GitHub Copilot Training

Mastering AI as a development assistant

Today's Agenda

Full Day Schedule (09:00 - 17:00)

Morning Sessions:

- 09:00 - Welcome & Kickoff
- 09:20 - Chapter 1: LLM Fundamentals
- 09:40 - Chapter 2 & 3: GitHub Copilot & IDE
- 10:15 - Exercise 1: Refactor Rescue
- 10:40 - **Break**
- 10:55 - Chapter 4 & 5: Prompt Engineering & Advanced Workflows
- 12:00 - **Lunch**

Afternoon Sessions:

- 13:00 - Chapter 6: GitHub Platform Integration
- 14:00 - Chapter 7: AI Security Best Practices
- 14:30 - **Break**
- 14:45 - Exercise 2: Context Matters
- 16:15 - Chapter 8: Making Impact
- 16:45 - Wrap-up & Q&A
- 17:00 - **End**

Introduction

- **Who are you?**
- **What are your expectations for today?**
- **Your specific goal(s)?**

How This Course Is Set Up

Learning Approach

Balanced Format:

- **Theory & Explanations** - Understand the why
- **Live demos** - See it in action
- **Hands-on exercises** - Practice yourself
- **Team discussions** - Learn from each other

Ground Rules:

- Questions welcome anytime
- Share experiences
- Focus on practical application
- We'll explain concepts before diving in

Chapter 1: LLM Fundamentals

Understanding the Technology

What are LLMs?

Large Language Models Explained

Core Concept:

- Trained on vast code & text datasets
- Pattern recognition at scale
- Context-aware predictions
- Not just "autocomplete"

Copilot's Foundation:

- OpenAI Codex base
- Fine-tuned for programming
- Multi-language support
- Continuous improvement

How LLMs are Trained

From Internet Data to AI Assistant

Training Process:

```
1. Pre-training (yearly)
   └─ Download ~10TB of text
   └─ 6,000 GPUs cluster
   └─ Compress into neural network
   └─ Cost: ~$2M, Time: ~12 days
   └─ Result: Base model

2. Fine-tuning (weekly)
   └─ 100K+ Q&A examples
   └─ Human feedback loop
   └─ Train for ~1 day
   └─ Result: Assistant model
```

Key Insight: Billions of parameters adjusted to predict the next word

Neural Networks: The Black Box

Why AI is "Next Word Prediction on Steroids"

How It Works:

- Input: "The cat sat on a..."
- Neural network processes
- Output: "mat" (97% probability)
- Billions of parameters working together

The Mystery:

- We can measure it works
- We can't explain WHY it works
- Parameters are incomprehensible to humans

```
    Input
      ↓
[Neural Network]
100B parameters
      ↓
 Most Probable
    Output
```

Remember:

- AI gives the **most probable** answer
- Not necessarily the **most correct**
- Based on patterns in training data

LLMs: Probability Machines

Most Probable ≠ Most Correct

Critical Understanding:

- LLMs always output what's **statistically likely**
- Trained on code that exists, including bugs
- Can confidently suggest incorrect solutions
- Will hallucinate plausible-sounding APIs

Example Hallucinations:

```
// AI might suggest this API that doesn't exist:
var result = String.ParseJson(jsonString);

// Or mix syntax from different languages:
public async Task<> getData() => {  // Mixed C#/JS
    return await fetch(url);
}
```

Bottom Line: Always verify AI suggestions - they're optimized for probability, not correctness

Probabilistic ≠ Deterministic

Understanding AI Behavior

Key Differences:

Traditional Code	LLM/Copilot
Same input → Same output	Same input → Varied outputs
Rule-based logic	Pattern-based suggestions
100% predictable	Probabilistic responses
Binary correct/incorrect	Confidence scores

Implication: Verify suggestions, don't assume correctness

Tokens & Context Windows

How Copilot "Reads" Your Code

```
Input: "function calculateTotal"
Tokens: ["function", "calculate", "Total"]
          └─ ~4000 token window limit ─┘
```

Context Window:

- Current file content
- Open tabs (relevant)
- Recent edits
- Comments & docstrings

Best Practice: Keep relevant context visible

Hallucinations & Guardrails

When AI Gets Creative

Common Hallucinations:

- Inventing APIs that don't exist
- Mixing syntax between languages
- Creating plausible but wrong logic
- Referencing non-existent files

Your Guardrails:

- ✅ Review before accepting
- ✅ Run tests immediately
- ✅ Check API documentation
- ✅ Trust but verify

Why Copilot vs Generic Chatbots

Purpose-Built for Developers

Generic AI	GitHub Copilot
General knowledge	Code-focused training
No IDE integration	Deep IDE integration
Copy-paste workflow	In-line suggestions
Limited code context	Full repository awareness
Generic responses	Language-specific patterns

Bottom Line: Built by developers, for developers

AI: Your Learning Accelerator

Transform How You Learn and Grow

Learning Revolution:

- 📚 Instant explanations in context
- 🔍 Discover patterns across languages
- 🚀 Explore new tech safely
- ⏱️ Get unstuck in seconds, not hours
- 🎯 Focus on concepts, not syntax

Example Prompts:

- "Explain this LINQ query and show SQL equivalent"
- "How would this pattern work in Python?"
- "Explain this like I'm a junior developer"
- "Compare async/await in C# vs JavaScript"

Bottom Line: Accelerate from months to weeks in mastering new skills

AI: Your Productivity Partner

Focus on What Matters Most

AI Handles the Tedious:

- ✅ Boilerplate generation
- ✅ Test scaffolding
- ✅ Documentation
- ✅ Data models
- ✅ Error parsing
- ✅ Repetitive refactoring

You Focus On:

- 🧠 Business logic
- 💡 Creative solutions
- 🎯 User experience
- 📊 Performance optimization
- 🏗️ Architecture decisions
- 🤝 Stakeholder value

Result: More time for meaningful, impactful work

Chapter 2: GitHub Copilot

From LLM Theory to Developer Tool

From LLMs to GitHub Copilot

Bridging AI and Development

Generic LLMs:

- Broad knowledge base
- General purpose text generation
- No code specialization
- Limited IDE integration

GitHub Copilot:

- Code-focused training
- Developer-specific fine-tuning
- Deep IDE integration
- Repository context awareness

The Evolution: LLM → Codex → GitHub Copilot → Your AI Pair Programmer

The GitHub Advantage

Integrated Development Experience

Unique Benefits:

- Trained on public GitHub repositories
- Understands project patterns
- PR and issue integration
- Security scanning built-in
- Team collaboration features

Not Just Code Completion:

- Documentation generation
- Test creation
- Code review assistance
- Security remediation

Privacy & Code Security

Your Code, Your Control

What Copilot Does:

- Processes code locally first
- Sends context for suggestions
- No training on private repos (Business/Enterprise)
- Configurable data retention

What You Control:

- File/folder exclusions
- Organization policies
- Audit logging
- Data residency options

Copilot Editions

Updated Pricing (June 2025)

Individual	Business	Enterprise
Personal use	Team features	Full platform
GitHub Copilot in IDE	Everything in Individual	Everything in Business
GitHub Copilot Chat	Private code security	Advanced security
CLI assistance	Knowledge bases	Custom models
$10/month	**$21/user/month**	**$39/user/month**

What's New: Premium requests for advanced AI models

Premium Requests

Enhanced AI Capabilities

What are Premium Requests?

- Access to more advanced AI models (Claude, Gemini)
- Higher quality code generation
- Better understanding of complex requirements
- More accurate refactoring suggestions

Usage Limits:

- **Business:** 300 premium requests/user/month
- **Enterprise:** 1000 premium requests

When to Use Premium:

- Complex architectural decisions
- Critical business logic
- Large-scale refactoring

Success Patterns

Early Adopter Insights

✅ What Works:

- Clear, specific prompts
- Iterative refinement
- Context-rich development
- Test-driven approach

❌ What Doesn't:

- Vague requirements
- Blind acceptance
- Ignoring suggestions
- Fighting the tool

Common Misconceptions

Myth vs Reality

Myth: "It will replace developers"
Reality: Augments human creativity

Myth: "It writes perfect code"
Reality: Requires review and refinement

Myth: "It knows my entire codebase"
Reality: Limited context window

Myth: "It's just fancy autocomplete"
Reality: Understands intent and patterns

Chapter 3: Copilot in IDE

Core Features & Capabilities

So... What Does This Mean Inside My IDE?

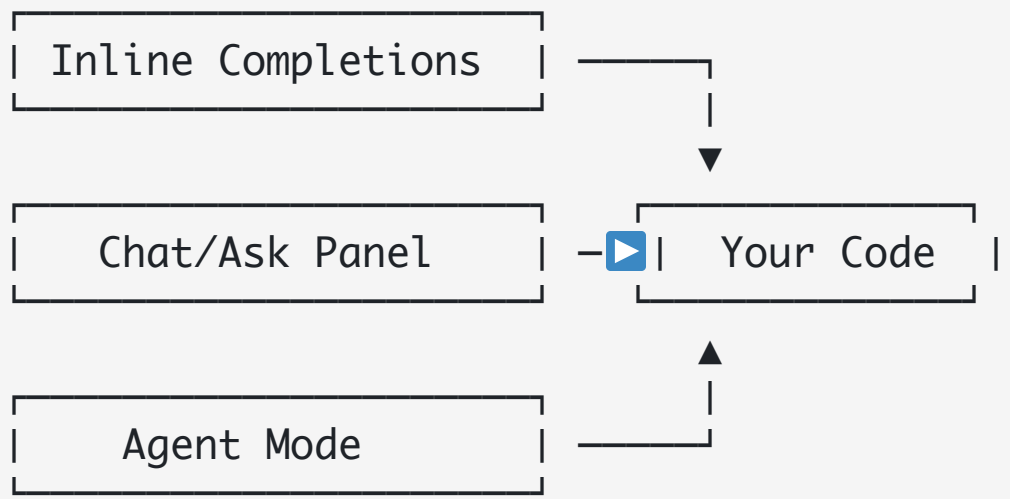
From Theory to Practice

Your Daily Workflow Changes:

Before Copilot	With Copilot
Google → Stack Overflow → Implement	Describe → Generate → Refine
Write boilerplate manually	Tab to accept suggestions
Context switch for docs	In-line documentation help
Debug alone	AI-assisted troubleshooting

Copilot Building Blocks

Three Modes of Interaction



Today's Journey:

1. Start with inline (quick wins)
2. Master Chat/Ask (morning focus)
3. Agent Mode (afternoon power)

Inline Completions

Quick Wins & Pitfalls

Quick Wins:

- Function signatures → implementations
- Test case generation
- Repetitive patterns
- Boilerplate code

Common Pitfalls:

- Accepting without reading
- Breaking naming conventions
- Inconsistent style
- Missing edge cases

Chat Panel Tour

Your AI Conversation Space

Key Features:

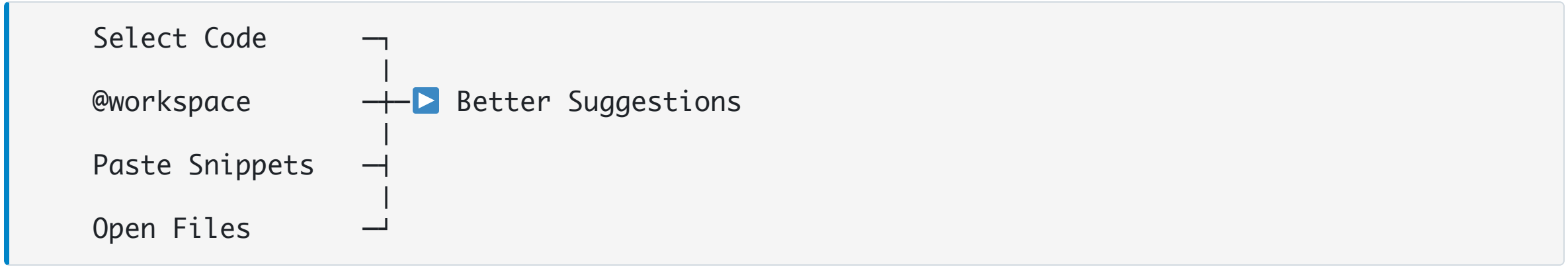
- 💬 Natural language queries
- 📝 Code block responses
- 🔧 Slash commands
- 📎 File attachments
- 🕒 Conversation history

Essential Commands:

- `/explain` - Understand code
- `/fix` - Debug errors
- `/tests` - Generate tests
- `/doc` - Add documentation

Managing Context in Chat

Feed the Right Information



Context Sources:

- Highlighted code blocks
- @workspace mentions
- Open editor tabs
- Pasted examples
- File references

Demo Plan

Refactoring Legacy Code with Chat

Scenario: Transform problematic legacy module

Steps:

1. Open messy legacy file
2. Ask Copilot to explain current behavior
3. Identify code smells together
4. Generate refactoring plan
5. Apply changes incrementally
6. Generate tests for validation

Demo - Initial Analysis

Understanding Legacy Code

```
// Legacy code example
public ArrayList ProcessData(ArrayList d) {
    var result = new ArrayList();
    for(int i=0; i<d.Count; i++) {
        dynamic item = d[i];
        if(item.active == true && item.value > 10) {
            result.Add(new {
                name = item.name.ToString().ToUpper(),
                val = item.value * 1.1
            });
        }
    }
    return result;
}
```

Chat Prompt: "Explain this function and identify improvement opportunities"

Demo - Refactoring Plan

AI-Generated Improvement Strategy

Copilot's Analysis:

1. Poor naming conventions
2. No type safety
3. Mixed concerns
4. No error handling
5. Outdated syntax

Suggested Refactoring:

- Modern ES6+ syntax
- Type definitions
- Separate filter/map
- Descriptive names
- Add validation

Demo - Implementation

Applying Refactoring

```
// After refactoring:
public List<ProcessedItem>
ProcessActiveItems(List<DataItem> items) {
    const decimal MIN_VALUE = 10;
    const decimal MULTIPLIER = 1.1m;

    return items
        .Where(item =>
            item.Active &&
            item.Value > MIN_VALUE)
        .Select(item => new ProcessedItem {
            Name = item.Name.ToUpper(),
            Value = item.Value * MULTIPLIER
        })
        .ToList();
}
```

Key Improvements:

- ✅ Type safety with generics
- ✅ Functional LINQ approach
- ✅ Named constants
- ✅ Single responsibility
- ✅ Clear method intent
- ✅ No dynamic types

Benefits:

- Easier to test
- More maintainable
- Better performance
- Compile-time safety

Demo - Test Generation

Validating Our Refactoring

Chat Prompt: "Generate comprehensive tests for both versions"

```
[TestClass]
public class ProcessActiveItemsTests {
    [TestMethod]
    public void ProcessActiveItems_FiltersInactiveItems() {
        // Test inactive items are excluded
    }

    [TestMethod]
    public void ProcessActiveItems_FiltersItemsBelowThreshold() {
        // Test value threshold works
    }

    [TestMethod]
    public void ProcessActiveItems_TransformsValidItemsCorrectly() {
        // Test uppercase and multiplier
    }
}
```

Copilot generates: Complete test suite with edge cases, fixtures, and assertions

Pitfalls with Chat & Fixes

Common Issues & Solutions

❌ Common Pitfalls:

- Vague prompts
- Missing context
- Accepting blindly
- Over-engineering
- Wrong assumptions

✅ How to Fix:

- Be specific with requirements
- Select relevant code first
- Always review suggestions
- Ask for simple solutions
- Provide constraints upfront

Remember: Copilot is your pair programmer, not your replacement

Exercise 1: Refactor Rescue

Hands-On with Chat

Exercise #1 - Refactor Rescue

Supermarket Receipt Refactoring Kata

Scenario: You've inherited a supermarket receipt system with special deals

Your Tasks:

1. 🔍 Use Chat to understand the pricing logic
2. 👃 Identify code smells (Long Method, Feature Envy)
3. ✅ Generate tests for 90%+ coverage
4. 🔧 Refactor while maintaining functionality
5. 🆕 Add new feature: 10% bundle discounts

Time: 25 minutes

Understanding the Domain

Current Special Deals System

Example Deals:

- 🪥 Buy 2 toothbrushes, get 1 free (€0.99 each)
- 🍎 20% discount on apples (€1.99/kg)
- 🍚 10% discount on rice (€2.49/bag)
- 🦷 5 tubes toothpaste for €7.49 (€1.79 each)
- 🍅 2 boxes cherry tomatoes for €0.99 (€0.69 each)

New Bundle Feature:

- Bundle: 1 toothbrush + 1 toothpaste = 10% off total
- Only complete bundles get discount
- Partial bundles = no discount

Exercise Rules & Deliverables

What You'll Produce

Repository: Supermarket Receipt Refactoring Kata

- Start with `main` branch (no tests)

Requirements:

- Achieve 90%+ test coverage
- Maintain all existing functionality
- Apply relevant refactorings

Deliverables:

- Comprehensive test suite
- Refactored receipt system
- New bundle discount feature (10% off complete bundles)

Prompt Checklist

Effective Chat Strategies

DO:

- ✅ Select code before asking
- ✅ Be specific about requirements
- ✅ Ask for explanations first
- ✅ Request tests with changes
- ✅ Iterate on suggestions

DON'T:

- ❌ Use Agent Mode (Chat only!)
- ❌ Accept without review
- ❌ Rush through changes
- ❌ Ignore test failures

Ready, Set, Go!

Repository:

https://github.com/OnCore-NV/Refactoring-Kata

Exercise Goals:

1. 🔍 Understand pricing logic with Chat
2. 👃 Identify code smells
3. ✅ Generate tests (90%+ coverage)
4. 🔧 Refactor the code
5. 🆕 Add bundle discount feature

Remember: Chat only, no Agent Mode!

Timeline (25 min):

```
Start (0:00)
├── Understand code (0:00-0:05)
├── Identify issues (0:05-0:10)
├── Plan refactoring (0:10-0:15)
├── Implement changes (0:15-0:20)
└── Write tests (0:20-0:25)
```

Deliverables:

- Test suite with 90%+ coverage
- Refactored receipt system
- Bundle discount (10% off complete bundles)

Quick Shareback

Team Insights (2-3 teams)

Share Your Experience:

1. What surprised you?
2. Most useful prompt?
3. Biggest challenge?
4. Key learning?

Common Discoveries:

- Context matters immensely
- Iterative prompting works best
- Tests reveal hidden issues
- AI explanations aid understanding

Chapter 4: Prompt Engineering

Mastering AI Communication

The Art of Prompt Engineering

Your Words Shape AI Output

Why It Matters:

- Better prompts = Better code
- Reduces iterations needed
- Saves debugging time
- Improves code quality
- Increases productivity

Key Principle:

AI responds to what you ask, not what you meant

The Prompt Spectrum:

```
Vague ──────────────────────► Specific

"Make a function"
      ↓
"Create a validation function"
      ↓
"Create an email validation function"
      ↓
"Create a C# method that validates
email format using regex, returns
bool, handles null input"
```

Anatomy of a Good Prompt

Essential Components

1. **Context** - Set the scene
2. **Task** - What needs to be done
3. **Constraints** - Rules and requirements
4. **Format** - Expected output structure
5. **Examples** - When helpful

```
// Poor prompt:
// "fix this"

// Good prompt:
// "Refactor this C# method to use async/await pattern,
// maintain all existing functionality, add proper error
// handling with try-catch, and include XML documentation"
```

Prompt Patterns That Work

Proven Templates for Success

Code Generation:

```
"Create a [language] [type] that:
- [requirement 1]
- [requirement 2]
- Uses [specific pattern]
- Returns [type]"
```

Refactoring:

```
"Refactor this code to:
- Follow [principle/pattern]
- Improve [metric]
- Maintain [requirement]
- Use [technology]"
```

Debugging:

```
"This [language] code throws [error].
Expected: [behavior]
Actual: [what happens]
Please identify and fix the issue."
```

Learning:

```
"Explain this [concept/code]:
- Like I'm a [level] developer
- Include [specific aspects]
- Provide [examples]"
```

Context is King

What to Include for Better Results

Code Context:

- Related classes/interfaces
- Method signatures
- Data structures
- Business rules

Technical Context:

- Framework version
- Target environment
- Performance requirements
- Security constraints

Example with Context:

```
// Context: This runs in a high-traffic web API with
// strict 100ms response time requirement
// Task: Optimize this product search to use caching
```

Iterative Refinement

The Conversation Approach

```
Initial: "Create a user service"
    ↓
Better: "Create a C# service class for user management"
    ↓
Refined: "Create a C# service class with CRUD operations
         for users, using repository pattern"
    ↓
Optimal: "Create a C# UserService class implementing
         IUserService with async CRUD methods, using
         IUserRepository, include logging and validation"
```

Pro Tip: Each iteration builds on the previous - don't start over!

Common Prompt Mistakes

What to Avoid

❌ Too Vague:

- "Make it better"
- "Fix the bug"
- "Add some tests"
- "Improve performance"

❌ Too Broad:

- "Create an entire application"
- "Refactor everything"
- "Add all features"

❌ Missing Context:

- No language specified
- No requirements stated
- No constraints mentioned
- No expected behavior

❌ Conflicting Instructions:

- "Make it simple but handle all edge cases"
- "Quick solution with perfect error handling"

Prompt Engineering Frameworks

CRISP & STAR Methods

CRISP Framework:

- **C**ontext - Set the scene
- **R**ole - Define AI's role
- **I**nstructions - Clear steps
- **S**pecifics - Details matter
- **P**roduct - Expected output

Example:

"As a senior developer (Role), refactor this legacy method (Context) to use modern C# patterns and LINQ (Instructions), maintaining all unit tests (Specifics), and return the updated code with async support (Product)"

STAR Method:

- **S**ituation - Current state
- **T**ask - What needs doing
- **A**ction - Steps to take
- **R**esult - Expected outcome

Example:

"Situation: Legacy code with nested loops
Task: Optimize for performance
Action: Use LINQ and parallel processing
Result: 50% faster execution time"

Advanced Prompt Techniques

Level Up Your Prompting

1. Chain of Thought:

```
"First, analyze this code for issues.
Then, suggest improvements.
Finally, implement the top 3 improvements."
```

2. Few-Shot Examples:

```
"Convert to this pattern:
Input: oldMethod() { sync code }
Output: async newMethod() { await code }
Now convert: myFunction() { ... }"
```

3. Role-Based:

```
"As a security expert, review this code for vulnerabilities"
"As a performance engineer, optimize this query"
```

Using AI to Generate Prompts

"One Prompt To Rule Them All"

> *I want you to become my Prompt engineer. Your goal is to help me craft the best possible prompt for my needs. The prompt will be used by you <OpenAI, copilot, etc>.*
>
> *You will follow the following process:*
>
> 1. *Your first response will be to ask me what the prompt should be about. I will provide my answer, but we will need to improve it through continual iterations by going through the next steps.*
> 2. *Based on my input, you will generate 2 sections:*
>   i. *Revised prompt (provide your rewritten prompt. It should be clear, concise, and easily understood by you)*
>   ii. *Questions (ask any relevant questions pertaining to what additional information is needed from me to improve the prompt)*
> 3. *We will continue this iterative process with me providing additional information to you and you updating the prompt in the Revised prompt section until I say we are done.*

Prompt Engineering Exercise

Extending Your Refactored Code

Building on Exercise 1: Now that you have a clean receipt system, let's practice prompt refinement

New Requirement: "Add a loyalty program to the receipt system"

Your Task:

1. Start with this vague prompt in Copilot Chat
2. Iteratively refine it using CRISP framework
3. Document improvements at each step
4. Implement the best solution

Time: 15 minutes

Example Progression

See the Difference

Vague Prompt:

```
Add loyalty points to receipts
```

CRISP-Enhanced Prompt:

```
// CONTEXT: Extending our refactored SupermarketReceipt system
// ROLE: Act as a senior developer following SOLID principles
// INTENT: Add loyalty program that calculates points by tier
// SPECIFIC: 1pt/€1, Tiers: Bronze/Silver/Gold (0/500/1000)
// Gold gets 2x points on produce, Silver gets 1.5x
// PATTERN: Use existing ISpecialDeal interface pattern
// Example: €50 receipt (€20 produce) = 70pts for Gold tier
```

Result: A first try that is much closer to what we actually want.

Compare Your Results

Evolution of Output Quality

Vague

- Generic point system
- No integration with receipts
- Missing business rules
- Hardcoded values

CRISP:

- Tier-based rewards
- Receipt integration
- Configurable rules
- Test coverage included

Key Insight: Context + Specificity = Better Code

Exercise Debrief

What Did You Learn?

Discussion Points:

1. How did output quality change?
2. Which additions made the biggest difference?
3. Time saved with better prompts?

Key Insights:

- Specificity reduces rework
- Context prevents assumptions
- Examples guide style
- Constraints ensure compliance

Prompt Library

Build Your Team's Assets

Create Templates For:

- Common refactoring tasks
- Standard components
- Test generation
- Documentation
- Bug fixes
- Code reviews

Example Team Template:

```
"Generate C# unit tests for [method]:
- Use xUnit framework
- Follow AAA pattern
- Include edge cases
- Mock dependencies with Moq
- Aim for 100% coverage"
```

Real-World Prompt Examples

From Our Codebase

API Endpoint:

```
"Create a REST endpoint:
- POST /api/orders
- Accept OrderDto
- Validate required fields
- Return 201 with location
- Handle conflicts with 409
- Use MediatR pattern"
- Write integration test using TestContainers.
```

Data Access:

```
"Implement repository method:
- Use EF Core with includes
- Sort by LastLogin desc
- Support pagination
- Return IQueryable
- Add index hints"
```

Chapter 5: Advanced IDE Workflows

Productivity Patterns & Best Practices

Choosing Your AI Model

Models by Task Type

🚀 General Coding:

- **GPT-4.1** - Fast, accurate default
- **GPT-4o** - Low latency + visuals
- **Claude Sonnet 3.7** - Structured output

🧠 Deep Reasoning:

- **o3** - Multi-step problems
- **Claude Opus 4** - Complex analysis
- **Claude Sonnet 4** - Balanced power
- **Gemini 2.5 Pro** - Large codebases

⚡ Quick Tasks:

- **o4-mini** - Fastest responses
- **Claude Sonnet 3.5** - Quick syntax help
- **Gemini 2.0 Flash** - Real-time + visuals

🏗️ Agent Mode Ready:

GPT-4.1, GPT-4o, all Claude Sonnets

👁️ Visual Understanding:

GPT models, Gemini Flash, Claude 4

Pro Tip: GPT-4.1 is the reliable default, use specialized models for specific needs

Prompt Templates

Reusable Patterns for Success

Create Your Library:

```
## Refactoring Template
"Refactor this C# code to:
- Follow SOLID principles
- Use async/await patterns
- Implement dependency injection
- Add proper exception handling"

## Debugging Template
"This C# code throws [exception type].
Expected: [behavior]
Actual: [error message]
Stack trace: [trace]
Help me identify and fix the issue."
```

Pro Tip: Save templates in team knowledge base

Terminal & CLI Prompting

Command Line Assistance

Copilot in Terminal:

- Complex command construction
- Script generation
- Error interpretation
- Pipeline creation

Example Prompts:

```
# "Find all C# files modified in last week"
find . -name "*.cs" -mtime -7

# "Build and run tests for solution"
dotnet build && dotnet test

# "Create NuGet package with version"
dotnet pack -c Release -p:PackageVersion=$(date +%Y.%m.%d)
```

Knowledge Base in IDE

Your Team's AI Memory

What Goes in Knowledge Base:

- Coding standards
- Architecture decisions
- Common patterns
- API documentation
- Troubleshooting guides

How Copilot Uses It:

- Contextual suggestions
- Consistent patterns
- Team-specific solutions
- Reduced hallucinations

Privacy & File Exclusions

Control What Copilot Sees

Exclusion Options:

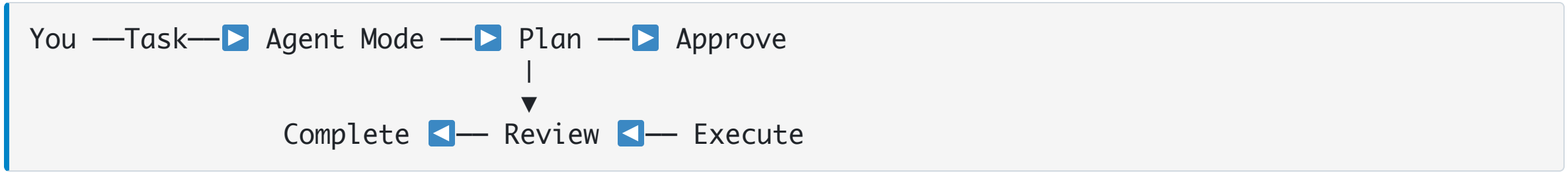
```
// .copilotignore
secrets/
*.env
**/credentials/**
private-docs/
```

Organization Policies:

- Data retention settings
- Telemetry controls
- Audit requirements
- Compliance rules

Agent Mode Preview

Autonomous Task Execution



When to Use:

- Multi-file changes
- Complex refactoring
- Feature implementation
- Systematic updates

When NOT to Use:

- Learning/exploration
- Critical business logic
- Without review time

Agent Mode Guardrails

Safety First Approach

Built-in Protections:

1. **Plan Phase**: Review before execution
2. **Approval Required**: Explicit consent
3. **Incremental Application**: Step-by-step
4. **Rollback Capability**: Undo changes

Your Responsibilities:

- Read the plan carefully
- Test after execution
- Maintain git history
- Document decisions

Demo - Library to API

Transform Receipt System to REST API

Live Demo: Converting our refactored receipt library into a web API

CRISP Prompt:

```
// CONTEXT: Refactored receipt library
// ROLE: API architect
// INTENT: Transform to Web API
// SPECIFIC: REST endpoints:
// - POST /api/receipts
// - GET /api/receipts/{id}
// - POST /api/deals
// PATTERN: RESTful, DTOs, 201s
```

Watch For:

- Agent's planning phase
- File structure choices
- Controller generation
- DTO mappings

Agent Mode in Action

Live Execution Flow

Agent's Plan:

1. Create ASP.NET project
2. Add receipt controller
3. Create API DTOs
4. Map domain ↔️ DTOs
5. Configure DI
6. Add Swagger docs

Your Review:

- Check architecture
- Verify endpoints
- Approve/modify plan

Results:

- Clean separation
- Proper HTTP semantics
- Testable design

Debugging with Copilot

From Error to Solution

Workflow:

1. Copy error to Chat
2. Add code context
3. Include stack trace
4. Get explanation
5. Apply fix
6. Verify solution

Effective Prompts:

- Full error message
- Relevant code snippet
- Expected behavior
- What you've tried
- Environment details

Productivity Patterns

Maximize Your Flow

Test-Driven Development:

```
1. Write test description
2. Let Copilot generate test
3. Implement to pass test
4. Refactor with confidence
```

Documentation-First:

```
1. Write function signature
2. Add detailed JSDoc
3. Let Copilot implement
4. Review and refine
```

Q&A Checkpoint

Your Questions So Far?

Common Topics:

- Context window limits
- Multi-language projects
- Integration with CI/CD
- Performance impact
- Learning resources

Remember:

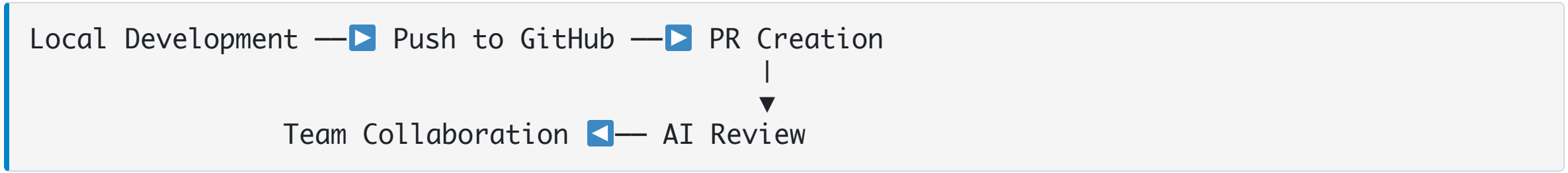
- No question too basic
- Share your use cases
- Learn from each other

From IDE to GitHub.com

Expanding Your Copilot Universe

Morning Focus: Individual productivity in IDE

Afternoon Shift: Team collaboration on platform



What's After Lunch?

Afternoon Preview

You'll Learn:

- PR automation features
- AI-powered code review
- Issue management with AI
- Security scanning integration
- Team knowledge sharing

You'll Do:

- Context-driven exercise
- Measure AI impact
- Build action plan

Lunch Break Prep

Before You Go...

Quick Tasks:

1. Note one morning insight
2. Think about team challenges
3. Prepare afternoon questions

Back at 13:00!

Welcome Back!

Round Table Discussion

Share Your Morning Insights:

- What was your key insight?
- Which feature surprised you?
- Team challenges identified?

Chapter 6: GitHub Platform Integration

Collaboration & Automation

Copilot in Pull Requests

AI-Powered PR Workflow

Automatic Features:

- PR summary generation
- Change categorization
- Impact analysis
- Test plan suggestions

How It Works:

```
Create PR
    ↓
Copilot Analyzes
    ↓
Generate Summary
    ↓
Suggest Tests
    ↓
Human Review
```

Assign Copilot as Reviewer

Your AI Team Member

Setup:

1. Add Copilot as reviewer
2. Receives code analysis
3. Posts review comments
4. Suggests improvements

Benefits:

- Consistent review standards
- Catches common issues
- Never tired or rushed
- Frees human reviewers

Remember: Supplement, don't replace human review

Issue Drafting & Task Creation

From Chat to Actionable Work

Workflow Example:

```
Discuss in Chat ──▶ Identify Tasks ──▶ Generate Issues ──▶ Assign & Track
```

Issue Templates with AI:

- Bug report generation
- Feature request drafting
- Technical debt documentation
- Enhancement proposals

Knowledge Bases on GitHub

Curate & Query Team Knowledge

What to Include:

- Architecture diagrams
- Coding standards
- Onboarding guides
- Common solutions
- Decision records

Query Examples:

- "How do we handle authentication?"
- "What's our testing strategy?"
- "Database schema conventions?"

Extensions in GitHub Chat

Expand Capabilities

Popular Extensions:

- Jira integration
- Slack notifications
- Documentation search
- API references
- Security tools

Custom Extensions:

- Team-specific tools
- Internal APIs
- Workflow automation

Demo - PR Automation

Creating Smart Pull Requests

Live Demo Flow:

1. Push feature branch
2. Create PR
3. Watch summary generation
4. Review test suggestions
5. Edit/enhance as needed

What to Notice:

- Accuracy of summary
- Suggested reviewers
- Test plan quality
- Label recommendations

Demo - AI Code Review

Copilot as Reviewer

Review Focus Areas:

- Security vulnerabilities
- Performance issues
- Code style violations
- Best practice suggestions
- Bug risk identification

Interaction Example:

- Copilot comments
- Developer responses
- Suggestion application
- Re-review cycle

Human Ownership

Do's and Don'ts

DO:

- ✅ Review all AI suggestions
- ✅ Add human context
- ✅ Verify security implications
- ✅ Test thoroughly
- ✅ Document decisions

DON'T:

- ❌ Auto-merge AI approvals
- ❌ Skip human review
- ❌ Ignore domain knowledge
- ❌ Trust blindly
- ❌ Blame AI for bugs

Q&A Checkpoint

Any Questions?

Chapter 7: AI Security Best Practices

Working Safely with AI Tools

The Golden Rules of AI Security

Protect Your Organization

NEVER Share:

- 🔐 API keys or tokens
- 🗝️ Passwords or credentials
- 💳 Customer PII data
- 🏛️ Proprietary algorithms
- 📊 Sensitive business data

ALWAYS:

- ✅ Use enterprise AI tools
- ✅ Review generated code
- ✅ Check data retention policies
- ✅ Follow company guidelines
- ✅ Report security concerns

Free vs Enterprise AI Tools

Know the Difference

❌ Free/Consumer AI:

- Data used for training
- No privacy guarantees
- Logs conversations
- Public model updates
- No audit trail

Risk: Code becomes public!

✅ GitHub Copilot Business / Enterprise:

- Zero data retention
- No training on your code
- SOC 2 compliant
- Audit logs available
- Enterprise controls

Safe: Your code stays yours!

Common Security Mistakes

What We've Seen Go Wrong

Real Incidents:

1. **API Key Exposure**: Developer asked ChatGPT to debug code with live AWS keys
2. **Customer Data Leak**: Pasted real customer database queries into free AI
3. **Algorithm Theft**: Proprietary trading logic ended up in public training data
4. **Compliance Violation**: GDPR data processed through non-compliant AI

Prevention: Think before you paste!

Safe AI Practices

Your Security Checklist

Before Using Any AI Tool:

```
☐ Is this tool approved by IT?
☐ Have I removed all secrets?
☐ Is the data anonymized?
☐ Do I understand retention?
☐ Am I following policy?
```

Pro Tip: Create test data sets for AI interactions

Handling Sensitive Code

When You Must Work with Secrets

Best Practices:

1. **Use placeholders**: Replace real values with `<API_KEY_HERE>`
2. **Environment variables**: Reference, don't embed
3. **Mock data**: Create realistic but fake examples
4. **Sanitize first**: Remove before sharing with AI

Example:

```
// DON'T: client.ApiKey = "sk-1234abcd...";
// DO: client.ApiKey = Environment.GetVariable("API_KEY");
```

GitHub Copilot Security Features

Built-in Protections

Automatic Filtering:

- Blocks common secret patterns
- Prevents generating real API keys
- Filters personally identifiable information
- Excludes files in .gitignore

Your Controls:

- Disable for specific files
- Exclude repositories
- Review telemetry settings
- Configure organization policies

Organization Policies

Protecting Your Team

GitHub Copilot Settings:

- Exclude specific file patterns
- Disable for sensitive repositories
- Require code review for AI suggestions
- Monitor usage through audit logs

Policy Example:

```
# .github/copilot-config.yml
disabled_for:
  - "**/*secret*"
  - "**/credentials/*"
```

Security Quiz

Test Your Knowledge

Which is safer to share with AI?

A) `password = "SuperSecret123!"`

B) `password = Environment.GetVariable("DB_PASS")`

C) `// TODO: Add password from vault`

Answer: B and C are safe, A exposes credentials

Remember: When in doubt, leave it out!

Exercise 2: Context Matters

A Hands-On Experiment

Exercise #2 - Round 1 Briefing

Build with Minimal Context

Your Task:

"Build a Team Lunch Voting App with Agent Mode"

Basic Features:

- Suggest restaurants for lunch
- Vote on today's options
- See the winning restaurant

Time: 30 minutes

Mode: Agent Mode only

Debrief - How Did It Go?

Common Round 1 Challenges

Typical Issues:

- Wrong tech stack (jQuery? Angular?)
- No clear voting rules
- Missing winner calculation
- Poor database design
- No time constraints

What Would Have Helped?

Round 2 - Context Pack

Rich Context Provided

Let's use proper context!

Clone the following repository: https://github.com/OnCore-NV/GitHub-Copilot-Track

Context Files Provided:

```
/context-pack
├── README.md              # Overview & instructions
├── requirements.md        # Voting rules, deadlines
├── tech-stack.md         # React, Java Spring Boot
├── api-spec.yaml         # OpenAPI specification
├── database-schema.sql # PostgreSQL schema
├── ui-mockup.md          # Design & components
└── code-patterns.md     # Examples & standards
```

Same Task: Build Team Lunch Voting App

Feeding Context to Agent

How to Provide Context

Methods:

1. **Attach Files:** Drag docs into chat
2. **@workspace:** Reference project patterns
3. **Paste Examples:** Show desired patterns
4. **Clear Constraints:** Specify requirements

```
"Build Team Lunch Voting App following the attached
requirements.md, matching our code-patterns and using the ui-mock.md as a style guide."
```

Round 2 Implementation

Build Again with Context

Fresh Start: Clear your project folder, except for the context-pack

Time: 25 minutes (5 min less!)

Focus: Quality over speed

Group Discussion

Your Experience:

- Was there an impact with the added context?
- How was the quality of the generated codebase?
- What surprised you?
- What more context would you add?

Group Discussion

Which Context Mattered Most?

Rank by Impact:

1. Architecture patterns
2. Code examples
3. Testing requirements
4. Style guide
5. Integration docs
6. UI specifications

Context Takeaways

Always Include...

Essential Context Elements:

- ✅ Clear requirements/constraints
- ✅ Existing patterns/examples
- ✅ Integration points
- ✅ Testing expectations
- ✅ Performance requirements

Team Action: Document your patterns!

Chapter 8: Making Impact

Adoption & Next Steps

Your Action Plan - Week 1

Start Strong

Daily Goals:

- [ ] Use inline completions for all coding
- [ ] Ask Copilot Chat 5 questions per day
- [ ] Generate one test suite
- [ ] Save 30 minutes on boilerplate

Your Action Plan - Week 2

Build Momentum

Skill Building:

- [ ] Master CRISP prompt framework
- [ ] Create 3 personal prompt templates
- [ ] Refactor legacy code with Chat
- [ ] Debug complex issue with AI help

Your Action Plan - Week 3

Level Up

Advanced Techniques:

- [ ] Use Agent Mode for multi-file feature
- [ ] Integrate AI into code reviews
- [ ] Build with context documentation
- [ ] Optimize model selection per task

Your Long-term Goals

3+ Months: AI-Powered Excellence

Technical Mastery:

- 🏗️ Architect with AI assistance
- 📚 Build company prompt repository
- 🔧 Custom tooling & workflows
- 📖 Continuous learning habit

Leadership & Impact:

- 🌟 Team AI champion
- 📝 Best practices documentation
- 🎓 Mentor others regularly
- 📈 Measure & share metrics

Anti-Patterns to Avoid

Common Pitfalls

Individual Anti-Patterns:

- 🚫 Accepting without review
- 🚫 Fighting the suggestions
- 🚫 Ignoring security warnings
- 🚫 Copy-paste programming

Team Anti-Patterns:

- 🚫 No shared standards
- 🚫 Forcing adoption
- 🚫 No knowledge sharing

Today's Key Takeaways

Your GitHub Copilot Toolkit

🧠 Core Concepts:

- LLMs predict, not memorize
- Context is everything
- CRISP framework for prompts
- Choose the right model

💡 Power Features:

- Inline completions save time
- Chat understands your code
- Agent Mode handles complexity
- Real-time collaboration

🛡️ Security First:

- Never share secrets with AI
- Use enterprise tools only
- Review all suggestions
- Enable protective policies

🚀 Remember:

- Start small, build habits
- Document what works
- Share with your team
- AI amplifies YOUR skills

Resources & Learning

Continue Your Journey

Official Resources:

- GitHub Copilot Docs
- VS Code Extension Guide
- Security Best Practices

Feedback Form

Help Us Improve

Your Input Matters!

[QR CODE PLACEHOLDER]

Contact

Stay Connected

Your Instructors:

- Name: [Instructor Name]
- Email: [instructor@email.com]

Remember: if you have any questions, feel free to reach out!

Thank You!

Happy Coding with GitHub Copilot!