Context-Aware Prompt Engine

Foundation Module for Enterprise AI Commerce System

---

1. What It Is

The Context-Aware Prompt Engine is a structured prompt construction layer that dynamically injects:

- System instruction
- User query
- Structured metadata
- Business constraints
- Output format enforcement

It ensures that LLM responses are controlled, compliant, and predictable in enterprise environments.

This is the foundational intelligence layer before adding retrieval, embeddings, or RAG.

---

2. Demo Scenario

User asks:

“Suggest running shoes under ₹5000”

The system:

1. Injects role instruction (Enterprise AI Commerce Assistant)
2. Injects pricing constraint
3. Enforces strict JSON output
4. Generates a structured response

---

3. What This Proves

- Context Engineering
- Structured Prompt Design
- Business Rule Injection
- Controlled Output
- Enterprise LLM Governance

This becomes the base LLM layer of the larger architecture.

Example 1 — Google Colab Environment

This example is optimized for quick execution in Colab.

Step 1 — Install Dependency (Colab Cell)

```
!pip install openai
```

Step 2 — Set API Key Securely

```
import os
from getpass import getpass

os.environ["OPENAI_API_KEY"] = getpass("Enter your OpenAI API Key: ")
```

Step 3 — Context-Aware Prompt Engine Code

```
from openai import OpenAI
import json

client = OpenAI()

class ContextAwarePromptEngine:

  def __init__(self, model_name="gpt-4o-mini"):
    self.model_name = model_name

  def build_prompt(self, user_query: str, max_budget: int):

    system_instruction = """
You are an Enterprise AI Commerce Assistant.
Follow all business rules strictly.
Always return valid JSON only.
Do not provide explanations outside JSON.
"""

    business_rules = f"""
Business Constraints:
- Recommend only running shoes.
- Maximum price allowed: ₹{max_budget}.
- Provide concise professional reasoning.
"""

      output_format = """
Return strictly in this JSON format:

{
  "recommendations": [
      {
        "name": "Product Name",
        "price": 0000,
        "reason": "Professional justification"
      }
  ]
}
"""

      final_prompt = f"""
{system_instruction}

{business_rules}

{output_format}

User Query:
{user_query}
"""
      return final_prompt

  def generate(self, user_query: str, max_budget: int):

      prompt = self.build_prompt(user_query, max_budget)
```

```
        response = client.chat.completions.create(
            model=self.model_name,
            messages=[{"role": "user", "content": prompt}],
            temperature=0.3
        )

        return response.choices[0].message.content


engine = ContextAwarePromptEngine()

result = engine.generate(
    user_query="Suggest running shoes suitable for daily jogging.",
    max_budget=5000
)

print(result)
```

Example 2 — VS Code Environment (Professional Structure)

This example follows proper enterprise file structuring.

Step 1 — Project Structure

```
context_prompt_engine/
│
├── main.py
├── requirements.txt
└── .env
```

Step 2 — requirements.txt

```
openai
python-dotenv
```

Install:

```
pip install -r requirements.txt
```

Step 3 — .env File

```
OPENAI_API_KEY=your_api_key_here
```

Step 4 — main.py

```
from openai import OpenAI
from dotenv import load_dotenv
import os


load_dotenv()


client = OpenAI()


class ContextAwarePromptEngine:
  """
  Enterprise Context Engineering Module
  """


  def __init__(self, model_name="gpt-4o-mini"):
    self.model_name = model_name

  def build_prompt(self, user_query: str, max_budget: int):
    system_instruction = """
You are an Enterprise AI Commerce Assistant.
You must comply with all business constraints.
Output must be valid JSON only.
No additional text outside JSON.
"""

    metadata = f"""
Metadata:
Category: Running Shoes
Budget Limit: ₹{max_budget}
"""

    output_schema = """
JSON Schema:

{
  "recommendations": [
    {
      "name": "Product Name",
      "price": 0000,
      "reason": "Professional reasoning"
    }
  ]
}
"""

    return f"""
{system_instruction}

{metadata}

{output_schema}
```

```
User Query:
{user_query}
"""

    def generate(self, user_query: str, max_budget: int):

        prompt = self.build_prompt(user_query, max_budget)

        response = client.chat.completions.create(
            model=self.model_name,
            messages=[
                {"role": "user", "content": prompt}
            ],
            temperature=0.2
        )

        return response.choices[0].message.content


if __name__ == "__main__":

    engine = ContextAwarePromptEngine()

    response = engine.generate(
        user_query="Suggest running shoes under ₹5000 for daily jogging.",
        max_budget=5000
    )

    print("Structured AI Output:")
    print(response)
```

Run:

```
python main.py
```

What This Module Is Doing Technically

1. Defines AI identity
2. Injects business metadata
3. Applies pricing constraint
4. Enforces strict JSON schema
5. Uses low temperature for deterministic output

Which Part of the Larger Goal This Solves

In the complete Enterprise AI Commerce Architecture, this module solves:

Foundation Layer: Context Engineering

It enables:

- Controlled LLM behavior
- Enterprise compliance
- Predictable formatting
- Safe generation

It is required before:

- Embedding generation
- Vector search
- Retrieval-augmented generation
- Hallucination mitigation
- LLM version tracking

Without this layer, the larger RAG system would produce uncontrolled and non-governed responses.