



AI Assisted Software Development

From Code to Copilot

John Michael Miller



Principal Software Engineer at Code Staffing



Played roles of developer, architect, devops engineer, platform engineer, test architect, release manager



AI/ML Enthusiast and advocate for
Effectively using AI to write code

LinkedIn: www.linkedin.com/in/johnmichaelmiller

Email: john.miller@codestaffing.com

Blog: <https://codemag.com/blog/AIPractitioner>

[AI Practitioner Resources](#)

AI Assisted Software Development



Welcome



Introductions

Who you are

Who do you work for

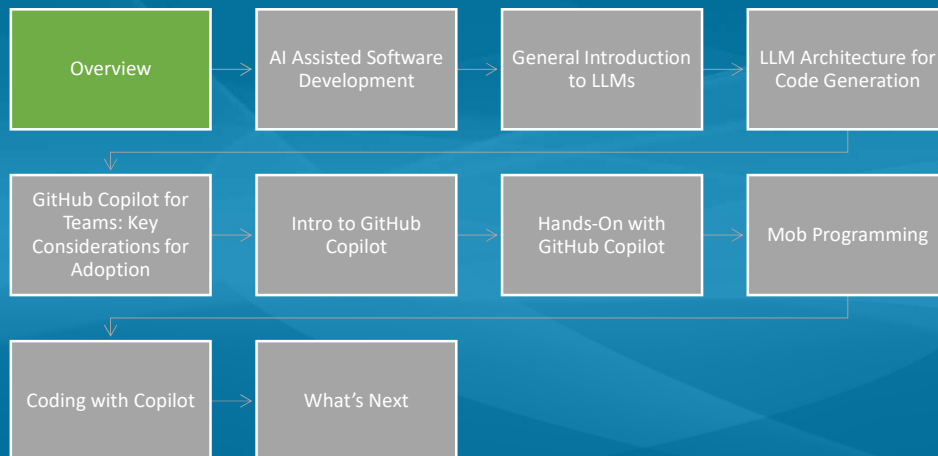
What you do

What you've done with AI tools

What you want to learn

Day One: Outline the day's goals and emphasize participation and hands-on exercises.

Agenda



Agenda: Quick walkthrough of topics; highlight hands-on demos and exercises.

Overview



Course Objective: Understand AI code generation using Copilot

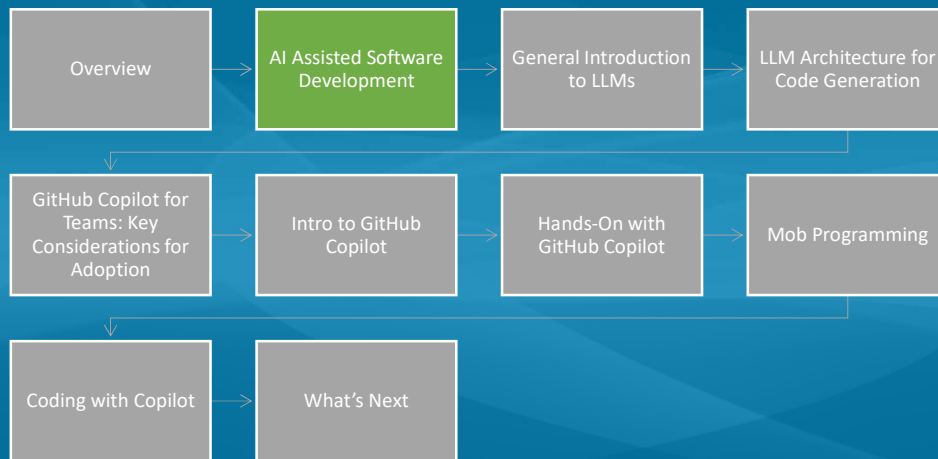
Course Perspective: AI assistance and assisting the AI

- Less emphasis on syntax
- More emphasis on architecture, design, code quality, refactoring
- Moving faster with less risk

::: notes State course objective: practical skills for using Copilot responsibly on legacy codebases. :::

Agenda

CODE
TRAINING



Agenda: Quick walkthrough of topics; highlight hands-on demos and exercises.

The AI Revolution?



"Programming hasn't changed, but how we go about it has changed, again."

- AI-assisted development is **evolutionary**, not revolutionary
- Programming has always been about **expressing human intent** to machines
- What changes is the **sophistication of our tools** for expressing intent
- The essence remains: bridging the gap between what we want and what machines can do

What is significant

Natural Language

Vast amount of knowledge available

Why AI Assisted Software Development



- If used effectively, it will give you superpowers
 - The courage to
 - Take on codebases that few would touch
 - Use technologies you should know but don't
 - Write more high-quality code than you have ever written before
 - Take on the nice to haves

Career Transformation: - Those who adapt: become 10x more productive, tackle bigger challenges, expand skill sets - Those who resist: may find themselves struggling with modern development expectations - New roles emerging: AI prompt engineers, AI code reviewers, AI-assisted architects

Superpowers Explained: - **Legacy codebases:** AI can quickly understand and explain complex, undocumented systems - **New technologies:** Learn frameworks/languages faster with AI as a coding partner - **Code quality:** AI suggests improvements, catches bugs, generates comprehensive tests - **Nice to haves:** Features that were “too time-consuming” become feasible

Examples to share: - Developer who used AI to modernize a 15-year-old PHP codebase in weeks instead of months - Team that adopted a new framework (React to Vue) with AI assistance in days - 80% reduction in boilerplate code writing time - Comprehensive test suites generated automatically

Key message: AI doesn't replace developers—it amplifies their capabilities

AI-First & Prompt-First



AI-First Development

A software engineering philosophy where AI is embedded across the entire SDLC—requirements, design, implementation, testing, documentation, compliance, and maintenance.

Prompt-First Development

A workflow pattern where prompts, instruction files, and chat modes are treated as first-class, version-controlled artifacts.

AI-First is the broad philosophy. Prompt-First is the tactical layer that enables predictable AI behavior. You can do Prompt-First without being AI-First, but not the reverse.

What Each Optimizes For



Focus Area	AI-First	Prompt-First
Scope	Entire SDLC	Interaction layer
Goal	Lifecycle integration	Deterministic AI behavior
Optimization	Velocity, governance	Prompt quality, reproducibility
Risk Controls	Human-in-loop, provenance	Versioned prompts, context control

This table is the heart of the comparison. AI-First is about organizational and architectural change. Prompt-First is about artifact discipline and predictable outputs.

How They Treat Artifacts



AI-First

- Requirements written with AI collaboration in mind
- AI-generated scaffolds, tests, docs
- Provenance enforced across all AI outputs
- Architecture assumes AI participation

Prompt-First

- Prompts and instruction files are version-controlled
- Prompts define behavioral contracts
- Reusable prompt modules
- Chat modes define safe, predictable interactions

AI-First changes *what* you build and *how* you build it. Prompt-First changes *how you communicate intent* to the AI.

Relationship Between the Two

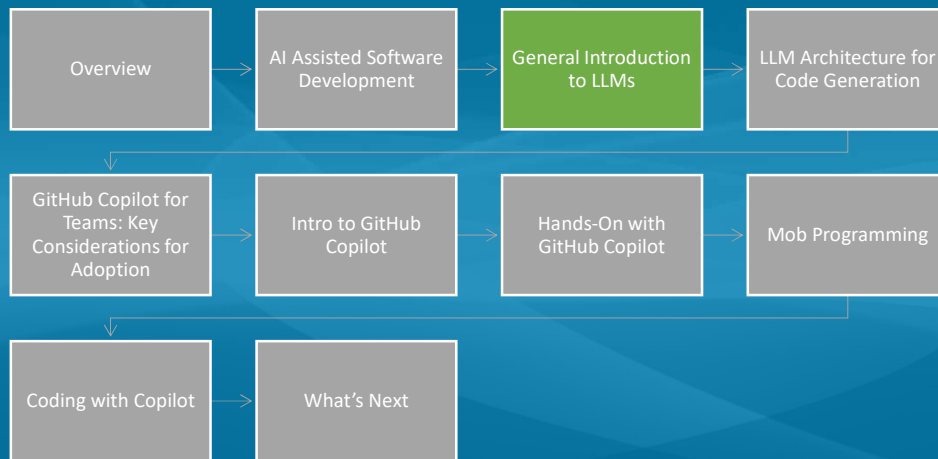


Prompt-First is a subset of AI-First.

- Prompt-First = mechanics
- AI-First = philosophy + architecture + lifecycle integration

This is the conceptual hierarchy. Prompt-First is necessary but not sufficient for AI-First maturity.

Agenda



Agenda: Quick walkthrough of topics; highlight hands-on demos and exercises.

General Introduction to LLMs



- Large Language Models
- Conceptual overview of how LLMs work
- Capabilities and limitations of LLMs in coding
- Software engineering with AI as a tool vs. “Vibe Coding”

Introduce LLMs at a high level: what they are, how they’re trained, and why they matter for developers.

Conceptual Overview of LLMs



- LLMs are trained on massive datasets of code and natural language
- They use transformer architecture to understand patterns
- Token-based processing: code is broken into tokens
- Attention mechanisms help models focus on relevant parts of context
- Generate probabilistic next-token predictions based on training patterns
- No true “understanding” - pattern matching at massive scale

Emphasize that LLMs don’t actually “understand” code like humans do

They use transformer architecture to recognize patterns and context

Token-based processing: code is broken into tokens (words, symbols, operators)

Explain that attention mechanisms are what make modern LLMs so powerful

The scale is what makes the difference - billions of parameters

When we say LLMs “understand” relationships, we mean they can statistically correlate patterns across tokens

This pattern matching is extremely sophisticated but lacks true semantic comprehension

Example: An LLM can recognize that a variable declared early in a function should be referenced later, not because it understands variables conceptually, but because it has seen this pattern millions of times in training

The “understanding” is really about statistical associations between code patterns, not conceptual knowledge

This is why LLMs can generate syntactically correct code that is logically flawed - they match patterns without true comprehension of the underlying logic

Capabilities



- Code completion and generation from natural language
- Refactoring and code optimization suggestions
- Bug detection and fixing recommendations
- Documentation generation
- Test case creation
- Code explanation and commenting
- Multi-language support and translation between languages

Capabilities: Describe common uses (completion, refactoring, tests) and show a quick live example if time permits.

Limitations*

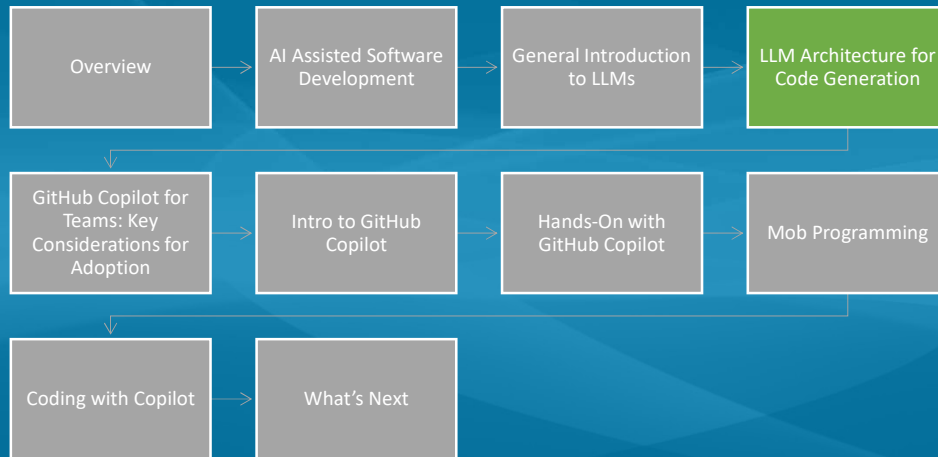


- No real-time knowledge updates (training cutoff dates)
- Can generate syntactically correct but logically flawed code
- Hallucination: confident-sounding but incorrect suggestions
- Context window limitations (finite memory)
- Does not “understand” business logic or domain-specific requirements
- Security vulnerabilities if not carefully reviewed
- Bias from training data

*Subject to change without notice

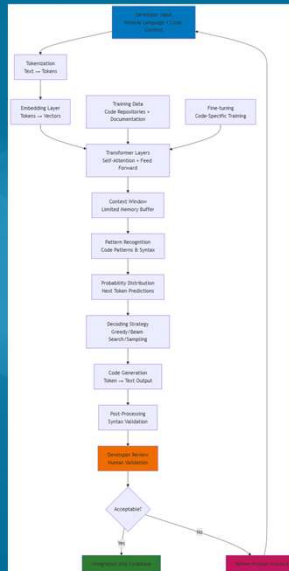
Limitations: Emphasize hallucination, context limits, and the need for human review and testing.

Agenda



Agenda: Quick walkthrough of topics; highlight hands-on demos and exercises.

LLM Architecture for Code Generation



Input Processing: - Developer provides natural language prompts + existing code context - Input is tokenized into discrete units the model can process - Tokens are converted to numerical vectors (embeddings)

Core Processing: - Transformer layers use self-attention to understand relationships - Context window maintains recent conversation/code history - Pattern recognition identifies relevant code structures and syntax

Output Generation: - Model generates probability distributions for next tokens - Decoding strategies determine how to select from probabilities - Output is converted back to readable code

Human in the Loop: - Critical validation step - AI is a tool, not a replacement - Feedback loop for prompt refinement and learning

Training Influence: - Large code repositories provide pattern knowledge - Fine-tuning specializes models for coding tasks - Quality of training data directly impacts output quality

Model Selection & Comparison



Name	Input Context Size	Output Context Size	Capabilities	Multiplier
Claude Haiku 4.5	128K	16K	Tools, Vision	0.33x
Claude Opus 4.1	68K	16K	Vision	10x
Claude Opus 4.5	128K	16K	Tools, Vision	3x
Claude Sonnet 4	128K	16K	Tools, Vision	1x
Claude Sonnet 4.5	128K	16K	Tools, Vision	1x
Gemini 2.5 P Gemini 3 Flash (Preview)	109K	64K	Tools, Vision	1x
Gemini 3 Flash (Preview)	109K	64K	Tools, Vision	0.33x
Gemini 3 Pro (Preview)	109K	64K	Tools, Vision	1x
GPT-4.1	111K	16K	Tools, Vision	0x
GPT-4o	64K	4K	Tools, Vision	0x
GPT-5	128K	128K	Tools, Vision	1x
GPT-5 mini	128K	64K	Tools, Vision	0x
GPT-5-Codex (Preview)	128K	128K	Tools, Vision	1x
GPT-5.1	128K	64K	Tools, Vision	1x
GPT-5.1-Codex	128K	128K	Tools, Vision	1x
GPT-5.1-Codex-Max	128K	128K	Tools, Vision	1x
GPT-5.1-Codex-Mini (Preview)	128K	128K	Tools, Vision	0.33x
GPT-5.2	128K	64K	Tools, Vision	1x
GPT-5.2-Codex	272K	128K	Tools, Vision	1x
Grok Code Fast 1	109K	64K	Tools	0x
Raptor mini (Preview)	200K	64K	Tools, Vision	0x

- Available models vary over time (Jan 18, 2026)
- Multipliers vary with Copilot Subscription level

Comparing LLMs for Code Generation



- Public Leaderboards
- Core Benchmarks
- Surveys & Deep-Dive Research
- Evaluation Frameworks

Public Leaderboards



Where to See Model-to-Model Comparisons

- LLM-Stats Coding Leaderboard
 - Aggregates 20+ coding benchmarks
 - Shows top performers across HumanEval, LiveCodeBench, etc.
- TechRadar's Coding LLM Guide
 - - Editorial comparison of strengths (debugging, test generation, etc.)
- Zencoder's 2026 Model Comparison
 - Breaks down accuracy, reasoning, and context window-

Core Benchmarks



What Models Are Actually Tested On

- HumanEval
 - Classic functional-correctness benchmark for Python
- LiveCodeBench
 - Contamination-free, holistic benchmark for modern LLMs
- MBPP (Mostly Basic Programming Problems)
 - Simple algorithmic tasks across languages
- SWE-Bench
 - Real-world GitHub issue resolution
- DevQualityEval
 - Evaluates test generation for Java and Go

Surveys & Deep-Dive Research



Understanding the Landscape

- Academic Surveys (arXiv)
 - Comprehensive overviews of techniques, benchmarks, and model families
- Benchmark Explainers (Vellum, Analytics Vidhya)
 - Strengths and weaknesses of each benchmark
 - How to interpret results
- Specialized Benchmarks
 - SwiftEval, domain-specific coding evaluations, etc

Evaluation Frameworks



- Tools for Running Your Own Tests
 - Symflower DevQualityEval
 - Open-source framework for code and test generation evaluation
 - CodeArena (HuggingFace)
 - Collective evaluation platform for coding tasks
 - Automatic Benchmark Generation Tools
 - Research into LLM-generated benchmarks and judge reliability

Selecting Models



- Select benchmarks aligned with your workflow
- Combine leaderboards + hands-on evaluation
- Build a repeatable internal benchmark suite
- Track contamination-free benchmarks for reliability

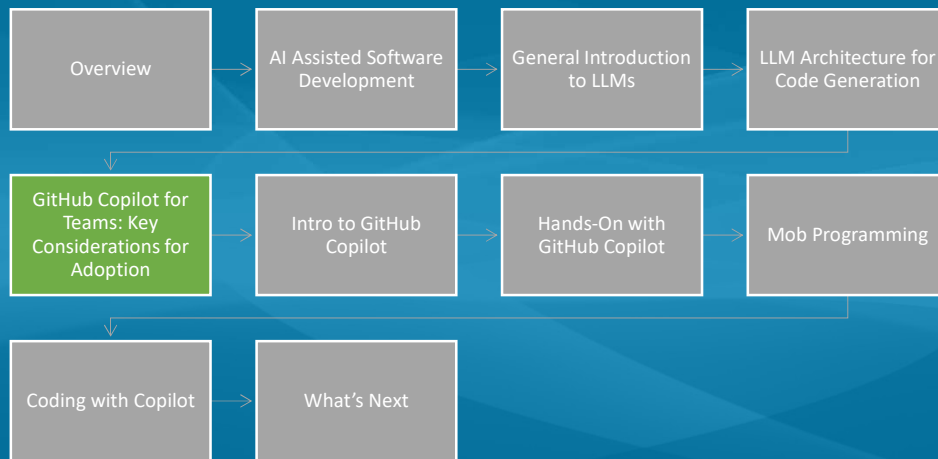
Contamination-Free Benchmarks

Why They Matter — and How They Work

Contamination-free benchmarks exist because modern LLMs are trained on **massive, scraped corpora** that often include the very benchmarks used to evaluate them. If a model has already seen the test set during training, its score is inflated — sometimes dramatically — and no longer reflects real reasoning or coding ability

A contamination-free benchmark is designed to **eliminate that inflation** and give you a score you can actually trust.

Agenda



Agenda: Quick walkthrough of topics; highlight hands-on demos and exercises.

GitHub Copilot for Teams

Key Considerations for Adoption



Empowering developers with AI while protecting your codebase

Outline governance, admin controls, and adoption factors (training, policy, developer onboarding).

Benefits for Organizations



- Accelerated Development
 - Faster prototyping, fewer boilerplate tasks
- Improved Documentation
 - Auto-generates comments and README content
- Enhanced Testing
 - Suggests unit tests and edge cases
- Team Productivity
 - Reduces cognitive load, supports onboarding

Highlight productivity, documentation, test generation, and onboarding benefits with brief examples.

Risks to Consider



- IP Leakage Concerns
 - Copilot may suggest code similar to public repositories
 - Risk of inadvertently using copyrighted or licensed code
 - Mitigation: Enable public code filters and review suggestions carefully
- Code Quality and Accuracy
 - AI-generated code may contain bugs, inefficiencies, or security flaws
 - Always validate and test before deployment
 - Treat Copilot as a drafting tool, not a source of truth
- Developer Overreliance
 - Risk of reduced understanding or critical thinking
 - Encourage code reviews and pair programming to maintain rigor

Cover IP leakage, code quality risks, and developer overreliance; suggest mitigations for each.

Governance and Compliance Risks



- Regulatory Compliance
 - Generated code may not meet industry-specific standards (e.g., HIPAA, PCI-DSS)
 - Organizations must enforce coding policies and audits
- Data Privacy and Security
 - Sensitive data should never be typed into prompts
 - Use Copilot in secure environments with clear usage guidelines
- Licensing Ambiguity
 - Copilot suggestions may resemble code under restrictive licenses
 - Legal teams should define acceptable use policies and monitor compliance

Discuss regulatory impacts, auditability, and how to enforce coding policies with automated checks.

IP and Data Protection



- Your code is **not used to retrain the model** (with Copilot for Business/Enterprise)
- Suggestions are generated locally — no code is shared unless feedback is submitted
- No leakage between users: your private code is **not exposed to others**
- Admins can **disable suggestions matching public code** for added safety

Clarify data flows, model retraining policy for enterprise plans, and recommended org controls to protect IP.

Licensing and Legal Considerations



- Copilot may suggest code similar to public repositories
- GitHub provides a **filter to block matching public code**
- Organizations should review Copilot's Terms of Service and Privacy Statement

Explain risks of suggested code resembling public repos and recommend legal review and filter settings.

Deployment Options



Plan	Key Features	IP Protection
Copilot Individual (Pro, Pro+)	Personal use, no admin controls	Limited
Copilot for Business	Admin controls, policy enforcement	Strong
Copilot for Enterprise	Org-wide policy, audit tools	Strongest

Summarize plan differences and pick considerations (control, audit, scale) for each offering.

Best Practices for Safe Use



- Enable public code filters
- Establish a review process
- Educate teams on responsible use and licensing awareness

Practical checklist: avoid secrets in prompts, enable public-code filters, and establish review processes.

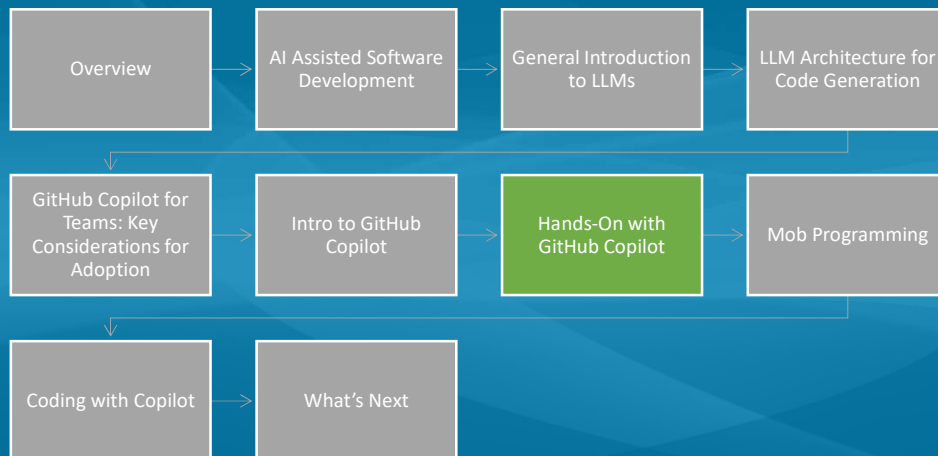
Resources



- Copilot Documentation:
 - <https://docs.github.com/en/copilot>
- Copilot for Business Overview
 - <https://github.com/features/copilot-for-business>
- Security and Privacy FAQ
 - <https://docs.github.com/en/copilot/security>

Point attendees to official docs and FAQs; recommend follow-up reading links on the slide.

Agenda



Agenda: Quick walkthrough of topics; highlight hands-on demos and exercises.

Hands-On with GitHub Copilot



- Installation and configuration
 - Installing the extension
 - Setting up authentication
 - Configuring settings
- Sharing configuration across an organization
 - Shared configuration templates (e.g., .copilot/settings.json) can be distributed across projects to standardize behavior.

<https://www.codemag.com/Blog/AI/AIASD-install-guide>

Walk through installation, auth, and a quick coding session; encourage participants to follow along.

Lab: Getting Started with GitHub Copilot



Duration: Follow along

Objectives

- Install and configure GitHub Copilot
- Verify authentication with GitHub account
- Explore the Copilot UI components

Activities

1. Install GitHub Copilot extension from VS Code marketplace
2. Sign in with your GitHub account (verify Copilot subscription)
3. Locate and explore:

- Chat window and chat history
 - New chat button
 - Quick chat feature (keyboard shortcut)
 - Settings menu
 - Model selection dropdown
4. Check your premium token usage bar
 5. Create a new chat and experiment with the interface

Success Criteria

- Copilot extension installed and authenticated
- Can open/close chat windows
- Understand difference between main chat and quick chat
- Know where to find chat history

Lab 1: Getting Started with GitHub Copilot

****Duration:**** 20-30 minutes

****Prerequisites:**** VS Code installed

Objectives

- Install and configure GitHub Copilot
- Verify authentication with GitHub account
- Explore the Copilot UI components

Activities

1. Install GitHub Copilot extension from VS Code marketplace

2. Sign in with your GitHub account (verify Copilot subscription)
3. Locate and explore:
 - Chat window and chat history
 - New chat button
 - Quick chat feature (keyboard shortcut)
 - Settings menu
 - Model selection dropdown
4. Check your premium token usage bar
5. Create a new chat and experiment with the interface

Success Criteria

- Copilot extension installed and authenticated
- Can open/close chat windows
- Understand difference between main chat and quick chat
- Know where to find chat history

Prompt Specificity



- Add error handling to my code
 - *Result: Generic response asking what type of errors, what language, what code?*
- Add error handling to my JavaScript function that calls an external API. I want to handle network timeouts, 404 errors, and JSON parsing failures. Return user-friendly error messages.
 - *Result: Better, but still generic without seeing actual code structure*
- @file:api-client.js Add comprehensive error handling to the fetchUserData function. Handle network timeouts (>5s), HTTP errors (404, 500, etc.), and JSON parsing failures. Return user-friendly error messages that match our existing error format in @file:error-types.js
 - *Result: Specific implementation that matches existing code patterns**
-

Lab: Understanding Context Management



Duration: Follow along

Objectives

- Learn to add context using @ symbols
- Understand context window limitations
- Practice writing effective prompts

Activities

1. Basic Context Addition:

- Use `@workspace` to search across your codebase
- Use `@file` to reference specific files
- Use `@terminal` to include terminal output in chat
- Use `@vscode` to ask VS Code-specific questions

2. Prompt Practice:

- Write a vague prompt, observe results
- Rewrite with specific context, compare results
- Add file references to improve accuracy

3. Context Window Experiment:

- Start a long conversation in one chat
- Notice when Copilot starts "forgetting" earlier context
- Practice starting new chats for new topics

Success Criteria

- Can use all @ context types
- Understand when to start fresh chat sessions
- Notice quality difference between vague and specific prompts

Lab: Chat Management & Workflow



Duration: Follow along

Objectives

- Organize chat sessions effectively
- Use chat history for reference
- Develop efficient workflow patterns

Activities

1. Chat Organization:

- Review your chat history
- Identify chats that should have been separate sessions
- Practice starting new chats at appropriate times

2. Context Preservation:

- Start a focused chat for one feature
- Add relevant context systematically
- Complete task without context overflow

3. Quick Chat Practice:

- Use main chat for primary task
- Use quick chat for side questions
- Return to main chat without losing context

4. Chat History Review:

- Find and reference previous solutions
- Learn from past prompts that worked well
- Identify patterns in effective conversations

Success Criteria

- Chat history is organized and meaningful
- Can find and reference previous solutions
- Efficient workflow developed for using multiple chat windows

Context Window Management

- Remember from the session:
 - Context is a ****limited resource****
 - Start new chat when changing focus areas
 - Keep conversations targeted and specific
 - When Copilot "forgets" earlier context, it's time for a new session

Lab: Exploring Copilot Modes



Duration: Follow along

Objectives

- Understand differences between Ask, Edit, and Agent modes
- Know when to use each mode
- Understand premium token implications

Activities

1. Ask Mode:

- Ask Copilot to explain a code snippet (no changes made)
- Request multiple implementation approaches
- Try different models and observe response quality
- **Note:** This doesn't consume premium tokens for advanced models

2. Edit Mode:

- Select code in a file
- Ask Copilot to refactor it

- Observe inline suggestions and changes
- Accept or reject proposed changes

3. Agent Mode:

- Ask Copilot to create a new file and add content
- Request changes across multiple files
- Have Copilot run terminal commands
- Check premium token usage after agent actions

Success Criteria

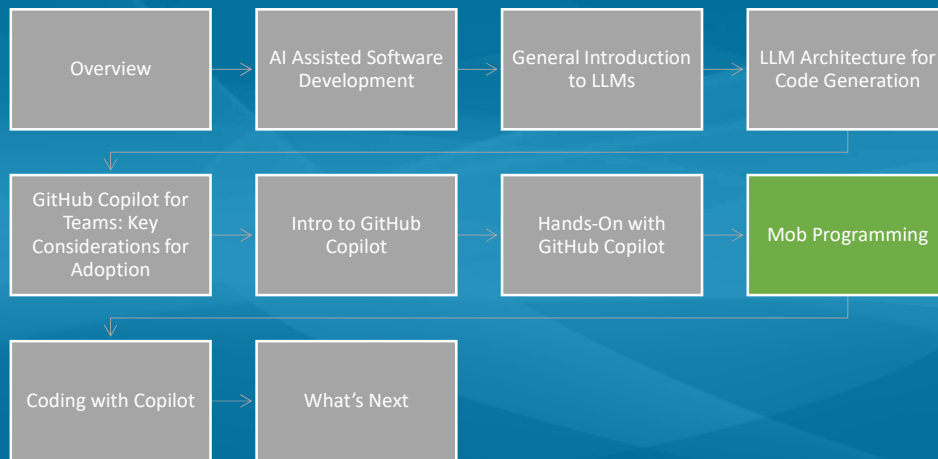
- Can distinguish when to use each mode
- Understand token consumption differences
- Successfully use agent mode for multi-file operations

IDE Support for AI Assistance



IDE / Editor	Built-In AI Features	Supported AI Assistants	Strengths	Limitations
VS Code	Deep AI integration through extensions; increasingly AI-first workflows	GitHub Copilot, Cline, ChatGPT-based extensions, Gemini integrations	Extremely flexible; huge ecosystem; top-tier AI support; widely adopted	Requires extension management; quality varies by plugin
Visual Studio (Windows)	Native GitHub Copilot integration; AI-powered IntelliCode	GitHub Copilot, IntelliCode	Strong enterprise + .NET support; excellent refactoring and debugging	Less flexible than VS Code for non-Microsoft stacks
JetBrains IDEs	JetBrains AI Assistant; code completion, refactoring, doc generation	JetBrains AI Assistant, GitHub Copilot	Deep static analysis + AI; strong multi-language support	JetBrains AI Assistant is subscription-based; Copilot integration not as seamless
Cursor IDE	AI-first editor; conversational coding; multi-file reasoning	Built-in AI models (GPT-based, Claude-based), Copilot alternatives	Designed for AI pair-programming; strong repo-wide reasoning	Not a traditional IDE; still maturing for large enterprise workflows
Replit	AI-powered Ghostwriter for code generation, debugging, and explanations	Ghostwriter	Great for beginners and rapid prototyping; browser-based	Less powerful for large, multi-module projects
Builder.io / Builder Code Editor	AI-enhanced coding environment with integrated assistants	Multiple AI integrations depending on setup	Strong web-dev focus; modern AI-native UX	Not a general-purpose IDE
Code-B Editors	Predictive code generation, debugging, and review	Multiple AI models depending on configuration	Strong AI-centric workflows; optimized for speed	Less mainstream; smaller ecosystem
Claude Code	Terminal-first AI coding assistant; autonomous repo-wide reasoning	Latest models from Anthropic and other via configuration	Exceptional multi-file context handling; ideal for agentic workflows and automated patching	Not a GUI IDE; best suited for terminal-centric development and large codebases

Agenda



Agenda: Quick walkthrough of topics; highlight hands-on demos and exercises.



Mob.sh: Install, Configure, and Use



A practical guide for teams adopting mob programming

Why mob programming

- AI Code generation is non-deterministic
 - A man with one watch always knows the time. A man with two is never sure.
- It's a useful technique if you haven't tried it
- Instead working alone, let's work together
- Mobbing started with one computer shared by the team
- Remote mobbing the team shares a WIP branch

Why Mob programming in a AIASD course?

AI code generation is nondeterministic. Attendees working on their own will see different output which will be difficult to explain. So rather than working on labs individually, work on them as a group using a shared codebase. This way they get the “hands-on” experience they expect without the nondeterministic nature of AI getting in the way.

Mob.sh makes this collaboration easier and more seamless.

What mob.sh Solves



- Smooth handoffs during mob/pair rotations
- Automatic WIP branches
- Zero-conf timers
- Clean commits and merges
- Works with any Git provider (GitHub, GitLab, Bitbucket)

Installing mob.sh



Requirements

- Git installed
- macOS, Linux, or WSL
- Optional: Homebrew

Install Options

- Homebrew (recommended)

```
brew install mob
```

- Curl installer

```
curl -sL  
https://install.mob.sh | sh
```

- Manual

- Download from <https://mob.sh>
- Add to PATH
- Verify with:

```
mob version
```

<https://www.codemag.com/Blog/AI/AIASD-install-guide>

Lab: Cloning the Course Repository



Duration: 10 minutes

Prerequisites: Git, GitHub account

1. Clone the AIASD-20260209 repository
2. Switch to the fundamentals branch

Objectives

- Fork and clone the class repositories

Success Criteria

- Cloned repository exists locally

Activities

Objective: Fork the course repos **Tasks**

Search GitHub for

AI-Assisted-Software-Development

Fork this repo

This will create a personal copy under your GitHub account

You can make changes without affecting the original repo

First-Time Configuration



Set your preferred editor

```
mob config editor "code"
```

Set your default WIP branch name

```
mob config wip-branch "mob-session"
```

Optional: Set timer duration

```
mob config timer 5
```

Starting a Mob Session



Begin a session

`mob start`

What happens:

- Creates/updates a WIP branch - Stashes local changes
- Opens your editor - Starts optional timer



Handoff to the Next Driver



When your rotation ends

`mob next`

This:

- Commits WIP
- Pushes to remote
- Prepares the next driver's environment

Finishing the Mob Session



When the feature is ready

mob done

This:

- Squashes WIP commits
- Merges into your main branch
- Cleans up the WIP branch



Role Rotation Tips



- Driver rotates every 5–7 minutes
- Navigator(s) guide the next steps
- Researchers support without interrupting flow
- Use a shared timer (mob.sh or external)

Lab: Mob Programming Setup



Duration: 15-20 minutes

Objectives

- Explore collaborative coding with Copilot
- Set up for team development

Activities

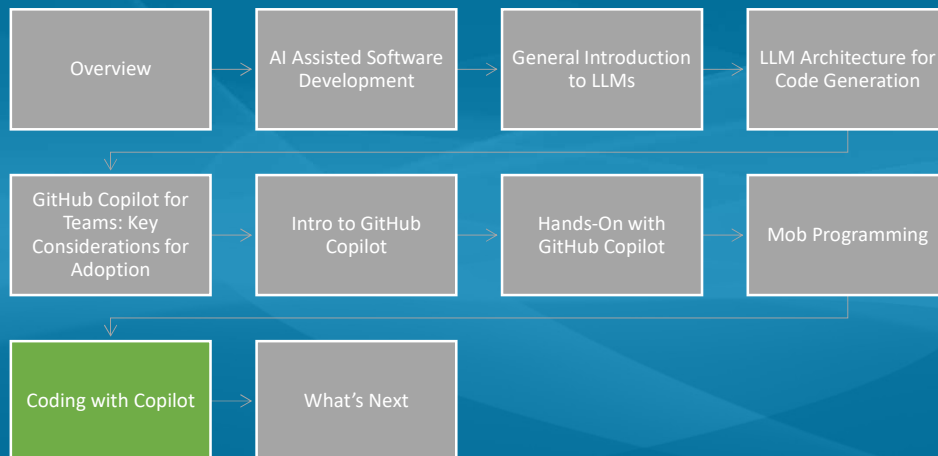
1. Configure for your environment
2. Practice driver/navigator roles with Copilot as assistant

Success Criteria

- Successfully participate in a mob session

Agenda

CODE
TRAINING



Agenda: Quick walkthrough of topics; highlight hands-on demos and exercises.

Collaborating on a Solution



- Project Setup
- Adding Features
 - Basic Arithmetic - Addition, subtraction, multiplication, division
 - Clear / Reset Function - Quickly resets the current input or entire calculation
 - Decimal Support - Allows entry and computation with decimal numbers
 - Sign Toggle (+/-) - Switches values between positive and negative
 - Percentage Function - Converts values to percentages for quick calculations
 - Memory Functions (M+, M-, MR, MC) - Store, recall, add to, or clear memory values
 - Error Handling - Displays errors such as division by zero
 - Simple, Intuitive Interface - Numeric keypad, operation buttons, and display screen
- Test Automation
 - Code Coverage
- Dependency Management
- Comparing Implementations
- Chat Management
- Intro to Evergreen Software Development

This slide outlines the collaborative development process we'll follow for building our calculator application — and mob programming will be central to how we work together.

Except for Project Setup and Basic Arithmetic functions, the mob controls the direction of the implementation. The other labs are optional and can be used when the mob directs the development of that feature or as suggestions should the mob struggle for direction.

We begin with **Project Setup**, where the team configures the environment, aligns on goals, and prepares the repo. This is done as a mob — one keyboard, one screen, and everyone contributing ideas in real time. It ensures shared understanding from the start.

At the end of the day, we introduce **Evergreen Software Development** — a mindset of continuous improvement.

Random ideas:
Implementing Observability

Lab: Calculator Project – Basic Arithmetic



Duration: 45-60 minutes

Objectives

- Use AI to generate starter code for arithmetic operations
- Understand how to validate AI-generated logic
- Integrate addition, subtraction, multiplication, and division functions

Activities

1. Project Initialization:

- Prompt AI to create a new project
- Review generated project structure
- Verify build configuration

2. Implement Basic Operations:

- Prompt AI to add methods for addition, subtraction, multiplication, and division

3. Review the Code:

- For correctness and edge cases

4. Build and Run:

- Use Copilot to help with build commands
- Troubleshoot any compilation errors with Copilot's help
- Run the application

• Success Criteria

- Working calculator with 4 basic operations
- Application compiles and runs successfully
- You've critically reviewed all generated code

Lab: Calculator Project – Clear / Reset



Duration: 15 minutes

Objectives

- Use AI to scaffold state-management logic
 - Implement CE (clear entry) and C (clear all) behaviors
 - Understand how AI can help reason about UI state transitions
2. Generate code for clearing the current input vs. full state
 3. Integrate the logic into your calculator's state object
 4. Test transitions by simulating user input sequences

• Success Criteria

- - CE clears only the active entry
- - C resets the entire calculator state

Activities

1. Prompt AI to outline the difference between CE and C

Lab: Calculator Project – Decimal Input



Duration: 12 minutes

Objectives

- Use AI to generate input-validation logic
- Prevent multiple decimal points
- Ensure decimals flow correctly through arithmetic operation

Activities

1. Ask AI to propose a strategy for handling decimal input
2. Generate code to block multiple decimals in a

single number

3. Integrate decimal support into the existing input parser
4. Test decimal operations using AI-generated test cases

Success Criteria

- Decimal input works without duplication errors
- Arithmetic with decimals produces correct results
- Learner can explain the validation logic

Lab: Calculator Project – Sign Toggle (+/-)



Duration: 8 minutes

Objectives

- Use AI to generate logic for toggling numeric sign
- Understand how sign toggling interacts with current input and stored values

Activities

1. Ask AI to generate a function that toggles sign on the active value

2. Integrate the function into the input workflow
3. Test sign toggling before and after entering digits

Success Criteria

- Sign toggle works consistently for integers and decimals
- Learner can explain how the toggle affects stored vs. active value

Lab: Calculator Project – Percentage



Duration: 15 minutes

Objectives

- Use AI to clarify how calculators interpret %
- Implement percentage logic for common patterns
- Validate behavior with AI-generated examples

Activities

1. Ask AI to explain how % should behave in a standard calculator
2. Generate code for:
 - $X \times Y\%$
 - $Y + X\%$

- $Y - X\%$

3. Test each pattern with AI-generated sample values

Success Criteria

- Percentage operations match standard calculator behavior
- Learner can articulate the interpretation rules for %

Lab: Calculator Project – Memory Functions (M+, M–, MR, MC)



Duration: 18 minutes

Objectives

- Use AI to design a memory subsystem
- Implement memory add, subtract, recall, and clear
- Validate memory behavior across multiple operations

Activities

1. Ask AI to propose a memory-state structure

2. Generate functions for M+, M–, MR, MC
3. Integrate memory operations into the calculator workflow
4. Test memory persistence across multiple calculations

Success Criteria

- Memory functions behave as expected
- Learner can explain how memory state is stored and updated

Lab: Calculator Project – Error Handling



Duration: 10 minutes

Objectives

- Use AI to identify common error conditions
- Implement error messages and recovery logic
- Ensure the calculator resets gracefully after errors

Activities

1. Ask AI to list typical calculator errors (e.g., divide by zero)

2. Generate code for error detection and display
3. Implement a reset path after an error
4. Test error scenarios using AI-generated test cases

• Success Criteria

- Errors are detected and displayed correctly
- Calculator recovers cleanly after reset
- Learner can describe the error-handling flow

Lab: Calculator Project – UI



Duration: 15 minutes

Objectives

- Use AI to scaffold UI event handlers
- Connect buttons to logic functions
- Validate end-to-end user workflow

Activities

1. Ask AI to generate event-binding code for numeric and operator buttons
2. Integrate logic functions from

previous labs

3. Test a full workflow:

- Enter decimal
- Toggle sign
- Apply percentage
- Store result in memory

Success Criteria

- UI correctly triggers all calculator functions
- Full workflow completes without errors
- Learner can explain how UI events map to logic functions

Lab: Calculator Project – Add Trigonometric Functions



Duration: 15 minutes

Objectives

- Integrate trigonometric functions into the calculator's operation set
- Use AI to generate math-library wrappers and input-parsing logic
- Ensure correct handling of degrees vs. radians

Activities

1. Ask AI to generate functions for sin, cos, and tan using your language's math library
2. Prompt AI to propose a strategy for handling degree/radian mode

3. Implement UI bindings or command triggers for each trig function
4. Use AI to generate a table of sample inputs and expected outputs

Success Criteria

- Trig functions compute correct values in the selected angle mode
- Degree/radian mode switching works consistently
- UI or command triggers correctly call the trig functions
- Learner can explain how AI-generated code was validated and refine

Lab: Calculator Project – Encapsulate Core Logic



Duration: 15 minutes

Objectives

- Separate UI concerns from computational logic
- Use AI to scaffold a standalone “core logic” module/class
- Ensure the UI communicates with the logic layer through a clean, well-defined API
- Validate that encapsulation improves testability and maintainability

Activities

1. - Ask AI to generate a dedicated component (e.g., CalculatorEngine, CalculatorCore) containing:

- Arithmetic operations
- State management
- Trig/percentage/memory logic (if implemented)

2. Review the AI-generated API surface and refine naming, inputs, and return types

3. Replace UI-embedded logic with calls into the new component

Success Criteria

- All calculator features run through the external logic component
- UI contains no computational logic — only event handling and display updates
- Learner can explain how encapsulation improves modularity, reuse, and AI-assisted development workflow

Lab: Calculator Project - Testing



Duration: 45-60 minutes

Objectives

- Generate unit tests with AI assistance
- Identify quality issues in generated tests
- Understand the importance of reviewing AI-generated tests

Activities

1. Generate Initial Tests:

- Prompt: "Create unit tests for the calculator operations"
- Review generated test structure
- **Critical Review:** Are tests calling your calculator code?

2. Fix Test Issues (Replicating Session Demo):

- If tests are too simple (like ``1 + 1 = 2`` without calling calculator):
- Identify the problem
- Ask Copilot to fix it: "Update tests to call Calculator class methods"
- Verify tests now test actual implementation

3. Run Tests:

- Execute test suite
- Review test output
- Debug any failing tests with Copilot's help

4. Add Edge Cases:

- Prompt: "Add tests for edge cases like division by zero"
- Verify exception handling tests are correct

Success Criteria

- Test suite with minimum 8 test cases
- All tests call actual calculator methods (not just language arithmetic)
- Tests include edge cases and error conditions
- All tests pass

Lab: Code Coverage



Duration: 30-40 minutes

Objectives

- Set up code coverage reporting
- Interpret coverage results
- Improve test coverage based on gaps

Activities

1. Enable Coverage Collection:

- Prompt: "Add code coverage reporting to my test project"
- Review package dependencies added
- Handle any NuGet/dependency issues with Copilot's help

2. Generate Coverage Report:

- Run tests with coverage enabled
- Review coverage percentage
- Identify uncovered code paths

3. Improve Coverage:

- Add tests for uncovered methods
- Re-run coverage to verify improvement
- Discuss: Is 100% coverage always necessary?

Success Criteria

- Code coverage reporting successfully configured
- Can generate and read coverage reports
- Achieved reasonable coverage (>80% line coverage)
- Understand what coverage metrics mean

Discussion Points

- Feature coverage vs. code coverage (as raised by Tom in the session)
- When is test coverage sufficient?
- Quality of tests vs. quantity

Lab: Dependency Management & Troubleshooting



Duration: 30-40 minutes

Objectives

- Use Copilot to resolve dependency issues
- Handle package restoration problems
- Practice iterative problem-solving with AI

Activities

1. Simulate or Identify a Dependency Issue:

- Introduce a version conflict (or use existing issue)
- Prompt: "I'm getting [specific error]. How do I fix it?"

2. Follow Copilot's Guidance:

- Review suggested solutions
- Evaluate multiple approaches if offered
- Choose best solution collaboratively

3. Iterative Resolution:

- If first solution doesn't work, provide error details

- Continue conversation until resolved

4. Common Issues to Practice:

- NuGet package source configuration
- MSTest adapter version conflicts
- .NET SDK targeting issues
- Package restoration failures

Success Criteria

- Successfully resolved at least one dependency issue
- Understand how to provide error context to Copilot
- Practiced iterative problem-solving approach

Real-World Scenario

- This lab replicates the exact dependency challenges encountered in the training session:
 - Updating test project to target .NET 8
 - Resolving NuGet.org package source mapping
 - MSTest adapter compatibility issues

Lab: Security Review



Duration: 30-40 minutes

Objectives

- Apply best practices learned in session
- Review code quality systematically
- Identify and fix issues

Activities

1. Security Review:

- Prompt: "Review this code for security vulnerabilities"
- Address any identified issues
- Add input validation where missing

Success Criteria

- Code has no obvious security issues
- Critically evaluated all AI suggestions

Lab: Code Quality Review



Duration: 30-40 minutes

Objectives

- Apply best practices learned in session
- Review code quality systematically
- Identify and fix issues

Activities

1. Code Quality Check:

- Prompt: "Suggest improvements for code quality and maintainability"

- Evaluate suggestions critically
- Implement valuable improvements

Success Criteria

- Comprehensive documentation added
- Critically evaluated all AI suggestions

Lab: Documentation



Duration: 30-40 minutes

Objectives

- Add documentation to a project
- Update existing documentation

Activities

1. Documentation:

- Ask Copilot to generate XML/doc comments
- Review for accuracy and completeness

- Add README with usage instruction
- Ask AI to update existing documentation

Success Criteria

- Comprehensive documentation added
- Critically evaluated all AI suggestions

Lab: Refactoring



Duration: 30-40 minutes

Objectives

- Apply best practices learned in session
- Review code quality systematically
- Identify and fix issues

Activities

1. Refactoring Exercise:

- Ask Copilot for alternative implementation approaches
- Compare different solutions

- Discuss trade-offs (as mentioned in session)

Success Criteria

- Code has no obvious security issues
- Comprehensive documentation added
- At least one refactoring improvement implemented
- Critically evaluated all AI suggestions

Lab: Model Comparison Exercise



Duration: 20-30 minutes

Objectives

- Compare outputs from different AI models
- Understand when to use premium vs standard models
- Monitor token usage

Activities

1. Same Prompt, Different Models:

- Choose a coding task (e.g., "implement bubble sort")
- Try with GPT-4 (standard - unlimited)
- Try with Claude Sonnet (premium - 1x token)
- Compare results for quality, style, completeness

2. Token Usage Analysis:

- Check premium token bar before and after
- Calculate tokens consumed
- Discuss: Was premium model worth the cost?

3. Best Use Cases:

- Identify tasks where standard models suffice

- Identify tasks requiring premium models
- Create personal guidelines for model selection

4. Ask Mode Advantage:

- Use Ask mode with premium models (no token cost)
- Compare to Agent mode token consumption

Success Criteria

- Compared at least 2 different models
- Understand token consumption impact
- Can make informed model selection decisions

Token Economics

- **Standard models (unlimited):** ChatGPT-4
- **Premium tokens (counted):**
- Claude Haiku 4.5: 1/3 token per request
- Claude Sonnet: 1x token per request
- O2 mini: 1/3 token per request
- New models: May be 10x when first released

Evergreen Software Development - Core Principles



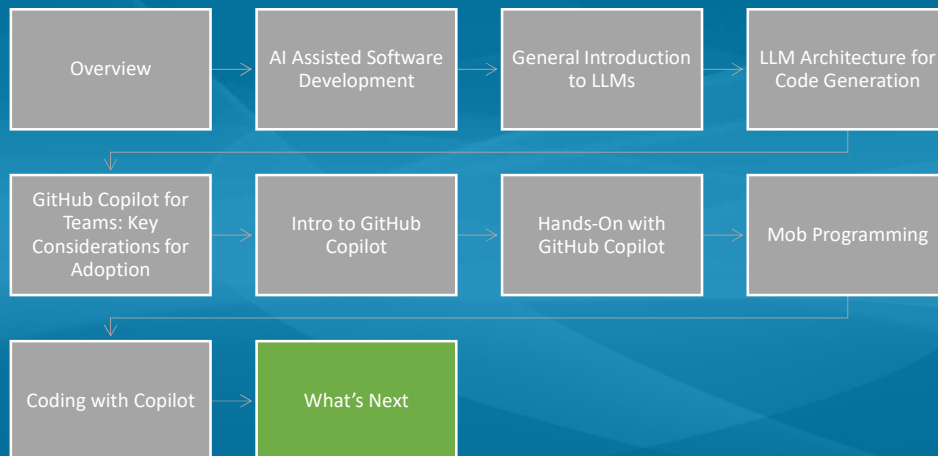
- **Intent-First Design**
 - Define the system's purpose, invariants, and boundaries before writing code to ensure long-term clarity.
- **Stable Interfaces, Evolving Internals**
 - Keep contracts predictable while allowing implementations to improve continuously.
- **Continuous Regeneration with Guardrails**
 - Use AI to rewrite or extend components safely, backed by tests, specs, and architectural constraints.
- **Modular, Replaceable Components**
 - Structure the system so any part can be regenerated, swapped, or upgraded without cascading breakage.
- **Lifecycle Governance**
 - Maintain quality through automated tests, versioning discipline, and human-in-the-loop validation.

Why Software Fails to Be Evergreen



- **Intent Rot**
 - The original purpose, constraints, and invariants are undocumented or lost, making safe regeneration impossible.
- **Unstable or Leaky Interfaces**
 - APIs, data contracts, and boundaries change unpredictably, causing cascading breakage when internals evolve.
- **Tightly Coupled Architecture**
 - Components depend on each other's internal details, preventing isolated regeneration or replacement.
- **Insufficient Guardrails**
 - Missing tests, specs, or validation layers mean AI-assisted regeneration can't be trusted to preserve behavior.
- **One-Off Patches and Drift**
 - Ad-hoc fixes accumulate, diverging the system from its intended design and making regeneration unsafe.

Agenda



Agenda: Quick walkthrough of topics; highlight hands-on demos and exercises.

Wrap-up



- Final Q&A
- Follow-up
- Support Files
- Thanks!