id}] ${d.text}`)
    .join("\n");
  return llm("Answer only from the sources, " +
    "cite [id], say if unknown.\n" +
    `${ctx}\nQ: ${q}`);
}
```

> **Gotcha:** RAG is only as good as its search. If retrieval returns the wrong chunks, the model answers confidently from the wrong chunks. Measure retrieval on its own before tuning prompts — see RAG retrieval.

MCP — the nervous system

MCP role	Web analogy
Host (Claude Code, an IDE, a chat app)	The browser
Client (one per server, in the host)	A `fetch` connection
Server (wraps GitHub, a DB, files)	An API with a standard contract
Tools / resources / prompts	Actions / readable data / templates

First, tool calling: you describe a function with a JSON schema, the model replies with a structured request to call it, and **your code** runs it and returns the result. Without a standard, every model-tool pair needs custom glue — N apps times M tools. The Model Context Protocol is one standard plug: build a server once and every MCP host can discover and use its tools at runtime. It deliberately borrows the idea of the Language Server Protocol, which let one language server work in every editor. Messages are JSON-RPC 2.0.

```
{
  "jsonrpc": "2.0",
  "id": 7,
  "method": "tools/call",
  "params": {
    "name": "list_events",
    "arguments": { "week": "2026-W47" }
```

```
  }
}
```

> **Gotcha:** MCP standardises the plug, not the safety. A server can expose a tool that deletes data as easily as one that reads it, and the spec leaves consent to the host — treat installing a server like installing a dependency with production credentials.

Skills — muscle memory

Part	Loaded
`name` + `description`	At startup, ~100 tokens per skill
`SKILL.md` body	When the task matches the description
`scripts/`, `references/`, `assets/`	Only when a step needs them

A skill is a folder with a `SKILL.md` file: the steps, the output format and the checks for one recurring task, written once and reused on every run. It is the agent's version of a team runbook — you stop re-explaining "how we write release notes" in every prompt. Loading is lazy, like a dynamic `import()`: only the short description sits in context, so you can install dozens of skills and pay for the body only when one is used. It is an open standard (Agent Skills), and Claude Code reads skills from `.claude/skills/<name>/SKILL.md`.

```
---
name: release-notes
description: Write release notes from merged
  PRs. Use when the user asks for a changelog.
---
1. List PRs merged since the last tag.
2. Group them: Features, Fixes, Breaking.
3. One line each, user-facing wording.
4. Check every Breaking item has a fix path.
```

> **Tip:** The `description` is the trigger. State what the skill does *and* when to use it, with the words a user would actually type; a vague "helps with docs" never activates.

AI agent — the brain plus hands

Phase	Example
Plan	"Find free slots, then book, then email"
Act	Call `calendar.list`, `calendar.book`
Observe	Read the tool result, or the error
Repeat or stop	Next step, or report the outcome

An agent is an LLM running in a loop with tools: it picks the next step, your code executes it, the result goes back into context, and the loop repeats until the job is done. Anthropic draws the line this way: in a **workflow** your code fixes the path; in an **agent** the model directs its own

process and tool use. It is the difference between a script that runs fixed steps and a colleague you hand a goal to — more flexible, and much harder to predict.

```
type Step = { tool?: string; args?: unknown;
  answer?: string };
declare function think(
  c: string[]): Promise<Step>;
declare function run(s: Step): Promise<string>;

async function agent(goal: string, max = 10) {
  const ctx = [goal];
  for (let i = 0; i < max; i++) {
    const step = await think(ctx);
    if (step.answer) return step.answer;
    ctx.push(await run(step)); // observe
  }
  throw new Error("step limit reached");
}
```

> **Note:** Loop design, tool design and evals go deeper than this sheet — see AI agents: patterns and evals.

Agent risks — permissions and goals

Risk	Fix
Tools reach systems they should not	Least privilege: fewest, narrowest tools
Irreversible action taken alone	Human approval for writes, sends, payments
Vague goal, wrong job done fast	Explicit goal, scope and “done” criteria
Runaway loop and spend	Step and budget caps
Instructions hidden in tool output	Treat retrieved text as data, not orders

Hands make the brain dangerous. Give an agent credentials the way you scope an OAuth token or an IAM role: only what this task needs, read before write, and approval before anything that cannot be undone. “Clean up the CRM” is a vague goal an agent will pursue literally and quickly; “merge duplicate contacts created this month, list changes, change nothing else” is a spec. In Claude Code, permission rules are checked deny first, then ask, then allow.

```
{
  "permissions": {
    "allow": ["Bash(npm run *)"],
    "ask": ["Bash(git commit *)"],
    "deny": [
```

```
      "Bash(git push *)",
      "Read(./.env)"
    ]
  }
}
```

> **Warning:** A deny pattern matches text, not intent: `Bash(git push *)` does not catch `git -C . push`. For anything that must never happen, remove the credential or sandbox the agent — don't rely on a rule the model can phrase its way around.

Designing all five together

Missing part	Symptom
RAG	Fluent answers that ignore your data
MCP	A pile of one-off integrations to maintain
Skills	Same task, different result every run
Agent loop	A chatbot that tells you what to click
Guardrails	A demo nobody lets near production

Most teams build one part — usually a chatbot over an LLM — and call it an agent. A useful company assistant needs the whole stack: the model reasons, retrieval grounds each answer in your data, MCP gives it one standard way into your systems, skills encode how your team does the job, and the agent loop carries the work through under tight

permissions. Trace a real request through every layer before you build, and you will see which one is missing.

```
"Reschedule my 1:1s clashing with the offsite"
LLM    understands the request, plans steps
RAG    finds the offsite dates in the wiki
MCP    calendar server: list and move events
Skill  "rescheduling" playbook: notify,
       keep the slot length, avoid Fridays
Agent  runs the steps; asks before sending
```

Further reading

- Claude Docs — Context windows
- Claude Docs — Tool use overview
- MCP specification 2026-07-28
- Agent Skills specification
- Anthropic — Building effective agents
- Claude Code — Permissions
- Deeper sheets: AI and ML for web developers, RAG retrieval, MCP, AI agents, OWASP for LLM and agentic apps