Towards AI

Member-only story

Designing Customized and Dynamic Prompts for Large Language Models

A Practical Comparison of Context-Building, Templating, and Orchestration Techniques Across Modern LLM Frameworks

Shenggang Li Following 19 min read · 22 hours ago

136 1

Photo by Free Nomad on Unsplash

Introduction

Imagine you're at a coffee shop, and ask for a coffee. Simple, right? But if you don't specify details like milk, sugar, or type of roast, you might not get exactly what you wanted. Similarly, when interacting with large language models (*LLMs*), how you ask — your prompts — makes a big difference. That's why creating customized (static) and dynamic prompts is important. Customized prompts are like fixed recipes; they're consistent, reliable, and straightforward. Dynamic prompts, on the other hand, adapt based on the context, much like a skilled barista adjusting the coffee order based on your mood or the weather.

Let's say you're building an AI-powered customer support chatbot. If you use only static prompts, the bot might provide generic responses, leaving users

frustrated. For example, asking “How can I help you today”? is static and might be too vague. But a dynamic prompt might incorporate the user’s recent interactions, asking something like, “I see you were checking our order status. Would you like help tracking it further”? This personalized approach can dramatically improve user satisfaction.

I’ll dive into practical comparisons of these prompting methods, exploring context-building strategies, templating frameworks, and orchestration tools. I’ll examine real-world examples, from simple Q&A bots to complex conversational AI, highlighting why getting prompt design right isn’t just nice to have — it’s important for creating effective, human-like interactions with modern language models.

Customized Prompts

Customized prompts are instructions we tailor specifically for the task we want the language model to handle. They’re written with a clear goal in mind — whether it’s solving a particular problem, answering a type of question, or fitting a certain use case. We usually use them when we want the model to be more focused and give precise answers.

Common uses include:

- Achieve more accurate, reliable, and task-specific outputs.
- Minimize ambiguity and irrelevant responses.
- Improve user satisfaction by clearly defining expectations.

Generic prompt:

```
"Summarize the article below"
```

Customized prompt:

```
"Summarize the following article in no more than three sentences, highlighting t
```

- The customized prompt provides clear boundaries (no more than three sentences) and specifically targets “economic impacts” affecting “small businesses”.
- The customized prompt helps *LLMs* produce concise, relevant summaries precisely matching user intent.

Let’s say we’re building an AI assistant to help employees check their remaining vacation days.

If we use a **generic prompt** like:

“How many vacation days do I have left”?

The model might respond:

> *“I’m not sure. Vacation policies vary by company. Please check with your HR department”*

It's vague because the *LLM* doesn't know who's asking, what company they're in, or how vacation is tracked.

Now, compare that to a **customized prompt**:

> *"You are an HR assistant for XYZ Corp. Jane Doe is an employee who has used 8 of her 20 annual vacation days. Answer her question using this context".*

Then the user asks:

> *"How many vacation days do I have left"?*

The model can now respond accurately:

> *"Hi Jane! You have 12 vacation days remaining for this year".*

This small tweak — embedding context — change the result from vague and unhelpful to specific and useful. That's the power of thoughtful prompt design.

Dynamic Prompts

Dynamic prompts change and adjust on the fly, based on the situation or what's happened earlier in the conversation. Instead of using the same fixed instructions every time, they update as things progress — like if new info comes in or the user's needs shift. People often use dynamic prompts to:

- Make responses feel more personal and relevant to the user

- Keep the conversation flowing naturally
- React to changes in what the user wants or says along the way

Example:

Imagine a chatbot assisting with troubleshooting internet connectivity issues.

Initial dynamic prompt:

```
User: "My internet connection is slow."
AI dynamically generates: "Are you experiencing slow connectivity on all devices
```

Next dynamic prompt (based on user's answer):

```
User: "Only my laptop is slow."
AI dynamically generates: "Since only your laptop is slow, have you recently ins
```

- Dynamic prompts use previous interactions (the user's answers) to tailor each subsequent response.
- The prompt is dynamically based on new context, ensuring each question remains relevant.

Relationship and Difference

- Customized prompts and dynamic prompts can coexist. Often, dynamic prompting uses customized templates as a baseline.
- Customized prompts often serve as starting points; dynamic prompting further adjusts prompts in real-time.

Aspect	Customized Prompts	Dynamic Prompts
Adaptability	Static, pre-defined	Adaptive, real-time
Context Awareness	Fixed context provided explicitly	Context evolves with user interactions
Implementation	Manually crafted based on expertise	Automatically generated, algorithm-based
Usage Scenario	Specific, known tasks	Interactive, ongoing conversations
Complexity	Relatively simpler	Typically more complex (algorithmic logic)

In short, customized prompts are clear, task-specific, and manually created for predictable outcomes. Dynamic prompts adapt to real-time contexts and interactions for personalized, fluid conversations.

Use Customized and Dynamic Prompts in Practice

Code Overview and Workflow Explanation

This *Python* script implements an AI-based data exploration script generator for data scientists, powered by Google's *Gemini* model and Retrieval-Augmented Generation (*RAG*). Its main goal is to help data scientists

interactively explore datasets by producing on-the-fly code snippets in response to their queries, all backed by an internal knowledge base.

The goal is to help users explore data by generating useful code snippets based on what they ask, pulling info from an internal knowledge base.

The system uses:

- *Vertex* AI's *Gemini* model (*gemini-2.0-flash-001*)
- *Vertex* AI *RAG* retrieval tool (*rag_retrieval_tool*) to fetch relevant examples from a dedicated knowledge corpus (*CORPUS_NAME*)
- Customized (static) and dynamic (adaptive) prompts to guide each generation step

Workflow:

1. A customized, explicit prompt (*system_prompt*) is defined upfront. It explicitly instructs the model on how to generate scripts.
2. When a user provides input, the conversation history is dynamically compiled into a new, adaptive prompt.
3. This combined prompt (system + dynamic) is passed to the Gemini model.
4. The model generates contextually relevant scripts.
5. Results are output to the user interactively.

1. Manually Crafted Prompts with Built-in Context Awareness

Customized Prompt

Customized prompts are static, manually designed instructions tailored explicitly to guide the model's overall behavior.

In the given code, the customized prompt is clearly identified as the *system_prompt*:

```
system_prompt = """
You are a helpful data exploration expert specializing in providing data explora

INSTRUCTIONS:
1. Generate concise and precise EDA scripts without additional explanations or t
2. Default to relevant examples in the EDA document, but do not reject reasonabl
3. If you are not sure of the answer, you still must give EDA scripts, but you c
"""
```

Key Features of this Customized Prompt:

- Clearly defined role: The assistant specializes explicitly in *EDA* snippets.
- Output instructions: The assistant must generate concise code without textual explanations.
- Fallback behavior: It instructs the model not to reject user queries outright, providing fallback scripts if uncertain.

This static prompt sets clear expectations and shapes every interaction consistently.

Dynamic Prompt

Dynamic prompts adapt in real-time based on the ongoing conversation. They're built incrementally, maintaining context-awareness throughout an interactive dialogue.

In this code, the dynamic prompt is constructed:

```
def build_dynamic_prompt(history):
    dialog = []
    for msg in history:
        if msg["role"] == "user":
            dialog.append(f"User: {msg['content']}")
        elif msg["role"] == "assistant":
            dialog.append(f"Assistant: {msg['content']}")
    full_prompt = "\n".join(dialog) + "\nAssistant:"
    return full_prompt
```

- Captures all historical interactions (user-assistant pairs).
- Maintains the evolving context to enhance continuity and relevance.
- Always appends "*\nAssistant:*" at the end, signaling the model to produce the next relevant response.

Example (Dynamic prompt after several interactions):

Initialization of Model with Customized Prompt:

```
rag_model = GenerativeModel(
    model_name="gemini-2.0-flash-001",
    tools=[rag_retrieval_tool],
    system_instruction=system_prompt  # <-- Customized prompt is here!
)
```

```
prompt = build_dynamic_prompt(conversation_history)
assistant_reply = get_code_snippet(prompt)
```

Each time the model generates content, it receives a dynamically built prompt:

```
User: Show me the first few rows of the customer transactions dataset.
Assistant: Sure! Here's a snippet to load and preview the data:
df.head()

User: Now filter it to show only transactions greater than $100.
Assistant: Got it. Here's the updated code:
df[df['transaction_amount'] > 100]
```

- The Customized Prompt (*system_prompt*) sets fixed behavior rules for the model, ensuring consistent, reliable behavior aligned with specific goals.
- The Dynamic Prompt (*build_dynamic_prompt*) enables fluid, contextual, adaptive interactions, responding effectively to changing user needs during ongoing dialogues.

Here is full code:

```
import os
import sys
from google import genai
import vertexai

from vertexai.generative_models import GenerativeModel, Tool
from vertexai.preview import rag
from google.genai.types import GenerateContentConfig, Retrieval, Tool, VertexRag
import torch
```

```
# ---------------------------------------------------------------------
# Project/Environment Setup
# ---------------------------------------------------------------------
PROJECT_ID = "#####"
LOCATION = os.environ.get("GOOGLE_CLOUD_REGION", "us-central1")
vertexai.init(project=PROJECT_ID, location=LOCATION)
client = genai.Client(vertexai=True, project=PROJECT_ID, location=LOCATION)

EMBEDDING_MODEL = "publishers/google/models/text-embedding-004"

CORPUS_NAME = "projects/########"

from vertexai.generative_models import GenerativeModel, Tool
from vertexai.preview import rag

# ---------------------------------------------------------------------
# RAG Retrieval Tool
# ---------------------------------------------------------------------
rag_retrieval_tool = Tool.from_retrieval(
    retrieval=rag.Retrieval(
        source=rag.VertexRagStore(
            rag_resources=[rag.RagResource(rag_corpus=CORPUS_NAME)],
            similarity_top_k=1,
            vector_distance_threshold=0.3
        )
    )
)

# ---------------------------------------------------------------------
# System (Base) Instruction
# ---------------------------------------------------------------------
system_prompt = """
You are a helpful data exploration code assistant specializing in providing EDA

INSTRUCTIONS:
1. Generate concise and precise EDA scripts without additional explanations or t
2. Default to relevant examples in the EDA document, but do not reject reasonabl
3. If you are not sure of the answer, you still must give EDA scripts, but you c
"""

# ---------------------------------------------------------------------
# Initialize the Gemini Model with RAG
# ---------------------------------------------------------------------
rag_model = GenerativeModel(
    model_name="gemini-2.0-flash-001",
    tools=[rag_retrieval_tool],
    system_instruction=system_prompt
)

# ---------------------------------------------------------------------
```

```
# Dynamic Prompt Management
# ---------------------------------------------------------------------
conversation_history = []

def build_dynamic_prompt(history):
    """
    Construct a dynamic prompt string from conversation history.
    The system instruction is already set within the model initialization.
    We only compile user/assistant pairs here.
    """
    dialog = []
    for msg in history:
        if msg["role"] == "user":
            dialog.append(f"User: {msg['content']}")
        elif msg["role"] == "assistant":
            dialog.append(f"Assistant: {msg['content']}")
    # Append an 'Assistant:' prompt for the new response
    full_prompt = "\n".join(dialog) + "\nAssistant:"
    return full_prompt



# ---------------------------------------------------------------------
# Generate Code Snippet
# ---------------------------------------------------------------------
def get_code_snippet(full_prompt: str) -> str:
    """
    Passes the constructed prompt to the model for generation.
    Returns the model's text response (cleaned as needed).
    """
    try:
        if not full_prompt or not isinstance(full_prompt, str):
            raise ValueError("Prompt must be a non-empty string")
        response = rag_model.generate_content(full_prompt)
        if not response or not response.text:
            raise ValueError("No valid response received from the model")
        # Simple cleaning example
        cleaned = response.text.replace('attr">', '').replace('Attr">', '').repl
        return cleaned.strip()
    except Exception as e:
        return f"❌ Error: {str(e)}"

# ---------------------------------------------------------------------
# Interactive Conversation Logic
# ---------------------------------------------------------------------
def interactive_mode():
    """
    Continuously prompt the user for input, update conversation history,
    generate a new dynamic prompt, and output the model's response.
    The loop ends if the user types 'exit' or 'quit'.
```

```
    """
    print("Entering interactive dynamic-prompt mode. Type 'exit' or 'quit' to st
    while True:
        user_input = input("📝 Your request: ").strip()
        if user_input.lower() in ["exit", "quit"]:
            print("Exiting interactive mode.")
            break

        # Store user's message in conversation history
        conversation_history.append({"role": "user", "content": user_input})

        # Build a dynamic prompt based on entire conversation
        prompt = build_dynamic_prompt(conversation_history)

        # Get the model's response
        assistant_reply = get_code_snippet(prompt)

        # Print the model's output
        print(f"\n🤖 Assistant:\n{assistant_reply}\n")

        # Add the assistant's reply back to the conversation history
        conversation_history.append({"role": "assistant", "content": assistant_r

# ------------------------------------------------------------------
# Main
# ------------------------------------------------------------------
def main():
    if len(sys.argv) > 1:
        user_prompt = " ".join(sys.argv[1:]).strip()
        conversation_history.append({"role": "user", "content": user_prompt})
        prompt = build_dynamic_prompt(conversation_history)
        assistant_reply = get_code_snippet(prompt)
        print(assistant_reply)
    else:
        interactive_mode()

if __name__ == "__main__":
    main()
```

2. Use `dspy` on Customized and Dynamic Prompting

DSPy (Declarative Structured Prompting for You) takes away the pain of stitching together giant prompt strings by giving you a tiny, structured workflow.

Imagine we're building a question-answering assistant that uses Retrieval-Augmented Generation (*RAG*). Without *DSPy*, we might write something like this:

```
# Pseudo-code WITHOUT DSPy
docs = rag_retrieve(user_question)                              # Step 1: fetch top
prompt = f"{docs}\nUser: {user_question}\nAssistant:"           # Step 2: manually s
response = gemini.generate_content(prompt).text                 # Step 3: call the m
print(response)                                                 # Step 4: display an
```

This ad-hoc approach works at first — but as our prompt gets more complex (multiple docs, safety checks, follow-up turns), we end up with a tangle of + operators and subtle bugs: missing newlines, forgotten sections, or mismatched braces. Silent mistakes in our prompt often lead to wildly wrong answers, and debugging them is hard.

How DSPy Fixes This?

A) Declare "Contract" Up Front:

```
class QA(Signature):
    question: str = InputField(desc="RAG context + user question")
    answer:   str = OutputField(desc="Model's final answer")
```

Here, we tell *DSPy*: "I always send in one string (*question*) and expect one string back (*answer*)."

B) Configure *LLM* and *RAG* Tool Once

```
lm = dspy.LM("vertex_ai/gemini-2.0-flash-001", project="…", location="…")
dspy.configure(lm=lm)
```

DSPy now knows our model, our *RAG* retriever is attached, and any system-level instructions are locked in.

C) Instantiate a Predictor

```
qa = Predict(QA, adapter="text")
```

This single line sets up everything: input validation, string formatting, model invocation, and output mapping.

D) Run Loop with Confidence

```
# Step A: fetch context
docs = rag_retrieve(user_question)

# Step B: build the dynamic prompt
full_prompt = f"{docs}\nUser: {user_question}\nAssistant:"

# Step C: call DSPy and get structured output
result = qa(question=full_prompt)
print("Answer:", result.answer)
```

Below is the complete implementation of a *DSPy*-based system for generating *EDA* scripts:

```
import os
import dspy
from dspy import InputField, OutputField, Signature, Predict

# ------------------------------------------------------------------
# 1. Configure DSPy to use Vertex AI's Gemini
# ------------------------------------------------------------------
lm = dspy.LM(
    "vertex_ai/gemini-2.0-flash-001",
    project="############",
    location="us-central1"
)
dspy.configure(lm=lm)

# ------------------------------------------------------------------
# 2. Define your structured prompt signature
# ------------------------------------------------------------------
class GenerateCode(Signature):
    """Structured Prompt for Generating data exploration Code"""
    input: str = InputField(desc="Full chat context + user request")
    response: str = OutputField(desc="Generated EDA snippet")

# ------------------------------------------------------------------
# 3. Create the DSPy prediction module
# ------------------------------------------------------------------
generate_code = Predict(signature=GenerateCode)

# ------------------------------------------------------------------
# 4. Interactive, context-aware loop
# ------------------------------------------------------------------
conversation_history = []

def interactive_dspy():
    print("🧠 DSPy Interactive Mode (type 'exit' or 'quit' to stop)\n")
    while True:
        user_input = input("📝 You: ").strip()
        if user_input.lower() in ("exit", "quit"):
            print("👋 Goodbye!")
            break

        # append user turn
        conversation_history.append({"role": "user", "content": user_input})
```

```
        # build prompt with full history
        dynamic_input = "\n".join(
            f"{turn['role'].capitalize()}: {turn['content']}"
            for turn in conversation_history
        ) + "\nAssistant:"

        # call DSPy
        result = generate_code(input=dynamic_input)

        # show assistant response and append to history
        print(f"\n🤖 Assistant:\n{result.response}\n")
        conversation_history.append({"role": "assistant", "content": result.resp

if __name__ == "__main__":
    interactive_dspy()
```

- The system_instruction string we pass when calling *dspy.LM(...)* serves as your customized prompt, locking in the assistant's overall behavior and static guidelines.
- By creating a GenerateCode signature and using *dspy.Predict(..., adapter="text")*, *DSPy* treats each call as "one prompt in, one raw string out," which effectively standardizes and simulates your dynamic prompt compilation.
- At each turn, we build the dynamic prompt by concatenating our *conversation_history* (user + assistant turns) with the new user request, then hand that single string to *generate_code(input=...)*, giving the model full context.
- Defining *GenerateCode* with *InputField* and *OutputField* makes our prompt schema explicit and reusable-*DSPy* enforces correct *inputs/outputs*, so we can easily plug it into follow-up queries or multi-step pipelines without rewriting glue code.

Why *DSPy*

- Fewer Prompt Bugs: *DSPy* checks that we provided the question field and that you handle the answer field-it won't let us typo a variable name or forget a newline.
- Easier Debugging: With *dspy.inspect_history()*, we can instantly replay the last few prompts and responses in a structured log, rather than printing raw strings.
- Backend Flexibility: Want to switch from Gemini to OpenAI? Just call dspy.configure(*lm=LM("openai/gpt-4")*)-your QA code stays the same..
- Modular Reuse: Once we've defined a Signature for QA, we can reuse it in

In short, *DSPy* turns your "hand-rolled prompt factory" into a small, declarative pipeline — making *RAG* applications more robust, maintainable, and easier to scale.

3. `dynamic-prompting` Works on Customized and Dynamic Prompting

The *dynamic-prompting* library is a lightweight and expressive framework for building flexible, real-time prompts for large language models (*LLMs*). It allows users to create prompt templates using Pythonic expressions and register dynamic variables that change over time or context. This approach is particularly useful when combining customized base instructions with real-time, context-sensitive updates, making it a good fit for applications like interactive *EDA* code assistants powered by *Gemini + RAG*.

The *dynamic_prompting* separates your static instructions ("system prompt") from the dynamic bits that change every time we call the model (like

conversation history or retrieved context). We start by writing one single template—say:

```
{conversation_history}
Assistant:
```

We now hand that template to a *PromptManagement* object, which keeps track of your base instructions and knows exactly where to plug in whatever we pass it at runtime.

How It Works in a Simple RAG Loop

1. Fetch relevant docs from our knowledge base via *RAG*.
2. Record the user's turn (and any previous assistant replies) in a list.
3. Call *PromptManagement.set_few_shots* (or simply *pm.prompt.format*) to merge the current history—and even a latest retrieval snippet—into the template.
4. Send the filled-in prompt to *Gemini* (or any *LLM*).

Because the template is defined once and never hand-edited again, we avoid the classic "missing newline", "forgot to include the system message", or "typo in my curly braces" bugs that plague manual string concatenation.

Here is the code of a *dynamic_prompting*-based system for generating *EDA* scripts:

```
import os
from tqdm.autonotebook import tqdm
from dynamic_prompting.llms.prompt import PromptManagement
from dynamic_prompting.llms.config import PromptConfig
from vertexai.generative_models import GenerativeModel, Tool
from vertexai.preview import rag

# ------------------------------------------------------------------
# 1) RAG Retrieval Tool
# ------------------------------------------------------------------
CORPUS_NAME = "####"

retrieval = rag.Retrieval(
    source=rag.VertexRagStore(
        rag_resources=[rag.RagResource(rag_corpus=CORPUS_NAME)],
        similarity_top_k=1,
        vector_distance_threshold=0.3
    )
)
rag_tool = Tool.from_retrieval(retrieval=retrieval)

# ------------------------------------------------------------------
# 2) Initialize Gemini Model with Static (Customized) Prompt
# ------------------------------------------------------------------
system_instruction = """
You are a helpful EDA expert specializing in data analysis.
INSTRUCTIONS:
- Provide concise, runnable Python EDA snippets.
- Default to examples in the documentation.
- If uncertain, still return code with a brief warning comment.
  you must give an answer
"""
gemini_model = GenerativeModel(
    model_name="gemini-2.0-flash-001",
    tools=[rag_tool],
    system_instruction=system_instruction
)

# ------------------------------------------------------------------
# 3) Set Up PromptManagement for Dynamic Prompting
# ------------------------------------------------------------------
# Define a template that injects the evolving conversation history
prompt_cfg = PromptConfig(
    prompt="""
{conversation_history}
Assistant:"""
)
```

```
pm = PromptManagement(prompt_cfg)

conversation_history = []

def get_dynamic_prompt():
    """
    Renders the prompt by formatting the conversation history
    into the static template managed by PromptManagement.
    """
    history = "\n".join(
        f"User: {turn['content']}" if turn["role"] == "user"
        else f"Assistant: {turn['content']}"
        for turn in conversation_history
    )
    return pm.prompt.format(conversation_history=history)

# ------------------------------------------------------------------
# 4) Generate Code Snippet
# ------------------------------------------------------------------
def generate_code(prompt_text: str) -> str:
    response = gemini_model.generate_content(prompt_text)
    return response.text.strip()

# ------------------------------------------------------------------
# 5) Interactive Loop
# ------------------------------------------------------------------
def interactive_mode():
    print("🚀 Entering dynamic-prompting mode. Type 'exit' to quit.")
    while True:
        user_input = input("📝 Your request: ").strip()
        if user_input.lower() in ("exit", "quit"):
            break

        # 5a) Append user turn
        conversation_history.append({"role": "user", "content": user_input})

        # 5b) Build and render dynamic prompt
        dynamic_prompt = get_dynamic_prompt()

        # 5c) Generate and display the code snippet
        snippet = generate_code(dynamic_prompt)
        print(f"\n🤖 Assistant:\n{snippet}\n")

        # 5d) Append assistant turn
        conversation_history.append({"role": "assistant", "content": snippet})

if __name__ == "__main__":
    interactive_mode()
```

- The *system_instruction* we pass when creating the *GenerativeModel* still serves as our customized prompt, locking in the assistant's static behavior.
- The *PromptManagement* instance holds a *PromptConfig.prompt* template with a *{conversation_history}* placeholder that we populate anew on each turn.
- Each time we call *PromptManagement.prompt.format(…),* it injects the latest chat history into template, producing a dynamic prompt that perfectly blends evolving context with predefined instructions..

Why *dynamic_prompting*

- Cleaner, more consistent prompts lead to more accurate, on-point responses — *LLMs* behave best when their instructions follow the exact same shape every time.
- Automatic context management means we never accidentally drop old turns or mix up our retrieval snippet with user text. Better context → fewer hallucinations.
- Easier experimentation: To tweak our prompt, we update one template file instead of hunting through Python code for every + *"\n"* + line.

Here, *dynamic_prompting* feels like magic glue: we write template once, then simply feed in fresh pieces each turn. Under the hood, it handles all the formatting, so the *RAG*-powered assistant stays sharp, consistent, and much easier to maintain than a tangle of *f"…{x}…{y}…"* everywhere.

5. Jinja2 on Customized and Dynamic Prompting

Jinja2 is a mature *Python* templating engine originally created for generating HTML, but it works just as well for composing *LLM* prompts. Instead of manually gluing together strings with + and f-strings every time our prompt changes, we define one template file that mixes our **static** system instructions with **dynamic** fields—like retrieval snippets or conversation turns. Under the hood, *Jinja2* parses this template once and then quickly substitutes in those variables at runtime.

RAG Example Without *Jinja2*:

```
# Manual string assembly for each turn
prompt = (
    system_instruction + "\n\n"
    "-- Retrieved Context --\n" + snippet + "\n\n"
    "-- Conversation History --\n"
    + "\n".join(f"{r['role']}: {r['content']}" for r in history)
    + "\n\nAssistant:"
)
response = model.generate_content(prompt)
```

This works, but as the prompts grow — adding safety checks, examples, loops over multiple documents — it becomes a tangle of +, parentheses, and hard-to-spot missing newlines. Typos or misplaced braces often lead to malformed prompts and unpredictable model outputs.

The *Jinja2*-Powered *RAG*:

```
{{ system_instruction }}
```

```
-- Retrieved Context --
{{ retrieval_snippet }}

-- Conversation History --
{% for turn in history %}
{{ turn.role }}: {{ turn.content }}
{% endfor %}

Assistant:
```

Load and render in *Python*:

```
tpl = jinja2.Template(open("prompt_template.txt").read())
prompt_text = tpl.render(
    system_instruction=system_instruction,
    retrieval_snippet=snippet,
    history=conversation_history
)
response = model.generate_content(prompt_text)
```

Now our code always follows the same structure, and we only update variables, never the template itself. For each interaction, the template is rendered with the updated context, and the resulting prompt string is then submitted to the *Gemini* model.

Here is the complete code for generating *EDA* scripts:

```
import jinja2
from vertexai.generative_models import GenerativeModel, Tool
from vertexai.preview import rag

# ------------------------------------------------------------------
# 1) RAG Retrieval Tool
# ------------------------------------------------------------------
```

```
CORPUS_NAME = "######"
retrieval = rag.Retrieval(
    source=rag.VertexRagStore(
        rag_resources=[rag.RagResource(rag_corpus=CORPUS_NAME)],
        similarity_top_k=1,
        vector_distance_threshold=0.3,
    )
)
rag_tool = Tool.from_retrieval(retrieval=retrieval)

# ------------------------------------------------------------------
# 2) Customized Prompt: static system instruction
# ------------------------------------------------------------------

system_instruction = """
You are a helpful data exploration expert specializing in efficient code generat

INSTRUCTIONS:
1. Generate concise and precise EDA scripts without additional explanations or p
2. Default to relevant examples from the knowledge base, but do not refuse reaso
3. If you are uncertain of the answer, still provide working EDA scripts, and in
"""

gemini_model = GenerativeModel(
    model_name="gemini-2.0-flash-001",
    tools=[rag_tool],                       # RAG tool is attached here
    system_instruction=system_instruction.strip(),
)

# ------------------------------------------------------------------
# 3) Jinja2 Template
# ------------------------------------------------------------------
template_source = """
{{ system_instruction }}

-- Conversation History --
{% for turn in history %}
{{ turn.role }}: {{ turn.content }}
{% endfor %}

Assistant:
"""
prompt_template = jinja2.Template(template_source)

# ------------------------------------------------------------------
# 4) Build prompt (no manual retrieval)
# ------------------------------------------------------------------
conversation_history: list[dict[str,str]] = []

def build_prompt() -> str:
```

```
        return prompt_template.render(
            system_instruction=system_instruction,
            history=conversation_history,
        )

    # ------------------------------------------------------------------
    # 5) Interactive loop
    # ------------------------------------------------------------------
    def interactive_mode():
        print("🚀 Entering Jinja2-powered mode (type 'exit'):")
        while True:
            user_input = input("📝 Your request: ").strip()
            if user_input.lower() in ("exit", "quit"):
                break

            # 5a) Record user turn
            conversation_history.append({"role":"User","content":user_input})

            # 5b) Build the combined prompt
            prompt_text = build_prompt()

            # 5c) Let Gemini auto-invoke RAG and generate code
            response = gemini_model.generate_content(prompt_text).text.strip()
            print(f"\n🤖 Assistant:\n{response}\n")

            # 5d) Record assistant turn
            conversation_history.append({"role":"Assistant","content":response})

    if __name__ == "__main__":
        interactive_mode()
```

- **Customized prompt:** The system_instruction provided during model initialization defines a stable, customized baseline for the assistant's behavior.
- **Dynamic prompt:** The *Jinja2* template injects the *RAG*-derived retrieval_snippet and iterates over the conversation_history, ensuring each new turn is reflected automatically.
- **Separation of concerns:** Using static instructions, retrieval context, and dialogue history into distinct template blocks, updates — such as adding

user metadata or extra sections — require only minimal edits to the template.

Why Jinja2?

- **Consistency**: Every prompt follows the exact same layout — no more accidental typos or missing sections.
- **Maintainability**: Changing our instructions or adding a new section (e.g. "Examples:") means editing one template file instead of hunting through code.
- **Readability**: Our prompt logic lives in a clean, human-readable template, separate from application logic.
- **Debugging**: When something goes wrong, we can print or lint our filled-in template before sending it to the model.

Using *Jinja2*, we spend less time wrestling with string bugs and more time tuning system instructions. That leads to sharper, more accurate *RAG*-augmented responses and a smoother development experience for beginners and experts alike.

6. LangChain on Customized and Dynamic Prompting

LangChain is a modular framework for building *LLM* applications that brings together prompt templates, memory, tool integration, and chain/agent logic. This let developers declaratively specify both static "system" instructions and dynamic, runtime context — while plugging in external tools such as *RAG* retrievers.

With *LangChain*, we define customized prompt just once when calling *initialize_agent(…)*: that's where we tell *Gemini* who it is ("data-exploration expert…") and attach the *RAG* tool. Then, *LangChain* takes care of merging together three pieces every time we call *agent.run(user_input)*:

1. System instructions (static, customized prompt)
2. Retrieved context (the top result from our knowledge base via rag_retriever)
3. Conversation history (everything we and the assistant have already said, pulled from *ConversationBufferMemory*)

Because *LangChain* sits in the driver's seat, we never have to manually glue strings together.

It automatically formats a prompt like:

```
You are a data exploration expert…

-- Retrieved Context --
<most relevant snippet from corpus>

-- Conversation History --
User: First question
Assistant: First answer
User: Second question

Assistant:
```

Two-Step RAG Example & Why It Helps

First *RAG* Prompt:

```
response1 = agent.run("What are the key stats I should check on my dataset?")
```

What happens: *LangChain* calls *rag_retriever("What are the key stats…")*, fetches a snippet of best practices from corpus, appends it to static system prompt, and sends the full request to Gemini. We get back a concise list of *EDA* steps..

Follow-up *RAG* Prompt:

```
response2 = agent.run("Show me the Python code to compute those stats on a DataF
```

What happens: Now the *chat_history* contains the previous Q&A, so *LangChain* builds a new prompt combining the same system instruction, a fresh *RAG* snippet if relevant, and the entire conversation. *Gemini* returns ready-to-run code that picks up exactly where we left off..

Here is the *LangChain* code for generating *EDA* scripts:

```
import os
from langchain_google_genai import ChatGoogleGenerativeAI
from langchain.agents import initialize_agent, Tool
from langchain.memory import ConversationBufferMemory
from vertexai.preview import rag

# —— 1) Vertex AI RAG Retrieval Setup ————————————————————————————
CORPUS = "#####"
retrieval = rag.Retrieval(
```

```
        source=rag.VertexRagStore(
            rag_resources=[rag.RagResource(rag_corpus=CORPUS)],
            similarity_top_k=1,
            vector_distance_threshold=0.3,
        )
    )

    def rag_retriever(query: str) -> str:
        # NOTE: use `.search()`, not `.retrieve()`
        hits = retrieval.search(query)
        return hits[0].text if hits else ""

    rag_tool = Tool(
        name="RAGRetriever",
        func=rag_retriever,
        description="Fetch relevant data-exploration snippets from the knowledge bas
    )

    # ── 2) LangChain LLM via Google GenAI (Gemini) ──────────────────────────
    from langchain_google_genai import ChatGoogleGenerativeAI

    llm = ChatGoogleGenerativeAI(
        model="gemini-2.0-flash-001",
        api_key=os.getenv("GOOGLE_API_KEY"),  # ← use `api_key`, not `google_api_key
        temperature=0.0,
        max_tokens=None,
        timeout=None,
        max_retries=2,
    )

    # ── 3) Agent + Memory Configuration ─────────────────────────────────────
    memory = ConversationBufferMemory(memory_key="chat_history", return_messages=Tru

    agent = initialize_agent(
        tools=[rag_tool],
        llm=llm,
        agent="zero-shot-react-description",
        memory=memory,
        verbose=False,
    )

    # ── 4) Interactive Loop ─────────────────────────────────────────────────
    def interactive_mode():
        print("🚀 LangChain + Gemini + RAG (type 'exit' to quit)")
        while True:
            user_input = input("📝 Your request: ").strip()
            if user_input.lower() in ("exit", "quit"):
                break

            # LangChain agent will:
```

```
            #   1) call rag_retriever(user_input) to fetch context
            #   2) build a prompt with system + history + retrieved snippet
            #   3) invoke Gemini under the hood
            response = agent.run(user_input)

            print(f"\n🤖 Assistant:\n{response}\n")

    if __name__ == "__main__":
        interactive_mode()
```

- **Customized Prompt:** We define our assistant's static behavior up front — e.g. "You are a data exploration expert. Provide minimal, runnable *EDA* snippets…" — when we configure the *LLM* and agent. This ensures *Gemini* always follows the same baseline instructions.
- **Dynamic Prompt:** On every invocation of *agent.run(user_input)*, *LangChain* automatically (1) retrieves the most relevant snippet from the knowledge base via our *RAGRetriever* tool, and (2) pulls in the full conversation history from *ConversationBufferMemory*. Those pieces are merged into the prompt at runtime-no manual string-gluing required..
- **Tool Integration:** The *rag_retriever()* function is wrapped as a *LangChain* Tool, cleanly isolating "fetch context" logic from "generate text" logic. The agent simply calls that tool when it needs retrieval, keeping the code modular.

Why *LangChain?*

- **Consistency & Fewer Bugs:** We define layout once — no more missing newlines or forgotten sections.
- **Tighter Retrieval Integration:** The agent only calls the *RAG* function when needed, reducing hallucinations by always grounding answers in the knowledge base.

- **Built-in Memory:** Follow-ups feel natural because the history is managed automatically, so we don't need to reinvent a context stack.
- **Rapid Iteration:** To tweak our system prompt, we can update one argument in initialize_agent; to change retrieval logic, we adjust only our *rag_retriever* function.

For AI system developers, *LangChain's* dynamic and customized prompting means we can focus on what we want assistant to do — interactive, *RAG*-powered *EDA* — without wrestling with low-level string manipulation.

Wrapping Up & Takeaways

We've tried out six ways to blend your "always-on" instructions with "right-this-second" context, and each has its sweet spot:

- **Quick experiments or throwaways:** If we just want to hack something together, hand-rolled string building or a tiny templating lib (like *dynamic-prompting*) gets us running in minutes.
- **A bit more logic in prompts:** *Jinja2* is fantastic when we need loops, if-else blocks, or reusable macros.
- **Modular, testable flows:** *DSPy*'s signature-based approach formalizes our inputs, outputs, and chaining — perfect for building and debugging multi-step pipelines.
- **Full-blown production bots:** *LangChain* gives us *RAG*, chat memory, tool invocation, and more, all in one framework — just be ready for a bigger install and a bit of a learning curve.

About me

With over 20 years of experience in software and database management and 25 years teaching IT, math, and statistics, I am a Data Scientist with extensive expertise across multiple industries.

You can connect with me at:

Email: datalev@gmail.com | LinkedIn | X/Twitter

Llm Gemini AI Prompt Retrieval Augmented Gen

Published in Towards AI

80K Followers · Last published 14 hours ago

Follow

The leading AI community and content platform focused on making AI accessible to all. Check out our new course platform: https://academy.towardsai.net/courses/beginner-to-advanced-llm-dev

Written by Shenggang Li

2.5K Followers · 77 Following

Following

Responses (1)

Alex Mylnikov

What are your thoughts?

Nehdiii
16 hours ago

Ihsen Mezni

Reply

More from Shenggang Li and Towards AI

In Towards AI by Shenggang Li

Graph Neural Networks: Unlocking the Power of Relationships in...

Exploring the Concepts, Types, and Real-World Applications of GNNs in Feature...

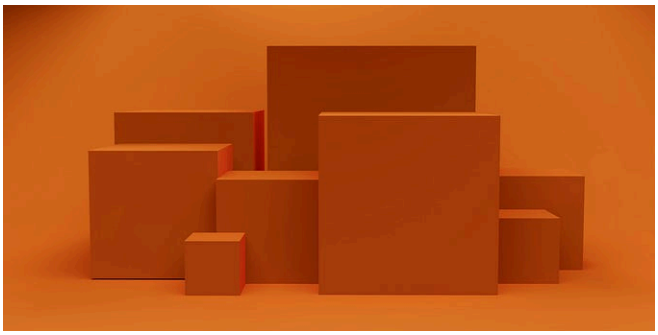
Jan 11 310 3

In Towards AI by Shenggang Li

Reinforcement Learning for Business Optimization: A Genetic...

Applying PPO and Genetic Algorithms to Dynamic Pricing in Competitive Markets

Mar 17 188 3



In Data Science Collective by Shenggang Li

Comparing Binary Forecasting Methods for Financial Time...

Introduction

3d ago 150 1

In Towards AI by Shenggang Li

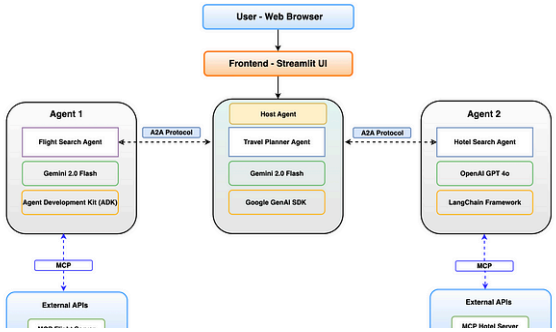
Stock Prices: Predictable Patterns or Pure Chance?

Statistical Analysis of Stock Market Trends Across Bull and Bear Phases, Uncovering...

Oct 31, 2024 338 8

See all from Shenggang Li See all from Towards AI

Recommended from Medium

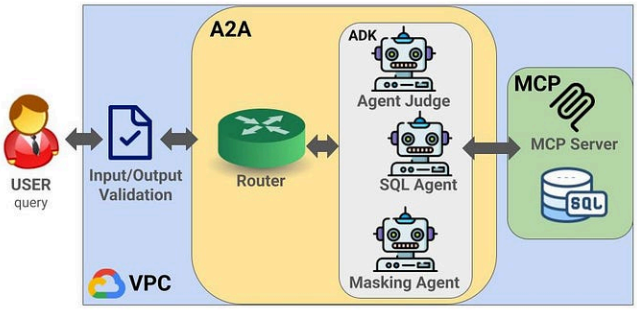


In AI Cloud Lab by Arjun Prabhulal

Building Multi-Agent AI App with Google's A2A (Agent2Agent)...

Multi Agent App using A2A Protocol , ADK and MCP

Apr 18 439 15



Rubens Zimbres

Agent Development Kit: Enhancing Multi-Agents Systems with A2A...

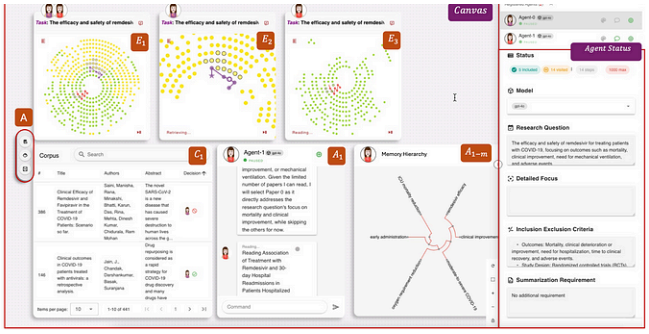
Lately we've been flooded with new innovations and product launches. I was at...

Apr 18 235 2

In Towards AI by Murat Şimşek

Building a Multi-Agent System with Multiple MCP Servers using...

A comprehensive guide to write custom MCP server, use existing MCP servers and multi-...

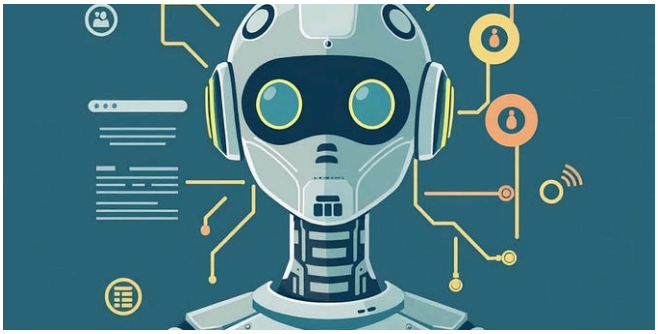


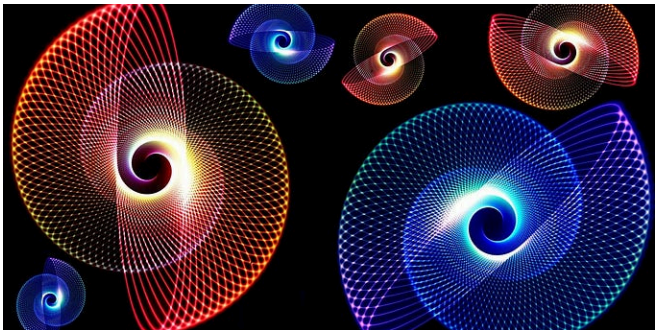
Cobus Greyling

InsightAgent Is Transforming Systematic Reviews with AI-...

This user interface introduces a new accelerated paradigm in performing...

5d ago 252 1 3d ago 48





In Coding Nexus by Algo Insights

How I Built My Own AI Web Agent (and Saved Hundreds a Month!)

A few months ago, I was shelling out way too much on subscription services for AI tools.

5d ago 380 7

In Fru.dev by Fru

15+ Top MCP Servers & Directories: Curated for Data Engineering, AI...

From Snowflake to Slack, explore the top MCP servers empowering LLMs with real-world...

6d ago 245 3

See more recommendations