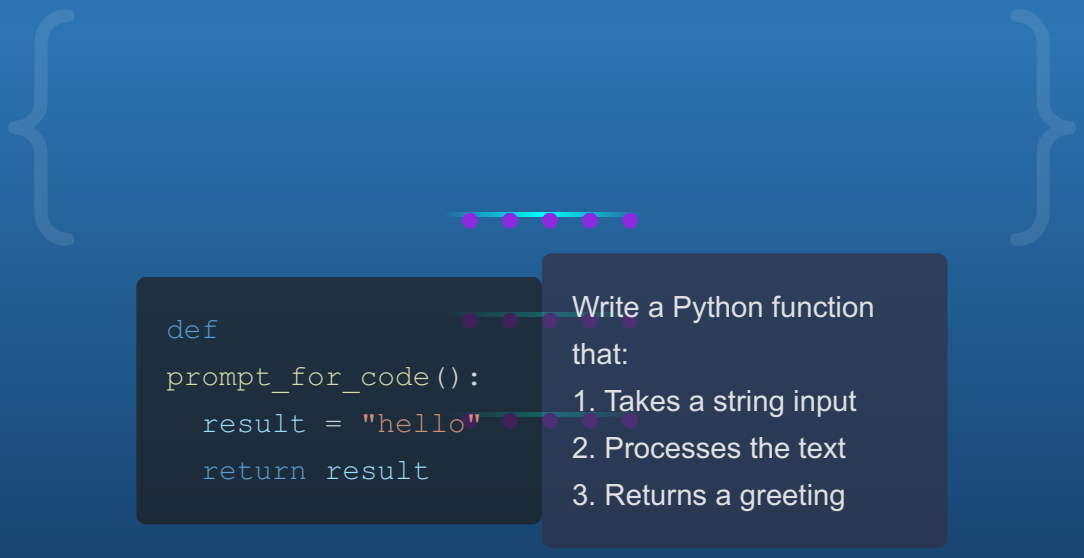
PROMPT ENGINEERING
FOR DEVELOPERS

Crafting Intelligent LLM Solutions



By Sunny Shivam

PROMPT ENGINEERING FOR DEVELOPERS

Crafting Intelligent LLM Solutions

sunny shivam

About the Book

Prompt Engineering for Developers: Crafting Intelligent LLM Solutions is a comprehensive, hands-on guide designed specifically for developers who want to harness the power of large language models (LLMs) in their daily work and projects. Rather than focusing on theoretical concepts, this book takes a practical, example-driven approach to teaching prompt engineering as an essential developer skill.

Who This Book Is For

This book is written for: - **Software developers** with Python knowledge who want to integrate LLMs into their applications - **Data scientists** looking to leverage LLMs for analysis, explanation, and automation tasks
- **Engineering teams** seeking to improve productivity through AI-powered development tools - **Technical leads** evaluating how to responsibly adopt LLM technology in their organizations

What You'll Learn

The book progresses from foundational concepts to advanced implementation techniques:

Core Foundations: Understanding LLMs from a developer's perspective, including API integration, cost management, and ethical considerations for responsible AI development.

Practical Techniques: Master essential prompting patterns for code generation, debugging, documentation, and data transformation—with examples across multiple programming languages.

Advanced Methods: Learn sophisticated techniques like chain-of-thought prompting, self-correction strategies, and prompt orchestration for complex workflows.

Real-World Projects: Build three comprehensive applications: - A smart code assistant for automating repetitive development tasks - An LLM-powered ML model explainer for making complex models understandable - A training debugger that analyzes ML experiments and suggests optimizations

Production-Ready Practices: Discover how to build robust, testable, and cost-effective LLM-powered systems that integrate seamlessly with existing development workflows.

Why This Book Is Different

Unlike academic treatments of prompt engineering, this book focuses relentlessly on practical application. Every concept is illustrated with working code, real-world scenarios, and actionable insights that developers can immediately apply. The author's 15 years of software development experience ensures that the advice is grounded in the realities of building and maintaining production systems.

The book emphasizes building a strong foundation in problem-solving and understanding the "why" behind each technique—empowering readers to adapt and innovate as the rapidly evolving field of AI continues to advance.

Whether you're looking to automate routine tasks, build intelligent features, or simply understand how to work effectively with LLMs, this book provides the practical knowledge and hands-on experience needed to succeed in the age of AI-assisted development.

About the Author

Sunny Shivam is a seasoned software developer and data science enthusiast with over 15 years of experience building intelligent systems and scalable applications. He holds an M.Tech in Data Science from BITS Pilani, where he developed deep expertise in machine learning, artificial intelligence, and data-driven solution design.

Throughout his career, Sunny has worked across diverse industries, from early-stage startups to enterprise organizations, consistently bridging the gap between cutting-edge research and practical implementation. His passion lies in making complex technologies accessible to developers, enabling teams to leverage the power of AI and machine learning in real-world applications.

As an early adopter of large language models in development workflows, Sunny has hands-on experience integrating LLMs into production systems, optimizing prompts for various use cases, and building robust AI-powered tools. He has mentored numerous developers in adopting prompt engineering practices and has contributed to open-source projects focused on developer productivity and AI tooling.

Contents

Chapter 1: Introduction to Prompt Engineering: The Developer's New Skillset

What is Prompt Engineering and Why It Matters for Developers

Prompt engineering is the practice of crafting effective inputs (prompts) to large language models (LLMs) to obtain desired outputs. For developers, it represents a paradigm shift in how we interact with software tools. Unlike traditional programming, where we write explicit instructions in code that machines follow precisely, prompt engineering involves communicating with AI systems in natural language to achieve computational goals.

The importance of prompt engineering for developers cannot be overstated:

1. **New Interface to Computing Resources**: Prompts are becoming a universal interface to powerful computational capabilities that would otherwise require complex programming or specialized knowledge.
2. **Productivity Multiplier**: Well-crafted prompts can dramatically accelerate development tasks like code generation, debugging, documentation, and data transformation.
3. **Competitive Advantage**: As LLMs become integrated into development workflows, proficiency in prompt engineering provides a significant edge in the job market.
4. **Bridge Between Technical and Non-Technical Domains**: Prompt engineering enables developers to work more effectively with non-technical stakeholders by transforming natural language requirements into working solutions more directly.

LLMs as Programmable Interfaces

Large Language Models represent a fundamentally new type of programmable interface with unique characteristics:

From APIs to LLMs: A Shift in Abstraction

Traditional APIs require developers to: - Learn specific endpoints and parameters - Format data according to strict schemas - Handle errors through defined error codes - Work within rigid constraints of what the API can do

In contrast, LLMs offer: - Natural language interaction - Flexibility in input formatting - Graceful handling of ambiguity - The ability to perform a vastly wider range of tasks through a single interface

The Programming Model of LLMs

```
 # Traditional API call
response = requests.post(
    "https://api.example.com/translate",
    json={"text": "Hello world", "source": "en", "target": "fr"},
    headers={"Authorization": "Bearer " + API_KEY}
)
result = response.json()["translated_text"]

# vs. LLM-based approach
response = openai.ChatCompletion.create(
    model="gpt-4",
    messages=[
        {"role": "user", "content": "Translate 'Hello world' from English to
French."}
    ]
)
result = response.choices[0].message.content
```

This fundamental shift means that developers need to think differently about:

- **Input Design**: Crafting prompts that clearly communicate intent
- **Output Parsing**: Extracting structured information from natural language responses
- **Error Handling**: Dealing with hallucinations, misunderstandings, and irrelevant outputs
- **Iteration**: Refining prompts based on observed outputs

Ethical Considerations and Responsible Use for Developers

As developers integrating LLMs into applications, we carry significant responsibility for the ethical use of these tools:

Potential Ethical Issues

1. **Bias Amplification**: LLMs may reproduce or amplify social biases present in their training data, which can manifest in generated code, documentation, or user-facing content.
2. **Misinformation Risk**: LLMs can generate plausible but incorrect information, including invalid code, inaccurate explanations, or false claims about technical topics.
3. **Intellectual Property Concerns**: Generated code may raise questions about originality, licensing, and attribution.
4. **Security Vulnerabilities**: LLMs may inadvertently suggest code with security flaws or sensitive information disclosure.

Responsible Development Practices

As developers working with LLMs, we should:

- **Validate Outputs**: Never blindly integrate LLM-generated code without review and testing
- **Set Clear Boundaries**: Be explicit about what types of requests your LLM-powered application should and should not fulfill
- **Provide Attribution**: When appropriate, disclose the use of AI-generated content
- **Design for Transparency**: Make users aware when they're interacting with AI systems
- **Monitor for Bias**: Regularly audit system outputs for signs of harmful bias

Setting Up Your Development Environment

To begin working with LLMs as a developer, you'll need to set up a proper environment:

API Keys and Access

Most commercial LLM providers require authentication via API keys:

1. **OpenAI (GPT-3.5, GPT-4)**:
2. Create an account at platform.openai.com
3. Navigate to API keys section and generate a new key
4. Set up billing (required for API access)
5. **Google (Gemini)**:
6. Get started at ai.google.dev

7. Create a project in Google Cloud Console
8. Enable the Gemini API and generate credentials
9. **Anthropic (Claude)**:
10. Request access at anthropic.com/earlyaccess
11. Once approved, generate API keys from the console

Essential Python Libraries

```
 # Core LLM interaction libraries
pip install openai google-generativeai anthropic

# Utility libraries for LLM applications
pip install langchain llama-index

# For embedding and vector operations
pip install sentence-transformers numpy

# Environment management
pip install python-dotenv
```

Environment Configuration

Best practice is to store API keys securely using environment variables:

```
 # .env file (add to .gitignore)
OPENAI_API_KEY=sk-your-key-here
ANTHROPIC_API_KEY=sk-ant-your-key-here
GOOGLE_API_KEY=your-google-key-here

# In your Python code
import os
from dotenv import load_dotenv

load_dotenv()  # Load environment variables from .env file

openai_api_key = os.getenv("OPENAI_API_KEY")
```

Version Control for Prompts in Development Workflows

As your prompt engineering practice matures, treating prompts as first-class development artifacts becomes essential:

Why Version Control Prompts?

1. **Reproducibility**: Ensuring consistent LLM behavior across development, testing, and production
2. **Collaboration**: Enabling team members to review and improve prompts
3. **Quality Assurance**: Tracking changes to identify when and how prompt modifications affect system behavior
4. **Auditability**: Maintaining records of prompt evolution for compliance or troubleshooting

Practical Approaches to Prompt Version Control

1. Structured Prompt Storage

```
 # prompts/code_generation.py
CODE_GENERATION_PROMPT = """
You are an expert Python developer. Generate a well-documented
function that {task_description}. Follow these guidelines:
- Use type hints
- Include docstrings in Google format
- Follow PEP 8 style conventions
- Handle edge cases appropriately
"""
```

2. Prompt Templates with Variables

```
 # Using a library like Jinja2 for template management
from jinja2 import Template

code_gen_template = Template("""
You are an expert {{ language }} developer with {{ years_experience }}+ years of
experience.
Generate a {{ purpose }} that accomplishes the following:

{{ task_description }}

Requirements:
```

```
{% for req in requirements %}
- {{ req }}
{% endfor %}
""")

# Generate specific prompt
prompt = code_gen_template.render(
    language="Python",
    years_experience=5,
    purpose="data processing function",
    task_description="Parse CSV files and extract specific columns",
    requirements=["Handle malformed data", "Be memory efficient", "Include logging"]
)
```

3. Prompt Versioning Strategies

Consider creating a formal system for prompt versioning:

```
 # prompts/registry.py
PROMPTS = {
    "code_generation": {
        "v1": "Original prompt focused on basic functionality",
        "v2": "Added type hints and docstring requirements",
        "v3": "Expanded to include edge case handling guidance",
        "current": "v3"  # Pointer to current version
    },
    "code_explanation": {
        "v1": "Basic explanation format",
        "v2": "Enhanced with step-by-step breakdown",
        "current": "v2"
    }
}
```

Conclusion

Prompt engineering represents a fundamental new skill for developers in the age of AI. By understanding the principles of effective prompt design, considering ethical implications, setting up a proper development environment, and treating prompts as versioned artifacts, you can harness the power of LLMs to transform your development workflow.

In the next chapter, we'll explore the technical underpinnings of LLMs and develop a deeper understanding of their capabilities and limitations from a developer's perspective.

Exercises

1. Set up your development environment with access to at least one LLM API.
2. Write a prompt that generates a function in your preferred programming language and experiment with variations to see how the output changes.
3. Create a simple versioning scheme for your prompts and implement it in a small project.
4. Reflect on potential ethical considerations for an LLM-powered application you might want to build.

Next Chapter →

Chapter 2: Understanding LLMs: A Developer's Perspective

Brief Overview of LLM Capabilities and Limitations

Large Language Models (LLMs) represent a revolutionary class of AI systems with capabilities that can transform development workflows. To effectively leverage these systems, developers must understand both what they excel at and where they fall short.

Key Capabilities

1. **Natural Language Understanding**: LLMs can parse and interpret complex instructions, requirements, and queries written in natural language.
2. **Code Generation**: Modern LLMs can generate syntactically correct code across dozens of programming languages, from simple functions to complex classes and algorithms.
3. **Code Explanation**: LLMs can analyze existing code and provide explanations of its purpose, logic, and implementation details.
4. **Translation**: LLMs can translate between natural languages and between programming languages.
5. **Data Extraction and Transformation**: LLMs can parse unstructured data and convert it into structured formats.
6. **Problem-Solving**: LLMs can apply reasoning to debug code, optimize algorithms, and solve programming challenges.

Key Limitations

1. **Hallucinations**: LLMs can generate content that appears plausible but is factually incorrect or nonsensical. This includes:
2. Inventing non-existent functions or libraries
3. Creating syntactically correct but logically flawed code
4. Confidently stating incorrect technical information

5. **Context Window Constraints**: LLMs have fixed limits on how much text they can process in a single interaction:

6. GPT-4 (as of this writing): ~32,000 tokens (~24,000 words)
7. Claude 2: ~100,000 tokens
8. Gemini Pro: ~32,000 tokens

These limitations affect your ability to provide context such as large code files or extensive requirements.

1. **Knowledge Cutoffs**: LLMs are trained on data up to a specific date, after which they have no knowledge:
2. GPT-4: Knowledge cutoff in early 2023
3. Other models have similar cutoffs

This affects their knowledge of recent language features, libraries, and best practices.

1. **Inconsistency**: LLMs may provide different solutions to the same prompt when called multiple times, which can introduce unpredictability in development workflows.

2. **Security Risks**: LLMs may inadvertently generate code with security vulnerabilities or suggest unsafe practices.

Popular LLM APIs

Several providers offer LLM access through well-documented APIs. Here's a comparison of the major options:

OpenAI (GPT-3.5, GPT-4)

Strengths: - Industry-leading capabilities for code-related tasks - Comprehensive documentation and community support - Flexible API with good tooling ecosystem

Considerations: - Higher pricing compared to some alternatives - Rate limits for new accounts - Data usage policies that may affect privacy-sensitive applications

API Basics:

```
import openai

# Configure with your API key
openai.api_key = "your-api-key"
```

```
response = openai.ChatCompletion.create(
    model="gpt-4",
    messages=[
        {"role": "system", "content": "You are a helpful assistant that generates
Python code."},
        {"role": "user", "content": "Write a function that finds prime numbers up to
n."}
    ]
)

print(response.choices[0].message.content)
```

Google (Gemini)

Strengths: - Strong performance on technical and scientific content - Integration with Google Cloud ecosystem - Competitive pricing

Considerations: - Newer API with evolving documentation - May require Google Cloud account setup

API Basics:

```
 import google.generativeai as genai

# Configure with your API key
genai.configure(api_key="your-api-key")

model = genai.GenerativeModel('gemini-pro')
response = model.generate_content("Write a function that finds prime numbers up to n
in Python.")

print(response.text)
```

Anthropic (Claude)

Strengths: - Very large context window (100K tokens) - Design focus on safety and reduction of hallucinations - Clear and thoughtful responses

Considerations: - May not match code generation capabilities of GPT-4 for complex tasks - More limited developer ecosystem

API Basics:

```
from anthropic import Anthropic

# Initialize with your API key
anthropic = Anthropic(api_key="your-api-key")

response = anthropic.completions.create(
    prompt=f"Human: Write a function that finds prime numbers up to n in Python.
\n\nAssistant:",
    model="claude-2",
    max_tokens_to_sample=1000
)

print(response.completion)
```

Open Source and Self-Hosted Options

For developers with privacy requirements or cost constraints, several open-source LLMs can be self-hosted:

- **Llama 2 / Llama 3**: Meta's powerful open-source models
- **Mixtral**: Mistral AI's mixture-of-experts model with strong performance
- **Code Llama**: Specialized for code generation tasks
- **Falcon**: Technology Innovation Institute's efficient model

Self-hosting requires substantial hardware resources but enables complete control over data and usage.

Basic API Calls and Handling Responses

Most LLM interactions follow a similar pattern regardless of provider:

1. Construct a prompt (input)
2. Send to the LLM API
3. Receive and process the response
4. Handle errors and edge cases

Core API Interaction Pattern

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

```
# Load API key from .env file
load_dotenv()
openai.api_key = os.getenv("OPENAI_API_KEY")

def get_completion(prompt, model="gpt-3.5-turbo", temperature=0):
    """
    Get a completion from the OpenAI API.

    Args:
        prompt: The prompt to send to the API
        model: The model to use (default: gpt-3.5-turbo)
        temperature: Controls randomness (0=deterministic, 1=creative)

    Returns:
        The completion text
    """
    try:
        response = openai.ChatCompletion.create(
            model=model,
            messages=[{"role": "user", "content": prompt}],
            temperature=temperature
        )
        return response.choices[0].message.content
    except openai.error.OpenAIError as e:
        print(f"OpenAI API error: {e}")
        return None
    except Exception as e:
        print(f"Unexpected error: {e}")
        return None

# Example usage
result = get_completion("Write a Python function to calculate the Fibonacci
sequence.")
print(result)
```

Structured Response Handling

For development workflows, you often need structured data rather than freeform text. You can request specific JSON formats:

```
def get_structured_response(prompt, format_instructions, model="gpt-4"):
    """
    Get a structured response from the LLM in JSON format.
```

```
    Args:
        prompt: The main prompt/question
        format_instructions: JSON format specification
        model: The model to use

    Returns:
        Parsed JSON response or None on error
    """
    import json

    full_prompt = f"""{prompt}

    Return your response as a JSON object with the following structure:
    {format_instructions}

    Ensure your response is valid JSON.
    """

    try:
        response = openai.ChatCompletion.create(
            model=model,
            messages=[{"role": "user", "content": full_prompt}],
            temperature=0  # Use deterministic output for consistent JSON
        )

        response_text = response.choices[0].message.content

        # Extract JSON part (in case there's surrounding text)
        try:
            # Try to parse the entire response
            parsed = json.loads(response_text)
            return parsed
        except json.JSONDecodeError:
            # If that fails, try to find and extract a JSON block
            import re
            json_match = re.search(r'```json\n(.*?)\n```', response_text, re.DOTALL)
            if json_match:
                try:
                    return json.loads(json_match.group(1))
                except json.JSONDecodeError:
                    print("JSON parsing failed even after extraction")
                    return None
            else:
                print("Could not extract JSON from response")
                return None

    except Exception as e:
```

```
        print(f"Error: {e}")
        return None

# Example usage
format_spec = """{
    "function_name": "string",
    "parameters": ["list of parameter names"],
    "complexity": "string (O-notation)",
    "code": "string (the Python code)"
}"""

result = get_structured_response(
    "Create a binary search function",
    format_spec
)

if result:
    print(f"Function: {result['function_name']}")
    print(f"Complexity: {result['complexity']}")
    print("Code:")
    print(result['code'])
```

Troubleshooting Common API Issues

When working with LLM APIs, several common issues may arise:

1. Rate Limiting and Quota Errors

Symptoms: - HTTP 429 "Too Many Requests" errors - Responses indicating rate limit exceeded

Solutions:

```
import time

def resilient_completion(prompt, max_retries=5, backoff_factor=2):
    """Make API calls with exponential backoff for rate limiting"""
    retries = 0
    while retries <= max_retries:
        try:
            return openai.ChatCompletion.create(
                model="gpt-3.5-turbo",
                messages=[{"role": "user", "content": prompt}]
            )
        except openai.error.RateLimitError:
```

```
            wait_time = backoff_factor ** retries
            print(f"Rate limit hit. Waiting {wait_time} seconds...")
            time.sleep(wait_time)
            retries += 1

    raise Exception("Max retries exceeded")
```

2. Context Length Exceeded

Symptoms: - Errors about tokens or input length being too long - Truncated responses

Solutions:

```
def chunked_completion(long_text, question, chunk_size=2000, overlap=200):
    """Process long documents by chunking with overlap"""
    chunks = []
    for i in range(0, len(long_text), chunk_size - overlap):
        chunk = long_text[i:i + chunk_size]
        chunks.append(chunk)

    responses = []
    for i, chunk in enumerate(chunks):
        prompt = f"Document chunk {i+1}/{len(chunks)}:\n\n{chunk}\n\n{question}"
        response = get_completion(prompt)
        responses.append(response)

    # Combine responses
    combined_prompt = f"Based on these analyses of different parts of a document:
\n\n"
    for i, r in enumerate(responses):
        combined_prompt += f"Analysis part {i+1}: {r}\n\n"
    combined_prompt += f"Now provide a complete answer to: {question}"

    return get_completion(combined_prompt)
```

3. Inconsistent Response Formats

Symptoms: - JSON parsing errors - Missing fields in structured outputs - Format inconsistencies

Solutions:

```
def format_enforcing_prompt(prompt, expected_format):
    """Create a prompt that enforces output format"""
    formatted_prompt = f"""
```

```
    {prompt}

    Your response MUST follow this exact format:
    {expected_format}

    If your response doesn't match this format exactly, it will break the system.
    Verify your response carefully before submitting.
    """
    return formatted_prompt
```

4. API Connection Issues

Symptoms: - Timeouts - Connection reset errors - Intermittent failures

Solutions:

```
 import backoff

@backoff.on_exception(backoff.expo,
                      (openai.error.APIConnectionError,
openai.error.ServiceUnavailableError),
                      max_tries=5)
def reliable_completion(prompt, **kwargs):
    """Make API calls with automatic retries for connection issues"""
    return openai.ChatCompletion.create(
        model="gpt-4",
        messages=[{"role": "user", "content": prompt}],
        **kwargs
    )
```

Cost Considerations and Token Management

LLM API costs can rapidly accumulate, especially in production systems. Understanding token usage and implementing cost controls is essential:

Understanding Tokens

Tokens are the fundamental unit of text processing in LLMs: - A token is approximately 4 characters or 0.75 words in English - Punctuation and special characters also count as tokens - Code often uses more tokens than natural language due to special characters and indentation

Token Counting

```
 import tiktoken

def count_tokens(text, model="gpt-4"):
    """Count the number of tokens in a text string"""
    encoding = tiktoken.encoding_for_model(model)
    tokens = encoding.encode(text)
    return len(tokens)

def estimate_cost(prompt_tokens, completion_tokens, model="gpt-4"):
    """Estimate cost in USD for token usage with common models"""
    costs = {
        "gpt-4": {"prompt": 0.03, "completion": 0.06},  # per 1K tokens
        "gpt-3.5-turbo": {"prompt": 0.0015, "completion": 0.002},  # per 1K tokens
        "gpt-4-32k": {"prompt": 0.06, "completion": 0.12},  # per 1K tokens
    }

    if model not in costs:
        raise ValueError(f"Unknown model: {model}")

    prompt_cost = (prompt_tokens / 1000) * costs[model]["prompt"]
    completion_cost = (completion_tokens / 1000) * costs[model]["completion"]

    return prompt_cost + completion_cost

# Example usage
text = "This is a sample prompt that will be sent to the LLM."
token_count = count_tokens(text)
print(f"Token count: {token_count}")
print(f"Estimated cost: ${estimate_cost(token_count, 150):.6f}")
```

Cost Optimization Strategies

1. **Use the Right Model for the Task** `python def choose_optimal_model(task_complexity, input_length, budget_sensitivity): """Select the most cost-effective model for a given task""" if task_complexity == "low" and budget_sensitivity == "high": return "gpt-3.5-turbo" # Cheaper, good for simpler tasks elif input_length > 30000: return "gpt-4-32k" # For very long contexts else: return "gpt-4" # For complex reasoning tasks`

2. **Prompt Optimization** ```python # Before: Verbose prompt verbose_prompt = """ I need you to carefully analyze this code and provide a detailed explanation of what it does, how it works, and identify any potential bugs or performance issues that might exist in the implementation. Please be thorough in your analysis and cover all aspects of the code.

def factorial(n): if n == 0: return 1 return n * factorial(n-1) """

After: Concise prompt concise_prompt = """ Explain this code and identify any issues:

def factorial(n): if n == 0: return 1 return n * factorial(n-1) """

Token reduction of ~50% ```

1. **Response Caching** ```python import hashlib import json import os import pickle

class LLMCache: """Simple cache for LLM responses to avoid redundant API calls"""

```
    def __init__(self, cache_dir=".llm_cache"):
        os.makedirs(cache_dir, exist_ok=True)
        self.cache_dir = cache_dir

    def _get_cache_key(self, prompt, model, temperature):
        """Generate a unique cache key"""
        key_data = {
            "prompt": prompt,
            "model": model,
            "temperature": temperature
        }
        key_str = json.dumps(key_data, sort_keys=True)
        return hashlib.md5(key_str.encode()).hexdigest()

    def get(self, prompt, model, temperature):
        """Retrieve cached response if available"""
        cache_key = self._get_cache_key(prompt, model, temperature)
        cache_path = os.path.join(self.cache_dir, cache_key)

        if os.path.exists(cache_path):
            try:
                with open(cache_path, 'rb') as f:
                    return pickle.load(f)
            except Exception:
                return None
        return None

    def set(self, prompt, model, temperature, response):
        """Cache a response"""
```

```
        cache_key = self._get_cache_key(prompt, model, temperature)
        cache_path = os.path.join(self.cache_dir, cache_key)

        with open(cache_path, 'wb') as f:
            pickle.dump(response, f)
```

Usage cache = LLMCache()

def cached_completion(prompt, model="gpt-3.5-turbo", temperature=0): """Get completion with caching""" # Check cache first cached = cache.get(prompt, model, temperature) if cached: print("Cache hit!") return cached

```
    # If not in cache, call the API
    print("Cache miss, calling API...")
    response = openai.ChatCompletion.create(
        model=model,
        messages=[{"role": "user", "content": prompt}],
        temperature=temperature
    )

    result = response.choices[0].message.content

    # Cache the result
    cache.set(prompt, model, temperature, result)

    return result
```

```

1. **Budget Management** ```python class LLMBudgetManager: """Track and limit LLM API spending"""

   def **init**(self, daily_budget=1.0): # Default $1/day self.daily_budget = daily_budget self.today_spend = 0 self.today_date = datetime.date.today()

   def track_request(self, prompt_tokens, completion_tokens, model): """Track cost of a request and check against budget""" # Reset counter if it's a new day current_date = datetime.date.today() if current_date > self.today_date: self.today_date = current_date self.today_spend = 0

   ```
       # Calculate cost
       cost = estimate_cost(prompt_tokens, completion_tokens, model)
       self.today_spend += cost
   ```
   ```

```
        # Check if over budget
        if self.today_spend > self.daily_budget:
            return False, cost, self.today_spend

        return True, cost, self.today_spend
```

Usage budget_mgr = LLMBudgetManager(daily_budget=5.0) # $5/day limit

def budget_aware_completion(prompt, model="gpt-3.5-turbo"): """Make API calls with budget awareness""" # Count tokens in prompt prompt_tokens = count_tokens(prompt, model)

```
    # Estimate completion tokens (rough estimate)
    est_completion_tokens = prompt_tokens * 1.5

    # Check budget before making call
    allowed, est_cost, total_spend = budget_mgr.track_request(
        prompt_tokens, est_completion_tokens, model
    )

    if not allowed:
        print(f"Request would exceed daily budget. Today's spend: $
{total_spend:.2f}")
        return None

    # Make the API call
    response = openai.ChatCompletion.create(
        model=model,
        messages=[{"role": "user", "content": prompt}]
    )

    # Update with actual tokens used
    actual_completion_tokens = response.usage.completion_tokens
    budget_mgr.track_request(prompt_tokens, actual_completion_tokens, model)

    return response.choices[0].message.content
```

```

## Conclusion

Understanding LLMs from a developer's perspective involves recognizing both their transformative capabilities and inherent limitations. By mastering the APIs, troubleshooting common issues, and implementing cost-effective practices, you can effectively integrate these powerful tools into your development workflow.
```

In the next chapter, we'll explore the art and science of prompt construction, focusing on how to craft effective instructions that yield the best possible results from LLMs.

Exercises

1. Set up API access to at least two different LLM providers and compare their responses to the same coding challenge.
2. Create a utility function that handles token counting, cost estimation, and response caching for LLM API calls.
3. Experiment with different error handling strategies to make your LLM interactions more resilient.
4. Compare the token usage and cost between verbose and concise versions of the same prompt.
5. Implement a simple command-line tool that uses an LLM API to help with a specific development task of your choice.

← Previous Chapter

Next Chapter →

Chapter 3: The Art and Science of Prompt Construction

Anatomy of a Prompt: Instructions, Context, Input Data, Output Format

A well-constructed prompt is the foundation of effective interaction with Large Language Models. Understanding the key components of prompts helps developers craft instructions that yield predictable, high-quality results.

The Four Core Components

1. **Instructions**: Clear directives that tell the model what to do
2. **Context**: Background information that helps the model understand the task
3. **Input Data**: The specific content the model should work with
4. **Output Format**: Specifications for how the response should be structured

Let's examine each component in detail:

1. Instructions

Instructions are explicit directives that guide the model's behavior. They should be: - Clear and specific - Action-oriented - Focused on a single task or a well-defined sequence of tasks

Examples:

```
 Poor instruction: "Help me with this code."
Better instruction: "Debug this Python function that should calculate factorial but
is producing incorrect results."

Poor instruction: "Write some documentation."
Better instruction: "Generate comprehensive JSDoc comments for this JavaScript
utility function."
```

2. Context

Context provides the background information that helps the model understand the scope, purpose, and constraints of the task. Effective context includes: - Relevant background information - Project-specific considerations - Technical requirements or constraints - Target audience information

Examples:

```
 Limited context: "We need API documentation."
Better context: "We're building a REST API for an e-commerce platform using
Express.js. The API will be used by frontend developers who are familiar with React
but have limited backend experience. Documentation should be comprehensive yet
accessible."
```

3. Input Data

Input data is the specific content the model needs to process. This might be: - Code to analyze or modify - Text to transform - Data to structure or extract information from - Problems to solve

Examples:

```
 Vague input: "Fix my sorting function."
Better input:
"Fix the following sorting function that should sort an array of objects by their
'priority' property in descending order:

function sortByPriority(items) {
    return items.sort((a, b) => a.priority - b.priority);
}
"
```

4. Output Format

Output format specifies how the response should be structured. Clear formatting instructions help ensure the model's response is immediately usable. This might include: - Specific structural requirements (JSON, XML, etc.) - Formatting conventions (markdown, HTML, etc.) - Response sections or components - Length constraints

Examples:

````
Unspecified format: "Give me information about common sorting algorithms."
Better format specification: "Compare quick sort, merge sort, and bubble sort.
Format your response as a markdown table with columns for: Algorithm Name, Average
Time Complexity, Space Complexity, Stability, and Best Use Case."
````

Putting It All Together

Here's an example of a well-constructed prompt that incorporates all four components:

````
# INSTRUCTIONS
Review the following Python function that calculates Fibonacci numbers and identify
any performance issues or bugs. Then provide an optimized version.

# CONTEXT
This function will be used in a web application that needs to calculate Fibonacci
numbers up to the 50th number. Performance is critical as this will be called
frequently.

# INPUT DATA
```python
def fibonacci(n):
    if n <= 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
````
````

OUTPUT FORMAT

Provide your response in the following structure: 1. Issues Identified (bullet points) 2. Optimized Solution (code block with comments) 3. Complexity Analysis (time and space)

```
 ## Core Principles: Clarity, Specificity, Conciseness, Role-playing, Constraints

Effective prompts adhere to several key principles that enhance the quality and
reliability of LLM outputs:

### 1. Clarity

Clarity ensures the model understands exactly what is being asked. Unclear prompts
lead to misinterpretations and irrelevant responses.

**Key practices:**
- Use simple, direct language
- Avoid ambiguity
- Define technical terms when necessary
- State the objective up front

**Example:**
```

Unclear prompt: "Make this code better." Clear prompt: "Refactor this Python function to improve its readability and efficiency. Specifically, reduce nested conditionals and optimize the loop structure."

```
 ### 2. Specificity

Specificity narrows the scope of the model's response, leading to more focused and
relevant outputs.

**Key practices:**
- Be explicit about requirements
- Specify the exact problem to solve
- Indicate desired approaches or techniques
- Mention constraints or limitations

**Example:**
```

General prompt: "Write a function to process data." Specific prompt: "Write a Python function that takes a CSV string containing user records (fields: id, name, email, signup_date) and returns a list of dictionaries, with dates converted to datetime objects and emails validated for correct format."

```
### 3. Conciseness

Conciseness focuses on brevity without sacrificing necessary information. While
context is important, excessive verbosity can dilute the core request.

**Key practices:**
- Remove unnecessary details
- Use direct, active language
- Focus on essential requirements
- Structure information logically

**Example:**
```

Verbose prompt: "I'm working on a project where I need to have a function that can take a string and then I need it to count how many times each word appears in the string because I want to analyze text frequency and I'm not sure how to approach this problem efficiently so I need a solution that works well for large texts too."

Concise prompt: "Create an efficient function that counts word frequency in a string, optimized for large texts."

```
### 4. Role-playing

Role-playing instructs the LLM to adopt a specific persona with relevant expertise,
leading to more appropriate responses.

**Key practices:**
- Define a specific role with relevant expertise
- Specify the role's perspective or approach
- Set the relationship between the role and the audience
- Provide context for why this role is appropriate

**Example:**
```

Basic prompt: "Explain how to structure a microservice architecture." Role-based prompt: "As an experienced system architect who has designed microservice systems for large-scale e-commerce platforms, explain the key considerations when structuring a microservice architecture for a startup that expects rapid growth."

```
### 5. Constraints

Constraints provide boundaries that help guide the model's response in terms of
scope, format, or approach.

**Key practices:**
- Set explicit limitations
- Define what should be excluded
- Specify resource constraints
- Indicate priority criteria

**Example:**
```

Unconstrained prompt: "Write a function to validate email addresses." Constrained prompt: "Write a JavaScript function to validate email addresses with these constraints: - No external libraries or dependencies - Must handle international domains - Maximum 30 lines of code - Prioritize readability over perfect validation"

```
## Basic Techniques: Zero-shot, Few-shot, Instruction-based

Different prompting techniques provide varied approaches to guiding LLM behavior,
each with specific advantages for different situations:

### 1. Zero-shot Prompting

Zero-shot prompting involves asking the model to perform a task without any
examples. This relies on the model's pre-trained knowledge.

**Best for:**
- Simple, common tasks
- When examples might bias the output
- Tasks the model is likely familiar with

**Example:**
```

Create a function in Python that validates whether a string is a valid IPv4 address.

```
**Advantages:**
- Simple and direct
- Requires minimal prompt engineering
- Tests the model's inherent capabilities

**Disadvantages:**
```

```
- May produce inconsistent results
- Less control over output format
- May fail for complex or uncommon tasks

### 2. Few-shot Prompting

Few-shot prompting provides one or more examples of the desired input-output pattern
before asking the model to perform a similar task.

**Best for:**
- Tasks with specific output formats
- Establishing patterns the model should follow
- Guiding the model toward a particular approach

**Example:**
```

Convert the following function signatures from JavaScript to TypeScript:

Example 1: JavaScript: function calculateTotal(prices, discount) { ... } TypeScript: function calculateTotal(prices: number[], discount: number): number { ... }

Example 2: JavaScript: function processUser(user, options) { ... } TypeScript: function processUser(user: UserType, options: ProcessOptions): UserResult { ... }

Now convert this one: JavaScript: function sortProducts(products, criteria, ascending) { ... }

```
 **Advantages:**
- Provides clear guidance through examples
- Reduces ambiguity
- Works well for pattern-following tasks

**Disadvantages:**
- Takes up more context window space
- May limit creativity
- Can bias the model toward specific approaches

### 3. Instruction-based Prompting

Instruction-based prompting provides detailed, step-by-step directions on how the
model should approach a task, often with explicit formatting requirements.

**Best for:**
- Complex multi-step tasks
- Tasks requiring specific methodologies
- Outputs needing standardized formatting
```

```
**Example:**
```

Analyze the security vulnerabilities in the following Node.js code:

```
 const express = require('express');
const app = express();

app.get('/user/:id', (req, res) => {
  const userId = req.params.id;
  const query = `SELECT * FROM users WHERE id = ${userId}`;
  db.execute(query).then(result => {
    res.json(result);
  });
});
```

Follow these steps in your analysis: 1. Identify each vulnerability and its type (e.g., SQL injection, XSS) 2. Explain why it's a vulnerability and potential exploit scenarios 3. Rate the severity (Low, Medium, High, Critical) 4. Provide a secure code alternative for each issue found 5. Suggest additional security best practices relevant to this code

Format your response as a structured report with clear headings for each vulnerability.

```
 **Advantages:**
- Provides detailed guidance
- Ensures comprehensive outputs
- Creates structured, predictable responses

**Disadvantages:**
- Can be lengthy and consume tokens
- May over-constrain the model
- Requires careful crafting to avoid confusion

### Hybrid Approaches

Often, the most effective prompts combine elements from multiple techniques:
```

INSTRUCTION

Implement a memory-efficient data structure for a least-recently-used (LRU) cache in Python.

EXAMPLES

Here's an example of how a similar data structure (Stack) might be implemented:

```
class Stack:
    def __init__(self, capacity):
        self.capacity = capacity
        self.items = []

    def push(self, item):
        if len(self.items) >= self.capacity:
            raise OverflowError("Stack is full")
        self.items.append(item)

    def pop(self):
        if not self.items:
            raise IndexError("Pop from empty stack")
        return self.items.pop()
```

REQUIREMENTS

Your LRU cache implementation should: 1. Have O(1) time complexity for lookups, insertions, and deletions 2. Support a configurable maximum size 3. Automatically remove least recently used items when full 4. Include methods: get(key), put(key, value), and remove(key) 5. Include proper docstrings and type hints

OUTPUT FORMAT

Provide your solution as a complete Python class with inline comments explaining key design decisions.

```
 ## Testing and Evaluating Prompt Effectiveness

The effectiveness of a prompt can be assessed across several dimensions:

### 1. Response Relevance

How well does the output address the actual request?

**Evaluation method:**
```python
def evaluate_relevance(prompt, response, criteria):
    """
    Evaluate the relevance of an LLM response against specific criteria

    Args:
        prompt: The original prompt
        response: The LLM's response
        criteria: List of required topics/elements

    Returns:
        Score and missing elements
    """
    score = 0
    missing = []

    for criterion in criteria:
        if criterion.lower() in response.lower():
            score += 1
        else:
            missing.append(criterion)

    relevance_score = score / len(criteria)
    return relevance_score, missing

# Example usage
prompt = "Explain the differences between REST and GraphQL APIs."
response = "REST APIs use standard HTTP methods and typically return fixed data
```
```

```
structures. They may require multiple requests to fetch related data. GraphQL allows
clients to specify exactly what data they need in a single request, reducing over-
fetching."
criteria = ["HTTP methods", "endpoint structure", "data fetching", "versioning",
"caching"]

score, missing = evaluate_relevance(prompt, response, criteria)
print(f"Relevance score: {score:.2f}")
print(f"Missing elements: {missing}")
```

2. Output Format Compliance

Does the response follow the requested format?

Evaluation method:

```
 import re
import json

def evaluate_format_compliance(response, format_type):
    """
    Check if response complies with requested format

    Args:
        response: The LLM response text
        format_type: Type of format expected ('json', 'markdown_table',
'bullet_list', etc.)

    Returns:
        Boolean indicating compliance and reason if non-compliant
    """
    if format_type == 'json':
        try:
            # Check if there's a code block with JSON
            json_match = re.search(r'```(?:json)?\s*(\{.*?\})\s*```', response,
re.DOTALL)
            if json_match:
                json_str = json_match.group(1)
                json.loads(json_str)  # Test if valid JSON
                return True, "Valid JSON found in code block"

            # Try to find a JSON object even without code block
            json_pattern = re.search(r'(\{[^{]*".*"[^}]*\})', response, re.DOTALL)
            if json_pattern:
                json_str = json_pattern.group(1)
```

```
                json.loads(json_str)  # Test if valid JSON
                return True, "Valid JSON found"

            return False, "No valid JSON found in response"
        except json.JSONDecodeError:
            return False, "JSON parsing failed"

    elif format_type == 'markdown_table':
        # Check for markdown table pattern
        has_table = bool(re.search(r'\|[\s\w]+\|[\s\w]+\|', response) and
                         re.search(r'\|[-:]+\|[-:]+\|', response))
        return has_table, "Markdown table not found" if not has_table else "Markdown
table found"

    elif format_type == 'bullet_list':
        # Check for bulleted list pattern
        has_bullets = bool(re.findall(r'^\s*[-*]\s+\w+', response, re.MULTILINE))
        return has_bullets, "Bullet list not found" if not has_bullets else "Bullet
list found"

    return False, "Format type not supported for evaluation"

# Example usage
response = """
Here's the data you requested:

```json
{
    "name": "API Comparison",
    "technologies": ["REST", "GraphQL", "gRPC"],
    "metrics": {
        "performance": [85, 92, 97],
        "learning_curve": [75, 68, 45]
    }
}
```

"""
```

is_compliant, reason = evaluate_format_compliance(response, 'json') print(f"Format compliant: {is_compliant}, Reason: {reason}")

```
### 3. Factual Accuracy

Does the response contain correct information?

**Evaluation method:**
```

```python
def evaluate_factual_accuracy(response, fact_checks):
    """
    Check response for factual accuracy against known truth

    Args:
        response: The LLM response
        fact_checks: Dictionary of facts to check {fact_description: truth_value}

    Returns:
        Accuracy score and incorrect facts
    """
    correct = 0
    incorrect = []

    for fact, truth in fact_checks.items():
        # Simple presence check - could be enhanced with NLP techniques
        fact_present = fact.lower() in response.lower()

        if fact_present == truth:
            correct += 1
        else:
            incorrect.append(fact)

    accuracy = correct / len(fact_checks) if fact_checks else 0
    return accuracy, incorrect

# Example usage
response = "JavaScript is a dynamically typed language that supports first-class
functions and prototypal inheritance. It was created in 1995 by Brendan Eich."

facts = {
    "JavaScript is dynamically typed": True,
    "JavaScript uses classical inheritance": False,
    "JavaScript was created by Brendan Eich": True,
    "JavaScript was created in 1990": False,
    "JavaScript supports first-class functions": True
}

accuracy, incorrect = evaluate_factual_accuracy(response, facts)
print(f"Factual accuracy: {accuracy:.2f}")
print(f"Incorrect facts: {incorrect}")
```

4. A/B Testing Prompts

Systematically compare different prompt variations to find the most effective approach.

Evaluation method:

```
import random

class PromptABTester:
    """A simple tool for A/B testing different prompt formulations"""

    def __init__(self, llm_function, evaluation_function):
        """
        Args:
            llm_function: Function that sends prompt to LLM and returns response
            evaluation_function: Function that scores response quality (0-1)
        """
        self.llm_function = llm_function
        self.evaluation_function = evaluation_function
        self.results = {}

    def test_prompt_variations(self, prompt_variations, trials=3):
        """
        Test multiple prompt variations with repeated trials

        Args:
            prompt_variations: Dict of {variation_name: prompt_text}
            trials: Number of times to test each variation

        Returns:
            DataFrame with results
        """
        import pandas as pd

        all_results = []

        for name, prompt in prompt_variations.items():
            self.results[name] = []

            for i in range(trials):
                response = self.llm_function(prompt)
                score = self.evaluation_function(response)

                self.results[name].append(score)
                all_results.append({
```

```
                    'variation': name,
                    'trial': i+1,
                    'score': score,
                    'prompt': prompt,
                    'response': response
                })

        results_df = pd.DataFrame(all_results)

        # Calculate aggregate statistics
        summary = results_df.groupby('variation')['score'].agg(['mean', 'std',
'min', 'max'])

        return results_df, summary

    def get_best_prompt(self):
        """Return the prompt variation with highest average score"""
        avg_scores = {name: sum(scores)/len(scores) for name, scores in
self.results.items()}
        best_variation = max(avg_scores, key=avg_scores.get)
        return best_variation, avg_scores[best_variation]

# Example usage
def mock_llm(prompt):
    """Mock LLM function for demonstration"""
    # This would be replaced by actual LLM API call
    responses = [
        "This is a detailed response that covers all requirements.",
        "This is a partial response that misses some key points.",
        "This response is thorough and well-structured."
    ]
    return random.choice(responses)

def mock_evaluator(response):
    """Mock evaluation function for demonstration"""
    # This would be replaced by actual evaluation logic
    if "detailed" in response or "thorough" in response:
        return 0.9
    return 0.6

# Create test variations
prompt_variations = {
    "basic": "Explain how virtual memory works in operating systems.",
    "detailed": "Explain how virtual memory works in operating systems. Include
paging, segmentation, and address translation.",
    "role_based": "As an OS kernel engineer, explain how virtual memory works to a
junior developer."
```

```
}

# Run the test
tester = PromptABTester(mock_llm, mock_evaluator)
results, summary = tester.test_prompt_variations(prompt_variations, trials=5)
best_prompt, best_score = tester.get_best_prompt()

print(f"Best prompt variation: {best_prompt} (Score: {best_score:.2f})")
print("\nSummary statistics:")
print(summary)
```

Conclusion

Prompt construction is both an art and a science. By understanding the anatomy of effective prompts, applying core principles, and leveraging appropriate prompting techniques, developers can achieve consistently high-quality results from LLMs. Regular testing and evaluation help refine prompts over time, leading to increasingly reliable and useful outputs.

In the next chapter, we'll explore essential prompting patterns specifically tailored for common developer tasks, from code generation to documentation and debugging.

Exercises

1. Take a simple prompt you've used before and improve it by explicitly incorporating the four components: instructions, context, input data, and output format.
2. Compare zero-shot, few-shot, and instruction-based approaches on the same task (e.g., generating a function to validate email addresses). Which performed best and why?
3. Create an A/B testing framework to evaluate prompt effectiveness for a specific use case (e.g., code explanation, bug finding).
4. Design a prompt that demonstrates all five core principles: clarity, specificity, conciseness, role-playing, and constraints.
5. Create a systematic evaluation method for one type of LLM task you commonly use (e.g., code generation, text summarization) with at least three different evaluation criteria.

← Previous Chapter

Next Chapter →

Chapter 4: Essential Prompting Patterns for Developers

In this chapter, we'll explore practical prompting patterns that developers can immediately apply to common tasks. These patterns serve as templates that you can adapt to your specific needs, saving time and improving consistency in your interactions with LLMs.

Code Generation: Generating Functions, Classes, Scripts

Function Generation Pattern

Use this pattern to generate well-structured, documented functions for specific programming tasks.

```
Create a [language] function named [function_name] that [purpose].

Input parameters:
- [param1_name] ([type]): [description]
- [param2_name] ([type]): [description]

Requirements:
- [requirement1]
- [requirement2]

Include appropriate error handling, type hints, and documentation.
```

Example - Python Data Processing Function:

```
Create a Python function named 'parse_log_entries' that extracts structured data
from application log files.

Input parameters:
- log_path (str): Path to the log file
- error_levels (list, optional): Specific error levels to filter for (e.g.,
["ERROR", "WARNING"])
- start_date (datetime, optional): Only process entries after this date

Requirements:
- Return a list of dictionaries with keys: timestamp, level, message, service
```

```
- Handle malformed log lines gracefully
- Support both plain text and gzip compressed logs
- Add type hints and comprehensive docstrings

Include appropriate error handling and performance optimization for large files.
```

Class Generation Pattern

Use this pattern to generate complete classes with properties, methods, and appropriate design patterns.

```
 Design a [language] class named [ClassName] that [purpose].

Properties:
- [property1_name] ([type]): [description]
- [property2_name] ([type]): [description]

Methods:
- [method1_name]([params]): [description]
- [method2_name]([params]): [description]

Requirements:
- [requirement1]
- [requirement2]

Additional context:
[any relevant information about usage, environment, etc.]
```

Example - TypeScript Component Class:

```
 Design a TypeScript class named 'DataTable' that implements a reusable, sortable
table component for a React application.

Properties:
- data (Array<Record<string, any>>): Source data to display
- columns (ColumnConfig[]): Column configuration including headers, field mappings
and formatting
- sortState (SortConfig): Current sort column and direction
- pageSize (number): Number of records per page
- currentPage (number): Current page index

Methods:
- render(): JSX.Element - Render the table with current configuration
- sort(columnId: string, direction: 'asc'|'desc'): void - Sort table data
```

```
- nextPage(): void - Navigate to next page
- previousPage(): void - Navigate to previous page
- onRowClick(handler: (row: Record<string, any>) => void): void - Set row click
handler

Requirements:
- Implement pagination with customizable page size
- Support client-side sorting for all common data types
- Allow custom cell renderers for complex data
- Follow React best practices for performance optimization
- Include proper TypeScript interfaces and types

Additional context:
The component will be used in a dashboard application that displays various data
views to users with different permission levels.
```

Script Generation Pattern

Use this pattern to generate complete scripts for automation tasks, data processing, or system operations.

```
 Create a [language] script that [purpose].

Inputs:
- [input1]: [description]
- [input2]: [description]

Expected output:
[description of what the script should produce]

Requirements:
- [requirement1]
- [requirement2]

Environment context:
[relevant environment information]
```

Example - Python ETL Script:

```
 Create a Python script that performs ETL (Extract, Transform, Load) operations on
website analytics data.

Inputs:
- analytics.csv: Daily website traffic data (columns: date, page_path, visitors,
```

```
bounce_rate, avg_time_on_page)
- product_mapping.json: JSON file mapping page paths to product categories

Expected output:
- A PostgreSQL database table with aggregated metrics by product category and date
- A summary report in CSV format showing week-over-week changes

Requirements:
- Handle missing or malformed data gracefully
- Implement logging for the ETL process
- Support incremental loads (only process new data)
- Include command-line arguments for configuration
- Optimize for memory efficiency with large input files

Environment context:
- Python 3.9+
- Will run as a scheduled task in Linux environment
- PostgreSQL 13 database
```

Code Explanation & Documentation

Code Comprehension Pattern

Use this pattern when you need to understand unfamiliar or complex code.

````
 Explain the following [language] code, focusing on:
1. Overall purpose
2. Key algorithms or data structures used
3. Control flow and execution path
4. Any potential edge cases or bugs
5. Performance characteristics

```[code block]```
````

Example:

````
 Explain the following JavaScript code, focusing on:
1. Overall purpose
2. Key algorithms or data structures used
3. Control flow and execution path
4. Any potential edge cases or bugs
5. Performance characteristics
````
````

```javascript
function findDuplicateTransactions(transactions) {
  const sorted = [...transactions].sort((a, b) => {
    return new Date(a.time) - new Date(b.time);
  });

  const potential = {};
  sorted.forEach(t => {
    const key = `${t.sourceAccount}_${t.targetAccount}_${t.category}_${t.amount}`;
    if (!potential[key]) potential[key] = [];
    potential[key].push(t);
  });

  const duplicates = [];
  Object.values(potential).forEach(group => {
    if (group.length <= 1) return;

    const result = [];
    for (let i = 0; i < group.length; i++) {
      if (result.length === 0) {
        result.push(group[i]);
        continue;
      }

      const last = result[result.length - 1];
      const current = group[i];
      const timeDiff = Math.abs(new Date(current.time) - new Date(last.time));
      const minutesDiff = timeDiff / (1000 * 60);

      if (minutesDiff <= 5) {
        result.push(current);
      } else if (result.length > 1) {
        duplicates.push([...result]);
        result.length = 0;
        result.push(current);
      } else {
        result.length = 0;
        result.push(current);
      }
    }

    if (result.length > 1) {
      duplicates.push(result);
    }
  });
```

```
  return duplicates;
}
```

```
### Documentation Generation Pattern

Use this pattern to generate comprehensive documentation for existing code.
```

Generate [format] documentation for the following [language] [code_type]. Include: - Purpose and functionality - Parameter descriptions - Return value details - Usage examples - Edge cases and exceptions

`[code block]`

```
**Example:**
```

Generate JSDoc documentation for the following JavaScript function. Include: - Purpose and functionality - Parameter descriptions - Return value details - Usage examples - Edge cases and exceptions

```
function throttle(fn, delay) {
  let lastCall = 0;
  let timeoutId = null;

  return function(...args) {
    const now = Date.now();
    const remaining = delay - (now - lastCall);

    if (remaining <= 0) {
      if (timeoutId) {
        clearTimeout(timeoutId);
        timeoutId = null;
      }

      lastCall = now;
      return fn.apply(this, args);
    } else if (!timeoutId) {
      timeoutId = setTimeout(() => {
        lastCall = Date.now();
        timeoutId = null;
        fn.apply(this, args);
      }, remaining);
    }
```

```
  };
}
```

```
## Debugging & Error Resolution

### Error Diagnosis Pattern

Use this pattern when you encounter error messages and need help troubleshooting.
```

I'm getting the following error when running my [language] code:

```
[error message]
```

Here's the relevant code:

`[code block]`

Please: 1. Explain what's causing this error 2. Suggest a solution with code examples 3. Explain how to prevent similar errors in the future

```
**Example:**
```

I'm getting the following error when running my Python code:

```
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Here's the relevant code:

```
def calculate_total_cost(items):
    total = 0
    for item in items:
        total = total + item['price']
    return total

inventory = [
    {'id': 1, 'name': 'Widget A', 'price': 10},
    {'id': 2, 'name': 'Widget B', 'price': '15'},
    {'id': 3, 'name': 'Widget C', 'price': 20},
]

print(calculate_total_cost(inventory))
```

Please: 1. Explain what's causing this error 2. Suggest a solution with code examples 3. Explain how to prevent similar errors in the future

```
### Code Review Pattern

Use this pattern to identify potential issues before they cause runtime errors.
```

Review the following [language] code for: 1. Bugs or logic errors 2. Performance issues 3. Security vulnerabilities 4. Style or best practice violations 5. Edge cases that aren't handled

`[code block]`

Suggest improvements with code examples.

```
**Example:**
```

Review the following Python code for: 1. Bugs or logic errors 2. Performance issues 3. Security vulnerabilities 4. Style or best practice violations 5. Edge cases that aren't handled

```
def authenticate_user():
    username = request.form.get('username')
    password = request.form.get('password')

    query = "SELECT * FROM users WHERE username='{}' AND password='{}'".format(
        username, password
    )

    result = db.execute(query)
    user = result.fetchone()

    if user:
        session['user_id'] = user[0]
        session['is_admin'] = user[3]
        return redirect('/dashboard')
    else:
        attempts = session.get('login_attempts', 0) + 1
        session['login_attempts'] = attempts

        if attempts > 3:
            time.sleep(3)  # Add delay after multiple failures

        return render_template('login.html', error="Invalid credentials")
```

Suggest improvements with code examples.

```
## Text Transformation

### Text Summarization Pattern

Use this pattern to condense documentation, comments, or other text while preserving
key information.
```

Summarize the following [text_type] in [target_length] while preserving the key technical details:

[text to summarize]

```
**Example:**
```

Summarize the following API documentation in 3-4 paragraphs while preserving the key technical details:

The User Authentication API provides endpoints for registering, authenticating, and managing user accounts. It supports both traditional username/password authentication as well as OAuth2 integrations with popular providers.

The main endpoint /api/v1/auth/login accepts POST requests with either username/password combinations or OAuth tokens. For username/password authentication, the request body must include "username" and "password" fields in JSON format. For OAuth, include "provider" and "access_token" fields. The API returns a JWT token with a configurable expiration (default: 24 hours).

Token validation can be performed against the /api/v1/auth/validate endpoint, which accepts GET requests with the Authorization header set to "Bearer {token}". This endpoint returns user details if the token is valid or a 401 status if invalid or expired.

Account registration is handled through the /api/v1/auth/register endpoint, accepting POST requests with required fields "username", "email", and "password". Additional optional fields include "full_name", "phone", and "preferences" (as a JSON object). Password requirements can be configured but default to minimum 8 characters with at least one number, one uppercase letter, and one special character.

Password reset functionality is provided via two endpoints: /api/v1/auth/request-reset (POST with "email" field) sends a time-limited reset token to the user's email, and /api/v1/auth/reset-password (POST with "token" and "new_password" fields) performs the actual password change.

Rate limiting is applied to all authentication endpoints, with default limits of 5 attempts per minute for login and 3 attempts per hour for password reset requests from the same IP address. These limits are configurable through the server's environment variables RATE_LIMIT_LOGIN and RATE_LIMIT_RESET.

All authentication attempts, successful or failed, are logged with timestamp, IP address, and user agent information. These logs can be accessed through the admin dashboard or directly from the database's auth_logs table.

```
 ### Text Translation Pattern

Use this pattern to translate between technical terminology, languages, or convert
between technologies.
```

Translate the following [source_format] to [target_format], maintaining all functionality:

`[source code or text]`

```
 **Example:**
```

Translate the following JavaScript React component to TypeScript, maintaining all functionality:

```
 import React, { useState, useEffect } from 'react';
import axios from 'axios';

function UserDashboard({ userId }) {
  const [userData, setUserData] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    async function fetchUserData() {
      try {
        const response = await axios.get(`/api/users/${userId}`);
        setUserData(response.data);
        setLoading(false);
      } catch (err) {
        setError('Failed to load user data');
        setLoading(false);
      }
    }

    fetchUserData();
```

```
  }, [userId]);

  function handleRefresh() {
    setLoading(true);
    fetchUserData();
  }

  if (loading) return <div className="loading">Loading...</div>;
  if (error) return <div className="error">{error}</div>;

  return (
    <div className="dashboard">
      <h1>Welcome, {userData.name}</h1>
      <div className="stats">
        <div className="stat-item">
          <span className="stat-label">Projects</span>
          <span className="stat-value">{userData.projects.length}</span>
        </div>
        <div className="stat-item">
          <span className="stat-label">Tasks</span>
          <span className="stat-value">{userData.tasks.filter(t => !
t.completed).length}</span>
        </div>
      </div>
      <button onClick={handleRefresh}>Refresh Data</button>
    </div>
  );
}

export default UserDashboard;
```

```
### Format Conversion Pattern

Use this pattern to transform between data formats (JSON, XML, CSV, etc.) or to
structure text in specific formats.
```

Convert the following [source_format] to [target_format]:

`[source data]`

Requirements: - [requirement1] - [requirement2]

```
**Example:**
```

Convert the following JSON data to a CSV format:

```
{
  "employees": [
    {
      "id": 101,
      "name": "John Smith",
      "department": "Engineering",
      "title": "Senior Developer",
      "skills": ["Python", "React", "MongoDB"],
      "projects": [
        {"id": "P-100", "name": "API Gateway"},
        {"id": "P-201", "name": "Data Pipeline"}
      ]
    },
    {
      "id": 102,
      "name": "Alice Johnson",
      "department": "Product",
      "title": "Product Manager",
      "skills": ["Agile", "Roadmapping", "User Research"],
      "projects": [
        {"id": "P-100", "name": "API Gateway"},
        {"id": "P-105", "name": "Mobile App"}
      ]
    }
  ]
}
```

Requirements: - Include headers in the first row - Format skills as comma-separated values within a single field - Include project IDs and names in separate columns - Create one row per employee-project combination

```
## Data Extraction: Pulling Structured Data from Unstructured Text

### Entity Extraction Pattern

Use this pattern to identify and extract specific entities from text.
```

Extract the following entities from this [text_type]: - [entity_type1] - [entity_type2] - [entity_type3]

Return the results as [format].

[text]

```
**Example:**
```

Extract the following entities from this error log: - Error codes - Timestamps - File paths - IP addresses

Return the results as a JSON object with arrays of each entity type.

[2023-05-15T09:23:45.123Z] ERROR [server.js:125] Failed to authenticate user from 192.168.1.105 - Error code AUTH-401 [2023-05-15T09:25:12.456Z] WARN [auth/middleware.js:85] Rate limit exceeded for IP 192.168.1.105 [2023-05-15T09:30:22.789Z] ERROR [database/connection.js:209] Failed to connect to database after 5 retries - Error code DB-503 [2023-05-15T09:31:01.234Z] INFO [server.js:50] Server restarting with updated configuration from /etc/app/config.json [2023-05-15T09:32:45.678Z] ERROR [api/users.js:75] Invalid request parameters from 192.168.1.210 - Error code API-400

```
### Tabular Data Extraction Pattern

Use this pattern to convert text descriptions or semi-structured information into structured tables.
```

Extract the following information from the text into a [table_format] with columns [column1, column2, ...]:

[text]

```
**Example:**
```

Extract the following information from the text into a markdown table with columns Method, Endpoint, Required Parameters, and Description:

Our REST API provides the following endpoints for managing user profiles:

GET /api/users - Returns a list of all users. Supports optional query parameters 'limit' (default 20) and 'offset' (default 0) for pagination.

GET /api/users/{id} - Returns details for a specific user. The 'id' parameter is required and must be a valid user identifier.

POST /api/users - Creates a new user. Requires 'email', 'username', and 'password' in the request body. Optional fields include 'fullName', 'role', and 'preferences'.

PUT /api/users/{id} - Updates an existing user. The 'id' parameter is required. At least one update field must be provided in the request body.

DELETE /api/users/{id} - Removes a user. The 'id' parameter is required. This operation cannot be undone.

PATCH /api/users/{id}/status - Updates only the user's status. Requires 'id' parameter and 'status' field in the request body with value 'active' or 'inactive'.

```
### Parsing Unstructured Data Pattern

Use this pattern to extract structured information from free-form text like emails,
documents, or requirements.
```

Parse the following [text_type] and extract: - [information1] - [information2] - [information3]

Format the output as [format]:

[text]

```
**Example:**
```

Parse the following customer support email and extract: - Product name and version - Issue category - Steps to reproduce - Customer contact information - Priority level (if mentioned)

Format the output as JSON:

Subject: Urgent: Dashboard crashes when filtering by date range in Analytics Pro 4.2

Hello Support Team,

I'm experiencing a critical issue with Analytics Pro version 4.2.1 on Windows 10. Whenever I try to filter dashboard data using a date range, the application completely crashes and I have to restart it.

Steps to reproduce: 1. Open the main dashboard 2. Click on "Date Filter" in the top right 3. Select "Custom Range" from the dropdown 4. Choose any start and end dates 5. Click "Apply Filter"

After clicking "Apply Filter," the screen freezes for about 5 seconds, then the application crashes with no error message.

This is blocking our monthly reporting process which we need to complete by end of day tomorrow, so I'd consider this a high priority issue.

Please contact me at john.smith@example.com or 555-123-4567 if you need more information.

Thanks, John Smith Senior Data Analyst Acme Corporation

```
## Examples in Multiple Programming Languages

### Python - Function Generation
```

Create a Python function named 'parse_log_file' that extracts and analyzes error messages from a log file.

Input parameters: - file_path (str): Path to the log file - error_types (list, optional): List of error types to focus on (e.g., ['ERROR', 'CRITICAL']) - start_date (str, optional): Only parse logs after this date (format: 'YYYY-MM-DD')

Requirements: - Return a dictionary with error types as keys and lists of error messages as values - Include timestamps and context information with each error - Handle compressed (.gz) log files automatically - Use generators for memory efficiency with large files - Include proper type hints and docstrings

```
### JavaScript - Debugging
```

I'm getting the following error when running my JavaScript React application:

```
TypeError: Cannot read properties of undefined (reading 'map')
```

Here's the relevant code:

```
function ProductList({ categoryId }) {
  const [products, setProducts] = useState(null);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    async function fetchProducts() {
      try {
        const response = await api.getProductsByCategory(categoryId);
        setProducts(response.data);
      } catch (err) {
        console.error('Failed to fetch products:', err);
      } finally {
        setLoading(false);
      }
    }
```

```
    fetchProducts();
  }, [categoryId]);

  if (loading) return <LoadingSpinner />;

  return (
    <div className="product-grid">
      {products.map(product => (
        <ProductCard key={product.id} product={product} />
      ))}
    </div>
  );
}
```

Please: 1. Explain what's causing this error 2. Suggest a solution with code examples 3. Explain how to prevent similar errors in the future

```
### Java - Code Explanation
```

Explain the following Java code, focusing on: 1. Overall purpose 2. Key algorithms or design patterns used 3. Thread safety considerations 4. Potential performance bottlenecks

```
public class ConnectionPool {
    private static ConnectionPool instance;
    private final List<Connection> availableConnections = new ArrayList<>();
    private final List<Connection> usedConnections = new ArrayList<>();
    private final int MAX_CONNECTIONS;

    private ConnectionPool(String url, String user, String password, int
maxConnections) {
        this.MAX_CONNECTIONS = maxConnections;
        try {
            for (int i = 0; i < maxConnections; i++) {
                availableConnections.add(
                    DriverManager.getConnection(url, user, password)
                );
            }
        } catch (SQLException e) {
            logger.severe("Failed to initialize connection pool: " +
e.getMessage());
        }
    }

    public static synchronized ConnectionPool getInstance(String url, String user,
                                                          String password, int
```

```
maxConnections) {
        if (instance == null) {
            instance = new ConnectionPool(url, user, password, maxConnections);
        }
        return instance;
    }

    public synchronized Connection getConnection() throws SQLException {
        if (availableConnections.isEmpty()) {
            if (usedConnections.size() < MAX_CONNECTIONS) {
                String url = usedConnections.get(0).getMetaData().getURL();
                availableConnections.add(DriverManager.getConnection(url, "", ""));
            } else {
                throw new SQLException("Connection limit reached, no available
connections!");
            }
        }

        Connection connection =
availableConnections.remove(availableConnections.size() - 1);
        usedConnections.add(connection);
        return connection;
    }

    public synchronized void releaseConnection(Connection connection) {
        usedConnections.remove(connection);
        availableConnections.add(connection);
    }
}
```

```
### C# - Class Generation
```

Design a C# class named 'FileProcessor' that handles batch processing of documents in different formats.

Properties: - ProcessedCount (int): Number of files successfully processed - FailedCount (int): Number of files that failed processing - ProcessingOptions (ProcessingOptions): Configuration settings for processing - OnProgressUpdate (event): Event that fires when progress changes

Methods: - ProcessDirectory(string path, bool recursive): Process all compatible files in a directory - ProcessFile(string filePath): Process a single file - GetSupportedFormats(): List - Return list of supported file formats - CancelProcessing(): Cancel any ongoing processing operation - GenerateReport(): ProcessingSummary - Create summary of processing operations

Requirements: - Support PDF, DOCX, and TXT files with different processing strategies - Implement proper error handling and logging - Use async/await for file operations - Follow C# naming conventions and best practices - Make the class extensible for adding new file format handlers ```

Conclusion

These essential prompting patterns form the foundation of effective LLM use in development workflows. By adapting these patterns to your specific needs, you can quickly generate reliable, high-quality outputs for a wide range of development tasks across different programming languages and environments.

In the next chapter, we'll explore advanced prompting techniques that provide even greater control over LLM responses, including Chain-of-Thought reasoning, self-correction strategies, and parameter tuning.

Exercises

1. Create a function generation prompt for a programming language of your choice that implements a data validation utility.
2. Take a complex function or class from your codebase and use the code explanation pattern to generate documentation for it.
3. Find a bug in your code and use the error diagnosis pattern to get help fixing it.
4. Use the format conversion pattern to transform a dataset between two different formats (e.g., JSON to CSV or XML to JSON).
5. Create a prompt that extracts structured information from unstructured API documentation or release notes.

← Previous Chapter

Next Chapter →

Chapter 5: Advanced Prompting Techniques for Enhanced Control

As you grow more experienced with prompt engineering, you'll want to move beyond basic prompting patterns to achieve more precise and reliable results. This chapter explores advanced techniques that give you greater control over LLM outputs, especially for complex development tasks.

Chain-of-Thought (CoT): Guiding LLMs through Multi-Step Reasoning

Chain-of-Thought prompting encourages LLMs to break down complex problems into logical steps before reaching a conclusion. This technique is particularly valuable for debugging, algorithm design, and other tasks that require structured reasoning.

The CoT Principle

The core idea behind CoT is to instruct the LLM to: 1. Decompose a complex problem into distinct steps 2. Reason through each step explicitly 3. Build toward the final solution incrementally

This mirrors how expert programmers approach difficult problems, leading to more accurate and explainable results.

Basic CoT Template

```
 I need to solve the following problem: [problem description]

Please think through this step-by-step:
1. First, analyze the problem and clarify what we need to accomplish
2. Identify the key components or subproblems
3. Solve each subproblem
4. Combine the solutions
5. Verify the result

For each step, explain your reasoning before moving to the next step.
```

Example: Debugging Complex Logic with CoT

````
 I need to find the bug in this function that's supposed to find the longest
palindromic substring in a string:

```python
def longest_palindrome(s):
    if not s:
        return ""

    longest = s[0]

    for i in range(len(s)):
        # Check odd-length palindromes
        temp = expand_from_center(s, i, i)
        if len(temp) > len(longest):
            longest = temp

        # Check even-length palindromes
        temp = expand_from_center(s, i, i+1)
        if len(temp) > len(longest):
            longest = temp

    return longest

def expand_from_center(s, left, right):
    while left >= 0 and right < len(s) and s[left] == s[right]:
        left += 1
        right -= 1

    return s[left+1:right]
````

Please think through this step-by-step: 1. First, understand what the function should do and how it's supposed to work 2. Trace through the logic of both functions 3. Identify any logical errors 4. Explain the bug(s) 5. Provide a corrected implementation

For each step, explain your reasoning before moving to the next step.

````
 ### Advanced CoT: Managing Complex Development Tasks

For more complex development tasks, structure your CoT to mirror software
development best practices:
````
````

I need to develop a solution for [complex task].

Please approach this step-by-step:

1. Requirements analysis:
2. Clarify the exact requirements
3. Identify edge cases and constraints
4. System design:
5. Propose an overall architecture
6. Identify key components and their interactions
7. Choose appropriate data structures and algorithms
8. Implementation planning:
9. Break down the solution into implementable units
10. Determine the sequence of implementation
11. Implementation:
12. Write the code with clear comments
13. Explain design choices
14. Testing strategy:
15. Outline test cases covering normal operation and edge cases
16. Consider potential failure points

For each step, provide your reasoning before moving to the next step.

````
 ### Implementing CoT in Code Generation Workflows

While CoT prompting can be used directly in conversation with an LLM, you can also
embed it in automated code generation workflows:

```python
def generate_complex_solution(problem_description, language="python"):
    """Generate solution for complex programming problems using CoT."""

    cot_prompt = f"""
    I need a solution for the following problem in {language}:
````
````

```
    {problem_description}

    Please solve this step-by-step:
    1. Analyze the problem requirements
    2. Identify the key algorithms or data structures needed
    3. Design the solution approach
    4. Implement the code with appropriate comments
    5. Analyze the time and space complexity

    For each step, explain your reasoning before moving to the next step.
    """

    response = llm_client.generate(cot_prompt)

    # Extract the final code solution from the response
    # (Implementation depends on your specific LLM and response format)
    solution_code = extract_code_from_response(response)

    return {
        "full_reasoning": response,
        "code_solution": solution_code
    }
```

Self-Correction & Iterative Prompting

Self-correction techniques encourage LLMs to review and refine their own outputs, mimicking the way developers iterate on their code through debugging and refactoring.

Basic Self-Correction Template

```
 Please solve the following problem: [problem description]

After providing your solution, critically evaluate it for:
1. Correctness
2. Edge cases
3. Efficiency
4. Readability

Then provide an improved version based on your evaluation.
```

Example: Self-Correcting Code Implementation

```
 Implement a function in Python that finds all anagrams of a given word in a list of
words.

After providing your solution, critically evaluate it for:
1. Correctness
2. Edge cases (empty strings, different letter cases, etc.)
3. Time and space complexity
4. Readability and Pythonic style

Then provide an improved version based on your evaluation.
```

Multi-Round Iterative Refinement

For complex problems, we can use multi-round refinement where each iteration focuses on a specific aspect of improvement:

```
def iterative_code_refinement(initial_prompt, iterations=3):
    """Generate and iteratively refine code through multiple LLM interactions."""

    # Initial solution
    current_solution = llm_client.generate(initial_prompt)
    code = extract_code(current_solution)

    refinement_aspects = [
        "correctness and edge cases",
        "performance optimization",
        "code readability and best practices"
    ]

    for i, aspect in enumerate(refinement_aspects[:iterations]):
        refinement_prompt = f"""
        Here is a code solution:

        ```
        {code}
        ```

        Please review this code focusing specifically on {aspect}.
        Identify any issues, explain them, and provide an improved version of the
code.
        """
```

```
        refinement_response = llm_client.generate(refinement_prompt)
        improved_code = extract_code(refinement_response)

        # Update current solution if improved code was provided
        if improved_code:
            code = improved_code

        print(f"Completed refinement round {i+1}/{iterations}: {aspect}")

    return code
```

Self-Debug Pattern

The self-debug pattern specifically targets error correction by having the LLM analyze and fix issues in its own output:

```
 Generate a [language] function that [task description].

Then act as a code reviewer and:
1. Test the function with various inputs
2. Identify any bugs or edge cases that aren't handled
3. Fix the identified issues
4. Explain the bugs and your fixes
```

Example of Self-Debug Pattern

```
 Generate a JavaScript function that parses a URL string and returns an object
containing its components (protocol, host, path, query parameters, etc.).

Then act as a code reviewer and:
1. Test the function with various inputs (including URLs with and without protocols,
query parameters, fragments, etc.)
2. Identify any bugs or edge cases that aren't handled
3. Fix the identified issues
4. Explain the bugs and your fixes
```

Controlling Output: Parameter Tuning

LLM APIs provide various parameters to control the nature of the generated outputs. Understanding these parameters is crucial for fine-tuning responses to specific development needs.

Temperature: Controlling Randomness vs. Determinism

Temperature (typically 0-1) controls the randomness of predictions:

- **Temperature = 0**: More deterministic, focused responses
- **Temperature = 0.7**: Balanced creativity and coherence
- **Temperature = 1**: More random, diverse, and creative outputs

When to Use Different Temperature Settings

Temperature	Best For	Development Use Cases
0.0 - 0.1	Deterministic outputs, factual responses	Code generation, debugging, technical explanations
0.2 - 0.5	Slightly varied but focused responses	Documentation generation, code comments, refactoring suggestions
0.6 - 0.8	Creative but coherent responses	Generating alternative approaches, brainstorming solutions
0.9 - 1.0	Highly diverse and unexpected outputs	Creative problem solving, generating test cases, finding edge cases

```
def generate_code(prompt, creativity_level="low"):
    """Generate code with appropriate temperature based on creativity needs."""

    # Map creativity levels to temperature values
    temp_mapping = {
        "none": 0.0,      # Purely deterministic
        "low": 0.2,       # Slight variations
        "medium": 0.5,    # Balanced
        "high": 0.8       # Creative approaches
    }

    temperature = temp_mapping.get(creativity_level, 0.0)

    response = llm_client.generate(
        prompt,
        temperature=temperature
    )
```

```
    return response
```

Top-P (Nucleus Sampling)

Top-P (typically 0-1) controls how the model selects tokens from the probability distribution:

- **Top-P = 0.1**: Only the most likely tokens (more focused)
- **Top-P = 0.5**: More variety but still relatively constrained
- **Top-P = 0.9**: Wider range of possible outputs

For most code-related tasks, a lower Top-P (0.1-0.3) is preferable as it leads to more precise outputs.

Frequency and Presence Penalties

These parameters discourage repetition and can be useful in longer generations:

- **Frequency penalty**: Reduces likelihood of repeating the same tokens
- **Presence penalty**: Reduces likelihood of repeating topics or themes

For code generation, moderate frequency penalties (0.1-0.3) can help avoid redundant code structures.

Max Tokens and Stopping Sequences

- **Max tokens**: Limits the length of the response
- **Stopping sequences**: Specific strings that tell the model to stop generating

```
def generate_function(function_spec):
    """Generate just a function without additional explanation."""

    prompt = f"Write a function that {function_spec}. Provide only the code without
explanation."

    response = llm_client.generate(
        prompt,
        max_tokens=500,
        stop=["\n\n", "```", "def ", "function "]  # Stop after function definition
    )

    return response
```

Parameter Selection Framework

```
def select_optimal_parameters(task_type, complexity):
    """Select optimal LLM parameters based on task requirements."""

    params = {
        "temperature": 0.0,
        "top_p": 1.0,
        "frequency_penalty": 0.0,
        "presence_penalty": 0.0
    }

    # Adjust based on task type
    if task_type == "code_generation":
        params["temperature"] = 0.0
        params["top_p"] = 0.1
    elif task_type == "code_explanation":
        params["temperature"] = 0.1
        params["top_p"] = 0.3
    elif task_type == "refactoring":
        params["temperature"] = 0.2
        params["frequency_penalty"] = 0.3
    elif task_type == "creative_solution":
        params["temperature"] = 0.7
        params["top_p"] = 0.9

    # Adjust based on complexity
    if complexity == "high":
        params["temperature"] = min(params["temperature"] + 0.1, 1.0)
        params["top_p"] = min(params["top_p"] + 0.1, 1.0)
    elif complexity == "low":
        params["temperature"] = max(params["temperature"] - 0.1, 0.0)
        params["top_p"] = max(params["top_p"] - 0.1, 0.0)

    return params
```

Persona-Based Prompting

Persona-based prompts instruct the LLM to adopt a specific expertise or perspective, leading to more specialized and appropriate outputs.

The Persona Template

```
Act as a [role/persona] with expertise in [specific skills/domains]. Your task is to [specific task].

Key characteristics of this role:
- [characteristic 1]
- [characteristic 2]
- [characteristic 3]

Now, please [specific request].
```

Developer Personas for Different Tasks

Senior Developer Persona

```
Act as a senior software developer with 15+ years of experience in production environments and expertise in system design, performance optimization, and best practices. Your task is to review the following code.

Key characteristics of your role:
- Focus on maintainability and scalability
- Attention to edge cases and error handling
- Awareness of performance implications
- Experience with enterprise coding standards

Now, please review this code and suggest improvements:

```[code block]```
```

### Security Expert Persona

```
Act as a cybersecurity expert specializing in application security with experience performing security audits and penetration testing. Your task is to identify security vulnerabilities in the following code.

Key characteristics of your role:
- Deep knowledge of OWASP Top 10 vulnerabilities
- Experience with secure coding practices
- Understanding of common attack vectors
- Ability to suggest practical security mitigations
```
```

````
Now, please review this code for security vulnerabilities:

```[code block]```
````

Code Optimization Persona

````
 Act as a performance optimization specialist who focuses on making code run
efficiently. Your expertise includes algorithmic optimization, memory management,
and profiling techniques. Your task is to optimize the following function.

Key characteristics of your role:
- Deep understanding of time and space complexity
- Experience with profiling tools and bottleneck identification
- Knowledge of language-specific optimization techniques
- Focus on measurable performance improvements

Now, please optimize this code:

```[code block]```
````

Creating Composite Personas

For complex tasks, you can create composite personas that combine multiple areas of expertise:

````
 Act as a technical lead at a financial technology company who has expertise in both
secure coding practices and high-performance systems. You specialize in designing
backend systems that handle financial transactions and must balance security,
compliance, and performance. Your task is to review and improve the following
payment processing code.

Key characteristics of your role:
- Knowledge of financial compliance requirements (PCI DSS)
- Experience with secure transaction processing
- Expertise in optimizing high-throughput systems
- Background in designing fault-tolerant architectures

Now, please review and improve this payment processing code:

```[code block]```
````
````

Implementing Persona Selection in Applications

For practical applications, you can create a library of personas and select the appropriate one based on the task:

```
 PERSONA_LIBRARY = {
    "senior_dev": {
        "intro": "Act as a senior software developer with 15+ years of
experience...",
        "characteristics": [
            "Focus on maintainability and scalability",
            "Attention to edge cases and error handling",
            "Awareness of performance implications"
        ]
    },
    "security_expert": {
        "intro": "Act as a cybersecurity expert specializing in application
security...",
        "characteristics": [
            "Deep knowledge of OWASP Top 10 vulnerabilities",
            "Experience with secure coding practices",
            "Understanding of common attack vectors"
        ]
    },
    # Additional personas...
}

def generate_with_persona(persona_key, task, content=None):
    """Generate content using a specific persona."""

    if persona_key not in PERSONA_LIBRARY:
        raise ValueError(f"Unknown persona: {persona_key}")

    persona = PERSONA_LIBRARY[persona_key]

    prompt = f"{persona['intro']}\n\nKey characteristics of your role:"

    for characteristic in persona["characteristics"]:
        prompt += f"\n- {characteristic}"

    prompt += f"\n\nNow, please {task}"

    if content:
        prompt += f":\n\n```\n{content}\n```"
```

```
    response = llm_client.generate(prompt)
    return response
```

Prompt Chaining and Orchestration Techniques

Complex development tasks often require multiple LLM calls, with the output of one prompt feeding into another. Prompt chaining creates sophisticated workflows that combine multiple prompting techniques.

Basic Prompt Chain Pattern

```
def multi_stage_code_development(task_description):
    """Generate code through multiple stages of refinement."""

    # Stage 1: Design the solution approach
    design_prompt = f"""
    I need to develop a solution for: {task_description}

    Please provide a high-level design with:
    1. Key components/functions needed
    2. Data structures to use
    3. Overall algorithm or approach
    4. Potential edge cases to handle

    Just focus on the design, not the implementation.
    """

    design = llm_client.generate(design_prompt)

    # Stage 2: Implement based on the design
    implementation_prompt = f"""
    Based on the following design:

    {design}

    Implement the complete solution in code. Include comments explaining key parts.
    """

    implementation = llm_client.generate(implementation_prompt)
    code = extract_code(implementation)

    # Stage 3: Test case generation
    test_prompt = f"""
```

```
    For the following code:

    ```
    {code}
    ```

    Generate comprehensive test cases that cover:
    1. Normal operation
    2. Edge cases
    3. Error conditions

    Provide the test cases as executable code.
    """

    test_cases = llm_client.generate(test_prompt)

    return {
        "design": design,
        "implementation": code,
        "tests": extract_code(test_cases)
    }
```

Advanced Orchestration: The Specialist Pattern

The specialist pattern uses different prompts/personas for different aspects of a complex task:

```
def develop_feature_with_specialists(feature_spec):
    """Develop a complete feature using specialist personas for each aspect."""

    specialists = {
        "architect": "system design and architecture",
        "implementer": "clean, efficient implementation",
        "security_expert": "security best practices",
        "tester": "comprehensive testing",
        "documenter": "clear documentation"
    }

    results = {}
    accumulated_context = feature_spec

    # Step through specialist chain
    for role, expertise in specialists.items():
        prompt = f"""
        Act as a specialist in {expertise}.
```

```
        Project context so far:
        {accumulated_context}

        Your task is to contribute the {role} perspective to this feature.
        """

        response = llm_client.generate(prompt)
        results[role] = response

        # Add this specialist's contribution to the accumulated context
        accumulated_context += f"\n\n{role.upper()} CONTRIBUTION:\n{response}"

    return results
```

Parallel Prompting with Aggregation

For some tasks, you can get multiple independent perspectives and then combine them:

```
 import asyncio

async def get_multiple_perspectives(code_to_review, perspectives=None):
    """Get multiple review perspectives on the same code and aggregate results."""

    if perspectives is None:
        perspectives = ["readability", "performance", "security", "maintainability"]

    async def get_perspective(aspect):
        prompt = f"""
        Review the following code focusing ONLY on {aspect}:

        ```
        {code_to_review}
        ```

        Provide specific issues and recommendations related to {aspect}.
        """

        return {
            "aspect": aspect,
            "review": await llm_client.generate_async(prompt)
        }

    # Get all perspectives in parallel
    review_tasks = [get_perspective(aspect) for aspect in perspectives]
    reviews = await asyncio.gather(*review_tasks)
```

```
    # Create aggregation prompt
    aggregation_prompt = f"""
    I have received the following code reviews from different perspectives:

    """

    for review in reviews:
        aggregation_prompt += f"""
        {review['aspect'].upper()} REVIEW:
        {review['review']}

        """

    aggregation_prompt += """
    Synthesize these reviews into a consolidated summary of:
    1. The most critical issues to address
    2. Recommended improvements in priority order
    3. Positive aspects of the code worth preserving
    """

    consolidated_review = await llm_client.generate_async(aggregation_prompt)

    return {
        "individual_reviews": reviews,
        "consolidated_review": consolidated_review
    }
```

Error Handling Strategies for Inadequate LLM Responses

When working with LLMs, you'll inevitably encounter responses that don't meet your requirements. Implementing robust error handling is crucial for production applications.

Response Validation Pattern

Always validate LLM outputs before using them in your application:

```
def validate_code_response(code, language="python"):
    """Validate that an LLM-generated code snippet is valid."""

    # Basic syntactic validation
    if language == "python":
        try:
            ast.parse(code)
```

```
            return True, "Valid Python syntax"
        except SyntaxError as e:
            return False, f"Python syntax error: {str(e)}"
    elif language == "javascript":
        # Use appropriate JS parser here
        pass

    # You could add additional validation like:
    # - Checking that specific functions exist
    # - Verifying that requirements are met
    # - Testing with example inputs

    return True, "Passed validation"
```

Retry with Enhanced Context

When an LLM response is inadequate, retry with additional context:

```
def get_working_solution(problem_statement, max_attempts=3):
    """Get a working solution by retrying with improved context."""

    prompt = f"Write a function that {problem_statement}. Include proper error
handling."

    for attempt in range(1, max_attempts + 1):
        print(f"Attempt {attempt}/{max_attempts}")

        response = llm_client.generate(prompt)
        code = extract_code(response)

        valid, message = validate_code_response(code)
        if valid:
            return code

        # If invalid, enhance the prompt with error information
        prompt = f"""
        You provided the following solution:

        ```
        {code}
        ```

        However, there was an issue: {message}

        Please provide a corrected solution that addresses this problem.
```

```
        Original task: Write a function that {problem_statement}
        """

    # If we exhaust attempts, raise an exception
    raise Exception(f"Failed to generate valid code after {max_attempts} attempts")
```

Fallback Chain Strategy

Implement a chain of fallbacks when an LLM response doesn't meet requirements:

```
def generate_with_fallbacks(prompt, validators=None):
    """Generate content with a series of fallback strategies."""

    if validators is None:
        validators = [basic_validator]

    # First attempt: Standard generation
    response = llm_client.generate(prompt)

    for validator in validators:
        valid, message = validator(response)
        if valid:
            return response

    # Fallback 1: Retry with more specific instructions
    enhanced_prompt = f"""
    Previous attempt did not meet the requirements because: {message}

    Let me clarify what's needed:
    {prompt}

    Please ensure your response meets all requirements.
    """

    response = llm_client.generate(enhanced_prompt)

    for validator in validators:
        valid, message = validator(response)
        if valid:
            return response

    # Fallback 2: Try with different parameters
    response = llm_client.generate(
        enhanced_prompt,
        temperature=0.0,  # Switch to deterministic mode
```

```
        max_tokens=2000   # Allow more space for complete response
    )

    for validator in validators:
        valid, message = validator(response)
        if valid:
            return response

    # Fallback 3: Try a different model (e.g., more capable)
    response = llm_client.generate(
        enhanced_prompt,
        model="more-capable-model"  # e.g., gpt-4 instead of gpt-3.5-turbo
    )

    for validator in validators:
        valid, message = validator(response)
        if valid:
            return response

    # If all fallbacks fail, return the best effort with a warning
    return {
        "response": response,
        "warning": "Response may not meet all requirements",
        "validation_message": message
    }
```

Evaluating LLM Output Quality Programmatically

Systematic evaluation of LLM outputs is essential for maintaining quality and improving prompt design over time.

Code Execution Evaluation

For code generation tasks, the ultimate test is whether the code executes correctly:

```
 import subprocess
import tempfile
import os

def evaluate_code_execution(code, test_input=None, expected_output=None, timeout=5):
    """Evaluate code by executing it and checking the output."""

    with tempfile.NamedTemporaryFile(suffix='.py', delete=False) as temp:
        temp_name = temp.name
```

```
        temp.write(code.encode('utf-8'))

    try:
        # Execute the code
        if test_input:
            # If we have test input, provide it via stdin
            process = subprocess.run(
                ['python', temp_name],
                input=test_input.encode('utf-8'),
                capture_output=True,
                timeout=timeout
            )
        else:
            process = subprocess.run(
                ['python', temp_name],
                capture_output=True,
                timeout=timeout
            )

        stdout = process.stdout.decode('utf-8')
        stderr = process.stderr.decode('utf-8')

        # Check if execution was successful
        if process.returncode != 0:
            return False, f"Execution failed with error: {stderr}"

        # If expected output is provided, check against it
        if expected_output is not None:
            if stdout.strip() == expected_output.strip():
                return True, "Output matches expected result"
            else:
                return False, f"Output doesn't match expected result.\nExpected:
{expected_output}\nActual: {stdout}"

        return True, stdout

    except subprocess.TimeoutExpired:
        return False, f"Execution timed out after {timeout} seconds"

    finally:
        # Clean up the temporary file
        if os.path.exists(temp_name):
            os.unlink(temp_name)
```

Unit Test Generation and Execution

Generate unit tests and use them to validate LLM-generated code:

```
def evaluate_with_generated_tests(code_solution, problem_description):
    """Evaluate code by generating and running tests for it."""

    # Generate test cases based on problem description
    test_generation_prompt = f"""
    For the following problem:
    {problem_description}

    Generate comprehensive pytest unit tests that cover normal cases, edge cases,
and error cases.
    Focus only on the tests, assuming the solution is implemented in a function
called 'solution'.
    """

    test_code = llm_client.generate(test_generation_prompt)
    test_code = extract_code(test_code)

    # Combine solution and tests in a temporary file
    full_code = f"""
{code_solution}

# Generated tests
{test_code}
    """

    # Execute the tests
    with tempfile.NamedTemporaryFile(suffix='.py', delete=False) as temp:
        temp_name = temp.name
        temp.write(full_code.encode('utf-8'))

    try:
        process = subprocess.run(
            ['pytest', temp_name, '-v'],
            capture_output=True
        )

        stdout = process.stdout.decode('utf-8')
        stderr = process.stderr.decode('utf-8')

        # Parse test results
        if process.returncode == 0:
```

```
            return True, "All tests passed", stdout
        else:
            return False, "Some tests failed", stdout + "\n" + stderr

    finally:
        if os.path.exists(temp_name):
            os.unlink(temp_name)
```

Functional Requirements Verification

Verify that the generated solution meets all functional requirements:

```
def verify_requirements_coverage(code, requirements):
    """Check if code likely addresses all specified requirements."""

    evaluation_prompt = f"""
    Given the following code:

    ```
    {code}
    ```

    And these requirements:

    """

    for i, req in enumerate(requirements, 1):
        evaluation_prompt += f"{i}. {req}\n"

    evaluation_prompt += """
    For each requirement, determine if the code addresses it:
    1. Respond with YES if the requirement is clearly addressed
    2. Respond with PARTIAL if the requirement is partly addressed
    3. Respond with NO if the requirement is not addressed

    Format your response as a JSON object with requirement numbers as keys and
assessment as values,
    with an additional 'explanation' field for each requirement.
    """

    response = llm_client.generate(evaluation_prompt)

    try:
        # Extract JSON from the response
        import re
```

```
        import json

        json_match = re.search(r'{.*}', response, re.DOTALL)
        if json_match:
            assessment = json.loads(json_match.group(0))

            # Calculate coverage percentage
            covered = sum(1 for v in assessment.values() if isinstance(v, dict) and
v.get('assessment') == 'YES')
            partial = sum(0.5 for v in assessment.values() if isinstance(v, dict)
and v.get('assessment') == 'PARTIAL')
            total_requirements = len(requirements)

            coverage_pct = (covered + partial) / total_requirements * 100 if
total_requirements > 0 else 0

            return {
                "coverage_percentage": coverage_pct,
                "detailed_assessment": assessment,
                "missing_requirements": [
                    req for i, req in enumerate(requirements, 1)
                    if str(i) in assessment and assessment[str(i)].get('assessment')
== 'NO'
                ]
            }
    except Exception as e:
        return {
            "error": f"Failed to parse assessment: {str(e)}",
            "raw_response": response
        }
```

Comprehensive Evaluation Framework

For production applications, implement a comprehensive evaluation framework:

```
class LLMCodeEvaluator:
    """Framework for evaluating LLM-generated code quality."""

    def __init__(self, code, language="python"):
        self.code = code
        self.language = language
        self.evaluation_results = {}

    def run_all_evaluations(self):
        """Run all available evaluations."""
```

```
        self.evaluate_syntax()
        self.evaluate_style()
        self.evaluate_complexity()
        self.evaluate_security()

        if self.language == "python":
            self.run_python_specific_evaluations()

        return self.get_summary()

    def evaluate_syntax(self):
        """Check for syntax errors."""
        if self.language == "python":
            try:
                ast.parse(self.code)
                self.evaluation_results["syntax"] = {
                    "pass": True,
                    "message": "No syntax errors detected"
                }
            except SyntaxError as e:
                self.evaluation_results["syntax"] = {
                    "pass": False,
                    "message": f"Syntax error: {str(e)}"
                }
        # Add handlers for other languages

    def evaluate_style(self):
        """Evaluate code style compliance."""
        # For Python, could use tools like flake8, black
        # For JS, could use ESLint
        # Here's a simplified example using an LLM for style evaluation:

        style_prompt = f"""
        Review this {self.language} code for style issues:

        ```
        {self.code}
        ```

        Identify any style issues according to common {self.language} conventions.
        Return your response as a JSON with 'issues' (array) and 'score' (0-10).
        """

        response = llm_client.generate(style_prompt)
        # Parse response and extract style evaluation
        # (Implementation details omitted)
```

```
        self.evaluation_results["style"] = {
            "pass": style_score > 7,
            "score": style_score,
            "issues": style_issues
        }

    # Additional evaluation methods...

    def get_summary(self):
        """Generate overall evaluation summary."""
        total_checks = len(self.evaluation_results)
        passed_checks = sum(1 for result in self.evaluation_results.values() if
result.get("pass"))

        return {
            "overall_score": passed_checks / total_checks if total_checks > 0 else
0,
            "passed_checks": passed_checks,
            "total_checks": total_checks,
            "detailed_results": self.evaluation_results
        }
```

Conclusion

Advanced prompting techniques give developers unprecedented control over LLM outputs. By mastering Chain-of-Thought reasoning, self-correction, parameter tuning, persona-based prompting, and effective error handling strategies, you can create more reliable, higher-quality solutions for complex development tasks.

The techniques in this chapter build upon the foundation established earlier and represent the current state of the art in prompt engineering for software development. As you apply these methods in your work, you'll develop an intuitive sense for which techniques work best for different types of tasks.

In the next chapter, we'll put these advanced techniques into practice by building a complete Smart Code Assistant that can help with a wide range of development tasks.

Exercises

1. Create a Chain-of-Thought prompt for solving a complex algorithmic problem, and compare the results with a simple prompt for the same problem.
2. Implement a self-correction workflow for a code generation task that includes multiple rounds of refinement.

3. Experiment with different temperature settings for the same code generation task, and document how the outputs differ.
4. Design three different personas for code review, focusing on different aspects (e.g., security, performance, readability), and compare their feedback on the same piece of code.
5. Build a simple prompt chaining system that generates code, tests, and documentation for a specific function in sequence.

← Previous Chapter

Next Chapter →

Chapter 6: Building Effective Developer Tooling for LLM Applications

In the previous chapters, we've explored the fundamentals of prompt engineering and various techniques to create effective prompts. Now, it's time to take our skills to the next level by implementing robust developer tooling for LLM applications. As LLMs become integral parts of modern software systems, proper tooling becomes essential for maintainability, scalability, and reliability.

6.1 Prompt Libraries and Reuse Patterns

6.1.1 The Need for Prompt Management

As your project grows, managing prompts becomes increasingly challenging. Without proper organization, you might face:

- Duplicate prompts across different parts of your application
- Inconsistent prompting styles and formats
- Difficulty in tracking which prompts work best for specific tasks
- Challenges in version control and prompt evolution

6.1.2 Building a Prompt Library

Let's create a simple yet effective prompt library in Python:

```
 # A basic prompt library implementation

class PromptTemplate:
    def __init__(self, template, required_variables=None):
        self.template = template
        self.required_variables = required_variables or []

    def format(self, **kwargs):
        # Check if all required variables are provided
        missing_vars = [var for var in self.required_variables if var not in kwargs]
        if missing_vars:
            raise ValueError(f"Missing required variables: {',
```

```
'.join(missing_vars)}")

        # Format the template with the provided variables
        return self.template.format(**kwargs)

class PromptLibrary:
    def __init__(self):
        self.prompts = {}

    def add_prompt(self, name, template, required_variables=None):
        self.prompts[name] = PromptTemplate(template, required_variables)

    def get_prompt(self, name, **kwargs):
        if name not in self.prompts:
            raise KeyError(f"Prompt '{name}' not found in the library")
        return self.prompts[name].format(**kwargs)
```

Usage example:

```
 # Initialize the library
prompt_lib = PromptLibrary()

# Add prompts with templates
prompt_lib.add_prompt(
    "code_explanation",
    "Explain the following {language} code:\n\n```{language}\n{code}\n```\n\nProvide
a detailed explanation including:",
    ["language", "code"]
)

prompt_lib.add_prompt(
    "bug_fix",
    "Fix the following {language} code that has a bug:\n\n```{language}\n{code}
\n```\n\nError message: {error}\n\nProvide the corrected code and explain the fix.",
    ["language", "code", "error"]
)

# Use the prompt template
python_code = "def factorial(n):\n    if n == 0:\n        return 1\n    return n *
factorial(n-1)"
formatted_prompt = prompt_lib.get_prompt("code_explanation", language="python",
code=python_code)
```

6.1.3 Advanced Prompt Organization Patterns

For larger projects, consider organizing prompts hierarchically:

```
 # Domain-specific prompt libraries
class CodePromptLibrary(PromptLibrary):
    def __init__(self):
        super().__init__()
        self._initialize_code_prompts()

    def _initialize_code_prompts(self):
        self.add_prompt("generate_function", "Write a {language} function that
{requirement}.", ["language", "requirement"])
        self.add_prompt("optimize_code", "Optimize the following {language} code for
{optimization_goal}:\n\n```{language}\n{code}\n```", ["language",
"optimization_goal", "code"])
        # More code-related prompts...
```

6.2 Debugging Tools for LLM Applications

6.2.1 Prompt Debugging

Debugging LLM applications presents unique challenges compared to traditional software. Let's implement a simple prompt debugger:

```
 import json
from datetime import datetime

class PromptDebugger:
    def __init__(self, log_file=None):
        self.log_file = log_file
        self.history = []

    def log_interaction(self, prompt, response, metadata=None):
        interaction = {
            "timestamp": datetime.now().isoformat(),
            "prompt": prompt,
            "response": response,
            "metadata": metadata or {}
        }
        self.history.append(interaction)
```

```
        if self.log_file:
            with open(self.log_file, 'a') as f:
                f.write(json.dumps(interaction) + "\n")

        return interaction

    def analyze_token_usage(self, interaction):
        if 'token_usage' in interaction['metadata']:
            usage = interaction['metadata']['token_usage']
            return f"Prompt tokens: {usage.get('prompt_tokens', 'N/A')}, " \
                   f"Completion tokens: {usage.get('completion_tokens', 'N/A')}, " \
                   f"Total tokens: {usage.get('total_tokens', 'N/A')}"
        return "Token usage data not available"

    def compare_interactions(self, interaction1_idx, interaction2_idx):
        if interaction1_idx >= len(self.history) or interaction2_idx >=
len(self.history):
            return "Invalid interaction indices"

        int1 = self.history[interaction1_idx]
        int2 = self.history[interaction2_idx]

        # Compare prompts
        prompt_diff = self._simple_diff(int1['prompt'], int2['prompt'])

        # Compare responses (simplified)
        response_similarity = self._calculate_similarity(int1['response'],
int2['response'])

        return {
            "prompt_differences": prompt_diff,
            "response_similarity": f"{response_similarity:.2f}%"
        }

    def _simple_diff(self, text1, text2):
        # A very simple diff implementation
        # In a real application, use a proper diff library
        if text1 == text2:
            return "No differences"

        # Basic character-by-character comparison
        diffs = []
        for i, (c1, c2) in enumerate(zip(text1, text2)):
            if c1 != c2:
                diffs.append(f"Pos {i}: '{c1}' vs '{c2}'")

        if len(text1) != len(text2):
```

```
            diffs.append(f"Length difference: {len(text1)} vs {len(text2)}")

        return diffs[:10]  # Only show first 10 differences

    def _calculate_similarity(self, text1, text2):
        # Simple similarity calculation
        # In a real application, use a more sophisticated algorithm
        common_chars = sum(1 for c1, c2 in zip(text1, text2) if c1 == c2)
        total_length = max(len(text1), len(text2))
        return (common_chars / total_length) * 100 if total_length > 0 else 100
```

Example usage with OpenAI's API:

```
import openai

debugger = PromptDebugger(log_file="llm_debug_log.jsonl")

def query_llm(prompt, model="gpt-3.5-turbo"):
    response = openai.ChatCompletion.create(
        model=model,
        messages=[{"role": "user", "content": prompt}]
    )

    content = response.choices[0].message.content

    # Log the interaction with metadata
    debugger.log_interaction(
        prompt=prompt,
        response=content,
        metadata={
            "model": model,
            "token_usage": response.usage._asdict() if hasattr(response, "usage")
else None,
            "finish_reason": response.choices[0].finish_reason
        }
    )

    return content
```

6.2.2 Visualizing LLM Behavior

To understand how changes in prompts affect LLM responses, visualization tools can be invaluable:

```
 import matplotlib.pyplot as plt
import numpy as np

def visualize_token_usage(debugger, last_n=10):
    """Visualize token usage for the last N interactions"""
    if len(debugger.history) == 0:
        return "No history available"

    # Get data for the last n interactions
    history = debugger.history[-last_n:]

    prompt_tokens = []
    completion_tokens = []
    labels = []

    for i, interaction in enumerate(history):
        metadata = interaction.get('metadata', {})
        usage = metadata.get('token_usage', {})

        prompt_tokens.append(usage.get('prompt_tokens', 0))
        completion_tokens.append(usage.get('completion_tokens', 0))
        labels.append(f"Query {i+1}")

    # Create stacked bar chart
    width = 0.35
    fig, ax = plt.subplots(figsize=(12, 7))

    ax.bar(labels, prompt_tokens, width, label='Prompt Tokens')
    ax.bar(labels, completion_tokens, width, bottom=prompt_tokens, label='Completion
Tokens')

    ax.set_ylabel('Token Count')
    ax.set_title('Token Usage by Query')
    ax.legend()

    plt.tight_layout()
    plt.xticks(rotation=45)
    plt.savefig('token_usage.png')
    plt.close()

    return "Token usage visualization saved as 'token_usage.png'"
```

6.3 Performance Profiling and Optimization

6.3.1 Measuring LLM Application Performance

Performance in LLM applications involves several metrics:

```
 import time
import statistics
from functools import wraps

class LLMProfiler:
    def __init__(self):
        self.metrics = {
            "latency": [],
            "token_throughput": [],
            "success_rate": {"success": 0, "failure": 0},
            "cost": []
        }

    def profile_request(self, func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            start_time = time.time()
            error = None
            response = None

            try:
                response = func(*args, **kwargs)
                self.metrics["success_rate"]["success"] += 1
            except Exception as e:
                error = e
                self.metrics["success_rate"]["failure"] += 1

            end_time = time.time()
            latency = end_time - start_time
            self.metrics["latency"].append(latency)

            # Calculate token throughput if possible
            if response and hasattr(response, "usage"):
                total_tokens = response.usage.total_tokens
                tokens_per_second = total_tokens / latency if latency > 0 else 0
                self.metrics["token_throughput"].append(tokens_per_second)

                # Calculate approximate cost (example for GPT-3.5-turbo)
```

```
                # Rates as of 2023, adjust as needed
                prompt_cost = response.usage.prompt_tokens * 0.0015 / 1000  #
$0.0015 per 1K tokens
                completion_cost = response.usage.completion_tokens * 0.002 / 1000  #
$0.002 per 1K tokens
                total_cost = prompt_cost + completion_cost
                self.metrics["cost"].append(total_cost)

            if error:
                raise error

            return response

        return wrapper

    def get_summary(self):
        latency_stats = {
            "min": min(self.metrics["latency"]) if self.metrics["latency"] else
None,
            "max": max(self.metrics["latency"]) if self.metrics["latency"] else
None,
            "avg": statistics.mean(self.metrics["latency"]) if
self.metrics["latency"] else None,
            "p95": self._percentile(self.metrics["latency"], 95),
            "p99": self._percentile(self.metrics["latency"], 99)
        }

        throughput_stats = {
            "avg": statistics.mean(self.metrics["token_throughput"]) if
self.metrics["token_throughput"] else None
        }

        success_rate = (
            self.metrics["success_rate"]["success"] /
            (self.metrics["success_rate"]["success"] + self.metrics["success_rate"]
["failure"])
            if (self.metrics["success_rate"]["success"] +
self.metrics["success_rate"]["failure"]) > 0
            else 0
        ) * 100

        total_cost = sum(self.metrics["cost"])

        return {
            "latency_ms": {k: v*1000 if v is not None else None for k, v in
latency_stats.items()},
            "throughput_tokens_per_sec": throughput_stats,
```

```
            "success_rate_percent": success_rate,
            "total_cost_usd": total_cost,
            "request_count": len(self.metrics["latency"])
        }

    def _percentile(self, data, percentile):
        if not data:
            return None
        sorted_data = sorted(data)
        index = int(len(sorted_data) * (percentile / 100))
        return sorted_data[index]
```

6.3.2 Optimizing Prompt Performance

We can optimize prompts in several ways:

1. **Prompt Compression Techniques:**

```
def compress_prompt(prompt, max_length=None):
    """Compress a prompt by removing redundancies while preserving meaning"""
    # Simple compression techniques
    compressed = prompt

    # Remove redundant whitespace
    compressed = " ".join(compressed.split())

    # Replace common verbose phrases
    replacements = {
        "Please provide a detailed explanation of": "Explain",
        "I would like you to": "",
        "It would be great if you could": "",
        "Can you please": "",
    }

    for verbose, concise in replacements.items():
        compressed = compressed.replace(verbose, concise)

    # If a maximum length is specified, truncate while preserving key instructions
    if max_length and len(compressed) > max_length:
        # This is a simplistic approach - a real implementation would be more
sophisticated
        lines = compressed.split('. ')
        result = []
        current_length = 0
```

```
        # Always include the first line (assumed to contain the main instruction)
        result.append(lines[0])
        current_length += len(lines[0])

        # Add as many additional lines as fit within max_length
        for line in lines[1:]:
            if current_length + len(line) + 2 <= max_length:  # +2 for the '. '
                result.append(line)
                current_length += len(line) + 2
            else:
                break

        compressed = '. '.join(result)
        if not compressed.endswith('.'):
            compressed += '.'

    return compressed
```

1. **Caching LLM Responses:**

```
 import hashlib
import json
import os
import pickle

class LLMResponseCache:
    def __init__(self, cache_dir="llm_cache", ttl_seconds=86400):
        """Initialize the cache with a directory and time-to-live"""
        self.cache_dir = cache_dir
        self.ttl_seconds = ttl_seconds
        os.makedirs(cache_dir, exist_ok=True)

    def _get_cache_key(self, prompt, model, temperature):
        """Create a unique cache key from the request parameters"""
        key_data = {
            "prompt": prompt,
            "model": model,
            "temperature": temperature
        }
        key_string = json.dumps(key_data, sort_keys=True)
        return hashlib.md5(key_string.encode()).hexdigest()

    def _get_cache_path(self, key):
        """Get the file path for a cache key"""
        return os.path.join(self.cache_dir, f"{key}.pkl")
```

```
    def get(self, prompt, model, temperature=0):
        """Retrieve a response from the cache if it exists and is still valid"""
        key = self._get_cache_key(prompt, model, temperature)
        cache_path = self._get_cache_path(key)

        if not os.path.exists(cache_path):
            return None

        # Check if cache has expired
        cache_age = time.time() - os.path.getmtime(cache_path)
        if cache_age > self.ttl_seconds:
            os.remove(cache_path)  # Remove expired cache
            return None

        try:
            with open(cache_path, 'rb') as f:
                return pickle.load(f)
        except:
            return None

    def set(self, prompt, model, temperature, response):
        """Store a response in the cache"""
        key = self._get_cache_key(prompt, model, temperature)
        cache_path = self._get_cache_path(key)

        with open(cache_path, 'wb') as f:
            pickle.dump(response, f)
```

Example usage of caching:

```
cache = LLMResponseCache()

def query_llm_with_cache(prompt, model="gpt-3.5-turbo", temperature=0):
    # Try to get from cache first
    cached_response = cache.get(prompt, model, temperature)
    if cached_response:
        print("Cache hit!")
        return cached_response

    # If not in cache, make the API call
    print("Cache miss, calling API...")
    response = openai.ChatCompletion.create(
        model=model,
        messages=[{"role": "user", "content": prompt}],
        temperature=temperature
    )
```

```
    # Store in cache for future use
    cache.set(prompt, model, temperature, response)

    return response
```

6.4 Integration with Existing Development Workflows

6.4.1 Command Line Tools for Prompt Engineering

Creating a simple CLI tool for prompt engineering:

```
 #!/usr/bin/env python
import argparse
import sys
import json
import openai
from pathlib import Path

def setup_argparser():
    parser = argparse.ArgumentParser(description="LLM Prompt Engineering CLI Tool")
    subparsers = parser.add_subparsers(dest="command", help="Commands")

    # Query LLM command
    query_parser = subparsers.add_parser("query", help="Query an LLM with a prompt")
    query_parser.add_argument("--prompt", "-p", help="The prompt to send")
    query_parser.add_argument("--prompt-file", "-f", help="File containing the
prompt")
    query_parser.add_argument("--model", "-m", default="gpt-3.5-turbo", help="LLM
model to use")
    query_parser.add_argument("--output", "-o", help="Save output to file")
    query_parser.add_argument("--temperature", "-t", type=float, default=0,
help="Temperature setting")

    # Test prompt variations command
    test_parser = subparsers.add_parser("test-variations", help="Test different
prompt variations")
    test_parser.add_argument("--variations-file", required=True, help="JSON file
with prompt variations")
    test_parser.add_argument("--model", "-m", default="gpt-3.5-turbo", help="LLM
model to use")
    test_parser.add_argument("--output-dir", "-o", default="./results",
help="Directory to save results")
```

```
    # Batch processing command
    batch_parser = subparsers.add_parser("batch", help="Process a batch of prompts")
    batch_parser.add_argument("--batch-file", required=True, help="JSON file with
prompts to process")
    batch_parser.add_argument("--model", "-m", default="gpt-3.5-turbo", help="LLM
model to use")
    batch_parser.add_argument("--output-dir", "-o", default="./results",
help="Directory to save results")

    return parser

def main():
    parser = setup_argparser()
    args = parser.parse_args()

    if args.command is None:
        parser.print_help()
        return

    # Initialize OpenAI API (assuming OPENAI_API_KEY environment variable is set)
    if not openai.api_key:
        print("Error: OpenAI API key not found. Set the OPENAI_API_KEY environment
variable.")
        sys.exit(1)

    if args.command == "query":
        handle_query_command(args)
    elif args.command == "test-variations":
        handle_test_variations_command(args)
    elif args.command == "batch":
        handle_batch_command(args)

def handle_query_command(args):
    # Get prompt from arguments or file
    if args.prompt:
        prompt = args.prompt
    elif args.prompt_file:
        with open(args.prompt_file, 'r') as f:
            prompt = f.read()
    else:
        print("Error: Either --prompt or --prompt-file must be specified")
        sys.exit(1)

    # Query the LLM
    response = openai.ChatCompletion.create(
        model=args.model,
        messages=[{"role": "user", "content": prompt}],
```

```
        temperature=args.temperature
    )

    # Process the response
    content = response.choices[0].message.content

    # Output handling
    if args.output:
        with open(args.output, 'w') as f:
            f.write(content)
        print(f"Response saved to {args.output}")
    else:
        print("\n--- Response ---\n")
        print(content)
        print("\n----------------\n")

    # Print usage statistics
    print(f"Token usage: {response.usage.total_tokens} tokens")
    print(f"  - Prompt: {response.usage.prompt_tokens} tokens")
    print(f"  - Completion: {response.usage.completion_tokens} tokens")

# Additional handler functions omitted for brevity

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

6.4.2 Integrating with VS Code Extensions

For VS Code integration, consider creating a simple extension that enables developers to interact with LLMs directly from their editor.

Key features might include: - Prompt templates accessible via snippets - Highlighted code selection to LLM processing - Preview window for LLM responses - Context-aware suggestions based on the current file

6.5 Testing Frameworks for LLM-Powered Features

6.5.1 Unit Testing LLM Prompts

Creating a testing framework for prompts:

```
 import unittest
from unittest.mock import patch
```

```
import json

class PromptTest(unittest.TestCase):
    """Base class for testing prompt templates and their expected responses"""

    def setUp(self):
        # Set up mock for OpenAI API
        self.openai_patcher = patch('openai.ChatCompletion.create')
        self.mock_openai = self.openai_patcher.start()

    def tearDown(self):
        self.openai_patcher.stop()

    def assert_prompt_contains(self, prompt, required_elements):
        """Assert that a prompt contains all required elements"""
        for element in required_elements:
            self.assertIn(element, prompt, f"Prompt should contain '{element}'")

    def assert_prompt_format(self, prompt, expected_format):
        """Assert that a prompt follows the expected format structure"""
        # This is a simplified check - real implementation would be more
sophisticated
        sections = expected_format.split("[section]")
        last_pos = 0

        for section in sections[1:]:  # Skip the first empty section
            section = section.strip()
            pos = prompt.find(section, last_pos)
            self.assertGreater(pos, -1, f"Prompt missing expected section:
'{section}'")
            last_pos = pos + len(section)

    def mock_llm_response(self, response_content, usage=None):
        """Helper to set up a mock LLM response"""
        if usage is None:
            usage = {"prompt_tokens": 10, "completion_tokens": 20, "total_tokens":
30}

        # Create a response object structure similar to OpenAI's
        response = type('obj', (object,), {
            'choices': [
                type('obj', (object,), {
                    'message': type('obj', (object,), {'content':
response_content}),
                    'finish_reason': 'stop'
                })
            ],
```

```
            'usage': type('obj', (object,), usage),
            'model': 'gpt-3.5-turbo'
        })

        self.mock_openai.return_value = response

# Example test class
class TestCodeGenerationPrompts(PromptTest):
    def test_python_function_prompt(self):
        from my_prompt_lib import get_function_generation_prompt

        # Test specific prompt generation
        prompt = get_function_generation_prompt(
            language="python",
            function_name="calculate_discount",
            description="Calculate the final price after applying a discount
percentage",
            parameters=["price", "discount_percentage"]
        )

        # Verify prompt structure
        self.assert_prompt_contains(prompt, ["python", "calculate_discount",
"price", "discount_percentage"])
        self.assert_prompt_format(prompt,
"[section]Task[section]Parameters[section]Requirements")

        # Mock the LLM response
        expected_code = "def calculate_discount(price, discount_percentage):\n
return price * (1 - discount_percentage / 100)"
        self.mock_llm_response(expected_code)

        # Test the full flow from prompt to response
        from my_llm_service import generate_code
        response = generate_code(prompt)

        # Verify the response handling
        self.assertEqual(response, expected_code)
```

6.5.2 Integration Testing for LLM Applications

For integration tests:

```
class LLMIntegrationTest(unittest.TestCase):
    """Base class for integration testing of LLM-powered features"""
```

```
    def setUp(self):
        # Real API calls but with a special test API key
        # Could use a staging/test environment for the API
        import os
        os.environ["OPENAI_API_KEY"] = os.environ.get("OPENAI_TEST_API_KEY")

        # Set lower temperature for more consistent results in tests
        self.default_test_params = {
            "temperature": 0.0,
            "max_tokens": 100  # Limit tokens for faster tests
        }

    def assert_response_matches_criteria(self, response, criteria):
        """Assert that an LLM response meets a set of criteria"""
        for criterion, expected in criteria.items():
            if criterion == "contains":
                for phrase in expected:
                    self.assertIn(phrase, response, f"Response should contain
'{phrase}'")
            elif criterion == "excludes":
                for phrase in expected:
                    self.assertNotIn(phrase, response, f"Response should not contain
'{phrase}'")
            elif criterion == "length_range":
                min_len, max_len = expected
                self.assertTrue(min_len <= len(response) <= max_len,
                                f"Response length {len(response)} outside range
{min_len}-{max_len}")
            # Add more criteria types as needed
```

6.6 Cost Optimization Techniques

6.6.1 Token Counting and Budget Management

Implement a token budget manager:

```
import tiktoken

class TokenBudgetManager:
    """Manages token usage and budgets for LLM applications"""

    def __init__(self, model_name="gpt-3.5-turbo", monthly_budget=None):
        self.model_name = model_name
        self.monthly_budget = monthly_budget
```

```
        self.encoding = tiktoken.encoding_for_model(model_name)

        # Cost per 1K tokens (adjust based on current pricing)
        self.cost_per_1k_tokens = {
            "gpt-3.5-turbo": {"input": 0.0015, "output": 0.002},
            "gpt-4": {"input": 0.03, "output": 0.06}
        }.get(model_name, {"input": 0.0015, "output": 0.002})

        self.current_usage = {
            "input_tokens": 0,
            "output_tokens": 0,
            "total_cost": 0.0
        }

    def count_tokens(self, text):
        """Count the number of tokens in a text string"""
        if not text:
            return 0
        token_ids = self.encoding.encode(text)
        return len(token_ids)

    def estimate_cost(self, input_text, estimated_output_length=None):
        """Estimate the cost of an LLM request"""
        input_tokens = self.count_tokens(input_text)

        # If output length not provided, estimate based on input length
        if estimated_output_length is None:
            estimated_output_tokens = input_tokens * 1.5  # Simple heuristic
        else:
            estimated_output_tokens = self.count_tokens(estimated_output_length)

        input_cost = (input_tokens / 1000) * self.cost_per_1k_tokens["input"]
        output_cost = (estimated_output_tokens / 1000) *
self.cost_per_1k_tokens["output"]

        return {
            "input_tokens": input_tokens,
            "estimated_output_tokens": estimated_output_tokens,
            "input_cost": input_cost,
            "estimated_output_cost": output_cost,
            "total_estimated_cost": input_cost + output_cost
        }

    def track_usage(self, input_text, output_text):
        """Track actual token usage and cost"""
        input_tokens = self.count_tokens(input_text)
        output_tokens = self.count_tokens(output_text)
```

```
        input_cost = (input_tokens / 1000) * self.cost_per_1k_tokens["input"]
        output_cost = (output_tokens / 1000) * self.cost_per_1k_tokens["output"]
        total_cost = input_cost + output_cost

        # Update running totals
        self.current_usage["input_tokens"] += input_tokens
        self.current_usage["output_tokens"] += output_tokens
        self.current_usage["total_cost"] += total_cost

        return {
            "input_tokens": input_tokens,
            "output_tokens": output_tokens,
            "input_cost": input_cost,
            "output_cost": output_cost,
            "total_cost": total_cost,
            "running_totals": self.current_usage.copy()
        }

    def check_budget_status(self):
        """Check status against monthly budget"""
        if self.monthly_budget is None:
            return {"has_budget": False}

        remaining_budget = self.monthly_budget - self.current_usage["total_cost"]
        usage_percentage = (self.current_usage["total_cost"] / self.monthly_budget)
* 100

        return {
            "has_budget": True,
            "monthly_budget": self.monthly_budget,
            "current_usage": self.current_usage["total_cost"],
            "remaining_budget": remaining_budget,
            "usage_percentage": usage_percentage,
            "status": "OK" if usage_percentage < 90 else "WARNING" if
usage_percentage < 100 else "EXCEEDED"
        }
```

6.6.2 Implementing Smart Caching

We've already covered a basic caching implementation earlier. Here's an enhancement with smarter invalidation strategies:

```
class SemanticCache:
    """A cache that uses semantic similarity to match similar prompts"""
```

```
    def __init__(self, embedding_model="text-embedding-ada-002",
similarity_threshold=0.95):
        self.cache = {}  # Maps embedding hash to (response, original_prompt)
        self.embedding_model = embedding_model
        self.similarity_threshold = similarity_threshold

    def _get_embedding(self, text):
        """Get embedding vector for text"""
        response = openai.Embedding.create(
            model=self.embedding_model,
            input=text
        )
        return response["data"][0]["embedding"]

    def _compute_similarity(self, embedding1, embedding2):
        """Compute cosine similarity between embeddings"""
        import numpy as np

        # Convert to numpy arrays for vector operations
        vec1 = np.array(embedding1)
        vec2 = np.array(embedding2)

        # Compute cosine similarity
        dot_product = np.dot(vec1, vec2)
        norm1 = np.linalg.norm(vec1)
        norm2 = np.linalg.norm(vec2)

        return dot_product / (norm1 * norm2)

    def get(self, prompt, model):
        """Try to retrieve a cached response based on semantic similarity"""
        try:
            prompt_embedding = self._get_embedding(prompt)

            best_match = None
            highest_similarity = 0

            for cache_key, (cached_response, original_prompt, cached_model) in
self.cache.items():
                # Skip if models don't match
                if model != cached_model:
                    continue

                # Calculate similarity with the cached prompt
                similarity = self._compute_similarity(prompt_embedding, cache_key)
```

```
                if similarity > highest_similarity:
                    highest_similarity = similarity
                    best_match = (cached_response, original_prompt, similarity)

            # Return the best match if it meets the threshold
            if best_match and highest_similarity >= self.similarity_threshold:
                return {"response": best_match[0],
                        "original_prompt": best_match[1],
                        "similarity": highest_similarity}

            return None

        except Exception as e:
            print(f"Error in semantic cache: {e}")
            return None

    def set(self, prompt, response, model):
        """Store a response in the cache"""
        try:
            prompt_embedding = self._get_embedding(prompt)
            # Use the embedding vector as key
            embedding_key = tuple(prompt_embedding)  # Convert to tuple so it's
hashable
            self.cache[embedding_key] = (response, prompt, model)
        except Exception as e:
            print(f"Error setting semantic cache: {e}")
```

6.7 Conclusion

In this chapter, we've explored various tools and techniques for building effective developer tooling for LLM applications. From prompt libraries and reuse patterns to debugging tools, performance optimization, testing frameworks, and cost management, we've covered the essential components needed to develop robust LLM-powered applications.

As you implement these tools in your projects, remember that the field of prompt engineering is rapidly evolving. Stay flexible and be prepared to adapt your tools and approaches as new best practices emerge. In the next chapter, we'll put these tools into practice with a hands-on project building a Smart Code Assistant.

6.8 Further Reading

- "Design Patterns for LLM Applications" by Various Authors
- "Efficient Natural Language Processing" by Various Authors

- "Software Engineering for AI-Powered Systems" by Various Authors
- OpenAI API Documentation
- LangChain and LlamaIndex Documentation for advanced tooling options

---

← Previous Chapter

Next Chapter →

Chapter 7: Hands-on Project 1: Building a Smart Code Assistant

In the previous chapters, we've explored various prompt engineering techniques, patterns, and tools for developing LLM-powered applications. Now it's time to put that knowledge into practice by building a practical tool that can help streamline your daily coding tasks. In this chapter, we'll create a Smart Code Assistant that leverages LLMs to automate common coding tasks.

7.1 Project Overview

7.1.1 Problem Statement

Software development involves numerous repetitive tasks that consume valuable time and mental energy:

- Writing boilerplate code for new classes, functions, or modules
- Creating comprehensive documentation for existing code
- Refactoring code for better readability or performance
- Understanding unfamiliar code or complex algorithms
- Writing unit tests for existing code
- Converting code between different programming languages

While integrated development environments (IDEs) offer some assistance, they often lack the flexibility and contextual understanding that LLMs can provide.

7.1.2 Solution: The Smart Code Assistant

We'll build a Python-based Smart Code Assistant that leverages the power of LLMs to:

1. Generate boilerplate code based on natural language specifications
2. Analyze and explain existing code
3. Suggest refactoring improvements for small functions
4. Generate unit tests for given functions
5. Assist with code documentation
6. Provide language conversion between Python, JavaScript, and Java

7.1.3 Technical Requirements

- Python 3.8+ environment
- OpenAI API key (or equivalent for another LLM provider)
- Command-line interface for easy integration with existing workflows
- Simple, modular architecture for future extensions
- Option to save results to files or copy to clipboard

7.2 Setting Up the Project

Let's begin by setting up our project structure and installing the necessary dependencies.

7.2.1 Project Structure

```
smart_code_assistant/
├── __init__.py
├── main.py              # Entry point for the command-line interface
├── code_assistant.py    # Core functionality
├── prompt_library.py    # Prompt templates
├── utils/
│   ├── __init__.py
│   ├── clipboard.py     # Clipboard utilities
│   ├── file_utils.py    # File handling utilities
│   └── token_counter.py # Token counting utilities
├── config.py            # Configuration handling
├── requirements.txt     # Project dependencies
└── README.md            # Project documentation
```

7.2.2 Installing Dependencies

Create a `requirements.txt` file with the following dependencies:

```
openai>=1.0.0
typer>=0.9.0
rich>=13.5.0
pyperclip>=1.8.2
tiktoken>=0.5.0
python-dotenv>=1.0.0
```

Install these dependencies using pip:

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

7.2.3 Configuration Setup

Create a `config.py` file to handle API keys and other settings:

```
import os
import dotenv
from pathlib import Path

# Load environment variables from .env file
dotenv.load_dotenv()

# API configuration
OPENAI_API_KEY = os.getenv("OPENAI_API_KEY")
DEFAULT_MODEL = os.getenv("DEFAULT_MODEL", "gpt-3.5-turbo")
MAX_TOKENS = int(os.getenv("MAX_TOKENS", "2048"))
TEMPERATURE = float(os.getenv("TEMPERATURE", "0.7"))

# Application paths
APP_DIR = Path.home() / ".smart_code_assistant"
CACHE_DIR = APP_DIR / "cache"
OUTPUT_DIR = APP_DIR / "output"

# Ensure directories exist
APP_DIR.mkdir(exist_ok=True)
CACHE_DIR.mkdir(exist_ok=True)
OUTPUT_DIR.mkdir(exist_ok=True)

# Default languages supported
SUPPORTED_LANGUAGES = [
    "python", "javascript", "typescript", "java", "c++", "csharp", "go", "rust"
]
```

Create a `.env` file in the project root to store your OpenAI API key:

```
OPENAI_API_KEY=your_api_key_here
DEFAULT_MODEL=gpt-3.5-turbo
```

7.3 Building the Core Components

7.3.1 Prompt Library

First, let's create our prompt library with templates for different coding tasks. Create a `prompt_library.py` file:

```
class PromptTemplate:
    def __init__(self, template, required_params=None):
        self.template = template
        self.required_params = required_params or []

    def format(self, **kwargs):
        # Ensure all required parameters are provided
        missing = [param for param in self.required_params if param not in kwargs]
        if missing:
            raise ValueError(f"Missing required parameters: {', '.join(missing)}")

        # Format the template with the provided parameters
        return self.template.format(**kwargs)


class CodePromptLibrary:
    def __init__(self):
        self.prompts = {}
        self._initialize_prompts()

    def _initialize_prompts(self):
        # Code generation prompts
        self.prompts["generate_function"] = PromptTemplate(
            """You are an expert software developer. Write a {language} function
that {description}.

Requirements:
{requirements}

Your function should be well-documented with comments explaining the logic.
Only return the code with no additional explanations.
""",
            ["language", "description", "requirements"]
        )

        # Code explanation prompts
        self.prompts["explain_code"] = PromptTemplate(
```

```
            """Explain the following {language} code in detail:

```{language}
{code}
```

Include in your explanation: 1. What the code does 2. The key components and their purpose 3. Any algorithms or patterns used 4. Potential edge cases or limitations """, ["language", "code"])

```
    # Refactoring prompts
    self.prompts["refactor_code"] = PromptTemplate(
        """Refactor the following {language} code to improve its {focus}:

{code}
```

Provide the refactored code and explain what improvements you made. Focus specifically on improving {focus} while maintaining the same functionality. """, ["language", "code", "focus"])

```
    # Unit test generation prompts
    self.prompts["generate_tests"] = PromptTemplate(
        """Write comprehensive unit tests for the following {language} function:

{code}
```

The tests should: 1. Cover normal cases, edge cases, and potential errors 2. Be well-structured and properly named 3. Use {test_framework} as the testing framework 4. Include comments explaining the purpose of each test case """, ["language", "code", "test_framework"])

```
    # Documentation generation prompts
    self.prompts["generate_docs"] = PromptTemplate(
        """Generate comprehensive documentation for the following {language} code:

{code}
```

The documentation should: 1. Follow {doc_style} documentation style 2. Include parameter descriptions, return values, and exceptions 3. Provide a clear overview of what the code does and how to use it 4. Include usage examples where appropriate """, ["language", "code", "doc_style"])

```
    # Code conversion prompts
    self.prompts["convert_code"] = PromptTemplate(
```
```

```
        """Convert the following {source_language} code to {target_language} while
maintaining the same functionality:

{code}
```

Ensure the converted code: 1. Follows the idiomatic conventions of {target_language} 2. Preserves the original functionality and logic 3. Includes equivalent error handling 4. Is well-commented to explain any non-trivial conversions """, ["source_language", "target_language", "code"])

```
def get_prompt(self, prompt_name, **kwargs):
    """Get a formatted prompt by name with the provided parameters"""
    if prompt_name not in self.prompts:
        raise ValueError(f"Unknown prompt: {prompt_name}")

    return self.prompts[prompt_name].format(**kwargs)
```

```
### 7.3.2 Core Code Assistant Implementation

Now, let's create the core `code_assistant.py` file that will handle interactions
with the LLM:

```python
import openai
import tiktoken
import json
import time
from pathlib import Path

import config
from prompt_library import CodePromptLibrary

class SmartCodeAssistant:
    def __init__(self, api_key=None, model=None):
        """Initialize the Smart Code Assistant"""
        self.api_key = api_key or config.OPENAI_API_KEY
        self.model = model or config.DEFAULT_MODEL
        self.prompt_library = CodePromptLibrary()

        # Configure OpenAI client
        openai.api_key = self.api_key

        # Initialize tokenizer for token counting
        self.tokenizer = tiktoken.encoding_for_model(self.model)
```
```

```
    def _send_request(self, prompt, temperature=None, max_tokens=None):
        """Send a request to the OpenAI API"""
        temperature = temperature if temperature is not None else config.TEMPERATURE
        max_tokens = max_tokens if max_tokens is not None else config.MAX_TOKENS

        try:
            response = openai.ChatCompletion.create(
                model=self.model,
                messages=[{"role": "user", "content": prompt}],
                temperature=temperature,
                max_tokens=max_tokens
            )
            return response.choices[0].message.content
        except Exception as e:
            raise Exception(f"Error calling OpenAI API: {str(e)}")

    def count_tokens(self, text):
        """Count the number of tokens in the given text"""
        return len(self.tokenizer.encode(text))

    def generate_function(self, description, language="python", requirements=""):
        """Generate code based on a natural language description"""
        prompt = self.prompt_library.get_prompt(
            "generate_function",
            language=language,
            description=description,
            requirements=requirements
        )
        return self._send_request(prompt, temperature=0.2)

    def explain_code(self, code, language="python"):
        """Explain the given code in detail"""
        prompt = self.prompt_library.get_prompt(
            "explain_code",
            language=language,
            code=code
        )
        return self._send_request(prompt)

    def refactor_code(self, code, focus="readability", language="python"):
        """Refactor code to improve a specific aspect"""
        prompt = self.prompt_library.get_prompt(
            "refactor_code",
            language=language,
            code=code,
            focus=focus
        )
```

```
        return self._send_request(prompt)

    def generate_tests(self, code, language="python", test_framework="pytest"):
        """Generate unit tests for the given code"""
        prompt = self.prompt_library.get_prompt(
            "generate_tests",
            language=language,
            code=code,
            test_framework=test_framework
        )
        return self._send_request(prompt)

    def generate_docs(self, code, language="python", doc_style="Google"):
        """Generate documentation for the given code"""
        prompt = self.prompt_library.get_prompt(
            "generate_docs",
            language=language,
            code=code,
            doc_style=doc_style
        )
        return self._send_request(prompt)

    def convert_code(self, code, source_language="python",
target_language="javascript"):
        """Convert code from one language to another"""
        prompt = self.prompt_library.get_prompt(
            "convert_code",
            source_language=source_language,
            target_language=target_language,
            code=code
        )
        return self._send_request(prompt)
```

7.3.3 Utility Functions

Create a utility module for file operations and clipboard interaction. First, create the `utils/file_utils.py`:

```
 import os
from pathlib import Path

def read_file(file_path):
    """Read content from a file"""
    path = Path(file_path)
    if not path.exists():
```

```
        raise FileNotFoundError(f"File not found: {file_path}")

    with open(path, 'r', encoding='utf-8') as file:
        return file.read()

def write_file(file_path, content):
    """Write content to a file"""
    path = Path(file_path)

    # Create directories if they don't exist
    path.parent.mkdir(parents=True, exist_ok=True)

    with open(path, 'w', encoding='utf-8') as file:
        file.write(content)

    return path

def get_language_from_extension(file_path):
    """Determine language from file extension"""
    extension_map = {
        '.py': 'python',
        '.js': 'javascript',
        '.ts': 'typescript',
        '.java': 'java',
        '.cpp': 'c++',
        '.cc': 'c++',
        '.c': 'c',
        '.h': 'c',
        '.hpp': 'c++',
        '.cs': 'csharp',
        '.go': 'go',
        '.rs': 'rust',
        '.rb': 'ruby',
        '.php': 'php',
        '.swift': 'swift',
        '.kt': 'kotlin'
    }

    extension = Path(file_path).suffix.lower()
    return extension_map.get(extension, 'text')
```

Now, create `utils/clipboard.py`:

```
 import pyperclip

def copy_to_clipboard(text):
```

```
    """Copy text to clipboard"""
    try:
        pyperclip.copy(text)
        return True
    except Exception as e:
        print(f"Failed to copy to clipboard: {e}")
        return False

def paste_from_clipboard():
    """Paste text from clipboard"""
    try:
        return pyperclip.paste()
    except Exception as e:
        print(f"Failed to paste from clipboard: {e}")
        return ""
```

Create `utils/token_counter.py`:

```
import tiktoken
import config

def count_tokens(text, model=None):
    """Count tokens in the given text"""
    model = model or config.DEFAULT_MODEL

    try:
        encoding = tiktoken.encoding_for_model(model)
        return len(encoding.encode(text))
    except Exception as e:
        # Fallback to approximate token count if encoding fails
        return len(text) // 4  # Rough approximation: ~4 characters per token
```

Create an empty `__init__.py` in the utils directory to make it a proper package:

```
# This file makes the utils directory a Python package
```

7.4 Building the Command-Line Interface

Let's create a command-line interface using Typer to make our tool easily accessible. Create the `main.py` file:

```
import typer
import sys
```

```
from pathlib import Path
from typing import Optional, List
from rich.console import Console
from rich.syntax import Syntax

import config
from code_assistant import SmartCodeAssistant
from utils.file_utils import read_file, write_file, get_language_from_extension
from utils.clipboard import copy_to_clipboard, paste_from_clipboard
from utils.token_counter import count_tokens

# Initialize Typer app and Rich console
app = typer.Typer(help="Smart Code Assistant - Your AI-powered coding companion")
console = Console()

# Initialize code assistant
assistant = SmartCodeAssistant()

def print_code(code, language):
    """Print code with syntax highlighting"""
    syntax = Syntax(code, language, theme="monokai", line_numbers=True)
    console.print(syntax)

@app.command("generate")
def generate_function(
    description: str = typer.Argument(..., help="Description of the function to
generate"),
    language: str = typer.Option("python", "--language", "-l", help="Programming
language"),
    requirements: str = typer.Option("", "--requirements", "-r", help="Additional
requirements"),
    output_file: Optional[Path] = typer.Option(None, "--output", "-o", help="Output
file path"),
    copy: bool = typer.Option(False, "--copy", "-c", help="Copy result to
clipboard")
):
    """Generate code based on a natural language description"""
    console.print(f"[bold blue]Generating {language} code for:[/bold blue]
{description}")

    try:
        result = assistant.generate_function(description, language, requirements)

        # Print the result
        print_code(result, language)

        # Save to file if requested
```

```
        if output_file:
            write_file(output_file, result)
            console.print(f"[green]Code saved to:[/green] {output_file}")

        # Copy to clipboard if requested
        if copy:
            copy_to_clipboard(result)
            console.print("[green]Code copied to clipboard![/green]")

    except Exception as e:
        console.print(f"[bold red]Error:[/bold red] {str(e)}")
        raise typer.Exit(code=1)

@app.command("explain")
def explain_code(
    file: Optional[Path] = typer.Option(None, "--file", "-f", help="File containing
code to explain"),
    language: Optional[str] = typer.Option(None, "--language", "-l",
help="Programming language"),
    from_clipboard: bool = typer.Option(False, "--clipboard", "-c", help="Read code
from clipboard"),
    output_file: Optional[Path] = typer.Option(None, "--output", "-o", help="Output
file path")
):
    """Explain code in detail"""
    # Get the code from file or clipboard
    if file:
        code = read_file(file)
        language = language or get_language_from_extension(file)
    elif from_clipboard:
        code = paste_from_clipboard()
        if not code:
            console.print("[bold red]No code found in clipboard![/bold red]")
            raise typer.Exit(code=1)
    else:
        # Interactive mode - read from stdin
        console.print("[bold blue]Enter code to explain (Ctrl+D to finish):[/bold
blue]")
        code = sys.stdin.read().strip()
        if not code:
            console.print("[bold red]No code provided![/bold red]")
            raise typer.Exit(code=1)

    language = language or "python"  # Default to Python if not specified

    console.print(f"[bold blue]Explaining {language} code...[/bold blue]")
```

```
    try:
        explanation = assistant.explain_code(code, language)

        # Print the explanation
        console.print("[bold green]Explanation:[/bold green]")
        console.print(explanation)

        # Save to file if requested
        if output_file:
            write_file(output_file, explanation)
            console.print(f"[green]Explanation saved to:[/green] {output_file}")

    except Exception as e:
        console.print(f"[bold red]Error:[/bold red] {str(e)}")
        raise typer.Exit(code=1)

@app.command("refactor")
def refactor_code(
    file: Optional[Path] = typer.Option(None, "--file", "-f", help="File containing
code to refactor"),
    language: Optional[str] = typer.Option(None, "--language", "-l",
help="Programming language"),
    focus: str = typer.Option("readability", "--focus", help="What to focus on
improving (e.g., readability, performance)"),
    from_clipboard: bool = typer.Option(False, "--clipboard", "-c", help="Read code
from clipboard"),
    output_file: Optional[Path] = typer.Option(None, "--output", "-o", help="Output
file path"),
    copy: bool = typer.Option(False, "--copy", help="Copy result to clipboard")
):
    """Refactor code to improve a specific aspect"""
    # Get the code from file or clipboard
    if file:
        code = read_file(file)
        language = language or get_language_from_extension(file)
    elif from_clipboard:
        code = paste_from_clipboard()
        if not code:
            console.print("[bold red]No code found in clipboard![/bold red]")
            raise typer.Exit(code=1)
    else:
        # Interactive mode - read from stdin
        console.print(f"[bold blue]Enter code to refactor (focus: {focus}, Ctrl+D to
finish):[/bold blue]")
        code = sys.stdin.read().strip()
        if not code:
            console.print("[bold red]No code provided![/bold red]")
```

```
            raise typer.Exit(code=1)

    language = language or "python"  # Default to Python if not specified

    console.print(f"[bold blue]Refactoring {language} code to improve {focus}...[/
bold blue]")

    try:
        refactored = assistant.refactor_code(code, focus, language)

        # Print the refactored code
        console.print("[bold green]Refactored code:[/bold green]")
        print_code(refactored, language)

        # Save to file if requested
        if output_file:
            write_file(output_file, refactored)
            console.print(f"[green]Refactored code saved to:[/green] {output_file}")

        # Copy to clipboard if requested
        if copy:
            copy_to_clipboard(refactored)
            console.print("[green]Refactored code copied to clipboard![/green]")

    except Exception as e:
        console.print(f"[bold red]Error:[/bold red] {str(e)}")
        raise typer.Exit(code=1)

@app.command("test")
def generate_tests(
    file: Optional[Path] = typer.Option(None, "--file", "-f", help="File containing
code to test"),
    language: Optional[str] = typer.Option(None, "--language", "-l",
help="Programming language"),
    framework: str = typer.Option("pytest", "--framework", help="Testing framework
to use"),
    from_clipboard: bool = typer.Option(False, "--clipboard", "-c", help="Read code
from clipboard"),
    output_file: Optional[Path] = typer.Option(None, "--output", "-o", help="Output
file path")
):
    """Generate unit tests for code"""
    # Get the code from file or clipboard
    if file:
        code = read_file(file)
        language = language or get_language_from_extension(file)
    elif from_clipboard:
```

```
        code = paste_from_clipboard()
        if not code:
            console.print("[bold red]No code found in clipboard![/bold red]")
            raise typer.Exit(code=1)
    else:
        # Interactive mode - read from stdin
        console.print(f"[bold blue]Enter code to generate tests for (using
{framework}, Ctrl+D to finish):[/bold blue]")
        code = sys.stdin.read().strip()
        if not code:
            console.print("[bold red]No code provided![/bold red]")
            raise typer.Exit(code=1)

    language = language or "python"  # Default to Python if not specified

    console.print(f"[bold blue]Generating {framework} tests for {language} code...[/
bold blue]")

    try:
        tests = assistant.generate_tests(code, language, framework)

        # Print the tests
        console.print("[bold green]Generated tests:[/bold green]")
        print_code(tests, language)

        # Save to file if requested
        if output_file:
            write_file(output_file, tests)
            console.print(f"[green]Tests saved to:[/green] {output_file}")

    except Exception as e:
        console.print(f"[bold red]Error:[/bold red] {str(e)}")
        raise typer.Exit(code=1)

@app.command("docs")
def generate_docs(
    file: Optional[Path] = typer.Option(None, "--file", "-f", help="File containing
code to document"),
    language: Optional[str] = typer.Option(None, "--language", "-l",
help="Programming language"),
    style: str = typer.Option("Google", "--style", help="Documentation style
(Google, NumPy, JSDoc, etc.)"),
    from_clipboard: bool = typer.Option(False, "--clipboard", "-c", help="Read code
from clipboard"),
    output_file: Optional[Path] = typer.Option(None, "--output", "-o", help="Output
file path")
):
```

```
    """Generate documentation for code"""
    # Get the code from file or clipboard
    if file:
        code = read_file(file)
        language = language or get_language_from_extension(file)
    elif from_clipboard:
        code = paste_from_clipboard()
        if not code:
            console.print("[bold red]No code found in clipboard![/bold red]")
            raise typer.Exit(code=1)
    else:
        # Interactive mode - read from stdin
        console.print(f"[bold blue]Enter code to document (using {style} style,
Ctrl+D to finish):[/bold blue]")
        code = sys.stdin.read().strip()
        if not code:
            console.print("[bold red]No code provided![/bold red]")
            raise typer.Exit(code=1)

    language = language or "python"  # Default to Python if not specified

    console.print(f"[bold blue]Generating {style} documentation for {language}
code...[/bold blue]")

    try:
        docs = assistant.generate_docs(code, language, style)

        # Print the documentation
        console.print("[bold green]Generated documentation:[/bold green]")
        print_code(docs, language)

        # Save to file if requested
        if output_file:
            write_file(output_file, docs)
            console.print(f"[green]Documentation saved to:[/green] {output_file}")

    except Exception as e:
        console.print(f"[bold red]Error:[/bold red] {str(e)}")
        raise typer.Exit(code=1)

@app.command("convert")
def convert_code(
    source_language: str = typer.Option(..., "--from", "-f", help="Source
programming language"),
    target_language: str = typer.Option(..., "--to", "-t", help="Target programming
language"),
    file: Optional[Path] = typer.Option(None, "--file", help="File containing code
```

```
to convert"),
    from_clipboard: bool = typer.Option(False, "--clipboard", "-c", help="Read code
from clipboard"),
    output_file: Optional[Path] = typer.Option(None, "--output", "-o", help="Output
file path"),
    copy: bool = typer.Option(False, "--copy", help="Copy result to clipboard")
):
    """Convert code from one language to another"""
    # Get the code from file or clipboard
    if file:
        code = read_file(file)
    elif from_clipboard:
        code = paste_from_clipboard()
        if not code:
            console.print("[bold red]No code found in clipboard![/bold red]")
            raise typer.Exit(code=1)
    else:
        # Interactive mode - read from stdin
        console.print(f"[bold blue]Enter {source_language} code to convert to
{target_language} (Ctrl+D to finish):[/bold blue]")
        code = sys.stdin.read().strip()
        if not code:
            console.print("[bold red]No code provided![/bold red]")
            raise typer.Exit(code=1)

    console.print(f"[bold blue]Converting code from {source_language} to
{target_language}...[/bold blue]")

    try:
        converted = assistant.convert_code(code, source_language, target_language)

        # Print the converted code
        console.print("[bold green]Converted code:[/bold green]")
        print_code(converted, target_language)

        # Save to file if requested
        if output_file:
            write_file(output_file, converted)
            console.print(f"[green]Converted code saved to:[/green] {output_file}")

        # Copy to clipboard if requested
        if copy:
            copy_to_clipboard(converted)
            console.print("[green]Converted code copied to clipboard![/green]")

    except Exception as e:
        console.print(f"[bold red]Error:[/bold red] {str(e)}")
```

```
        raise typer.Exit(code=1)

@app.command("info")
def show_info():
    """Show information about the Smart Code Assistant"""
    console.print("[bold blue]Smart Code Assistant[/bold blue]")
    console.print("Your AI-powered coding companion")
    console.print("\n[bold green]Available commands:[/bold green]")
    console.print("  generate    - Generate code from a description")
    console.print("  explain     - Explain code in detail")
    console.print("  refactor    - Refactor code to improve specific aspects")
    console.print("  test        - Generate unit tests for code")
    console.print("  docs        - Generate documentation for code")
    console.print("  convert     - Convert code between languages")
    console.print("  info        - Show this information")

    console.print("\n[bold green]Configuration:[/bold green]")
    console.print(f"  Model: {config.DEFAULT_MODEL}")
    console.print(f"  Max tokens: {config.MAX_TOKENS}")
    console.print(f"  Temperature: {config.TEMPERATURE}")
    console.print(f"  Supported languages: {', '.join(config.SUPPORTED_LANGUAGES)}")

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

7.5 Project Usage Examples

Let's explore how to use our Smart Code Assistant for various tasks.

7.5.1 Generate a Function

```
 # Generate a binary search function in Python
python main.py generate "implement a binary search algorithm for a sorted list" --
language python --requirements "Must handle edge cases like empty lists and include
proper documentation"
```

7.5.2 Explain Code

```
 # Explain code from a file
python main.py explain --file complex_algorithm.py
```

```
# Explain code from clipboard
python main.py explain --clipboard --language javascript
```

7.5.3 Refactor Code

```
# Refactor code from a file to improve performance
python main.py refactor --file slow_function.py --focus performance --output
improved_function.py

# Refactor code from clipboard to improve readability
python main.py refactor --clipboard --focus readability --language python
```

7.5.4 Generate Tests

```
# Generate tests for a function in a file
python main.py test --file my_function.py --framework pytest --output
test_my_function.py

# Generate tests for code in clipboard
python main.py test --clipboard --language javascript --framework jest
```

7.5.5 Generate Documentation

```
# Generate documentation for a file
python main.py docs --file undocumented_code.py --style Google --output
documented_code.py

# Generate documentation for code in clipboard
python main.py docs --clipboard --language typescript --style JSDoc
```

7.5.6 Convert Code Between Languages

```
# Convert Python code to JavaScript
python main.py convert --from python --to javascript --file algorithm.py --output
algorithm.js

# Convert JavaScript code from clipboard to Python
python main.py convert --from javascript --to python --clipboard --copy
```

7.6 Enhancing the Smart Code Assistant

Now that we have the core functionality in place, let's explore some ways to enhance our tool.

7.6.1 Adding a Simple Caching Mechanism

To avoid unnecessary API calls and reduce costs, let's implement a simple caching mechanism:

```
# Add to code_assistant.py

import hashlib
import json
import os
from pathlib import Path
import time

class SimpleCache:
    def __init__(self, cache_dir=None, ttl=3600):
        """Initialize the cache with a directory and time-to-live in seconds"""
        self.cache_dir = Path(cache_dir or config.CACHE_DIR)
        self.ttl = ttl
        self.cache_dir.mkdir(parents=True, exist_ok=True)

    def _get_cache_key(self, prompt, model):
        """Create a hash key from the prompt and model"""
        key_str = f"{prompt}:{model}"
        return hashlib.md5(key_str.encode()).hexdigest()

    def _get_cache_path(self, key):
        """Get the file path for a cache key"""
        return self.cache_dir / f"{key}.json"

    def get(self, prompt, model):
        """Get cached response if available and not expired"""
        key = self._get_cache_key(prompt, model)
        cache_path = self._get_cache_path(key)

        if not cache_path.exists():
            return None

        # Check if cache has expired
        if time.time() - cache_path.stat().st_mtime > self.ttl:
            os.remove(cache_path)
            return None
```

```
        try:
            with open(cache_path, 'r', encoding='utf-8') as f:
                cache_data = json.load(f)
                return cache_data["response"]
        except:
            return None

    def set(self, prompt, model, response):
        """Cache a response"""
        key = self._get_cache_key(prompt, model)
        cache_path = self._get_cache_path(key)

        cache_data = {
            "prompt": prompt,
            "model": model,
            "response": response,
            "timestamp": time.time()
        }

        with open(cache_path, 'w', encoding='utf-8') as f:
            json.dump(cache_data, f, ensure_ascii=False, indent=2)
```

Update the `SmartCodeAssistant` class to use the cache:

```
 # Update _send_request method in code_assistant.py

def __init__(self, api_key=None, model=None, use_cache=True):
    # ... existing code ...
    self.use_cache = use_cache
    self.cache = SimpleCache() if use_cache else None

def _send_request(self, prompt, temperature=None, max_tokens=None):
    """Send a request to the OpenAI API with caching"""
    temperature = temperature if temperature is not None else config.TEMPERATURE
    max_tokens = max_tokens if max_tokens is not None else config.MAX_TOKENS

    # Try to get from cache if enabled
    if self.use_cache:
        cached_response = self.cache.get(prompt, self.model)
        if cached_response:
            return cached_response

    try:
        response = openai.ChatCompletion.create(
            model=self.model,
```

```
            messages=[{"role": "user", "content": prompt}],
            temperature=temperature,
            max_tokens=max_tokens
        )
        content = response.choices[0].message.content

        # Store in cache if enabled
        if self.use_cache:
            self.cache.set(prompt, self.model, content)

        return content
    except Exception as e:
        raise Exception(f"Error calling OpenAI API: {str(e)}")
```

7.6.2 Adding Progressive Enhancement with File Context

Let's enhance our code assistant to consider surrounding file context when processing partial code:

```
 # Add to code_assistant.py

def extract_file_context(self, file_path, target_lines=None, context_lines=5):
    """Extract context from a file around the target lines"""
    with open(file_path, 'r', encoding='utf-8') as f:
        all_lines = f.readlines()

    if target_lines is None:
        return "".join(all_lines)

    # Convert target_lines to a range if it's a single number
    if isinstance(target_lines, int):
        target_start = max(0, target_lines - 1)
        target_end = target_start + 1
    else:
        target_start = max(0, target_lines[0] - 1)
        target_end = min(len(all_lines), target_lines[1])

    # Extract the target code
    target_code = "".join(all_lines[target_start:target_end])

    # Get context before target
    context_before_start = max(0, target_start - context_lines)
    context_before = "".join(all_lines[context_before_start:target_start])

    # Get context after target
    context_after_end = min(len(all_lines), target_end + context_lines)
```

```
    context_after = "".join(all_lines[target_end:context_after_end])

    # Compile everything with markers
    result = ""
    if context_before:
        result += "/* Context before target code */\n" + context_before

    result += "/* Target code */\n" + target_code

    if context_after:
        result += "/* Context after target code */\n" + context_after

    return result

def refactor_with_context(self, code, file_path, target_lines, focus="readability",
language=None):
    """Refactor code with surrounding file context"""
    if not language:
        language = get_language_from_extension(file_path)

    # Extract code with context
    code_with_context = self.extract_file_context(file_path, target_lines,
context_lines=5)

    prompt = self.prompt_library.get_prompt(
        "refactor_with_context",
        language=language,
        code=code_with_context,
        focus=focus
    )

    return self._send_request(prompt)
```

Add the new prompt template to the `PromptLibrary`:

```
 # Add to _initialize_prompts in prompt_library.py

self.prompts["refactor_with_context"] = PromptTemplate(
    """Refactor the target code in the following {language} code to improve its
{focus}.
The file contains context before and after the target code to help you understand
its purpose.
Only modify the code between the "/* Target code */" markers.
```

```{language}
{code}
```

Provide ONLY the refactored target code portion and explain what improvements you made. The surrounding context is for reference only and should not be included in your response. Focus specifically on improving {focus} while maintaining the same functionality. """, ["language", "code", "focus"])

```
### 7.6.3 Adding a Project-Level Assistant

Let's extend our code assistant to understand project-level context:

```python
# Add to code_assistant.py

def analyze_project_structure(self, project_dir, max_files=20,
file_extensions=None):
    """Analyze project structure to provide context for code generation"""
    project_path = Path(project_dir)
    if not project_path.exists() or not project_path.is_dir():
        raise ValueError(f"Invalid project directory: {project_dir}")

    # Default file extensions to analyze
    if file_extensions is None:
        file_extensions = [".py", ".js", ".ts", ".java", ".c", ".cpp", ".h", ".hpp"]

    # Find relevant files
    all_files = []
    for ext in file_extensions:
        all_files.extend(project_path.glob(f"**/*{ext}"))

    # Limit the number of files to analyze
    all_files = all_files[:max_files]

    # Extract file names and structures
    project_structure = {
        "project_name": project_path.name,
        "files": [],
        "imports": [],
        "classes": [],
        "functions": []
    }

    for file_path in all_files:
        rel_path = file_path.relative_to(project_path)
```
```

```
        try:
            content = read_file(file_path)

            # Extract high-level info from the file
            file_info = {
                "path": str(rel_path),
                "extension": file_path.suffix,
                "size_bytes": file_path.stat().st_size
            }

            project_structure["files"].append(file_info)

            # Very simple extraction of Python imports, classes, and functions
            # In a real implementation, use AST parsing or other proper code
analysis
            if file_path.suffix == ".py":
                # Simple regex-based extraction
                import re

                # Find imports
                imports = re.findall(r'^import\s+(.+?)$|^from\s+(.+?)\s+import',
content, re.MULTILINE)
                for imp in imports:
                    imp_name = imp[0] or imp[1]
                    if imp_name:
                        project_structure["imports"].append(imp_name)

                # Find classes
                classes = re.findall(r'^class\s+([A-Za-z0-9_]+)', content,
re.MULTILINE)
                for cls in classes:
                    project_structure["classes"].append({
                        "name": cls,
                        "file": str(rel_path)
                    })

                # Find functions
                functions = re.findall(r'^def\s+([A-Za-z0-9_]+)', content,
re.MULTILINE)
                for func in functions:
                    project_structure["functions"].append({
                        "name": func,
                        "file": str(rel_path)
                    })

        except Exception as e:
```

```
            print(f"Error analyzing file {rel_path}: {e}")

    return project_structure

def generate_code_with_project_context(self, description, project_dir,
language=None):
    """Generate code with project context"""
    try:
        # Analyze project structure
        project_structure = self.analyze_project_structure(project_dir)

        # Determine language from project if not specified
        if language is None:
            # Simple heuristic: use most common language in project
            extensions = [f["extension"] for f in project_structure["files"]]
            if extensions:
                from collections import Counter
                most_common_ext = Counter(extensions).most_common(1)[0][0]
                language = get_language_from_extension(f"file{most_common_ext}")
            else:
                language = "python"  # Default

        # Create prompt with project context
        prompt = f"""You are an expert software developer.
I want you to generate {language} code based on the following description:

{description}

The code will be part of an existing project with the following structure:
Project name: {project_structure['project_name']}
Files: {', '.join(f['path'] for f in project_structure['files'][:10])}

Key classes in the project: {', '.join(cls['name'] for cls in
project_structure['classes'][:10])}
Key functions in the project: {', '.join(func['name'] for func in
project_structure['functions'][:10])}
Common imports: {', '.join(project_structure['imports'][:10])}

Generate code that follows the style and conventions of this existing project.
Only return the code with minimal explanatory comments.
"""

        return self._send_request(prompt, temperature=0.2)

    except Exception as e:
        raise Exception(f"Error generating code with project context: {str(e)}")
```

7.7 Practical Use Cases

Here are some practical use cases for our Smart Code Assistant:

7.7.1 Automating Repetitive Coding Tasks

Task: Creating REST API endpoint handlers

```
python main.py generate "create a Flask REST API endpoint for user registration that validates email, username, and password" --language python --requirements "Must include input validation, error handling, and follow RESTful principles"
```

Task: Generating database models

```
python main.py generate "create a SQLAlchemy model for a blog post with title, content, author, publication date, and tags" --language python
```

7.7.2 Understanding Legacy Code

Task: Explaining complex algorithms

```
python main.py explain --file legacy_algorithm.py
```

Task: Documenting undocumented functions

```
python main.py docs --file undocumented_module.py --style Google
```

7.7.3 Improving Code Quality

Task: Refactoring for performance

```
python main.py refactor --file slow_function.py --focus performance
```

Task: Creating unit tests for existing code

```
python main.py test --file data_processor.py --framework pytest
```

7.7.4 Cross-Language Development

Task: Converting Python utility to JavaScript

```
python main.py convert --from python --to javascript --file utils.py --output
utils.js
```

7.8 Best Practices and Limitations

7.8.1 Best Practices

1. **Always review the generated code**: While LLMs can provide good starting points, always review the code for correctness, security issues, and alignment with your needs.
2. **Break down complex tasks**: For better results, break complex coding tasks into smaller, more manageable pieces.
3. **Provide clear requirements**: The more specific your descriptions and requirements are, the better the generated code will be.
4. **Use project context**: When working on existing projects, providing project-level context will help generate more consistent and compatible code.
5. **Cache responses**: To reduce API costs and improve response times, implement caching for frequently requested tasks.

7.8.2 Limitations

1. **Code accuracy**: LLMs may generate code with logical errors or incorrect implementations, especially for complex algorithms.
2. **Security considerations**: Generated code might contain security vulnerabilities, so always review it carefully.
3. **Context limits**: LLMs have context window limitations, so they might struggle with understanding very large codebases or files.
4. **Language limitations**: Performance varies across programming languages, with better results typically for popular languages like Python and JavaScript.
5. **API costs**: Extensive use of LLMs can incur significant API costs, so monitor usage carefully.

7.9 Future Enhancements

Our Smart Code Assistant is just the beginning. Here are some potential future enhancements:

1. **IDE integration**: Develop plugins for popular IDEs like VS Code, PyCharm, and IntelliJ.
2. **More advanced project understanding**: Implement deeper static analysis of project structures and coding patterns.
3. **Code review capabilities**: Add features to review code changes and suggest improvements.
4. **Custom fine-tuning**: Train models on specific codebases to better match company coding styles and patterns.
5. **Collaborative features**: Allow teams to share and rate prompt templates and responses.
6. **Version control integration**: Integrate with Git to understand code history and changes over time.

7.10 Conclusion

In this chapter, we've built a practical Smart Code Assistant that demonstrates how prompt engineering can be applied to solve real-world coding challenges. By leveraging LLMs, we've created a tool that can generate code, explain existing code, refactor for improvements, create tests, and assist with documentation.

The key takeaway is that effective prompt engineering allows us to guide LLMs to produce valuable coding assistance. By structuring prompts with clear instructions, relevant context, and specific requirements, we can obtain high-quality results across a range of coding tasks.

As you continue your prompt engineering journey, consider how you might extend and customize this tool for your specific development needs. The techniques and patterns demonstrated here can be applied to a wide range of software development tasks beyond what we've covered.

In the next chapter, we'll build on these skills to create another practical application: an LLM-powered ML Model Explainer and Debugger.

← Previous Chapter

Next Chapter →

Chapter 8: Hands-on Project 2: LLM-Powered ML Model Explainer

In this chapter, we'll build on the prompt engineering skills we've developed to create a sophisticated tool that addresses one of the most challenging aspects of machine learning: model interpretability. We'll develop an LLM-powered ML Model Explainer that can analyze, interpret, and explain complex machine learning models in accessible language.

8.1 Project Overview

8.1.1 The Problem: ML Model Black Boxes

Machine learning models, especially deep learning networks, are often criticized as "black boxes" due to their complexity and lack of interpretability. This presents several challenges:

For Data Scientists and ML Engineers: - Difficulty understanding why a model makes specific predictions - Challenges in debugging model performance issues - Inability to explain model behavior to stakeholders - Struggles with model validation and trust

For Business Stakeholders: - Lack of confidence in automated decision-making - Regulatory compliance requirements for explainable AI - Need to understand model limitations and appropriate use cases - Difficulty in communicating AI capabilities to customers

For Developers Integrating ML Models: - Uncertainty about when models might fail - Challenges in debugging production issues - Difficulty in setting appropriate confidence thresholds - Need to understand model requirements and constraints

8.1.2 Solution: LLM-Powered Model Explainer

We'll create a comprehensive tool that leverages LLMs to:

1. **Analyze Model Architecture**: Break down complex model structures into understandable components
2. **Explain Hyperparameters**: Interpret the significance of various hyperparameter choices
3. **Describe Training Approaches**: Explain the training methodology and its implications
4. **Interpret Model Behavior**: Analyze predictions and feature importance
5. **Provide Usage Guidance**: Offer recommendations for appropriate model deployment

6. **Generate Documentation**: Create comprehensive model documentation automatically

8.1.3 Technical Scope

Our ML Model Explainer will support: - **Frameworks**: TensorFlow/Keras, PyTorch, Scikit-learn - **Model Types**: Neural networks (CNN, RNN, LSTM, Transformer), tree-based models, linear models - **Analysis Types**: Architecture explanation, hyperparameter interpretation, performance analysis - **Output Formats**: Interactive reports, markdown documentation, API responses

8.2 Setting Up the Project

8.2.1 Project Structure

```
ml_model_explainer/
├── __init__.py
├── main.py                 # CLI interface
├── explainer/
│   ├── __init__.py
│   ├── core.py            # Core explanation engine
│   ├── model_analyzers/   # Model-specific analyzers
│   │   ├── __init__.py
│   │   ├── base.py        # Base analyzer class
│   │   ├── keras_analyzer.py
│   │   ├── pytorch_analyzer.py
│   │   └── sklearn_analyzer.py
│   ├── prompt_templates.py # Specialized ML prompts
│   └── report_generator.py # Report generation
├── utils/
│   ├── __init__.py
│   ├── model_utils.py     # Model loading utilities
│   ├── visualization.py   # Visualization helpers
│   └── export_utils.py    # Export functionality
├── config.py
├── requirements.txt
└── examples/              # Example models and notebooks
    ├── sample_models/
    └── notebooks/
```

8.2.2 Dependencies

Create `requirements.txt`:

```
 openai>=1.0.0
tensorflow>=2.12.0
torch>=2.0.0
scikit-learn>=1.3.0
pandas>=2.0.0
numpy>=1.24.0
matplotlib>=3.7.0
seaborn>=0.12.0
plotly>=5.15.0
typer>=0.9.0
rich>=13.5.0
jinja2>=3.1.0
python-dotenv>=1.0.0
tiktoken>=0.5.0
```

8.2.3 Configuration

Create `config.py`:

```
 import os
from pathlib import Path
import dotenv

# Load environment variables
dotenv.load_dotenv()

# API Configuration
OPENAI_API_KEY = os.getenv("OPENAI_API_KEY")
DEFAULT_MODEL = os.getenv("DEFAULT_MODEL", "gpt-4")
TEMPERATURE = float(os.getenv("TEMPERATURE", "0.3"))
MAX_TOKENS = int(os.getenv("MAX_TOKENS", "3000"))

# Application paths
APP_DIR = Path.home() / ".ml_model_explainer"
CACHE_DIR = APP_DIR / "cache"
REPORTS_DIR = APP_DIR / "reports"
MODELS_DIR = APP_DIR / "models"

# Ensure directories exist
for directory in [APP_DIR, CACHE_DIR, REPORTS_DIR, MODELS_DIR]:
    directory.mkdir(exist_ok=True)

# Supported frameworks
SUPPORTED_FRAMEWORKS = ["tensorflow", "pytorch", "sklearn"]
```

```
# Model type categories
MODEL_CATEGORIES = {
    "neural_networks": ["Sequential", "Model", "CNN", "RNN", "LSTM", "GRU",
"Transformer"],
    "tree_based": ["RandomForest", "GradientBoosting", "XGBoost", "LightGBM"],
    "linear": ["LinearRegression", "LogisticRegression", "SVM", "Ridge", "Lasso"],
    "clustering": ["KMeans", "DBSCAN", "HierarchicalClustering"],
    "ensemble": ["VotingClassifier", "BaggingClassifier", "AdaBoost"]
}

# Default visualization settings
VIZ_SETTINGS = {
    "figure_size": (12, 8),
    "dpi": 300,
    "style": "seaborn-v0_8",
    "color_palette": "husl"
}
```

8.3 Building the Core Components

8.3.1 Specialized ML Prompt Templates

Create `explainer/prompt_templates.py`:

```
class MLPromptTemplates:
    def __init__(self):
        self.templates = self._initialize_templates()

    def _initialize_templates(self):
        return {
            "model_architecture_analysis": """
You are an expert machine learning engineer and educator. Analyze the following
model architecture and provide a comprehensive explanation.

Model Information:
Framework: {framework}
Model Type: {model_type}
Architecture Summary:
{architecture_summary}

Layer Details:
{layer_details}
```

```
Please provide:
1. **High-Level Overview**: What type of model this is and its primary purpose
2. **Architecture Breakdown**: Explain each component and its role
3. **Design Rationale**: Why this architecture is suitable for the intended task
4. **Strengths and Limitations**: What this model does well and where it might
struggle
5. **Computational Complexity**: Discuss the model's resource requirements

Use clear, educational language that would be accessible to both technical and non-
technical stakeholders.
""",

            "hyperparameter_explanation": """
You are an ML expert explaining model hyperparameters to a team that includes both
technical and business stakeholders.

Model: {model_type}
Hyperparameters:
{hyperparameters}

Training Configuration:
{training_config}

For each hyperparameter, explain:
1. **What it controls**: The aspect of model behavior it influences
2. **Current value significance**: Why this specific value was chosen
3. **Impact of changes**: How increasing/decreasing would affect the model
4. **Tuning considerations**: Guidelines for optimization

Conclude with an overall assessment of the hyperparameter choices and their
implications for model performance and behavior.
""",

            "training_methodology_analysis": """
You are an experienced ML practitioner explaining the training approach for a
machine learning model.

Training Details:
Model Type: {model_type}
Training Method: {training_method}
Dataset Information: {dataset_info}
Training Configuration: {training_config}
Performance Metrics: {performance_metrics}

Please explain:
1. **Training Approach**: The methodology used and why it's appropriate
2. **Data Preparation**: How the data was processed for training
```

```
3. **Optimization Strategy**: The learning approach and convergence strategy
4. **Validation Method**: How model performance was evaluated during training
5. **Performance Interpretation**: What the metrics tell us about model quality
6. **Potential Issues**: Any concerns or limitations evident from the training
process

Provide actionable insights about the model's reliability and expected performance.
""",

            "prediction_explanation": """
You are an AI explainability expert helping users understand a specific model
prediction.

Model Information:
Type: {model_type}
Task: {task_type}
Input Features: {input_features}
Prediction: {prediction}
Confidence/Probability: {confidence}

Feature Importance (if available):
{feature_importance}

Please provide:
1. **Prediction Summary**: What the model predicted and confidence level
2. **Key Drivers**: Which features most influenced this prediction
3. **Decision Logic**: The reasoning process the model likely followed
4. **Confidence Assessment**: How reliable this prediction appears to be
5. **Alternative Scenarios**: How changing key inputs might affect the outcome
6. **Limitations**: What this prediction doesn't tell us

Make the explanation accessible to non-technical users while maintaining accuracy.
""",

            "model_comparison": """
You are an ML consultant comparing different model approaches for a specific
problem.

Models to Compare:
{model_details}

Comparison Criteria:
- Performance metrics
- Interpretability
- Computational requirements
- Robustness
- Maintenance complexity
```

```
For each model, analyze:
1. **Strengths**: What it does particularly well
2. **Weaknesses**: Where it struggles or has limitations
3. **Use Case Fit**: How well it matches the intended application
4. **Trade-offs**: What you sacrifice vs. what you gain

Conclude with a recommendation for which model to use in different scenarios.
""",

            "model_deployment_guidance": """
You are a production ML expert providing deployment guidance for a trained model.

Model Details:
Type: {model_type}
Performance: {performance_summary}
Requirements: {requirements}
Constraints: {constraints}

Please provide guidance on:
1. **Deployment Readiness**: Is this model ready for production use?
2. **Infrastructure Requirements**: What resources and setup are needed?
3. **Monitoring Strategy**: What to track in production
4. **Risk Assessment**: Potential failure modes and mitigation strategies
5. **Maintenance Plan**: How to keep the model performing well over time
6. **Scaling Considerations**: How to handle increasing load or changing
requirements

Include specific, actionable recommendations for successful deployment.
"""
        }

    def get_template(self, template_name):
        if template_name not in self.templates:
            raise ValueError(f"Template '{template_name}' not found")
        return self.templates[template_name]

    def format_template(self, template_name, **kwargs):
        template = self.get_template(template_name)
        return template.format(**kwargs)
```

8.3.2 Base Model Analyzer

Create `explainer/model_analyzers/base.py`:

```
 from abc import ABC, abstractmethod
import json
from pathlib import Path
from typing import Dict, Any, List, Optional

class BaseModelAnalyzer(ABC):
    """Base class for model analyzers"""

    def __init__(self, model_path: Optional[str] = None):
        self.model_path = model_path
        self.model = None
        self.model_info = {}
        self.analysis_cache = {}

    @abstractmethod
    def load_model(self, model_path: str) -> Any:
        """Load the model from file"""
        pass

    @abstractmethod
    def extract_architecture(self) -> Dict[str, Any]:
        """Extract model architecture information"""
        pass

    @abstractmethod
    def extract_hyperparameters(self) -> Dict[str, Any]:
        """Extract model hyperparameters"""
        pass

    @abstractmethod
    def get_model_summary(self) -> str:
        """Get a string summary of the model"""
        pass

    @abstractmethod
    def predict_sample(self, sample_input: Any) -> Dict[str, Any]:
        """Make a prediction on sample input and return explanation data"""
        pass

    def analyze_model(self) -> Dict[str, Any]:
        """Perform comprehensive model analysis"""
        if not self.model:
            raise ValueError("Model not loaded. Call load_model() first.")

        analysis = {
```

```
            "framework": self.get_framework_name(),
            "model_type": self.get_model_type(),
            "architecture": self.extract_architecture(),
            "hyperparameters": self.extract_hyperparameters(),
            "summary": self.get_model_summary(),
            "complexity": self.analyze_complexity(),
            "metadata": self.extract_metadata()
        }

        self.analysis_cache = analysis
        return analysis

    @abstractmethod
    def get_framework_name(self) -> str:
        """Return the ML framework name"""
        pass

    @abstractmethod
    def get_model_type(self) -> str:
        """Return the specific model type"""
        pass

    def analyze_complexity(self) -> Dict[str, Any]:
        """Analyze model computational complexity"""
        # Default implementation - can be overridden
        complexity = {
            "estimated_parameters": "Unknown",
            "memory_usage": "Unknown",
            "inference_time": "Unknown"
        }
        return complexity

    def extract_metadata(self) -> Dict[str, Any]:
        """Extract additional metadata"""
        metadata = {
            "model_path": self.model_path,
            "analysis_timestamp": None,
            "version_info": {}
        }
        return metadata

    def save_analysis(self, output_path: str):
        """Save analysis results to file"""
        if not self.analysis_cache:
            raise ValueError("No analysis cached. Run analyze_model() first.")

        output_path = Path(output_path)
```

```
        with open(output_path, 'w') as f:
            json.dump(self.analysis_cache, f, indent=2, default=str)

    def load_analysis(self, analysis_path: str) -> Dict[str, Any]:
        """Load previously saved analysis"""
        with open(analysis_path, 'r') as f:
            self.analysis_cache = json.load(f)
        return self.analysis_cache
```

8.3.3 Keras/TensorFlow Analyzer

Create `explainer/model_analyzers/keras_analyzer.py`:

```
 import tensorflow as tf
from tensorflow import keras
import numpy as np
from typing import Dict, Any, List, Optional
import json

from .base import BaseModelAnalyzer

class KerasModelAnalyzer(BaseModelAnalyzer):
    """Analyzer for Keras/TensorFlow models"""

    def load_model(self, model_path: str):
        """Load Keras model"""
        try:
            self.model = keras.models.load_model(model_path)
            self.model_path = model_path
            return self.model
        except Exception as e:
            raise ValueError(f"Failed to load Keras model: {str(e)}")

    def extract_architecture(self) -> Dict[str, Any]:
        """Extract detailed architecture information"""
        if not self.model:
            raise ValueError("Model not loaded")

        architecture = {
            "model_class": type(self.model).__name__,
            "total_layers": len(self.model.layers),
            "input_shape": self.model.input_shape if hasattr(self.model,
'input_shape') else None,
            "output_shape": self.model.output_shape if hasattr(self.model,
'output_shape') else None,
```

```
            "layers": []
        }

        # Extract layer information
        for i, layer in enumerate(self.model.layers):
            layer_info = {
                "index": i,
                "name": layer.name,
                "class": type(layer).__name__,
                "config": self._safe_config_extract(layer),
                "input_shape": layer.input_shape if hasattr(layer, 'input_shape')
else None,
                "output_shape": layer.output_shape if hasattr(layer, 'output_shape')
else None,
                "trainable_params": layer.count_params() if hasattr(layer,
'count_params') else 0,
                "activation": getattr(layer, 'activation', None)
            }

            # Add layer-specific information
            if hasattr(layer, 'units'):
                layer_info['units'] = layer.units
            if hasattr(layer, 'filters'):
                layer_info['filters'] = layer.filters
            if hasattr(layer, 'kernel_size'):
                layer_info['kernel_size'] = layer.kernel_size
            if hasattr(layer, 'strides'):
                layer_info['strides'] = layer.strides
            if hasattr(layer, 'dropout'):
                layer_info['dropout_rate'] = layer.rate

            architecture["layers"].append(layer_info)

        return architecture

    def _safe_config_extract(self, layer) -> Dict[str, Any]:
        """Safely extract layer configuration"""
        try:
            config = layer.get_config()
            # Remove non-serializable items
            safe_config = {}
            for key, value in config.items():
                try:
                    json.dumps(value)  # Test if serializable
                    safe_config[key] = value
                except (TypeError, ValueError):
                    safe_config[key] = str(value)
```

```
            return safe_config
        except:
            return {"error": "Could not extract configuration"}

    def extract_hyperparameters(self) -> Dict[str, Any]:
        """Extract model hyperparameters"""
        hyperparams = {
            "optimizer": None,
            "loss_function": None,
            "metrics": [],
            "total_parameters": self.model.count_params() if hasattr(self.model,
'count_params') else 0,
            "trainable_parameters": sum([layer.count_params() for layer in
self.model.layers if layer.trainable])
        }

        # Extract optimizer information if model is compiled
        if hasattr(self.model, 'optimizer') and self.model.optimizer:
            optimizer = self.model.optimizer
            hyperparams["optimizer"] = {
                "class": type(optimizer).__name__,
                "learning_rate": float(optimizer.learning_rate) if
hasattr(optimizer, 'learning_rate') else None,
                "config": self._extract_optimizer_config(optimizer)
            }

        # Extract loss function
        if hasattr(self.model, 'compiled_loss') and self.model.compiled_loss:
            hyperparams["loss_function"] = str(self.model.compiled_loss)

        # Extract metrics
        if hasattr(self.model, 'compiled_metrics') and self.model.compiled_metrics:
            hyperparams["metrics"] = [str(metric) for metric in
self.model.compiled_metrics.metrics]

        return hyperparams

    def _extract_optimizer_config(self, optimizer) -> Dict[str, Any]:
        """Extract optimizer configuration"""
        try:
            config = optimizer.get_config()
            safe_config = {}
            for key, value in config.items():
                try:
                    json.dumps(value)
                    safe_config[key] = value
                except (TypeError, ValueError):
```

```
                safe_config[key] = str(value)
        return safe_config
    except:
        return {"error": "Could not extract optimizer config"}

def get_model_summary(self) -> str:
    """Get model summary as string"""
    if not self.model:
        raise ValueError("Model not loaded")

    # Capture model summary
    summary_lines = []
    self.model.summary(print_fn=lambda x: summary_lines.append(x))
    return '\n'.join(summary_lines)

def predict_sample(self, sample_input: Any) -> Dict[str, Any]:
    """Make prediction and return explanation data"""
    if not self.model:
        raise ValueError("Model not loaded")

    # Ensure input is in correct format
    if not isinstance(sample_input, np.ndarray):
        sample_input = np.array(sample_input)

    if len(sample_input.shape) == len(self.model.input_shape) - 1:
        sample_input = np.expand_dims(sample_input, axis=0)

    # Make prediction
    prediction = self.model.predict(sample_input, verbose=0)

    prediction_info = {
        "input_shape": sample_input.shape,
        "output_shape": prediction.shape,
        "prediction": prediction.tolist(),
        "input_data": sample_input.tolist()
    }

    # Add confidence for classification tasks
    if len(prediction.shape) > 1 and prediction.shape[1] > 1:
        prediction_info["confidence"] = float(np.max(prediction))
        prediction_info["predicted_class"] = int(np.argmax(prediction))
        prediction_info["class_probabilities"] = prediction[0].tolist()

    return prediction_info

def get_framework_name(self) -> str:
    return "tensorflow"
```

```
    def get_model_type(self) -> str:
        if not self.model:
            return "Unknown"

        model_class = type(self.model).__name__

        # Determine model type based on architecture
        if "Sequential" in model_class:
            return "Sequential Neural Network"
        elif "Model" in model_class:
            return "Functional Neural Network"
        else:
            return f"Custom {model_class}"

    def analyze_complexity(self) -> Dict[str, Any]:
        """Analyze computational complexity"""
        if not self.model:
            return super().analyze_complexity()

        total_params = self.model.count_params()
        trainable_params = sum([layer.count_params() for layer in self.model.layers
if layer.trainable])

        # Estimate memory usage (rough approximation)
        # Each parameter typically takes 4 bytes (float32)
        estimated_memory_mb = (total_params * 4) / (1024 * 1024)

        complexity = {
            "total_parameters": total_params,
            "trainable_parameters": trainable_params,
            "non_trainable_parameters": total_params - trainable_params,
            "estimated_memory_mb": round(estimated_memory_mb, 2),
            "model_size_layers": len(self.model.layers),
            "complexity_category": self._categorize_complexity(total_params)
        }

        return complexity

    def _categorize_complexity(self, param_count: int) -> str:
        """Categorize model complexity based on parameter count"""
        if param_count < 1000:
            return "Very Simple"
        elif param_count < 100000:
            return "Simple"
        elif param_count < 1000000:
            return "Moderate"
```

```
        elif param_count < 10000000:
            return "Complex"
        else:
            return "Very Complex"
```

8.3.4 Core Explainer Engine

Create `explainer/core.py`:

```
 import openai
from typing import Dict, Any, Optional, List
import json
from pathlib import Path

import config
from .prompt_templates import MLPromptTemplates
from .model_analyzers.keras_analyzer import KerasModelAnalyzer
from .model_analyzers.pytorch_analyzer import PyTorchModelAnalyzer
from .model_analyzers.sklearn_analyzer import SklearnModelAnalyzer

class MLModelExplainer:
    """Core explanation engine for ML models"""

    def __init__(self, api_key: Optional[str] = None, model: str = None):
        self.api_key = api_key or config.OPENAI_API_KEY
        self.model = model or config.DEFAULT_MODEL
        self.prompt_templates = MLPromptTemplates()

        # Configure OpenAI
        openai.api_key = self.api_key

        # Model analyzer mapping
        self.analyzers = {
            "tensorflow": KerasModelAnalyzer,
            "keras": KerasModelAnalyzer,
            "pytorch": PyTorchModelAnalyzer,
            "sklearn": SklearnModelAnalyzer
        }

        self.current_analyzer = None
        self.current_analysis = None

    def load_model(self, model_path: str, framework: str = None) -> Dict[str, Any]:
        """Load and analyze a model"""
        if framework is None:
```

```
            framework = self._detect_framework(model_path)

        if framework not in self.analyzers:
            raise ValueError(f"Unsupported framework: {framework}")

        # Initialize appropriate analyzer
        self.current_analyzer = self.analyzers[framework](model_path)
        self.current_analyzer.load_model(model_path)

        # Perform initial analysis
        self.current_analysis = self.current_analyzer.analyze_model()

        return self.current_analysis

    def _detect_framework(self, model_path: str) -> str:
        """Detect ML framework from model file"""
        path = Path(model_path)

        # Check file extensions and patterns
        if path.suffix == '.h5' or 'keras' in str(path) or 'tensorflow' in
str(path):
            return "tensorflow"
        elif path.suffix == '.pt' or path.suffix == '.pth' or 'pytorch' in
str(path):
            return "pytorch"
        elif path.suffix == '.pkl' or path.suffix == '.joblib':
            return "sklearn"
        else:
            # Default to tensorflow if unsure
            return "tensorflow"

    def explain_architecture(self, detail_level: str = "comprehensive") -> str:
        """Generate explanation of model architecture"""
        if not self.current_analysis:
            raise ValueError("No model analysis available. Load a model first.")

        # Prepare architecture details
        architecture = self.current_analysis["architecture"]
        layer_details = self._format_layer_details(architecture["layers"])

        prompt = self.prompt_templates.format_template(
            "model_architecture_analysis",
            framework=self.current_analysis["framework"],
            model_type=self.current_analysis["model_type"],
            architecture_summary=json.dumps(architecture, indent=2),
            layer_details=layer_details
        )
```

```
        response = self._send_request(prompt)
        return response

    def explain_hyperparameters(self) -> str:
        """Generate explanation of model hyperparameters"""
        if not self.current_analysis:
            raise ValueError("No model analysis available. Load a model first.")

        hyperparams = self.current_analysis["hyperparameters"]

        # Format training configuration
        training_config = {
            "total_parameters": hyperparams.get("total_parameters", "Unknown"),
            "trainable_parameters": hyperparams.get("trainable_parameters",
"Unknown"),
            "optimizer": hyperparams.get("optimizer", {}),
            "loss_function": hyperparams.get("loss_function", "Unknown")
        }

        prompt = self.prompt_templates.format_template(
            "hyperparameter_explanation",
            model_type=self.current_analysis["model_type"],
            hyperparameters=json.dumps(hyperparams, indent=2),
            training_config=json.dumps(training_config, indent=2)
        )

        response = self._send_request(prompt)
        return response

    def explain_prediction(self, sample_input: Any, include_feature_importance: bool
= False) -> str:
        """Explain a specific prediction"""
        if not self.current_analyzer:
            raise ValueError("No model loaded. Load a model first.")

        # Get prediction details
        prediction_info = self.current_analyzer.predict_sample(sample_input)

        # Format input features (simplified)
        input_features = f"Input shape: {prediction_info['input_shape']}"

        # Feature importance placeholder (would need additional implementation)
        feature_importance = "Feature importance analysis not yet implemented"
        if include_feature_importance:
            # This would require additional analysis like SHAP, LIME, etc.
            pass
```

```
        prompt = self.prompt_templates.format_template(
            "prediction_explanation",
            model_type=self.current_analysis["model_type"],
            task_type=self._infer_task_type(),
            input_features=input_features,
            prediction=json.dumps(prediction_info["prediction"]),
            confidence=prediction_info.get("confidence", "N/A"),
            feature_importance=feature_importance
        )

        response = self._send_request(prompt)
        return response

    def generate_deployment_guidance(self, target_environment: str = "production")
-> str:
        """Generate deployment guidance for the model"""
        if not self.current_analysis:
            raise ValueError("No model analysis available. Load a model first.")

        # Prepare performance summary
        complexity = self.current_analysis.get("complexity", {})
        performance_summary = {
            "parameter_count": complexity.get("total_parameters", "Unknown"),
            "memory_usage": complexity.get("estimated_memory_mb", "Unknown"),
            "complexity_category": complexity.get("complexity_category", "Unknown")
        }

        # Prepare requirements and constraints
        requirements = {
            "framework": self.current_analysis["framework"],
            "model_type": self.current_analysis["model_type"],
            "memory_requirements": f"{complexity.get('estimated_memory_mb',
'Unknown')} MB"
        }

        constraints = {
            "environment": target_environment,
            "scalability_needs": "To be determined",
            "latency_requirements": "To be determined"
        }

        prompt = self.prompt_templates.format_template(
            "model_deployment_guidance",
            model_type=self.current_analysis["model_type"],
            performance_summary=json.dumps(performance_summary, indent=2),
            requirements=json.dumps(requirements, indent=2),
```

```
            constraints=json.dumps(constraints, indent=2)
        )

        response = self._send_request(prompt)
        return response

    def generate_comprehensive_report(self) -> Dict[str, str]:
        """Generate a comprehensive explanation report"""
        if not self.current_analysis:
            raise ValueError("No model analysis available. Load a model first.")

        report = {
            "architecture_explanation": self.explain_architecture(),
            "hyperparameter_explanation": self.explain_hyperparameters(),
            "deployment_guidance": self.generate_deployment_guidance(),
            "model_summary": self.current_analysis["summary"],
            "technical_details": json.dumps(self.current_analysis, indent=2)
        }

        return report

    def _format_layer_details(self, layers: List[Dict]) -> str:
        """Format layer details for prompt"""
        details = []
        for layer in layers:
            layer_str = f"Layer {layer['index']}: {layer['name']}
({layer['class']})"
            if layer.get('units'):
                layer_str += f" - Units: {layer['units']}"
            if layer.get('filters'):
                layer_str += f" - Filters: {layer['filters']}"
            if layer.get('kernel_size'):
                layer_str += f" - Kernel Size: {layer['kernel_size']}"
            if layer.get('trainable_params'):
                layer_str += f" - Parameters: {layer['trainable_params']}"
            details.append(layer_str)

        return '\n'.join(details)

    def _infer_task_type(self) -> str:
        """Infer the type of ML task based on model architecture"""
        if not self.current_analysis:
            return "Unknown"

        architecture = self.current_analysis["architecture"]
        output_shape = architecture.get("output_shape")
```

```
        if output_shape:
            if isinstance(output_shape, (list, tuple)) and len(output_shape) > 1:
                output_size = output_shape[-1] if output_shape[-1] is not None else
1
                if output_size == 1:
                    return "Binary Classification or Regression"
                elif output_size > 1:
                    return "Multi-class Classification"

        return "Unknown Task Type"

    def _send_request(self, prompt: str, temperature: float = None) -> str:
        """Send request to OpenAI API"""
        temperature = temperature if temperature is not None else config.TEMPERATURE

        try:
            response = openai.ChatCompletion.create(
                model=self.model,
                messages=[{"role": "user", "content": prompt}],
                temperature=temperature,
                max_tokens=config.MAX_TOKENS
            )
            return response.choices[0].message.content
        except Exception as e:
            raise Exception(f"Error calling OpenAI API: {str(e)}")
```

8.3.5 Report Generator

Create `explainer/report_generator.py`:

```
 from jinja2 import Template
from pathlib import Path
import json
from datetime import datetime
from typing import Dict, Any, Optional

class ReportGenerator:
    """Generate formatted reports from model explanations"""

    def __init__(self, template_dir: Optional[str] = None):
        self.template_dir = Path(template_dir) if template_dir else
Path(__file__).parent / "templates"
        self.template_dir.mkdir(exist_ok=True)
        self._create_default_templates()
```

```
    def _create_default_templates(self):
        """Create default report templates"""
        # HTML Report Template
        html_template = """
<!DOCTYPE html>
<html>
<head>
    <title>ML Model Explanation Report</title>
    <style>
        body { font-family: Arial, sans-serif; max-width: 1200px; margin: 0 auto;
padding: 20px; }
        h1, h2, h3 { color: #2c3e50; }
        .section { margin-bottom: 30px; padding: 20px; border-left: 4px solid
#3498db; background-color: #f8f9fa; }
        .technical-details { background-color: #f1f2f6; padding: 15px; border-
radius: 5px; font-family: monospace; }
        .metadata { color: #7f8c8d; font-size: 0.9em; }
        table { width: 100%; border-collapse: collapse; margin: 10px 0; }
        th, td { padding: 10px; text-align: left; border-bottom: 1px solid #ddd; }
        th { background-color: #34495e; color: white; }
    </style>
</head>
<body>
    <h1>ML Model Explanation Report</h1>

    <div class="metadata">
        <p>Generated on: {{ timestamp }}</p>
        <p>Model Path: {{ model_path }}</p>
        <p>Framework: {{ framework }}</p>
    </div>

    <div class="section">
        <h2>Model Architecture</h2>
        {{ architecture_explanation }}
    </div>

    <div class="section">
        <h2>Hyperparameters</h2>
        {{ hyperparameter_explanation }}
    </div>

    <div class="section">
        <h2>Deployment Guidance</h2>
        {{ deployment_guidance }}
    </div>

    <div class="section">
```

```
        <h2>Technical Summary</h2>
        <div class="technical-details">
            <pre>{{ model_summary }}</pre>
        </div>
    </div>

    {% if technical_details %}
    <div class="section">
        <h2>Raw Analysis Data</h2>
        <details>
            <summary>Click to expand technical details</summary>
            <div class="technical-details">
                <pre>{{ technical_details }}</pre>
            </div>
        </details>
    </div>
    {% endif %}
</body>
</html>
        """

        # Markdown Report Template
        markdown_template = """
# ML Model Explanation Report

**Generated on:** {{ timestamp }}
**Model Path:** {{ model_path }}
**Framework:** {{ framework }}

## Model Architecture

{{ architecture_explanation }}

## Hyperparameters

{{ hyperparameter_explanation }}

## Deployment Guidance

{{ deployment_guidance }}

## Technical Summary
```

{{ model_summary }}

```
 {% if technical_details %}
## Raw Analysis Data

<details>
<summary>Technical Details</summary>

```json
{{ technical_details }}
```

{% endif %} """

```
     # Save templates
    with open(self.template_dir / "report.html", "w") as f:
        f.write(html_template.strip())

    with open(self.template_dir / "report.md", "w") as f:
        f.write(markdown_template.strip())

def generate_html_report(self, explanations: Dict[str, Any], analysis: Dict[str,
Any], output_path: str):
    """Generate HTML report"""
    template_path = self.template_dir / "report.html"
    with open(template_path, "r") as f:
        template = Template(f.read())

    report_data = {
        "timestamp": datetime.now().strftime("%Y-%m-%d %H:%M:%S"),
        "model_path": analysis.get("metadata", {}).get("model_path", "Unknown"),
        "framework": analysis.get("framework", "Unknown"),
        "architecture_explanation":
self._format_for_html(explanations.get("architecture_explanation", "")),
        "hyperparameter_explanation":
self._format_for_html(explanations.get("hyperparameter_explanation", "")),
        "deployment_guidance":
self._format_for_html(explanations.get("deployment_guidance", "")),
        "model_summary": explanations.get("model_summary", ""),
        "technical_details": json.dumps(analysis, indent=2)
    }

    html_content = template.render(**report_data)

    with open(output_path, "w") as f:
        f.write(html_content)

    return output_path
```
```

```
def generate_markdown_report(self, explanations: Dict[str, Any], analysis: Dict[str,
Any], output_path: str):
    """Generate Markdown report"""
    template_path = self.template_dir / "report.md"
    with open(template_path, "r") as f:
        template = Template(f.read())

    report_data = {
        "timestamp": datetime.now().strftime("%Y-%m-%d %H:%M:%S"),
        "model_path": analysis.get("metadata", {}).get("model_path", "Unknown"),
        "framework": analysis.get("framework", "Unknown"),
        "architecture_explanation": explanations.get("architecture_explanation",
""),
        "hyperparameter_explanation": explanations.get("hyperparameter_explanation",
""),
        "deployment_guidance": explanations.get("deployment_guidance", ""),
        "model_summary": explanations.get("model_summary", ""),
        "technical_details": json.dumps(analysis, indent=2)
    }

    markdown_content = template.render(**report_data)

    with open(output_path, "w") as f:
        f.write(markdown_content)

    return output_path

def _format_for_html(self, text: str) -> str:
    """Format text for HTML display"""
    # Convert line breaks to HTML line breaks
    text = text.replace('\n', '<br>\n')

    # Convert **bold** to <strong>
    import re
    text = re.sub(r'\*\*(.*?)\*\*', r'<strong>\1</strong>', text)

    # Convert *italic* to <em>
    text = re.sub(r'\*(.*?)\*', r'<em>\1</em>', text)

    return text
```

```
 ## 8.4 Building the Command-Line Interface

Create `main.py`:
```

```python
import typer
from pathlib import Path
from typing import Optional
from rich.console import Console
from rich.panel import Panel
from rich.markdown import Markdown

import config
from explainer.core import MLModelExplainer
from explainer.report_generator import ReportGenerator

app = typer.Typer(help="ML Model Explainer - AI-powered model interpretation")
console = Console()

@app.command("explain")
def explain_model(
    model_path: Path = typer.Argument(..., help="Path to the model file"),
    framework: Optional[str] = typer.Option(None, "--framework", "-f", help="ML
framework (tensorflow, pytorch, sklearn)"),
    output_dir: Optional[Path] = typer.Option(None, "--output", "-o", help="Output
directory for reports"),
    format: str = typer.Option("markdown", "--format", help="Report format (html,
markdown, json)"),
    sections: str = typer.Option("all", "--sections", help="Sections to include
(all, architecture, hyperparams, deployment)")
):
    """Explain a machine learning model comprehensively"""

    console.print(f"[bold blue]Loading model:[/bold blue] {model_path}")

    try:
        # Initialize explainer
        explainer = MLModelExplainer()

        # Load and analyze model
        analysis = explainer.load_model(str(model_path), framework)

        console.print(f"[green]✓ Model loaded successfully[/green]")
        console.print(f"Framework: {analysis['framework']}")
        console.print(f"Model Type: {analysis['model_type']}")
        console.print(f"Total Parameters: {analysis.get('complexity',
{}).get('total_parameters', 'Unknown')}")

        # Generate explanations based on requested sections
        explanations = {}
```

```
        if sections == "all" or "architecture" in sections:
            console.print("\n[bold blue]Generating architecture explanation...[/bold
blue]")
            explanations["architecture_explanation"] =
explainer.explain_architecture()

        if sections == "all" or "hyperparams" in sections:
            console.print("[bold blue]Generating hyperparameter explanation...[/bold
blue]")
            explanations["hyperparameter_explanation"] =
explainer.explain_hyperparameters()

        if sections == "all" or "deployment" in sections:
            console.print("[bold blue]Generating deployment guidance...[/bold
blue]")
            explanations["deployment_guidance"] =
explainer.generate_deployment_guidance()

        # Add model summary
        explanations["model_summary"] = analysis["summary"]

        # Display explanations
        console.print("\n" + "="*80)

        if "architecture_explanation" in explanations:
            console.print(Panel(Markdown(explanations["architecture_explanation"]),
                                title="[bold]Architecture Explanation[/bold]",
border_style="blue"))

        if "hyperparameter_explanation" in explanations:

console.print(Panel(Markdown(explanations["hyperparameter_explanation"]),
                                title="[bold]Hyperparameter Explanation[/bold]",
border_style="green"))

        if "deployment_guidance" in explanations:
            console.print(Panel(Markdown(explanations["deployment_guidance"]),
                                title="[bold]Deployment Guidance[/bold]",
border_style="yellow"))

        # Generate report if output directory specified
        if output_dir:
            output_dir = Path(output_dir)
            output_dir.mkdir(parents=True, exist_ok=True)

            report_generator = ReportGenerator()
            model_name = model_path.stem
```

```
            if format == "html":
                report_path = output_dir / f"{model_name}_explanation.html"
                report_generator.generate_html_report(explanations, analysis,
str(report_path))
                console.print(f"[green]HTML report saved to:[/green] {report_path}")

            elif format == "markdown":
                report_path = output_dir / f"{model_name}_explanation.md"
                report_generator.generate_markdown_report(explanations, analysis,
str(report_path))
                console.print(f"[green]Markdown report saved to:[/green]
{report_path}")

            elif format == "json":
                import json
                report_path = output_dir / f"{model_name}_analysis.json"
                full_report = {
                    "analysis": analysis,
                    "explanations": explanations
                }
                with open(report_path, 'w') as f:
                    json.dump(full_report, f, indent=2, default=str)
                console.print(f"[green]JSON analysis saved to:[/green]
{report_path}")

    except Exception as e:
        console.print(f"[bold red]Error:[/bold red] {str(e)}")
        raise typer.Exit(code=1)

@app.command("predict")
def explain_prediction(
    model_path: Path = typer.Argument(..., help="Path to the model file"),
    input_data: str = typer.Argument(..., help="Input data (JSON format or file
path)"),
    framework: Optional[str] = typer.Option(None, "--framework", "-f", help="ML
framework"),
    output_file: Optional[Path] = typer.Option(None, "--output", "-o", help="Output
file for explanation")
):
    """Explain a specific model prediction"""

    console.print(f"[bold blue]Loading model for prediction explanation:[/bold blue]
{model_path}")

    try:
        # Initialize explainer
```

```
        explainer = MLModelExplainer()

        # Load model
        explainer.load_model(str(model_path), framework)

        # Parse input data
        import json
        import numpy as np

        if Path(input_data).exists():
            # Load from file
            with open(input_data, 'r') as f:
                if input_data.endswith('.json'):
                    sample_input = json.load(f)
                else:
                    # Assume it's a text file with comma-separated values
                    sample_input = [float(x.strip()) for x in f.read().split(',')]
        else:
            # Parse as JSON string
            sample_input = json.loads(input_data)

        # Convert to numpy array
        sample_input = np.array(sample_input)

        console.print(f"[green]✓ Input data loaded, shape: {sample_input.shape}[/
green]")

        # Generate prediction explanation
        console.print("[bold blue]Generating prediction explanation...[/bold blue]")
        explanation = explainer.explain_prediction(sample_input)

        # Display explanation
        console.print(Panel(Markdown(explanation),
                            title="[bold]Prediction Explanation[/bold]",
border_style="cyan"))

        # Save to file if requested
        if output_file:
            with open(output_file, 'w') as f:
                f.write(explanation)
            console.print(f"[green]Explanation saved to:[/green] {output_file}")

    except Exception as e:
        console.print(f"[bold red]Error:[/bold red] {str(e)}")
        raise typer.Exit(code=1)

@app.command("compare")
```

```
def compare_models(
    model_paths: str = typer.Argument(..., help="Comma-separated list of model
paths"),
    output_dir: Optional[Path] = typer.Option(None, "--output", "-o", help="Output
directory for comparison report")
):
    """Compare multiple models"""

    paths = [p.strip() for p in model_paths.split(',')]
    console.print(f"[bold blue]Comparing {len(paths)} models...[/bold blue]")

    try:
        model_analyses = []

        for i, model_path in enumerate(paths):
            console.print(f"[blue]Analyzing model {i+1}:[/blue] {model_path}")

            explainer = MLModelExplainer()
            analysis = explainer.load_model(model_path)
            model_analyses.append({
                "path": model_path,
                "analysis": analysis,
                "explainer": explainer
            })

        # Generate comparison (simplified for this example)
        console.print("\n[bold green]Model Comparison Summary:[/bold green]")

        for i, model_data in enumerate(model_analyses):
            analysis = model_data["analysis"]
            console.print(f"\n[bold]Model {i+1}:[/bold]
{Path(model_data['path']).name}")
            console.print(f"  Framework: {analysis['framework']}")
            console.print(f"  Type: {analysis['model_type']}")
            console.print(f"  Parameters: {analysis.get('complexity',
{}).get('total_parameters', 'Unknown')}")
            console.print(f"  Complexity: {analysis.get('complexity',
{}).get('complexity_category', 'Unknown')}")

        if output_dir:
            # Generate detailed comparison report
            output_dir = Path(output_dir)
            output_dir.mkdir(parents=True, exist_ok=True)

            comparison_data = {
                "models": model_analyses,
                "timestamp": datetime.now().isoformat()
```

```
            }

            with open(output_dir / "model_comparison.json", 'w') as f:
                json.dump(comparison_data, f, indent=2, default=str)

            console.print(f"[green]Comparison data saved to:[/green] {output_dir /
'model_comparison.json'}")

    except Exception as e:
        console.print(f"[bold red]Error:[/bold red] {str(e)}")
        raise typer.Exit(code=1)

@app.command("info")
def show_info():
    """Show information about the ML Model Explainer"""
    console.print(Panel.fit("""
[bold blue]ML Model Explainer[/bold blue]

AI-powered tool for understanding and explaining machine learning models.

[bold green]Supported Frameworks:[/bold green]
• TensorFlow/Keras
• PyTorch
• Scikit-learn

[bold green]Available Commands:[/bold green]
• explain    - Comprehensive model explanation
• predict    - Explain specific predictions
• compare    - Compare multiple models
• info       - Show this information

[bold green]Configuration:[/bold green]
• Model: {model}
• Supported formats: .h5, .pt, .pth, .pkl, .joblib
    """.format(model=config.DEFAULT_MODEL),
    title="ML Model Explainer", border_style="blue"))

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

8.5 Usage Examples and Practical Applications

8.5.1 Explaining a Keras Model

```
 # Explain a complete Keras model
python main.py explain ./models/image_classifier.h5 --framework tensorflow --
output ./reports --format html

# Explain only architecture and hyperparameters
python main.py explain ./models/text_classifier.h5 --sections
architecture,hyperparams --format markdown
```

8.5.2 Understanding Model Predictions

```
 # Explain a prediction with JSON input
python main.py predict ./models/sentiment_model.h5 '[0.1, 0.5, 0.3, 0.8]' --output
prediction_explanation.md

# Explain prediction with input from file
python main.py predict ./models/price_predictor.pkl ./data/sample_input.json --
framework sklearn
```

8.5.3 Comparing Different Models

```
 # Compare multiple models
python main.py compare "./models/model_v1.h5,./models/model_v2.h5,./models/
model_v3.h5" --output ./comparison_reports
```

8.6 Best Practices and Limitations

8.6.1 Best Practices

1. **Model Documentation**: Always maintain detailed documentation about your model's training process, data preprocessing, and intended use cases.

2. **Explanation Validation**: Cross-check LLM explanations with your domain knowledge and established ML principles.

3. **Context Awareness**: Provide business context when generating explanations for stakeholders.
4. **Regular Updates**: Keep explanations current as models are retrained or updated.
5. **Security Considerations**: Be cautious about exposing sensitive model details in explanations.

8.6.2 Current Limitations

1. **Framework Coverage**: Our current implementation has basic support for major frameworks but may need extension for specialized architectures.
2. **Feature Importance**: True feature importance analysis requires additional tools like SHAP or LIME integration.
3. **Explanation Accuracy**: LLM explanations should be validated by domain experts.
4. **Complex Architectures**: Very complex or custom architectures may require specialized analysis approaches.

8.7 Future Enhancements

Potential improvements for our ML Model Explainer include:

1. **Advanced Interpretability**: Integration with SHAP, LIME, and other interpretability tools
2. **Visualization Generation**: Automatic creation of architecture diagrams and performance plots
3. **Interactive Dashboards**: Web-based interface for exploring model explanations
4. **Custom Model Support**: Extensible architecture for specialized model types
5. **Deployment Monitoring**: Integration with model monitoring and drift detection tools

8.8 Conclusion

In this chapter, we've built a sophisticated ML Model Explainer that demonstrates how LLMs can be used to make complex machine learning models more interpretable and accessible. By combining traditional model analysis techniques with the natural language generation capabilities of LLMs, we've created a tool that can bridge the gap between technical complexity and human understanding.

The key insights from this project include:

1. **Prompt Specialization**: Domain-specific prompts yield much better results than generic explanations

2. **Structured Analysis**: Breaking down model analysis into clear components (architecture, hyperparameters, etc.) enables more focused explanations
3. **Multi-Format Output**: Different stakeholders need different types of explanations and reports
4. **Framework Abstraction**: Using a base analyzer class allows for easy extension to new ML frameworks

In the next chapter, we'll build on these concepts to create an ML Training Debugger and Optimizer that can help identify and resolve common training issues.

← Previous Chapter

Next Chapter →

Chapter 9: Hands-on Project 3: ML Training Debugger and Optimizer

Building on our ML Model Explainer from Chapter 8, we now tackle one of the most frustrating aspects of machine learning: debugging failed or suboptimal training runs. This chapter focuses on creating an intelligent system that can analyze training logs, identify issues, and provide actionable optimization recommendations.

9.1 The Training Debug Challenge

9.1.1 Common Training Problems

Machine learning training often fails in subtle ways: - **Convergence Issues**: Models that won't converge or converge too slowly - **Overfitting/Underfitting**: Poor generalization or insufficient learning - **Hyperparameter Problems**: Learning rates too high/low, poor batch sizes - **Technical Issues**: Memory problems, gradient explosions, vanishing gradients

9.1.2 Why LLMs Help

Traditional debugging relies on manual interpretation of metrics and charts. LLMs can: - Recognize patterns across thousands of training runs - Correlate multiple metrics simultaneously - Provide contextual explanations in natural language - Suggest specific, actionable fixes

9.2 Project Architecture

Our ML Training Debugger will have three core components: 1. **Log Parser**: Extract metrics from TensorFlow, PyTorch, or custom logs 2. **Pattern Analyzer**: Detect common training issues automatically 3. **LLM Explainer**: Generate insights and optimization recommendations

9.3 Core Implementation

9.3.1 Training Log Parser

```
 # training_debugger.py
import pandas as pd
import re
import json
from pathlib import Path

class TrainingLogParser:
    def __init__(self, log_path):
        self.log_path = Path(log_path)
        self.data = None

    def parse_logs(self):
        """Parse training logs from various formats"""
        if self.log_path.suffix == '.csv':
            return self._parse_csv()
        elif self.log_path.suffix == '.json':
            return self._parse_json()
        else:
            return self._parse_text_logs()

    def _parse_csv(self):
        """Parse CSV training history"""
        self.data = pd.read_csv(self.log_path)
        self.data.columns = [col.lower().replace(' ', '_') for col in
self.data.columns]
        return self.data

    def _parse_text_logs(self):
        """Extract metrics from text logs using regex"""
        with open(self.log_path, 'r') as f:
            content = f.read()

        # Common patterns for training logs
        patterns = {
            'epoch': r'Epoch (\d+)',
            'loss': r'loss:\s*([0-9.]+)',
            'accuracy': r'accuracy:\s*([0-9.]+)',
            'val_loss': r'val_loss:\s*([0-9.]+)',
            'val_accuracy': r'val_accuracy:\s*([0-9.]+)',
            'lr': r'lr:\s*([0-9.e-]+)'
```

```
        }

        data_dict = {key: [] for key in patterns.keys()}

        lines = content.split('\n')
        for line in lines:
            epoch_match = re.search(patterns['epoch'], line)
            if epoch_match:
                current_epoch = int(epoch_match.group(1))

                # Extract all metrics for this epoch
                epoch_data = {'epoch': current_epoch}
                for metric, pattern in patterns.items():
                    if metric == 'epoch':
                        continue
                    match = re.search(pattern, line)
                    if match:
                        epoch_data[metric] = float(match.group(1))

                # Add to data_dict
                for key in data_dict.keys():
                    data_dict[key].append(epoch_data.get(key, None))

        self.data = pd.DataFrame(data_dict).dropna()
        return self.data
```

9.3.2 Issue Detection System

```
class TrainingIssueDetector:
    def __init__(self, data):
        self.data = data
        self.issues = []

    def detect_all_issues(self):
        """Run all issue detection methods"""
        self.detect_overfitting()
        self.detect_learning_rate_issues()
        self.detect_convergence_problems()
        self.detect_loss_explosions()
        return self.issues

    def detect_overfitting(self):
        """Detect train/validation performance gaps"""
        if 'loss' in self.data.columns and 'val_loss' in self.data.columns:
            train_loss = self.data['loss'].dropna()
```

```
            val_loss = self.data['val_loss'].dropna()

            if len(train_loss) > 10 and len(val_loss) > 10:
                # Check recent performance gap
                recent_train = train_loss.tail(5).mean()
                recent_val = val_loss.tail(5).mean()
                gap = (recent_val - recent_train) / recent_train

                if gap > 0.15:  # 15% gap threshold
                    self.issues.append({
                        'type': 'overfitting',
                        'severity': 'high' if gap > 0.3 else 'medium',
                        'description': f'Validation loss {gap:.1%} higher than
training loss',
                        'metrics': {
                            'train_loss': recent_train,
                            'val_loss': recent_val,
                            'gap': gap
                        }
                    })

    def detect_learning_rate_issues(self):
        """Detect learning rate problems"""
        if 'lr' in self.data.columns:
            current_lr = self.data['lr'].iloc[-1]

            if current_lr > 0.1:
                self.issues.append({
                    'type': 'learning_rate_too_high',
                    'severity': 'high',
                    'description': f'Learning rate {current_lr} may be too high',
                    'metrics': {'current_lr': current_lr}
                })
            elif current_lr < 1e-6:
                self.issues.append({
                    'type': 'learning_rate_too_low',
                    'severity': 'medium',
                    'description': f'Learning rate {current_lr} may be too low',
                    'metrics': {'current_lr': current_lr}
                })

    def detect_convergence_problems(self):
        """Detect poor convergence"""
        if 'loss' in self.data.columns:
            loss_series = self.data['loss'].dropna()

            if len(loss_series) > 20:
```

```
                # Check if loss is still decreasing in recent epochs
                recent_improvement = (loss_series.iloc[-10:].iloc[0] -
loss_series.iloc[-1]) / loss_series.iloc[-10:].iloc[0]

                if recent_improvement < 0.01:  # Less than 1% improvement
                    self.issues.append({
                        'type': 'poor_convergence',
                        'severity': 'medium',
                        'description': 'Loss not improving in recent epochs',
                        'metrics': {'recent_improvement': recent_improvement}
                    })

    def detect_loss_explosions(self):
        """Detect sudden loss increases"""
        if 'loss' in self.data.columns:
            loss_series = self.data['loss'].dropna()

            # Check for sudden jumps (>5x increase)
            pct_changes = loss_series.pct_change().fillna(0)
            explosions = pct_changes > 5.0

            if explosions.any():
                explosion_idx = explosions.idxmax()
                self.issues.append({
                    'type': 'loss_explosion',
                    'severity': 'critical',
                    'description': 'Sudden loss explosion detected',
                    'metrics': {
                        'explosion_epoch': explosion_idx,
                        'change_factor': pct_changes.loc[explosion_idx]
                    }
                })
```

9.3.3 LLM-Powered Analysis Engine

```
 import openai
import json

class TrainingAnalyzer:
    def __init__(self, api_key):
        openai.api_key = api_key

    def analyze_training_run(self, data, issues, hyperparams=None):
        """Generate comprehensive training analysis"""
```

```
        # Prepare training summary
        summary = self._create_training_summary(data, issues)

        prompt = f"""You are an expert ML engineer analyzing a training run.

Training Summary:
{summary}

Detected Issues:
{json.dumps(issues, indent=2)}

Current Hyperparameters:
{json.dumps(hyperparams or {}, indent=2)}

Please provide:
1. **Overall Assessment**: How is this training run performing?
2. **Root Cause Analysis**: What's causing the main issues?
3. **Specific Recommendations**: What should be changed next?
4. **Priority Actions**: What to fix first?

Be specific and actionable in your recommendations."""

        response = openai.ChatCompletion.create(
            model="gpt-4",
            messages=[{"role": "user", "content": prompt}],
            temperature=0.3,
            max_tokens=2000
        )

        return response.choices[0].message.content

    def suggest_hyperparameter_optimization(self, current_params, issues,
performance_data):
        """Suggest specific hyperparameter changes"""

        prompt = f"""You are a hyperparameter optimization expert. Based on the
training issues and performance, suggest specific parameter adjustments.

Current Hyperparameters:
{json.dumps(current_params, indent=2)}

Training Issues Detected:
{json.dumps([issue['type'] for issue in issues])}

Performance Data:
- Final Loss: {performance_data.get('final_loss', 'Unknown')}
- Best Validation: {performance_data.get('best_val', 'Unknown')}
```

```
- Training Epochs: {performance_data.get('epochs', 'Unknown')}

Provide specific recommendations for:
1. **Learning Rate**: Exact values to try
2. **Batch Size**: Optimal batch size
3. **Architecture Changes**: If needed
4. **Regularization**: Dropout, weight decay adjustments
5. **Training Schedule**: Learning rate schedules, early stopping

Focus on the top 2-3 most impactful changes."""

        response = openai.ChatCompletion.create(
            model="gpt-4",
            messages=[{"role": "user", "content": prompt}],
            temperature=0.2,
            max_tokens=1500
        )

        return response.choices[0].message.content

    def _create_training_summary(self, data, issues):
        """Create a concise training summary"""
        summary = []

        if 'epoch' in data.columns:
            summary.append(f"Total epochs: {data['epoch'].max()}")

        if 'loss' in data.columns:
            loss_series = data['loss'].dropna()
            summary.append(f"Final loss: {loss_series.iloc[-1]:.4f}")
            summary.append(f"Best loss: {loss_series.min():.4f}")

        if 'val_loss' in data.columns:
            val_loss = data['val_loss'].dropna()
            summary.append(f"Final val_loss: {val_loss.iloc[-1]:.4f}")
            summary.append(f"Best val_loss: {val_loss.min():.4f}")

        if 'accuracy' in data.columns:
            acc = data['accuracy'].dropna()
            summary.append(f"Final accuracy: {acc.iloc[-1]:.3f}")

        summary.append(f"Issues detected: {len(issues)}")

        return "\n".join(summary)
```

9.3.4 Main Training Debugger Interface

```
class MLTrainingDebugger:
    def __init__(self, api_key):
        self.analyzer = TrainingAnalyzer(api_key)

    def debug_training_run(self, log_path, hyperparams=None):
        """Complete debugging workflow"""

        # 1. Parse logs
        print("📊 Parsing training logs...")
        parser = TrainingLogParser(log_path)
        data = parser.parse_logs()
        print(f"✅ Loaded {len(data)} training records")

        # 2. Detect issues
        print("🔍 Detecting training issues...")
        detector = TrainingIssueDetector(data)
        issues = detector.detect_all_issues()
        print(f"⚠️  Found {len(issues)} potential issues")

        # 3. Generate analysis
        print("🤖 Generating AI analysis...")
        analysis = self.analyzer.analyze_training_run(data, issues, hyperparams)

        # 4. Get optimization suggestions
        performance_data = {
            'final_loss': data['loss'].iloc[-1] if 'loss' in data.columns else None,
            'best_val': data['val_loss'].min() if 'val_loss' in data.columns else
None,
            'epochs': data['epoch'].max() if 'epoch' in data.columns else len(data)
        }

        optimization = self.analyzer.suggest_hyperparameter_optimization(
            hyperparams or {}, issues, performance_data
        )

        return {
            'data': data,
            'issues': issues,
            'analysis': analysis,
            'optimization_suggestions': optimization
        }
```

9.4 Usage Examples

9.4.1 Basic Usage

```
 # Initialize debugger
debugger = MLTrainingDebugger(api_key="your-openai-key")

# Analyze a training run
result = debugger.debug_training_run(
    log_path="training_history.csv",
    hyperparams={
        "learning_rate": 0.001,
        "batch_size": 32,
        "optimizer": "adam"
    }
)

# Print analysis
print("=== TRAINING ANALYSIS ===")
print(result['analysis'])
print("\n=== OPTIMIZATION SUGGESTIONS ===")
print(result['optimization_suggestions'])
```

9.4.2 CLI Interface

```
 import typer
from rich.console import Console

app = typer.Typer()
console = Console()

@app.command()
def debug(
    log_file: str = typer.Argument(..., help="Path to training log file"),
    api_key: str = typer.Option(..., envvar="OPENAI_API_KEY", help="OpenAI API
key"),
    output: str = typer.Option(None, help="Save report to file")
):
    """Debug a machine learning training run"""

    console.print(f"[blue]Analyzing training log:[/blue] {log_file}")
```

```
    try:
        debugger = MLTrainingDebugger(api_key)
        result = debugger.debug_training_run(log_file)

        # Display results
        console.print("\n[bold green]🎯 Training Analysis[/bold green]")
        console.print(result['analysis'])

        console.print("\n[bold blue]⚡ Optimization Suggestions[/bold blue]")
        console.print(result['optimization_suggestions'])

        # Save if requested
        if output:
            with open(output, 'w') as f:
                f.write(f"# Training Debug Report\n\n")
                f.write(f"## Analysis\n{result['analysis']}\n\n")
                f.write(f"## Optimization\n{result['optimization_suggestions']}")
            console.print(f"[green]Report saved to {output}[/green]")

    except Exception as e:
        console.print(f"[red]Error: {e}[/red]")

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

9.5 Advanced Features

9.5.1 Experiment Comparison

```
def compare_experiments(self, experiment_logs):
    """Compare multiple training experiments"""

    experiments = []
    for log_path in experiment_logs:
        parser = TrainingLogParser(log_path)
        data = parser.parse_logs()
        detector = TrainingIssueDetector(data)
        issues = detector.detect_all_issues()

        experiments.append({
            'name': Path(log_path).stem,
            'final_loss': data['loss'].iloc[-1] if 'loss' in data.columns else None,
            'best_val': data['val_loss'].min() if 'val_loss' in data.columns else
None,
```

```
            'issues': len(issues),
            'converged': len([i for i in issues if i['type'] == 'poor_convergence'])
== 0
        })

    # Generate comparison analysis
    comparison_prompt = f"""Compare these ML experiments and identify the best
performing approach:

{json.dumps(experiments, indent=2)}

Provide:
1. **Best Experiment**: Which performed best and why?
2. **Key Patterns**: What patterns lead to success?
3. **Recommendations**: What to try next?"""

    # ...rest of comparison logic
```

9.5.2 Visualization Integration

```
import matplotlib.pyplot as plt

def generate_training_plots(data, output_dir="plots"):
    """Generate training visualization plots"""

    fig, axes = plt.subplots(2, 2, figsize=(12, 8))

    # Loss curves
    if 'loss' in data.columns:
        axes[0,0].plot(data['epoch'], data['loss'], label='Training Loss')
        if 'val_loss' in data.columns:
            axes[0,0].plot(data['epoch'], data['val_loss'], label='Validation Loss')
        axes[0,0].set_title('Loss Curves')
        axes[0,0].legend()

    # Accuracy curves
    if 'accuracy' in data.columns:
        axes[0,1].plot(data['epoch'], data['accuracy'], label='Training Accuracy')
        if 'val_accuracy' in data.columns:
            axes[0,1].plot(data['epoch'], data['val_accuracy'], label='Validation
Accuracy')
        axes[0,1].set_title('Accuracy Curves')
        axes[0,1].legend()

    # Learning rate schedule
```

```
    if 'lr' in data.columns:
        axes[1,0].plot(data['epoch'], data['lr'])
        axes[1,0].set_title('Learning Rate Schedule')
        axes[1,0].set_yscale('log')

    plt.tight_layout()
    plt.savefig(f"{output_dir}/training_analysis.png", dpi=300, bbox_inches='tight')
    plt.close()
```

9.6 Best Practices and Limitations

9.6.1 Best Practices

1. **Log Everything**: Include learning rates, gradient norms, and custom metrics
2. **Consistent Formatting**: Use standard logging formats for easier parsing
3. **Domain Context**: Provide model architecture and dataset information
4. **Multiple Runs**: Compare across experiments for better insights
5. **Human Validation**: Always verify LLM suggestions with domain knowledge

9.6.2 Current Limitations

1. **Pattern Recognition**: LLMs may miss domain-specific issues
2. **Causation vs Correlation**: May suggest fixes that don't address root causes
3. **Framework Specifics**: Different frameworks have unique debugging needs
4. **Resource Costs**: Extensive analysis can be expensive with API calls

9.7 Conclusion

The ML Training Debugger demonstrates how LLMs can transform the traditionally manual and expertise-heavy process of training diagnosis. By combining automatic issue detection with intelligent analysis, we can:

- **Reduce Debug Time**: Quickly identify common training problems
- **Improve Training Success**: Get specific, actionable recommendations
- **Learn Faster**: Understand why certain approaches work or fail
- **Scale Expertise**: Make advanced debugging accessible to more developers

The key insight is that LLMs excel at pattern recognition across training metrics when provided with the right context and structured prompts. This approach can significantly accelerate the iterative process of ML model development.

In Part 2 of this book, we'll shift focus to architectural considerations for deploying LLM-powered systems at enterprise scale.

← Previous Chapter

Index