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:
    - Clarify the exact requirements
    - Identify edge cases and constraints
2. System design:
    - Propose an overall architecture
    - Identify key components and their interactions
    - Choose appropriate data structures and algorithms
3. Implementation planning:
    - Break down the solution into implementable units
    - Determine the sequence of implementation
4. Implementation:
    - Write the code with clear comments
    - Explain design choices
5. Testing strategy:
    - Outline test cases covering normal operation and edge cases
    - 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

```python
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:

```python
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:

```python
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:

```python
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.