Faysal Aziz

Engineering Before Generation

Spec-Based Development for Production Software - 2 days

FA-0188 · 2 days

Agentic & Generative AI · Software Development

Introduction

Turn requirements into systems you can explain, test, secure, operate, and maintain.

Spec-based development becomes valuable only when the people writing and executing the specification understand what software is actually doing underneath it. A detailed specification cannot compensate for weak reasoning about control flow, state, data structures, interfaces, failure modes, dependencies, security boundaries, or deployment. It can simply produce bad software more systematically.

That distinction has become more important as AI-assisted coding moves deeper into mainstream software engineering. Current descriptions of spec-driven development increasingly position the specification as the persistent source of truth connecting requirements, architecture, implementation, and validation rather than treating an AI prompt as the design itself. Microsoft describes the approach as aligning intent before allowing AI to accelerate implementation, while IBM similarly describes the specification as the agreed foundation for what is being built and how it should behave.

This two-day programme therefore approaches spec-based development from the engineering side first. Participants will examine how software moves from a problem statement into algorithms, data structures, components, APIs, processes, services and deployable systems before considering how an AI coding system should be instructed to implement it. The objective is not to turn two days into a compressed computer-science degree. It is to establish the minimum engineering discipline required to recognise whether generated code makes architectural and operational sense.

The course also addresses technology selection. Python, Java, C#, C++ and modern .NET are not interchangeable syntax choices. They have different runtime models, ecosystems, deployment characteristics and areas where they are particularly useful. The same distinction applies to monoliths, services, microservices, background workers and agents. An agent is not automatically an AI agent: many

enterprise agents are deterministic processes responsible for monitoring, orchestration, communication or local execution, while AI-enabled agents introduce models, tools, context, guardrails and additional security boundaries.

One terminology point is worth correcting explicitly. **.NET Framework is legacy technology for many new-development scenarios, but it is not literally dead.** Microsoft continues servicing supported .NET Framework versions, including 4.8/4.8.1, while modern cross-platform development has moved to the unified .NET platform. As of September 2026, .NET 10 is the current LTS release, while .NET 11 is at RC stage ahead of its expected November release. This course therefore distinguishes maintaining legitimate Framework workloads from choosing Framework for new architecture without a defensible reason.

Security is treated in the same way. DevSecOps is not a security scan placed at the end of DevOps. OWASP describes secure development as incorporating security into requirements, design, implementation, verification and operations, with DevSecOps integrating controls such as threat modelling, secrets management, SAST, DAST and software-composition analysis into the delivery process. Participants will therefore learn to put security requirements and trust boundaries into the specification itself rather than asking an AI system to "make it secure" after implementation.

The instructor brings **over 30 years of industry experience** and will teach this as an engineering programme rather than an academic survey. The emphasis will be on the concepts and practices that developers are expected to apply when building real production systems, including knowing when AI-generated code should be rejected regardless of how convincing it appears.

Learning outcomes

By the end of the programme, participants will be able to:

- Explain the relationship between requirements, specification, architecture, implementation, testing and deployment.
- Trace program execution, state changes, data movement and failure paths.
- Apply essential algorithms and data-structure reasoning during software design.
- Decompose software requirements into components, interfaces and implementation responsibilities.
- Write specifications suitable for human developers and AI-assisted development.
- Distinguish specification from prompts, implementation instructions and generated code.
- Evaluate AI-generated code rather than accepting it purely because it compiles.
- Recognise technical debt created by uncontrolled AI-assisted development.
- Select appropriate implementation technologies across Python, Java, C#, modern .NET and C++.
- Distinguish modern .NET from legacy .NET Framework workloads.
- Decide when modular monoliths, services, microservices, background processes or agents are appropriate.
- Distinguish deterministic software agents from AI-enabled agents.
- Understand API-based, on-premises and tool-enabled agent architectures.
- Incorporate security requirements, trust boundaries and validation into specifications.

- Understand the role of CI/CD and DevSecOps in taking specified software into production.
- Maintain traceability from requirement through implementation, test and operational behaviour.

Prerequisites

Participants must have:

- Professional programming experience.
- Ability to write working code without AI assistance.
- Strong understanding of variables, functions, classes and control structures.
- Working knowledge of object-oriented programming.
- Working knowledge of APIs and HTTP.
- Working knowledge of Git.
- Basic SQL knowledge.
- Good experience with posix development tools.
- Ability to read unfamiliar source code.
- Ability to debug a program using logs, breakpoints or equivalent tooling.
- Access to either Codex, Claude Code or Gemini backend (either APi and web based authentication)
- Access to pro version of either ChatGPT, Claude or Gemini

Training outline

1. Software Engineering Foundations for Spec-Based Development

From Business Problem to Executable Software

1. Requirements
2. Constraints
3. Functional and non-functional behaviour
4. Acceptance criteria
5. System boundaries

Understanding Program Execution

1. Control flow
2. Data flow

3. State and mutation
4. Function and method calls
5. Exceptions and failure paths
6. Synchronous and asynchronous execution

Algorithms and Data Structures for Engineering Decisions

1. Complexity and Big-O reasoning
2. Arrays and lists
3. Stacks and queues
4. Hash maps and sets
5. Trees and graphs
6. Searching and sorting
7. Algorithm selection

2. Designing Software Before Generating Code

Software Decomposition

1. Separation of concerns
2. Cohesion and coupling
3. Interfaces and contracts
4. Dependency management
5. SOLID principles

Architecture Fundamentals

1. Layered architecture
2. Modular architecture
3. Domain boundaries
4. Monoliths and modular monoliths
5. Services and microservices
6. Event-driven components

Data and Interface Design

1. Data models
2. API contracts
3. Validation rules
4. Error contracts
5. Versioning
6. Idempotency

3. Spec-Based Development

Specification Structure

1. Problem and objective
2. Scope and exclusions
3. Functional requirements
4. Non-functional requirements
5. Architecture constraints
6. Data requirements
7. Interface contracts
8. Security requirements
9. Acceptance criteria

Requirements Decomposition

1. Features
2. Components
3. Dependencies
4. Implementation tasks
5. Verification criteria

Specification Quality

1. Ambiguity removal
2. Assumption management
3. Constraint definition
4. Edge conditions

5. Failure behaviour
6. Traceability

Specification-Driven AI Development

1. Specification versus prompting
2. Context management
3. Controlled code generation
4. Incremental implementation
5. Generated-code review
6. Specification drift
7. Technical-debt control

4. Programming Language and Platform Selection

Python

1. Automation and orchestration
2. APIs and backend services
3. Data and AI workloads
4. Agent development

Java

1. Enterprise services
2. JVM ecosystem
3. Concurrent applications
4. Microservices

C# and Modern .NET

1. .NET runtime
2. ASP.NET Core
3. Enterprise APIs and services
4. Cross-platform workloads
5. .NET Framework versus modern .NET

C++

1. Native applications
2. Performance-sensitive components
3. Systems integration
4. Lightweight native agents

Technology Selection Criteria

1. Runtime requirements
2. Performance
3. Portability
4. Ecosystem
5. Operational complexity
6. Maintainability

5. Services, Microservices and Software Agents

Service Architecture

1. Process boundaries
2. Service contracts
3. Service communication
4. State ownership
5. Failure isolation

Microservices Engineering

1. Appropriate use cases
2. Distributed-system complexity
3. Service discovery
4. Messaging
5. Observability

Agent Architecture

1. Deterministic agents

2. Monitoring agents
3. Automation agents
4. On-premises agents
5. API-connected agents
6. AI-enabled agents

AI Agent Components

1. Models
2. Instructions
3. Tools
4. Context
5. Memory
6. Guardrails
7. Human control points

Pub Sub Systems for Multi Agent Synchronization & Orchestration

1. Clusters
2. Message queues
3. Transmission and Consumption
4. Guardrails
5. HITL

6. Secure Software Engineering and DevSecOps

Security by Design

1. Assets and trust boundaries
2. Threat modelling
3. Authentication
4. Authorization
5. Secrets
6. Input and output validation

Secure Development Lifecycle

1. Security requirements
2. Secure implementation
3. Dependency management
4. Code review
5. SAST and DAST
6. Software composition analysis

DevSecOps Flow

1. Source control
2. Build
3. Automated testing
4. Security gates
5. Artifact management
6. Deployment
7. Monitoring and feedback

Agent and API Security

1. Least privilege
2. Service identities
3. Credential management
4. Tool permissions
5. Audit trails
6. Untrusted input boundaries

7. Engineering Quality in AI-Assisted Development

Verification of Generated Software

1. Compilation and static analysis
2. Unit testing
3. Integration testing
4. Contract testing
5. Security testing

Code Quality

1. Readability
2. Maintainability
3. Duplication
4. Dependency control
5. Refactoring

Specification-to-Production Traceability

1. Requirement to implementation
2. Implementation to test
3. Test to deployment
4. Production feedback
5. Specification evolution

The increased use of coding agents makes this discipline particularly relevant: current agent tooling emphasises controlled execution, explicit instructions, tool boundaries, guardrails, tracing and evaluation rather than unrestricted code generation.

Disclaimer

This training outline is intended to serve as a structured guideline for delivery rather than an immutable syllabus. The trainer may amend, reorder, expand, reduce or substitute topics where reasonably necessary to reflect participant capability, available training time, technological developments, organisational requirements or instructional priorities, without prior notice.

© 2026 Faysal Aziz · Kuala Lumpur