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:

```python
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

```python
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.
```