AI for Developers: GitHub Copilot & Cursor — Complete Study Notes

> **Audience:** Complete beginners to AI tools. No prior AI knowledge assumed. **Purpose:** Exam-ready, concept-focused, beginner-friendly revision notes.

Table of Contents

---

1. Getting Started: AI Tools Overview

1.1 What is AI in a Code Editor?

Think of a normal text editor where you type code. Now imagine a very smart assistant sitting right inside that editor who can:

- Predict what you are about to type and suggest it automatically
- Write entire sections of code based on a short description you give it
- Answer your questions about your code without you leaving the editor

- Take on complete tasks — like adding a new feature — and work through them on its own

That is exactly what tools like **GitHub Copilot** and **Cursor** do. They bring AI directly into your coding environment so you can write code faster, understand code better, and automate repetitive work.

These tools do **not** replace the developer — they assist the developer. You stay in control. You review, accept, or reject every suggestion.

> **Important:** This course is for developers who already know how to write code. AI tools are most powerful when combined with your existing programming knowledge — not as a replacement for it.

1.2 Categories of AI Developer Tools

Not all AI tools for developers are the same. They fall into four main categories:

1. External AI Chatbots

Examples: ChatGPT, Google Gemini

- You chat with them through a website or app — they are not inside your editor
- Great for general coding questions, brainstorming, and planning new projects
- **Limitation:** They do not have access to your actual code unless you copy and paste it in — which is impractical for large projects
- Best when the task does not require deep knowledge of your specific codebase

2. AI-Powered Code Completion

Examples: GitHub Copilot, Cursor

- These tools live inside your code editor
- They are **context-aware** — they understand the file you are working on, the surrounding code, and recently edited files
- They suggest what you might want to type next, line by line, as you write
- Much smarter than basic autocomplete — they understand intent, not just syntax

3. Coding Agents ⭐

Examples: GitHub Copilot (Agent Mode), Cursor (Agent Mode), Claude Code

- The most powerful category
- These agents can **generate entire files**, work across multiple files, and autonomously implement complete features
- They understand your whole codebase and can be given extra context and instructions
- Unlike chatbots, they act inside your project — reading, editing, and creating files
- As a developer, learning to **control and direct these agents** is the most important skill to develop

4. Code Review Tools

Examples: Bugbot (by Cursor), CodeRabbit

- These tools run in the cloud and automatically review **Pull Requests** (suggested code changes) on platforms like GitHub
- They act as an extra pair of AI eyes on every code change before it gets merged
- They can catch bugs, suggest improvements, and provide a baseline for human reviewers
- Not covered in depth in this course, but important to know they exist

> **Summary:** Chatbots are for general Q&A. Completion tools help as you type. Agents handle entire features. Review tools check your code automatically.

1.3 Important Limitations to Know

Before diving into these tools, there are key limitations every developer should understand:

Cost

- Almost all serious AI developer tools require a **paid subscription**
- Free tiers exist but are very limited — they work for a few examples but not for real daily use
- Running AI models is expensive for the providers, so they charge for it
- **GitHub Copilot** requires a Pro or Pro+ plan for meaningful use
- **Cursor** requires at least a Pro plan; heavy agent usage may need Pro+ or Ultra

Non-Determinism (Results Vary)

- AI is **not deterministic** — this means running the exact same prompt twice will likely produce slightly different results each time
- The same feature you ask the AI to build today may be built slightly differently tomorrow
- This is by design — there is randomness built into how AI models generate text
- **Practical impact:** Do not assume the AI will always do the same thing. Always review what it produces.

Knowledge Cutoff

- AI models are trained on data up to a certain date — called the **knowledge cutoff**
- After that date, they are unaware of new library versions, new APIs, or new tools
- When working with recently released technology, always provide the AI with up-to-date documentation yourself

Results Will Differ From Examples

- When following along with course videos or tutorials, the AI results you get will likely not match what is shown
- This is normal and expected — model updates, random variation, and your project's different context all contribute
- Focus on understanding the **approach and techniques**, not the exact output

---

2. GitHub Copilot — Fundamentals

What is GitHub Copilot?

GitHub Copilot is an AI-powered extension (add-on) that plugs into code editors like **Visual Studio Code** and adds smart, AI-driven features. Think of it as a highly capable coding assistant that lives inside your editor.

Key capabilities:

- **AI code completion** — suggests code as you type
- **Code generation** — writes entire blocks of code for you
- **Built-in AI chat** — you can ask it questions or give it tasks
- **AI agents** — it can autonomously work on your project

Subscription: Copilot is mostly a paid tool. There is a free tier but it is very limited. Most developers need the **Pro** or **Pro+** plan. You must be signed in with a GitHub account to use it.

2.1 AI-Powered Code Completions

When you start typing in your editor, Copilot quietly watches and offers **suggestions** in faded text. You have not accepted them yet — you choose whether to use them or not.

Accepting a suggestion:

- Press **Tab** to accept the full suggestion
- Press **Cmd/Ctrl + →** to accept one word at a time (partial accept)
- Press **Escape** to dismiss the suggestion

Hovering over suggestions: When Copilot offers a suggestion, you can hover over it to see **multiple alternative versions** of the same suggestion. This lets you pick the best one instead of blindly accepting the first one.

Important terms:

- **AI Code Completion** — Copilot predicts what code you want to write and shows it to you before you finish typing. Example: you type `name =` and Copilot suggests `name = input("Enter your name: ")`.
- **Next Edit Suggestions** — A smarter feature where Copilot suggests a change **in a different line** based on an edit you just made. For example, if you rename a function, it suggests updating all the places where that function is called. You can turn this on or off in the Copilot toolbar.
- **Comment-based prompting** — You can write a comment like `# Function that combines two strings` and Copilot will suggest the actual code for that function. Comments act like instructions to the AI.

2.2 Chat Modes: Ask, Edit, Plan, Agent

GitHub Copilot's chat window (accessible from the sidebar in VS Code) offers several **modes**. Each mode has a different purpose:

Ask Mode

- Works like **ChatGPT inside your editor**
- You can ask questions: "What does this function do?" or "How do I add error handling here?"
- It **does not** write or change any code — it only answers questions and explains things
- It is aware of your project's files, so it gives context-aware answers
- **Best for:** Planning, exploring ideas, understanding code

Edit Mode

- The AI can make **targeted edits** to specific files or sections you point at
- Best for **small, focused changes** in a single file
- May be removed in future versions as Agent mode has become more powerful

Plan Mode

- Think of this as a **prompt improvement assistant**
- You give it a task, and instead of jumping straight into coding, it first **analyzes your request**, may ask you clarifying questions, and generates a **step-by-step plan**
- You review the plan and then tell it to proceed
- The AI **cannot edit files** in Plan mode — it only plans
- **Best for:** Complex features where you want to think through the approach before coding

Agent Mode ⭐ (Most Important)

- This is the **most powerful mode**
- The agent can: create files, edit files, create folders, run terminal commands, and work across multiple files — all on its own
- You give it a task and it works autonomously, asking for your permission before doing anything sensitive (like running a terminal command)
- You always get a **preview of changes** (green = added, red = removed) before you accept
- **Best for:** Complete feature implementation, working across the whole codebase

> **Simple Analogy:** Ask mode is like asking a colleague for advice. Plan mode is like asking them to write a project plan. Agent mode is like handing them the keyboard and saying "Go build it."

2.3 AI Agents in GitHub Copilot

An **AI Agent** is an AI system that can take on multi-step tasks by itself, making decisions and using tools along the way, without needing you to guide every single step.

How Agent Mode works:

1. You describe a task in plain English
2. The agent reads your files to understand the project
3. It writes or edits code, creates files, and may run commands
4. It asks for your permission before any sensitive action (e.g., running a terminal command)
5. You review the changes and accept or reject them

Controlling Permissions

Agent mode asks before doing things that could be risky:

- **Allow once** — allows the action just this one time, in this session
- **Allow in workspace** — allows it for this project going forward
- **Always allow** — allows it everywhere, forever
- **Skip / Deny** — blocks the action; you can write a short reason

> This is a safety mechanism to prevent the AI from accidentally deleting files or breaking your project.

Custom Agents

You can create your own specialized agents, for example:

- A **Research Agent** that searches the web and reads docs
- A **Code Review Agent** that checks code quality

Each custom agent has:

- A name and description
- A prompt that tells it what to do
- A specific set of **tools** it is allowed to use (e.g., read files, search the web)

Combining Agents

You can chain agents together. For example:

1. Use a **Research Agent** to find best practices from official docs
2. Switch to **Agent Mode** and say: "Fix the issues identified by the research"

2.4 Prompt & Context Engineering

Prompt Engineering is the skill of writing good instructions (called **prompts**) to get the best results from an AI. It is one of the most important skills when working with AI tools.

Think of it this way: if you give a new colleague vague instructions, they will produce vague results. The same is true for AI.

Tips for writing good prompts:

- **Be specific** — "Add error handling to the `write_file` function in `main.py`" is better than "Improve the code"
- **Provide context** — tell the AI which files are relevant, what technology you are using, what the expected outcome is
- **Give examples** — point the AI at an existing function or file that shows the pattern you want
- **Split complex tasks** — instead of one giant prompt, break big features into smaller steps and work through them one at a time
- **Iterate** — if the result is not perfect, ask follow-up questions or request changes; but if the AI keeps failing, fix it yourself and start a new session

Context Engineering

Context is the information you provide to the AI to help it understand the situation better. The AI only knows what you tell it (plus what it can discover from your open files).

Ways to add context in GitHub Copilot chat:

- The **currently open file** is automatically included as context
- Reference files explicitly using the **add context button**
- Add **URLs** to documentation or web pages
- Paste in **code examples** from official docs
- Use the `#` key while typing to quickly add a file or piece of context

> **Key insight:** Explicitly providing context is always better than hoping the AI will find it on its own.

2.5 Prompt Engineering Best Practices

These are structured rules for getting the best results from any AI coding tool. Think of these as habits to build:

1. Be Concise AND Precise

- Keep your prompt focused and to the point — do not add unnecessary filler or padding
- But do be accurate and clear about what you want
- Example of a bad prompt: *"Hey, can you maybe help me with the code? I'm thinking something related to the user login flow..."*
- Example of a good prompt: *"Add email validation to the `login()` function in `auth.py`. Return a 400 error if the email is not in valid format."*

2. No Unnecessary Context

- Providing the **right** context is crucial — but providing context that does not matter is counterproductive
- Too much irrelevant context confuses the AI and wastes tokens (which cost money)
- Only reference files you **know** are relevant, not files you merely **think** might be relevant
- The same applies to documentation links or pasted examples — only include what is truly needed

3. Think → Plan → Prompt

- Do not just start typing a prompt and "fix things as you go"
- **Think first:** What exactly do I need? What files are involved? What is the expected output?
- **Plan:** Break it into smaller steps if needed
- **Then prompt:** Write a clear, structured instruction
- If you find yourself sending lots of follow-up corrections, it usually means your initial prompt needed more upfront planning
- Use **Plan Mode** in Copilot/Cursor for anything that is not a trivial change

4. Do Not Hide Challenges from the AI

- If you know a task involves a tricky part or a common pitfall — tell the AI upfront
- Do not withhold that information to "test" the AI — this wastes your time and tokens
- Example: If you know that a certain API has a quirky authentication method, mention it and include the correct approach in your prompt
- Include both the **problem** and your **recommended solution approach** directly in the prompt

5. Explicitly Tell the AI Which Tools to Use

- AI agents have access to many tools — built-in tools (web search, terminal), MCP servers, skills, sub-agents, etc.
- Do not hope the AI will automatically choose the right tool — **tell it explicitly**
- Example: Instead of hoping it searches the web, say *"Search the web for the latest documentation on X"*
- Example: If you want it to use the Context7 MCP, say *"Use Context7 to find documentation"*

> **Core principle across all best practices: YOU are in control. YOU steer the AI. The AI is a powerful tool, but it works best when given clear, deliberate direction.**

2.6 Configuration & Customization

Editor Settings

Go to **Settings → search "Copilot"** to find all GitHub Copilot-related settings. These include:

- Enabling/disabling next edit suggestions
- Enabling/disabling AI completions for specific file types
- Experimental features (often added here first)

GitHub Account Settings

On github.com under Copilot settings, you can:

- Enable or disable specific **AI models**
- Enable or disable the **Copilot coding agent**
- Control preview/experimental features

Changing the AI Model

- Open the Command Palette and type "Change completions model" to switch which AI model powers your code completions
- Inside the chat, use the model dropdown to select a different AI model per request

---

Instruction Files

Instruction files let you give Copilot standing rules that are automatically applied to relevant requests — without repeating them in every prompt.

- Stored in your project's `.github/` folder

- Use the `apply_to` field to target specific file types (e.g., `*.py` for Python files only)
- Written in plain text — just describe what you want the AI to do
- **Example use case:** "Always add type hints in Python functions" or "Follow PEP8 style rules"
- Committed with your code, so your whole team benefits

AGENTS.MD

AGENTS.MD is a special file recognized by multiple AI coding tools (GitHub Copilot, Cursor, and others). It is a Markdown file where you write general instructions that apply to all AI interactions in your project.

- Place it in the **root folder** to apply it to everything, or in a **subfolder** to apply it only when working on files in that folder
- Unlike instruction files, it is **not restricted** to specific file types
- Since it is always loaded, keep it short and focused — every line costs tokens (usage)
- Works across multiple AI tools, not just Copilot

Agent Skills

Skills are a special kind of instruction file, stored in `.github/skills/<skill-name>/skill.md`.

- Each skill has a **name** and **description**
- The full skill instructions are **only loaded when the AI decides they are relevant** based on the description — this saves tokens
- Example: A "Performance Review" skill that is only loaded when the task involves checking code speed
- Perfect for company-specific rules, best practices, or code patterns you want to enforce

Prompt Files

Prompt files let you save reusable prompts as commands.

- Stored in `.github/prompts/`
- Written as Markdown files with metadata (name, mode, description)
- Activated in chat by typing `/` followed by the prompt file name
- Example: a `/analyze-code` command that runs a code quality check every time you use it

Built-in Slash Commands

In the chat, typing `/` brings up a list of built-in commands:

- `/fix` — describe an error and ask the AI to fix it
- `/tests` — generate automated tests for a file
- `/search` — search the codebase
- `/help` — explains what each command does

2.7 Background & Cloud Agents

Background Agent

- Runs in the **background** while you keep working on something else
- Uses **Git worktrees** — it copies your project to a separate temporary folder, works there, and merges the result back when done (so it does not interfere with your current work)
- Best for tasks that do not need constant input (e.g., "Write documentation for all public functions")

Cloud Agent

- Runs **in the cloud** (on GitHub's servers), not on your machine
- Requires your project to be linked to a **GitHub repository**
- When done, it creates a **Pull Request** (a suggested change) on GitHub for you to review
- Best for larger, non-urgent tasks you want to hand off completely

Checkpoints (Safe Undo)

A **checkpoint** is a saved snapshot of your code at a specific point in a chat session.

- Every message you send in Agent mode creates a checkpoint automatically
- If you do not like what the agent did, click **"Restore Checkpoint"** on that message to undo all changes made after it
- You can go forward and backward through checkpoints unlimited times
- Works even if you already accepted the changes — it still reverts the files
- This is your safety net when AI does something unexpected

> **Key point:** AI is not deterministic. Even with the same instruction, running it again may produce a slightly different result because there is randomness built into how AI generates text.

---

3. Cursor IDE — Fundamentals

3.1 What is Cursor?

Cursor is a full **IDE (Integrated Development Environment)** — not just an extension. It is built on top of Visual Studio Code (it is literally a modified copy of VS Code called a **fork**), but it has AI features built deeply into every part of it.

Because Cursor is based on VS Code:

- It looks and feels like VS Code
- You can use all the same VS Code extensions
- All the same keyboard shortcuts work

But Cursor adds:

- Extremely powerful **tab completion** that understands your whole codebase
- A built-in AI chat and agent system
- Deep AI integration for inline editing, terminal commands, and more

Pricing: Cursor has a free tier but it is very limited. You need at least a **Pro plan** for real use. For heavy agent work, the Pro+ or Ultra plans are needed.

Privacy: Cursor does **not** use your code to train its AI models. In Settings → General → Privacy Mode, you can confirm this. This is very important for companies working on proprietary (secret) code.

3.2 Tab Completion

Cursor's **Tab Completion** is one of its most celebrated features. It goes far beyond simple code suggestions.

What it does:

- Predicts your next lines of code — not just the current line
- Can make **multi-line edits** across different parts of your file
- Understands the context of recently worked-on files
- Just keep pressing **Tab** to fly through your code

How to use it:

- When Cursor shows a suggestion (faded text), press **Tab** to accept it
- Press **Cmd/Ctrl + →** to accept one word or symbol at a time
- Press **Escape** to dismiss the suggestion

"View" button: Sometimes Cursor shows a small "View" button above a suggestion. Clicking it shows you the full suggested change in detail before you accept.

Comment-based prompting: Just like in Copilot, adding a comment like `# append to file` will cause Cursor to suggest a full function implementing that behaviour.

Why it is powerful:

- Cursor remembers recently edited files and uses them to make more relevant suggestions
- It detects patterns — if you rename a variable, it will suggest updating that same variable name everywhere else in the file, all through Tab presses

Tab Completion Settings (Cursor Settings → Tab)

- Enable/disable the feature globally
- Enable/disable **partial accepts** (word-by-word acceptance)
- Enable/disable suggestions inside comments

3.3 Inline Chat & Edits

Besides tab completion, Cursor has an **Inline Chat** that you open with **Cmd/Ctrl + K** on top of highlighted code.

Three inline modes:

1. **Edit (default)** — Highlight code, describe what to change, and Cursor edits only that selected code. Result shown as green (added) / red (removed) diff.

2. **Quick Question** — Highlight code and ask a question about it. The AI explains the code without making any changes. Useful for understanding unfamiliar code.
3. **Edit Full File** — Like edit mode, but the AI is free to make changes anywhere in the entire file, not just the highlighted section. Useful when a small change has ripple effects.

Inline Chat in the Terminal: You can also open inline chat inside the integrated terminal with **Cmd/Ctrl + K**. Describe a command in plain English and Cursor generates the terminal command for you. You still have to manually press Enter to run it (safety by design).

3.4 Chat & Agent Modes in Cursor

Cursor's sidebar chat (open with **Cmd/Opt/Ctrl + I** or the top-right icon) offers several modes:

Ask Mode

- Pure conversation — no code editing
- Ask questions about your code, request explanations, discuss improvements
- Great for getting up to speed on a new codebase quickly

Agent Mode ⭐ (Default and Most Used)

- The AI actively **edits your code, creates files, and runs commands**
- Works across multiple files at once
- You see the changes as red/green diffs and can accept or reject per block or per file
- The entire chat history is the context — the AI remembers everything from earlier in the same chat session

Plan Mode

- The AI first creates a **detailed plan** before writing any code
- May ask clarifying questions to make the plan as accurate as possible
- You review (and can edit) the plan before clicking **Build** to kick off Agent mode
- Best for complex tasks or when you are not 100% sure your instructions are complete

Background Mode (Remote/Cloud)

- Requires a connected GitHub repository
- The agent works in the cloud and submits a **Pull Request** when done
- You do not need to be online or watching — it runs on GitHub's servers

Selecting AI Models

- You can pick which AI model to use (e.g., GPT-4, Claude, etc.)
- **Auto mode** — lets Cursor pick the best model; gives **unlimited usage** because Cursor may use cheaper models when appropriate
- **Max mode** — forces the top model with maximum context; burns tokens faster, can run out of monthly allowance

- **Deep Think / Brain mode** — activates slower but more thorough reasoning for complex tasks

> **Token** = a unit of text (roughly a word or part of a word). AI services charge based on the number of tokens processed. Every message, every file you share, and every response costs tokens.

3.5 Checkpoints & Reverting Changes

Every message you send in Agent mode creates a **checkpoint** — a snapshot of your code at that moment.

- To go back, click the small **back/revert button** next to any past message
- This undoes all changes made after that message — even if you already accepted the changes
- You can **go forward** again by pressing the checkpoint button again
- Useful safety net when the AI does something unexpected or wrong

Queuing Messages: While the agent is working, you can type a new prompt and press Enter to **queue** it. The agent will finish what it is doing and then pick up your queued message at the next safe point. This is useful for follow-up work without interrupting the current task.

---

4. Cursor IDE — Advanced

4.1 Context Engineering in Cursor

Just like with GitHub Copilot, giving Cursor the right context is the single most important skill for getting good results.

Adding Context with `@`

- Type `@` in the chat to open a context picker
- You can add: files, folders, documentation, terminal sessions, code from other branches, or tell it to search the web

Why use `@` instead of just typing a filename?

- Using `@filename` **loads the actual file content** into the chat and creates a clear reference
- Just typing the filename is less reliable — there is a chance Cursor might not find the right file

Adding Documentation from the Web

- Paste a URL directly into the chat to include a webpage as context
- Or add `Search the web to learn how to use X` as an instruction to make Cursor search the web on its own
- For well-known libraries, you can also add docs from the built-in Cursor docs library

Using XML Tags for Structure

When pasting long content (like official documentation examples) into your prompt, wrapping it in XML-style tags helps the AI understand where the example starts and ends:

```
<openai-docs>
  ... paste documentation here ...
</openai-docs>
```

This is not required but improves clarity for complex prompts.

Knowledge Cutoff Problem

AI models have a **knowledge cutoff date** — they were trained on data up to a certain point in time. After that date, they do not know about new library versions, new APIs, or new tools. If the AI suggests an outdated approach, you can:

- Paste in the relevant section from the current official documentation
- Tell it to search the web for the latest version
- Use an MCP tool like Context7 to fetch up-to-date docs automatically

---

4.2 Cursor Rules & AGENTS.MD

Cursor Rules

Cursor Rules are standing instructions (like permanent prompts) that the AI follows automatically without you having to repeat them in every chat.

- Created in Settings → Rules/Memories/Commands → Add Rule
- Stored in `.cursor/rules/` folder as `.mdc` files
- Can be:
  - **User rules** — apply to all your projects globally
  - **Project rules** — apply to this project only
- Can target **specific file types** with a glob pattern (e.g., `*.tsx` for React files, `*.py` for Python)
- Written in plain text — just describe what you want

Example rule:

> "Create functions via the function keyword, not as arrow functions. Exception: anonymous functions and closures should use arrow functions."

Benefits: You write the rule once and it is automatically applied every time Cursor works on matching files. No need to repeat it in every prompt.

AGENTS.MD in Cursor

Same concept as in GitHub Copilot — a Markdown file of instructions that Cursor loads automatically:

- Place in the **root folder** → applies to all files
- Place in a **subfolder** → only applies when working on files in that subfolder
- Cursor loads it automatically based on which files are being touched
- Can have multiple `agents.md` files at different folder levels for fine-grained control

- Advantage over Cursor Rules: also works with other AI tools (like GitHub Copilot) if they support the convention
- Disadvantage: less fine-grained control compared to Cursor Rules

---

4.3 AI Tools: Web Search, Browser & Terminal

In Agent mode, Cursor has access to **tools** — built-in capabilities beyond just writing code.

Web Search Tool

- Cursor can search the internet to find up-to-date documentation, answers, or examples
- Enable it in Settings → Agents → Context → Web Search Tool
- You can turn on **Auto Accept** so it searches without asking each time
- To encourage the AI to search, include a phrase like "Search the web for the latest X documentation" in your prompt

Browser Tool

- Cursor can connect to a **browser** (either your actual browser or Cursor's built-in one)
- It can visit URLs, take screenshots ("snapshots"), and check if the app looks correct
- It can interact with the app (click buttons, enter text) and read browser console logs
- Useful for testing: after writing code, the agent can open the app and verify it works

Terminal (Command Execution) Tool

- The agent can run terminal commands — install packages, run tests, start servers, etc.
- **By default, it always asks for permission before running a command** — never blindly executes
- You can add trusted commands to an **allow list** so you are not asked repeatedly
- Cursor runs commands in a **sandbox** (a safe, isolated environment) by default:
    - The sandbox prevents the AI from deleting files outside your project
    - Some commands (like installing packages) may fail in the sandbox — they need to run outside it
    - You can choose "Run Everything" mode to allow commands outside the sandbox, but this is less safe

---

4.4 MCP Servers (Model Context Protocol)

MCP (Model Context Protocol) is a standard (a set of rules) invented to let AI agents discover and use external tools — tools that are not built into the AI editor.

Simple analogy: Think of MCP like a "plugin store" for AI agents. Each MCP server is a plugin that adds a new capability to your AI assistant.

Why MCP matters:

- Extends what the AI can do beyond its built-in tools
- Tools can include: database access, design systems, internal company APIs, third-party services, documentation systems

- Cursor supports MCP, meaning you can add any MCP-compatible tool to Cursor

Adding an MCP Server in Cursor

1. Go to the Cursor website's MCP directory
2. Click "Add to Cursor" next to the tool you want
3. Confirm and click Install in Cursor settings
4. The tool is now available to the AI agent

Context7 MCP (Example)

Context7 is a popular MCP tool that helps Cursor find **up-to-date, official documentation** for any library or technology.

- Normally the AI relies on its training data (which has a cutoff date)
- With Context7, the AI can look up the latest docs at the time of the request
- To use it, include **"Use Context7"** in your prompt — this hints the agent to use this specific MCP tool
- If the AI identifies the wrong library, you can **reject the command and explain** why (e.g., "Library name is X, not Y")

MCP Permissions

When an agent wants to use an MCP tool, it asks for permission first:

- Allow once
- Allow always in this workspace
- Deny (with an optional explanation)

4.5 Cursor Memories & Custom Commands

Memories

Memories are an automatic learning feature in Cursor (must be enabled in settings).

- If you repeat certain preferences or instructions across multiple chats, Cursor may automatically detect this pattern and **store it as a memory**
- The memory then acts like an automatically created rule — it is applied to future chats
- You do not have to do anything — it happens on its own
- You can view and manage stored memories in Settings → Rules/Memories/Commands

Custom Commands

Commands are shortcuts that wrap a longer prompt into a simple keyword, activated using `/` in the chat.

- Created in Settings → Rules/Memories/Commands → Add Command
- Can be global (across all projects) or project-specific
- Written in plain natural language — you describe what the command should do
- Example: A `/smart-docs` command that tells the agent: "For the received instruction, explore documentation using Context7 to learn how to correctly use the feature or technology"

- Instead of typing the full instruction every time, you just type `/smart-docs`

Default Cursor Commands:

- `/agent-review` — built-in command to review your code (provided by Cursor)

CursorIgnore File

(Mentioned at the end of Section 4 — concept): Similar to `.gitignore`, a `.cursorignore` file lets you tell Cursor which files and folders to **ignore** when scanning your project for context. This is useful for:

- Keeping sensitive files (like `.env` with secret keys) out of the AI's context
- Excluding large auto-generated files that would waste the context window
- Protecting proprietary code from being analyzed

---

5. Key AI Terminology Glossary

All terms that are likely to appear in an exam are listed here with simple explanations.

Term	Simple Explanation
Code Review Tool	An AI tool that automatically reviews code changes (Pull Requests) on platforms like GitHub. Examples: Bugbot, CodeRabbit
Non-Deterministic	AI outputs are non-deterministic — the same prompt can produce different results each time because randomness is built into how AI generates text
External AI Chatbot	An AI chat interface accessed through a website/app (e.g., ChatGPT, Gemini). Not integrated into your editor; lacks direct access to your codebase
AI (Artificial Intelligence)	Technology that makes machines perform tasks that normally require human intelligence, like understanding text or writing code
LLM (Large Language Model)	The type of AI behind tools like Copilot and Cursor. It is trained on massive amounts of text and code to understand and generate language. GPT-4, Claude, and Gemini are examples
Model	The specific AI brain being used. Different models have different strengths — some are faster, some are more accurate, some think deeper
Prompt	The instruction or question you give to an AI. Like typing a message to a very smart assistant
Prompt Engineering	The skill of writing prompts in a way that gets the best possible result from the AI. Being specific, clear, and structured
Context	All the information the AI has access to when answering your question — your open files, things you pasted in, previous messages in the chat
Context Window	The maximum amount of text an AI can "hold in memory" at one time. If a chat gets too long, older parts fall out of the window

Term	Simple Explanation
Context Engineering	Deliberately choosing and providing the right information to the AI to make its output better
Token	A small unit of text (roughly a word or part of a word). AI tools are priced based on how many tokens are processed
AI Agent	An AI system that can take multi-step tasks, use tools, make decisions, and work autonomously toward a goal — not just answer one question
Agent Mode	A mode in Copilot/Cursor where the AI actively edits files, creates files, and runs commands on its own
IDE	Integrated Development Environment — a software application where you write, test, and run code. Examples: VS Code, Cursor
IDE Extension	An add-on that plugs into an IDE to add extra features. GitHub Copilot is an extension that adds AI features to VS Code
Fork	A copy of an existing project as the starting point for a new one. Cursor is a fork of VS Code
Checkpoint	A saved snapshot of your code state at a specific moment in a chat session. Lets you undo AI changes
Diff View	A split view showing what changed — removed code in red, added code in green
MCP (Model Context Protocol)	A standard protocol that lets AI agents discover and use external tools (plugins), broadening what they can do
MCP Server	A tool that exposes capabilities to an AI agent through the MCP protocol. Example: a documentation lookup tool
Knowledge Cutoff	The date after which an AI model has no knowledge of events or changes. AI is unaware of things released after this date
Tab Completion	In Cursor, an AI feature that predicts the next lines of code and lets you accept them with the Tab key
Inline Chat	A chat interface that opens directly inside the code editor (not a sidebar), activated with a keyboard shortcut
Instruction File	A configuration file (`.github/copilot-instructions.md`) where you write rules that GitHub Copilot always follows
AGENTS.MD	A Markdown file recognized by multiple AI coding tools (Copilot, Cursor) containing instructions that are automatically loaded into agent contexts
Cursor Rules	Project-specific or global rules stored as `.mdc` files in Cursor that the AI agent automatically follows
Prompt File	A saved, reusable prompt stored as a Markdown file. Activatable via `/command` shortcut in the chat

Term	Simple Explanation
Agent Skills	Specialized instruction files for GitHub Copilot that are only loaded when the AI determines they are relevant to the current task
Plan Mode	A mode in both Copilot and Cursor where the AI creates a detailed, editable plan before writing any code
Background Agent	An agent that runs in the background (using Git worktrees) while you continue working, handling tasks that do not need constant interaction
Cloud Agent	An agent that runs on GitHub's cloud servers instead of your machine, and delivers results as a Pull Request
Pull Request	A request to merge code changes into a main codebase, typically reviewed before being accepted. Used by cloud agents to deliver their work
Git Worktree	A Git feature that lets you have multiple working copies of the same repository at once. Used by background agents to work without interrupting your copy
Sandbox	A protected execution environment that limits what commands can do — for example, preventing deletion of files outside the project
Deterministic	Something that always produces the exact same output for the same input. AI is **not** deterministic — the same prompt can give slightly different results each time
Context7	An MCP tool that fetches up-to-date official documentation for libraries, solving the knowledge cutoff problem
Cursor Memories	An automatic feature where Cursor learns your preferences from repeated prompts and stores them as persistent rules
Wipe Coding	Accepting all AI-generated changes without reviewing them. A bad practice — always review AI code
Auto mode (Cursor)	A model selection mode where Cursor chooses the best model automatically, providing unlimited usage by falling back to cheaper models when appropriate
Max mode (Cursor)	A mode that uses the top AI model with maximum context for better results but higher token consumption

Quick Revision Summary

Section 1 — Getting Started

- **Four categories of AI tools:** Chatbots (ChatGPT/Gemini), Code Completion (Copilot/Cursor), Coding Agents (Copilot Agent/Cursor Agent), Code Review (Bugbot/CodeRabbit)
- AI tools require **paid subscriptions** for real use — free tiers are very limited
- AI is **non-deterministic** — same prompt can give different results each time
- AI has a **knowledge cutoff** — it does not know about changes after its training date
- This course is for **developers** — AI works best combined with programming knowledge, not as a replacement

GitHub Copilot

- AI extension for VS Code — requires paid subscription
- **Code completion** → suggest code as you type; accept with Tab
- **Next Edit Suggestions** → suggests changes in other lines based on your edits
- **Four chat modes:** Ask (Q&A only), Edit (targeted edits), Plan (plan first), Agent (full autonomous execution)
- **Agent Mode** = most powerful; creates/edits files, runs commands, asks permission before risky actions
- **Checkpoints** = safe undo for AI changes
- **Prompt files** = reusable prompts activated with `/command`
- **Instruction files** = standing rules automatically applied to AI interactions
- **AGENTS.MD** = universal instruction file recognized across tools
- **Agent Skills** = smart-loaded specialized instructions (only loaded when relevant)
- **Background Agent** = works in the background using Git worktrees
- **Cloud Agent** = works on GitHub cloud, delivers a Pull Request

Cursor

- Full AI-powered IDE (fork of VS Code) — requires paid subscription
- **Privacy Mode** = your code is not used for training (important for companies)
- **Tab Completion** = multi-line AI predictions, accept with Tab
- **Inline Chat (Cmd/Ctrl + K)** = edit, explain, or full-file-edit selected code
- **Three chat modes:** Ask (explore/explain), Plan (plan first), Agent (write code autonomously)
- **Context via `@`** = explicitly add files, docs, URLs to the AI's context
- **Cursor Rules (.mdc files)** = standing instructions applied automatically per file type
- **AGENTS.MD** = universal instruction file, works like Cursor Rules but cross-tool
- **MCP** = plugin system for adding extra tools to the AI (e.g., Context7 for documentation)
- **Tools** = Web Search, Browser preview, Terminal commands — all with permission prompts
- **Memories** = automatic rules Cursor creates by learning your preferences
- **Custom Commands** = reusable prompt shortcuts activated with `/`
- **Sandbox** = protects your computer when the AI runs terminal commands
- **Checkpoint / Revert** = undo AI changes even after accepting them

Universal Best Practices

1. **Be concise AND precise** — clear, focused prompts beat long, vague ones
2. **No unnecessary context** — only reference files and docs you know are relevant
3. **Think → Plan → Prompt** — plan before you type; use Plan Mode for non-trivial tasks
4. **Do not hide challenges from the AI** — share known pitfalls and recommended solutions upfront
5. **Explicitly tell the AI which tools to use** — do not hope it picks the right tool automatically
6. **Split complex tasks** into smaller, focused instructions
7. **Always review AI-generated code** — never just accept blindly (avoid "wipe coding")
8. **Use checkpoints / version control** as your safety net
9. **Understand the knowledge cutoff** — for new libraries, paste docs or use web search / MCP tools
10. **Do not outsource everything** — you are still the developer; AI assists, you steer

---

Notes compiled from lecture transcripts — AI for Developers with GitHub Copilot, Cursor AI & ChatGPT course.