GitHub Copilot Context Variables (Hash Context)

Last Updated: December 2025

Context variables (prefixed with #) allow you to provide specific information to GitHub Copilot Chat, making responses more accurate and relevant to your current workspace and situation.

Core Context Variables

#codebase

Provides workspace-wide context including project structure, dependencies, and patterns.

Use Cases:

- Architectural decisions and design patterns
- Understanding project-wide conventions
- Framework and technology stack analysis
- Cross-file dependency analysis

Example:

```
How should I structure authentication in this project? #codebase
Analyze the current testing patterns across the codebase #codebase
```

#editor

Includes the visible content of the active editor tab.

Use Cases:

- Context about currently viewed file
- Understanding the editing session context
- Providing context without selecting specific text
- Multi-pane editor scenarios

Example:

```
Explain the design pattern used in this file #editor
How can I improve the performance of this implementation? #editor
```

#selection

Adds the currently selected text or code block as context.

Use Cases:

- Targeted analysis of specific code sections
- Refactoring specific functions or methods
- Explaining particular algorithms or logic
- Debugging specific code blocks

Example:

```
/explain #selection this algorithm's time complexity
/fix #selection memory leak in event handlers
/tests #selection comprehensive unit tests needed
```

`#file:<filename>`

References a specific file by name or pattern.

Use Cases:

- Comparing implementations across files
- Understanding file-specific patterns
- Referencing configuration or setup files
- Cross-file refactoring planning

Example:

```
Compare this implementation with #file:utils/helpers.js
How does this relate to #file:package.json dependencies?
Update this to match patterns in #file:components/Header.tsx
```

`#file` (Interactive)

Opens a quick picker to select files from your workspace.

Use Cases:

- When you're unsure of exact filename
- Browsing related files
- Including multiple file contexts
- Exploring project structure

Advanced Context Variables

`#sym:<symbol>`

References specific symbols (functions, classes, variables) in your workspace.

Use Cases:

- Understanding symbol usage across codebase

- Refactoring method signatures or class structures
- Analyzing dependencies and relationships
- Finding all references to specific symbols

Example:

```
Show all usages of #sym:UserService
How can I refactor #sym:calculateTotal for better performance?
```

`#sym` (Interactive)

Opens a symbol picker for workspace-wide symbol search.

Use Cases:

- Exploring available symbols
- Finding related functions or classes
- Understanding symbol relationships
- Code navigation assistance

Terminal Context Variables

`#terminalSelection`

Includes currently selected text in the terminal.

Use Cases:

- Analyzing command output
- Understanding error messages
- Debugging command-line issues
- Script optimization

Example:

```
/explain #terminalSelection this error output
How do I fix #terminalSelection compilation errors?
```

`#terminalLastCommand`

Provides context about the most recently executed terminal command and its output.

Use Cases:

- Debugging failed commands
- Understanding build errors
- Optimizing CLI workflows
- Troubleshooting deployment issues

Example:

```
/fix #terminalLastCommand build failure
Explain why #terminalLastCommand failed
Alternative approaches to #terminalLastCommand
```

Specialized Context Variables

#VSCodeAPI

Provides VS Code Extension API context for extension development.

Use Cases:

- Building VS Code extensions
- Understanding VS Code APIs
- Extension development best practices
- VS Code integration patterns

Example:

```
How do I create a tree view with #VSCodeAPI?
Implement command registration using #VSCodeAPI
```

#git

References Git repository information and history.

Use Cases:

- Understanding commit history
- Branch management strategies
- Merge conflict resolution
- Git workflow optimization

Example:

```
Analyze recent changes in #git history
Help resolve #git merge conflicts
Suggest #git branching strategy
```

#npm / #package

References package.json and dependency information.

Use Cases:

- Dependency management
- Package updates and compatibility
- Build script optimization
- Dependency analysis

Example:

```
Update #package dependencies safely
Optimize #npm scripts for faster builds
```

Context Combination Strategies

Multi-Context Usage

Combine multiple context variables for comprehensive analysis:

```
# Architecture Analysis
Analyze the relationship between #file:controllers/UserController.js and
#sym:UserService #codebase

# Debugging with Full Context
Why is #selection failing when #terminalLastCommand shows no errors?
#file:package.json

# Comprehensive Code Review
Review #selection for security issues, considering #codebase patterns and
#file:security-config.js
```

Progressive Context Building

Build context progressively through conversation:

```
1. "What's the overall architecture of this project? #codebase"
2. "How does #file:UserService.js fit into this architecture?"
3. "Can you refactor #selection to follow the established patterns?"
```

Best Practices

1. Choose the Right Context Level

Task Scope	Recommended Context	Example
Function-level	`#selection`	Refactor specific method
File-level	`#file:filename`	Understand file structure

Task Scope	Recommended Context	Example
Module-level	`#file` + related files	Cross-file refactoring
Project-level	`#codebase`	Architectural decisions

2. Context Specificity

```
❌ Generic: "Fix this code #codebase"
✅ Specific: "Fix memory leak in #selection event handlers based on
patterns in #codebase"
```

3. Contextual Relevance

```
❌ Irrelevant: "#file:README.md help with database query optimization"
✅ Relevant: "#selection optimize this database query
#file:database/config.js"
```

4. Progressive Disclosure

Start broad, then narrow:

```
1. "What testing patterns does #codebase use?"
2. "Generate tests for #file:UserService.js following those patterns"
3. "/tests #selection with edge case coverage"
```

Context Variable Reference

Variable	Scope	Best For	Example
`#codebase`	Entire workspace	Architecture, patterns	Project structure analysis
`#editor`	Active editor	Current file context	File-level understanding
`#selection`	Selected text	Targeted analysis	Function refactoring
`#file:name`	Specific file	File comparison	Cross-file patterns
`#sym:name`	Symbol references	Symbol analysis	Refactoring impacts
`#terminalSelection`	Terminal selection	Command output	Error analysis
`#terminalLastCommand`	Last command	Command debugging	Build troubleshooting
`#VSCodeAPI`	VS Code API	Extension development	VS Code integration

Drag and Drop Context

File Drag and Drop

- Drag files from explorer into chat
- Automatically adds file context
- Works with multiple files
- Preserves file relationships

Code Snippet Drag and Drop

- Drag selected code into chat
- Maintains syntax highlighting
- Preserves indentation and formatting
- Includes line number references

Troubleshooting Context Issues

Problem	Solution	Example
Context too broad	Use more specific context	`#selection` instead of `#codebase`
Missing file context	Verify file exists and is saved	Check file path spelling
Symbol not found	Ensure symbol is in workspace	Rebuild symbol index if needed
Terminal context empty	Run command first	Execute command before referencing
Large context limit	Break into smaller requests	Use progressive context building

Enterprise Context Features

Custom Context Providers

- Organization-specific context sources
- Internal API documentation context
- Company coding standards context
- Private repository context

Context Security

- Respects file access permissions
- Honors .gitignore patterns
- Excludes sensitive configuration files
- Audit logging for context usage

Context Performance Tips

Efficient Context Usage

- Use smallest relevant context scope
- Combine related contexts in single request
- Cache frequently used context patterns

- Avoid redundant context in follow-ups

Context Limits

- Be aware of token limits for large contexts
- Prioritize most relevant context sections
- Use progressive context revelation
- Consider breaking large analyses into steps

Remember: Effective context usage is key to getting precise, relevant assistance from GitHub Copilot Chat!