Week 2: AI-Assisted Code Generation & Pattern Recognition

Complete Step-by-Step Learning Materials

PRE-WEEK PREPARATION

Required Setup

1. **Cursor IDE**: Ensure latest version installed with GPT-4 enabled
2. **Alternative Tools**: GitHub Copilot, Amazon Q Developer
3. **Project Setup**: Have a sample project ready in your preferred language
4. **Documentation Access**: Bookmark language-specific style guides

LEARNING MODULE 1: Advanced Prompt Patterns

Video Resources

Topic	Video & Link	Duration
The Art of Prompt Engineering for Developers	**https://www.youtube.com/watch?v=Fo-QP_SY9Dc (YouTube)**	**20 min**
GitHub Copilot: Advanced Prompting Techniques (VS Code Team)	**https://www.youtube.com/watch?v=SLMfhuptCo8 (YouTube)**	**15 min**
Cursor AI: Complete Guide for Developers	**https://www.youtube.com/watch?v=gbOpXQAVB3s (YouTube)**	**25 min**
Chain-of-Thought Prompting for Code	**https://www.youtube.com/watch?v=AFE6x81AP4k (YouTube)**	**12 min**

Step-by-Step Tutorial: Mastering Prompt Patterns

Step 1: Basic vs Advanced Prompts

Basic Prompt:

"Create a user authentication function"

Advanced Prompt:

"Create a secure user authentication function that:
1. Accepts email and password
2. Validates input format using regex
3. Handles bcrypt password hashing
4. Returns JWT token on success
5. Includes proper error handling for invalid credentials
6. Follows {LANGUAGE} best practices for async operations
7. Add comprehensive JSDoc/comments

Please implement step by step with explanation."

Step 2: Chain-of-Thought Pattern

Template:

"I need to solve [PROBLEM]. Let me break this down:

Step 1: [First requirement]
Step 2: [Second requirement]
Step 3: [Third requirement]

For each step, please:
- Explain the approach
- Write the code
- Explain potential issues
- Suggest improvements

Start with Step 1."

Example Usage:

"I need to create a caching layer for database queries. Let me break this down:

Step 1: Design the cache interface with get/set/delete operations
Step 2: Implement Redis integration with connection handling
Step 3: Add cache invalidation strategies and TTL management

For each step, please:
- Explain the approach
- Write the code
- Explain potential issues
- Suggest improvements

Start with Step 1."

Step 3: Context-Rich Prompts

Template:

"Context: [Your project context]
Current Architecture: [Brief description]
Technology Stack: [Languages/frameworks]
Constraints: [Performance, security, etc.]

Task: [Specific requirement]
Expected Output: [What you want to achieve]
Quality Criteria: [How to measure success]"

Practice Exercise 1: Prompt Pattern Mastery

Take any simple task (e.g., "create a TODO list API")

1. **Write 3 different prompts**:
    - Basic prompt (1 sentence)
    - Chain-of-thought prompt (structured breakdown)
    - Context-rich prompt (full context + constraints)
2. **Compare outputs** and document which produces better results
3. **Iterate** on the best prompt 2-3 times to refine

LEARNING MODULE 2: Code Architecture with AI

Video Tutorials

Topic	Video & Link	Duration
Building Full-Stack Apps with AI Assistance (Theo – t3.gg)	**https://www.youtube.com/watch?v=3D_CBoo83rU (YouTube)**	**30 min**
***AI-Powered API Design & Development* (closest recent match)**	**https://www.youtube.com/watch?v=-dSKp-salt0 (YouTube)**	**25 min**
Clean Code Architecture with AI Tools (Uncle Bob clip)	**https://www.youtube.com/watch?v=aBfmaVwXBP4 (YouTube)**	**35 min**

Supplementary Resources

- **Martin Fowler's Architecture Patterns** (reference during AI prompting)
- **Language-Specific Architecture Guides** (Spring Boot, Express.js, FastAPI, ASP.NET)

Step-by-Step Tutorial: Architectural AI Prompting

Step 1: System Design Prompts

Template for System Architecture:

"Design a {SYSTEM_TYPE} system with the following requirements:

Functional Requirements:
- [Requirement 1]
- [Requirement 2]
- [Requirement 3]

Non-Functional Requirements:
- Performance: [Specify targets]

- Scalability: [User/data volume]
- Security: [Authentication/authorization needs]

Please provide:
1. High-level architecture diagram (text/ASCII)
2. Component breakdown with responsibilities
3. Database schema design
4. API endpoint structure
5. Technology stack recommendations for {YOUR_TECH_STACK}

Start with the high-level architecture."

Step 2: Class/Module Design Prompts

Template for Code Structure:

"Based on the following business logic, design a well-structured {LANGUAGE} implementation:

Business Requirements:
[Detailed requirements]

Design Constraints:
- Follow SOLID principles
- Use appropriate design patterns
- Include proper error handling
- Add comprehensive interfaces/contracts
- Consider testability

Please provide:
1. Class hierarchy and interfaces
2. Method signatures with documentation
3. Dependency injection setup
4. Error handling strategy
5. Unit test structure outline

Language: {YOUR_LANGUAGE}
Framework: {YOUR_FRAMEWORK}"

Practice Exercise 2: Architecture Design

Scenario: E-commerce Order Management System

1. **System-Level Design** (20 min):
    - Use AI to design overall architecture
    - Get component breakdown and data flow
    - Review and refine the architecture
2. **Code-Level Design** (25 min):
    - Design Order, Customer, Product classes
    - Define interfaces and relationships
    - Generate skeleton code structure

Deliverable: Complete architecture document with code skeleton

LEARNING MODULE 3: Pattern Recognition & Quality Assessment

Video Tutorials

Topic	Video & Link	Duration
Code Review Best Practices with AI Tools (GitHub)	https://www.youtube.com/watch?v=Dgss8IYLk3s (YouTube)	20 min
SOLID Principles in Modern Development (Codevolution)	https://www.youtube.com/watch?v=CubcodsKgSo (YouTube)	30 min
Design Patterns in Practice – When & How (ArjanCodes)	https://www.youtube.com/watch?v=vzTrLpxPF54 (YouTube)	25 min

Step-by-Step Tutorial: Quality Assessment Framework

Step 1: Code Quality Checklist

When AI generates code, evaluate:

SOLID Principles Check

- **S**ingle Responsibility: Does each class/function have one job?
- **O**pen/Closed: Can you extend without modifying?
- **L**iskov Substitution: Are inheritance relationships correct?
- **I**nterface Segregation: Are interfaces focused and minimal?
- **D**ependency Inversion: Does it depend on abstractions?

Code Smell Detection

- Long methods (>20 lines)
- Large classes (>200 lines)
- Duplicate code blocks
- Deep nesting (>3 levels)
- Magic numbers/strings
- Poor variable naming

Pattern Appropriateness

- Is the chosen pattern solving the right problem?
- Is it over-engineering a simple solution?
- Are there simpler alternatives?

Step 2: Pattern Quality Assessment

AI Pattern Evaluation Prompt:

"Please analyze this AI-generated code and evaluate:

[PASTE CODE HERE]

Assessment Criteria:
1. Design Pattern Usage: Is the pattern appropriate? Over-engineered?
2. SOLID Principles: Which principles are followed/violated?
3. Code Smells: Identify potential issues
4. Testability: How easy would this be to unit test?

5. Maintainability: Will this be easy to modify/extend?
6. Performance: Any obvious performance concerns?

Provide:
- Overall quality score (1-10)
- Top 3 strengths
- Top 3 areas for improvement
- Specific refactoring suggestions"

Practice Exercise 3: Code Quality Detective

1. **Generate 3 different implementations** of the same feature using AI
2. **Apply quality checklist** to each implementation
3. **Score each implementation** (1-10) with justification
4. **Create improvement suggestions** for the lowest-scored implementation

LEARNING MODULE 4: Multi-file Context & Consistency

Video Resources

Topic	Video & Link	Duration
Cursor Composer: Codebase Context & Multi-file Editing	https://www.youtube.com/watch?v=V9_RzjqCXP8 (YouTube)	15 min
Managing Large Codebases with AI (TechLead-style demo)	https://www.youtube.com/watch?v=Sm6TC7eDRIM (YouTube)	20 min
GitHub Copilot Workspace: Multi-file AI Development	https://www.youtube.com/watch?v=RZpKFlfioDc (YouTube)	18 min

Step-by-Step Tutorial: Context-Aware Generation

Step 1: Cursor Context Commands

Essential Cursor Commands:

@codebase - Reference entire codebase
@files - Reference specific files
@docs - Reference documentation
@web - Search web for context
@git - Reference git history/changes

Context-Rich Prompt Template:

"Given my current codebase context (@codebase), I need to add a new feature:

Feature: [Description]

Requirements:
1. Follow existing patterns in @files [similar-file.js]
2. Maintain consistency with @files [style-guide.md]
3. Integrate with existing @files [related-component.js]

Please:
1. Analyze existing patterns first
2. Generate code that matches the style
3. Ensure proper integration points
4. Maintain naming conventions
5. Follow established error handling patterns"

Step 2: Consistency Verification

Consistency Check Prompt:

"Compare this new implementation with existing codebase patterns:

New Code: [PASTE HERE]
Reference Files: @files [file1.js, file2.js, file3.js]

Check for consistency in:
1. Naming conventions
2. Function/class structure
3. Error handling approach
4. Import/export patterns
5. Documentation style
6. Testing approach

Identify any inconsistencies and suggest alignments."

Practice Exercise 4: Consistency Master (30 minutes)

1. **Select 3 similar files** from your project
2. **Generate a new component** that should match their patterns
3. **Use context commands** to ensure consistency
4. **Verify consistency** using the checking prompts

LEARNING MODULE 5: Language-Specific Optimization

Video Resources by Language

JavaScript / Node.js

- **AI-Powered JavaScript Development 2024 (Fireship)** – https://www.youtube.com/watch?v=TBIjgBVFjVI YouTube (10 min)
- **Building APIs with AI – Express.js Best Practices** – https://www.youtube.com/watch?v=fhRHGAgvrh4 YouTube (25 min)

Python

- **Python AI Development Workflow 2024 (Real Python)** – https://www.youtube.com/watch?v=nOogLLcOFqI YouTube (30 min)
- **FastAPI with AI Code Generation Tutorial** – https://www.youtube.com/watch?v=lV82LDKT16A YouTube (20 min)

Java

- **Spring Boot Development with AI Assistance (Java Brains)** – https://www.youtube.com/watch?v=k3fSQpz2Esg YouTube (25 min)

React

- **React Development with AI Tools 2024 (Jack Herrington)** – https://www.youtube.com/watch?v=SUxClOpf9Bo YouTube (20 min)

. NET / C#

- **Generative AI in Any .NET App with Semantic Kernel (Nick Chapsas)** – https://www.youtube.com/watch?v=f_hqGlt_2E8 YouTube (30 min)

Language-Specific Prompt Templates

Java/Spring Boot Template

"Create a {FEATURE} implementation following Java/Spring Boot best practices:

Requirements: [Your requirements]

Java-Specific Considerations:
- Use proper annotations (@Service, @Repository, @Component)
- Implement proper exception handling with custom exceptions
- Follow Java naming conventions (PascalCase for classes, camelCase for methods)
- Use Optional<T> for nullable returns
- Implement proper logging with SLF4J
- Add validation annotations where appropriate
- Include proper JavaDoc documentation
- Consider thread safety if applicable

Spring Boot Specific:
- Use dependency injection properly
- Configure properties in application.yml format
- Include proper testing with @SpringBootTest
- Follow RESTful conventions for controllers"

Node.js/Express Template

"Create a {FEATURE} implementation following Node.js/Express best practices:

Requirements: [Your requirements]

Node.js-Specific Considerations:
- Use async/await for asynchronous operations
- Implement proper error handling middleware
- Follow JavaScript naming conventions (camelCase)
- Use const/let appropriately (no var)
- Implement proper input validation with joi or similar
- Include comprehensive JSDoc comments
- Use environment variables for configuration
- Handle promises properly (no callback hell)

Express-Specific:

- Use proper middleware structure
- Implement RESTful routing
- Add proper HTTP status codes
- Include request/response validation
- Use proper error handling middleware
- Implement proper logging with winston or similar"

Python/FastAPI Template

"Create a {FEATURE} implementation following Python/FastAPI best practices:

Requirements: [Your requirements]

Python-Specific Considerations:
- Follow PEP 8 style guidelines
- Use type hints throughout
- Implement proper error handling with custom exceptions
- Use dataclasses or Pydantic models for data structures
- Include comprehensive docstrings (Google/NumPy style)
- Use context managers where appropriate
- Follow Python naming conventions (snake_case)
- Use list comprehensions and generators appropriately

FastAPI-Specific:
- Use Pydantic models for request/response validation
- Implement proper dependency injection with Depends()
- Add comprehensive OpenAPI documentation
- Use proper HTTP status codes and responses
- Implement proper error handling with HTTPException
- Include proper async/await usage"

Practice Exercise 5: Language Mastery (45 minutes)

1. **Choose your primary language** from the templates above
2. **Implement a complete CRUD API** using the language-specific prompt
3. **Review the output** against the best practices checklist
4. **Refine 2-3 times** to perfect the implementation

HANDS-ON TASKS DETAILED GUIDE

Task 1: Micro-Service Skeleton Challenge

Objective: Generate a complete REST API structure

Step-by-Step Process:
Define Requirements (5 min):

Service: User Management API
Endpoints: CRUD operations for users
Database: PostgreSQL
Authentication: JWT
Testing: Unit tests included

1. **Craft Architecture Prompt** (5 min):

   "Create a complete User Management microservice with:

Architecture:
- RESTful API with proper HTTP methods
- Database layer with connection pooling
- Service layer for business logic
- Controller layer for HTTP handling
- Authentication middleware
- Error handling middleware
- Input validation
- Logging system

Technology: {YOUR_STACK}
Database: PostgreSQL

Include:
1. Project structure
2. Dependencies/package configuration
3. Database schema
4. All endpoint implementations
5. Authentication setup
6. Error handling

7. Basic unit tests
8. README with setup instructions"

2. **Generate and Review** (15 min):
    - Generate the skeleton
    - Review against quality checklist
    - Test compilation/syntax
3. **Document Results** (5 min):
    - Note what worked well
    - Identify gaps or issues
    - Record prompt refinements needed

Success Criteria:

- Complete project structure
- All CRUD endpoints implemented
- Database integration working
- Authentication implemented
- Error handling present
- Tests included

Task 2: Design Pattern Implementation

Objective: Implement and evaluate 3 design patterns

Patterns to Implement:

1. **Strategy Pattern**: Payment processing (Credit Card, PayPal, Bank Transfer)
2. **Observer Pattern**: Event notification system
3. **Factory Pattern**: Database connection factory

Implementation Process:

For Each Pattern (15 min each):

Generate Implementation (8 min):

"Implement the {PATTERN_NAME} pattern for {USE_CASE}:

Context: [Specific business context]
Requirements: [Functional requirements]

Please provide:
1. Interface/abstract class definitions
2. Concrete implementations
3. Client code example
4. Unit tests
5. Explanation of why this pattern fits

Language: {YOUR_LANGUAGE}
Follow {LANGUAGE} best practices and naming conventions."

1. **Quality Assessment** (5 min):
    - Apply quality checklist
    - Check pattern appropriateness
    - Verify SOLID principles
2. **Document Evaluation** (2 min):
    - Pattern fit: Good/Poor and why
    - Code quality: Score 1-10
    - Improvements needed

Comparison Analysis (10 min):

- Which pattern implementation was best?
- What made it better?
- How could prompt be improved for lower-quality outputs?

Task 3: Legacy Code Modernization

Objective: Transform legacy code using modern practices

Process:

1. **Select Legacy Code** (5 min):
    - Find a 50+ line function in your codebase
    - Or use provided legacy examples

Analysis Prompt (10 min):

"Analyze this legacy code and identify modernization opportunities:

[PASTE LEGACY CODE]

Please identify:
1. Code smells and anti-patterns
2. Security vulnerabilities
3. Performance issues
4. Maintainability problems
5. Missing error handling
6. Outdated language features

Provide specific examples and explanations."

2. **Modernization Prompt** (15 min):

    "Modernize this legacy code following current best practices:

[PASTE LEGACY CODE]

Modernization Requirements:
- Use modern {LANGUAGE} features
- Implement proper error handling
- Add input validation
- Improve readability and maintainability

- Add comprehensive documentation
- Include unit tests
- Follow SOLID principles
- Address security concerns

Provide:
1. Refactored code
2. Explanation of changes made
3. Migration strategy
4. Testing approach"

3. **Before/After Comparison** (10 min):
    - Code quality improvement
    - Maintainability gains
    - Performance implications
    - Testing coverage

Task 4: Cross-Language Translation

Objective: Convert functionality between programming languages

Example Scenario: Convert a Python data processing function to Java

Process:

1. **Source Code Selection** (5 min):
    - Choose a moderately complex function (30-50 lines)
    - Should include: loops, conditionals, data structures, error handling

Translation Prompt (15 min):

"Convert this {SOURCE_LANGUAGE} function to {TARGET_LANGUAGE}:

[PASTE SOURCE CODE]

Translation Requirements:
1. Maintain exact business logic
2. Use idiomatic {TARGET_LANGUAGE} patterns
3. Follow {TARGET_LANGUAGE} naming conventions

4. Use appropriate data structures for {TARGET_LANGUAGE}
5. Implement proper error handling in {TARGET_LANGUAGE} style
6. Add appropriate documentation/comments
7. Include type annotations where applicable
8. Optimize for {TARGET_LANGUAGE} performance patterns

Provide:
- Converted function
- Explanation of language-specific adaptations
- Unit tests in {TARGET_LANGUAGE}
- Performance considerations"

2. **Verification** (10 min):
    - Logic equivalence check
    - Language idiom appropriateness
    - Performance characteristics
3. **Documentation** (5 min):
    - Translation challenges faced
    - Language-specific optimizations made
    - Potential issues to watch for

Task 5: Real Project Integration

Objective: Generate 70% of a real feature implementation

Feature Selection Criteria:

- Medium complexity (3-5 files involved)
- Clear business requirements
- Integrates with existing codebase
- Has testable outcomes

Implementation Process:
Feature Analysis (10 min):

"Analyze this feature requirement for implementation:

Feature: [DETAILED_DESCRIPTION]
Existing Codebase Context: @codebase
Integration Points: @files [relevant-files]

Please provide:
1. Implementation approach
2. Files that need modification
3. New files to create
4. Database changes required
5. API contract changes
6. Testing strategy
7. Potential risks and mitigation
8. Estimated effort breakdown"

1.

Code Generation (30 min):

For Each Component (Generate separately):

"Implement {COMPONENT_NAME} for the {FEATURE_NAME} feature:

Context: @codebase
Integration: @files [related-files]

Requirements:
[SPECIFIC_COMPONENT_REQUIREMENTS]

Ensure:
- Follows existing codebase patterns
- Maintains consistency with @files [style-files]
- Proper error handling
- Comprehensive logging
- Input validation
- Unit tests included
- Documentation added"

2. **Integration Testing** (15 min):
    - Compile/syntax check
    - Integration points verification

- Basic functionality testing

3. **Quality Review** (5 min):
    - Code quality assessment
    - Pattern consistency check
    - Documentation completeness

Week 3 - Test-Driven Development with AI

Module 1 - Understanding TDD + AI Foundation

What is Test-Driven Development? (The Simple Explanation)

Think of TDD like building with Lego blocks, but you **describe what you want to build before you start building it**.

Traditional Development (Risky):

```
Write Code → Hope It Works → Fix Bugs → Repeat
```

TDD with AI (Safe & Fast):

```
Describe What You Want (Test) → AI Helps Build It → Verify It Works → Improve
```

The Red-Green-Refactor Cycle (Your New Best Friend)

🔴 **RED Phase**: Write a failing test (it should fail because the feature doesn't exist yet)

🟢 **GREEN Phase**: Write the simplest code possible to make the test pass

🔵 **REFACTOR Phase**: Clean up the code while keeping tests green

Real Example: Building a Simple Calculator

Let's walk through a complete example that you can follow step-by-step:

Step 1: RED - Write the Failing Test First

Prompt for your AI assistant:

```
I want to build a calculator function that can add two numbers.
Create a failing test using Jest that:
- Tests adding 2 + 3 = 5
- Tests adding negative numbers
- Tests adding decimal numbers
- Uses clear, descriptive test names
```

AI generates this test:

javascript

```javascript
// calculator.test.js
describe('Calculator', () => {
  test('should add two positive numbers correctly', () => {
    const result = add(2, 3);
    expect(result).toBe(5);
  });

  test('should add negative numbers correctly', () => {
    const result = add(-2, -3);
    expect(result).toBe(-5);
  });

  test('should add decimal numbers correctly', () => {
    const result = add(1.5, 2.7);
    expect(result).toBeCloseTo(4.2);
  });

  test('should handle adding positive and negative numbers', () => {
    const result = add(5, -3);
    expect(result).toBe(2);
  });

});
```

Run the test: It fails because `add` function doesn't exist yet.

Step 2: GREEN - Make It Pass

Prompt for AI:

```
Create the simplest add function that will make these tests pass.
Don't over-engineer it, just make it work.
```

AI generates:

javascript
```javascript
// calculator.js
function add(a, b) {
  return a + b;
}

module.exports = { add };
```

Run the test: All tests pass! Green phase complete.

Step 3: REFACTOR - Improve the Code

Prompt for AI:

```
Look at this add function and tests. Can you improve the code quality
while keeping all tests passing? Consider:
- Input validation
- Error handling
- Code documentation
```

AI improves it:

javascript
```javascript
// calculator.js
/**
 * Adds two numbers together
 * @param {number} a - First number
 * @param {number} b - Second number
 * @returns {number} Sum of a and b
 * @throws {Error} If inputs are not numbers
 */
```

```
function add(a, b) {
  if (typeof a !== 'number' || typeof b !== 'number') {
    throw new Error('Both arguments must be numbers');
  }

  if (isNaN(a) || isNaN(b)) {
    throw new Error('Arguments cannot be NaN');
  }

  return a + b;
}


module.exports = { add };
```

But wait! Now we need more tests for error cases:

Prompt for AI:

```
Add tests for the error handling I just added to the add function.
```

AI adds:

javascript

```
// Additional tests
test('should throw error when first argument is not a number', () =>
{
  expect(() => add('5', 3)).toThrow('Both arguments must be
numbers');
});

test('should throw error when second argument is not a number', () =>
{
  expect(() => add(5, '3')).toThrow('Both arguments must be
numbers');
});

test('should throw error when arguments are NaN', () => {
  expect(() => add(NaN, 5)).toThrow('Arguments cannot be NaN');
});
```

Module 2 - Test Case Generation Strategies

1. Unit Tests

Prompt template:

```
Generate unit tests for [function name] that:
- Tests all public methods
- Covers success and failure scenarios
- Includes boundary value testing
- Uses [framework] syntax
- Follows AAA pattern (Arrange, Act, Assert)
```

Integration Tests

Prompt template:

```
Create integration tests for [component/service] that:
- Tests interaction with external dependencies
- Includes database operations
- Tests API endpoints
- Handles network failures
- Uses realistic test data
```

2. Edge Case Discovery with AI

The Secret Sauce: AI is incredibly good at thinking of edge cases you'd never consider.

Prompt for comprehensive edge case discovery:

```
Analyze this function and generate tests for every possible edge case.

Think like a hacker trying to break the system:
```

```
[paste your function]


Consider:
- Null/undefined/empty inputs
- Extremely large or small numbers
- Special characters and Unicode
- Concurrent access scenarios
- Memory limitations
- Network timeouts
- Invalid data types
- Boundary conditions
- Race conditions
- Security vulnerabilities
```

3. Mock and Stub Creation

AI prompt for mocks:

```
Create Jest mocks for these dependencies:
- Database connection
- External API calls
- File system operations
- Email service

Include realistic return values and error scenarios.
```

Generated example:

javascript
```
// AI-generated mock setup
jest.mock('../database', () => ({
  findUser: jest.fn(),
  createUser: jest.fn(),
}));

jest.mock('../emailService', () => ({
  sendWelcomeEmail: jest.fn().mockResolvedValue({ messageId: '12345'
}),
}));

const mockDatabase = require('../database');
const mockEmailService = require('../emailService');
```

4. Test Documentation

Prompt for self-documenting tests:

```
Rewrite these tests to be self-documenting with:
- Clear, descriptive test names
- Meaningful variable names
- Comments explaining business logic
- Given-When-Then structure
```

Before:

javascript
```
test('login test', () => {
  const result = login('email', 'pass');
  expect(result).toBeTruthy();
});
```

After (AI-improved):

javascript
```
test('should successfully authenticate user with valid email and
password', () => {
  // Given: A user with valid credentials
```

```
  const validEmail = 'john.doe@company.com';
  const validPassword = 'SecurePass123!';

  // When: User attempts to login
  const authResult = login(validEmail, validPassword);

  // Then: Authentication succeeds and returns token
  expect(authResult).toHaveProperty('token');

expect(authResult.token).toMatch(/^[A-Za-z0-9-_=]+\.[A-Za-z0-9-_=]+\.
[A-Za-z0-9-_.+/=]*$/);
});
```

Module - 3 The "Legacy Code Challenge"

"This all sounds great for new projects, but what about our existing 100,000+ line codebase with zero tests?"

The Solution: A proven, gradual approach that works in any company.

Phase 1: Assessment and Quick Wins

Step 1: Codebase Analysis with AI

Prompt for codebase analysis:

```
I'm going to paste sections of our existing codebase.
For each section, analyze and provide:

1. Risk Assessment (High/Medium/Low risk if this breaks)
2. Test Complexity (Easy/Medium/Hard to test)
3. Business Impact (Critical/Important/Nice-to-have)
4. Recommended Testing Strategy
5. Priority Score (1-10)

[Paste your code sections]
```

Real Example Analysis:

javascript

```javascript
// Existing legacy function
function processPayment(userId, amount, cardToken, options) {
  const user = database.getUser(userId);
  if (!user) throw new Error('User not found');

  const card = paymentGateway.validateCard(cardToken);
  if (!card.valid) throw new Error('Invalid card');

  const fee = calculateFee(amount, user.tier);
  const total = amount + fee;

  if (user.balance < total) {
    throw new Error('Insufficient funds');
  }

  const transaction = paymentGateway.charge(cardToken, total);
  user.balance -= total;
  database.updateUser(user);

  emailService.sendReceipt(user.email, transaction);
  return transaction;
}
```

AI Analysis Result:

```
ANALYSIS RESULTS:
1. Risk Assessment: HIGH - Handles money transactions
2. Test Complexity: HARD - Multiple external dependencies
3. Business Impact: CRITICAL - Revenue affecting
4. Testing Strategy: Wrapper + Integration tests
5. Priority Score: 10/10 - TEST IMMEDIATELY

RECOMMENDED APPROACH:
- Create wrapper function for dependency injection
- Mock all external services
- Test all error conditions
- Add integration tests for happy path
- Monitor in production during rollout
```

Step 2: The "Wrapper Strategy" - Making Untestable Code Testable

Instead of rewriting everything, wrap it:

AI Prompt:

```
Take this legacy function and create a testable wrapper version that:
- Accepts dependencies as parameters
- Maintains backward compatibility
- Allows easy mocking
- Preserves existing functionality

[paste legacy function]
```

AI generates testable wrapper:

javascript

```javascript
// New testable version
function processPaymentWithDeps(userId, amount, cardToken, options =
{}, deps = {}) {
  // Dependency injection with defaults
  const {
    database = require('./database'),
    paymentGateway = require('./paymentGateway'),
    emailService = require('./emailService'),
    calculateFee = require('./feeCalculator')
  } = deps;

  const user = database.getUser(userId);
  if (!user) throw new Error('User not found');

  const card = paymentGateway.validateCard(cardToken);
  if (!card.valid) throw new Error('Invalid card');

  const fee = calculateFee(amount, user.tier);
  const total = amount + fee;

  if (user.balance < total) {
    throw new Error('Insufficient funds');
```

```
  }

  const transaction = paymentGateway.charge(cardToken, total);
  user.balance -= total;
  database.updateUser(user);

  emailService.sendReceipt(user.email, transaction);
  return transaction;
}

// Maintain backward compatibility
function processPayment(userId, amount, cardToken, options) {
  return processPaymentWithDeps(userId, amount, cardToken, options);
}

module.exports = { processPayment, processPaymentWithDeps };
```

Now create comprehensive tests:

javascript

```
// processPayment.test.js
describe('Payment Processing', () => {
  let mockDeps;

  beforeEach(() => {
    mockDeps = {
      database: {
        getUser: jest.fn(),
        updateUser: jest.fn()
      },
      paymentGateway: {
        validateCard: jest.fn(),
        charge: jest.fn()
      },
      emailService: {
        sendReceipt: jest.fn()
      },
      calculateFee: jest.fn()
    };
  });
```

```
  test('should process payment successfully for valid inputs', async
() => {
    // Arrange
    const mockUser = { id: 1, balance: 1000, tier: 'premium', email:
'test@example.com' };
    const mockTransaction = { id: 'tx_123', amount: 105 };

    mockDeps.database.getUser.mockReturnValue(mockUser);
    mockDeps.paymentGateway.validateCard.mockReturnValue({ valid:
true });
    mockDeps.calculateFee.mockReturnValue(5);
    mockDeps.paymentGateway.charge.mockReturnValue(mockTransaction);

    // Act
    const result = await processPaymentWithDeps(1, 100, 'card_123',
{}, mockDeps);

    // Assert
    expect(result).toEqual(mockTransaction);
    expect(mockDeps.database.updateUser).toHaveBeenCalledWith({
      ...mockUser,
      balance: 895 // 1000 - 105
    });
    expect(mockDeps.emailService.sendReceipt).toHaveBeenCalledWith(
      'test@example.com',
      mockTransaction
    );
  });

  test('should throw error for insufficient funds', async () => {
    // Arrange
    const mockUser = { id: 1, balance: 50, tier: 'basic' };

    mockDeps.database.getUser.mockReturnValue(mockUser);
    mockDeps.paymentGateway.validateCard.mockReturnValue({ valid:
true });
    mockDeps.calculateFee.mockReturnValue(5);

    // Act & Assert
```

```
    await expect(processPaymentWithDeps(1, 100, 'card_123', {},
mockDeps))
      .rejects.toThrow('Insufficient funds');

    // Verify no side effects occurred
    expect(mockDeps.paymentGateway.charge).not.toHaveBeenCalled();
    expect(mockDeps.database.updateUser).not.toHaveBeenCalled();
    expect(mockDeps.emailService.sendReceipt).not.toHaveBeenCalled();
  });
});
```

Phase 2: The "Safety Net" Strategy

Step 1: Characterization Tests

What are Characterization Tests? Tests that capture the current behavior of legacy code, even if that behavior has bugs.

AI Prompt for characterization tests:

```
Create characterization tests for this legacy function.
Don't worry about whether the behavior is correct - just capture
exactly what it does now.
Include all edge cases and weird behaviors you can find.

[paste legacy function]
```

Example characterization test:

javascript

```
// Captures current behavior, bugs and all
describe('Legacy Function Characterization', () => {
  test('returns null when input is empty string (current behavior)',
() => {
    // This might be a bug, but it's current behavior
    expect(legacyFunction('')).toBe(null);
  });

  test('throws TypeError when passed undefined (current behavior)',
() => {
```

```
    expect(() => legacyFunction(undefined)).toThrow(TypeError);
  });

  test('rounds down decimal inputs (undocumented behavior)', () => {
    expect(legacyFunction(3.7)).toBe(3);
  });
});
```

Step 2: Gradual Refactoring with Test Coverage

Once you have characterization tests, you can safely refactor:

AI Prompt:

```
I have these characterization tests that pass for my legacy function.
Help me refactor the function to be cleaner while keeping all tests
passing.
If any current behavior seems like a bug, create additional tests for
the corrected behavior.

[paste tests and function]
```

Module 4 - Tool-Specific Implementation

Using Cursor IDE

1. TDD Workflow in Cursor:

- Use `Ctrl+K` to generate tests: "Generate unit tests for this function"
- Use `Ctrl+L` for chat: "Help me implement TDD for this feature"
- Use `Ctrl+I` for inline edits: "Add error handling tests"

2. Cursor-specific prompts:

```
@codebase Generate comprehensive tests for the user authentication
module, including edge cases and mocks for external dependencies.
```

Using GitHub Copilot

1. Comment-driven test generation:

javascript
```
// Generate tests for user registration with email validation, password strength, and duplicate checking
// TODO: Add tests for edge cases and error handling
```

2. Copilot Chat prompts:

```
/tests Generate unit tests for the UserService class with Jest, including mocks for database operations
```

Using Amazon Q Developer

1. Q Chat for TDD:

```
Q: I need to implement TDD for this payment processing function. Generate failing tests first, then minimal implementation.

[Paste function signature]
```

2. Q Code Suggestions:

javascript
```
// Type 'test' and let Q suggest comprehensive test patterns
test('should handle payment processing with', // Q will suggest completions
```

Assignment Tasks To Try

Task 1: TDD Sprint

Objective: Build a complete feature using AI-assisted TDD

Steps:

1. Choose a feature (e.g., shopping cart, user profile, file upload)
2. Use AI to generate comprehensive failing tests
3. Implement minimal code to pass tests
4. Refactor with AI assistance
5. Add integration tests
6. Measure coverage and quality

Success Criteria:

- 90%+ test coverage
- All tests pass
- Code is maintainable and readable

Task 2: Legacy Code Test Coverage

Objective: Add tests to existing untested code

Steps:

1. Identify a legacy function/class
2. Use AI to analyze and generate test plan
3. Create wrapper or facade if needed
4. Generate comprehensive test suite
5. Refactor original code safely

Success Criteria:

- 85%+ coverage improvement
- No breaking changes

- Improved code quality metrics

Task 3: Edge Case Discovery

Objective: Use AI to find and test edge cases

Steps:

1. Select critical business logic
2. Ask AI to identify potential edge cases
3. Generate tests for each edge case
4. Implement handling for discovered issues
5. Document edge case scenarios

Success Criteria:

- 10+ edge cases identified and tested
- Robust error handling implemented
- Clear documentation of edge cases

Assessment and Metrics

Quality Metrics to Track

1. Coverage Metrics

- Line coverage: Target 85%+
- Branch coverage: Target 80%+
- Function coverage: Target 90%+

2. Test Quality Metrics

- Test readability score (use AI to assess)
- Test maintainability index
- Test execution time

3. Development Metrics

- Time to write tests (should decrease 60%)
- Bugs caught in testing vs production
- Developer satisfaction with testing workflow