Reimagine SDLC with AI — Exam Preparation Notes

> **Audience:** Complete beginners to SDLC and AI
> **Purpose:** Quick revision, concept clarity, exam-ready
> **Tool used in course:** GitHub Copilot

Table of Contents

---

1. What is SDLC?

SDLC stands for **Software Development Life Cycle**. Think of it as a step-by-step recipe for building software. Just like cooking a meal needs planning, buying ingredients, cooking, and tasting — software needs a proper process too.

Formally, SDLC is a **structured framework** that defines how to turn a business idea (requirement) into working software through a series of well-defined phases.

- It guides a team from the very first idea of a project all the way to its release and maintenance.
- It defines **what to do**, **who does it**, and **what the result should be** in each stage.
- There are many types of SDLC models (e.g., Waterfall, Agile, Scrum), but all share similar phases.

> **Simple analogy:** Building a house. You need to plan (what rooms?), design (where do the doors go?), construct (lay bricks), inspect (is it safe?), hand over the keys (deploy), and do repairs when needed (maintain). That is the life cycle of a house — SDLC is the same idea for software.

---

2. Phases of SDLC

There are **six main phases** in SDLC. Each has specific goals, activities, and outputs (called **deliverables**).

Phase 1: Planning & Requirement Analysis

The very first step. Before writing a single line of code, we must understand **what** we are building and **why**.

Goal: Understand what the business needs, define the project scope, and check if the project is feasible.

Key Activities:

- Talk to stakeholders (business users, managers, etc.) through interviews, workshops, and surveys.
- Document two types of requirements:
  - **Functional Requirements** — What the system *must do*. Example: "Users must be able to submit a loan application."
  - **Non-Functional Requirements** — *How well* the system must perform. Example: "The checkout page must load in under 2 seconds."
- Check if the project is technically and economically possible (**feasibility study**).
- Plan the timeline, budget, and resources.
- Identify and plan for risks.

Deliverables:

- Requirements specification document
- Project charter
- Risk assessment matrix
- Resource allocation plan

Phase 2: System Design

Now we take the requirements and design **how the system will be built**.

Goal: Create a technical blueprint of the system.

Key Activities:

- Design the overall system structure (**system architecture**).
- Create database designs (how data is stored).
- Design screens and user flows (**UI/UX**).
- Define how different parts of the system communicate (**APIs**).
- Specify security and performance needs.

Deliverables:

- System design document
- Architecture diagrams
- Database design
- API specification
- UI mockups and wireframes

Phase 3: Implementation (Coding)

This is where engineers actually **write the code**. The design from phase 2 is turned into working software.

Key Activities:

- Write frontend code (the part users see).
- Write backend code (the logic happening behind the scenes).
- Build the database.
- Integrate with third-party services (e.g., payment gateways, WhatsApp messaging).
- Write unit tests to check individual pieces of the code.
- Conduct code reviews.

Deliverables:

- Source code
- Database scripts
- Integration modules
- Code review reports

Phase 4: Testing

The software is tested to make sure it **does what it is supposed to do** and handles unexpected situations correctly.

Key Activities:

- Write and execute test cases (detailed tests).
- Perform unit testing, integration testing, and full system testing.
- Run **User Acceptance Testing (UAT)** — the business users try the software.
- Log and fix bugs (errors in the software).
- Performance testing — does the software handle high usage?

Deliverables:

- Test plans and test cases
- Bug reports
- Performance test results
- Quality assurance certification

Phase 5: Deployment

The software is **released to real users** in the production environment (the live system).

Key Activities:

- Prepare the production environment (the server/platform where the software will live).
- Deploy (install/release) the software.
- Verify that the software works correctly in production.
- Train end users.
- Write user guides and admin manuals.

Deliverables:

- Deployed application
- Deployment guides
- User training materials
- Go-live verification reports

Phase 6: Maintenance

After release, the software needs **continuous care**. Updates, bug fixes, and improvements keep the software healthy.

Key Activities:

- Monitor system performance.
- Fix user-reported issues.
- Apply security patches.
- Add new features (which may start a new SDLC cycle).
- Plan system upgrades.

Deliverables:

- Maintenance reports
- Performance monitoring dashboards
- Security update documents
- Enhancement reports

---

3. Why SDLC Matters

Without SDLC, software projects can fail. Here are three real problems that SDLC prevents:

Problem	What goes wrong without SDLC	How SDLC fixes it
Building the wrong thing	Engineers build a video calling app, but the business needed a chat app	Requirements phase ensures the right product is built
Conflicting requirements	Finance team wants complex tax logic, legal team wants compliance tracking — engineers get two different stories	SDLC structures stakeholder communication and resolves conflicts before coding
Budget overruns	Hidden complexities (like credit bureau integration) are discovered during coding, adding weeks of cost	Design phase identifies all requirements upfront

> **Key point:** Coding is just one part of SDLC. Software development is about solving business problems, managing people, ensuring quality, and delivering value — not just writing code.

---

4. AI Concepts: Agent, Agentic AI, Non-Deterministic

This is one of the most important sections for your exam. Several AI terms appear frequently.

AI Agent

An **AI agent** is a system that performs a *specific, well-defined task* based on given instructions.

- It works within boundaries set by its design.
- Think of it as a specialized robot — it does its job well, but nothing more.

Examples:

- A customer service chatbot — designed only to answer customer questions.
- A coding AI — designed only to write or explain code.

Agentic AI

Agentic AI is a more advanced form of AI that can:

- **Break down complex goals** into smaller steps on its own.
- **Plan and execute** those steps autonomously (without constant human instruction).
- **Coordinate multiple AI agents** or tools to achieve the final goal.

> **Simple analogy:** If a regular AI agent is a worker who does one task, **agentic AI is the project manager** — it understands the big goal, creates a plan, assigns tasks to the right workers (other AI agents or tools), monitors progress, and handles problems.

Example: Instead of just "create a chart from sales data" (a simple AI agent), an agentic AI can handle: "Analyze this data, find top-performing products, build charts, write a summary, and produce a complete report" — all from a single instruction.

How it works step by step:

1. Understands the high-level goal from natural language.
2. Creates a multi-step plan.
3. Delegates each step to specialized tools or agents.
4. Monitors execution — if something fails, it adapts.
5. Combines all results into a final output.

Non-Deterministic AI

This is a crucial concept to understand when working with AI.

Non-deterministic means that if you give the AI the **same instruction twice**, it may give you **two different answers**.

- AI does not work like a calculator (where 2+2 always gives 4).
- Each response is influenced by patterns, randomness, and context — so results vary.

> **Why it matters:** Even if you copy the exact same prompt used in a course, your result from GitHub Copilot may look different. This is normal and expected.

Practical impact:

- AI-generated documents, code, or test cases may differ across two runs.
- Always review AI output — never assume it is perfect.
- Use custom instructions and standards to guide AI toward more consistent results.

AI is Not a Magic Wand

While AI is powerful, it has clear limitations:

- **Cannot replace experienced engineers** — AI lacks domain expertise and full business context.
- **Cannot act beyond its design** — a coding AI cannot schedule a meeting; a communication AI cannot write code.
- **May generate incorrect, insecure, or non-compliant output** — e.g., AI-generated code might accidentally expose private user data (PII data), violating regulations.
- **Human accountability remains** — engineers are still responsible for reviewing and validating AI output.

> **Analogy:** AI is like a **power drill**. You can build a shelf faster with it, but a human still controls it. Relying entirely on the power drill — without knowing how to use it — can cause damage.

---

5. GitHub Copilot Overview

GitHub Copilot is the AI tool used throughout this course. It is a product by GitHub (owned by Microsoft) and is powered by AI.

Its evolution:

Stage	What it does
Code completion tool	Suggests code as you type
Chat assistant	You ask questions; Copilot answers
Agentic AI (Copilot agent)	Works autonomously on tasks in the background, like a team member

What GitHub Copilot can do in SDLC:

- Generate interview questions for requirement gathering.
- Create Product Requirement Documents (PRDs).
- Write Technical Requirement Documents (TRDs).
- Write and explain code.
- Create unit tests and performance testing scripts.
- Generate deployment configuration files (Dockerfiles, Kubernetes manifests).
- Review pull requests and explain code changes.

What GitHub Copilot cannot do:

- Schedule meetings or send emails.
- Deploy applications to cloud servers directly.
- Guarantee production-ready, secure, fully correct output.
- Understand your organization's unique culture or unstated rules.

Important technical note: GitHub Copilot, by default, can only access the repository it is assigned to. To let it access other repositories, you must set up a **Personal Access Token**.

---

6. AI Prompts — Talking to AI Effectively

What is a Prompt?

A **prompt** is the instruction or question you give to an AI. It is how you communicate with AI to get a result.

> **Simple analogy:** Telling someone to cook dinner. "Make food" is too vague and unhelpful. "Please make pasta for 4 people, something quick and with beef" gives just enough detail for a good result. That specific instruction is a good **prompt**.

Good Prompt vs. Bad Prompt

Type	Example	Problem
Vague / Bad prompt	"Make me a financial app."	Too broad — financial could mean loans, stocks, or payments. AI will guess.
Good prompt	"Create a REST API for loan applications using Go and PostgreSQL, following the design in the TRD document at [link]."	Specific, technical, and pointed — AI knows exactly what to build.

Rules for Writing Good Prompts

- **Be specific** — describe exactly what you need, not just a general idea.
- **Avoid repetition** — repeating the same instruction in a single prompt confuses the AI.
- **Provide context** — mention the technology, documents, standards, or examples that are relevant.
- **Use technical terms** — GitHub Copilot understands software terminology (REST API, PostgreSQL, etc.), so use them freely; no need to over-explain basics.
- **Treat Copilot like an inexperienced engineer** — give clear, written specifications the same way you would brief a new hire.

Key insight: The better your prompt, the better the AI's response. Clear communication with AI is a skill worth developing.

---

7. Phase 1: Planning & Requirement Analysis

How AI Helps with Requirement Gathering

When the business team prepares for interviews with stakeholders, they can use GitHub Copilot to generate **preliminary interview questions** tailored to each role (e.g., marketing manager, accountant, car manager).

- Access GitHub Copilot directly from the web browser at `github.com/copilot` — no coding tool needed.

- A **GitHub Copilot space** can be created as a dedicated workspace for a project, containing all related documents, images, and conversations for better AI context.

Product Requirement Document (PRD)

A **PRD** (Product Requirement Document) is a written document that explains how the system should behave for a specific area of the product — from the user or business perspective.

Other names for this: Functional Requirement Document, System Requirement Specification (SRS).

What a PRD typically includes:

- Business goals and context
- List of features and expected behavior
- User roles and their workflows
- Compliance and regulatory requirements

AI's role: GitHub Copilot can generate a draft PRD from interview results. However:

- Always review the output — AI does not know your organization's specific rules.
- AI output is non-deterministic, so two PRDs generated from similar instructions may look different.
- Custom instructions in the repository help enforce a standard format.

GitHub Projects — Managing Work

GitHub Projects is a task management board (similar to Trello or Jira). In SDLC with AI, it tracks the status of all tasks.

- A **Kanban board** is a common format: tasks move across columns — Backlog → Ready → In Progress → In Review → Done.
- Each task is called an **issue** in GitHub.
- You can assign issues to GitHub Copilot, and it will autonomously work on them.

What is an issue? An issue is a work item — it could be "Create a PRD for marketing" or "Build the login API." It must:

- Have a clear description of what needs to be done.
- Include acceptance criteria (what does "done" look like?).
- Be small enough to complete within 2–3 days.

What is a pull request? After completing work on a branch (a separate copy of the code/files), a **pull request** is a proposal to merge those changes into the main branch. It is a safe checkpoint — changes are reviewed before being officially added.

GitHub Copilot: Smart but Disconnected

One of the most important concepts for the exam:

Every time you assign an issue to GitHub Copilot, it behaves like a **brand new team member** who:

- Has no memory of previous tasks.
- Does not automatically know what other Copilot instances have worked on.
- Will only be aware of context if you **explicitly tell it** — through links, file references, or issue descriptions.

Practical example:

- Copilot writes a PRD for "Car Management" and creates a `vehicles` table.
- You assign another issue to Copilot for a different feature.
- The second Copilot creates its own `vehicles` table — possibly with different column names!
- Solution: In the new issue, explicitly reference the existing table file from the first PR.

> **Key exam point:** AI output from one task does not automatically feed into the next. Context must be provided manually and explicitly every time.

Custom Instructions — Enforcing Standards

Organizations have standards for everything — document formats, code style, naming conventions. Since AI is non-deterministic, how do we ensure consistent output?

The answer is **custom instructions** in the repository.

- Create a file `.github/copilot-instructions.md` in the root of the repository — this applies to all tasks in that repo.
- Create `.github/instructions/*.instructions.md` files for specific folders — e.g., a custom format only for the PRD folder.
- Instructions are written in plain English and Copilot reads them before working.

What to put in instructions:

- Project context and goals.
- Folder structure rules.
- Document format requirements.
- Coding language and framework requirements.

Important limitation: Instructions have a character limit, so keep them focused and concise.

---

8. Phase 2: System Design

Technical Requirement Document (TRD)

Once the PRD (what the system should do) is approved, the next step is to create a **TRD** — Technical Requirement Document — which explains **how the system will technically achieve those requirements**.

Document	Answers	Written by
PRD	*What* should the system do?	Product owner / Business analyst
TRD	*How* will the system do it?	System architect / Tech lead

Typical TRD contents:

- System architecture (how components are structured).
- **API specification** — the contract for how different parts of the system communicate.
- Database design — table names, columns, data types.
- Error handling rules.
- Security and performance requirements.
- UML diagrams (e.g., sequence diagrams showing the flow of actions).

API (Application Programming Interface) — A defined way for two software systems to talk to each other. Example: When you order food on an app, the app "calls" an API on the restaurant's server to place the order.

Creating a TRD with GitHub Copilot

- Break the PRD down into individual features.
- Create one issue per feature for TRD creation.
- Provide a link to the relevant section of the PRD in each issue.
- Assign the issue to Copilot.

The disconnected problem again: If multiple TRD issues are assigned to Copilot, each Copilot instance may define the same shared database table differently. Solution: Consolidate the common table first, then reference it explicitly in all subsequent issues.

Context-aware tasks: To make Copilot aware of existing documents:

- Paste the direct link to the specific file/section in the issue description or pull request comment.
- Copilot can then read that document and build upon it.

---

9. Phase 3: Implementation (Coding)

What Happens in the Implementation Phase

Engineers take the TRD and write actual code. This phase touches:

- **Frontend** — the part users interact with (buttons, screens, forms).
- **Backend** — the logic behind the scenes (processing data, database operations).
- **Database** — where data is stored.
- **Integrations** — connecting to external services.
- **Unit testing** — writing small tests to verify individual functions work correctly.

Important: The "deployment" in this phase is only to a **development environment** (a test server), not to real users.

Source Code Standards

Every organization has a **coding standard** — rules about which language to use, how to structure files, naming conventions, and more. This ensures:

- New team members can join quickly and understand the codebase.
- The team spends time building, not arguing about tools.

For GitHub Copilot, the coding standard must be written in a markdown file (`.github/copilot-instructions.md`) so Copilot can read and follow it.

> **Think of it like this:** A new intern needs to be shown how the team works. GitHub Copilot is that intern — it learns from written documentation, not by "watching" colleagues work.

Model Context Protocol (MCP)

MCP — Model Context Protocol — is one of the most important new AI concepts in this course.

What is it? MCP is a **standard communication bridge** that allows GitHub Copilot to connect to and use external tools — tools that are *not* part of GitHub by default.

> **Simple analogy:** Copilot is a project manager. MCP is the "phone" the project manager uses to call specialized team members — like a Figma designer, a Postman tester, or a Jira project tracker.

Why do we need MCP?

- By default, GitHub Copilot only knows about the current repository.
- With MCP, Copilot can access Figma designs, Postman collections, Atlassian, Notion, and more.

How it works:

- An **MCP server** acts as the bridge between Copilot and the external tool.
- MCP servers can be installed in Visual Studio Code (from the Extensions menu).
- Some MCP servers are pre-built into the GitHub web interface (like GitHub MCP and Playwright).

Examples in the course:

- **GitHub MCP** — lets Copilot access issues, pull requests, and GitHub resources.
- **Figma MCP** — lets Copilot read UI designs from Figma and generate corresponding frontend code.
- **Postman MCP** — lets Copilot read API collections from Postman and generate performance test scripts.

GitHub Copilot Agent Mode in VS Code

When you use GitHub Copilot from inside **Visual Studio Code** (VS Code), it works in two modes:

Mode	Description	Analogy
Web (assign issue)	You assign the task; Copilot works independently on its own server	You brief an employee; they work alone at their desk
VS Code Agent Mode	You sit with Copilot while it works on *your* machine	You work side-by-side with a colleague on your laptop

VS Code Agent Mode advantages:

- Copilot can see your local environment settings and files.
- Copilot can run terminal commands (with your permission).
- Can use multiple MCP servers simultaneously.

VS Code Agent Mode disadvantages:

- Uses significantly more **premium requests** (quota).

GitHub Copilot Firewall

GitHub Copilot's internet access is controlled by a **security firewall** to prevent sensitive data leaks.

- If Copilot tries to access a blocked URL, it will warn you in the session log.
- You can add trusted URLs to a **custom allowlist** in the repository settings under Copilot → Coding Agent.

Merge Conflicts

A **merge conflict** happens when two separate branches of code change the same part of a file, and Git cannot automatically decide which version to keep.

- This commonly happens when two Copilot instances work on related features in parallel.
- **Merge conflicts must be resolved by a human** — AI cannot reliably resolve them on its own, because each Copilot instance is unaware of what the other did.

---

10. Phase 4: Testing

What is Testing?

Testing verifies that the software **does what the requirements said it should do**. If it doesn't, that gap is called a **bug**.

> **Simple example:** A calculator should compute 3 + 5 = 8. If it shows 9, that's a bug.

Test Scenarios vs. Test Cases

These two terms are closely related but different:

Term	What it is	Example
Test Scenario	High-level description of *what* to test	"Verify the calculator performs addition correctly"
Test Case	Specific, step-by-step test with input and expected result	"Input: 3 + 5 → Expected output: 8"

One scenario can have many test cases.

Positive and Negative Testing

Type	Purpose	Example
Positive test case	Verifies the system works correctly with valid inputs	Enter 10 + 7, expect 17
Negative test case	Verifies the system correctly rejects invalid inputs	Enter -5 (invalid for 0–99 range), expect an error message

Both types are necessary. Together, they confirm the system handles both normal and unexpected situations correctly.

Performance Testing

Performance testing checks how well the software performs under heavy use.

- Validates **non-functional requirements** like "the system must handle 100 operations per second."
- If the system only handles 70 operations per second, that is a **performance bug**.
- Tools used: **K6**, Artillery, JMeter.

GitHub Copilot's role in testing:

- Generates test scenarios and test cases from PRDs (functional testing).
- Generates performance test scripts (e.g., K6 scripts) based on API specifications or Postman collections.
- **Cannot run the scripts itself** — you still need to execute them in your own environment.

---

11. Phase 5: Deployment

What is Deployment?

Deployment means releasing the software so that real users can start using it.

Modern software often follows a **microservice architecture** — the application is split into many small, independently deployable services.

Key Deployment Concepts

Container

A **container** packages an application along with everything it needs to run — the code, libraries, settings, and runtime. It is a self-contained, portable unit.

> **Analogy:** A shipping container holds goods and can be transported by any ship, truck, or train. Similarly, a software container holds an app and can run on any system that supports containers.

- To create a container, you write a **Dockerfile** — a text file with instructions for building the container.
- **Docker** is the most popular tool for building and running containers.

Kubernetes

Kubernetes is a platform that manages many containers running together — it is called a **container orchestration platform**.

- When you have dozens or hundreds of microservices, Kubernetes handles starting, stopping, scaling, and monitoring them.
- Deploying to Kubernetes requires a **manifest file** — a text file defining what the application needs (e.g., how many copies to run).
- **Helm** is a package manager for Kubernetes that bundles multiple manifest files into a reusable package called a **Helm chart**.

Infrastructure as Code (IaC)

Instead of clicking buttons in a cloud provider's website to set up servers, **Infrastructure as Code** means writing scripts (using tools like Terraform or OpenTofu) that automatically create and configure the cloud infrastructure.

- Repeatable, consistent, and reduces human error.
- GitHub Copilot can help generate these scripts.

CI/CD — Continuous Integration / Continuous Deployment

CI/CD is a practice where every code change is automatically built, tested, and deployed — reducing manual steps and human errors.

- **Continuous Integration (CI):** Every code change is automatically tested.
- **Continuous Deployment (CD):** Passing changes are automatically deployed.
- **GitHub Actions** is the tool used to define and run these automated workflows.

GitHub Copilot can generate GitHub Actions scripts (YAML files) to automate the deployment process.

GitHub Copilot's Role in Deployment

- Generate **Dockerfiles** to containerize backend and frontend applications.
- Generate **Kubernetes deployment manifests**.
- Generate **Helm charts**.
- Generate **GitHub Actions** deployment workflows.
- Cannot deploy the application by itself — it helps you write the scripts; you run them.

---

12. Phase 6: Maintenance

After deployment, the software enters the maintenance phase. The goal is to keep the software running well, fix issues, and improve it over time.

Key activities:

- Monitor performance.
- Fix bugs reported by users.
- Apply security patches.
- Add new features or improvements — which may trigger a new SDLC cycle.

GitHub Copilot's role in maintenance:

- Still in its early stages for deployment and maintenance compared to earlier SDLC phases.
- Can assist with bug fixes, code updates, and documentation.

---

13. GitHub Copilot Best Practices & Constraints

Best Practices for Using GitHub Copilot

1. **Always maintain healthy skepticism** — Copilot output is non-deterministic and may not be perfect. Always review, validate, and adjust.
2. **Provide explicit context** — Copilot has no memory between tasks. Link to relevant files, PRDs, or TRDs in every issue.
3. **Use custom instructions** — Define organization standards in `.github/copilot-instructions.md` files. This guides Copilot toward consistent output.
4. **Write clear, detailed issues** — The issue description is Copilot's prompt. Vague issues produce vague or incorrect results.
5. **Start small** — Identify a few low-risk use cases when introducing Copilot. Avoid starting with critical or highly complex systems.
6. **Humans make final decisions** — AI-generated artifacts (PRDs, code, test cases) are starting points, not finished products.

Tasks Well-Suited for GitHub Copilot

- Drafting requirement or technical documents.
- Writing or improving code for well-defined features.
- Fixing specific bugs.
- Updating documentation.
- Creating unit tests.
- Generating performance test scripts.

Tasks Better Done by Humans

- **Complex, multi-repository tasks** that need deep integration awareness.
- **Security-sensitive work** — authentication, personal data handling.
- **Ambiguous tasks** where the requirement is unclear or open-ended.

- **Learning-oriented tasks** where the engineer needs to develop skills.
- **Production incidents** that require fast, reliable, accountable decisions.

Cost Constraints

- GitHub Copilot requires a **paid subscription** for advanced features (assigning issues, using premium AI models).
- Each interaction with Copilot (chat or assigned issue) consumes **premium request quota**.
- Working sessions also consume **GitHub Action minutes quota**.
- If quotas run out, some Copilot features become unavailable until the quota resets.
- Using Copilot Chat from VS Code consumes **significantly more** premium requests than using the GitHub web interface.

---

14. Key AI Terms Quick Reference

This section lists all important AI and SDLC terms you are likely to be examined on, with simple explanations.

Term	Simple Explanation
SDLC	Step-by-step process for building software, from idea to launch and maintenance
AI Agent	An AI that performs one specific task (e.g., answer questions, write code)
Agentic AI	Advanced AI that plans, breaks down complex goals, and coordinates multiple tools or agents to complete them
Non-Deterministic AI	AI that may give different answers to the same prompt at different times
Prompt	The instruction or question you give to an AI
GitHub Copilot	AI tool by GitHub that assists across the entire SDLC
GitHub Copilot Agent Mode	Mode where Copilot works autonomously on assigned tasks
MCP (Model Context Protocol)	A standard bridge that lets GitHub Copilot connect to and use external tools (Figma, Postman, etc.)
MCP Server	The bridge/connector between Copilot and an external tool
Functional Requirements	Requirements that define *what* the software must do
Non-Functional Requirements	Requirements that define *how well* the software must perform (speed, security, reliability)
PRD	Product Requirement Document — business-focused description of what the system should do

Term	Simple Explanation
TRD	Technical Requirement Document — engineer-focused description of *how* to build the system
API	A defined way for two software systems to communicate with each other
Issue (GitHub)	A task or work item tracked in GitHub Projects
Pull Request	A proposal to merge code/file changes from one branch into another
Branch	A separate, isolated copy of code/files for developing a feature without touching the main version
Merge Conflict	When two branches change the same part of a file and Git cannot auto-resolve which to keep
Container	A self-contained package of an application and everything it needs to run
Docker	Tool for building and running containers
Dockerfile	A text file with instructions to build a Docker container
Kubernetes	Platform that manages and orchestrates many running containers
Helm	Package manager for Kubernetes; bundles multiple manifests into a Helm chart
CI/CD	Continuous Integration / Continuous Deployment — automation of building, testing, and deploying code
GitHub Actions	Tool for defining and running automated CI/CD workflows
Infrastructure as Code (IaC)	Using scripts (not manual clicks) to set up and manage cloud infrastructure
Unit Test	A small test that checks one specific function or component of the code
Test Scenario	High-level description of what to test
Test Case	Specific step-by-step test with inputs and expected results
Positive Test Case	Test with valid inputs to verify correct behavior
Negative Test Case	Test with invalid inputs to verify correct error handling
Performance Testing	Testing how well the software performs under heavy load
K6	A script-based performance testing tool
Healthy Skepticism	The practice of always reviewing and questioning AI-generated output before accepting it
GitHub Copilot Space	A dedicated Copilot workspace with curated context (files, docs, instructions) for a specific project
Custom Instructions	Files in `.github/` folder that guide Copilot to follow organization standards

Term	Simple Explanation
Premium Request	A paid usage unit consumed each time you interact with advanced Copilot features
Kanban Board	A visual board for tracking tasks through columns: To Do → In Progress → Done
Sprint	A short work cycle (typically 2 weeks) in Agile methodology where a team completes a set of tasks
Backlog	A list of all tasks waiting to be worked on
Artifact	Any document or file produced during SDLC (PRD, code, test plan, deployment script)
PII	Personally Identifiable Information — private user data that must be protected by law
Feasibility Study	An assessment of whether a project is technically and economically possible
UAT (User Acceptance Testing)	Testing done by actual business users to verify the software matches their needs
Microservice Architecture	Designing software as many small, independent services that work together

Final Summary

- **SDLC** gives software development a structured, repeatable process with clear phases.
- **AI (especially Agentic AI)** can assist at nearly every SDLC phase — but human judgment remains essential.
- **GitHub Copilot** is an AI tool that works as an autonomous, intelligent assistant throughout the SDLC.
- **Good prompts** are the key to getting useful AI output — be specific, provide context, and use technical language.
- **AI is non-deterministic** — always review and validate what it produces.
- **Custom instructions** + **explicit context** in issues are the two most effective ways to improve Copilot output consistency.
- **MCP** extends Copilot's capabilities beyond GitHub to tools like Figma and Postman.
- **Healthy skepticism toward AI output is not optional — it is a professional responsibility.**

Notes compiled from course: "Reimagine Software Development Life Cycle (SDLC) with AI"
Sections covered: 1 through 8 — Welcome, Intro to SDLC & AI, Planning, System Design, Implementation, Testing, Deployment, Maintenance