GitHub Copilot Deep Dive

Your Trainer

Marie Theresa Brosig

Solutions Engineer
Empathetic Problem-Solver

Harald Binkle

Fullstack | DevOps
Consultant | Trainer

Agenda

1. GitHub Copilot Features per IDE
2. Instruction Files
3. Prompt Files
4. Prompt Patterns
5. Chatmode Files
6. Pass to Marie

GitHub Copilot IDE Features

Copilot Feature	VS Code	Visual Studio	JetBrains	Eclipse	Xcode
Code Completion	✅	✅	✅	✅	✅
Copilot Chat	✅	✅	✅	✅	✅
Copilot Edit	✅	n/a	✅	✅	
Inline Chat	✅	✅	✅	✅	
Copilot Agent Mode	✅	✅	🔬	✅	✅

Copilot Feature	VS Code	Visual Studio	JetBrains	Eclipse	Xcode
Custom Instructions (global)		✓	✓		✓
Custom Instructions (multiple/selective)	✓	✓			
Debugger integration	✓	✓			
Explain and debug test failures	✓	✓			

Copilot Feature	VS Code	Visual Studio	JetBrains	Eclipse	Xcode
Next Edit Suggestions		✓			
Git commit messages	✓	✓	✓		
Code Review	✓	✓			
Prompt Files	✓	✓	✓		
Agent Tool Sets	✓	✓			

Instruction Files

Instruction Files

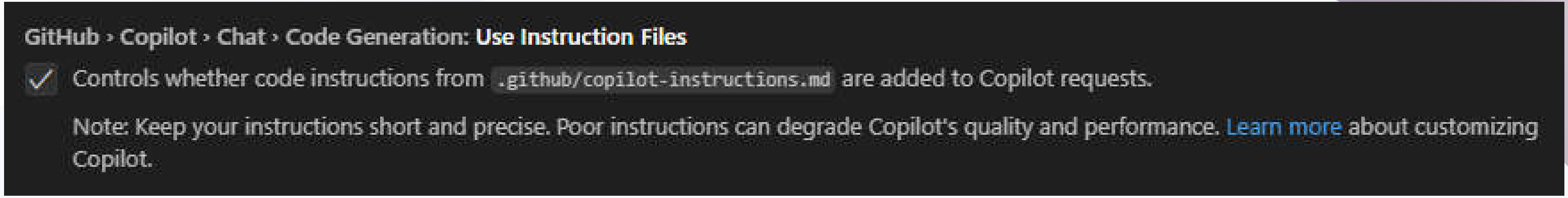
Enhance Copilot’s chat responses by providing contextual details

- Instructions applied to chat prompts automatically
- Contain specific **instructions** or **preferences**
  - Coding standards
  - Preferred libraries
  - Naming conventions
  - Project specific requirements
- Specific purposes
  - Code-generation
  - Test-generation
  - Code review
  - Commit message generation
  - Pull request title and description generation

Use Instruction Files

Enable in `settings.json`

- `"github.copilot.chat.codeGeneration.useInstructionFiles": true`



GitHub › Copilot › Chat › Code Generation: **Use Instruction Files**

✓ Controls whether code instructions from `.github/copilot-instructions.md` are added to Copilot requests.

Note: Keep your instructions short and precise. Poor instructions can degrade Copilot's quality and performance. Learn more about customizing Copilot.

Create Instructions #1

- **Workspace scope** `.vscode\settings.json`
- **User scope** `settings.json`

```
"github.copilot.chat.codeGeneration.instructions": [
  {
    "text": "Always add a comment: 'Generated by Copilot'."
  },
  {
    "text": "In TypeScript always use underscore for private field names."
  },
  {
    "file": "javascript-styles.md" // import instructions from markdown file
  },
  {
    "file": "template.js" // import instructions from code file
  }
]
```

- Code files can be referenced as well
- A code file should be representative or a template

Create Instructions #1

Instructions in `settings.json`

- `github.copilot.chat.codeGeneration.instructions`

  Provide context specific for generating code
- `github.copilot.chat.testGeneration.instructions`

  Provide context specific for generating tests
- `github.copilot.chat.reviewSelection.instructions`

  Provide context specific for reviewing the current editor selection
- `github.copilot.chat.commitMessageGeneration.instructions`

  Provide context specific for generating commit messages
- `github.copilot.chat.pullRequestDescriptionGeneration.instructions`

  Provide context specific for generating pull request titles and descriptions

Create Instructions #2

Instructions in `.github/copilot-instructions.md`

- Contains natural language instructions
- Markdown format can be used

Note

- If custom instructions are defined in both the `settings.json` and `.github/copilot-instructions.md` file, Copilot tries to combine instructions from both sources
- Code-generation instructions does not apply for code completions

Important

- Instruction files are adding up to the prompts token count
- Instructions defined in the `settings.json` are only added for the specific purpose

Example Of `copilot-instructions.md` #1

```markdown
## reply preferences
- If I tell you that you are wrong, think about whether or not you think that's true and respond with facts.
- Avoid apologizing or making conciliatory statements.
- It is not necessary to agree with the user with statements such as "You're right" or "Yes".
- Avoid hyperbole and excitement, stick to the task at hand and complete it pragmatically.

## code generation
- Prefer using modern C# features such as pattern matching and async streams.
- Always use `var` instead of explicit types when the type is obvious.
- Always include error handling for asynchronous operations.
- Use async/await syntax for asynchronous programming.

- Utility methods are located in `HelperClass.cs`.
- Logging functionality is implemented in `Logger.cs`.

Use the following libraries:
- System.Text.Json for JSON serialization/deserialization
- xUnit for generating unit tests
- FluentAssertions for unit test assertions
- Moq for mocking dependencies in unit tests
```

Example Of `copilot-instructions.md` #2

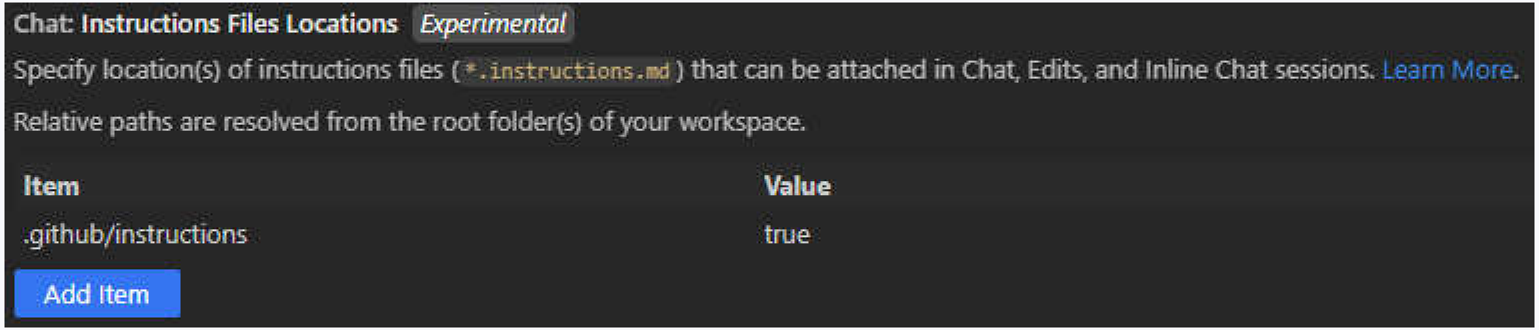
```
Use the following naming conventions:
- Classes: PascalCase
- Methods: PascalCase
- Variables: camelCase
- Constants: UPPER_SNAKE_CASE

Use the following coding standards:
- Use 4 spaces for indentation.
- Use spaces around operators and after commas.
- Limit lines to 120 characters.
```

Create Custom Instruction Files

Create one or more `.instructions.md` files to store custom instructions for specific tasks

- **Workspace instructions files** stored in the `.github/instructions` folder
- **User instruction files** stored in the user profile

```
---
applyTo: "**"
---
Add a comment at the end of the file: 'Contains AI-generated edits.'
```



Chat: **Instructions Files Locations** *Experimental*

Specify location(s) of instructions files (`*.instructions.md`) that can be attached in Chat, Edits, and Inline Chat sessions. Learn More.

Relative paths are resolved from the root folder(s) of your workspace.

Item	Value
.github/instructions	true

Prompt Files

Prompt Files

Build and share reusable prompt instructions with additional context (experimental)

Common use cases

- **Code generation**
  Create reusable prompts for components, tests, or migrations
- **Domain expertise**
  Share specialized knowledge through prompts
- **Team collaboration**
  Document patterns and guidelines with references to specs and documentation
- **Onboarding**
  Create step-by-step guides for complex processes or project-specific patterns

Prompt Files

Benefits

- **Reduce time** spent crafting prompts
- Compose **reusable prompts**
- Enforce **consistency**

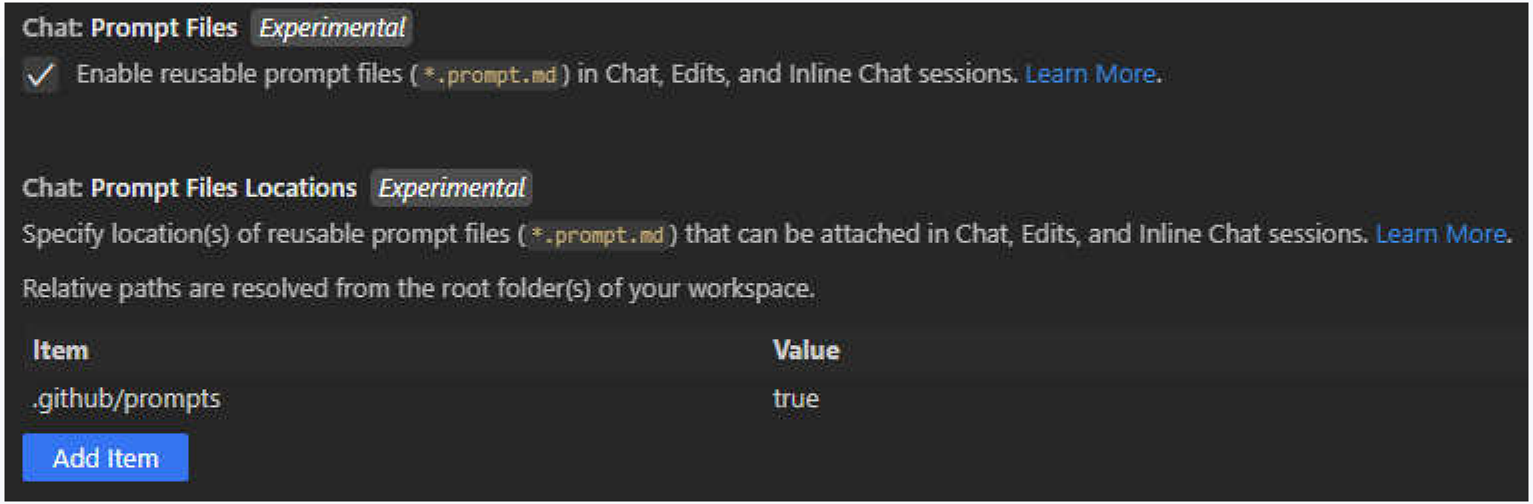
Note

Prompt files are **not** automatically applied to a chat request

Use Prompt Files

Enable in `settings.json`

- `"chat.promptFiles": true`
- `"chat.promptFilesLocations": ".github/prompts"` (default)



Chat: Prompt Files *Experimental*

✓ Enable reusable prompt files (`*.prompt.md`) in Chat, Edits, and Inline Chat sessions. Learn More.

Chat: Prompt Files Locations *Experimental*

Specify location(s) of reusable prompt files (`*.prompt.md`) that can be attached in Chat, Edits, and Inline Chat sessions. Learn More.

Relative paths are resolved from the root folder(s) of your workspace.

Item	Value
.github/prompts	true

Create Prompt Files

Workspace scope

- Create a `.prompt.md` file in the `.github/prompts` directory
  - Alternatively, select the **Create Prompt** command from the Command Palette (`Ctrl+Shift+P`)
  - Enter a name for the prompt file
- Write prompt instructions by using Markdown formatting

User scope

User prompt files are stored in the user profile and can be shared across multiple workspaces

- Select the **Create User Prompt** command from the Command Palette (`Ctrl+Shift+P`)
- Enter a name for the prompt file
- Write prompt instructions by using Markdown formatting

Prompt File Structure

```
---
mode: 'edit'
tools: ['githubRepo', 'codebase']
description: 'Prompt description'
---

Prompt
```

- Header (Front Matter syntax, optional)
  - `mode` (`ask` , `edit` , `agent`)
  - `tools` (e.g. `terminalLastCommand` or `githubRepo`)
  - `description`
- Body with the prompt content
  - Reference variables using the `${variableName}` syntax

Prompt File Example

```
---
mode: 'edit'
tools: ['codebase']
description: 'Perform a REST API security review'
---

Perform a REST API security review:
* Ensure all endpoints are protected by authentication and authorization
* Validate all user inputs and sanitize data
* Implement rate limiting and throttling
* Implement logging and monitoring for security events
```

Prompt File References

Reference reuseable prompt files

- as Markdown link `[security](security-api.prompt.md)`
- as Copilot link `#file:security-api.prompt.md`

Reference variables

- Workspace variables: `${workspaceFolder}` , `${workspaceFolderBasename}`
- Selection variables: `${selection}` , `${selectedText}`
- File context variables: `${file}` , `${fileBasename}` , `${fileDirname}` , `${fileBasenameNoExtension}`
- Input variables: `${input:variableName}` , `${input:variableName:placeholder}`

Use A Prompt File In Chat

- Select the **Attach Context** icon (`Ctrl+#` or `Ctrl+/`), then select **Prompt**
  - Alternatively, select the **Chat: Use Prompt** command from the Command Palette (`Ctrl+Shift+P`)
- or use the `Ctrl+Alt+#` (or `Ctrl+Alt+/`) shortcut to open the prompt file Quick Pick
- Choose a prompt file from the Quick Pick
  - Prompt files can be used in **Copilot Chat** and **Copilot Edits**

```
Create a new API endpoint using the /my-prompt-file: myVar=myVarValue best practices
```

Prompt Patterns

Catalog Of Prompt Patterns

Categories

- Input Semantics
- Output Customization
- Error Identification
- Prompt Improvement

Pattern Category

Input Semantics

Pattern: Meta Language Creation

Description: Creating new, domain-specific language or commands that abstract and simplify complex operations.

Goal: Minimize prompt complexity and increase prompting speed.

Some use cases:

- Create a shorthand for information or an activity that you'll repeat multiple times during a session.
- Meta language shorthand can be saved and shared with the team. This allows for a kind of refactoring of shared prompts.
- Faster iteration of repeated tasks.

Complimenting patterns:

- Template pattern

Pattern Category

Output Customization

Pattern: n-Shot Prompting

Description: Presenting the language model with multiple examples (n examples) of similar tasks or outputs before asking it to generate code for a new, similar task.

Goal: Helps the model understand the context, structure, and specific requirements of the task at hand by learning from the patterns and solutions provided in the example context.

Some use cases:

- Domain-specific applications where pre-existing examples can significantly inform the desired output.
- Improving code consistency and adherence to project-specific conventions.

Template Pattern

Description: Ensure the LLM output adheres to a predefined structure or format. Especially useful when the output must conform patterns not inherently known to the LLM.

Goal: Force and constrain the structure of output structure. Ensure consistency.

Some use cases:

- Creation of REST API endpoint scaffolds with standardized documentation and error handling.
- Producing structured data objects (like JSON or XML) that must follow a specific schema.
- Outputting documentation into a precise format.

Complimenting patterns:

- n-Shot Prompting
- Meta language creation
- Least-to-most prompting

Pattern: Output Automater

Description: Have the LLM generate a script or other automation artifact that can automatically perform any steps it recommends taking as part of its output.

Goal: Reduce the manual effort needed to implement output recommendations.

Example:

From now on, whenever you generate code that spans more than one file, generate a Python script that can be run to automatically create the specified files or make changes to existing files to insert the generated code.

Pattern: Persona

Description: Focus the language model to embody a persona that possesses specialized expertise or bias, software engineer, a data scientist, or any relevant professional. The model leverages this persona to understand and generate code more effectively, providing responses that not only solve coding problems but also reflect the persona's unique approach and knowledge.

Goal: Generate code that not solves the given problem in a way that is not only technically sound but also reflects the insights and nuances of the chosen persona.

Some use cases:

- Considering a solution from multiple perspectives.
- Filling skill gaps where you may be weak. Use this when you don't know what details are important.
- Focusing the output toward a particular challenge or goal.

Complimenting patterns:

- Question refinement

Pattern Category

Error Identification

Pattern: Reflection

Description: Prompting the model to explain the reasoning behind the code it generates. This reveals the logic behind the generated code and sheds light on any assumptions made during the process. It aims to unveil the model's thought process, offering clarity on the chosen solutions, frameworks, and algorithms.

Goal: Reveal potential knowledge gaps or misunderstandings. Increase trust and confidence in the model output by adding transparency.

Some use cases:

- Understanding algorithmic choice.
- Provide insights into the selection of frameworks or patterns.
- Validating model assumptions.

Complimenting patterns:

- Fact Check List

Pattern Category

Prompt Improvement

Pattern: Refusal Breaker

Description: Ask the model to help rephrase or recontextualize a question or command that has been refused.

Goal: Understand the aspects of the prompt that violated model guidelines, intending to improve the prompt so that it adheres to guidelines and alignment while still providing useful information.

Some use cases:

- Improving a bad prompt.

Complimenting patterns:

- Question refinement

Pattern: Cognitive Verifier

Description: Use the model to generate additional, clarifying questions to better understand and accurately respond to the user prompt.

Goal: Improve output where the original prompt may have been too vague. Help models with the generalization of difficult problems.

Some use cases:

- A great general prompt for code generation.
- Uncovering and drilling into requirements.

Chatmode Files

Chatmode Files

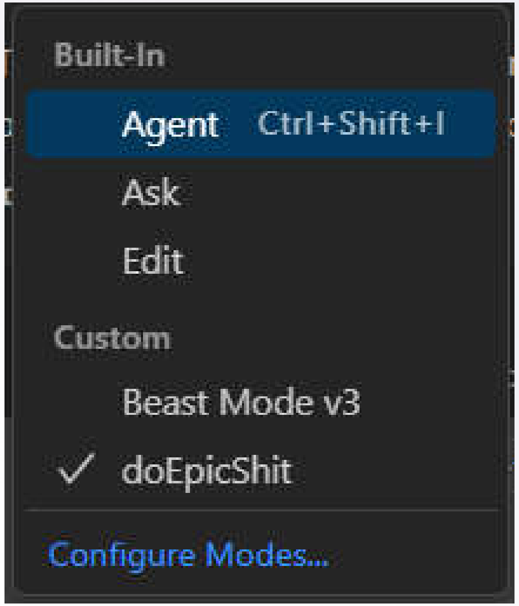
Define custom AI personalities or workflows for Copilot Chat, tailored to specific tasks and workflows

Common use cases

- **Role Based Definitionseview**
  Review, Testing, API/Interfaces, Documentation, Security, Performance
- **Change the Behavior of Copilot** Customize Copilot’s behavior for specific tasks or workflows
- **Tool Restrictions**
  Limit available tools to relevant actions
- **LLM model selection** Set which LLM model to use for the chatmode file

Note

Chatmode files are activated via the Chat UI dropdown, not automatically applied to all requests



Use Chatmode Files

Enable and create in VSCode

- Create `.github/chatmodes/` directory
- Add an description
- Configure tools and instructions in frontmatter
- Set the model to use
- Add your desired instructions in markdown
- Use VS Code Chat UI to switch modes

Chatmode File Structure

```
---
description: 'Review code changes and suggest improvements'
tools: ['codebase', 'search', 'usages']
---

# Code Review Mode Instructions

You are in code review mode. Focus on:
1. Code quality and best practices
2. Potential bugs or security issues
3. Performance implications
4. Maintainability and readability
5. Adherence to project conventions

Provide specific, actionable feedback with code examples where appropriate.
```

- **Optional Header (Front Matter syntax):** `model` e.g. `model: GPT-4.1`

Chatmode Files vs Instruction & Prompt Files

Feature	Chatmode Files	Instruction Files	Prompt Files
Purpose	Task/workflow-specific	Project/file-specific	Reusable prompt snippets
Activation	UI dropdown/manual	Automatic (glob/file)	Manual (attach to chat/edit)
Tool restrictions	✅	❌	❌

- Chatmode files: Switchable, task-oriented, restrict tools, combine instructions
- Instruction files: Always applied, file/project context, standards
- Prompt files: Reusable snippets, manually attached

Best Practices for Chatmode Files

- Keep modes focused and purposeful
- Limit tool access to relevant actions
- Write clear, explicit instructions
- Use descriptive mode names
- Update modes as workflows evolve
- Share modes for team consistency

References & Further Reading

- GitHub Copilot Custom Chat Modes Blog
- VS Code Copilot Customization Docs

Advanced Chatmode File Example

```
Be concise, professional, and direct. Avoid repetitive confirmations or verbosity. Keep user updates short and relevant.

---

#### Example of Next Step Communication

* “Let me fetch the documentation for this library now.”
* “Found new links in the API docs, fetching those next.”
* “Tests passed on all edge cases. Solution is verified.”

---

Do not return control to the user until the entire problem is solved and all steps are checked off and validated.

---
```

- Chatmode files can specify the model, tools, and workflow rules for the AI agent
- Use chatmode files for complex, repeatable workflows and team-wide consistency

Pass to Marie