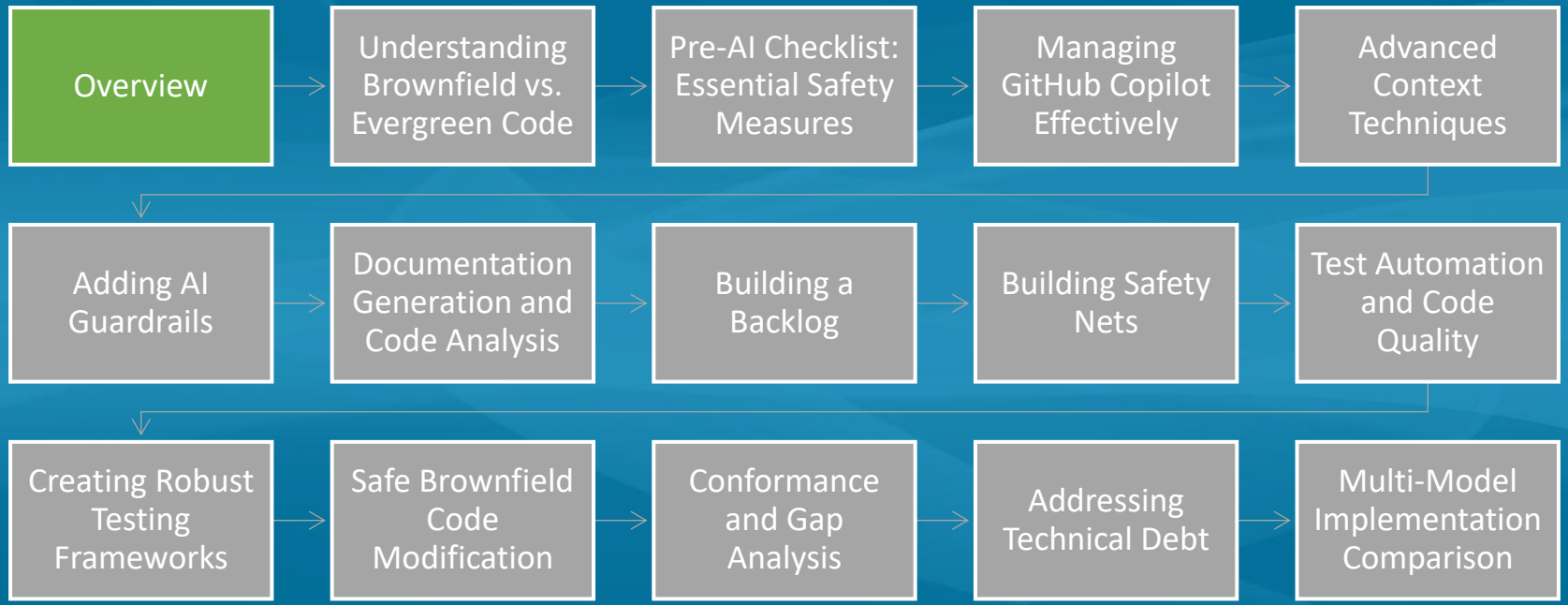
CODE TRAINING

AI Assisted Software Development

From Code to Copilot

Agenda

CODE TRAINING



Overview

CODE TRAINING

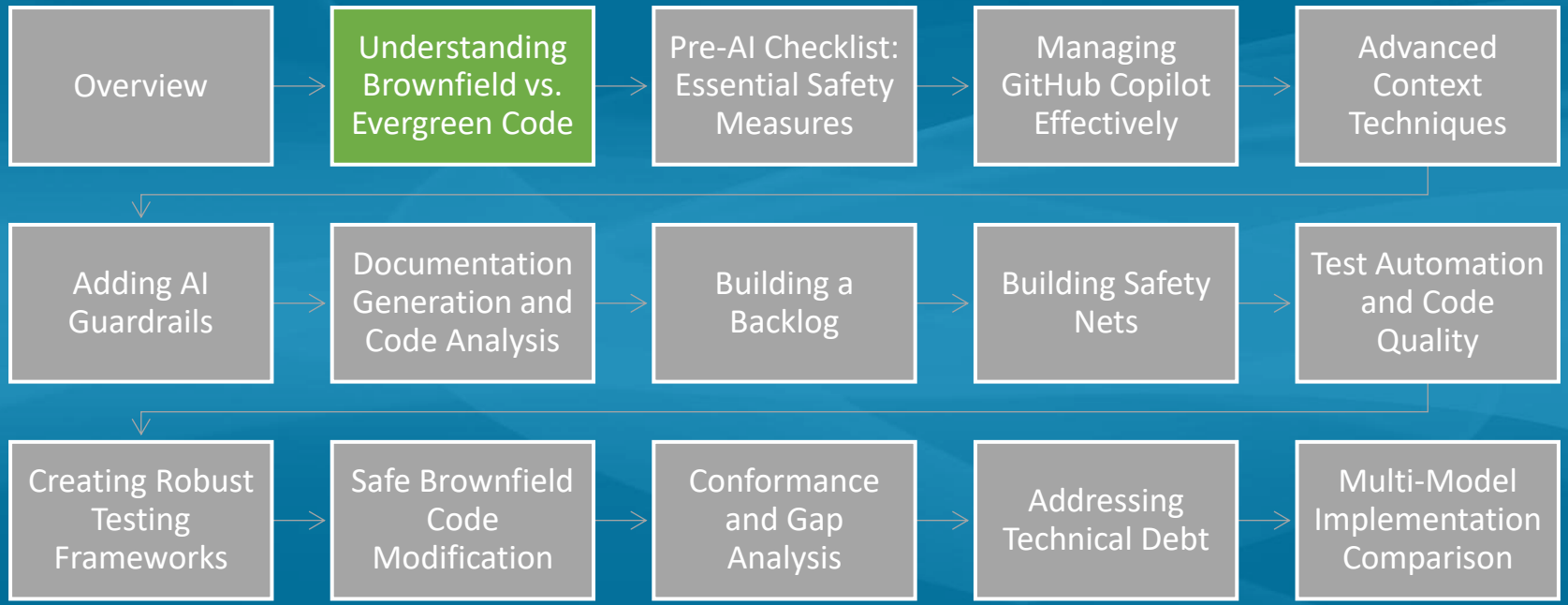
Course Objective: Understand how to manipulate Copilot into successfully modifying legacy codebases

Course Perspective: Focus on AI assistance and assisting AI in developing software

- Protecting existing codebases from AI agents
- Achieving evergreen maturity
- Moving fast without breaking things

Agenda

CODE TRAINING



Lab: Clone the AI-Assisted-Software-Development Repository

Duration: 10 minutes

Prerequisites: Git, GitHub account

Objectives

- Fork the AI-Assisted-Software-Development repo

Activities

1. Clone the git@github.com:johnmillerATcodemag-com/AI-Assisted-Software-Development.gitrepository
2. Switch to the brownfield branch

Success Criteria

- Cloned repository exists locally

What Defines Brownfield Code

CODE TRAINING

- Brownfield code is
  - Existing systems with history, constraints, and accumulated decisions
  - Code shaped by real users, real deadlines, and real production incidents
  - Software that has survived contact with reality
  - It’s the best we could do at the time with the tools and resources available
- Brownfield code is not
  - “Bad code”
  - “Legacy” in the pejorative sense
  - A sign of failure or poor engineering

Why Codebases Degrade

CODE TRAINING

- Shifting business requirements
  - Code reflects the business at the moment it was written
- Team turnover and knowledge loss
  - Tribal knowledge evaporates
- Technology evolution
  - Frameworks, languages, and patterns age
- Entropy and patchwork fixes
  - Quick fixes accumulate
  - Architectural drift emerges
- Missing or outdated guardrails
  - Tests, documentation, and standards fall behind

The Evergreen Philosophy

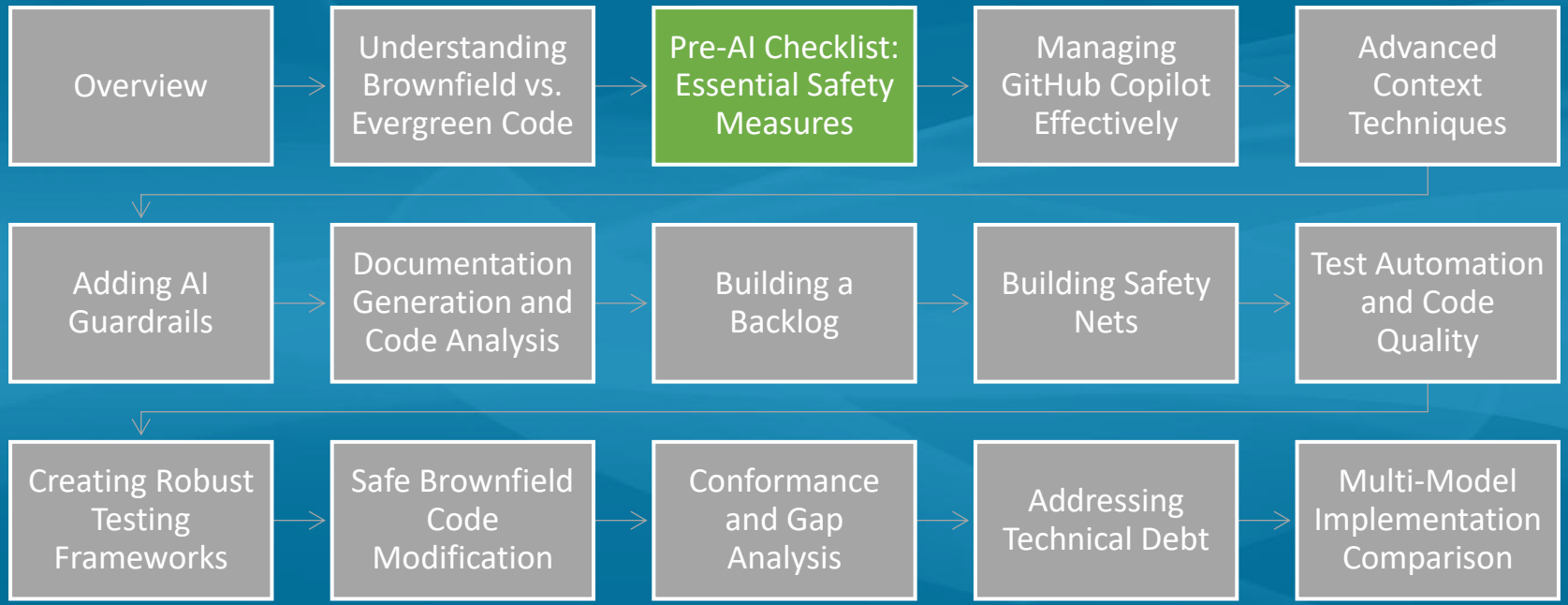
- Evergreen code asks one question:

  “If we rewrote this today, would it look the same?”
- If the answer is “yes,” the system is evergreen
- If the answer is “no,” the gap defines your modernization roadmap
- Evergreen is about continuous alignment, not perfection
- Small, frequent improvements beat large rewrites

Why Evergreen?

- Evergreen is a high bar that all brownfield codebases should strive to achieve
- Before AIASD
  - Evergreen was difficult to maintain
  - Moving from brownfield to evergreen was cost prohibited
- AI can assist with
  - Identifying technical debt
  - Prioritizing modernizations
  - Implementing improvements
  - Validating implementations
  - Documenting progress

Agenda


Overview
Understanding Brownfield vs. Evergreen Code
Pre-AI Checklist: Essential Safety Measures
Managing GitHub Copilot Effectively
Advanced Context Techniques
Adding AI Guardrails
Documentation Generation and Code Analysis
Building a Backlog
Building Safety Nets
Test Automation and Code Quality
Creating Robust Testing Frameworks
Safe Brownfield Code Modification
Conformance and Gap Analysis
Addressing Technical Debt
Multi-Model Implementation Comparison

Essential Safety Measures

CODE TRAINING

- AI accelerates development, but it also accelerates mistakes
- Strong safety nets must be in place before introducing AI into a brownfield codebase
- These practices reduce risk, increase confidence, and protect production systems

Backup & Rollback Strategies

CODE TRAINING

- Use branching strategies that isolate AI-generated changes
- Commit early and often to create natural rollback points
- Archive snapshots of critical modules before modernization
- Ensure you can revert any AI-assisted change without drama
- Use feature flags to separate release from deployment

Confidence Frameworks

CODE TRAINING

- Strong tests are the backbone of safe AI-assisted refactoring
- Unit, integration, and behavioral tests validate AI output
- Coverage matters less than signal quality
- Tests should detect regressions, not just assert happy paths
- If all of the test automation passes, how confident are you to deploy to production?

Change Review Processes

- Treat AI as a junior developer: everything gets reviewed
- Use human-in-the-loop validation for correctness and intent
- Require architectural review for structural changes
- Enforce standards through linters, static analysis, and policy checks
- Leverage AI to reduce the review burden

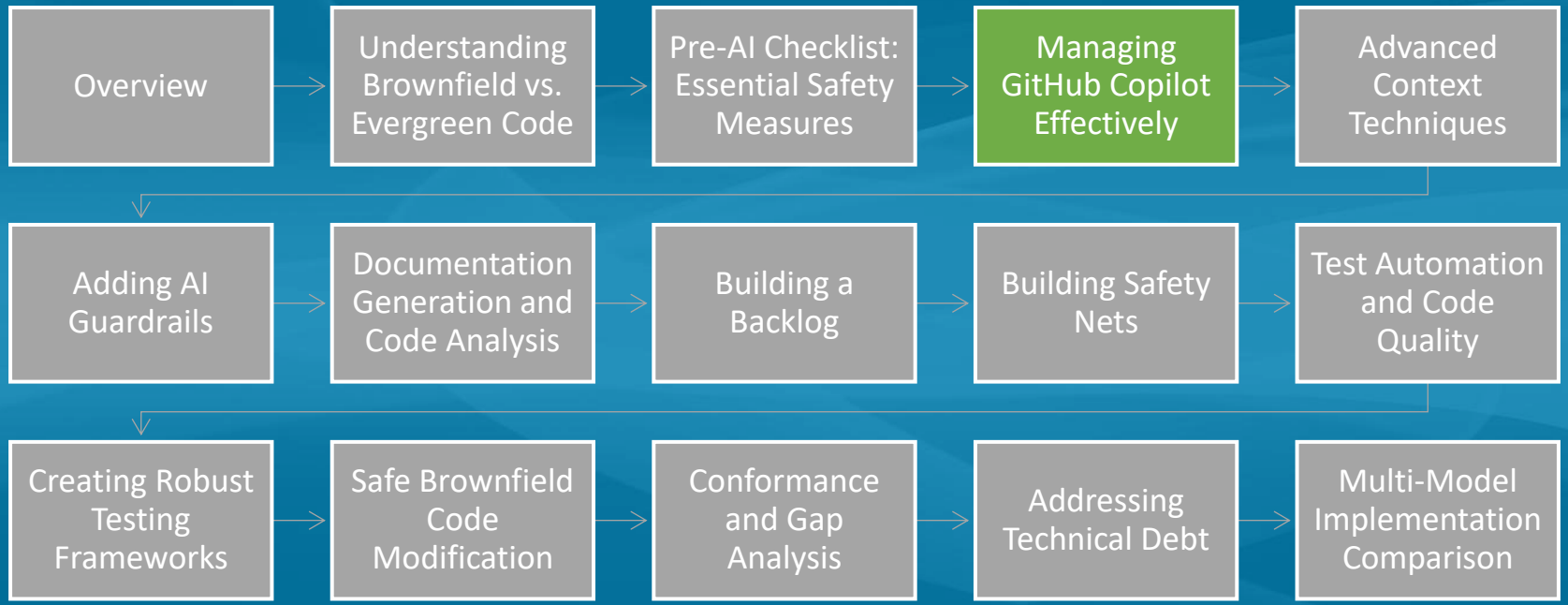
Keeping Change Sets Small

CODE TRAINING

- Small diffs are easier to review and validate
- Small changes reduce merge conflicts and regression risk
- AI should be instructed to limit scope intentionally
- Small changes accumulate into large improvements over time
- Beware: AI can produce huge amounts of code quickly

Agenda

CODE TRAINING



Managing GitHub Copilot Effectively

CODE TRAINING

- Copilot is powerful, but not entirely autonomous
- Effective use requires structure, guardrails, and clear intent
- Treat Copilot as a developer whose output improves with guidance
- Your process determines the quality of its contributions

Understanding Context & Tokens

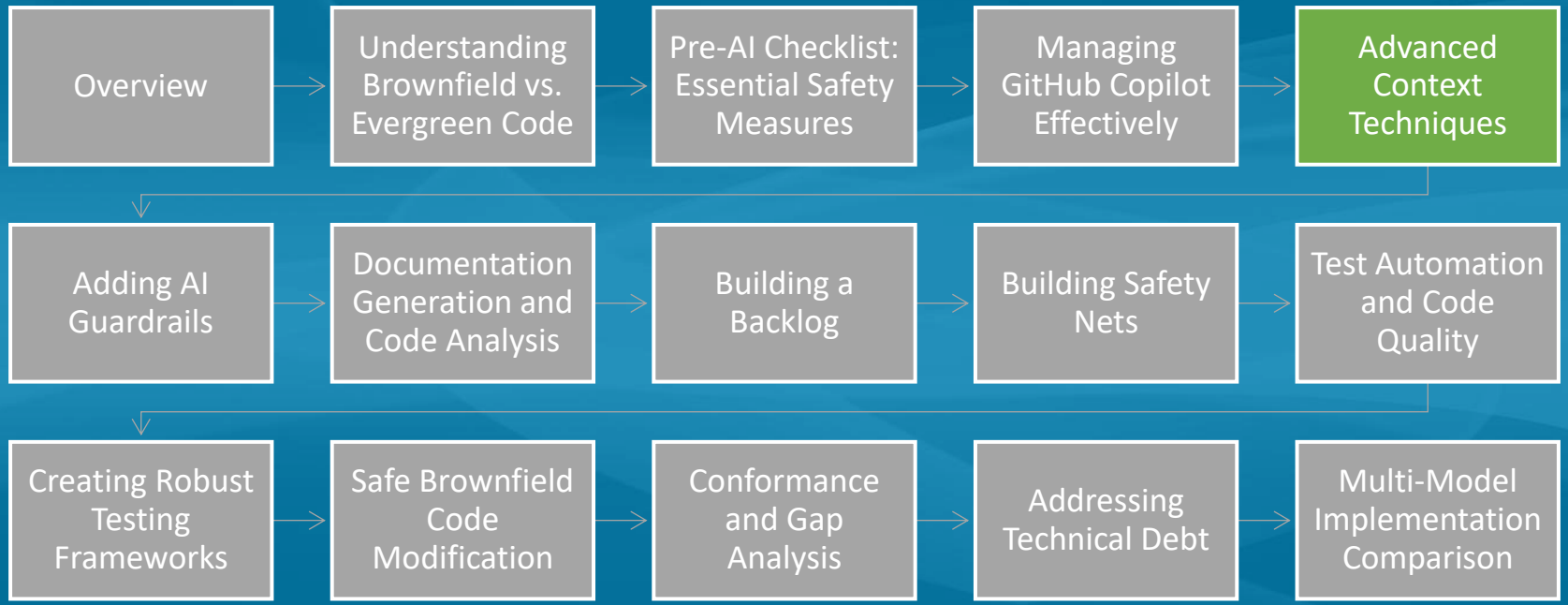
- Copilot can only “see” a limited amount of text at once
- Large files, long conversations, or complex repos can exceed context
- Important details may fall out of the window without you realizing
- Use these techniques to keep context focused:
  - Summaries
  - Instruction files
  - Modular prompts
  - Smaller working sets

Prompt Engineering Best Practices

CODE TRAINING

- Be explicit about goals, constraints, and success criteria
- Provide examples of the desired pattern or style
- Break large tasks into smaller, testable steps
- Use instruction files for stable rules and architectural boundaries
- Ask Copilot to explain its reasoning when correctness matters

Agenda


Overview
Understanding Brownfield vs. Evergreen Code
Pre-AI Checklist: Essential Safety Measures
Managing GitHub Copilot Effectively
Advanced Context Techniques
Adding AI Guardrails
Documentation Generation and Code Analysis
Building a Backlog
Building Safety Nets
Test Automation and Code Quality
Creating Robust Testing Frameworks
Safe Brownfield Code Modification
Conformance and Gap Analysis
Addressing Technical Debt
Multi-Model Implementation Comparison

Advanced Context Techniques

CODE TRAINING

- Modern AI tools rely heavily on context quality
- Developers can shape context intentionally
- Reduces hallucinations, drift, and rework
- Strong context discipline is a core AI-era skill

Token Estimation & Overflow Detection

CODE TRAINING

- Models have strict token limits
- Overflow causes silent failures:
  - Missing requirements
  - Contradictions
  - Forgotten rules
- Techniques to stay within limits:
  - Summaries
  - Chunking
  - Scoped prompts
  - Instruction files

Silent Failure Modes

CODE TRAINING

What Overflow Looks Like

- Missing requirements
- Contradictions
- Forgotten rules
- Inconsistent reasoning
- Loss of architectural constraints

Technique: Summaries

CODE TRAINING

How Summaries Help

- Compress large files into short, high-signal descriptions
- Preserve intent without overwhelming the context window
- Reuse summaries across prompts
- Reduce noise and improve model alignment

Technique: Chunking

CODE TRAINING

How Chunking Works

- Break large tasks into smaller, self-contained steps
- Provide only the relevant portion of the code
- Validate each chunk before moving on
- Prevents the model from being overloaded

Technique: Scoped Prompts

CODE TRAINING

Benefits

- Limit the model’s focus to a single module or function
- Reduce irrelevant context
- Improve accuracy and reduce hallucinations
- Keep token usage predictable

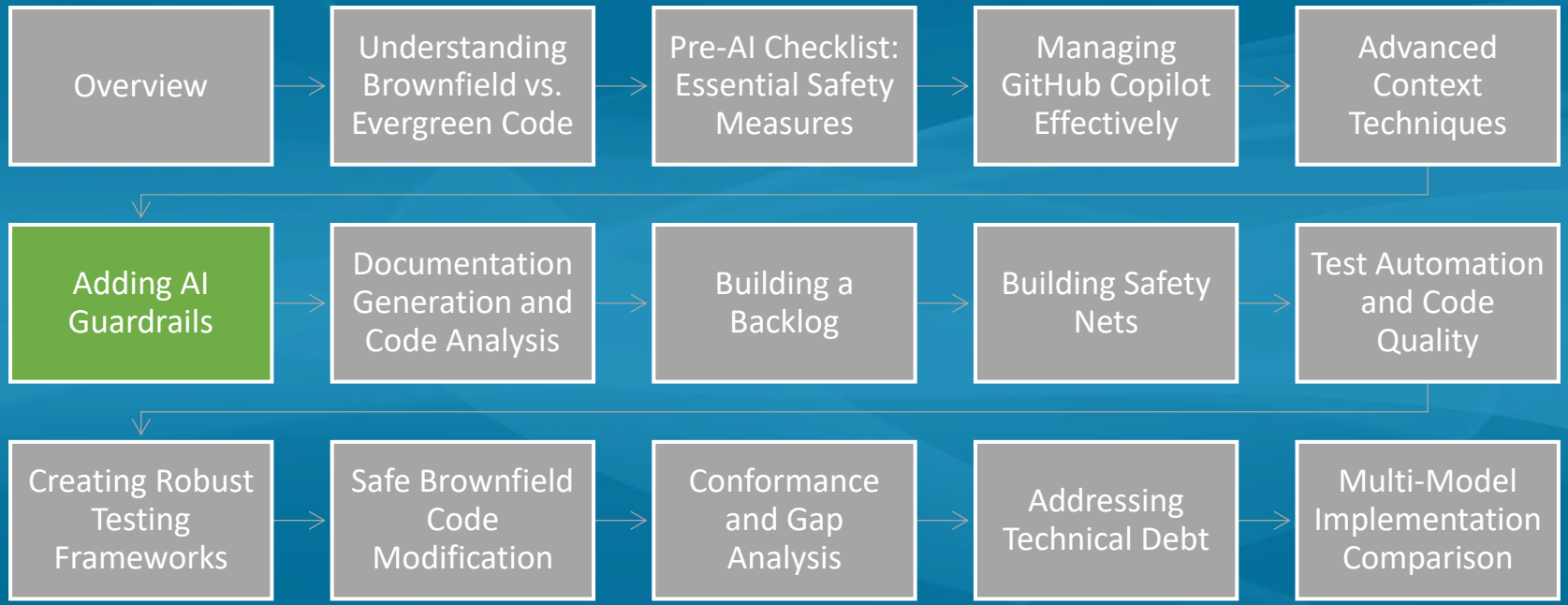
Technique: Instruction Files

CODE TRAINING

Why They Matter

- Move stable rules out of the active prompt
- Provide persistent architectural and style guidance
- Reduce repeated tokens across sessions
- Keep prompts short and high-signal

Agenda


Overview
Understanding Brownfield vs. Evergreen Code
Pre-AI Checklist: Essential Safety Measures
Managing GitHub Copilot Effectively
Advanced Context Techniques
Adding AI Guardrails
Documentation Generation and Code Analysis
Building a Backlog
Building Safety Nets
Test Automation and Code Quality
Creating Robust Testing Frameworks
Safe Brownfield Code Modification
Conformance and Gap Analysis
Addressing Technical Debt
Multi-Model Implementation Comparison

Adding AI Guardrails

CODE TRAINING

- What are instructions, prompts, and Agents
- Creating instruction, prompt, and Agent files
- Meta prompts that generate these files
- Instructions for generating artifacts
- Enforcing provenance for AI-assisted artifacts

Instructions, Prompts & Agents

CODE TRAINING

Definitions

- **Instructions** – Persistent rules that guide the model’s behavior
- **Prompts** – Task-specific requests defining intent and constraints
- **Agents** – Pre-configured personas optimized for workflows

Creating Instruction, Prompt & Agent Files

CODE TRAINING

Why create files?

- Ensures repeatability
- Reduces token usage
- Provides version-controlled guardrails
- Enables team-wide consistency

File types

- `.github/instructions/myinstructions.instructions.md`
- `.github/prompts/myprompt.prompt.md`
- `.github/chatmodes/mychatmode.chatmode.md`

Meta Prompts

CODE TRAINING

Meta prompts guide:

- Creation of instruction files
- Generation of reusable prompts
- Construction of Agents
- Provide consistent formatting, structure, content

Instructions for Generating Artifacts

CODE TRAINING

Best practices

- Define the artifact type
- Specify required sections
- Provide examples or templates
- Include acceptance criteria
- Require the model to restate constraints

Enforcing Provenance for AI Artifacts

Provenance requirements

- Declare:
  - AI involvement
  - Model used
  - Date generated
  - Human reviewer
- Store provenance in headers, footers, or side cars
- Track revisions in version control

Exercise: Copy the Core Instructions

Duration: 10 minutes

Objectives:

- Understand file organization for AI-assisted output policies
- Practice copying files between repositories
- Ensure compliance with output metadata requirements

Activities:

1. Locate `.github/instructions/ai-assisted-output.instructions.md` in the AI-Assisted-Software-Development repository
2. Copy the file into the `.github/instructions` folder of the current repository
3. Copy these files as well:
   - chatmode-file.instructions.md
   - instruction-files.instructions.md
   - instruction-prompt-files.instructions.md
   - prompt-file.instructions.md
4. Verify the copied files matches the original
5. Review the instructions

Success Criteria:

- The files are present in the current repo
- The content matches the source file
- No metadata or formatting is lost

Instructions for AI Generated Artifacts

The one instruction file that rules them all

AI-Assisted Output Instructions

- Ensures provenance and logging for all AI-assisted outputs
- Defines required metadata, logging workflow, and quality gates
- Protects code quality and enables audits

Required Provenance Metadata

CODE TRAINING

- Every AI-assisted artifact must include:
  - ai_generated: true
  - model: provider/model@version
  - operator: username
  - chat_id: unique chat identifier
  - prompt: exact prompt text
  - started/ended: timestamps
  - task_durations & total_duration
  - ai_log: path to conversation log
  - source: who/what created the file

Metadata Placement Policy

CODE TRAINING

- Use YAML front matter for Markdown and similar formats
- For binaries/images, use a sidecar `<artifact>.meta.md`
- Never use sidecars for Markdown

AI Chat Logging Workflow

- Each chat creates a unique log folder: `ai-logs/yyyy/mm/dd/<chat-id>/`
- Required files:
  - conversation.md (full transcript)
  - summary.md (objectives, decisions, outcomes)
  - artifacts/ (optional)
- Never reuse chat logs between sessions

Summary: Why This Matters

- Enables auditability and trust in AI outputs
- Protects against orphaned or unverifiable artifacts
- Supports team collaboration and compliance

Core Instruction files

- chatmode-file.instructions.md
  - Defines the structure and contents of agents
- instruction-files.instructions.md
  - Defines the structure and contents of instruction files
- prompt-file.instructions.md
  - Defines the structure and contents of prompts
- instruction-prompt-files.instructions.md
  - Defines the structure and contents of prompts the create instruction files

Exercise: Create a Prompt File

Duration

10 minutes

Objectives

- Understand prompt structure
- Practice defining task intent
- Apply constraints and success criteria

Activities

1. Prompt Copilot to create a prompt file that creates an instruction file for evergreen software development
2. Review the prompt

Success Criteria

- Prompt is clear, scoped, and reusable
- Includes constraints and success criteria
- Avoids unnecessary context

Exercise: Create an Instruction File for Evergreen Development

CODE TRAINING

Duration

15 minutes

Objectives

- Capture evergreen principles
- Define architectural boundaries
- Specify modernization rules

Activities

1. Submit the Evergreen Instructions prompt
2. Review the instructions

Success Criteria

- Instruction file is stable and reusable
- Reflects evergreen development values
- Provides clear guardrails

Exercise: Generate Instruction Files

Duration

20 minutes

Objectives

- Use meta prompts to scale instruction-file creation
- Capture module-specific rules
- Encode domain and architectural constraints

Activities

1. Prompt Copilot to create instruction files for the standards and conventions of the tech stack
2. Review instructions

Success Criteria

- Instruction files reflect real system constraints
- Meta prompts produce consistent structure
- Files are ready for team use

Exercise: Context-Related Issues

Duration

10 minutes

Objectives

- Identify missing context
- Detect token overflow risks
- Improve prompt scoping

Activities

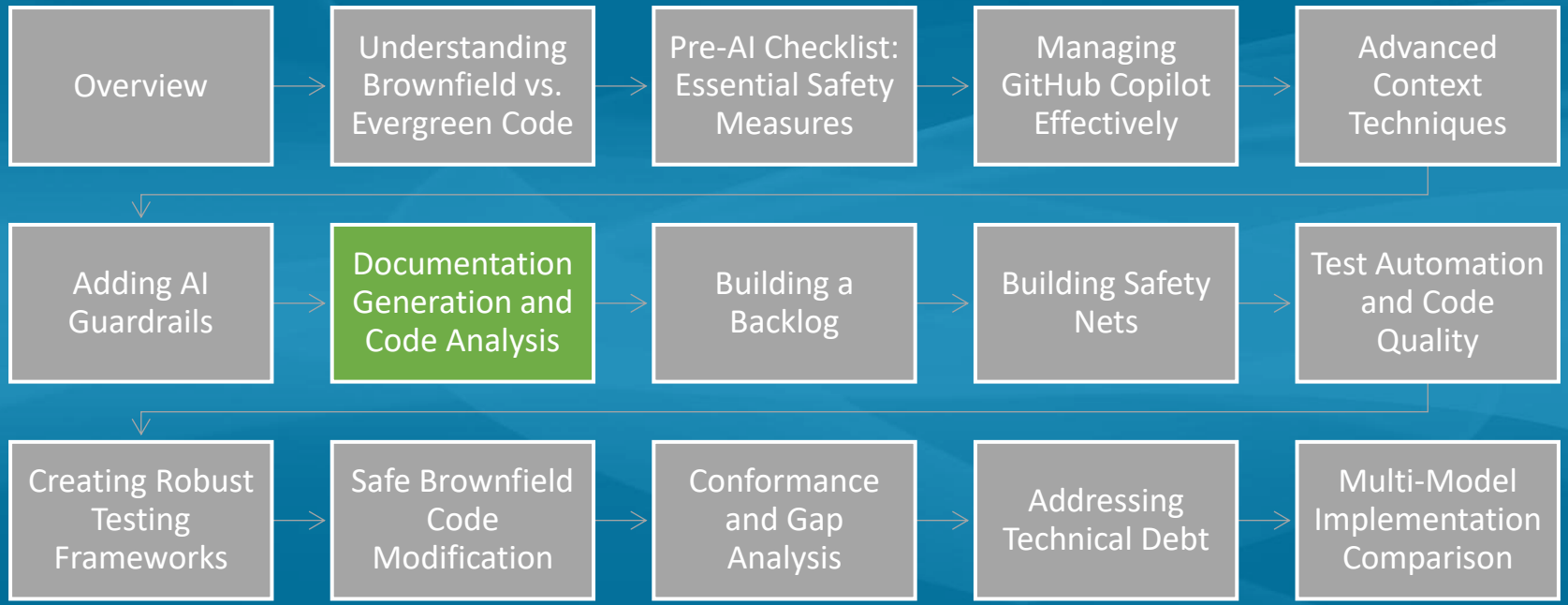
1. Copy the check-context.prompt.md file from the AIASD repository
2. Review the prompt
3. Submit the prompt
4. Review the output

Success Criteria

- Correctly identified context gaps

Agenda

CODE TRAINING



Documentation Generation & Code Analysis

CODE TRAINING

- Automated README and documentation updates
- Architecture diagram generation
- Complex code explanation and mapping
- Identifying technical debt hotspots
- Exercises for hands-on practice

Automated README & Documentation Updates

CODE TRAINING

Capabilities

- Generate or update README files
- Create module-level documentation
- Produce API references and usage examples
- Keep documentation aligned with code changes
  - Create a documentation instruction file

Architecture Diagram Generation

CODE TRAINING

What AI can generate

- High-level system diagrams
- Module dependency graphs
- Data flow diagrams
- Deployment topologies

Complex Code Explanation & Mapping

CODE TRAINING

AI can help with:

- Explaining unfamiliar or legacy code
- Mapping call chains and dependencies
- Identifying hidden coupling
- Translating code into human-readable narratives

Identifying Technical Debt Hotspots

CODE TRAINING

AI can detect:

- Outdated patterns
- Duplicate logic
- Missing tests
- High-complexity functions
- Security risks

Exercise: Brownfield Code Documentation

CODE TRAINING

Duration

15 minutes

Objectives

- Practice generating documentation for legacy code
- Identify missing or unclear areas
- Produce high-signal summaries

Activities

1. Select a brownfield module or file
2. Ask AI to generate:
   - A summary
   - Key responsibilities
   - Inputs/outputs
   - Known risks
3. Add provenance metadata
4. Review with a partner

Success Criteria

- Documentation is accurate and concise
- Risks and gaps are clearly identified
- Provenance is included

Create Architecture Diagrams

CODE TRAINING

AI-generated diagrams include:

- System boundaries
- Module interactions
- Data flows
- Deployment environments

Exercise: Fork the AIASD-20260209-BF Repo

Duration

20 minutes

Objectives

- Explore an unfamiliar codebase

Activities

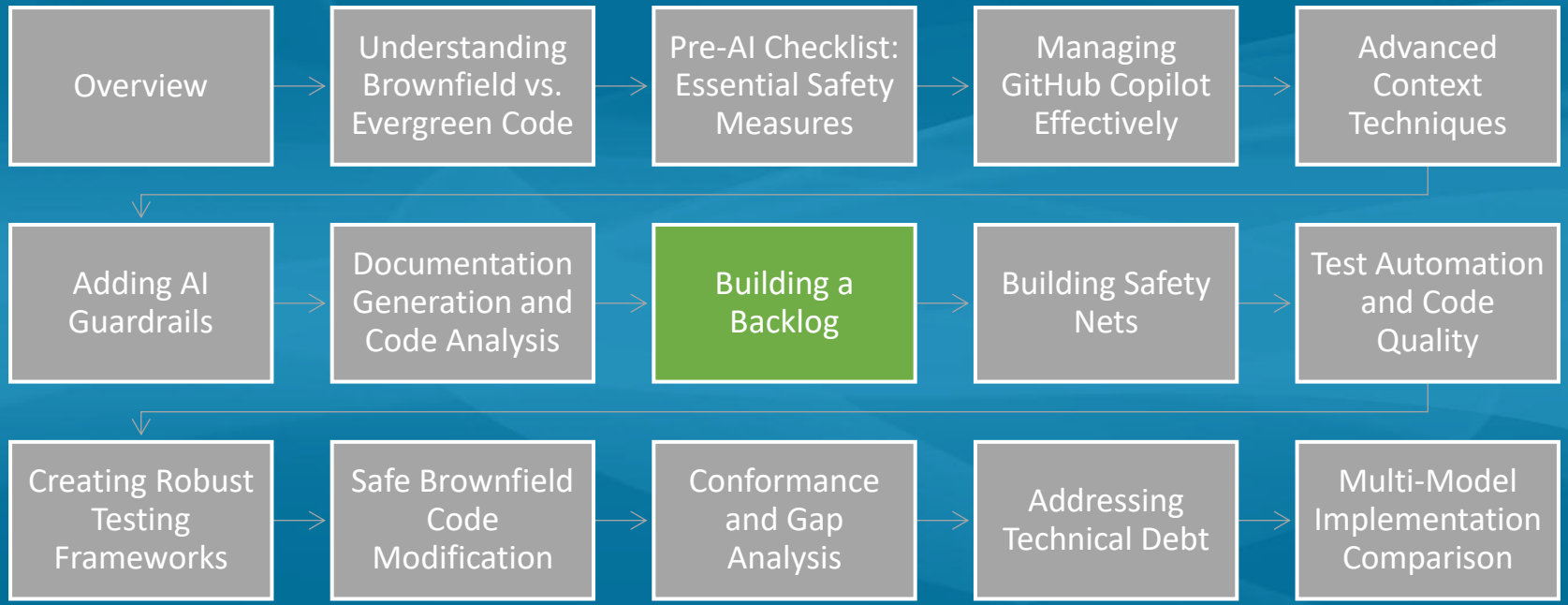
1. Fork this repo
   https://github.com/j0hnnymiller/AIASD-20260209-BF.git
2. Clone the forked repo
3. Create a GitHub PAT
   https://github.com/settings/tokens
4. Store the PAT in the GITHUB_TOKEN environment variable
5. Add github instructions, prompts
6. Generate documentation, diagrams etc.

Success Criteria

- Repo is available locally
- Project documentation created

Agenda

CODE TRAINING



Building a Backlog

CODE TRAINING

- Identifying technical debt
- Automating the creation of GitHub issues
- Exercise: Building the backlog

Identifying Technical Debt

CODE TRAINING

AI can surface:

- Outdated patterns
- High-complexity functions
- Duplicate logic
- Missing tests
- Security vulnerabilities
- Architectural drift

Benefits

- Faster discovery
- More consistent classification
- Prioritized modernization roadmap

Exercise: Building the Backlog

Duration

20 minutes

Objectives

- Practice identifying technical debt
- Convert findings into actionable GitHub issues
- Apply consistent structure and provenance
- Prioritize issues based on risk and impact

Activities

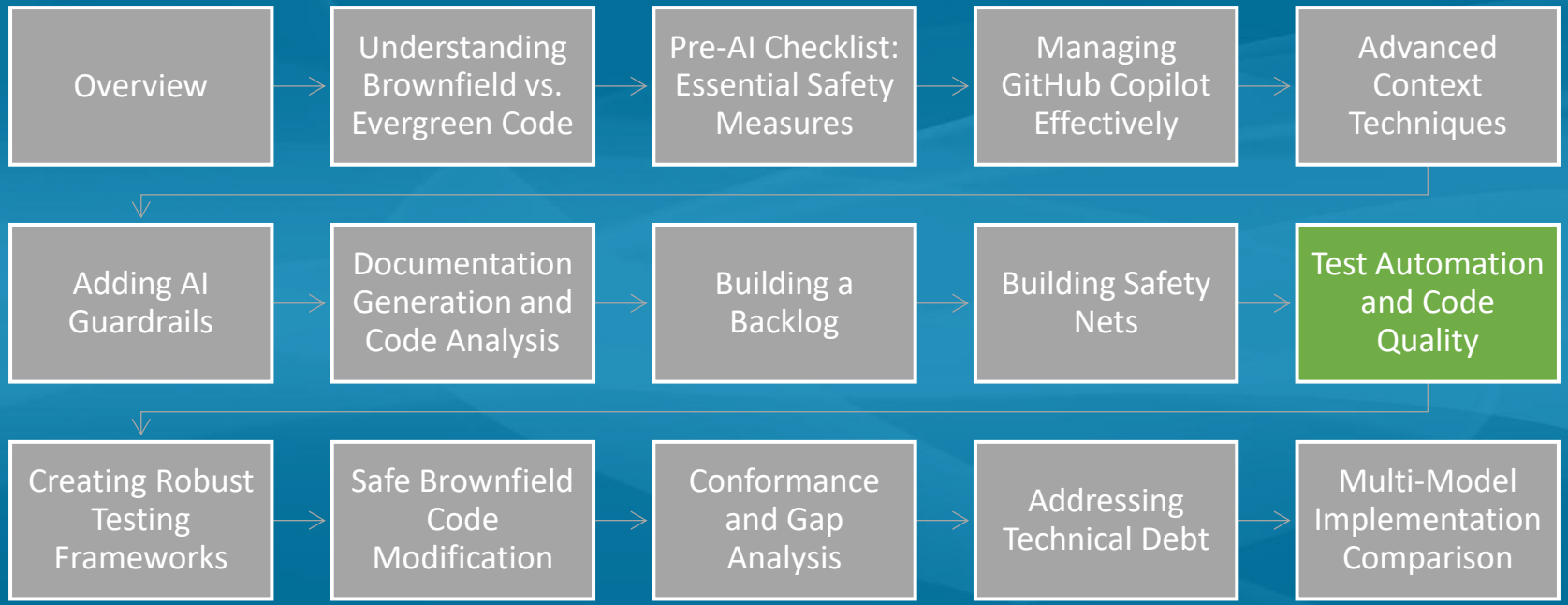
1. Select a brownfield module or file.
2. Ask AI to identify:
   - Technical debt
   - Risks
   - Missing tests
   - Architectural violations
3. Convert each finding into a GitHub issue with:
   - Title
   - Description
   - Acceptance criteria
   - Labels
   - Provenance metadata
4. Prioritize the issues using impact vs. effort.
5. Share your backlog with a partner for review.

Success Criteria

- Issues are clear, actionable, and well-structured
- Provenance metadata is included
- Prioritization reflects real risk and effort
- Backlog is ready for team review

Agenda

CODE TRAINING



Test Automation & Code Quality

CODE TRAINING

- AI-assisted test generation (unit, integration, E2E)
- Intelligent linting beyond static analysis
- Coverage analysis and test adequacy assessment
- Automated quality gates
- Exercise: Strengthening test automation & quality

AI-Assisted Test Generation

CODE TRAINING

AI can generate:

- **Unit tests** for functions, classes, and utilities
- **Integration tests** for module interactions
- **End-to-end tests** for full workflows
- **Edge-case tests** and regression scenarios
- **Contract tests** for APIs and services

Benefits

- Rapid coverage expansion
- Consistent structure and naming
- Reduced onboarding time

Intelligent Linting

CODE TRAINING

AI-enhanced linting can detect:

- Architectural violations
- Anti-patterns
- Unsafe refactors
- Missing documentation
- Inconsistent naming or domain terminology

Why it matters

- Goes beyond syntax
- Enforces architectural guardrails
- Reduces long-term technical debt

Coverage Analysis

AI can help evaluate:

- Coverage gaps
- Missing edge cases
- Over-testing of implementation details
- Under-testing of business logic
- Redundant or brittle tests

Outcomes

- More meaningful coverage
- Better alignment with real behavior
- Reduced maintenance burden

Automated Quality Gates

Quality gates can enforce:

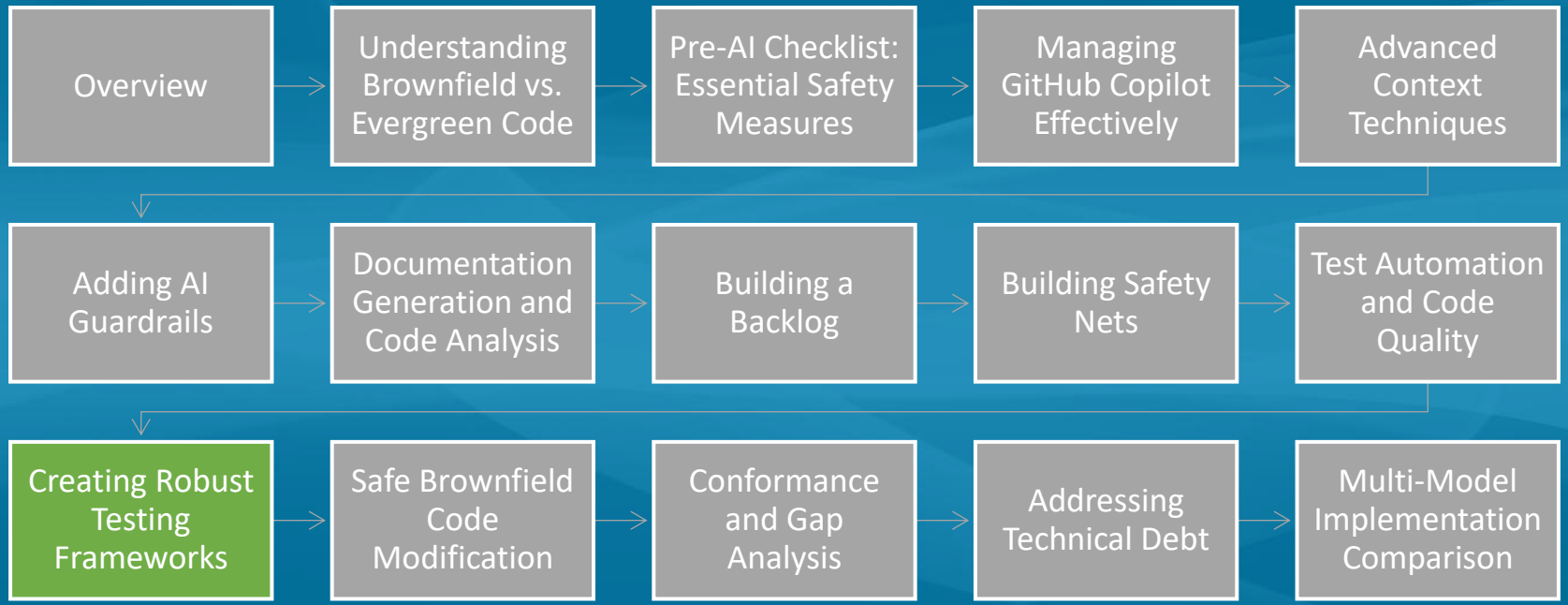
- Minimum test coverage
- Linting and architectural checks
- Provenance requirements
- PR-level test generation
- Risk scoring for changes

Benefits

- Prevents regressions
- Ensures consistent quality
- Supports evergreen development

Agenda

CODE TRAINING



Creating Robust Testing Frameworks

CODE TRAINING

- Generating comprehensive test suites
- Managing test suites over time
- Test review and validation strategies
- Balancing test coverage with maintainability
- Exercise: Strengthening your testing framework

Managing Test Suites Over Time

CODE TRAINING

Key Practices

- Regularly prune obsolete tests
- Update tests alongside code changes
- Maintain clear naming and structure
- Use coverage reports to guide improvements
- Version-control test strategy documents

Test Review & Validation Strategies

AI-assisted review can:

- Detect missing assertions
- Identify redundant tests
- Suggest edge cases
- Flag inconsistent patterns

Human reviewers focus on:

- Intent correctness
- Business logic validation
- Architectural alignment

Exercise: Strengthening Your Testing Framework

Duration

20 minutes

Objectives

- Identify gaps in an existing test suite
- Use AI to generate missing tests
- Improve maintainability and structure
- Validate tests for correctness and intent

Activities

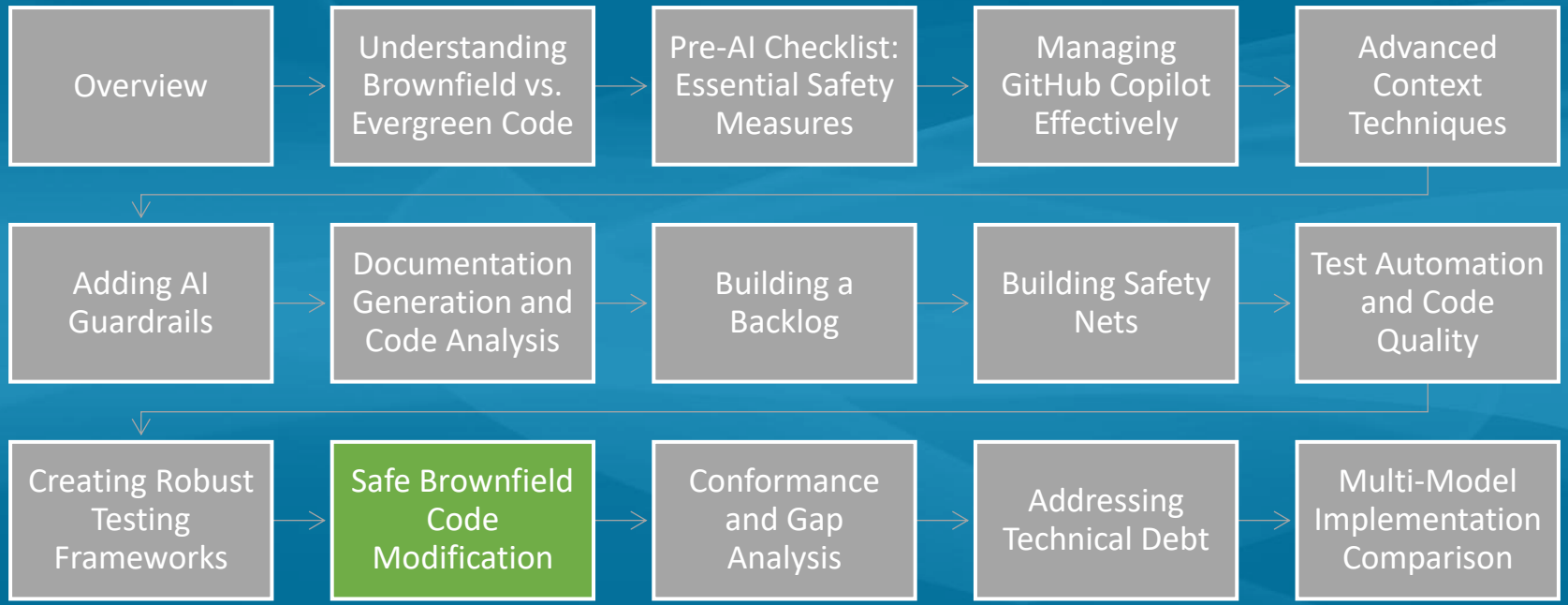
1. Select a brownfield module or function.
2. Review existing tests for:
   - Coverage gaps
   - Redundant or brittle tests
   - Missing edge cases
3. Ask AI to generate missing tests.
4. Validate AI-generated tests for correctness.
5. Refactor or reorganize tests for clarity.
6. Add provenance metadata to all new tests.

Success Criteria

- Coverage gaps are identified and addressed
- AI-generated tests are validated and correct
- Test suite readability and structure improve
- Provenance metadata is included

Agenda

CODE TRAINING



Safe Brownfield Coding

CODE TRAINING

- Using feature flags to minimize risk
- As-Is and To-Be test suites
- Testing in production
- Retiring feature flags
- Exercise: Implementing a feature flag

Using Feature Flags

CODE TRAINING

Why feature flags matter

- Enable incremental rollout
- Allow instant rollback
- Reduce blast radius
- Support A/B testing and shadow traffic
- Decouple deployment from release

Best practices

- Keep flags short-lived
- Name flags clearly
- Document intent and retirement criteria

As-Is and To-Be Test Suites

CODE TRAINING

As-Is tests

- Capture current behavior
- Protect against regressions
- Document legacy expectations

To-Be tests

- Define desired future behavior
- Guide modernization
- Validate new patterns and architecture

Testing in Production

CODE TRAINING

Safe production testing techniques

- Feature-flag-controlled exposure
- Shadow traffic
- Canary releases
- Observability dashboards
- Error-budget-based rollout

Benefits

- Real-world validation
- Early detection of edge cases
- Reduced risk of full-scale failures

Retiring Feature Flags

CODE TRAINING

Why retirement matters

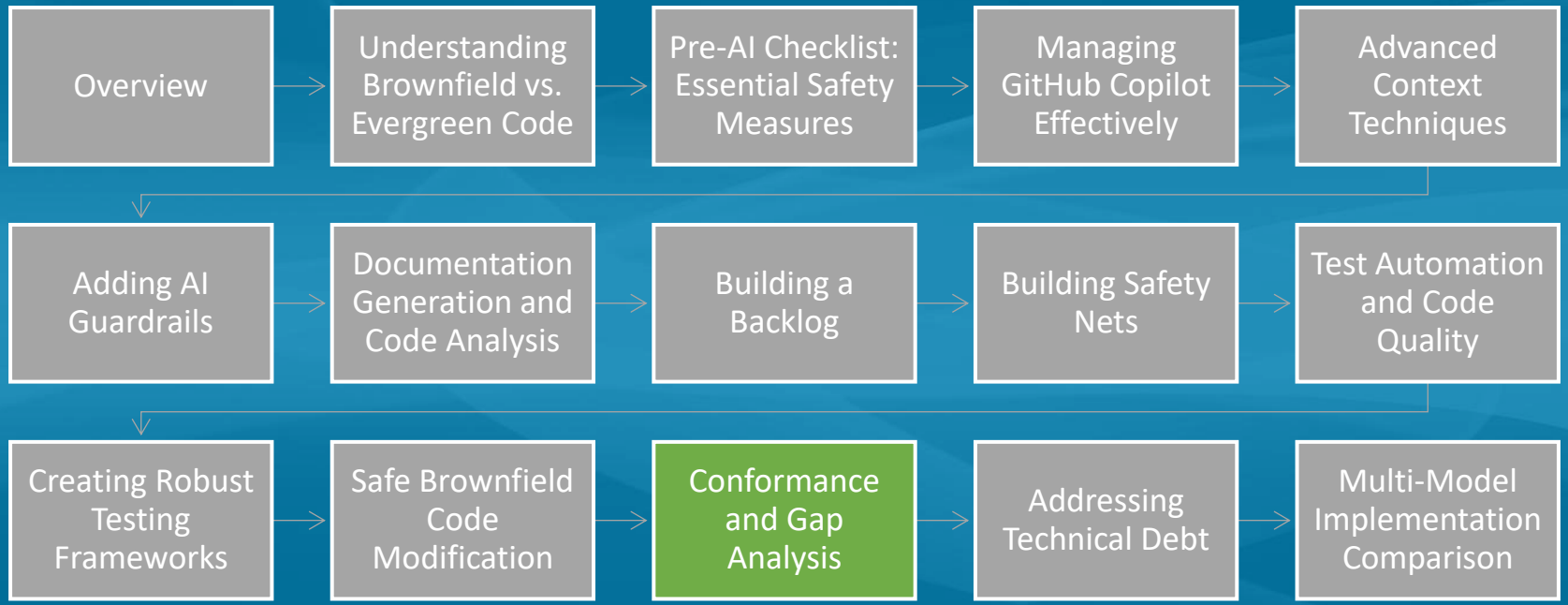
- Prevents flag bloat
- Reduces cognitive load
- Simplifies code paths
- Ensures long-term maintainability

Retirement workflow

- Validate stability
- Remove old code paths
- Update documentation
- Add provenance to the change

Agenda

CODE TRAINING



Conformance & Gap Analysis

CODE TRAINING

- Comparing implementations against instruction files
- Automated issue generation from conformance gaps
- Prioritizing technical debt remediation
- Creating actionable remediation plans
- Exercises for hands-on practice

Prioritizing Technical Debt Remediation

CODE TRAINING

Prioritization factors

- Risk to stability
- Frequency of use
- Security implications
- Architectural importance
- Effort vs. impact

Approaches

- Impact/effort matrix
- Risk scoring
- Dependency analysis

Creating Actionable Remediation Plans

A strong remediation plan includes:

- Clear problem definition
- Root cause analysis
- Proposed solution
- Step-by-step implementation plan
- Rollback strategy
- Test updates
- Provenance metadata

Exercise: Generate Issues to Make the Codebase Evergreen

Duration

15 minutes

Objectives

- Identify conformance gaps
- Convert gaps into actionable issues
- Apply consistent structure and provenance
- Prioritize issues based on risk and impact

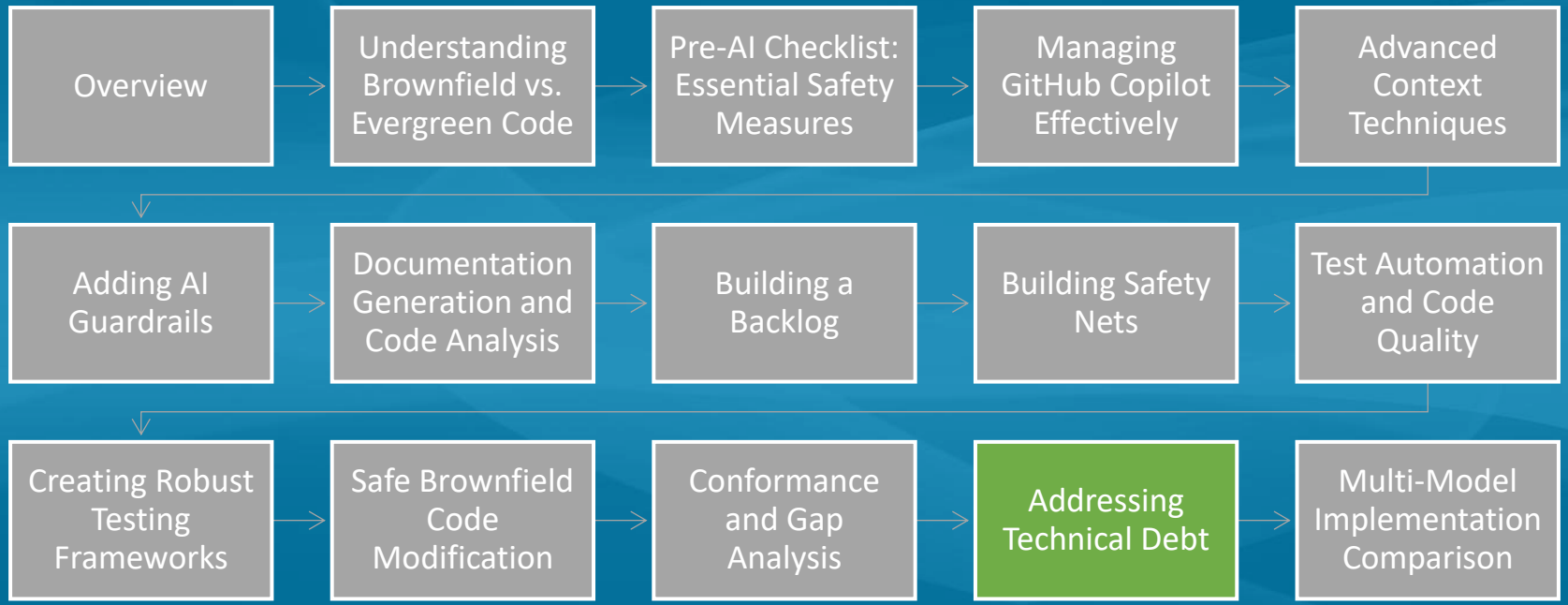
Activities

1. Select a brownfield module or file.
2. Compare it against the project's instruction file.
3. Ask AI to identify conformance gaps.
4. Convert each gap into a GitHub issue with:
   - Title
   - Description
   - Violated rule
   - Suggested remediation
   - Acceptance criteria
   - Provenance metadata
5. Prioritize the issues.

Success Criteria

- Issues are clear, actionable, and aligned with instruction files
- Provenance metadata is included
- Prioritization reflects real risk and effort
- Backlog is ready for team review

Agenda


Overview
Understanding Brownfield vs. Evergreen Code
Pre-AI Checklist: Essential Safety Measures
Managing GitHub Copilot Effectively
Advanced Context Techniques
Adding AI Guardrails
Documentation Generation and Code Analysis
Building a Backlog
Building Safety Nets
Test Automation and Code Quality
Creating Robust Testing Frameworks
Safe Brownfield Code Modification
Conformance and Gap Analysis
Addressing Technical Debt
Multi-Model Implementation Comparison

Addressing Technical Debt

CODE TRAINING

- Prompting Copilot to address debt
- Assigning issues to Copilot
- What Copilot does with assigned issues
- Exercises for hands-on practice

Prompting Copilot to Address Technical Debt

CODE TRAINING

Effective prompts include:

- Clear description of the debt
- Constraints and architectural rules
- Expected outcomes
- Required tests and documentation updates
- Provenance requirements

Benefits

- Faster remediation
- Consistent application of patterns
- Reduced manual effort

Assigning Issues to Copilot

CODE TRAINING

How assignment works

- Convert technical debt into GitHub issues
- Provide context, constraints, and acceptance criteria
- Use Copilot to draft remediation steps
- Let Copilot propose code changes in PRs

Why assign issues?

- Creates a repeatable workflow
- Keeps humans in the reviewer role
- Ensures traceability and provenance

Exercise: Prompt Copilot to Address Technical Debt

Duration

10 minutes

Objectives

- Practice writing high-signal prompts
- Apply architectural constraints
- Produce safe, incremental remediation requests

Activities

1. Select a small piece of technical debt.
2. Write a prompt that includes:
   - Description of the debt
   - Constraints and rules
   - Expected behavior
   - Required tests and documentation
3. Ask Copilot to propose a remediation.
4. Review the output for correctness.

Success Criteria

- Prompt is clear, scoped, and actionable
- Copilot produces a safe, incremental change
- Output aligns with architectural rules
- Provenance metadata is included

Exercise: Assigning an Issue to Copilot

Duration

10 minutes

Objectives

- Convert technical debt into a structured issue
- Provide Copilot with actionable context
- Practice writing acceptance criteria

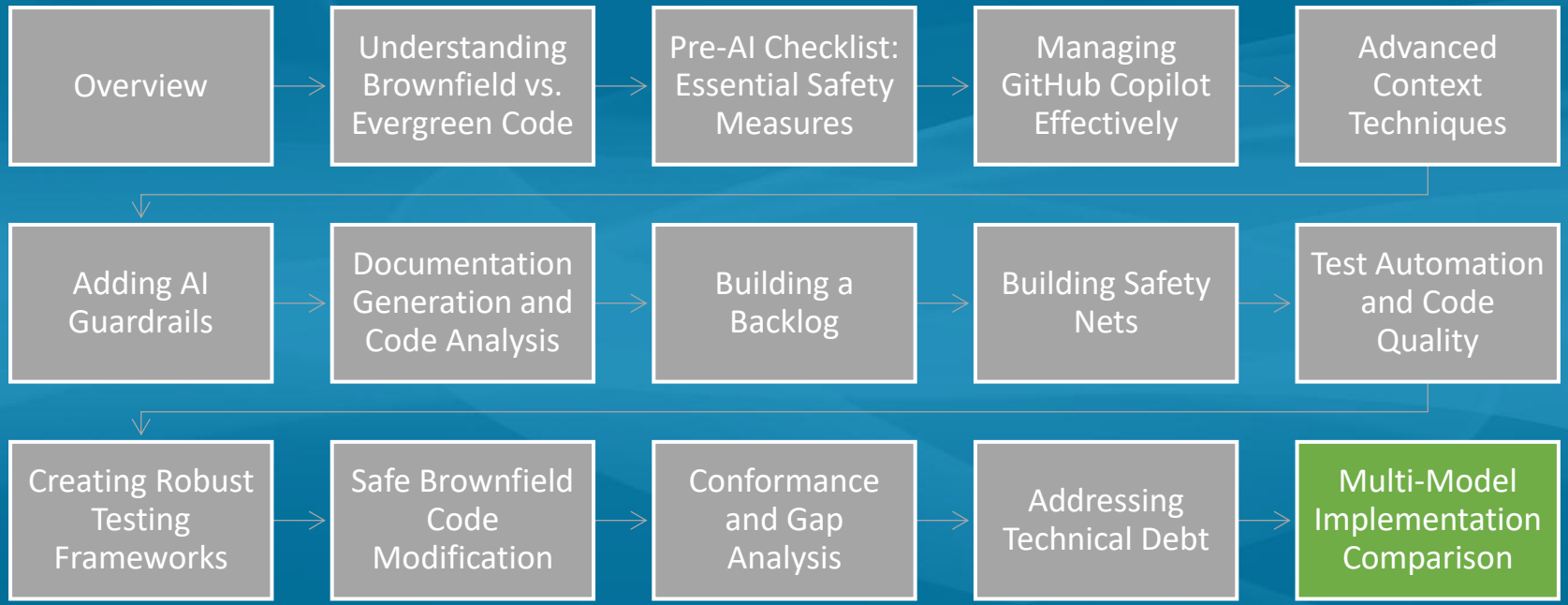
Activities

1. Select a technical debt item.
2. Create a GitHub-style issue with:
   - Title
   - Description
   - Impact and risk
   - Acceptance criteria
   - Provenance metadata
3. Assign the issue to Copilot.
4. Review Copilot's proposed remediation.

Success Criteria

- Issue is clear and well-structured
- Acceptance criteria are testable
- Copilot produces a relevant draft
- Provenance metadata is present

Agenda


Overview
Understanding Brownfield vs. Evergreen Code
Pre-AI Checklist: Essential Safety Measures
Managing GitHub Copilot Effectively
Advanced Context Techniques
Adding AI Guardrails
Documentation Generation and Code Analysis
Building a Backlog
Building Safety Nets
Test Automation and Code Quality
Creating Robust Testing Frameworks
Safe Brownfield Code Modification
Conformance and Gap Analysis
Addressing Technical Debt
Multi-Model Implementation Comparison

Multi-Model Implementation Comparison

- Implementing changes with different AI models
- Comparing approaches and outcomes
- Risk assessment and quality evaluation
- Best practice synthesis
- Exercises for hands-on practice

Implementing Changes With Different AI Models

CODE TRAINING

Why use multiple models?

- Different reasoning styles
- Different strengths (refactoring, documentation, architecture)
- Cross-validation reduces risk
- Helps detect missing context or contradictions

Typical use cases

- Refactoring comparisons
- Documentation consistency checks
- Architecture proposal validation

Comparing Approaches & Outcomes

CODE TRAINING

What to compare

- Code structure and clarity
- Architectural alignment
- Test quality
- Documentation completeness
- Risk level of proposed changes

Benefits

- Identifies the safest implementation
- Surfaces hidden assumptions
- Highlights model-specific biases

Risk Assessment & Quality Evaluation

CODE TRAINING

Risk indicators

- Missing tests
- Large or unnecessary refactors
- Violations of instruction files
- Unclear or undocumented behavior

Quality indicators

- Small, incremental changes
- Clear reasoning
- Strong test coverage
- Alignment with evergreen principles

Best Practice Synthesis

CODE TRAINING

Combine the strengths of each model

- Use one model for architecture
- Another for implementation
- Another for documentation
- Cross-validate tests and reasoning

Outcome

- Higher quality
- Lower risk
- More predictable modernization

Exercise: Prompt Multiple Models to Address Technical Debt

Duration

15 minutes

Objectives

- Compare outputs from different models
- Identify strengths and weaknesses
- Evaluate risk and quality

Activities

1. Select a small technical debt item.
2. Prompt two or more models to propose a fix.
3. Compare outputs for:
   - Safety
   - Clarity
   - Test coverage
   - Architectural alignment
4. Synthesize the best elements into a final solution.

Success Criteria

- Differences between models are clearly identified
- Risks and strengths are evaluated
- Final synthesized solution is safe and incremental
- Provenance metadata is included