Module 1: Introduction to AI in Programming

CS5013 – Programming with AI
Lecture Notes

V. Krishna Nandivada
Department of Computer Science and Engineering
Indian Institute of Technology Madras

Before we begin

Welcome. The goal of this module is not to teach you to build a large language model. The goal is to give you a working mental model of *what AI coding assistants actually are*, *why they sometimes amaze you and sometimes embarrass you*, and *what responsibilities come with using them*. Everything that follows is written for someone who can already program in Java, but who has never opened up the hood of a coding assistant.

Read these notes alongside the slides; the notes carry the prose, the examples, and the asides, while the slides carry the talking points. The first time you try a worked example, please type the code yourself rather than copy-pasting. The mistakes you will make in the process will be more useful to you than any explanation I can write.

Parts of these notes were prepared/corrected using AI coding assistants (Claude Code, Gemini, ChatGPT). Final responsibility for the content rests with the author.

1 From hand-cranked machines to AI co-authors

Software engineering is, in one sense, the long story of programmers offloading boring work to tools and being left with more interesting work to do. Every generation of programmers has watched some part of their daily job become invisible. Every generation has had to learn a new set of responsibilities to take its place.

Let us walk that timeline briefly. It is not strictly necessary to know the history to use an AI coding assistant, but it is enormously helpful in resisting the urge to treat them as magic.

1.1 Era 1: hand-cranked machines

In the 1950s, "writing a program" meant translating the algorithm in your head into a sequence of machine instructions and physically punching those instructions onto cards. Compile-time errors were not red squiggles; they were piles of cards rejected by the operator. The programmer was also the operator and, often, the test engineer.

The tools that came along to relieve this were assemblers and, eventually, high-level languages such as FORTRAN, COBOL, and LISP. Once those tools existed, programmers stopped worrying about exactly which register held which value (a job they had previously taken very seriously) and started worrying about expressing algorithms more clearly.

A pattern that will repeat

Notice that the new tool did not make programmers redundant. It moved their attention up the stack. Each subsequent layer of abstraction has done the same. Coding assistants are simply the latest layer, and they too will move our attention upward.

1.2 Era 2: editors, compilers, and the trust of tools

Editors like `ed`, `vi`, and `Emacs` let programmers manipulate text fluidly rather than physically. Compilers, instead of merely translating, began to check types, warn about likely mistakes, and optimise generated code. Something new happened in this era: programmers started to *trust their tools*. "If the compiler says it's a type error, then it's a type error" is, when you think about it, an act of faith.

This trust will be relevant when we discuss AI assistants later, because it is a different kind of trust than the one a compiler asks for. A compiler is deterministic and well-specified. An AI assistant is neither.

1.3 Era 3: integrated development environments

The integrated development environment (IDE) era brought together editor, compiler, linker, debugger, and *semantic understanding of the project as a whole*. Suddenly your editor could "go to definition", find every caller of a method, rename a symbol everywhere safely, and propose a fix when a missing import was the only thing standing between you and a green build. Refactoring, which used to be the kind of thing you only attempted under desperate circumstances, became routine.

If you have grown up with VS Code or IntelliJ IDEA, it is easy to underestimate how much intelligence already lives in your IDE before any AI is involved. A modern IDE knows your code's types, your imports, your project structure, your version control state, your test results, and your code style. That is the substrate on which AI assistants now sit.

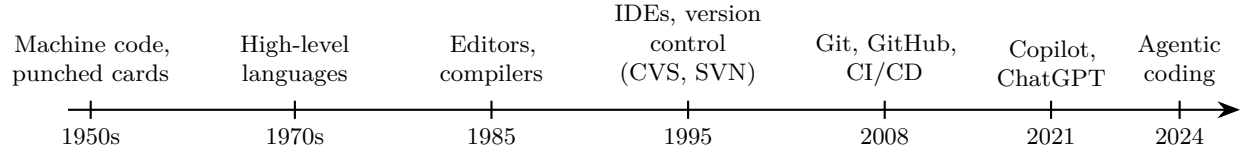
1.4 Era 4: distributed, version-controlled collaboration

Git (2005) and GitHub (2008) reshaped how software is built by groups of people. A change is now an artefact (a commit) that can travel, be reviewed, and be discussed asynchronously. Continuous Integration (CI) runs a test suite on every commit, and Continuous Delivery (CD) packages and rolls out the successful builds automatically. Pull requests turn "working with my teammates" into a structured process, complete with linting, security scanning, and code review.

This era is also when the conventions you now take for granted solidified: small commits with explanatory messages, branches per feature, semantic versioning of dependencies, automated build pipelines, and so on. We will see in Module 3 that AI tools are increasingly woven into this workflow: generating commit messages, summarising pull requests, even labelling issues.

1.5 Era 5: AI-assisted programming

GitHub Copilot popularised inline AI code completion in 2021. ChatGPT made AI conversation about code mainstream in late 2022. Since then, an ecosystem of tools (Claude Code, Cursor, Cline, Continue, and many others) has turned the IDE into a place where you do not just *edit* code but *converse* about it. Agentic systems take this further by allowing the assistant to run commands, edit files, and iterate towards a goal you describe in English.



What disappears in this era is the most mechanical kind of coding: boilerplate, syntactic conversions, naming, and routine refactors. What appears in its place is a new set of responsibilities: writing good prompts, verifying generated code, watching for licence taint, and not pasting your secrets into a chat window.

1.6 A unifying observation

Across all five eras, the programmer's true job has not changed: take a fuzzy human goal and turn it into a precise, verified, executable artefact. What has changed is *where* the precision is enforced. Once it was in the punched card; now it is in the prompt and in the verification step. If you forget that, AI assistants will happily generate plausible nonsense and ship it for you.

2 What is a large language model, really?

When we say "large language model" (LLM), we mean a statistical model that has learned to assign probabilities to sequences of tokens. Given some context c, it can tell us, for each candidate next token t, an estimate of $P(t \mid c)$. "Generating text" is then just sampling tokens one by one and feeding each one back as part of the context for the next.

That is the whole headline. Everything else, transformers, embeddings, attention, is mechanism in service of that one goal.

2.1 Tokens, not words

The first thing to internalise is that an LLM does not see your input as words. It sees it as *tokens*, where a token is whatever the model's tokenizer produces. A typical tokenizer (e.g., Byte Pair Encoding) chops text into sub-word units that may or may not look like words.

For example, the Java statement

```
1 System.out.println("Hi");
```

might be tokenised into something like `System`, `.`, `out`, `.`, `println`, `("`, `Hi`, `");` depending on the tokenizer. A rare identifier like `ArrayListOfWeirdThings` may be split into four or five tokens, while a common one like `ArrayList` might be one or two.

Why this matters in practice

Costs and context-window limits are measured in tokens, not characters and not lines. When a vendor tells you the model supports "128k tokens of context", that is roughly the size of a moderate Java file with all its imports and comments, not the size of an entire Maven project.

2.2 Embeddings: tokens as vectors

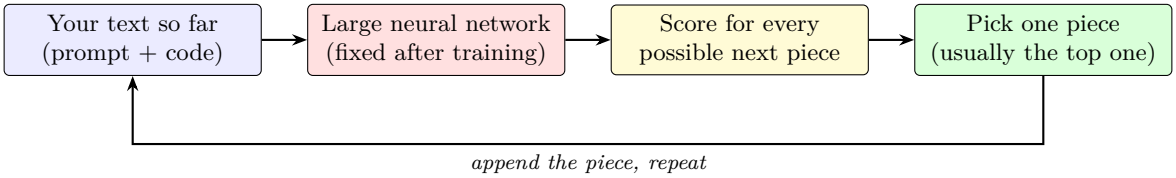
Every token, once identified, is mapped into a high-dimensional vector, its *embedding*. These vectors are not arbitrary numbers; the model has learned them in such a way that semantically related tokens land in nearby regions of space. "`HashMap`" and "`TreeMap`" end up closer to each other than to "`Thread`"; "`for`" and "`while`" end up close.

The same idea ("meaning is location in vector space") also powers *semantic code search*, which is what your copilot does when it tries to find the most relevant pieces of your repository to inject into the prompt. We will return to this in Module 6 when we build small RAG (retrieval-augmented generation) systems ourselves.

2.3 How the model actually turns a prompt into code

At a first pass, an LLM is best understood as *a very elaborate autocomplete*. You give it a chunk of text: your prompt, the comment you just wrote, the code above the cursor. It hands back text, but only a tiny slice at a time: exactly one token. To produce a whole method, the model plays that "guess the next piece" game hundreds or thousands of times, each time appending its

own output to the input and running the whole thing again. That is the entire generation loop. Every capability we will discuss later (writing tests, refactoring, explaining code, chatting) is that same loop with a different prompt in front of it.



Two facts about this loop deserve emphasis, because between them they explain a lot of the odd behaviour you will meet later.

First, the network in the middle is *frozen* once training is over. Its internal numbers (there are billions of them) do not change when you talk to it. Every call is a fresh evaluation. The model does not "remember" anything from your previous session; anything it appears to remember is simply text that has been fed back into the input. When we discuss context windows in Section 4, this is why they matter: the context is the model's only handle on anything about you.

Second, the loop itself is boring. There is no reasoning module, no planner, no algorithm-picker. The cleverness lives in exactly two places: (a) how the neural network is constructed, and (b) what it was trained on. Everything the model appears to "understand" is a consequence of those two choices.

2.4 Where does the ability come from?

The network, as noted, has billions of internal numbers: its *parameters*. Not one of them was written by a human. They are all set by an automatic training procedure that repeats one very simple game an unimaginable number of times.

The game is: take a piece of real text or code from the training corpus, hide a portion of it, and ask the network to guess the hidden piece. Compare the guess with the truth. Nudge the numbers a tiny bit so that next time this input comes up, the guess will be a little closer to the truth. Do that trillions of times over text and code drawn from an enormous corpus.

That is essentially the entire trick. Nobody sat down and told the model what a Java `for` loop looks like, or that `ArrayList` lives in `java.util`. It absorbed those facts as a side-effect of getting good at the guessing game. It also absorbed a great many things we did not intend for it to absorb: popular bugs, licence headers, project-specific idioms. That is why later sections of these notes worry about provenance and secrets.

One more intuition to hold on to before we look at the training game in detail. Because the network can consult *all* the text you have given it when producing each next piece, it can pick up patterns that span long distances. If you declared a variable called `userName` fifty lines above the cursor, and then wrote a comment saying "print the user's name", the model has a real chance of using `userName` correctly. It can see both the declaration and the comment at once. This ability to relate far-apart pieces of text is a large part of why modern LLMs are useful for code in a way that earlier statistical models were not.

The next section looks at the training game ("pre-training") in more detail, and then at how that raw model is polished into the assistants you actually use.

2.5 Pre-training and fine-tuning

A modern LLM is trained in two broad stages.

Pre-training. The model is shown an enormous corpus of text and code and asked, repeatedly, to predict the next token. That is the entire training signal. There is no human teacher saying "this is wrong, that is right". The model just keeps adjusting its parameters to make the actually-observed next token more likely. After hundreds of billions of such updates, the model has internalised an astonishing amount of structure: syntax, idioms, API (application programming interface) names, common bug patterns, even rough world knowledge.

Fine-tuning. A pre-trained base model is good at *predicting* text but not at *following instructions* like "write a Java method that does X". Additional fine-tuning stages teach it to follow instructions and to be helpful.

Code-specific models add an extra wrinkle: they are often trained or fine-tuned on a code-heavy dataset, sometimes with extra objectives such as *fill-in-the-middle*, which we will see shortly.

3 Code-specific LLMs

Why bother training models specifically on code, when general-purpose LLMs already see plenty of it? A few reasons:

- Code has *strict syntax*: it either compiles or it does not, and we can use the compiler/tests as a sharp signal.
- Code has *rich long-range structure*: scopes, types, package layouts, control flow.
- Code corpora are abundant and machine-checkable in ways that prose is not.

3.1 Codex: the prototype

The first widely-noted code model was *Codex* (Chen et al., 2021), built by fine-tuning a GPT (generative pre-trained transformer) model on a large body of public source code, primarily in Python. Codex powered the first version of GitHub Copilot. It introduced a now-standard benchmark, *HumanEval*, consisting of programming problems whose solutions are graded against hidden unit tests rather than text similarity.

3.2 Other code-specialised families

Several open-weights families have followed, each with their own design choices:

- **Code Llama** (Meta): code-tuned variants of LLaMA in multiple sizes.
- **StarCoder / StarCoder2** (BigCode): trained on a curated dataset of permissively licensed source code ("The Stack") with explicit opt-out support for repositories whose authors do not want to be included.
- **DeepSeek-Coder**, **Qwen-Coder**: strong open-weights coder families with multilingual support.

General-purpose flagship models (Claude, GPT-4-class, Gemini) match or exceed dedicated coder models on common benchmarks, but the dedicated coder models often have better licensing for embedding into your own tooling.

3.3 Fill-in-the-middle

Realistic code editing rarely consists of "predict the next token after the cursor"; usually you have code *before* and *after* where you are working. Many code models are trained with a *fill-in-the-middle (FIM)* objective. The input is structured like

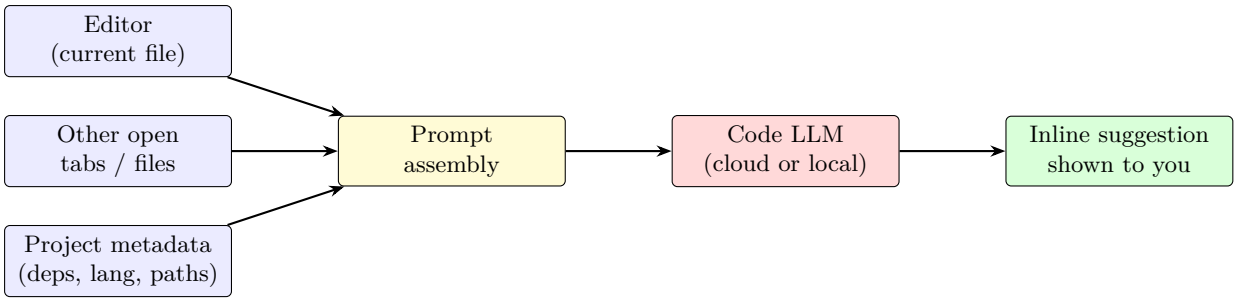
$$\langle \text{prefix} \rangle \mid \langle \text{suffix} \rangle \mid \langle \text{middle} \rangle$$

and the model learns to produce the middle given both prefix and suffix. This is what makes inline "insert here" suggestions feel coherent with the code that comes after.

4 What a "coding copilot" is actually doing

The word "copilot" has become a generic noun for any AI coding assistant. Under the hood, all of them follow a similar five-step recipe:

1. Collect *context* from your editor: the current file around the cursor, recently edited files, project metadata, and possibly other open tabs.
2. Assemble that context into a *prompt*, perhaps adding some boilerplate instructions and possibly retrieved snippets from elsewhere in your repository.
3. Send the prompt to a code LLM, which is usually hosted in the cloud (some setups run a smaller model locally).
4. Receive a completion, post-process it (e.g., strip half-finished lines), and display it to you as an inline suggestion or a chat message.
5. Optionally, learn from your acceptance or rejection – aggregated, sometimes only as telemetry.



4.1 The context window is a small room

Every LLM has a finite *context window*: the maximum number of tokens it can attend to at once. This is the room into which the copilot must squeeze *everything* you want it to consider. Anything that does not fit is, from the model's point of view, simply not there.

For a small Java exercise, your single file plus your imports comfortably fits. For a real codebase, the copilot has to be selective: usually it includes the file around the cursor, the open tabs, perhaps the contents of the methods you have called recently, and a few semantically-retrieved snippets from elsewhere. The strategy for filling the context is what often distinguishes good copilots from mediocre ones; it is not really about the underlying model at all.

4.2 Semantic retrieval over your code

For projects larger than "a handful of files", most copilots build an *embedding index* of your code. They split your code into chunks, compute an embedding vector for each chunk, and store them. When you ask a question or position your cursor, the copilot embeds the query, finds the nearest chunks, and stuffs them into the prompt. This is exactly the mechanism behind *retrieval-augmented generation* (RAG), which we will revisit in Module 6.

4.3 What is in the training data?

A code-trained LLM has seen, broadly:

- Public source code from open-source repositories, filtered by language, size, and licence.
- Documentation, Q&A discussions, technical books, and blog posts.
- In some cases, user-submitted prompts and accepted completions (unless the product offers an opt-out).

There are two consequences worth remembering. First, the model's knowledge of any specific library is only as fresh as its training cutoff; new APIs are invisible to it. Second, anything you type into a prompt may be logged by the provider, which is the foundation of the data-leakage discussion later in this chapter.

4.4 Three faces of an AI assistant

You will encounter AI coding assistants in three modes, each with different trade-offs:

- **Inline completion**: as you type, the model proposes the next few lines. Low latency, small context, mostly mechanical work.
- **Chat**: you ask a question or describe a task in a side panel. Higher latency, larger context, conversational.
- **Agent**: the assistant can run commands, edit files, run your tests, and iterate. Useful for larger tasks but also the highest-risk mode. The assistant can do real things to your machine.

4.5 A first taste in Java

Here is a tiny scene that the rest of the course will revisit many times. You start typing a method and write only a comment describing the intent:

```
// Returns true iff s is a palindrome, ignoring case
// and non-letter characters. Examples:
// "A man, a plan, a canal: Panama" -> true
// "race a car" -> false
public static boolean isPalindrome(String s) {
    // cursor here
}
```

A reasonably good copilot, faced with this comment and signature, will propose something like:

```
int i = 0, j = s.length() - 1;
while (i < j) {
    while (i < j && !Character.isLetter(s.charAt(i))) i++;
    while (i < j && !Character.isLetter(s.charAt(j))) j--;
```

```
    if (Character.toLowerCase(s.charAt(i)) !=
        Character.toLowerCase(s.charAt(j))) return false;
    i++; j--;
}
return true;
```

Two things are worth noticing. First, the model picked up the *intent* from the comment, not just the signature; the examples in the comment carried more weight than any name. Second, the result is a perfectly reasonable two-pointer implementation. It is not a particularly creative one, but it works.

You should still write a small test for it. The next chapter will explain why.

5 What AI assistants are good at – and what they are not

After you have used one of these tools for a week, a pattern emerges. There are categories of tasks where the assistant feels like a tireless co-author. In other categories it confidently produces nonsense. Learning to tell the two apart is half of the skill of using AI in development.

5.1 Where AI assistants shine

- **Boilerplate generation.** Getters, setters, equals/hashCode, builders, simple DTOs (data transfer objects), REST (representational state transfer) controllers wired to existing services, simple CRUD (create/read/update/delete) methods. Code that is mostly determined by a pattern.
- **Syntactic translation.** "Rewrite this Python in Java." "Convert this SQL into a JPA (Java persistence API) criteria query." The pattern is shared; only the surface changes.
- **Local refactoring.** Renaming, extracting a method, splitting a long method into smaller ones, dedup-ing a few similar branches.
- **Explaining unfamiliar code.** "What does this regex do?" "Walk me through this method."
- **Drafting happy-path tests.** "Write a JUnit test that verifies this method returns the expected value on these inputs."
- **Recalling APIs.** "Which class in `java.nio.file` do I use to atomically rename a file?"

In all of these cases, the model is doing pattern-matching against a vast corpus of similar code. There is no real *novelty* required; the answer is similar to many answers the model has already seen.

5.2 Where they struggle

- **Novel logic.** If your problem genuinely does not look like anything in the training data, the model will produce something that looks like related code rather than something that actually solves your problem.
- **Large-scale reasoning.** A program is not the sum of its methods. Architectural invariants, concurrency, performance budgets, deployment constraints: all of these live across many files, often more than fit in the context window.
- **Exact arithmetic and precise specifications.** An LLM is not a calculator. If your spec says "round half-even to two decimal places", do not trust the model to remember that consistently across the codebase.

- **New or rare libraries.** If a library was released after the training cutoff, the model has no idea it exists. It will still happily invent plausible-looking code for it.
- **Security-sensitive code.** "Probably right" is not good enough when the cost of being wrong is a CVE (an entry in the public Common Vulnerabilities and Exposures catalogue).

5.3 Hallucinations

The most important failure mode to internalise is the *hallucination*: the model produces something that looks right but is not. In code, hallucinations look like:

- method names that do not exist (`String.reverse()`; there is no such method on `String`),
- imports of packages that do not exist,
- invented method signatures (`List.removeIf(int)`, when the real signature takes a `Predicate`),
- "confidently wrong" algorithms with off-by-one errors or wrong invariants,
- fabricated documentation URLs.

A small Java example:

```
// Prompt: "Sort xs in descending order using a built-in Java method."
List<Integer> xs = new ArrayList<>(List.of(3, 1, 4, 1, 5, 9));
xs.sortDescending(); // <-- does not exist
```

The compiling form is:

```
xs.sort(Comparator.reverseOrder());
```

You may or may not reproduce this specific mistake – the exact hallucination depends on which model you are using and even on the moment. The point is not the particular example but the failure mode.

The compiler will catch this particular hallucination instantly. Many will not be that obvious.

The calibration problem

LLMs are trained to produce *plausible* output, not *calibrated* output. The model's tone does not change when it is uncertain. It will state a wrong fact with the same confidence as a correct one. This is the deepest reason why *verification is non-negotiable*.

5.4 A rule of thumb

When you receive AI-generated code, ask: "How cheaply can I verify this?"

- If you can compile and run a test in seconds, accept and verify.
- If you can read it in 30 seconds, accept after reading.
- If it touches security, money, or persistent state, assume it is wrong until you have evidence otherwise.
- If you would not commit it without a code review, do not commit it just because a model wrote it.

6 Ethics, law, and the leaky prompt

The technical risks above are visible to you immediately; the wrong code does not compile. The next set of risks is less visible but more consequential. Every time you accept an AI suggestion, three questions are implicitly being answered for you. If you don't answer them deliberately, you have answered them by default, usually badly.

1. Whose code is this?
2. What licence does it carry?
3. What did I just send to a third party?

6.1 Copyright of AI-generated code

Copyright law treats AI-generated work differently in different jurisdictions, and the law is still being written. A few positions are common:

- Output produced purely by a machine, without human creative input, may not be eligible for copyright in some jurisdictions.
- A human-edited derivative may be copyrightable, but only to the extent of the human contribution.
- Output that closely reproduces a copyrighted training-data example can create infringement risk for whoever ships it, regardless of how the output was generated.

The practical implication for you is to treat an AI suggestion the way you would treat a snippet copied from Stack Overflow: *know where it came from and what its licence allows you to do with it.*

6.2 A whirlwind tour of open-source licences

A simplified comparison, not legal advice:

Licence	**What you can do**	**What you must do**
MIT	Use, modify, redistribute, even commercially.	Keep the copyright notice and licence text.
BSD-3	Same as MIT, basically.	Same, plus no using contributor names for endorsement.
Apache 2.0	Same as MIT.	Keep notices; explicit patent grant; state changes.
LGPL	Use as a library, even in closed source.	Release modifications to the library itself.
GPL v2/v3	Use, modify, redistribute.	*Distribute the entire derived work under the same licence.*
AGPL	Same as GPL.	Same, even if users only interact with your code over a network.
Unspecified	Often nothing without explicit permission.	Assume "all rights reserved".

Why this matters for AI suggestions: if the model has been trained on, say, a GPL-licensed library and reproduces a recognisable chunk of it, and you paste that chunk into your MIT-licensed project and ship it, you may be in violation of the GPL's copyleft requirement. Some copilots offer "duplicate detection" that flags suggestions which closely match public code. Use it when it is available.

6.3 Data leakage

Anything you put in a prompt may be:

- logged by the provider for debugging,
- used to improve the service,
- used (depending on terms and product tier) as future training data,
- in rare worst-case bugs, served back to other users.

The categories of input you most need to keep out of prompts are:

- API keys, database credentials, tokens,
- personally identifiable information (PII),
- customer data covered by a contract,
- unreleased product designs or unannounced features,
- security-sensitive code (auth, crypto, payments).

A scenario you must never let happen to you

```
String dbUrl = "jdbc:postgresql://prod-db:5432/orders";
String dbUser = "orders_admin";
String dbPass = "S3cret!Real-Password"; // pasted into a chat
```

"Hey, can you write me a JDBC method that connects to this?" You wanted help with a JDBC method. You just shared production credentials with a third-party service. The credential should be rotated immediately and never again typed into a chat window.

6.4 Practical hygiene

A handful of habits will prevent most of the trouble:

- Use an enterprise tier with a documented "do not train on my inputs" policy when working with sensitive code.
- Redact secrets before pasting; reference them by placeholder.
- Prefer local models for genuinely sensitive data.
- Keep an explicit record in your project of which AI tools were used and where.
- For copyleft-incompatible projects, prefer assistants that filter suggestions against public code, and read suggestions for obvious copy-paste.

7 How we will work in this course

A few ground rules that the rest of the course assumes.

Disclose. Every submission must say what was AI-generated, what was hand-written, and how you verified correctness. You can use as much AI as you like. You cannot pretend not to have.

Verify. Code that you have not compiled, run, or tested does not count as a submission. The compiler and the test runner are your friends.

Respect provenance. If you accept a suggestion that you have reason to believe was copied from a specific source, attribute it. If the source's licence is incompatible with the assignment, do not use it.

Keep secrets out. No real credentials, no real customer data, no copyrighted material you cannot redistribute. This is non-negotiable.

What is not allowed. Submitting AI-generated code you have not read. Pasting another student's code into an AI assistant and submitting the rewrite as your own. Either is treated as academic misconduct.

In the next module we will get serious about *prompting*: how to ask the model for what we want, how to recover when it misunderstands, and how to compose prompts into pipelines. Before then, make sure your environment is set up (see Hands-on Session 1.1) and that you have run an AI suggestion past your compiler at least once. That single act is the foundation of every safe workflow we will build.

Further reading

- Vaswani et al., *Attention Is All You Need*, NeurIPS 2017.
- Brown et al., *Language Models are Few-Shot Learners*, NeurIPS 2020.
- Chen et al., *Evaluating Large Language Models Trained on Code*, 2021 (Codex).
- Open Source Initiative, `https://opensource.org/licenses`.
- Free Software Foundation, GPL FAQ, `https://www.gnu.org/licenses/gpl-faq.html`.
- GitHub Copilot documentation, `https://docs.github.com/en/copilot`.