AI in Software Development

Efficient AI Pair-Programming Strategies

Valentin Zuld - 23 Iulie 2025

Goals

- Master context-rich prompting for accurate, style-aligned suggestions
- Embed Copilot in coding, testing & PR review without slowing flow
- Apply the RED checklist (Read → Execute tests → Diff-review) to ship safe code

Agenda

0 : 00	Intro & Why AI Matters (20 min)
0 : 20	Copilot Toolbox – live tour (30 min)
0 : 50	Prompt Patterns + micro-exercises (40 min)
1 : 30	Break #1 (10 min)
1 : 40	Injecting Context Like a Pro (40 min)
2 : 20	Verification, Security & Team Practices (30 min)
2 : 50	Measuring Success & KPIs (10 min)
3 : 00	Break #2 (10 min)
3 : 10	Q&A + Hands-On Lab (50 min)
4 : 00	Wrap-Up

AI is Reshaping Dev Work

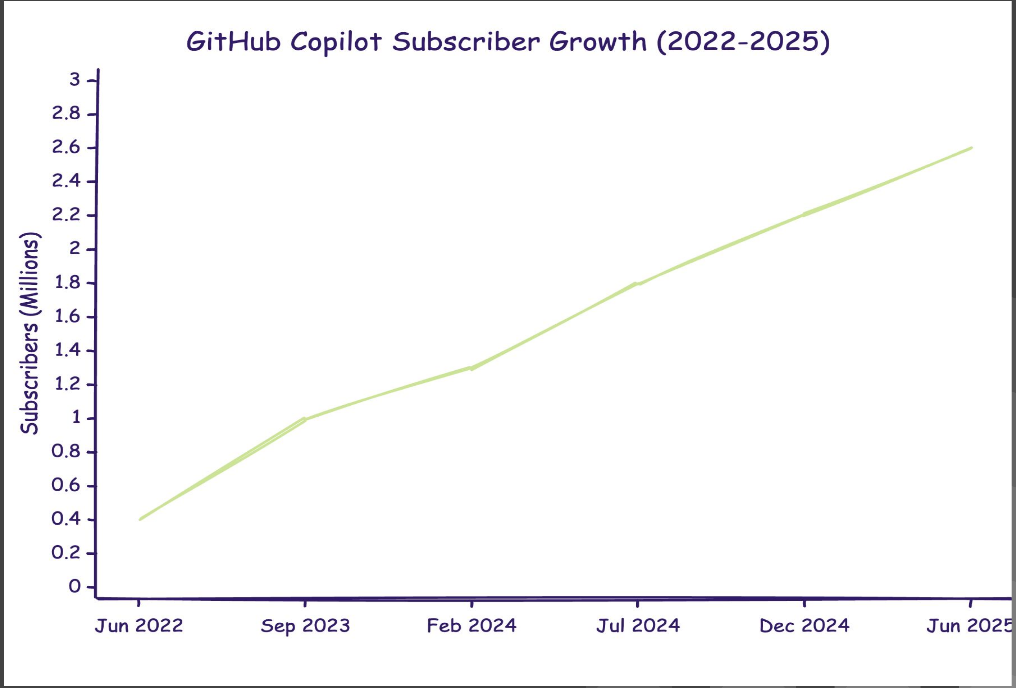
📊 Impact Metric (2024-25)

- 70 % of Copilot early adopters say they're more productive - source: Microsoft

- Controlled study: users finished coding/search/summarize tasks 29 % faster with Copilot - source: Microsoft

- 81 % of devs list "increase productivity" as the #1 benefit of AI tools (Stack Overflow '24)

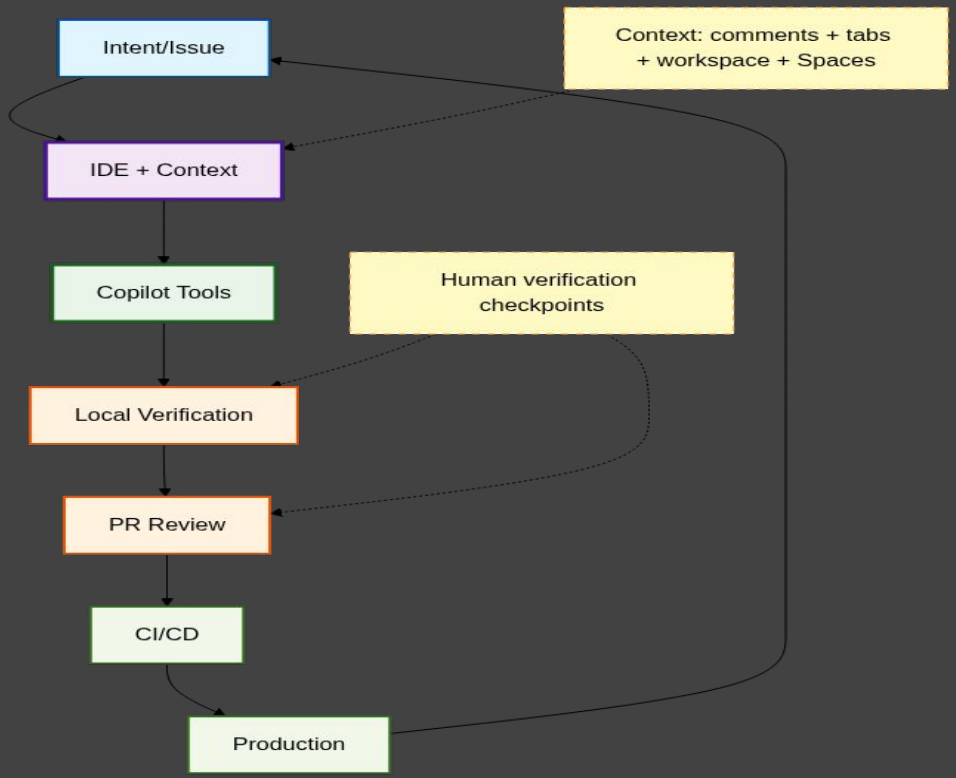
  https://survey.stackoverflow.co/2024/ai



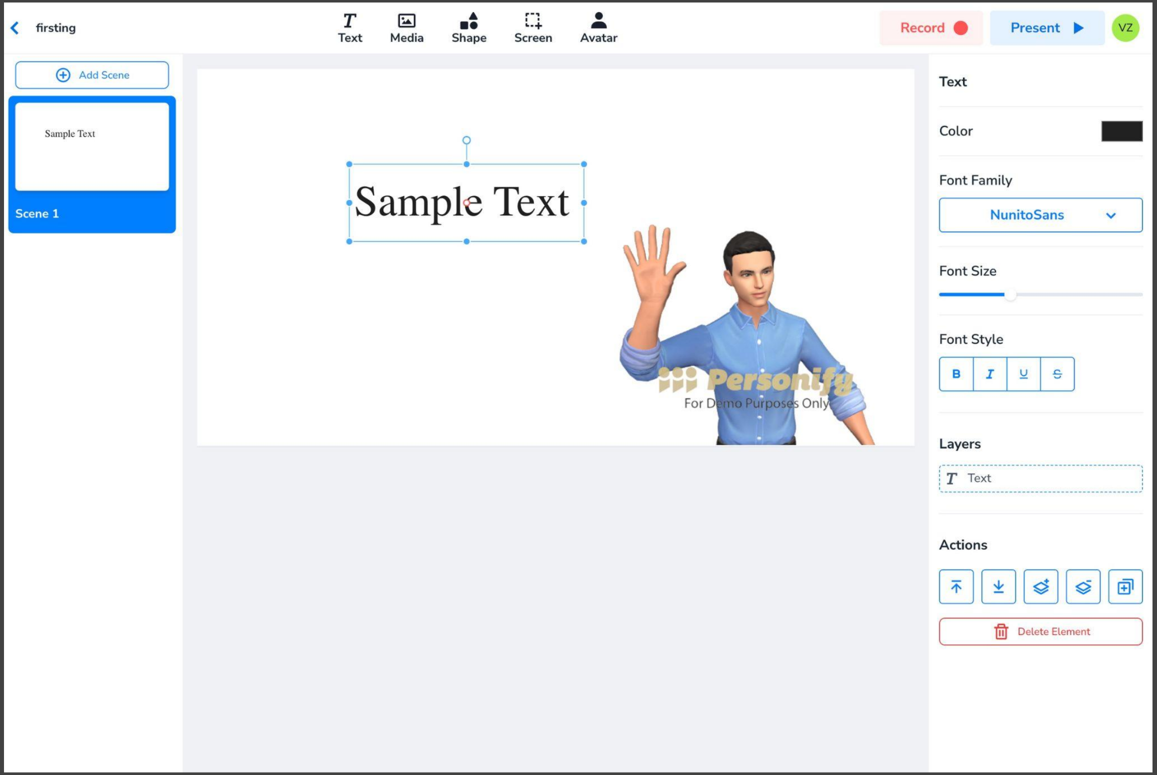
Impact by Experience Level: Gains vs. Trade-offs

	🚀 Gains	⚠️ Watch-outs
Juniors	• Instant explanations & examples • Faster prototypes & POCs • Exposure to idiomatic code	• Shallow grasp of fundamentals • Higher risk of accepting hallucinations • May skip debugger practice
Seniors	• 22 % faster coding on routine tasks (Jellyfish, 2025) • Offload boilerplate, focus on architecture • Rapid test-scaffold generation	• Can be 19 % slower if mis-prompting (METR RCT, Jul 2025) • Extra review load for AI code • Risk of “rubber-stamp” oversight

AI Feedback Loop



Real-World AI-Assisted Projects



Text

Color ■

Font Family: NunitoSans

Font Size

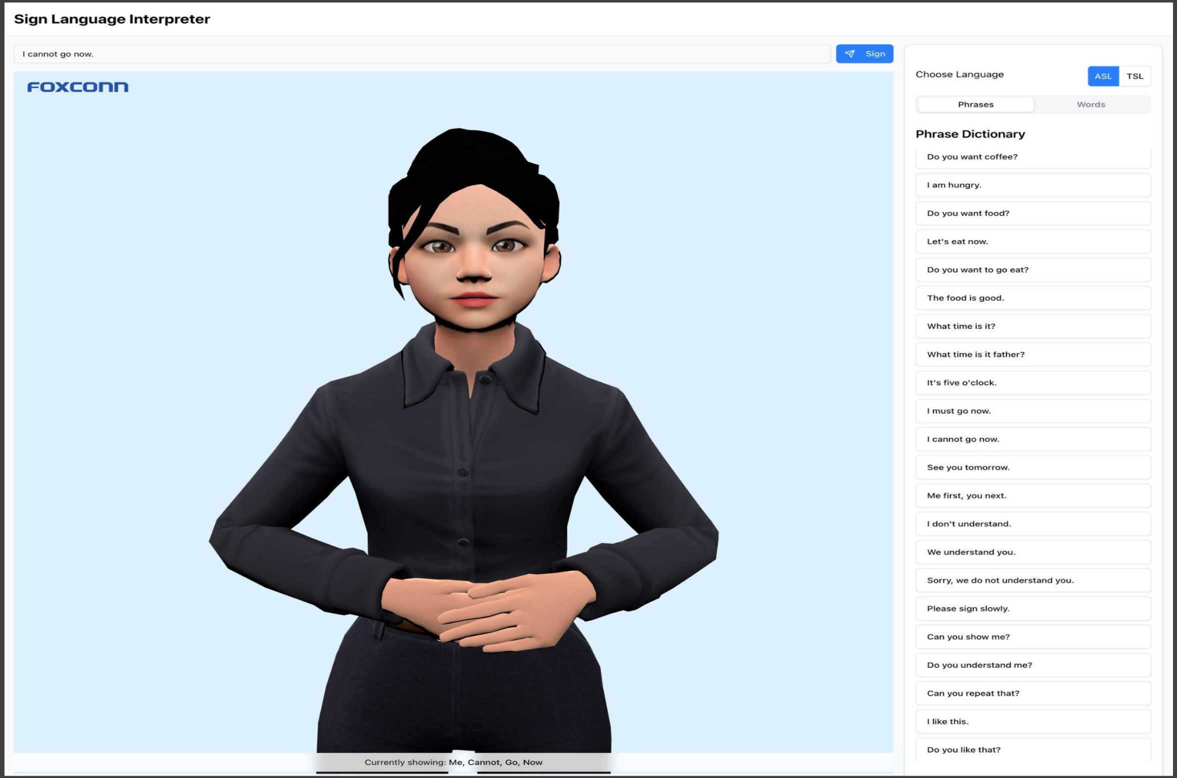
Font Style: B *I* U ~~S~~

Layers: Text

Actions

PRESENTER

Real-World AI-Assisted Projects



Sign Language Interpreter

I cannot go now.

Currently showing: Me, Cannot, Go, Now

Phrase Dictionary

- Do you want coffee?
- I am hungry.
- Do you want food?
- Let's eat now.
- Do you want to go eat?
- The food is good.
- What time is it?
- What time is it father?
- It's five o'clock.
- I must go now.
- I cannot go now.
- See you tomorrow.
- Me first, you next.
- I don't understand.
- We understand you.
- Sorry, we do not understand you.
- Please sign slowly.
- Can you show me?
- Do you understand me?
- Can you repeat that?
- I like this.
- Do you like that?

SPEECH TO
SIGN LANGUAGE

Real-World AI-Assisted Projects

Inventory

Machine Dashboard

Supervisor

Maintenance

Today’s Focus

Copilot’s super-power = the context you feed it

- 3 prompt recipes (FCE, Edge-Case Booster, Test-First)
- 4 context levers (comments, neighbours, workspace, docs)

Copilot Toolbox

1. Inline Suggestions – real-time completions as you type
2. Copilot Chat – ask “/explain”, “/tests”, “/fix” right inside the IDE
3. Copilot Agent Mode (preview) – multi-file refactors & test generation
4. PR / Code Review – request Copilot as a reviewer on GitHub.com
5. Copilot Spaces – curate docs & specs so Copilot answers with domain knowledge (public preview)

Inline Suggestions

Essential Shortcuts

- **Accept**: Tab (full suggestion) or Ctrl+→ (word by word)
- **Navigate**: Alt+] (next) / Alt+[(previous suggestion)
- **Reject**: Esc or keep typing to override
- **Partial accept**: Highlight good part → Tab → discard rest

```
// ✅ Good context
// Calculate shipping cost based on weight and distance
function calculateShipping(weight, distance) {
    // Copilot gets clear intent

// ❌ Vague context
function calc(w, d) {
    // Copilot has to guess
```

Copilot Chat

Essential Slash Commands

- /explain - Decode complex code, algorithms, or regex patterns
- /tests - Generate unit tests with edge cases and mocks
- /fix - Debug failing tests or runtime errors
- /optimize - Improve performance, reduce complexity

Advanced Chat Techniques

Reference specific code
"Refactor the #calculateDiscount function to use #PricingStrategy pattern"

Multi-step conversations
1. "/explain this authentication flow"
2. "Now add rate limiting to step 3"
3. "Generate tests for the rate limiter"

Context-aware requests

"Convert this REST endpoint to GraphQL, keeping the same validation"

Copilot Chat

Smart Context References

- #filename - Reference specific files
- #functionName - Target exact functions/classes
- **Selection-aware** - Automatically uses highlighted code as context

Beyond Code Generation

- **Documentation**: "Write API docs for this controller"
- **Code review**: "What could go wrong with this implementation?"
- **Learning**: "Explain the trade-offs between these two approaches"

Agent Mode

What Agent Mode Does

- **Multi-file operations**: Refactor across multiple files simultaneously
- **Context-aware changes**: Understands file relationships and dependencies
- **Intelligent planning**: Breaks down complex tasks into steps
- **Test generation**: Creates comprehensive test suites automatically

When to Use Agent Mode

- Large refactoring across multiple components
- Adding new features that touch many files
- Generating test coverage for existing code
- API redesigns with multiple endpoints

Agent Mode

Prompt: "Add user role-based permissions to the entire auth system"

Agent Mode:
1. Analyzes auth-related files
2. Updates User model + migration
3. Modifies AuthController methods
4. Updates middleware functions
5. Generates comprehensive tests
6. Updates API documentation

Best Practices

- Start with **clear, specific goals**
- Review each file change before accepting
- Test thoroughly - agent mode can introduce subtle bugs
- **Current status**: Preview feature (VS Code extension)

Copilot Spaces - Your Team's AI Knowledge Base

What Spaces Solve

- Copilot knows Stack Overflow but not your company's APIs
- Domain knowledge scattered across wikis, Confluence, Slack
- Repeated explanations of internal systems to AI

How Spaces Work

- Upload team docs, API schemas, coding standards
- Copilot references YOUR content when generating suggestions
- Persistent context across all conversations

Copilot Spaces - Your Team's AI Knowledge Base

Example Setup

```
/Backend-Team-Space/
├── api-guidelines.md
├── database-schema.sql
├── error-handling-patterns.js
├── deployment-checklist.md
└── business-rules/
    ├── pricing-logic.md
    └── user-permissions.md
```

Tips:

- Include real code examples from your codebase
- Add "gotchas" documentation - common mistakes to avoid
- Update when major changes happen
- Current status: Public preview (GitHub Copilot Enterprise)

What Copilot Actually Sees

Context Signals (ranked)

1. Current file (full buffer)
2. Open tabs / neighbours
3. Symbols in workspace index
4. Additional artefacts: specs, tests, docs, Spaces

Implication: Feed domain rules & style near your cursor or via a Space for best accuracy.

Reminder: “Garbage context = garbage suggestions.”

Demo Time

Why Prompt Engineering Matters

The Reality: AI Follows Instructions Literally

❌ Vague: "Create a login function"
Result: Basic username/password, no validation, hardcoded responses

✅ Specific: "Create secure login with email validation, bcrypt hashing, rate limiting (5 attempts/min), JWT response, proper error handling"

Result: Production-ready authentication with security best practices

Point: Time saved comes from clarity, not magic.

Why Prompt Engineering Matters

Real Impact on Development Speed

- **Bad prompts** → 3-4 iterations → 20+ minutes for simple function
- **Good prompts** → Working code first try → 2-3 minutes total
- **Compound effect** → 2+ hours saved per day across team

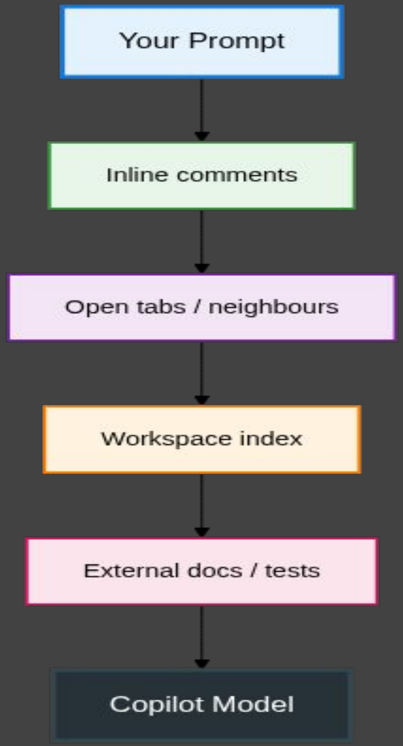
The "Garbage In = Garbage Out" Problem

- AI amplifies your input quality
- Unclear requirements → Unclear code
- Missing constraints → Security vulnerabilities
- No examples → Wrong assumptions about data format

Prompt Engineering = Better Code Reviews

- Specific prompts lead to self-documenting code
- Clear constraints reduce back-and-forth in PRs
- Examples in prompts become comments in code

Mental Model: Prompt → Context Funnel → Prediction



Higher up the funnel = bigger impact on suggestion quality.

Copilot never sees closed files; bring context to the model.

Recipe 1: FCE Pattern

Function: Gives AI clear intent and naming context

Constraints: Prevents hallucinations and edge case bugs

Examples: Shows exact data formats and business logic

❌ "Create a payment processor"

✅ "Function: Process credit card payments via Stripe

Constraints: USD only, $5-$10k limits, retry failed payments 2x

Examples: {amount: 2999, currency: 'USD'} → {success: true, id: 'ch_123'}"

- Include error examples: "Invalid email → {valid: false, error: 'Invalid format'}"
- Use real data formats from your system

Recipe 2: Edge-Case Booster

The Problem: AI Loves Happy Paths

- Default AI behavior: assumes perfect input, network, and conditions
- Real world: null values, network timeouts, malformed data, race conditions
- Result: Code that works in demos but breaks in production

Edge-Case Booster Template

```
// Handle user authentication with edge cases:
// - null/undefined inputs (return early with error)
// - expired/malformed tokens (throw AuthError)
// - rate limiting exceeded (429 status)
// - network failures (retry 3x with backoff)
// - database connection lost (graceful degradation)
  async function authenticateUser(token) {
```

Why it works: Explicitly calls out failure modes AI should consider

Test-First Specification

The Power of Tests as Specifications

- Tests become **executable requirements** that AI can follow
- Eliminates ambiguity about expected behavior
- Forces you to think through edge cases upfront
- Built-in verification when AI generates the implementation

```
// Should validate email format, reject typos, handle international domains
describe('Email validator', () => {
        it('accepts valid emails: user@domain.com, test+tag@example.org')
        it('rejects invalid: missing@.com, double@@domain.com, spaces in email')
        it('handles edge cases: unicode domains, 64+ char local parts')
        it('normalizes input: trims whitespace, converts to lowercase')
})

// Now prompt Copilot: "Implement the validateEmail function for these tests"
```

Common Prompt Failures

- Vague intent (“optimize code”)
- Missing constraints (perf, side-effects, API limits)
- No examples → model guesses format
- Oversized prompt (>200 lines) → truncation / loss of context
- Asking multiple distinct tasks in one shot
- Assuming AI knows your codebase

What “Context” Really Means

Everything the model receives before predicting next tokens

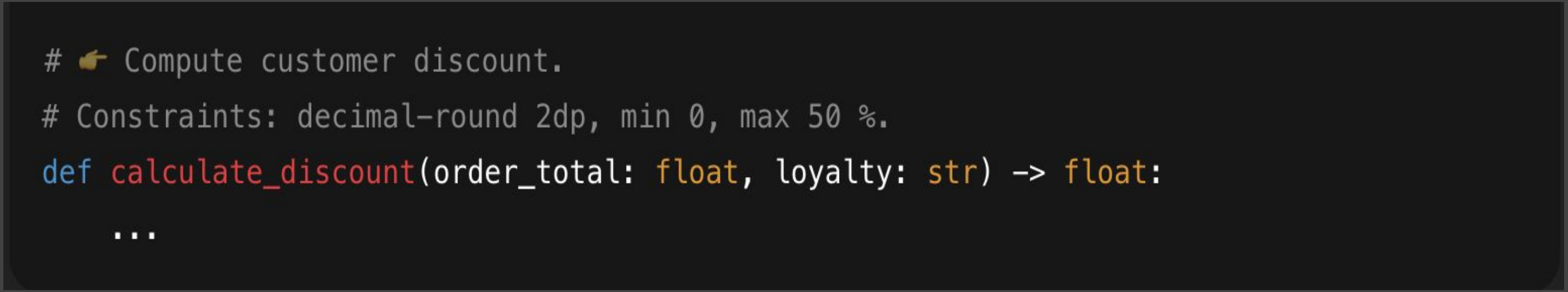
- Current file buffer
- Open/neighbor files
- Workspace index (symbols, paths)
- Extra artefacts: style guides, schemas, docs

Better context ⇒ fewer hallucinations, on-style code

The Four Context Levers

1. Inline comments / doc-strings - Feature intent, constraints
2. Open tabs & neighbor files - Interfaces, config, tests
3. Workspace index - Cross-file calls, types
4. External docs & style files - .editorconfig, STYLE_GUIDE.md, schema.md

Inline Comments & Docstrings



```
# 👉 Compute customer discount.
# Constraints: decimal-round 2dp, min 0, max 50 %.
def calculate_discount(order_total: float, loyalty: str) -> float:
    ...
```

- Place intent right above cursor for best weight
- Keep it concise < 20 tokens so model doesn’t trim deeper context

Open Tabs & Neighbor Files

- Copilot looks at all open buffers treat tabs as “prompt boosters”
- Open interface / test / schema files before prompting
- VS Code shortcut: Ctrl + K O to quickly open symbol’s file

Workspace Index & Symbols

- VS Code & GitHub create a searchable index of your repo
- Chat supports #-mentions: #calculate_discount, #controllers/OrderController
- Large monorepo? Enable local index for privacy + speed (settings → Copilot)

External Docs & Style Files

- .editorconfig – Copilot respects indent, max-line, quote style
- STYLE_GUIDE.md – add naming conventions → fewer review nits
- Domain artefacts – schema.md, api.yml, Confluence export

Multi-File Context Strategies

- Open key files - interfaces, main classes, tests
- Use Chat with #-mentions - Reference specific functions
- Work in small chunks - Refactor one class/module at a time
- Keep related files open - Maintain context between changes

Pro Tips

- Use "Split Editor" to keep context visible
- Copilot Chat: "Refactor #UserService to use #DatabaseInterface"
- Create temporary "REFACTOR_NOTES.md" with context

Documentation-Driven Development with AI

Write Docs First, Code Second

```
# User Authentication Service
## Requirements
- Support OAuth2 + API keys
- Rate limiting: 100 req/min per user
- Audit log all auth events
## API Design
POST /auth/login -> {token, expires_at}
## Error Handling
- 429 for rate limits
- 401 for invalid credentials
```

Then Prompt Copilot

```
// Implement the UserAuth service per DOCS.md above
// Use JWT tokens, Redis for rate limiting
  class UserAuthService {
```

Quick-Win Stats

Action	Accuracy Gain*
Open neighbor test file	+22 % correct first-try suggestions
Add .editorconfig	–65 % manual formatting fixes reported by VS Code lint logs
Provide schema.md	3× fewer SQL type errors in staging pipeline (internal sample)

*Internal & GitHub docs measurements 2023-25.

Don’t Trust Blindly

Reality check

- LLMs can invent non-existent packages → “slop” risk (20 % of code samples)
- Copilot will happily complete syntactically good but logically wrong code
- Every suggestion is a hypothesis until you test it.

Verification Checklist

R.E.D. - Your AI Code Safety Net

- **Read**: Scan the diff → does it match intent & style?
- **Execute**: Run the full test suite, not just new tests
- **Diff-review**: Compare against your mental model

Read - Code Review Questions

✅ Does this solve the actual problem I described?

✅ Are variable names consistent with our codebase?

✅ Any magic numbers or hardcoded values that should be constants?

✅ Does error handling match our team patterns?

✅ Are there any TODO comments or incomplete sections?

When Copilot Suggestions Fail

Common Failure Patterns

- **Wrong API usage** - Check official docs vs. Copilot suggestion
- **Logic errors** - AI follows pattern but misses business rules
- **Performance issues** - AI optimizes for readability, not speed
- **Integration bugs** - Doesn't understand your specific environment

Debug Strategy

1. Isolate the AI-generated portion
2. Add logging/breakpoints to see actual vs. expected behavior
3. Compare with working examples in your codebase

Iterating on Failed Prompts

Prompt Refinement Process

❌ First try: "Create user validation"
❌ Second try: "Validate user input with error handling"
✅ Third try: "Validate user registration form:
  - Email: RFC 5322 format
  - Password: 8+ chars, 1 symbol, 1 number
    - Return: {valid: boolean, errors: string[]}"

Iteration Techniques

- Add **concrete examples** of input/output
- Specify **error conditions** explicitly
- Reference **existing code patterns** in your project
- Use **FCE pattern** consistently

Security First:
Secrets & Vulnerabilities

- Copilot can surface real hard-coded secrets from public repos (GitGuardian)
- 23 % of Copilot snippets in a CWE-25 audit were insecure by default (Source)

Mitigations

- Secret scanning (GitGuardian, native GitHub).
- SAST / CodeQL in CI.
- Prompt Copilot:
  "Add input validation & safe defaults. Flag any potential CWE-"

Building Safety Nets for AI Code

CI/CD Pipeline Safeguards

- **Pre-commit hooks**: ESLint, Prettier, secret detection before code leaves local
- **Pull request gates**: Automated security scans block merge until clean
- **Dependency scanning**: Flag vulnerable packages AI might suggest
- **Code coverage requirements**: Ensure AI-generated code includes tests

Team Process Integration

- **Mandatory security review** for AI code touching auth/payments
- **Automated alerts** when AI suggests deprecated APIs
- **Quality gates**: Block deployment if coverage drops below threshold

Code Review Best Practices with AI

New Review Questions

- ✅ "Does this match the original prompt intent?"
- ✅ "Are edge cases properly handled?"
- ✅ "Any non-existent packages or APIs?"
- ✅ "Security: input validation, safe defaults?"

Review Process Updates

- Tag AI-generated code in PR descriptions
- Require tests for all AI suggestions
- Senior dev approval for critical path changes

Team Standards for AI-Generated Code

Establish Team Guidelines

```
# Team AI Code Standards
1. Always run tests before committing AI code
2. Use FCE pattern for complex functions
3. Required: peer review for AI database/security code
4. Shared prompt library in team docs
```

Quality Gates

- Same standards as human code (linting, coverage, performance)
- Extra scrutiny for authentication, data handling, API integrations

Personal Productivity KPIs

Track Your Own AI Impact

- **Feature velocity**: Stories completed per sprint (before/after Copilot)
- **Focus time**: Hours spent on creative vs. repetitive coding
- **Learning curve**: Time to implement unfamiliar APIs/frameworks
- **Code review feedback**: Reduction in style/syntax comments

Weekly Self-Assessment Questions

- Did I spend less time on boilerplate this week?
- Am I tackling more complex problems than before?
- How much time did I save on test writing?
- Did AI help me learn new patterns/libraries?

Team Impact Metrics

Collective Benefits to Track

- **Sprint burndown improvement**: More consistent velocity
- **Knowledge sharing**: Junior devs ramping up faster
- **Code consistency**: Fewer style debates in PRs
- **Technical debt**: More time for refactoring/cleanup

Team Health Indicators

- Reduced overtime during crunch periods
- Faster onboarding for new team members
- More time for architecture discussions vs. syntax fixes
- Increased participation in code reviews (less tedious work)

Career Development Wins

How AI Makes You a Better Developer

- **Learn faster**: Exposure to new patterns and best practices
- **Focus on design**: Less time debugging syntax, more on architecture
- **Broader skill set**: AI helps you work outside comfort zone
- **Mentoring ability**: Teach AI techniques to colleagues

Professional Growth Metrics

- New technologies adopted this quarter
- Complex problems solved vs. routine tasks
- Leadership opportunities (teaching AI practices)
- Innovation time (freed up from mundane coding)