GitHub Copilot Editing Session

Copilot Edits enables AI-powered multi-file editing sessions where you can iterate quickly on code changes across your entire workspace. This feature allows for complex refactoring, feature implementation, and codebase-wide improvements.

What is Copilot Edits?

Copilot Edits is an advanced feature that allows you to:

- **Make coordinated changes** across multiple files simultaneously
- **Iterate on complex edits** with AI assistance throughout the process
- **Preview changes** before applying them to your codebase
- **Maintain context** across editing sessions for better consistency
- **Create new files** as part of a larger change
- **Review and accept/reject** changes before applying

Opening Copilot Edits

VS Code:

- Open Command Palette: `Cmd+Shift+P` (Mac) / `Ctrl+Shift+P` (Windows)
- Search: "Copilot Edits: Open"
- Or click the Copilot Edits icon in the sidebar

The Edits Interface:

1. **Working Set**: Files included in the editing session
2. **Prompt Area**: Describe the changes you want
3. **Preview Panel**: See proposed changes before applying
4. **Accept/Reject**: Control which changes to apply

Starting an Editing Session

1. Working Set Selection

Add files to your working set that will be part of the editing session:

- **Primary files**: Files you want to directly modify
- **Context files**: Files that provide important context (interfaces, types, configurations)
- **Related files**: Files that might be affected by changes

2. Session Initialization

```
# Example session start prompts:
"Start editing session: refactor user authentication to use JWT tokens"
"Begin multi-file edit: implement dark mode feature across components"
"Create editing session: add TypeScript strict mode to entire project"
```

3. Iterative Development

- Make specific requests for each iteration
- Review proposed changes before accepting
- Provide feedback to refine edits
- Add or remove files from working set as needed

Best Practices for Effective Editing Sessions

Be Specific and Precise

```
❌ Vague: "Make the code better"
✅ Specific: "Extract database connection logic into a singleton service
class"

❌ Generic: "Add error handling"
✅ Precise: "Add try-catch blocks with custom exceptions for all API calls"
```

Decompose Large Tasks

```
Large Task: "Migrate from REST to GraphQL"

Decomposed:
1. "Create GraphQL schema definitions for existing REST endpoints"
2. "Implement GraphQL resolvers for user-related operations"
3. "Update frontend queries to use GraphQL instead of REST"
4. "Add error handling and validation to GraphQL resolvers"
5. "Update tests to work with new GraphQL implementation"
```

Maintain Context Throughout Session

```
# Session flow example:
1. "Create TypeScript interfaces for user data structures"
2. "Now implement UserService class using those interfaces"
3. "Update existing components to use the new UserService"
4. "Add unit tests for the UserService methods"
```

Advanced Editing Patterns

Cross-File Refactoring

```
"Refactor authentication logic:
- Extract AuthService from LoginComponent
- Update all components using auth to use AuthService
- Add proper error handling and loading states
- Update tests to mock AuthService"
```

Framework Migrations

```
"Migrate React class components to functional components:
- Convert components in /components directory
- Update lifecycle methods to use hooks
- Maintain existing prop interfaces and behavior
- Update corresponding test files"
```

Architecture Improvements

```
"Implement clean architecture pattern:
- Create domain entities and use cases
- Add repository interfaces and implementations
- Update controllers to use dependency injection
- Add proper error boundaries and exception handling"
```

Working Set Management

Optimal Working Set Size

- **Small sessions (3-8 files)**: Best for focused refactoring or feature additions
- **Medium sessions (8-15 files)**: Good for module-level changes or migrations
- **Large sessions (15+ files)**: Use carefully, break into smaller iterations

File Selection Strategy

```
# Core files (always include):
- Files being directly modified
- Interface/type definition files
- Configuration files

# Context files (include when relevant):
- Related service/utility files
- Test files for modified components
- Documentation files that need updates

# Reference files (add as needed):
- Similar implementations for pattern consistency
- Framework configuration files
- External API interfaces
```

Code Generation from Chat Integration

Seamless Chat-to-Edit Flow

1. **Explore in Chat**: Discuss implementation approaches
2. **Generate Code**: Ask for specific code examples

3. **Apply to Project**: Use "Apply in Editor" or "Create New File"
4. **Iterate in Edits**: Refine and extend using editing session

Example Workflow

```
Chat: "How should I implement user authentication with JWT?"
→ Copilot provides implementation options

Chat: "Generate a complete AuthService class with login/logout methods"
→ Copilot generates code block

Action: Click "Create New File" → AuthService.ts created

Edit Session: "Now update all components to use this AuthService"
→ Multi-file editing session begins
```

Advanced Features

Context-Aware Editing

- **Framework Intelligence**: Understands React, Vue, Angular, Spring Boot patterns
- **Language Conventions**: Follows language-specific best practices
- **Project Patterns**: Maintains consistency with existing codebase patterns
- **Dependency Management**: Updates imports and dependencies automatically

Smart Preview and Validation

- **Change Preview**: See all modifications before applying
- **Conflict Detection**: Identifies potential merge conflicts
- **Dependency Validation**: Ensures imports and references remain valid
- **Test Impact Analysis**: Identifies tests that may need updates

Collaborative Editing

- **Team Consistency**: Follows shared coding standards and patterns
- **Code Review Integration**: Generates change summaries for PR descriptions
- **Documentation Updates**: Automatically updates related documentation

- **Changelog Generation**: Creates commit messages and change logs

Troubleshooting Editing Sessions

Common Issues and Solutions

Problem	Cause	Solution
Edits not applied correctly	Outdated file context	Refresh working set, save all files
Inconsistent changes	Missing context files	Add related files to working set
Large number of conflicts	Working set too large	Break into smaller, focused sessions
Context loss between iterations	Session too long	Restart session with clear objectives

Performance Optimization

```
# For better performance:
- Keep working sets under 20 files when possible
- Use specific, actionable prompts
- Save files frequently during session
- Close unused files to reduce context overhead
```

Enterprise and Team Features

Team Editing Standards

- **Shared Templates**: Organization-wide editing session templates
- **Code Review Integration**: Automatic PR generation from editing sessions
- **Compliance Checking**: Ensure edits meet security and quality standards
- **Audit Trails**: Track all AI-generated changes for compliance

Advanced Workflow Integration

- **CI/CD Integration**: Run tests automatically after editing sessions
- **Branch Management**: Create feature branches for editing sessions

- **Deployment Pipelines**: Integrate with deployment workflows
- **Quality Gates**: Enforce code quality checks before applying edits

Session Types and Templates

Common Session Templates

Feature Implementation

```
Working Set: [ComponentFiles, TestFiles, TypeDefinitions, ConfigFiles]
Prompt: "Implement [FeatureName] with:
- New components following existing patterns
- Comprehensive error handling
- Unit and integration tests
- Updated documentation"
```

Bug Fix Session

```
Working Set: [AffectedFiles, TestFiles, RelatedUtilities]
Prompt: "Fix [BugDescription] by:
- Identifying root cause
- Implementing robust solution
- Adding regression tests
- Updating error handling"
```

Refactoring Session

```
Working Set: [TargetFiles, InterfaceFiles, TestFiles, Documentation]
Prompt: "Refactor [Component/Module] to:
- Improve code organization and readability
- Extract reusable utilities
- Update tests and documentation
- Maintain existing API contracts"
```

Tips for Success

Session Planning

1. **Define clear objectives** before starting
2. **Identify all affected files** upfront
3. **Plan iteration steps** to avoid context loss
4. **Prepare rollback strategy** for complex changes

During the Session

1. **Review each iteration** before accepting changes
2. **Test incrementally** to catch issues early
3. **Save frequently** to preserve progress
4. **Provide specific feedback** to guide AI improvements

After the Session

1. **Run comprehensive tests** to verify changes
2. **Review all modifications** for consistency
3. **Update documentation** as needed
4. **Create meaningful commit messages** describing changes

Common Multi-File Patterns

Add New Field Across Layers

```
Prompt: Add a "category" field (String) to the Expense:
- Add to Expense.java entity with JPA annotations
- Add to ExpenseDTO.java
- Update ExpenseMapper to map the field
- Add filter by category to ExpenseRepository
- Update ExpenseService with category filtering
- Add category parameter to controller endpoints
```

Rename Across Codebase

```
Prompt: Rename the class "ExpenseManager" to "ExpenseService":
- Update class name and file name
- Update all imports
- Update all references
- Update test files
```

Add New Feature

```
Prompt: Add expense export functionality:
- Create ExportService.java with CSV export method
- Create ExportController.java with /api/expenses/export endpoint
- Add export button to expenses.html template
- Create ExportServiceTest.java with unit tests
```

Implement Interface

```
Prompt: Create an Auditable interface and implement it:
- Create Auditable.java interface with createdAt, updatedAt, createdBy
- Implement in Expense.java entity
- Implement in Category.java entity
- Add JPA auditing configuration
```

Practical Exercises

Exercise 1: Add a New Field

1. Open Copilot Edits
2. Add Entity, DTO, Service, Controller to working set
3. Prompt: "Add a 'tags' field (List) across all layers"
4. Review and apply changes
5. Verify the application compiles

Exercise 2: Refactor Package Structure

1. Add multiple files to working set

2. Prompt: "Move all DTOs from com.expense.dto to com.expense.model.dto"
3. Review import updates
4. Apply and verify

Exercise 3: Add Complete Feature

1. Prompt: "Create a complete Category management feature with CRUD operations"
2. Review generated files
3. Accept/reject individual changes
4. Fill in any gaps manually

Copilot Edits transforms how you approach complex code changes, enabling rapid iteration and comprehensive refactoring while maintaining code quality and consistency across your entire project.