🔌 Module 6: Production and Integration

> ***Weeks 15-16*** | *Taking prompts to production*

Week 15: API Integration and Automation

🔗 Working with LLM APIs

Theory: Production systems interact with LLMs through APIs, not chat interfaces.

Basic API Call (Python + OpenAI):

```python
from openai import OpenAI

client = OpenAI(api_key="your-api-key")

def call_llm(prompt, model="gpt-4", temperature=0.7):
    response = client.chat.completions.create(
        model=model,
        messages=[
            {"role": "system", "content": "You are a helpful assistant."},
            {"role": "user", "content": prompt}
        ],
        temperature=temperature,
        max_tokens=1000
    )
    return response.choices[0].message.content
```

Claude API Call:

```python
import anthropic

client = anthropic.Anthropic(api_key="your-api-key")

def call_claude(prompt, model="claude-3-opus-20240229"):
    response = client.messages.create(
        model=model,
        max_tokens=1024,
        messages=[
            {"role": "user", "content": prompt}
        ]
    )
    return response.content[0].text
```

API Parameters Explained:

Parameter	Purpose	Typical Values
`model`	Which model to use	gpt-4, claude-3-opus

`temperature`	Randomness	0.0 - 1.0
`max_tokens`	Output limit	100 - 4000
`top_p`	Nucleus sampling	0.0 - 1.0
`stop`	Stop sequences	["\n", "END"]

⚠️ Rate Limiting and Error Handling

Rate Limit Handling:

```
import time
from openai import RateLimitError

def call_with_retry(prompt, max_retries=3, base_delay=1):
    for attempt in range(max_retries):
        try:
            return call_llm(prompt)
        except RateLimitError:
            delay = base_delay * (2 ** attempt)  # Exponential backoff
            print(f"Rate limited. Waiting {delay}s...")
            time.sleep(delay)
    raise Exception("Max retries exceeded")
```

Comprehensive Error Handling:

```
from openai import OpenAI, APIError, RateLimitError, Timeout

def robust_api_call(prompt):
    try:
        response = call_llm(prompt)
        return {"success": True, "data": response}

    except RateLimitError:
        return {"success": False, "error": "rate_limit",
                "message": "Too many requests. Retry later."}

    except Timeout:
        return {"success": False, "error": "timeout",
                "message": "Request timed out."}

    except APIError as e:
        return {"success": False, "error": "api_error",
                "message": str(e)}

    except Exception as e:
        return {"success": False, "error": "unknown",
                "message": str(e)}
```

Error Categories:

Error Type	Cause	Solution
429 Rate Limit	Too many requests	Exponential backoff
401 Unauthorized	Invalid API key	Check configuration
400 Bad Request	Invalid parameters	Validate input
500 Server Error	Provider issue	Retry with backoff
Timeout	Slow response	Increase timeout, retry

📦 Batch Processing

Batch Processing Pattern:

```
import asyncio
from typing import List

async def process_batch(prompts: List[str], batch_size=5):
    """Process prompts in batches to respect rate limits."""
    results = []

    for i in range(0, len(prompts), batch_size):
        batch = prompts[i:i+batch_size]

        # Process batch concurrently
        batch_results = await asyncio.gather(
            *[call_llm_async(p) for p in batch]
        )
        results.extend(batch_results)

        # Rate limit pause between batches
        if i + batch_size < len(prompts):
            await asyncio.sleep(1)

    return results
```

Queue-Based Processing:

```
from queue import Queue
from threading import Thread
import time

class PromptProcessor:
    def __init__(self, rate_limit=10):
        self.queue = Queue()
        self.rate_limit = rate_limit
        self.results = {}

    def add_task(self, task_id, prompt):
        self.queue.put((task_id, prompt))
```

```
    def process(self):
        while not self.queue.empty():
            task_id, prompt = self.queue.get()
            result = call_llm(prompt)
            self.results[task_id] = result
            time.sleep(1 / self.rate_limit)  # Respect rate limit

    def get_result(self, task_id):
        return self.results.get(task_id)
```

📊 Monitoring and Logging

Logging Setup:

```
import logging
from datetime import datetime

logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(levelname)s - %(message)s',
    handlers=[
        logging.FileHandler('llm_calls.log'),
        logging.StreamHandler()
    ]
)

logger = logging.getLogger('llm_api')

def logged_api_call(prompt, **kwargs):
    start_time = datetime.now()

    logger.info(f"Request started | Prompt length: {len(prompt)}")

    try:
        response = call_llm(prompt, **kwargs)

        duration = (datetime.now() - start_time).total_seconds()
        logger.info(f"Request completed | Duration: {duration:.2f}s")

        return response

    except Exception as e:
        logger.error(f"Request failed | Error: {str(e)}")
        raise
```

Metrics to Track:

```
class APIMetrics:
    def __init__(self):
```

```
        self.total_requests = 0
        self.successful_requests = 0
        self.failed_requests = 0
        self.total_tokens = 0
        self.total_latency = 0

    def record_request(self, success, tokens, latency):
        self.total_requests += 1
        if success:
            self.successful_requests += 1
        else:
            self.failed_requests += 1
        self.total_tokens += tokens
        self.total_latency += latency

    def get_stats(self):
        return {
            "total_requests": self.total_requests,
            "success_rate": self.successful_requests / self.total_requests,
            "avg_tokens": self.total_tokens / self.total_requests,
            "avg_latency": self.total_latency / self.total_requests
        }
```

Week 16: Capstone Project

🎯 Design Your Solution

Project Planning Template:

```
# Capstone Project: [Project Name]

## Problem Statement
What problem are you solving with prompt engineering?

## Use Case
Who will use this? In what context?

## Requirements
### Functional
- [ ] Requirement 1
- [ ] Requirement 2

### Non-Functional
- [ ] Performance: <2s latency
- [ ] Accuracy: >90%
- [ ] Cost: <$X per 1000 requests

## Prompt Design

### System Prompt
[Your system prompt]
```

```
### User Input Template
[How user inputs will be formatted]

### Output Specification
[Expected output format]

## Evaluation Plan
- Test cases: [number]
- Metrics: [list]
- Success criteria: [thresholds]
```

📝 Implementation Examples

Example 1: Email Classifier System

```
SYSTEM_PROMPT = """
You are an email classification system. Classify incoming
emails into exactly one of these categories:

Categories:
- URGENT: Requires immediate attention (complaints, outages)
- SUPPORT: Technical help requests
- BILLING: Payment, invoice, refund inquiries
- SALES: New business inquiries
- SPAM: Unsolicited promotional content
- OTHER: Doesn't fit other categories

Response format (JSON only):
{
  "category": "[CATEGORY]",
  "confidence": "[HIGH/MEDIUM/LOW]",
  "summary": "[1 sentence summary]",
  "suggested_action": "[recommended next step]"
}
"""

def classify_email(subject, body):
    prompt = f"""
    Classify this email:

    Subject: {subject}
    Body: {body}
    """

    response = call_llm(
        system=SYSTEM_PROMPT,
        user=prompt,
        temperature=0
    )
```

```
    return json.loads(response)
```

Example 2: Code Review Assistant

```
REVIEW_PROMPT = """
You are a senior software engineer conducting code review.

Review this code for:
1. Bugs and potential issues
2. Security vulnerabilities
3. Performance concerns
4. Code style and readability
5. Best practices

<code language="{language}">
{code}
</code>

Format your review as:

## Summary
[Overall assessment in 1-2 sentences]

## Critical Issues 🔴
[List any bugs or security issues]

## Improvements 🟡
[Suggested enhancements]

## Minor Notes 🟢
[Style and minor suggestions]

## Example Fix
[Show corrected code for most important issue]
"""
```

📊 Present and Document

Documentation Template:

```
# [Project Name] - Prompt Engineering Solution

## Overview
Brief description of the solution and its purpose.

## Architecture
[Diagram or description of how prompts flow]

## Prompts Used
```

```
### Main Prompt (v1.2)
```

[Full prompt text]

```
### Fallback Prompt
```

[Alternative prompt]

```
## Performance Metrics

| Metric | Target | Achieved |
|--------|--------|----------|
| Accuracy | 90% | 94% |
| Latency | <2s | 1.3s |
| Cost | $0.05/req | $0.03/req |

## Lessons Learned
1. What worked well
2. What was challenging
3. What you'd do differently

## Future Improvements
1. Planned enhancement 1
2. Planned enhancement 2
```

🎓 Course Completion Checklist

✅ Foundations

- [ ] Understand tokens, context, temperature
- [ ] Write clear, specific instructions

✅ Core Strategies

- [ ] Use structured formatting (XML, JSON)
- [ ] Apply few-shot learning
- [ ] Implement chain-of-thought

✅ Advanced Techniques

- [ ] Design prompt chains
- [ ] Implement RAG patterns
- [ ] Handle edge cases safely

✅ Domain Applications

- [ ] Create content with appropriate tone
- [ ] Generate and review code

- [ ] Build conversational systems

✅ **Evaluation**

- [ ] Define and measure metrics
- [ ] A/B test prompts
- [ ] Version and document prompts

✅ **Production**

- [ ] Integrate with APIs
- [ ] Handle errors gracefully
- [ ] Monitor and optimize

🎯 Final Tips

1. **Start simple** - Add complexity only when needed
2. **Test often** - Catch issues early
3. **Document everything** - Future you will thank you
4. **Stay updated** - Models and best practices evolve
5. **Share knowledge** - Contribute to the community

Congratulations! You've completed the Prompt Engineering Course! 🎉