GitHub Copilot Custom Instructions

GitHub Copilot Custom Instructions allow you to personalize AI responses to match your coding style, team standards, and project requirements. These instructions help ensure consistency and relevance across all your Copilot interactions.

Overview

Custom instructions provide context and preferences that Copilot uses to tailor its responses. They can be set at multiple levels:

- **Personal**: Individual developer preferences
- **Workspace**: Project-specific instructions
- **Repository**: Shared team standards
- **Organization**: Enterprise-wide policies (GitHub Enterprise)

Instruction Types

Code Generation Instructions

Control how Copilot generates new code across your projects.

Setting: `github.copilot.chat.codeGeneration.instructions`

Example:

```
{
  "github.copilot.chat.codeGeneration.instructions": "
    - Use TypeScript with strict type checking
    - Follow functional programming patterns when possible
    - Include comprehensive JSDoc comments for all public functions
    - Use descriptive variable names (no abbreviations)
    - Implement proper error handling with custom error classes
    - Follow the company ESLint configuration
    - Use async/await instead of Promise chains
    - Include unit tests for all new functions
  "
}
```

Test Generation Instructions

Specify testing preferences and patterns.

Setting: `github.copilot.chat.testGeneration.instructions`

Example:

```
{
  "github.copilot.chat.testGeneration.instructions": "
    - Use Jest as the testing framework
    - Follow AAA pattern (Arrange, Act, Assert)
    - Create comprehensive test suites with positive, negative, and edge
cases
    - Use descriptive test names that explain the scenario
    - Mock external dependencies using jest.mock()
    - Include integration tests for API endpoints
    - Aim for 90% code coverage minimum
    - Use test-driven development approach
  "
}
```

Code Review Instructions

Guide Copilot's code review and analysis capabilities.

Setting: `github.copilot.chat.reviewSelection.instructions`

Example:

```
{
  "github.copilot.chat.reviewSelection.instructions": "
    - Focus on security vulnerabilities (OWASP Top 10)
    - Check for performance optimization opportunities
    - Verify proper error handling and logging
    - Ensure code follows SOLID principles
    - Review for accessibility compliance (WCAG 2.1 AA)
    - Check for proper documentation and comments
    - Validate proper use of design patterns
    - Ensure consistent code style and formatting
  "
}
```

Commit Message Instructions

Standardize commit message generation.

Setting: `github.copilot.chat.commitMessageGeneration.instructions`

Example:

```
{
  "github.copilot.chat.commitMessageGeneration.instructions": "
    - Follow Conventional Commits format: type(scope): description
    - Use present tense and imperative mood
    - Keep first line under 50 characters
    - Include detailed description for complex changes
    - Reference issue numbers when applicable
    - Use these types: feat, fix, docs, style, refactor, test, chore
    - Include breaking change indicators when needed
  "
}
```

Repository-Level Instructions

.github/copilot-instructions.md

Place instructions in your repository root for team-wide consistency.

File Location: `.github/copilot-instructions.md`

Example Content:

```
# Copilot Instructions for [Project Name]

## Code Style
- Use 2-space indentation for JavaScript/TypeScript
- Use 4-space indentation for Python
- Maximum line length: 100 characters
- Use trailing commas in multiline structures

## Architecture Patterns
- Follow Clean Architecture principles
- Use dependency injection for services
- Implement repository pattern for data access
- Use command/query separation (CQRS) for complex operations

## Testing Requirements
```

```
- Unit tests are mandatory for all business logic
- Integration tests for API endpoints
- E2E tests for critical user journeys
- Mock external services in tests

## Security Guidelines
- Never commit secrets or API keys
- Use environment variables for configuration
- Implement proper input validation
- Follow least privilege principle

## Documentation
- Update README for any new features
- Include API documentation for endpoints
- Add inline comments for complex business logic
- Update CHANGELOG.md for releases
```

Advanced Configuration

Language-Specific Instructions

Create targeted instructions for different programming contexts:

```
{
  "github.copilot.chat.codeGeneration.instructions": {
    "typescript": "
      - Use strict TypeScript configuration
      - Prefer interfaces over types for object shapes
      - Use enums for constants with semantic meaning
      - Implement proper generic constraints
    ",
    "python": "
      - Follow PEP 8 style guidelines
      - Use type hints for all function parameters and returns
      - Prefer dataclasses over regular classes for data containers
      - Use context managers for resource handling
    ",
    "java": "
      - Use Spring Boot conventions and annotations
      - Implement proper exception handling with custom exceptions
      - Use builder pattern for complex object creation
      - Follow Java naming conventions
    "
  }
}
```

Framework-Specific Instructions

Tailor instructions to your technology stack:

```
## React Development
- Use functional components with hooks
- Implement proper error boundaries
- Use React.memo for performance optimization
- Follow component composition patterns
- Use custom hooks for reusable logic

## Spring Boot
- Use constructor injection over field injection
- Implement proper validation with Bean Validation
- Use profiles for environment-specific configuration
- Follow RESTful API design principles

## Express.js
- Use middleware for cross-cutting concerns
- Implement proper error handling middleware
- Use async/await for asynchronous operations
- Follow RESTful routing conventions
```

Enterprise Instructions

For organizations using GitHub Enterprise:

```
{
  "github.copilot.enterprise.instructions": {
    "security": {
      "dataClassification": "Treat all code as confidential",
      "compliance": "Ensure SOC 2 Type II compliance",
      "auditLogging": "Log all AI-generated code for review"
    },
    "codeStandards": {
      "licensing": "Use MIT license for internal libraries",
      "dependencies": "Only use pre-approved npm packages",
      "architecture": "Follow microservices architecture patterns"
    }
  }
}
```

Instruction Hierarchy

When multiple instruction sources exist, Copilot follows this priority order:

1. **Explicit prompt instructions** (highest priority)
2. **Repository** `.github/copilot-instructions.md`
3. **Workspace-level VS Code settings**
4. **User-level VS Code settings**
5. **Default Copilot behavior** (lowest priority)

Best Practices

Writing Effective Instructions

1. **Be Specific and Clear**

```
❌ "Write good code"
✅ "Use TypeScript strict mode, include JSDoc comments, and implement
proper error handling"
```

1. **Provide Context**

```
❌ "Follow our standards"
✅ "Follow the Airbnb JavaScript style guide with these modifications:
[list specific changes]"
```

1. **Include Examples**

```
✅ "Function naming convention: use camelCase for functions (getUserData),
PascalCase for classes (UserService)"
```

1. **Be Consistent**

```
✅ "Always use async/await instead of .then() chains for Promise handling"
```

Instruction Management

1. **Version Control Instructions**

2. Store repository instructions in version control
3. Review and update instructions regularly
4. Document changes to instruction sets
5. **Team Collaboration**
6. Discuss instruction changes in team meetings
7. Create pull requests for instruction updates
8. Maintain consistency across projects
9. **Testing Instructions**
10. Verify instructions work as expected
11. Test with various code scenarios
12. Gather feedback from team members

Troubleshooting

Common Issues

Problem	Solution
Instructions not being followed	Check instruction hierarchy and ensure proper file placement
Conflicting instructions	Review multiple instruction sources and resolve conflicts
Instructions too generic	Make instructions more specific and actionable
Performance issues	Reduce instruction length and complexity

Debugging Instructions

1. **Test with Simple Prompts**
2. Use basic code generation requests
3. Verify instructions are applied correctly
4. **Check File Locations**
5. Ensure `.github/copilot-instructions.md` exists

6. Verify VS Code settings are correct
7. **Review Instruction Syntax**
8. Check for JSON formatting errors
9. Validate markdown syntax in instruction files

Enterprise Features

Centralized Instruction Management

- Deploy instructions across organization repositories
- Enforce mandatory instruction compliance
- Audit instruction usage and effectiveness

Template Instructions

- Create reusable instruction templates
- Share best practices across teams
- Maintain consistency across projects

Analytics and Insights

- Track instruction effectiveness
- Monitor code quality improvements
- Measure developer productivity gains

Migration Guide

From Legacy Configurations

If migrating from older Copilot configurations:

1. **Review Existing Settings**
2. Identify current customizations
3. Document team preferences
4. **Create Instruction Files**

5. Convert settings to new instruction format
6. Test instructions with team
7. **Gradual Rollout**
8. Implement instructions incrementally
9. Gather team feedback and adjust

Examples by Use Case

Startup/Small Team

```
# Focus on rapid development and flexibility
- Prioritize working code over perfect architecture
- Use modern JavaScript/TypeScript features
- Include basic error handling and logging
- Write tests for critical business logic
```

Enterprise/Large Team

```
# Focus on maintainability and standards
- Follow established architecture patterns
- Include comprehensive documentation
- Implement full test coverage
- Ensure security and compliance requirements
- Follow code review guidelines
```

Open Source Project

```
# Focus on community and contribution
- Follow project contribution guidelines
- Include clear documentation and examples
- Ensure backward compatibility
- Write comprehensive tests
- Follow semantic versioning
```

Custom instructions are powerful tools for making GitHub Copilot work exactly how you and your team need it to. Start with basic instructions and gradually refine them based on your experience and team feedback.