GitHub Copilot Prompt Files 📄

Prompt Files are reusable, shareable templates that standardize how you interact with GitHub Copilot. They enable consistent prompting patterns across teams and projects while capturing best practices for specific development scenarios.

🎯 What Are Prompt Files?

Prompt Files allow you to:

- **Standardize Prompts**: Create consistent interaction patterns
- **Share Knowledge**: Distribute proven prompting strategies
- **Automate Context**: Pre-configure context variables and parameters
- **Enhance Productivity**: Reduce repetitive prompt writing

File Structure

```
# .github/prompts/code-review.prompt.md

---
title: "Code Review Assistant"
description: "Comprehensive code review with security and performance
focus"
author: "Development Team"
tags: ["review", "security", "performance"]
context:
  - "#selection"
  - "#file"
variables:
  - name: "focus_areas"
    type: "array"
    default: ["security", "performance", "maintainability"]
---

## Prompt Template

Review the provided code focusing on {{focus_areas}}.

### Analysis Areas:
1. **Security**: Check for vulnerabilities and secure coding practices
2. **Performance**: Identify optimization opportunities
```

```
3. **Maintainability**: Assess code clarity and documentation

### Output Format:
- Issues found (with severity levels)
- Specific recommendations
- Code examples for improvements

Please provide actionable feedback with line-specific suggestions.
```

🚀 Getting Started with Prompt Files

Basic Prompt File Creation

1. **Create Directory**: `.github/prompts/` in your repository
2. **File Naming**: Use `.prompt.md` extension
3. **Add Metadata**: Include frontmatter with configuration
4. **Write Template**: Create reusable prompt content

Simple Example

```
# .github/prompts/bug-fix.prompt.md

---
title: "Bug Fix Assistant"
description: "Systematic approach to debugging and fixing issues"
context:
  - "#file"
  - "#terminal"
---

## Bug Analysis

Analyze this code for potential bugs and provide fixes:

1. Identify the root cause
2. Suggest specific fixes
3. Explain the reasoning
4. Provide test cases to prevent regression

Focus on common issues like null pointer exceptions, memory leaks, and
logic errors.
```

🛠️ Advanced Prompt File Features

Variable Substitution

```
---
title: "API Endpoint Generator"
variables:
  - name: "http_method"
    type: "select"
    options: ["GET", "POST", "PUT", "DELETE"]
  - name: "resource_name"
    type: "string"
    required: true
  - name: "auth_required"
    type: "boolean"
    default: true
---

Create a {{http_method}} endpoint for {{resource_name}} resource.

Requirements:
- Authentication: {{#if auth_required}}Required{{else}}Not required{{/if}}
- Input validation
- Error handling
- Response formatting
```

Context Integration

```
---
title: "Test Generator"
context:
  - "#file:src/main/java/**/*.java"
  - "#selection"
  - "pom.xml"
variables:
  - name: "test_framework"
    type: "select"
    options: ["JUnit", "TestNG", "Mockito"]
    default: "JUnit"
---

Generate comprehensive unit tests using {{test_framework}} for the
selected code.

Requirements:
```

```
- Test all public methods
- Include edge cases
- Use proper assertions
- Mock external dependencies
- Follow AAA pattern (Arrange, Act, Assert)

Consider the project structure from the context files.
```

Conditional Logic

```
---
title: "Documentation Generator"
variables:
  - name: "doc_type"
    type: "select"
    options: ["API", "User Guide", "Technical Spec"]
  - name: "include_examples"
    type: "boolean"
    default: true
---

Generate {{doc_type}} documentation.

{{#if doc_type === "API"}}
Include:
- Endpoint descriptions
- Request/response schemas
- Authentication details
- Error codes
{{/if}}

{{#if doc_type === "User Guide"}}
Include:
- Step-by-step instructions
- Screenshots placeholders
- Troubleshooting section
{{/if}}

{{#if include_examples}}
Provide practical examples for each section.
{{/if}}
```

📁 Prompt File Organization

Repository Structure

```
.github/
├── prompts/
|   ├── development/
|   |   ├── code-review.prompt.md
|   |   ├── refactor.prompt.md
|   |   └── optimization.prompt.md
|   ├── testing/
|   |   ├── unit-tests.prompt.md
|   |   ├── integration-tests.prompt.md
|   |   └── e2e-tests.prompt.md
|   ├── documentation/
|   |   ├── api-docs.prompt.md
|   |   ├── readme.prompt.md
|   |   └── comments.prompt.md
|   └── templates/
|       ├── spring-boot-controller.prompt.md
|       └── react-component.prompt.md
```

Category Guidelines

Category	Purpose	Examples
`development/`	Core coding tasks	Reviews, refactoring, optimization
`testing/`	Test creation and debugging	Unit tests, integration tests
`documentation/`	Content generation	README files, API docs, comments
`templates/`	Code scaffolding	Controllers, components, utilities
`maintenance/`	Code upkeep	Dependency updates, security patches

💡 Best Practices

✅ Do's

1. **Use Clear Metadata**: Descriptive titles and tags `yaml title: "Spring Boot Entity Generator" description: "Generate JPA entities with validation annotations" tags: ["spring-boot", "jpa", "entity", "validation"]`

2. **Provide Context**: Include relevant files and selections ```yaml context:

   - "#file:src/main/resources/application.properties"
   - "#file:pom.xml"
   - "#selection" ```

3. **Use Variables Wisely**: Make prompts flexible but not complex ```yaml variables:

   - name: "entity_name" type: "string" required: true
   - name: "table_name" type: "string" default: "{{entity_name | lowercase}}" ```

4. **Structure Output**: Specify expected response format ```markdown ### Expected Output:

5. Entity class with annotations
6. Repository interface
7. Basic service methods
8. Controller endpoints ```

❌ Don'ts

1. **Don't Over-Complicate**: Keep prompts focused and simple ```markdown # Bad - Too many variables and conditions variables:

   - name: "framework" # 20+ options
   - name: "database" # 15+ options
   - name: "auth_type" # 10+ options

# Good - Focused scope variables: - name: "entity_name" type: "string" required: true ```

1. **Don't Ignore Context**: Always consider what Copilot needs to know ```markdown # Bad - No context Generate a REST controller.

# Good - With context context: - "#file:src/main/java/*/.java" Generate a REST controller following the existing patterns. ```

1. **Don't Forget Validation**: Test prompts thoroughly ```markdown # Include validation in your prompts Validate the generated code for:
2. Syntax errors
3. Coding standards compliance
4. Security best practices ```

🎨 Practical Examples

Spring Boot Development

```
# .github/prompts/spring-boot-service.prompt.md

---
title: "Spring Boot Service Layer"
description: "Generate service class with business logic and error
handling"
context:
  - "#file:src/main/java/**/entity/*.java"
  - "#file:src/main/java/**/repository/*.java"
variables:
  - name: "entity_name"
    type: "string"
    required: true
  - name: "include_validation"
    type: "boolean"
    default: true
---

Generate a Spring Boot service class for {{entity_name}} entity.

Requirements:
- Use @Service annotation
- Implement CRUD operations
- Include proper error handling
- Add transaction management
{{#if include_validation}}
- Include input validation
- Add custom exceptions
{{/if}}
```

```
Follow the repository patterns from the context files.
```

Testing Framework

```
# .github/prompts/selenium-test.prompt.md

---
title: "Selenium Test Generator"
description: "Create comprehensive UI tests with page object model"
context:
  - "#file:src/test/java/**/*.java"
  - "#file:testng.xml"
variables:
  - name: "page_name"
    type: "string"
    required: true
  - name: "test_scenarios"
    type: "array"
    default: ["happy_path", "validation", "error_handling"]
---

Generate Selenium tests for {{page_name}} page.

Create:
1. Page Object class with locators and methods
2. Test class with {{test_scenarios}} scenarios
3. TestNG configuration
4. Data providers if needed

Follow existing test patterns and use WebDriverManager.
```

🔧 Advanced Techniques

Multi-Step Workflows

```
---
title: "Complete Feature Implementation"
description: "End-to-end feature development workflow"
variables:
  - name: "feature_name"
    type: "string"
    required: true
```

```
steps:
  - prompt: "entity-generator"
    variables:
      entity_name: "{{feature_name}}"
  - prompt: "service-generator"
    variables:
      entity_name: "{{feature_name}}"
  - prompt: "controller-generator"
    variables:
      entity_name: "{{feature_name}}"
  - prompt: "test-generator"
    variables:
      component_name: "{{feature_name}}"
---

This workflow will create a complete feature implementation including:
1. Entity class
2. Service layer
3. REST controller
4. Unit tests

Each step builds on the previous one.
```

Team Collaboration

```
---
title: "Code Review Checklist"
description: "Team-specific code review criteria"
author: "Senior Development Team"
version: "2.1"
context:
  - "#selection"
  - ".github/PULL_REQUEST_TEMPLATE.md"
variables:
  - name: "review_type"
    type: "select"
    options: ["feature", "bugfix", "refactor", "security"]
---

Perform {{review_type}} code review using team standards:

## Security Review
- [ ] Input validation present
- [ ] SQL injection prevention
- [ ] Authentication/authorization checks

## Performance Review
```

```
- [ ] Database query optimization
- [ ] Memory usage consideration
- [ ] Caching strategy

## Maintainability Review
- [ ] Code comments and documentation
- [ ] Unit test coverage
- [ ] Following team conventions

Provide specific line-by-line feedback.
```

🚨 Common Pitfalls

Variable Overuse

Problem: Too many variables make prompts complex **Solution**: Keep variables focused and essential

Context Bloat

Problem: Including too much irrelevant context **Solution**: Be selective about context files

Poor Organization

Problem: Prompt files scattered without structure **Solution**: Use clear directory organization

🎓 Learning Exercises

Beginner

1. Create a simple prompt file for code commenting
2. Use variables to make a generic test generator
3. Practice context selection for different scenarios

Intermediate

1. Build workflow prompts for complete features
2. Create team-specific review prompts

3. Implement conditional logic in prompts

Advanced

1. Design multi-step development workflows
2. Create organization-wide prompt libraries
3. Integrate prompt files with CI/CD pipelines

Remember: Prompt Files are powerful tools for standardizing and sharing AI interactions. Start simple, iterate based on usage, and always test your prompts thoroughly before sharing with the team.