FACILITATOR_GUIDE

This document provides a detailed facilitator's guide for running the 3-hour workshop.

Timing and Flow Tips

- **Keep labs hands-on**: Minimize lecture time, maximize coding time
- **Walking around**: Circulate during labs to help with issues
- **Backup plans**: Have pre-built examples ready if participants struggle
- **Flex time**: Each section has 5-10 minutes of flex time built in

- **Energy management**: Take 5-minute breaks between major sections
- **Advanced participants**: Have optional extension exercises ready
- **Copilot Instructions**: Repository uses `.github/copilot-instructions.md` for automatic configuration - no manual setup required by participants!

---ilitator's Guide: Using AI for Application Development with GitHub Copilot (.NET Edition)

This document provides a detailed facilitator's guide for running the 3-hour workshop.

0. Kickoff & Setup (0:00 – 0:15, 15 min)

You do:

- Welcome participants, introduce goals: *"We'll learn how to use AI (Copilot) to help with requirements, code, tests, docs, and workflow in .NET projects."*

- Explain **Copilot Instructions** concept and `.github/copilot-instructions.md` approach.

- Quick demo: show that the repository already has instructions configured automatically.

- Present the **Workshop Presentation Deck** to guide the session and provide visual context.

Participants do:

- Confirm environment:

  - VS Code open

  - GitHub Copilot enabled

  - .NET 9 SDK installed (`dotnet --version`)

  - Clone the repository and checkout the `main` branch:

    ```
    git clone https://github.com/centricconsulting/ai-coding-workshop.git
    cd ai-coding-workshop
    git checkout main
    ```

- Open the repository in VS Code

- **Copilot Instructions are automatically active** via `.github/copilot-instructions.md` (no manual setup needed!)

- Verify build: `dotnet build` and `dotnet test`

0.5. GitHub Copilot Features Overview (0:15 – 0:30, 15 min)

You do - Quick tour of Copilot capabilities:

Inline Completions (Ghost Text)

- As you type, Copilot suggests code in gray text
- Press `Tab` to accept, `Esc` to dismiss
- `Alt+]` / `Alt+[` to cycle through suggestions
- Works in comments, code, and tests

Copilot Chat Panel (`Ctrl+Alt+I` or `Cmd+Shift+I`)

- Open chat interface for conversational coding
- Ask questions: *"How do I configure logging in .NET?"*
- Request code: *"Create a repository interface for Task entity"*
- Iterate on solutions without leaving VS Code

Inline Chat (`Ctrl+I` or `Cmd+I`)

- Quick chat directly in your editor at cursor position
- Perfect for small edits: *"Add error handling"* or *"Make this method async"*
- Less context switching than full chat panel

Slash Commands - Shortcuts for common tasks:

- `/explain` - Understand code functionality
- `/fix` - Suggest fixes for errors or bugs
- `/tests` - Generate unit tests for selected code
- `/doc` - Create documentation comments
- `/refactor` - Improve code structure
- `/new` - Scaffold new files or projects
- `/clear` - Clear chat history

Demo: Show `/tests` on a method, `/explain` on complex code

📝 Plan First with Agents (Demo)

Goal: Demonstrate how Copilot (in Agent Mode) or a custom agent like `@planner` can propose a step-by-step plan before any code is changed.

How to show it:

- In Copilot Chat (Agent Mode), ask: "Propose a step-by-step plan to refactor LegacyTaskProcessor to use async/await, add logging, and follow Object Calisthenics."
- Review the plan with the group. Edit or reorder steps as needed.
- Only then, ask Copilot (or `@engineer`) to implement the plan, step by step.

Discussion:

- Why is planning first valuable?
- Did the plan catch any issues or clarify the approach?
- How does this workflow compare to direct code generation?

Tip: Encourage participants to always ask for a plan before executing large or multi-file changes.

Agent Mode & MCP Tools

Agent Mode allows Copilot to operate more autonomously, chaining together multiple steps, tools, and reasoning to solve complex tasks. This is especially powerful for:

- Multi-file or cross-cutting changes
- Automated codebase analysis or refactoring
- Running evaluation or test suites with minimal manual intervention
- Integrating with Model Context Protocol (MCP) tools for advanced workflows

MCP Tools provide specialized capabilities (e.g., evaluation, tracing, agent orchestration) that can be invoked directly in Agent Mode. These tools enable:

- Automated evaluation of code or models
- Tracing and observability for AI workflows
- Agent-driven code generation and review

How to use Agent Mode:

1. Select "Agent" from the Copilot agents dropdown (or use the command palette).
2. Describe your goal in natural language (e.g., "Refactor all service classes to use async/await and add logging").
3. For advanced scenarios, reference MCP tools directly (e.g., "Evaluate this model using the aitk-evaluation_planner tool").
4. Review the proposed plan and results; iterate as needed.

When to use Agent Mode:

- When a task spans multiple files or requires orchestration
- For codebase-wide refactoring or analysis
- To automate repetitive or evaluation-heavy workflows
- When you want Copilot to propose and execute a plan, not just a single edit

Facilitator Tips:

- Encourage participants to try Agent Mode for at least one lab (e.g., Lab 3 or 4)
- Compare results from Ask/Edit vs. Agent Mode
- Demonstrate invoking an MCP tool (e.g., evaluation or tracing) and discuss the output

Example prompt:

> "Use Agent Mode to generate unit tests for all public methods in the Application layer, then run the tests and summarize the results."

Chat Participants (Agents) - Specialized assistants:

- `@workspace` - Answers about your entire codebase
  - *"@workspace Where is the Task entity defined?"*
  - *"@workspace How is logging configured?"*
- `@vscode` - VS Code settings and commands
  - *"@vscode How do I change theme?"*
- `@terminal` - Terminal commands and shell help
  - *"@terminal How do I run tests in watch mode?"*
- `@azure` - Azure-specific guidance (if available)

Demo: Show `@workspace` finding code across solution

Context Variables - Provide specific context:

- `#file` - Reference specific files
  - *"Refactor #file:TaskService.cs to use dependency injection"*
- `#selection` - Reference selected code
  - *"Add unit tests for #selection"*
- `#editor` - Current file context
- `#terminalSelection` - Selected terminal output

Demo: Select code, use `#selection` in chat

Copilot Edits (Multi-file editing)

- Edit multiple files at once with AI guidance
- Add files to working set, describe changes
- Copilot proposes changes across all files
- Review and accept/reject changes

Demo: Add multiple files, request cross-cutting change

Participants do (Quick practice):

1. Try inline completion by typing a method comment
2. Open Copilot Chat (`Ctrl+Alt+I`), ask: *"What testing frameworks are used in this project?"*
3. Select a method, use Inline Chat (`Ctrl+I`): *"Add XML documentation"*
4. Try a slash command: `/explain` on any method
5. Use `@workspace`: *"@workspace Where is ITaskRepository implemented?"*

---

1. Controlling Context with Copilot Instructions (0:30 – 1:00, 30 min)

You do:

- Explain why *context matters* for Copilot output.
- Show the `.github/copilot-instructions.md` file in the repository.
- Explain that this file automatically configures Copilot for everyone working in this repo (no manual setup needed).
- **Emphasize Section 1: TDD Workflow** - "When asked to implement a feature, propose/emit tests before code"
- Show difference with/without instructions (e.g., generate a class, note coding style vs messy defaults).
- Highlight key instructions in the file:
  - **TDD first**: Write tests before implementation
  - Coding style (file-scoped namespaces, `nameof`, async/await, sealed classes)
  - Clean Architecture project layout (Domain/Application/Infrastructure/API)
  - DDD aggregates and value objects
  - Test rules (xUnit + FakeItEasy, organize by feature, class-per-method folders)
  - Conventional commits
  - OpenTelemetry for observability

Participants do (Lab 1) - Following TDD Red-Green-Refactor:

Scenario: Create a `NotificationService` that sends task notifications via email and SMS.

Step 1: Create Interface First (Design)

1. Ask Copilot Chat: *"Create an INotificationService interface in the Application layer for sending email and SMS notifications about tasks. Include methods for both individual and combined notifications."*
2. Review generated interface - should be in `src/TaskManager.Application/Services/INotificationService.cs`

Step 2: Write Tests FIRST (Red)

1. Ask Copilot: *"Create xUnit tests for NotificationService in the pattern specified in .github/copilot-instructions.md. Organize tests by method with separate test classes. Use FakeItEasy for mocking ILogger. Test happy path and all guard clauses."*
2. Review test structure: Should create folder `tests/TaskManager.UnitTests/Services/NotificationServiceTests/` with separate test classes per method
3. Run tests: `dotnet test` - **Tests should FAIL** (Red) because NotificationService doesn't exist yet

Step 3: Implement Code (Green)

1. Ask Copilot: *"Implement NotificationService that passes all the tests. Follow the coding style in .github/copilot-instructions.md: sealed class, file-scoped namespace, ILogger dependency injection, async/await, guard clauses with nameof."*

2. Review implementation - verify it follows all conventions:

- ✅ `sealed class`
- ✅ File-scoped namespace
- ✅ Constructor with `ILogger` and null check using `nameof`
- ✅ Async methods with proper `CancellationToken` support
- ✅ Guard clauses at method start (fail fast, no else)
- ✅ Structured logging with parameters

1. Run tests: `dotnet test` - **Tests should PASS** (Green)

Step 4: Observe & Reflect (Refactor)

1. Review the generated code quality:

    - Does it follow Clean Architecture (Application layer, no infrastructure concerns)?
    - Are test names descriptive?
    - Is the code intention-revealing?

2. Try asking Copilot: *"Are there any improvements we could make to this code?"*

Key Learning Points to Emphasize:

- ✅ **TDD enforces design thinking** - interface and tests force you to think about API before implementation
- ✅ **Copilot respects instructions** - consistent style across all generated code
- ✅ **Tests document behavior** - reading tests tells you exactly what the service does
- ✅ **Red-Green-Refactor cycle** - see tests fail, then pass, then improve

- ⚠️ **Don't skip the "Red" step** - if you write implementation first, you miss design feedback from tests

Common Mistakes to Call Out:

- ❌ Asking for implementation before tests (violates TDD)
- ❌ Not organizing tests by feature/method (makes tests hard to navigate)
- ❌ Accepting code without verifying it follows instructions
- ❌ Not running tests after each step

---

2. Requirements → Backlog → Code (1:00 – 1:45, 45 min)

You do:

- Introduce the idea: AI can turn **requirements** → **backlog items** → **tests** → **code**.
- Demo:
  User story: "As a user, I want to manage a list of tasks so I can track progress."
  → Copilot generates backlog items (stories), acceptance criteria, test stubs.

Participants do (Lab 2):

1. Write prompt: *"Generate 3 backlog items for a task manager, with acceptance criteria."*
2. Pick one (e.g., Add Task).
3. Generate a unit test skeleton in xUnit for `AddTask`.
4. Implement `TaskService.AddTask` with Copilot.
5. Run `dotnet test` → verify.

---

3. Code Generation & Refactoring in .NET (1:45 – 2:30, 45 min)

You do:

- Show Copilot scaffolding: create a `TasksController` with minimal API.
- Show refactor of messy method (provided in repo):
  - Before: long function, nested ifs, poor naming
  - After: Copilot helps split into smaller methods, add async, ILogger logging.

Participants do (Lab 3):

1. Use `@workspace` to understand the API structure: *"@workspace Show me the API endpoint extensions"*
2. Scaffold minimal Web API endpoints using Chat:
   - `GET /tasks/{id}` - *"Implement the GetTaskByIdAsync endpoint in EndpointExtensions"*
   - `POST /tasks` - Use inline chat (`Ctrl+I`) with `#file:EndpointExtensions.cs`
3. Use `/refactor` on the `LegacyTaskProcessor.ProcessTaskBatch` method:
   - Select the method, Chat: `/refactor enforce guard clauses and add async`
   - Or use Inline Chat: *"Refactor this to use async/await and add logging"*
4. Use `/tests` on refactored code to generate unit tests
5. Re-run `dotnet build && dotnet test`.

---

4. Testing, Documentation, Workflow (2:30 – 2:45, 15 min)

You do:

- Show Copilot generating:
  - xUnit tests using `/tests` command
  - README docs using `/doc` command
  - Commit message using Chat with staged changes context
  - PR summary with `@workspace` for full context

Participants do (Lab 4):

1. Select a method in `TaskService`, use `/tests` to generate xUnit tests
2. Use `/doc` to generate XML documentation for a class or method
3. Stage changes (`git add`), then use Chat: *"Write a Conventional Commit message for these staged changes"*
4. Ask Chat: *"@workspace Draft a PR description including intent, scope, and risks for the changes I made"*
5. Generate a README section: *"Create a Getting Started section for the API in #file:README.md"*

---

5. Wrap-Up & Discussion (2:45 – 3:00, 15 min)

You do:

- Recap: where Copilot helped (backlog shaping, scaffolding, refactoring, testing, docs, workflow).
- Call out **anti-patterns**:
  - Prompt roulette (unversioned prompts, inconsistent results)
  - Over-trusting Copilot without tests
  - Letting AI sneak domain logic into API layer
- Next steps:
  - Standardize Copilot Instructions in team repos
  - Build shared prompt/playbook library
  - Apply to real legacy code modernization

Participants do:

- Share takeaways.
- Ask Q&A: where would they use this tomorrow?

Troubleshooting Common Issues

Copilot Not Working

- **Check subscription**: Verify active GitHub Copilot subscription
- **Extension enabled**: Ensure Copilot extension is installed and enabled in VS Code
- **Authentication**: Sign out and back in to GitHub in VS Code
- **Instructions not loading**:
  - Ensure you're working in the repository root (where `.github/` folder exists)
  - Restart VS Code to reload repository-level instructions

    - Check that `.github/copilot-instructions.md` exists in the repo
    - Try Command Palette → "GitHub Copilot: Restart Language Server"

.NET Build Issues

- **Wrong version**: Ensure .NET 9 SDK is installed (`dotnet --version`)
- **Missing dependencies**: Run `dotnet restore` in project directory
- **Path issues**: Use absolute paths or ensure correct working directory

Copilot Generating Wrong Code

- **Check instructions**: Verify workshop instructions are properly configured
- **Context matters**: Include relevant files in VS Code workspace
- **Prompt clarity**: Be specific about requirements and constraints
- **Restart Copilot**: Command Palette → "GitHub Copilot: Restart Language Server"

Deliverables Recap

- **Repo**: Clean Architecture solution with Domain/Application/Infrastructure/API layers in the `main` branch
- **Copilot Instructions**: `.github/copilot-instructions.md` (automatically applied, repository-level configuration)
- **Documentation**:
    - Main README with workshop outline
    - Facilitator's Guide (this document)
    - Detailed Lab Walkthroughs in `docs/labs/`
    - Starter Projects README with architecture details
- **Code Examples**: Console app, Web API with OpenTelemetry, legacy code for refactoring (LegacyTaskProcessor)
- **Test Infrastructure**: xUnit test stubs with FakeItEasy ready for participants

---

Appendix: GitHub Copilot Chat Participants & Custom Agents

GitHub Copilot offers both built-in chat participants and custom Copilot agents that provide specialized assistance for various development tasks. Understanding the difference and when to use each helps participants get the most relevant and accurate responses.

Built-in Chat Participants vs. Custom Copilot Agents

Built-in Chat Participants are VS Code's native Copilot features accessed with `@` syntax (e.g., `@workspace`, `@vscode`). These provide general-purpose assistance.

Custom Copilot Agents are specialized, configurable agents defined in `.agent.md` files in the `.github/agents/` directory. These provide role-specific guidance and can be selected from the agents dropdown in Copilot Chat.

Built-in Chat Participants

These are accessed using `@` syntax in Copilot Chat.

1. General Chat (Default)

When to use:

- General coding questions
- Conceptual explanations
- Architecture discussions
- Best practices inquiries

How to use:

- Simply type your question in chat without any prefix
- Or use `@copilot` explicitly

Examples:

```
What is the repository pattern in DDD?
How should I structure a .NET Web API project?
Explain the benefits of async/await in C#
```

Best for:

- ✅ Learning concepts
- ✅ Getting design advice
- ✅ Understanding patterns
- ✅ General programming questions

Limitations:

- ⚠️ No direct access to your codebase context (use @workspace for that)
- ⚠️ May give generic answers without project-specific context

---

2. @workspace

When to use:

- Questions about YOUR specific codebase
- Finding code across the project
- Understanding project structure
- Locating implementations or definitions

How to use:

- Type `@workspace` followed by your question
- Copilot will search and analyze your entire workspace

Examples:

```
@workspace Where is the Task entity defined?
@workspace How is logging configured in this project?
```

```
@workspace Find all implementations of IRepository
@workspace Show me how authentication is handled
@workspace What testing frameworks are used?
```

Best for:

- ✅ Code navigation and discovery
- ✅ Understanding existing implementations
- ✅ Finding patterns used in your project
- ✅ Locating specific classes, methods, or files
- ✅ Understanding project conventions

Workshop Tips:

- Emphasize `@workspace` in Labs 2-4 when participants need to understand existing code
- Show how it finds code across all layers (Domain, Application, Infrastructure, API)
- Demonstrate finding repository interfaces, endpoint patterns, test structures

---

3. @vscode

When to use:

- Questions about VS Code functionality
- Setting up extensions
- Configuring workspace settings
- Keyboard shortcuts
- Editor features

How to use:

- Type `@vscode` followed by your question

Examples:

```
@vscode How do I change the editor theme?
@vscode What's the keyboard shortcut for formatting code?
@vscode How do I configure auto-save?
@vscode How do I debug a .NET application?
@vscode How do I set up a launch configuration?
```

Best for:

- ✅ VS Code configuration
- ✅ Editor productivity tips
- ✅ Extension recommendations
- ✅ Debugging setup
- ✅ Workspace customization

Workshop Tips:

- Use when participants struggle with VS Code features
- Helpful for debugging configuration questions
- Good for keyboard shortcut discovery

---

4. @terminal

When to use:

- Shell command questions
- Terminal operations
- Command-line tool usage
- Script writing

How to use:

- Type `@terminal` followed by your question
- Ask about bash, zsh, PowerShell, or cmd commands

Examples:

```
@terminal How do I run tests in watch mode?
@terminal What command finds all .cs files?
@terminal How do I check git commit history?
@terminal Create a command to list all NuGet packages
@terminal How do I build in release mode?
```

Best for:

- ✅ Shell commands
- ✅ Git operations
- ✅ .NET CLI commands
- ✅ Build scripts
- ✅ Terminal productivity

Workshop Tips:

- Useful for participants unfamiliar with .NET CLI
- Help with git commands during Labs 4
- Good for showing batch operations

---

5. @azure (If Available)

When to use:

- Azure-specific questions
- Cloud deployment guidance
- Azure service configuration
- Azure CLI commands

How to use:

- Type `@azure` followed by your question

Examples:

```
@azure How do I deploy a .NET Web API to Azure App Service?
@azure What's the best Azure service for hosting this application?
@azure How do I configure Application Insights?
@azure Generate an Azure Bicep template for this project
```

Best for:

- ✅ Azure deployment strategies
- ✅ Service recommendations
- ✅ Configuration guidance
- ✅ Azure CLI commands
- ✅ Infrastructure as Code

Workshop Notes:

- Not core to this workshop but useful for deployment discussions
- Mention in wrap-up as next steps
- Good for "how would I deploy this?" questions

Custom Copilot Agents

Custom Copilot agents are specialized assistants defined in `.github/agents/*.agent.md` files. They are accessed by selecting them from the **agents dropdown** in Copilot Chat.

Available Custom Agents in This Workshop

1. **Check** - Code review and improvement suggestions
    - Reviews code for best practices, refactoring opportunities, and code smells
    - Use before submitting PRs
    - Located at `.github/agents/Check.agent.md`
2. **Architect** - Architecture and planning for documentation
    - Limited to Markdown files only
    - Creates detailed architectural plans and ADRs
    - Integrates with GitHub issue management
    - Located at `.github/agents/architect.agent.md`
3. **Plan** - Strategic planning and analysis
    - Read-only tools for research and planning
    - Develops comprehensive implementation strategies
    - Analyzes codebases before making changes
    - Located at `.github/agents/plan.agent.md`
4. **API Architect** - API design and implementation
    - Generates three-layer API designs (service, manager, resilience)
    - Creates fully implemented code
    - Located at `.github/agents/api-architect.agent.md`
5. **Expert .NET Software Engineer** - .NET best practices
    - Expert guidance on C#, SOLID, testing, performance
    - Located at `.github/agents/expert-dotnet-software-engineer.agent.md`

How to use custom agents:

1. Open Copilot Chat in VS Code
2. Click the **agents dropdown** at the top of the chat panel
3. Select the desired agent (e.g., Check, Plan, Architect)
4. Type your prompt - the agent's specialized instructions are automatically applied

---

Comparison Table

Type	Name	Access Method	Best Use Case	Example
Built-in	Default	General chat	Concepts, patterns, best practices	"What is Clean Architecture?"
Built-in	@workspace	`@workspace`	Find code, understand structure	"@workspace Where is ITaskRepository?"
Built-in	@vscode	`@vscode`	Settings, shortcuts, configuration	"@vscode How do I format on save?"
Built-in	@terminal	`@terminal`	Shell commands, scripts	"@terminal Run tests in watch mode"
Built-in	@azure	`@azure`	Deployment, Azure services	"@azure Deploy to App Service"
Custom Agent	Check	Agents dropdown	Code review before PR	Select Check, then "Review this code"
Custom Agent	Plan	Agents dropdown	Strategic planning	Select Plan, then "Plan implementation for feature X"
Custom Agent	Architect	Agents dropdown	Architecture docs	Select Architect, then "Design ADR for decision"

Teaching Chat Participants & Agents in the Workshop

Section 0.5 (Copilot Features Tour)

Demo Strategy (5-7 minutes on participants and agents):

1. **Show Default Chat**:

   ```
   What is the repository pattern?
   ```

   → Generic explanation

2. **Compare with @workspace**:

   ```
   @workspace Where is the repository pattern implemented?
   ```

   → Specific file locations in YOUR project

3. **Show the Difference**:

   - Default: Generic, educational
   - @workspace: Specific, actionable

4. **Quick @terminal Demo**:

   ```
   @terminal How do I run tests with detailed output?
   ```

   → Shows actual commands for their environment

Throughout Labs - Usage Guide

Lab	Recommended Tools	When to Use
Lab 1	Default, @workspace, Plan agent	Concepts, finding test patterns, planning tests
Lab 2	@workspace, Default, Architect agent	Finding entities, understanding layers, planning features
Lab 3	@workspace, @terminal, Check agent	Finding endpoints, refactoring commands, code review
Lab 4	@workspace, @terminal, Check agent	Git commands, understanding test structure, PR review

Common Participant Mistakes

1. **Using Default Instead of @workspace**:

   - **Mistake**: "Where is the Task entity?"
   - **Better**: "@workspace Where is the Task entity?"
   - **Result**: Specific file path vs. generic explanation

2. **Using @workspace for Concepts**:

   - **Mistake**: "@workspace What is dependency injection?"
   - **Better**: "What is dependency injection?" (default)
   - **Result**: Educational explanation vs. code search

3. **Not Using @terminal for Commands**:

   - **Mistake**: "How do I run tests?"
   - **Better**: "@terminal How do I run tests?"
   - **Result**: Generic vs. environment-specific commands

4. **Not Using Custom Agents for Specialized Tasks**:

   - **Mistake**: Asking default chat to review code
   - **Better**: Select **Check** agent from dropdown for code review
   - **Result**: Generic feedback vs. structured review following project standards

5. **Not Using Plan Agent Before Big Changes**:

   - **Mistake**: Diving into implementation without planning
   - **Better**: Select **Plan** agent to develop strategy first
   - **Result**: Ad-hoc changes vs. comprehensive, thought-out approach

---

Advanced Tips for Facilitators

Combining Chat Participants and Custom Agents

You can use chat participants and custom agents in sequence:

1. **Understand concept** (Default):

   ```
   Explain the CQRS pattern
   ```

2. **Find in code** (@workspace):

   ```
   @workspace Show me examples of commands and queries in this project
   ```

3. **Review code** (Check agent):

   Select **Check** agent from dropdown:

   ```
   Review the command handlers for best practices
   ```

4. **Run tests** (@terminal):

   ```
   @terminal Run tests for the command handlers
   ```

Context Variables with Participants and Agents

Combine chat participants and agents with context variables for precision:

```
@workspace What tests exist for #file:CreateTaskCommandHandler.cs?
```

```
@terminal How do I run tests in #file:CreateTaskCommandHandlerTests.cs?
```

With custom agents:

```
[Select Check agent] Review #file:TaskService.cs for improvements
```

When Participants and Agents Don't Help

Sometimes participants need to:

- **Read documentation**: Point them to official docs
- **Debug interactively**: Use VS Code debugger
- **Review logs**: Look at actual error messages
- **Ask human experts**: Some questions need human judgment

Selection Flowchart for Participants

```
START: I have a task
    ↓
Do I need specialized assistance?
    ↓ YES → Select custom agent from dropdown
    │         ├─ Code review? → Check
    │         ├─ Planning? → Plan
    │         ├─ Architecture docs? → Architect
```

```
    │         └─ API design? → API Architect
    ↓ NO
    ↓
Is it about MY code?
    ↓ YES → Use @workspace
    ↓ NO
    ↓
Is it about VS Code?
    ↓ YES → Use @vscode
    ↓ NO
    ↓
Is it about terminal/commands?
    ↓ YES → Use @terminal
    ↓ NO
    ↓
Is it about Azure/deployment?
    ↓ YES → Use @azure
    ↓ NO
    ↓
Use Default Chat
```

Troubleshooting

@workspace Not Finding Code

Symptoms: Says "I couldn't find..." for code that exists

Solutions:

1. Ensure all relevant files are in workspace (not excluded)
2. Wait for indexing to complete (check bottom status bar)
3. Reload window: `Cmd/Ctrl+Shift+P` → "Reload Window"
4. Try being more specific: include file names or paths

Custom Agents Not Appearing

Symptoms: Custom agents not showing in dropdown

Solutions:

1. Ensure `.agent.md` files are in `.github/agents/` directory
2. Files must have proper `chatagent` frontmatter
3. Reload VS Code window: `Cmd/Ctrl+Shift+P` → "Reload Window"
4. Check VS Code is version 1.106 or later

Chat Participants Not Available

Symptoms: @azure or other participants don't work

Solutions:

1. Check Copilot extension version (update if needed)
2. Verify subscription includes advanced features
3. Some modes require specific extensions installed

4. Try restarting VS Code

Wrong Mode Selected

Symptoms: Generic answers when you wanted specific ones

Solutions:

1. Check which mode you're using (shown in chat)
2. Rephrase with explicit mode: "@workspace [question]"
3. Clear chat and start over with correct mode

Practice Exercise for Participants

5-Minute Hands-On (during Section 0.5):

Ask participants to try each mode:

1. **Default Chat**:

   ```
   What is Test-Driven Development?
   ```

   Expected: Conceptual explanation of TDD

2. **@workspace**:

   ```
   @workspace Where are the xUnit tests located?
   ```

   Expected: Actual file paths in their project

3. **@terminal**:

   ```
   @terminal Show me the command to build and test
   ```

   Expected: `dotnet build && dotnet test` or similar

4. **@vscode**:

   ```
   @vscode How do I toggle the terminal?
   ```

   Expected: Keyboard shortcut (Ctrl+` or Cmd+`)

Debrief:

- Which mode gave the most useful answer for each question?
- When would you use each mode during the labs?

Key Takeaways for Facilitators

✅ **@workspace is most important** for this workshop - emphasize it!
✅ **Default chat** is good for learning concepts
✅ **@terminal** helps with .NET CLI commands

✅ **Modes are contextual** - teach when to use which
✅ **Practice makes perfect** - participants learn by using them

Don't Overwhelm:

- Focus on @workspace and default chat primarily
- Introduce other modes as needed during labs
- Reference this appendix for detailed explanations
- Let participants discover advanced usage naturally

Additional Resources

- GitHub Copilot Chat Documentation
- Using Chat Participants
- Copilot Context Variables