lab-04-testing-documentation-workflow

- Lab 4: Testing, Documentation & Workflow with GitHub Copilot
  - Overview
  - Prerequisites
  - Part 1: Comprehensive Test Generation (5 minutes)
    - Scenario: Increase Test Coverage
    - 1.1 Generate Unit Tests for a Method
    - 1.2 Generate Integration Tests
    - 1.3 Run Complete Test Suite
  - Part 2: Generate Documentation (3 minutes)
    - Scenario: Document Your API
    - 2.1 Add XML Documentation to Classes
    - 2.2 Generate API Documentation (README)
    - 4.2 Review and Refine
  - Key Learning Points
    - ✅ Testing Best Practices
    - ✅ Documentation Efficiency
    - ✅ Version Control Quality
    - ✅ Code Review Preparation
  - Extension Exercises (If Time Permits)
    - Exercise 1: Generate CHANGELOG.md
    - Exercise 2: Create Contributing Guidelines
    - Exercise 3: API Client SDK Documentation
  - Success Criteria
  - Workshop Wrap-Up
    - ✅ Test-Driven Development (Lab 1)
    - ✅ Requirements to Code (Lab 2)
    - ✅ Code Generation & Refactoring (Lab 3)
    - ✅ Testing, Documentation & Workflow (Lab 4)
  - Troubleshooting
    - /tests Generates Incomplete Tests
    - /doc Generates Generic Comments
    - Commit Message Too Generic
    - PR Description Missing Details
  - Next Steps Beyond Workshop
    - Apply to Real Projects
    - Advanced Copilot Usage
    - Continue Learning
  - Additional Resources

Lab 4: Testing, Documentation & Workflow with GitHub Copilot

Duration: 15 minutes
Learning Objectives:

- Generate comprehensive test suites using `/tests` command
- Create documentation with `/doc` command
- Write Conventional Commit messages with AI assistance

- Draft PR descriptions using `@workspace` for full context
- Integrate Copilot into complete development workflow

Overview

This lab brings together everything you've learned by focusing on the "glue" activities that complete the development lifecycle:

1. **Testing** - Generate comprehensive test coverage
2. **Documentation** - Create clear, maintainable docs
3. **Version Control** - Write meaningful commit messages
4. **Code Review** - Prepare thorough PR descriptions

These activities are often rushed or skipped, but Copilot makes them fast and consistent.

Prerequisites

- ✅ Completed Labs 1, 2, and 3
- ✅ Working Task Manager API with CRUD operations
- ✅ Git initialized with commits from previous labs
- ✅ Familiar with all Copilot features (chat, inline chat, slash commands, context variables)

Part 1: Comprehensive Test Generation (5 minutes)

Scenario: Increase Test Coverage

You have basic tests from TDD, but need comprehensive coverage including edge cases, integration tests, and error scenarios.

1.1 Generate Unit Tests for a Method

Step 1: Select Target Method

Open `src/TaskManager.Application/Commands/CreateTaskCommandHandler.cs` and select the `HandleAsync` method.

Step 2: Use /tests Command

With the method selected, open Copilot Chat (`Ctrl+Alt+I` / `Cmd+Shift+I`) and enter:

```
/tests
```

Or use Inline Chat (`Ctrl+I` / `Cmd+I`):

```
/tests
```

Step 3: Review Generated Tests

Copilot should generate comprehensive tests covering:

```
namespace TaskManager.UnitTests.Commands.CreateTaskCommandHandlerTests;

public sealed class HandleAsyncTests
{
    private readonly ITaskRepository _repository;
    private readonly ILogger<CreateTaskCommandHandler> _logger;
    private readonly CreateTaskCommandHandler _sut;

    public HandleAsyncTests()
    {
        _repository = A.Fake<ITaskRepository>();
        _logger = A.Fake<ILogger<CreateTaskCommandHandler>>();
        _sut = new CreateTaskCommandHandler(_repository, _logger);
    }

    [Fact]
    public async Task HandleAsync_WithValidCommand_CreatesTask()
    {
        // Arrange
        var command = new CreateTaskCommand
        {
            Title = "Test Task",
            Description = "Test Description",
            Priority = "High",
            DueDate = DateTime.UtcNow.AddDays(7)
        };

        // Act
        var result = await _sut.HandleAsync(command);

        // Assert
        Assert.NotNull(result);
        Assert.Equal("Test Task", result.Title);
        Assert.Equal("High", result.Priority.Name);
        A.CallTo(() => _repository.AddAsync(
            A<Domain.Entities.Task>._,
            A<CancellationToken>._))
            .MustHaveHappenedOnceExactly();
    }

    [Fact]
    public async Task
          HandleAsync_WithNullCommand_ThrowsArgumentNullException()
    {
        // Act & Assert
        await Assert.ThrowsAsync<ArgumentNullException>(() =>
            _sut.HandleAsync(null!));
    }

    [Theory]
    [InlineData("")]
    [InlineData("   ")]
    public async Task
          HandleAsync_WithInvalidTitle_ThrowsArgumentException(string
          invalidTitle)
    {
        // Arrange
        var command = new CreateTaskCommand
        {
            Title = invalidTitle,
            Priority = "Medium"
        };

        // Act & Assert
        await Assert.ThrowsAsync<ArgumentException>(() =>
            _sut.HandleAsync(command));
    }

    [Theory]
    [InlineData("InvalidPriority")]
    [InlineData("SuperUrgent")]
    [InlineData("")]
    public async Task
          HandleAsync_WithInvalidPriority_ThrowsArgumentException(string
          invalidPriority)
    {
        // Arrange
        var command = new CreateTaskCommand
        {
            Title = "Test Task",
            Priority = invalidPriority
        };

        // Act & Assert
        await Assert.ThrowsAsync<ArgumentException>(() =>
            _sut.HandleAsync(command));
    }

    [Fact]
    public async Task HandleAsync_WithPastDueDate_ThrowsArgumentException()
    {
        // Arrange
        var command = new CreateTaskCommand
        {
            Title = "Test Task",
            Priority = "Low",
            DueDate = DateTime.UtcNow.AddDays(-1)
        };

        // Act & Assert
        await Assert.ThrowsAsync<ArgumentException>(() =>
            _sut.HandleAsync(command));
    }

    [Fact]
    public async Task HandleAsync_WithNullDueDate_CreatesTaskSuccessfully()
    {
        // Arrange
        var command = new CreateTaskCommand
        {
            Title = "Test Task",
            Priority = "Medium",
            DueDate = null
        };

        // Act
        var result = await _sut.HandleAsync(command);

        // Assert
        Assert.NotNull(result);
        Assert.Null(result.DueDate);
    }

    [Fact]
    public async Task HandleAsync_LogsInformationMessages()
    {
        // Arrange
        var command = new CreateTaskCommand
        {
            Title = "Test Task",
            Priority = "Critical"
        };

        // Act
        await _sut.HandleAsync(command);

        // Assert
        A.CallTo(_logger)
            .Where(call =>
                call.Method.Name == "Log" &&
                call.GetArgument<LogLevel>(0) == LogLevel.Information)
            .MustHaveHappened(Repeated.AtLeast.Twice);
    }
    [Fact]
    public async Task HandleAsync_RespectsCancellationToken()
    {
        // Arrange
        var command = new CreateTaskCommand
        {
            Title = "Test Task",
            Priority = "Low"
        };
        var cts = new CancellationTokenSource();
        cts.Cancel();

        // Act & Assert
        await Assert.ThrowsAnyAsync<OperationCanceledException>(() =>
            _sut.HandleAsync(command, cts.Token));
    }
}
```

1.2 Generate Integration Tests

For API endpoints, generate integration tests:

```
Create integration tests for the POST /tasks endpoint using
WebApplicationFactory. Test:
- Valid request returns 201 Created
- Invalid priority returns 400 Bad Request with ProblemDetails
- Past due date returns 400 Bad Request
- Missing required fields returns 400 Bad Request
- Response body contains all expected fields
Use xUnit and realistic test data
```

Expected Output - `tests/TaskManager.IntegrationTests/Api/TaskEndpointsTests.cs`:

```
namespace TaskManager.IntegrationTests.Api;

public sealed class TaskEndpointsTests :
          IClassFixture<WebApplicationFactory<Program>>
{
    private readonly HttpClient _client;

    public TaskEndpointsTests(WebApplicationFactory<Program> factory)
    {
        _client = factory.CreateClient();
    }

    [Fact]
    public async Task PostTask_WithValidRequest_Returns201Created()
    {
        // Arrange
        var request = new
        {
            title = "Integration Test Task",
            description = "Testing POST endpoint",
            priority = "High",
            dueDate = DateTime.UtcNow.AddDays(7)
        };

        // Act
        var response = await _client.PostAsJsonAsync("/tasks", request);

        // Assert
        response.EnsureSuccessStatusCode();
        Assert.Equal(HttpStatusCode.Created, response.StatusCode);

        var task = await response.Content.ReadFromJsonAsync<TaskResponse>();
        Assert.NotNull(task);
        Assert.Equal("Integration Test Task", task.Title);
        Assert.Equal("High", task.Priority);
    }

    [Theory]
    [InlineData("InvalidPriority")]
    [InlineData("")]
    [InlineData("SuperCritical")]
    public async Task
          PostTask_WithInvalidPriority_Returns400BadRequest(string
          invalidPriority)
    {
        // Arrange
        var request = new
        {
            title = "Test Task",
            priority = invalidPriority,
            dueDate = DateTime.UtcNow.AddDays(1)
        };

        // Act
        var response = await _client.PostAsJsonAsync("/tasks", request);

        // Assert
        Assert.Equal(HttpStatusCode.BadRequest, response.StatusCode);

        var problem = await
          response.Content.ReadFromJsonAsync<ProblemDetails>();
        Assert.NotNull(problem);
        Assert.Contains("priority", problem.Detail,
          StringComparison.OrdinalIgnoreCase);
    }
    [Fact]
    public async Task PostTask_WithPastDueDate_Returns400BadRequest()
    {
        // Arrange
        var request = new
        {
            title = "Test Task",
            priority = "Medium",
            dueDate = DateTime.UtcNow.AddDays(-7)
        };

        // Act
        var response = await _client.PostAsJsonAsync("/tasks", request);

        // Assert
        Assert.Equal(HttpStatusCode.BadRequest, response.StatusCode);

        var problem = await
          response.Content.ReadFromJsonAsync<ProblemDetails>();
        Assert.NotNull(problem);
        Assert.Contains("due date", problem.Detail,
          StringComparison.OrdinalIgnoreCase);
    }
}
```

1.3 Run Complete Test Suite

```
dotnet test --logger "console;verbosity=detailed"
```

Verify coverage includes:

- ✅ Happy path tests
- ✅ Guard clause tests
- ✅ Edge case tests (nulls, empty strings, boundaries)
- ✅ Validation tests
- ✅ Error handling tests
- ✅ Cancellation token tests
- ✅ Logging verification tests
- ✅ Integration tests

Part 2: Generate Documentation (3 minutes)

Scenario: Document Your API

Create comprehensive documentation for the Task Manager API.

2.1 Add XML Documentation to Classes

Step 1: Select a Class

Open `src/TaskManager.Application/Commands/CreateTaskCommandHandler.cs` and select the class declaration.

Step 2: Use /doc Command

With the class selected, use Inline Chat (`Ctrl+I`):

```
/doc
```

Expected Output:

```
/// <summary>
/// Handles the creation of new tasks in the task management system.
/// Validates input, creates task entities, and persists them to the repository.
/// </summary>
public sealed class CreateTaskCommandHandler
{
    private readonly ITaskRepository _repository;
    private readonly ILogger<CreateTaskCommandHandler> _logger;

    /// <summary>
    /// Initializes a new instance of the <see cref="CreateTaskCommandHandler"/> class.
    /// </summary>
    /// <param name="repository">The repository for persisting task entities.</param>
    /// <param name="logger">The logger for structured logging.</param>
    /// <exception cref="ArgumentNullException">
    /// Thrown when <paramref name="repository"/> or <paramref name="logger"/> is null.
    /// </exception>
    public CreateTaskCommandHandler(
        ITaskRepository repository,
        ILogger<CreateTaskCommandHandler> logger)
    {
        _repository = repository ?? throw new ArgumentNullException(nameof(repository));
        _logger = logger ?? throw new ArgumentNullException(nameof(logger));
    }

    /// <summary>
    /// Handles the creation of a new task based on the provided command.
    /// </summary>
    /// <param name="command">The command containing task creation details.</
          param>
    /// <param name="cancellationToken">Token to cancel the asynchronous
          operation.</param>
    /// <returns>The newly created task entity.</returns>
    /// <exception cref="ArgumentNullException">
    /// Thrown when <paramref name="command"/> is null.
    /// </exception>
    /// <exception cref="ArgumentException">
    /// Thrown when command contains invalid data (empty title, invalid
          priority, past due date).
    /// </exception>
    public async Task<Domain.Entities.Task> HandleAsync(
        CreateTaskCommand command,
        CancellationToken cancellationToken = default)
    {
        // ... implementation ...
    }
}
```

2.2 Generate API Documentation (README)

Ask Copilot Chat:

```
Create an API documentation section for #file:README.md that documents all the
Task Manager API endpoints (POST, GET, GET by ID, PUT, DELETE). Include:
- Endpoint URL
- HTTP method
- Request body examples
- Response examples
- Status codes
- Error responses
Format as Markdown
```

Expected Output - Added to README.md:

```
## API Documentation

### Base URL

http://localhost:5000


### Endpoints

#### 1. Create Task
Creates a new task with priority and optional due date.

**Endpoint**: `POST /tasks`
```

~~~markdown
**Request Body**:
```json
{
  "title": "Complete project documentation",
  "description": "Write comprehensive API documentation",
  "priority": "High",
  "dueDate": "2025-10-30T17:00:00Z"
---
## API Documentation

### Base URL

`http://localhost:5000`

### Endpoints

#### 1. Create Task
Creates a new task with priority and optional due date.

**Endpoint**: `POST /tasks`

**Request Body:**
```json
{
    "title": "Complete project documentation",
    "description": "Write comprehensive API documentation",
    "priority": "High",
    "dueDate": "2025-10-30T17:00:00Z"
}
```

**Success Response** (201 Created):
```json
{
    "id": "3fa85f64-5717-4562-b3fc-2c963f66afa6",
    "title": "Complete project documentation",
    "description": "Write comprehensive API documentation",
    "priority": "High",
    "status": "Todo",
    "dueDate": "2025-10-30T17:00:00Z",
    "createdAt": "2025-10-20T10:30:00Z"
}
```

**Note**: The response uses `status` field (enum: Todo, InProgress, Done,
          Cancelled) rather than boolean `isCompleted`. See Lab 3
          documentation for details.

**Error Responses:**
~~~

~~~~markdown
```json
{
    "type": "https://tools.ietf.org/html/rfc7231#section-6.5.1",
    "title": "Bad Request",
    "status": 400,
    "detail": "Invalid priority name: SuperUrgent"
}
```

Other possible errors:
- `400 Bad Request` - Invalid priority or past due date
- `500 Internal Server Error` - Server error
---
    "priority": "High",
    "status": "InProgress",
    "createdAt": "2025-10-20T10:30:00Z"
  }
]
```


---

#### 3. Get Task by ID

Retrieves a specific task by its unique identifier.

**Endpoint**: `GET /tasks/{id}`

**Path Parameters**:

- `id` (required): Task unique identifier (GUID)

**Success Response** (200 OK):

```json
{
  "id": "3fa85f64-5717-4562-b3fc-2c963f66afa6",
  "title": "Complete project documentation",
  "description": "Write comprehensive API documentation",
  "priority": "High",
  "status": "Todo",
  "dueDate": "2025-10-30T17:00:00Z",
  "createdAt": "2025-10-20T10:30:00Z"
}
```


**Error Responses**:

- `404 Not Found` - Task with specified ID does not exist
~~~~

#### 4. Update Task

Updates an existing task's properties.

Endpoint: `PUT /tasks/{id}`

Request Body:

```json
{
  "title": "Updated task title",
  "description": "Updated description",
  "priority": "Critical",
  "status": "Done",
  "dueDate": "2025-11-01T17:00:00Z"
}
```

Success Response (200 OK):

```json
{
  "id": "3fa85f64-5717-4562-b3fc-2c963f66afa6",
  "title": "Updated task title",
  "description": "Updated description",
  "priority": "Critical",
  "status": "Done",
  "dueDate": "2025-11-01T17:00:00Z",
  "createdAt": "2025-10-20T10:30:00Z"
}
```

Error Responses:

- `400 Bad Request` - Invalid data
- `404 Not Found` - Task not found

#### 5. Delete Task

Permanently deletes a task.

Endpoint: `DELETE /tasks/{id}`

Success Response (204 No Content)

Error Responses:

- `404 Not Found` - Task not found

### Priority Values

Valid priority values for task creation and updates:

- `Low` (0)
- `Medium` (1)
- `High` (2)
- `Critical` (3)

### Error Response Format

All error responses follow [RFC 7807 Problem Details](https://datatracker.ietf.org/doc/html/rfc7807) format:

```json
{
    "type": "https://tools.ietf.org/html/rfc7231#section-6.5.1",
    "title": "Bad Request",
    "status": 400,
    "detail": "Invalid priority name: SuperUrgent"
}
```

### 2.3 Generate Architecture Documentation (ADR)

Ask Copilot:

```text
Create an Architecture Decision Record (ADR) in docs/adr/001-clean-architecture-choice.md documenting why we chose Clean Architecture for this Task Manager application. Include:
- Context
- Decision
- Consequences (positive and negative)
- Alternatives considered
Follow the ADR template format
```

## Part 3: Write Conventional Commit Messages (3 minutes)

### Scenario: Commit Your Changes

You've made significant changes across multiple files. Write meaningful commit messages following Conventional Commits format.

### 3.1 Stage Changes

```bash
git add src/TaskManager.Api/Extensions/EndpointExtensions.cs
git add src/TaskManager.Application/Queries/
git add tests/TaskManager.IntegrationTests/
```

### 3.2 Generate Commit Message

In Copilot Chat:

```text
Write a Conventional Commit message for the staged changes. Include a concise subject line (<=72 chars) and a detailed body explaining what was added.
```

Expected Output:

```text
feat(api): add GET endpoints for task retrieval

- Add GET /tasks endpoint with optional completed filter
- Add GET /tasks/{id} endpoint for single task retrieval
- Implement GetTasksQuery and GetTaskByIdQuery handlers
- Add integration tests for both endpoints
- Update endpoint documentation in README

These endpoints complete the CRUD API for task management,
allowing clients to retrieve tasks individually or as a list
with optional filtering by completion status.
```

### 3.3 Commit with Generated Message

```bash
git commit -m "feat(api): add GET endpoints for task retrieval

- Add GET /tasks endpoint with optional completed filter
- Add GET /tasks/{id} endpoint for single task retrieval
- Implement GetTasksQuery and GetTaskByIdQuery handlers
- Add integration tests for both endpoints
- Update endpoint documentation in README
```

These endpoints complete the CRUD API for task management,
allowing clients to retrieve tasks individually or as a list
with optional filtering by completion status."

### 3.4 Practice: Stage More Changes

Stage the refactored legacy code:

```bash
git add src/TaskManager.Infrastructure/Legacy/LegacyTaskProcessor.cs
git add tests/TaskManager.UnitTests/Legacy/
```

Ask Copilot:

```text
Write a Conventional Commit message for refactoring LegacyTaskProcessor.
          Include details about what was improved (async/await, guard clauses,
          logging, extracted methods).
```

Expected Output:

```text
refactor(legacy): modernize LegacyTaskProcessor with clean code principles

- Convert synchronous methods to async/await pattern
- Replace nested if statements with guard clauses
- Add structured logging with ILogger throughout
- Extract ProcessSingleTaskAsync and IsTaskValid methods
- Improve error handling with proper exception catching
- Add comprehensive unit tests

Reduces complexity from cyclomatic complexity of 15 to 4 per method.
Follows Object Calisthenics: one level of indentation per method.
```

Commit:

```bash
git commit -F- <<'EOF'
refactor(legacy): modernize LegacyTaskProcessor with clean code principles

- Convert synchronous methods to async/await pattern
- Replace nested if statements with guard clauses
- Add structured logging with ILogger throughout
- Extract ProcessSingleTaskAsync and IsTaskValid methods
```

- Improve error handling with proper exception catching
- Add comprehensive unit tests

Reduces complexity from cyclomatic complexity of 15 to 4 per method.
Follows Object Calisthenics: one level of indentation per method.
EOF

## Part 4: Draft Pull Request Descriptions (4 minutes)

### Scenario: Prepare for Code Review

You've completed the workshop implementation. Create a comprehensive PR description for review.

### 4.1 Generate PR Description with @workspace

In Copilot Chat:

```text
@workspace Draft a Pull Request description for all the changes made in this branch. Include:
- Summary of changes
- What was implemented (features, refactoring, tests)
- Intent and motivation
- Testing performed
- Breaking changes (if any)
- Checklist for reviewers
Use Markdown formatting
```

Expected Output:

````markdown
# Task Manager API - Complete Implementation

## Summary
This PR implements a complete CRUD API for task management following Clean Architecture and DDD patterns, with comprehensive test coverage and documentation.

## Changes

### Features Implemented
- ✅ **Create Task** (`POST /tasks`) - Create tasks with priority and due date
- ✅ **List Tasks** (`GET /tasks`) - Retrieve all tasks with optional completion filter
````

```markdown
- ✅ **Get Task by ID** (`GET /tasks/{id}`) - Retrieve single task
- ✅ **Update Task** (`PUT /tasks/{id}`) - Update task properties
- ✅ **Delete Task** (`DELETE /tasks/{id}`) - Remove task

### Architecture & Design
- Implemented Clean Architecture layers (Domain, Application, Infrastructure, API)
- Applied DDD patterns: Aggregates, Value Objects, Factory Methods
- CQRS pattern for commands and queries
- Dependency injection throughout
- Proper error handling with ProblemDetails (RFC 7807)

### Code Quality Improvements
- Refactored `LegacyTaskProcessor` to modern standards:
  - Async/await pattern
  - Guard clauses (no nested ifs)
  - Structured logging
  - Extracted methods for single responsibility
  - Reduced cyclomatic complexity from 15 to 4
- Applied Object Calisthenics principles:
  - One level of indentation per method
  - No else keyword
  - Wrapped primitives in value objects
  - Descriptive naming (no abbreviations)

### Testing
- **Unit Tests**: 45+ tests covering all handlers, commands, queries, and domain logic
- **Integration Tests**: 12+ tests for all API endpoints
- Test framework: xUnit v3 with FakeItEasy for mocking
- Test organization: Feature-based with class-per-method folders
- Coverage: ~92% code coverage (excluding infrastructure)

### Documentation
- XML documentation for all public APIs
- Comprehensive README with:
  - API endpoint documentation
  - Request/response examples
  - Error response formats
- Architecture Decision Records (ADRs) for key decisions
- Inline code comments for complex logic

## Intent & Motivation
This implementation serves as a reference for:
- AI-assisted development workflow with GitHub Copilot
- Clean Architecture in .NET 9
- TDD practices (Red-Green-Refactor)
- DDD patterns in practice
- Modern C# conventions and best practices
```

## Testing Performed

### Automated Tests
```bash
dotnet build    # ✅ Success, 0 warnings
dotnet test     # ✅ 57 passed, 0 failed
```

### Manual API Testing
All endpoints tested with curl:
- ✅ Valid requests return correct responses
- ✅ Invalid data returns 400 Bad Request with details
- ✅ Not found scenarios return 404
- ✅ Server errors return 500 with ProblemDetails

### Performance
- Average response time: <50ms for CRUD operations
- No memory leaks detected in load testing

## Breaking Changes
⚠️ **None** - This is initial implementation

## Migration Required
None - Uses in-memory repository for workshop purposes

## Reviewer Checklist
Please verify:
- [ ] All tests pass (`dotnet test`)
- [ ] Build succeeds with no warnings (`dotnet build`)
- [ ] Code follows `.github/copilot-instructions.md` conventions
- [ ] Clean Architecture dependencies respected (no circular references)
- [ ] Domain logic stays in Domain layer (no business logic in API/Infrastructure)
- [ ] All public APIs have XML documentation
- [ ] Error handling uses proper status codes and ProblemDetails
- [ ] Async/await pattern used consistently
- [ ] Structured logging with ILogger (no string interpolation)
- [ ] Guard clauses for parameter validation
- [ ] Tests organized by feature and method

## Related Issues
Closes #1 - Implement Task Manager CRUD API
Closes #2 - Refactor legacy code
Closes #3 - Add comprehensive test coverage

## Screenshots
N/A - API only, no UI

```
## Deployment Notes
- Requires .NET 9.0 SDK or higher
- Uses in-memory data store (no database configuration needed)
- Default port: 5000 (HTTP)

## Next Steps (Future Work)
- [ ] Add database persistence (EF Core with SQL Server)
- [ ] Implement authentication/authorization
- [ ] Add pagination and advanced filtering
- [ ] Implement task notification service
- [ ] Add OpenTelemetry distributed tracing
- [ ] Create Swagger/OpenAPI documentation
```

4.2 Review and Refine

Review the generated PR description and adjust:

- Add specific issue numbers
- Include actual test counts
- Add screenshots if you created a UI
- Highlight any specific areas needing review

Key Learning Points

✅ Testing Best Practices

1. **/tests Command**: Generates comprehensive test suites instantly
2. **Test Coverage**: Happy path, edge cases, error conditions, cancellation
3. **Test Organization**: Feature-based folders, class-per-method files
4. **Integration Tests**: WebApplicationFactory for full API testing

✅ Documentation Efficiency

1. **/doc Command**: XML documentation generated from code context
2. **API Docs**: Clear examples with request/response formats
3. **Architecture Docs**: ADRs document important decisions
4. **Consistency**: AI ensures consistent documentation style

✅ Version Control Quality

1. **Conventional Commits**: Structured, parsable commit messages
2. **Semantic Commits**: Type, scope, description format
3. **Detailed Bodies**: Explain what, why, and how
4. **Changelog Ready**: Commits can generate CHANGELOG.md automatically

✅ Code Review Preparation

1. **@workspace Context**: Full codebase understanding for PR description
2. **Comprehensive PRs**: All changes documented and explained
3. **Reviewer Checklist**: Clear acceptance criteria

4. **Impact Analysis**: Breaking changes, migrations, deployment notes

---

Extension Exercises (If Time Permits)

Exercise 1: Generate CHANGELOG.md

Ask Copilot to generate a CHANGELOG.md file from your commit history:

```
Generate a CHANGELOG.md file based on the git commit history. Group by
version, follow Keep a Changelog format.
```

Exercise 2: Create Contributing Guidelines

Generate CONTRIBUTING.md with guidelines for contributors:

```
Create a CONTRIBUTING.md file that explains:
- How to set up the development environment
- Coding conventions (reference .github/copilot-instructions.md)
- Testing requirements
- PR process
- Commit message format
```

Exercise 3: API Client SDK Documentation

Generate documentation for consuming the API:

```
Create a quick start guide in docs/guides/api-quickstart.md for developers
consuming our Task Manager API. Include authentication (if applicable),
endpoint examples, and common error handling patterns.
```

Success Criteria

You've completed this lab successfully when:

- ✅ Comprehensive test suite generated with `/tests` (unit + integration)
- ✅ All public APIs have XML documentation via `/doc`
- ✅ API documentation in README.md with examples
- ✅ Conventional Commit messages written for all changes
- ✅ Complete PR description drafted with `@workspace`
- ✅ All tests passing
- ✅ Documentation is clear and maintainable
- ✅ Ready for code review

Workshop Wrap-Up

Congratulations! You've completed all four labs. You now know how to:

✅ Test-Driven Development (Lab 1)

- Follow Red-Green-Refactor cycle
- Use Copilot to generate tests before implementation
- Apply Copilot Instructions for consistent code quality

✅ Requirements to Code (Lab 2)

- Decompose user stories into backlog items
- Generate acceptance criteria
- Implement features using TDD
- Maintain Clean Architecture across all layers

✅ Code Generation & Refactoring (Lab 3)

- Generate complete API endpoints with context
- Refactor legacy code with `/refactor`
- Apply Object Calisthenics principles
- Use Copilot Edits for multi-file changes

✅ Testing, Documentation & Workflow (Lab 4)

- Generate comprehensive test coverage
- Create clear documentation
- Write meaningful commit messages
- Prepare thorough PR descriptions

Troubleshooting

/tests Generates Incomplete Tests

Problem: Tests don't cover all edge cases
Solution: Be explicit: "/tests including edge cases, error handling, and cancellation"

/doc Generates Generic Comments

Problem: XML comments don't add value beyond method signature
Solution: Select more context (class + method), provide business context in prompt

Commit Message Too Generic

Problem: Copilot generates "Update files" type messages
Solution: Stage related changes only, provide context: "Write commit for adding GET endpoints"

PR Description Missing Details

Problem: PR description is too high-level
Solution: Use @workspace and be specific: "Include testing details, breaking changes, and reviewer checklist"

Next Steps Beyond Workshop

Apply to Real Projects

1. Add `.github/copilot-instructions.md` to your team's repositories
2. Establish Conventional Commits standard
3. Use `/tests` for all new code
4. Use `/doc` for public APIs
5. Use `@workspace` in daily work

Advanced Copilot Usage

1. Custom instructions for team-specific patterns
2. Copilot for Business with organization policies
3. Fine-tuned models for domain-specific code
4. Integration with CI/CD pipelines

Continue Learning

- Practice TDD with different features
- Explore advanced DDD patterns
- Learn OpenTelemetry for observability
- Study Clean Architecture in depth

Additional Resources

- xUnit Documentation
- Conventional Commits Specification
- Keep a Changelog
- RFC 7807 Problem Details
- GitHub Copilot Best Practices
- Clean Architecture by Uncle Bob