READ MODE · ANSWER FIRST

# LangChain for LLM Application Development

An interactive, visual explanation of LangChain for LLM Application Development, from setup through the complete system.

CHEAT SHEET · 6 ESSENTIAL IDEAS

## The whole story in 6 lines

LangChain for LLM Application Development becomes easier to reason about when every stage is connected as one system.

1. A useful app does not stop at raw text generation. It shapes the prompt before the call and shapes the answer after the call - like...
2. Memory is not magic. It is a policy for deciding what past conversation stays visible when there is only so much room in the prompt.
3. A chain is like a factory assembly line for your app. Each station does one job and passes the result to the next.
4. The key move here is retrieval. Instead of the model guessing from its general training, you feed it the specific evidence it needs.
5. Evaluation is like grading your app's homework. You give it known questions, check the answers, and track whether changes you make...
6. An agent is powerful because it can adapt on the fly - but risky for the same reason.

STAGE 1 OF 8

# Setup


01 SETUP

Welcome. Before we start building, let us cover the big picture. LangChain is a toolkit that helps you turn a raw AI model into a real application - one with structure, memory, and the ability to use outside tools. Think of it like going from a bare engine to a fully assembled car. This explainer follows the same big topics covered in the DeepLearning.AI course.

A prompt template is a reusable fill-in-the-blank frame for instructions. Instead of writing a brand new prompt from scratch every time, you create a pattern once and just swap in the current question and context. Like a form letter where the structure stays the same but the details change.

A parser is the cleanup crew. When the model sends back its answer, the parser turns that loose text into a shape your app can actually trust - like plain text, JSON, or a neat list of fields. Without a parser, you are hoping the model formats things right every time.

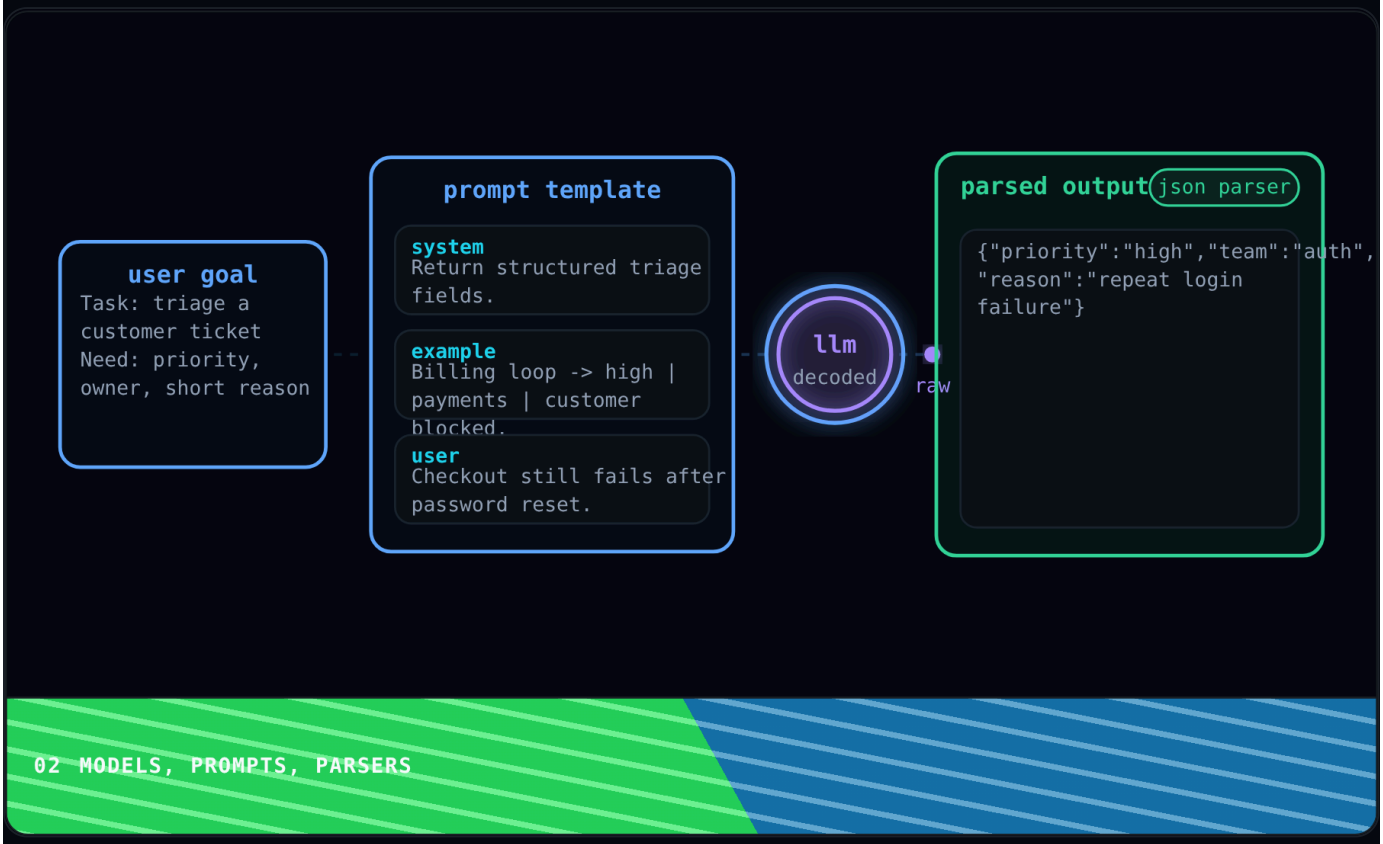
Memory is how the app keeps track of what already happened in the conversation. Without it, every turn feels like the first turn - the model has no idea what you talked about thirty seconds ago. Memory might keep the last few messages or compress older chat into a short summary.

A chain is like a small assembly line in a factory. One step does its job and passes the result to the next step. Instead of cramming everything into one giant prompt and hoping for the best, you break the work into manageable pieces that hand off to each other.

Over the next six stages we will build up the complete picture: wrapping a model call with prompts and parsers, adding memory, connecting steps into chains, answering questions from documents, testing whether it all works, and finally letting the model choose its own tools as an agent. Let us start with the most basic LangChain move: one clean model call.

STAGE 2 OF 8

# Models, Prompts, Parsers




02 MODELS, PROMPTS, PARSERS

Now that we share the vocabulary, let us see it in action. When a user types something, LangChain does not just pass it straight to the model. It wraps that raw input with structure - the prompt template we just learned about - so the model knows exactly what role to play and what kind of answer to give.

The template fills in the current question, instructions, and sometimes examples. Try switching the Template control in the sidebar between Direct, Few-shot, and Chat to see how much structure each one carries. The more precise the setup, the more useful the answer.

That finished prompt is sent to the model. One nice thing about LangChain: you can swap models in and out from this one spot without rewriting the rest of your workflow. The template and parser stay the same even if the brain behind them changes.

The raw answer comes back rich in language but loose in shape. That is fine for chatting, but an application usually needs something stricter. Imagine your app needs to fill a database row or update a dashboard - it cannot work with a paragraph of prose.

This is where the parser earns its keep. Remember from the Setup, the parser is the cleanup crew? Use the Parser control in the sidebar to switch between Text, JSON, and Fields. Watch how each format shapes the output differently. The parser turns that loose answer into the exact shape the rest of your app expects.

That is the first LangChain building block: template in, model call, parser out. A clean sandwich. But right now each call is completely independent - the model has no idea what you asked it five seconds ago. Next we will fix that by adding memory, so the app can actually remember what happened earlier in the conversation.

STAGE 3 OF 8

# Memory


03 MEMORY

We just saw how one model call works: template, model, parser - a clean sandwich. But real chat apps do not get the luxury of isolated calls. Every answer changes what the next question means. If someone says "what about the blue one?" your app needs to know what "the blue one" refers to.

As more turns pile up, the prompt gets crowded. There is only so much room, and every new message takes up space. The app has to choose: keep the whole transcript, only the most recent turns, or a shorter summary of the older stuff?

Switch the Memory control in the sidebar to Buffer. Buffer memory is the simplest idea - just keep everything. It is easy to understand and nothing gets lost. But it is like never throwing away any notes from class. Eventually your desk is buried. Now try Window and Summary to see how the other strategies trim the history.

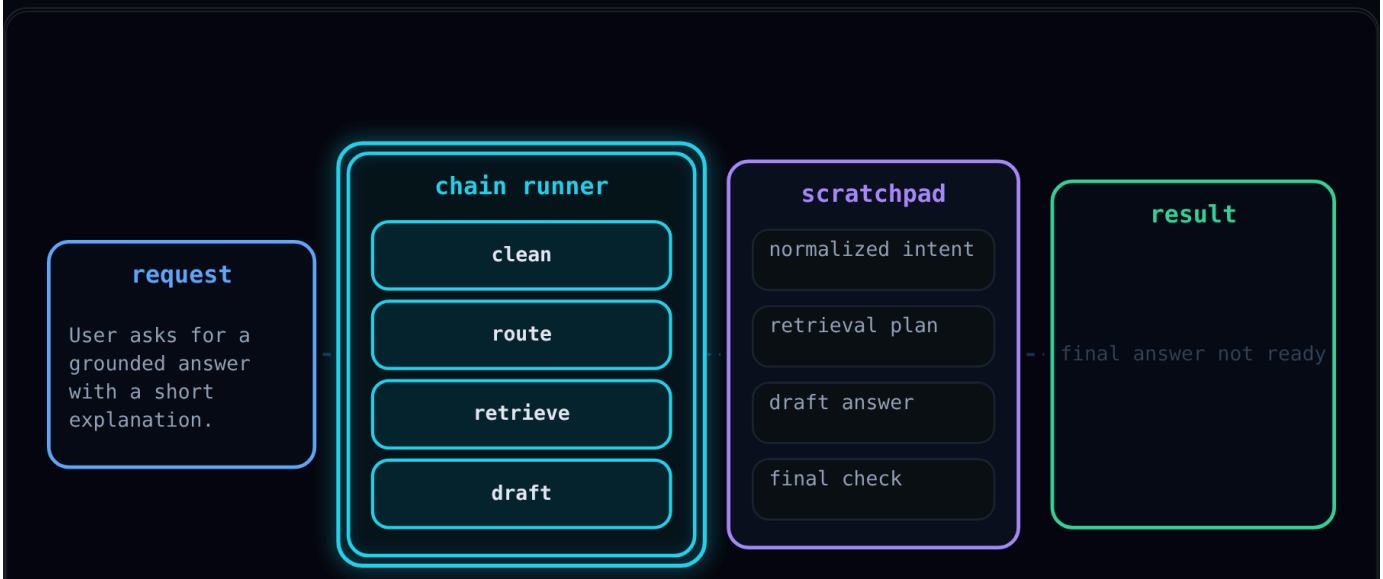
Window memory is more practical. It keeps only the last few turns and lets the older ones fade away. Think of it like your phone's recent messages - you can see the latest conversation, but messages from last week have scrolled out of view.

Summary memory is the cleverest option. It compresses the older conversation into a short note - like writing "we discussed project timelines and agreed on March" instead of keeping twenty messages of back-and-forth. You lose some detail, but you keep the big picture and save room for what comes next.

So memory is really about context management - it decides what the next prompt gets to "remember." Remember that prompt template from Stage 1? Memory is what fills in the conversation history part of that template. Now that our app can remember, let us connect several steps together so model calls can work as a team instead of acting alone.

STAGE 4 OF 8

# Chains




04 CHAINS

Memory gave us the ability to carry context forward. Now we need motion - the ability to do several things in sequence. Remember how we described a chain in the Setup? A small assembly line where one step produces output and the next step picks it up. Let us see that in action.

The first step might clean the question, classify what kind of request it is, or fill in missing details. That output becomes the input to the next step. The handoff is explicit - you can see exactly what was passed along, unlike stuffing everything into one enormous prompt.

As the chain moves forward, it keeps a scratchpad of intermediate results. This is incredibly useful for debugging. If the final answer is wrong, you can look at the scratchpad and see exactly where things went off track - was it step two? Step four? You do not have to guess.

Not every chain is a straight line. Try the Pattern control in the sidebar: switch from Linear to Branch to see how a chain routes different questions to different specialists, or try Parallel to see tasks running side by side before joining results. LangChain models all these shapes without losing track of the data.

By the time the final step runs, the app has much more than the raw user text it started with. It has structured context built up by each earlier step - cleaned data, classifications, intermediate answers. That accumulated context is why chains usually produce steadier, more reliable outputs.

This is the workflow style that made LangChain popular: compose small, focused steps instead of writing one giant do-everything prompt. And here is where it gets really interesting - next we will use that same composition style to answer questions from your own documents, not just from what the model already knows.

STAGE 5 OF 8

# Document QA


05 DOCUMENT QA

Remember how chains let us route information between steps? Document question answering uses that same assembly-line idea, but with an important new ingredient: retrieval from outside documents. The model is no longer relying only on what it learned during training.

When a user asks a question, the retriever searches the document store for the most relevant passages. Think of it like a librarian who hears your question and pulls the three most relevant books off the shelf. This is how your private data enters the workflow.

Only the top passages make it through, because prompt space is limited - remember, there is only so much room in the model's context window, just like we talked about with memory. That retrieval filter is why good document QA feels grounded and specific instead of vague and generic.

LangChain then combines those retrieved passages with the original question. Use the Combine control in the sidebar to compare the strategies: Stuff packs everything into one prompt, Map processes each passage separately, and Refine improves the answer step by step with each passage it reads.

The model answers with the retrieved evidence right there in front of it. If citations are enabled, the answer also points back to exactly which sources it used - so you can check for yourself where the information came from. No more blind trust.

This is one of the course's most important ideas: connect your model to your own data so it answers with evidence, not just confidence. But here is the uncomfortable question - how do you actually know these answers are good? Next we will build an evaluation loop to find out.

STAGE 6 OF 8

# Evaluation


06 EVALUATION

We just built a document QA workflow that retrieves evidence and generates grounded answers. It looks good. But here is the uncomfortable truth: AI models can sound incredibly confident even when they are completely wrong. How do we know our app is actually working well and not just sounding convincing?

The answer is evaluation - and it starts with a test set. You create a collection of questions where you already know what a good answer looks like. This makes your LLM app testable in the same way unit tests make ordinary software testable. No more guessing.

The app runs those test cases through the same prompts, chains, and retrieval logic it uses for real users. Remember all those building blocks we assembled - the templates, the memory, the chains, the retriever? Every single one of them gets exercised. Nothing is hidden from the test loop.

A judge then scores the answers. Switch the Judge control in the sidebar to see the different approaches: Criteria checks against explicit rules like "is the answer factually correct?", Labeled compares to known reference answers, and Pairwise puts two versions of your app side by side to see which is better.

Those scores roll up into a scoreboard so regressions become visible. Try toggling the Drift control in the sidebar to inject a weak answer and watch the scoreboard react. This is how you catch sneaky problems - like when tweaking your prompt template helped accuracy but accidentally broke the parser output format we set up back in Stage 1.

Evaluation closes the loop between design and evidence. Without it, LangChain work is just guesswork with extra steps. And you will really want this discipline for what comes next - agents. When the model starts choosing its own tools instead of following a fixed chain, knowing whether it is making good choices becomes even more critical.

STAGE 7 OF 8

# Agents


Remember how evaluation taught us to watch system behavior carefully and measure whether changes help or hurt? That discipline becomes absolutely critical now. With agents, the workflow is no longer a fixed chain you designed in advance - the model is choosing its own path in real time.

The planner reads the goal and decides what to do first. Sometimes it can answer directly from what it knows. Other times it needs a tool to gather more information. This is fundamentally different from chains, where you decided the steps ahead of time - here the model is making those decisions on the fly.

When the planner calls a tool, the outside world replies with an observation - search results, database rows, API responses. That observation gets fed back into the planner for the next round. This is why agents feel like a loop rather than the straight assembly line we saw with chains.

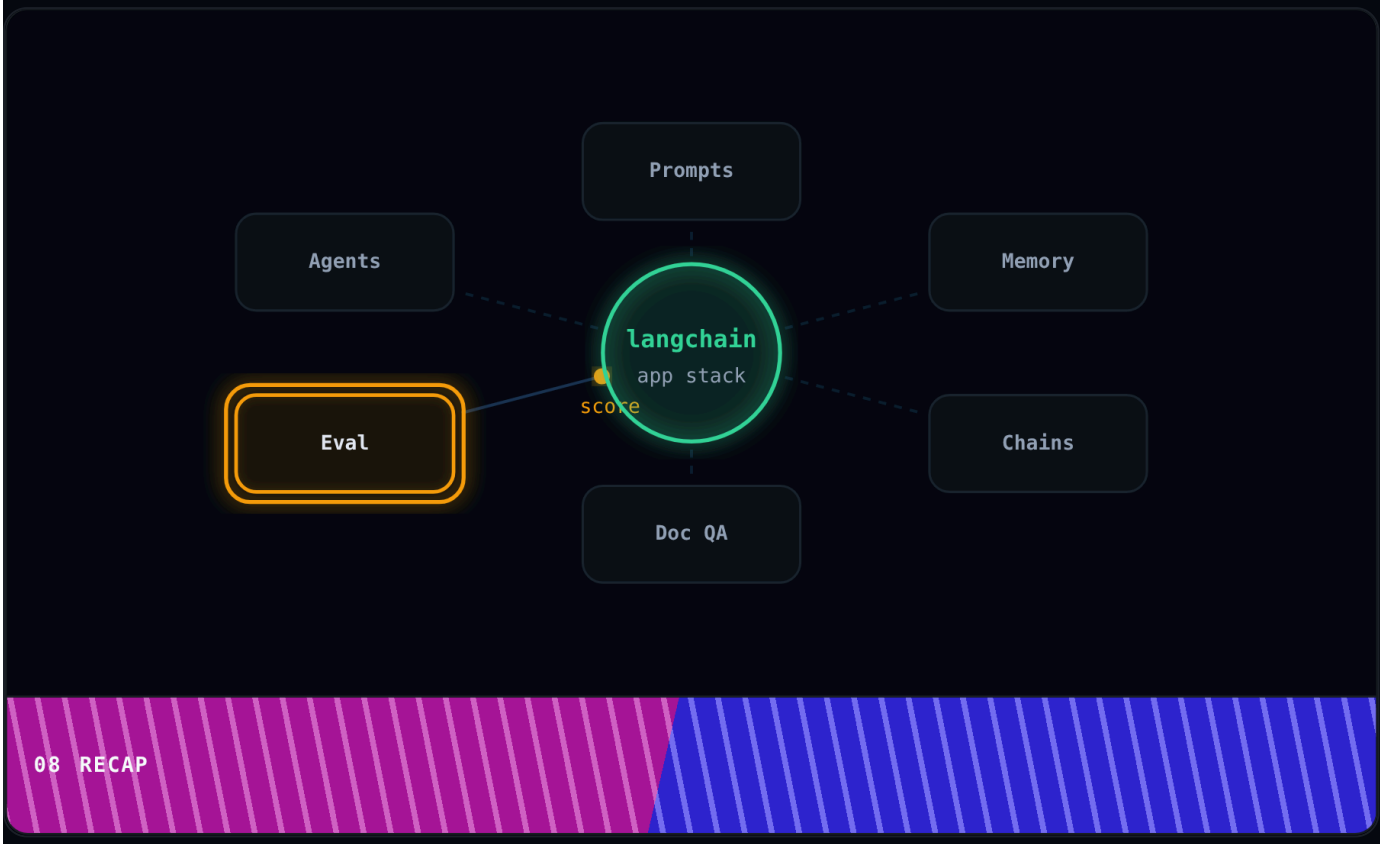
The planner keeps a scratchpad of what it has learned - similar to the intermediate state we saw in chains, but now the model itself decides what to write down and what to do next. If the first tool result was incomplete, it can try a different approach instead of giving up.

Use the Outcome control in the sidebar to see how the loop resolves differently. Answer shows a clean finish, Recover shows the agent hitting a dead end and trying a different tool, and Stall shows it going in circles. When it stalls, the fix is usually better tool descriptions or tighter prompts. This is where that evaluation loop from the last stage really pays off.

That completes the LangChain course arc: we went from structured model calls, to memory, to chains, to document retrieval, to evaluation, and finally to agents that choose their own path. Now let us zoom out and see how all these pieces connect into one picture.

STAGE 8 OF 8

# Recap




08 RECAP

We started with prompts, models, and parsers - the clean sandwich of template in, model call, parser out. That gave every model call a dependable input shape and a dependable output shape. Without that foundation, nothing else we built would have been reliable.

Then we added memory, so the app could stop acting like every turn was the first turn. Whether it was a full buffer, a sliding window, or a compressed summary, memory gave the app the ability to carry conversation forward - filling in the history section of that same prompt template.

Next came chains - the assembly line that connected multiple steps into one workflow. Instead of cramming everything into a single giant prompt, we broke work into focused steps that passed results to each other, with a scratchpad we could inspect when something went wrong.

After that we connected the app to real documents. Retrieval brought private evidence into the prompt - your PDFs, your notes, your reports - so the model could answer from your actual data instead of guessing from its general training.

We also added evaluation, because "it looks right" is not good enough. Test cases, judges, and scoreboards turned our improvements into something measurable, so we could tell whether changes actually helped or just seemed to.

And finally, agents handed the steering wheel to the model itself. The planner could choose tools, read observations, and loop until it had enough evidence to answer. Powerful, but only safe because of the evaluation discipline we built just before.

Put together, the picture is straightforward: LangChain helps you build LLM applications one layer at a time. That is why the course moves from simple model calls to stateful, grounded, and tool-using systems. Each layer depends on the ones below it, and together they form a complete, practical toolkit.