

# Creating Basic LLM Applications

---

Now that we've explored the architecture behind large language models (LLMs), let's move forward and build a practical LLM-powered application. Modern LLMs differ significantly from traditional AI systems in several fundamental ways:

- **Generalization capabilities:** LLMs can adapt to new tasks with minimal or no retraining, leveraging knowledge acquired during pre-training.
- **Streaming mode:** Unlike batch-processing models, LLMs can generate responses in real-time, token by token, enabling interactive user experiences.
- **Context windows:** LLMs process input within a finite context length (but much larger vs traditional AI), which determines how much prior conversation or document content they can consider when generating a response.

In this chapter, we'll focus on building an LLM application for **TaskFriend**, the AI assistant introduced in the previous chapter. We'll implement core features such as multi-round conversations and system prompt engineering to shape AI behavior.

## The story so far...

You've got the basics of LLM architectures down, and you're ready to build features for your app. Your app already has a task manager, and you want to help users take their task management to the next level. You decide to build a chatbot into your app to help users analyze, prioritize, and break large tasks down into smaller, more manageable tasks.

## Goals

- Explore streaming responses for a more natural conversational experience
- Implement multi-turn conversations to maintain context across user interactions
- Understand how system prompts define and control AI behavior

## Intitializing the environment

### Setting up the API key

Before we start work on in any notebook, we'll need to load the [API key for Model Studio](#). This ensures that we can call APIs of Qwen models we'll be using throughout this course.

If you're unsure about how to find your **Model Studio** API key, refer to the [00 Setting Up the Environment](#) file.

```
# Load Model Studio API key
import os
from config.load_key import load_key
load_key(
    confirmation=False          # ← Change to "True" if you want to change
```

```
your API key
)
```

## Setting up the LLM client

We'll use the openai-compatible interface provided by DashScope to interact with Qwen models (or other models we use throughout the course).

```
import os
from openai import OpenAI
import logging

logging.getLogger().setLevel(logging.ERROR)

client = OpenAI(
    api_key=os.getenv("DASHSCOPE_API_KEY"),
    base_url="https://dashscope-intl.aliyuncs.com/compatible-mode/v1",
)
```

## Setting up streaming mode responses

We've already set up the `get_qwen_stream_response` function in the previous chapter, so we're just going to reuse it here.

```
# Streaming mode response
def get_qwen_stream_response(query, system_prompt, temperature, top_p):
    response = client.chat.completions.create(
        model="qwen-plus",
        messages=[
            {"role": "system", "content": system_prompt},
            {"role": "user", "content": query}
        ],
        temperature=temperature,
        top_p=top_p,
        stream=True
    )

    for chunk in response:
        try:
            content = chunk.choices[0].delta.content
            if content: # Skip empty chunks
                yield content
        except Exception as e:
            print(f"Error parsing chunk: {e}")
            yield f"Error parsing response: {e}"
```

```
# User query
query = "Tell me about yourself"

# System prompt
system_prompt = "You are TaskFriend, a helpful AI assistant that helps
users manage daily tasks, prioritize work, and optimize time."
```

## Multi-turn Conversations: How Do They Work?

Traditional AI systems often handle one-off queries, such as answering "What is the weather today?" But real-world productivity challenges—like planning a busy week or breaking down a major project—require multiple clarifying questions, adjustments, and feedback loops. Multi-turn conversations enable LLMs to support these complex, evolving interactions by maintaining context and guiding users through structured workflows, just as a human assistant would.

For **TaskFriend**, this means helping users go from vague goals like "I have too much on my plate" to concrete, prioritized action plans through a series of intelligent exchanges.

To build robust multi-turn applications, it is essential to understand both the underlying mechanisms and the practical strategies that mitigate these limitations.

### Context Windows

Compared with traditional AI systems that would always reset after a turn, modern LLMs understand and remember our conversation across multiple turns? The secret lies in **context windows**.

The context window defines how much prior conversation and information the model can "remember" in a single prompt. For example, when a user says, "I need to prepare for a presentation next week," and later asks, "Can I fit in the gym too?", the model must recall the earlier mention of the presentation to assess scheduling feasibility.

Qwen models support large context windows ([Qwen3](#) supports up to 128K tokens), allowing it to retain extensive dialogue history, task lists, calendar constraints, and wellness preferences. However, as conversations grow, developers must manage token usage to avoid truncation of critical context.

The following diagram is a simplified representation of how context accumulates across turns in our app, **TaskFriend**:

```
graph LR
    S["System Prompt: 'You are TaskFriend...'"] --> C["Context Window"]
    U1["User: 'I need to finish a report due Friday'"] --> C
    A1["Assistant: 'How many hours do you have free this week?'"] --> C
    U2["User: 'About 10 hours, but I want to exercise daily'"] --> C
    A2["Assistant: 'Let's block 2-hour focus sessions and 30-minute workouts'"] --> C
    C --> M["Model generates next response"]
```

Each message adds to the context, enabling **TaskFriend** to remember past conversations and pick off where the user left off.

## Building the multi-turn conversation function

We've developed a simple chat interface for **TaskFriend**, and now we're going to enhance the `get_qwen_stream_response` function we learned in the previous chapter. The good news? It's actually simpler than you think.

All we need to do is **maintain context** across turns — not just send a single query, but remember the conversation so far. This is what transforms a one-off Q&A into a **real assistant** that can help you plan, reflect, and grow.

Here's how we do it:

1. **Take the current conversation** — not just the latest message, but the full history of what's been said.
2. **Stream the response** token by token, so it feels alive and responsive.
3. **Accumulate the full response** so we can save it back into the conversation for future turns.

Let's look at the upgraded function:

```
# Streaming mode response
# def get_qwen_stream_response(query, system_prompt, temperature, top_p):

# Renaming the function for better visibility
def get_qwen_stream_response_accumulate(query, system_prompt, temperature,
top_p):
    response = client.chat.completions.create(
        model="qwen-plus",
        messages=[
            {"role": "system", "content": system_prompt},
            {"role": "user", "content": query}
        ],
        temperature=temperature,
        top_p=top_p,
        stream=True
    )

    full_response = ""
    for chunk in response:
        try:
            content = chunk.choices[0].delta.content
            if content:
                full_response += content
                yield content
        except Exception as e:
            yield f"[ERROR] {e}"
```

At first glance, this looks very similar to our original function — and that's by design. The key addition is `full_response`, which accumulates each piece of the AI's reply as it streams in.

Why do we need this?

Because while `yield content` sends each word to the screen instantly (giving that satisfying "typing" effect), we also need the complete response to add back into the conversation history. Without it, our app, **TaskFriend** wouldn't remember what it just said — and the next turn would lose all context.

💡 Think of it like a whiteboard:

- `yield content` is the marker drawing one word at a time.
- `full_response` is the full sentence you save to refer back to later.

For ease of use and testing, we've developed a chabot interface for **TaskFriend**, which lets us plug-in our `get_qwen_stream_response_accumulate` function (or any function that you customize) into it via the `call_llm_fn`. Let's try it out:

```
from taskfriend import chat, config
from taskfriend.chat import wrap_streaming_for_chat, wrap_rag_for_chat

# Wrap function for compatibility
wrapped_llm = wrap_streaming_for_chat(
    get_qwen_stream_response_accumulate
)

chat.chat_interface(
    full_conversation=[],
    client=client,
    call_llm_fn=wrapped_llm,
)
```

Now, every time you type a message, **TaskFriend** sees the full context, streams its reply, and stores it for the next round. You can ask follow-ups like:

"What should I do to balance these tasks?" And it will remember your earlier conversations about your tasks and its conflicts — because it's all in the conversation history.

But here's the catch: **context isn't infinite**.

As the chat grows longer, we hit the limits of the model's **context window** — the maximum number of tokens it can process at once. If we don't manage this, older messages get silently dropped, and **TaskFriend starts forgetting**.

The context window demystified

So what actually happens behind the scenes? **Context is finite**, and what happens if the **conversation exceeds the context window**? Let's plug some new parameters into our **TaskFriend** app and visualize

what's going on behind the scenes. We're going to start with a context window of 400 tokens, just to visualize what happens when we exceed the context window:

```
chat.chat_interface(
    full_conversation=[],
    client=client,
    call_llm_fn=wrapped_llm,
    use_context_window=True,
    context_window=400,           # Does not take effect when
    use_context_window=False
    show_truncated=True,
    show_context_preview=True,
)
```

When you start chatting, you'll notice something new: a preview of the actual context being sent to the model.

```
=====
ACTUAL CONTEXT SENT TO LLM
=====
[ 0] System    ( 50t) → You are TaskFriend, a helpful AI...
[ 1] User      ( 80t) → I need to finish a report by Friday...
[ 2] Assistant (120t) → That sounds important! How many hours...
[ 3] User      ( 90t) → I also have to prep for a team meeting...

🇮🇹 Total: 340/400 tokens used
=====
```

This is exactly what the LLM sees — no more, no less.

Now, keep typing. Add a few more messages. Eventually, you'll see this:

```
🗑️ TRUNCATED MESSAGES (not sent to model):
-----
User: I wanted to start learning Spanish this month...
Assistant: That's a great goal! Maybe 15 minutes a day...
-----
```

### 💥 Your messages are being dropped.

Even though they're still in your `full_conversation` list, they're too far back to fit in the 400-token window. The system had to make a choice: **keep the recent exchange or the old one** — and it chose recency.

At first, this might not seem like a big deal. But imagine **TaskFriend** forgot you already decided to block 7–8 AM for deep work, or that you hate meetings on Fridays. That's not helpful — that's **harmful**.

So how do we fix it?

- We can't make the context window bigger forever (even 128K runs out eventually).
- We can't keep all messages — **we need to remember the meaning**.

## Summarization: The Multi-purpose Solution

We now know that **long conversations hit a wall** — the context window. And when they do, messages get dropped.

But what if we could **remember the meaning** of those messages — not the exact words, but the **key decisions, goals, and constraints** — and feed that back into the conversation? That's where **summarization** comes in.

For example:

"User plans to finish a report by Friday, block 2-hour focus sessions, and avoid Friday meetings. Also wants to start learning Spanish with 15-min daily practice."

That's just ~30 tokens — but it preserves the key decisions and intentions from hundreds of tokens of conversation.

This is exactly what summarization does: **it turns volume into value**.

Let's upgrade **TaskFriend** with a summarizer function to create **memory that lasts**.

```
def my_summarizer(text, client):
    print("💡 Summarizing dropped conversation...", flush=True)

    response = client.chat.completions.create(
        model="qwen-plus",
        messages=[
            {"role": "system", "content": """
                Summarize the key points from this conversation.
                Focus on tasks mentioned, decisions made, goals, and action
                items.

                Be concise — max 4 sentences.
                """},
            {"role": "user", "content": text}
        ],
        temperature=0.5,
        max_tokens=100,  # Limit the length of the output, try
        # to aim for 1/4 or less of total context
        top_p=0.9,
        stream=False
    )
    summary = response.choices[0].message.content.strip()
    print(f"📝 Summary: {summary}", flush=True)
    return summary
```

```
# Launch chat with the summarizer we built
chat.chat_interface(
    full_conversation=[],
    client=client,
    call_llm_fn=wrapped_llm,
    use_context_window=True,
    context_window=400,          # Does not take effect when
use_context_window=False
    show_truncated=True,
    show_context_preview=True,
    summarize_dropped=True,
    summarizer_fn=my_summarizer  # ← Plug in your summarizer!
)
```

Now, when old messages are about to be dropped, TaskFriend doesn't just delete them — it summarizes them first.

You'll see output like:

```
💡 Summarizing dropped conversation...
📝 Summary: User needs to finish a report by Friday and prep for a team
meeting. Agreed to block 2-hour focus sessions and avoid meetings on
Fridays. Also wants to start learning Spanish with 15-min daily practice.
```

And that summary gets injected into the context right after the system message.

Now, even though the original messages are gone, TaskFriend still knows:

- The report is due Friday
- You prefer not to have meetings on Fridays
- You're trying to learn Spanish

It's like giving your AI a **notebook**, like a real-life assistant — it doesn't remember every word, but it remembers what mattered.

💡 **Pro Tip:** You can even summarize multiple times — like a rolling memory log — to handle very long conversations.

This technique is used in real-world AI assistants — and now, you've built it yourself.

## Other benefits of summarization

Summarization isn't just about saving tokens — it unlocks several advanced capabilities that make LLM applications more robust, scalable, and user-friendly:

- **Improved coherence over long sessions:** By preserving high-level intent and decisions, summarization helps maintain conversational continuity, even as individual messages are dropped.
- **Enhanced memory efficiency:** Instead of storing raw dialogue history, you can store compact summaries — ideal for saving to databases or syncing across devices.

- **Support for reflection and planning:** Summaries can be reused to generate weekly reviews, progress reports, or goal-tracking dashboards — turning chat logs into actionable insights.
- **Better user experience:** Users feel heard and remembered. Even if they return after days, the AI can "recall" their goals from summaries, creating a sense of continuity.
- **Foundation for hierarchical memory systems:** Summaries can be summarized again, forming a "memory tree" or "rolling log" — a technique used in advanced agents like AutoGPT and BabyAGI.
- **Reduced hallucination risk:** When the model has access to distilled truths (e.g., "User wants to finish report by Friday"), it's less likely to invent conflicting plans or forget constraints.

In short, summarization transforms your LLM app from a reactive chatbot into a **proactive, memory-aware assistant** — a crucial step toward building truly intelligent agents.

## What you've built so far

You've now created an AI assistant that:

- Streams responses in real time
- Maintains conversation history
- Summarizes and remembers key decisions
- Operates within realistic token limits

This is no longer just a chatbot — it's a **cognitive partner** with memory, focus, and purpose.

And the best part? You did it **without retraining the model**. All of this behavior comes from **smart prompting, context management, and modular design** — the hallmarks of modern LLM applications.

In the next section, we'll explore how to shape *TaskFriend's personality and behavior\** — not through code, but through the system prompt.

## Shaping AI Behavior: System Prompts

---

So far, we've focused on *how* to build a multi-turn chat app. Now it's time to answer: **who is TaskFriend, really?**

The answer lies in the **system prompt** — the invisible rulebook that defines the AI's personality, tone, knowledge, and boundaries.

Unlike traditional software, where behavior is hardcoded, LLMs are shaped **primarily through language**. And the most powerful line of code in any LLM app is often the first one:

```
{"role": "system", "content": "You are TaskFriend, a helpful AI assistant..."}
```

## TaskBuckaroo: The cowboy experience

Let's see how system prompts affect our application. Let's do a fun spin on **TaskFriend**, and give them that wild-west vibe as **TaskBuckaroo**, the friendly cowboy, productivity outlaw.

```
# Input our cowboy-themed system prompt
system_prompt = ( """
    Howdy, partner! Yeehaw! You're TaskBuckaroo, a Wild West productivity
    outlaw.
    Talk like a cowboy. Use phrases like 'reckon', 'y'all', and 'howdy'.
    Be helpful, but make it sound like you're sippin' coffee by the
    campfire.
    """
)

chat.chat_interface(
    full_conversation=[],
    client=client,
    call_llm_fn=wrapped_llm,
    system_prompt_override=system_prompt
)
```

It's highly entertaining! But is it *helpful*?

Now imagine this in a real app. A user asks for career advice. The AI responds in pirate talk. Or Shakespearean English. Or refuses to answer because it's "in character."

This is **prompt drift** — when the AI's personality overrides its purpose.

- 💥 The AI isn't broken.
- 💥 The model isn't faulty.
- ❌ The system prompt failed to set boundaries.

## The problem with "be creative" prompts

Loose prompts like:

- "Be fun and engaging!"
- "Talk like a superhero!"
- "Use emojis and slang!"

... lead to unpredictable behavior because they:

- Prioritize style over substance
- Lack guardrails for inappropriate contexts
- Allow the AI to forget its core task

And once the AI starts roleplaying, it's hard to pull it back.

## The fix: A professional, structured system prompt

Let's redesign the prompt to be clear, consistent, and controllable.

```
# Input our professional, structured system prompt
system_prompt = ( """
    You are TaskFriend, a professional AI assistant for productivity and
    work-life balance.

    # Rules:
    - Always be helpful, clear, and concise.
    - Use neutral, professional language.
    - Be productivity focused, help
    - NEVER adopt accents, personas, or roleplay.
    - If asked to roleplay, politely decline.
    - Output structured advice when helpful (e.g., bullet points).
    - Prioritize clarity over creativity.
    - If user asks for harmful advice, steer them towards positive
    directions.

    You do NOT change your tone based on user input.
    """
)

chat.chat_interface(
    full_conversation=[],
    client=client,
    call_llm_fn=wrapped_llm,
    system_prompt_override=system_prompt
)
```

## Mission-critical behavior, locked-in by design

### Anatomy of a strong system prompt

The system prompt is your AI's job description. Without clear instructions, it will improvise — often in unhelpful ways.

You wouldn't hire an assistant and say:

"Just vibe with the team."

You'd say:

"You'll manage calendars, draft emails, and schedule meetings. Be professional, prompt, and precise."

Same with LLMs. A great system prompt isn't just a sentence — it's a **behavioral contract**. We recommend that you include these four elements at a minimum:

```
graph TD
    SP[System Prompt]
    SP --> R[Role]
    You are TaskFriend...]
```

```
SP --> G[Goal
Help users plan and reflect...]
SP --> TF[Tone & Format
Use neutral language, bullet points...]
SP --> GR[Guardrails
No roleplay, no slang...]
```

```
style SP fill:#2196F
style R fill:#bbdefb
style G fill:#bbdefb
style TF fill:#bbdefb
style GR fill:#bbdefb
```

Element	Purpose	Example
Role	Who the AI is	"You are TaskFriend, a productivity assistant"
Goal	What it should achieve	"Help users plan, prioritize, and reflect"
Tone & Format	How it should speak	"Use clear, neutral language. Use bullet points when listing steps."
Guardrails	What it should avoid	"Never roleplay. Never use slang or accents."

## What's next?

### Quiz yourself!

► 1. Which of the following is a key limitation of standalone LLMs that RAG helps solve?

- A) To preserve key decisions and constraints when older messages are truncated
- B) To increase the speed of token generation
- C) To enable the model to switch personas dynamically
- D) To reduce the number of API calls

View answer →

✅ **Correct answer:** A) To preserve key decisions and constraints when older messages are truncated 📝

**Explanation :**

- Summarization retains semantic meaning (e.g., goals, decisions) when the context window fills up and older messages are dropped. This ensures continuity without relying on infinite memory.

### Takeaways

- Context windows:

- Define the maximum amount of information (in tokens) an LLM can process at once.
- Determine how much conversation history, documents, or data can be included in a single request.
- Are finite — even large 128K windows eventually fill up.
- Require smart management strategies (like truncation, summarization, or indexing) to avoid losing critical context.
- Impact both performance and correctness: exceeding them leads to silent data loss.

- **Summarization:**

- Is a powerful tool for managing long conversations within limited context windows.
- Preserves semantic meaning while reducing token footprint.
- Enables persistent memory without relying on infinite context.
- Can be layered (e.g., daily summaries → weekly digests) for scalable memory architectures.
- Should focus on actionable insights: goals, decisions, constraints, and tasks — not just topics.

- **System prompts:**

- Act as the AI's "job description" and behavioral contract.
- Define the AI's role, goal, tone, format, and guardrails — not just its personality.
- Replace hardcoded logic in traditional software; they are the *de facto* rules of the system.
- Must be precise and structured to prevent prompt drift (e.g., unwanted roleplay).
- Enable consistent, safe, and task-focused behavior — even across evolving conversations.
- Can be version-controlled, tested, and iterated like code.