Chapter 2

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:

   - Inventing non-existent functions or libraries

   - Creating syntactically correct but logically flawed code

   - Confidently stating incorrect technical information

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

   - GPT-4 (as of this writing): ~32,000 tokens (~24,000 words)

   - Claude 2: ~100,000 tokens

   - Gemini Pro: ~32,000 tokens

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

3. **Knowledge Cutoffs**: LLMs are trained on data up to a specific date, after which they have no knowledge:

- GPT-4: Knowledge cutoff in early 2023

- Other models have similar cutoffs

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

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

5. **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:

[PYTHON]

```
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:

[PYTHON]

```
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:

[PYTHON]

```
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

[PYTHON]

```
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:

[PYTHON]

```
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:

[PYTHON]

```
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:

[PYTHON]

```
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:

[PYTHON]

```
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:

[PYTHON]

```
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

[PYTHON]

```
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%
```

3. **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
```

4. **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.