Module 1: Introduction to AI in Programming

CS5013: Programming with AI

V. Krishna Nandivada

Department of Computer Science and Engineering
Indian Institute of Technology Madras

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

Welcome to CS5013. Academic Formalities

- Project: 25 marks.
- Mid Sem = 25 marks, Final = 50 marks.
- Lots of hands on exercises = no marks, only learning. 
- Relative grading 

Welcome to CS5013. Academic Formalities

- Project: 25 marks.
- Mid Sem = 25 marks, Final = 50 marks.
- Lots of hands on exercises = no marks, only learning. 
- Relative grading 
- Extra marks
  - During the lecture time - individuals can get additional 5 marks.
  - How? - Ask a *good* question, answer a *chosen* question, make a good point! Take 0.5 marks each. Max one mark per day per person.

Welcome to CS5013. Academic Formalities

- Project: 25 marks.
- Mid Sem = 25 marks, Final = 50 marks.
- Lots of hands on exercises = no marks, only learning. 
- Relative grading 
- Extra marks
  - During the lecture time - individuals can get additional 5 marks.
  - How? - Ask a *good* question, answer a *chosen* question, make a good point! Take 0.5 marks each. Max one mark per day per person.
- Attendance requirement – as per institute norms.
  - Proxy attendance - is not a help; actually a disservice.

Welcome to CS5013. Academic Formalities

- Project: 25 marks.
- Mid Sem = 25 marks, Final = 50 marks.
- Lots of hands on exercises = no marks, only learning. 😊
- Relative grading ☹
- Extra marks
  - During the lecture time - individuals can get additional 5 marks.
  - How? - Ask a *good* question, answer a *chosen* question, make a good point! Take 0.5 marks each. Max one mark per day per person.
- Attendance requirement – as per institute norms.
  - Proxy attendance - is not a help; actually a disservice.

Contact (Anytime) :
Instructor: Krishna, Email: nvk@iitm.ac.in, Office: SSB 406.
TA : Omkar Dhawal (cs21d202@smail), Yuvraj Thalukdar (cs23d009@smail), Rahul Utkoor (cs25d005@smail), Mitesh Sharma (cs25s011@smail.iitm.ac.in)

What and what's not: Programming with AI

- **Learn**, Programming with AI, **Learn** Programming, with AI.
- This is not a course on building LLMs from scratch.
- It is a course on building software with LLMs as a collaborator.
- We will study what AI coding assistants can do, when they fail, and how to use them *responsibly*.
- Pre-requisites we assume: object-oriented programming (Java), basic data structures and algorithms.
  - Yes, Java only.
- Default stack for this course: **Java**, **VS Code**, and either **GitHub Copilot** or **Claude Code**.

What you will be able to do after the course

- Write effective prompts that drive an AI assistant to produce working code.
- Recognise when the assistant is hallucinating, leaking secrets, or producing licence-tainted output.
- Use AI for autocompletion, refactoring, debugging, testing, and documentation.
- Build small applications that call LLMs through APIs.
- Reason about the *ethical, legal, and societal* implications of AI-assisted development.

How the course is delivered

- 50 lecture hours, distributed across 8 modules.
- Roughly one hands-on session for every three lectures (about 17 hands-on sessions in total).
- One assignment per week, 14 in total, released as each module finishes.
- Continuous assessment: assignments + hands-on submissions + written tests.

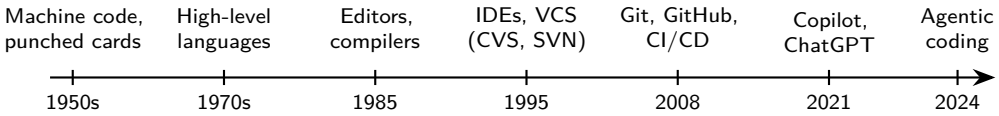
A note on tool use

Using AI tools in this course is encouraged, but every submission must declare what was AI-generated, what was human-written, and how the student verified correctness.

Why start with history?

- Each new tool reshaped what a programmer's day looks like.
- Compilers, IDEs, version control, package managers, and now AI assistants: each one took some boring work away and added new responsibilities.
- Understanding the trajectory helps us treat AI assistants as the *latest layer of abstraction*, not magic.

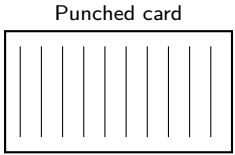
A timeline of programmer productivity tools



- Each era *shifted what a programmer spent time on*.
- Notice the gap between IDE-era productivity tools and AI-era assistants is small in years but large in capability.

Era 1: Manual coding

- Programs written in machine code or assembly.
- Edit-compile-debug cycle measured in *hours*.
- Mistakes were physical: a punched card in the wrong place ruined a batch.
- Programmers were also *operators*. They ran the machine.



Era 2: High-level languages and editors

- FORTRAN, COBOL, LISP, C: programmers stopped writing the machine instructions directly.
- Editors like `ed`, `vi`, `Emacs`: text could be revisited freely.
- The compiler became a *tool the programmer trusts*. "The compiler is always right."
- Boring work that disappeared: register allocation, instruction selection, calling conventions.
- New responsibilities: language semantics, type systems, library APIs.

Era 3: Integrated development environments

- Turbo Pascal, Visual Studio, Eclipse, IntelliJ IDEA.
- Static analysis became always-on: type-checking, jump-to-definition, find-references, refactor-rename.
- Autocomplete became baseline: "Ctrl+Space" to see what's available.
- Refactoring became a *verb*: rename across files, extract method.
- The IDE *understands* the project, not just the file.

Era 4: Distributed collaboration

- Git (2005), GitHub (2008), GitLab, Bitbucket.
- Continuous Integration and Continuous Delivery: every commit runs tests automatically.
- Code review became asynchronous and global.
- Issue trackers, pull request templates, linters in CI.
- Boring work that disappeared: “which version did I send you”.
- New responsibilities: branch strategy, dependency management

Era 5: AI-assisted programming

- GitHub Copilot popularised inline AI completion (2021).
- ChatGPT (2022) made the chat interface mainstream for code generation.
- Tools like Claude Code, Cursor, Cline turned the IDE into an *agent host*.
- What disappears: boilerplate, syntactic conversions, mechanical refactors.
- What is new: prompting and verification of generated code; licence and provenance hygiene come along for the ride.

What stays constant across all eras

The programmer's true job

Translate a fuzzy human goal into a precise, verified executable artefact.

- Each tool shifted *where* the precision and *performance* is enforced.
- AI assistants did not remove the precision. They moved it to the prompt and to verification.
- If the precision is missing, the AI happily fills it with plausible-looking nonsense.

What is a language model?

- A language model assigns probabilities to sequences of tokens.
- Given a context c, it predicts the probability $P(t \mid c)$ for each candidate next token t.
- A *large* language model has billions of parameters and is trained on terabytes of text and code.
- Generation = sampling tokens one at a time, feeding each output back as new context.

Tokens, not words

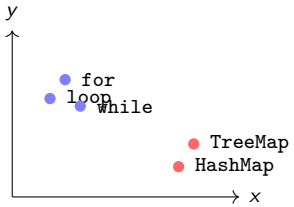
- Text and code are broken into *tokens* by a tokenizer (e.g. Byte Pair Encoding).
- A token might be a whole word, a sub-word, a punctuation mark, or a few characters.
- Cost and context window are counted in tokens, *not* characters or lines.

Input string	**Plausible token split**
`System.out.println("Hi");`	`System`, `.`, `out`, `.`, `println`, `("`, `Hi`, `");`
`ArrayList<Integer>`	`Array`, `List`, `<`, `Integer`, `>`

Exact tokenization depends on the model's tokenizer. The split above is illustrative.

Embeddings: tokens as vectors

- Each token is mapped to a high-dimensional vector (the *embedding*).
- Semantically related tokens land in nearby regions of the vector space.
- Embeddings are also how *semantic search* is done over code, in copilots and RAG (Retrieval Augmented Generation).

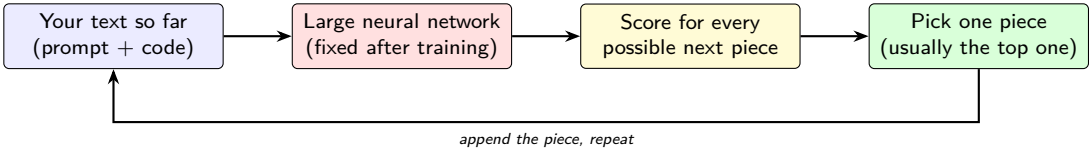


Conceptual 2-D projection.

How does an LLM actually produce code?

- Think of an LLM as *a very elaborate autocomplete*.
- You give it some text: your prompt, a comment, the code above the cursor.
- It produces text back, but only *a tiny piece at a time* (one token).
- To generate a whole method, it plays that "guess the next piece" game *hundreds of times*, each time feeding its own output back in as new context.
- That loop is essentially all the model does. Writing tests, refactoring, explaining code: all of it is the *same loop* with different prompts.

The generation loop, sketched



- The network's internal numbers are *frozen* once training is done. Nothing about your prompt is "remembered" between calls.
- All the cleverness is in (a) how the network is built and (b) what it was trained on. The loop itself is boring.

So where does the ability come from?

- The network has *billions* of internal numbers (called *parameters*).
- Every one of those numbers was *set during training*, not written by a human.
- Together they encode the patterns the model absorbed from an enormous amount of text and code.
- The training procedure itself is astonishingly simple:
  - show the model a piece of real text with a portion hidden,
  - ask it to guess the hidden piece,
  - nudge the numbers so its guess gets a little closer to the truth.
- Do that *trillions* of times. Programming knowledge emerges as a *side-effect* of getting good at this one game.
- That training game is exactly what we look at next.

Pre-training: next-token prediction

- Show the model an enormous corpus of text and code.
- For every position, ask: "what is the next token?"
- Adjust the model's parameters to make the right answer more likely.
- Repeat until the loss stops improving.
- This single objective is enough to make the model learn syntax, idioms, common APIs, and rough semantics.

Fine-tuning and alignment

- Pre-trained models are good at *predicting* text but not at *following instructions*.
- Code-specific fine-tuning may emphasise correctness, formatting, and adherence to coding conventions.

Why code deserves its own LLMs

- Code has strict syntax and a sparse correctness signal: it either compiles or it doesn't, it either passes tests or it doesn't.
- Code has more long-range structure than typical prose (scopes, types, modules).
- Code corpora (e.g., open-source on GitHub) are abundant and machine-checkable.
- Hence, models trained or fine-tuned specifically on code often outperform general LLMs on coding tasks of the same size.

Codex (2021): the prototype

- OpenAI's Codex was a GPT-style model fine-tuned heavily on public source code.
- Powered the first version of GitHub Copilot.
- Demonstrated that natural-language docstrings could reliably drive code generation.
- Famous benchmark: HumanEval, the pass rate on hand-written Python problems with hidden tests.

Other notable code-specialised LLMs

- **Code Llama** (Meta): code-specialised variants of LLaMA, with sizes from 7B upward.
- **StarCoder / StarCoder2** (BigCode): trained on *permissively licensed* source code, with opt-out support.
- **DeepSeek-Coder**, **Qwen-Coder**: strong open-weights coder families.
- Many newer general models (Claude, GPT-4-family, Gemini) match or exceed dedicated coder models on common benchmarks.

Specific benchmark numbers move fast; do not memorise rankings, learn the categories.

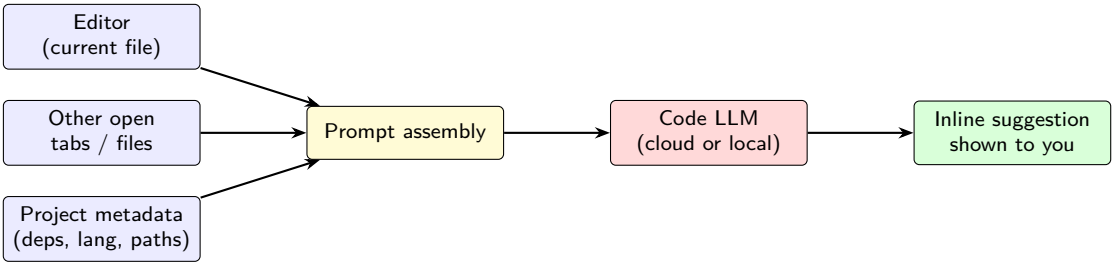
Fill-in-the-middle (FIM)

- Most autocomplete is *not* "predict next token after the cursor".
- Realistic editing has code *before* and *after* the cursor.
- Code-LLMs are often trained with a special FIM objective:
  - Input: $\langle \text{prefix} \rangle \langle \text{SEP} \rangle \langle \text{suffix} \rangle$
  - Output: the *missing middle*.
- This is what powers "insert here" inline suggestions in IDEs.

What “a copilot” actually does

1. Collect *context* from your editor.
2. Build a *prompt* from that context (and possibly extra retrieved snippets).
3. Send the prompt to a code LLM hosted somewhere.
4. Receive a completion, post-process it, and show it to you.
5. Learn from your acceptance / rejection (in some products).

Inside a copilot suggestion



Context windows

- The context window is the maximum number of tokens the model can attend to in one call.
- Modern models have context windows ranging from a few thousand to hundreds of thousands of tokens.
- The copilot has to *choose* what to put in. Usually:
  - the current file around the cursor,
  - recently edited files,
  - retrieved code snippets relevant to the cursor.
- Anything that doesn't fit is invisible to the model.

Embeddings for code retrieval

- For larger projects, the copilot builds an *index of embeddings* of your code.
- When you ask a question, it embeds the query and finds nearby code chunks.
- Those chunks are injected into the prompt as context.
- This is the same mechanism behind Retrieval-Augmented Generation (RAG), which we revisit in Module 6.

What is in the training data?

- Public source code repositories, often filtered for licence, language, and size.
- Documentation, Q&A sites, technical books, blog posts.
- Some providers *also* train on user prompts unless an opt-out is selected. Read the privacy terms.
- The training data has a cutoff date; the model does not know about libraries or APIs released after that.

Inline completion vs. chat vs. agent

- **Inline completion:** you type, the model proposes the next few lines. Fast, low-latency, low-context.
- **Chat:** you ask a question in a side panel. Higher latency, larger context, conversational.
- **Agent:** the assistant can also *take actions*: run commands, edit files, run tests, iterate.
- Each mode is a different trade-off between speed, autonomy, and risk.

A first taste in Java

```java
// Java method: returns true iff s is a palindrome,
// ignoring case and non-letter characters.
public static boolean isPalindrome(String s) {
    // <-- imagine your cursor is here. Copilot suggests:
    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;
}
```

Notice: the assistant picked up the *intent* from the comment, not just the signature.

Where AI assistants shine

- **Boilerplate generation:** getters/setters, builders, simple CRUD, etc.
- **Syntactic translation:** “rewrite this Python in Java”, “convert this SQL into JPA”.
- **Refactoring:** renaming, extracting methods, removing duplication on small scopes.
- **Explaining unfamiliar code:** “what does this regex do?”.
- **Writing tests for existing behaviour:** happy-path unit tests and simple mocks.
- **Recalling APIs:** “which class do I use to read a CSV?”.

Where AI assistants struggle

- **Novel logic** that does not look like anything in the training data.
- **Reasoning over very large codebases** (more code than fits in the context window).
- **Exact arithmetic** and very precise specifications without examples.
- **Unfamiliar or new libraries** released after the training cutoff.
- **Long multi-step plans** without intermediate verification.
- **Security-sensitive code** where being "mostly right" is not enough.

Hallucinations

- LLMs are trained to be *plausible*, not *truthful*.
- In code, hallucinations look like:

Hallucinations

- LLMs are trained to be *plausible*, not *truthful*.
- In code, hallucinations look like:
  - method names that don't exist (`String.reverse()`),
  - imports from packages that don't exist,
  - "confidently wrong" algorithms (off-by-one, wrong invariant),
  - fabricated documentation links.
- The fix is verification: compile, run, test, read.

A small hallucination in Java

```
// Prompt: "Sort the list 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 correct (compiling) version:

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

Lesson: even a one-line completion can be wrong. Always look at the suggestion.

A small hallucination in Java

```
// Prompt: "Sort the list 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 correct (compiling) version:

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

Lesson: even a one-line completion can be wrong. Always look at the suggestion.

You may or not see the exact behavior. Depends on your model.

Calibration: the model doesn't know what it doesn't know

- LLMs rarely say "I'm not sure".
- The confidence in the output is unrelated to its correctness.
- This is different from a compiler error. The compiler tells you when something is wrong.
- Treat AI output the way you treat *a smart but unverified collaborator's first draft*.

What does “large-scale reasoning” mean?

- A program is more than a sum of methods.

What does “large-scale reasoning” mean?

- A program is more than a sum of methods.
- Real engineering needs reasoning about: invariants across modules, concurrency, performance budgets, deployment, observability.

What does “large-scale reasoning” mean?

- A program is more than a sum of methods.
- Real engineering needs reasoning about: invariants across modules, concurrency, performance budgets, deployment, observability.
- LLMs struggle to keep all of that in their head at once.

What does “large-scale reasoning” mean?

- A program is more than a sum of methods.
- Real engineering needs reasoning about: invariants across modules, concurrency, performance budgets, deployment, observability.
- LLMs struggle to keep all of that in their head at once.
- Use AI for *local* reasoning; keep *global* reasoning with you.

Rule of thumb for “how much should I trust this output?”

- If you can *verify it cheaply* (compile, run a test) $\rightarrow$ accept and verify.

Rule of thumb for "how much should I trust this output?"

- If you can *verify it cheaply* (compile, run a test) $\rightarrow$ accept and verify.
- If you can *look at it and read it* in 30 seconds $\rightarrow$ accept after reading.

Rule of thumb for "how much should I trust this output?"

- If you can *verify it cheaply* (compile, run a test) $\rightarrow$ accept and verify.
- If you can *look at it and read it* in 30 seconds $\rightarrow$ accept after reading.
- If it touches *security, money, or persistence* $\rightarrow$ assume it is wrong until proven otherwise.

Rule of thumb for "how much should I trust this output?"

- If you can *verify it cheaply* (compile, run a test) $\rightarrow$ accept and verify.
- If you can *look at it and read it* in 30 seconds $\rightarrow$ accept after reading.
- If it touches *security, money, or persistence* $\rightarrow$ assume it is wrong until proven otherwise.
- If you would not commit it without a code review, do not commit it just because a model wrote it.

Three questions every AI-using programmer must ask

1. **Whose code is this?** Copyright and provenance.

Three questions every AI-using programmer must ask

1. **Whose code is this?** Copyright and provenance.
2. **What licence does it carry?** Compatibility with my project.

Three questions every AI-using programmer must ask

1. **Whose code is this?** Copyright and provenance.
2. **What licence does it carry?** Compatibility with my project.
3. **What did I just send to a third party?** Data leakage.

Copyright of AI-generated code

- Copyright law is jurisdiction-specific and still evolving for AI-generated artefacts.

Copyright of AI-generated code

- Copyright law is jurisdiction-specific and still evolving for AI-generated artefacts.
- Common positions:

Copyright of AI-generated code

- Copyright law is jurisdiction-specific and still evolving for AI-generated artefacts.
- Common positions:
  - Purely machine-generated output may not be eligible for copyright in some jurisdictions.

Copyright of AI-generated code

- Copyright law is jurisdiction-specific and still evolving for AI-generated artefacts.
- Common positions:
  - Purely machine-generated output may not be eligible for copyright in some jurisdictions.
  - A human-edited derivative may be copyrightable, but only the human contribution.

Copyright of AI-generated code

- Copyright law is jurisdiction-specific and still evolving for AI-generated artefacts.
- Common positions:
  - Purely machine-generated output may not be eligible for copyright in some jurisdictions.
  - A human-edited derivative may be copyrightable, but only the human contribution.
- Practical implication: a generated snippet that closely reproduces a copyrighted source can still create an infringement risk.

Copyright of AI-generated code

- Copyright law is jurisdiction-specific and still evolving for AI-generated artefacts.
- Common positions:
  - Purely machine-generated output may not be eligible for copyright in some jurisdictions.
  - A human-edited derivative may be copyrightable, but only the human contribution.
- Practical implication: a generated snippet that closely reproduces a copyrighted source can still create an infringement risk.
- When in doubt, treat AI suggestions as you would a snippet copied from Stack Overflow: *know where it came from and what its licence allows*.

A whirlwind tour of open-source licences

Licence	What you can do	What you must do
MIT	Use, modify, redistribute, even commercially.	Keep the copyright notice and licence text.

A whirlwind tour of open-source licences

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 do not use the contributor names to endorse.

A whirlwind tour of open-source licences

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 do not use the contributor names to endorse.
Apache 2.0	Same as MIT.	Keep notices; explicit patent grant; state changes.

A whirlwind tour of open-source licences

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 do not use the contributor names to endorse.
Apache 2.0	Same as MIT.	Keep notices; explicit patent grant; state changes.
LGPL	Use as a library, including in closed source.	Release modifications to the library itself.

A whirlwind tour of open-source licences

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 do not use the contributor names to endorse.
Apache 2.0	Same as MIT.	Keep notices; explicit patent grant; state changes.
LGPL	Use as a library, including in closed source.	Release modifications to the library itself.
GPL v2/v3	Use, modify, redistribute.	Distribute the entire derived work under the same licence (copyleft).

A whirlwind tour of open-source licences

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 do not use the contributor names to endorse.
Apache 2.0	Same as MIT.	Keep notices; explicit patent grant; state changes.
LGPL	Use as a library, including in closed source.	Release modifications to the library itself.
GPL v2/v3	Use, modify, redistribute.	Distribute the entire derived work under the same licence (copyleft).
AGPL	Same as GPL.	Same, even if the user only *interacts over a network*.

A whirlwind tour of open-source licences

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 do not use the contributor names to endorse.
Apache 2.0	Same as MIT.	Keep notices; explicit patent grant; state changes.
LGPL	Use as a library, including in closed source.	Release modifications to the library itself.
GPL v2/v3	Use, modify, redistribute.	Distribute the entire derived work under the same licence (copyleft).
AGPL	Same as GPL.	Same, even if the user only *interacts over a network*.
Proprietary / unlicensed	Often none without permission.	Assume "all rights reserved" if no licence is stated.

This is a simplification. In real projects, ask your legal counsel for the binding answer.

Why MIT vs. GPL matters when an AI suggests code

- Imagine the AI was trained partly on GPL code.

Why MIT vs. GPL matters when an AI suggests code

- Imagine the AI was trained partly on GPL code.
- It suggests a chunk that closely resembles a GPL-licensed file.

Why MIT vs. GPL matters when an AI suggests code

- Imagine the AI was trained partly on GPL code.
- It suggests a chunk that closely resembles a GPL-licensed file.
- If you paste that into an MIT-licensed project and ship it, you may be in violation.

Why MIT vs. GPL matters when an AI suggests code

- Imagine the AI was trained partly on GPL code.
- It suggests a chunk that closely resembles a GPL-licensed file.
- If you paste that into an MIT-licensed project and ship it, you may be in violation.
- Tools that match suggestions against public code (e.g., Copilot's duplicate detection) exist for this reason.

Why MIT vs. GPL matters when an AI suggests code

- Imagine the AI was trained partly on GPL code.
- It suggests a chunk that closely resembles a GPL-licensed file.
- If you paste that into an MIT-licensed project and ship it, you may be in violation.
- Tools that match suggestions against public code (e.g., Copilot's duplicate detection) exist for this reason.
- Course rule: do not ship AI-generated code into a copyleft-incompatible project without checking provenance.

A concrete GPL-violation scenario in Java

Hypothetical. You prompt for a "bytes to hex string" helper. The assistant replies with:

```
public static String toHexString(byte[] bytes) {
    char[] hexChars = new char[bytes.length * 2];
    int j = 0;
    for (final int v : bytes) {
        hexChars[j++] = HEX_ARRAY[(v & 0xf0) >> 4];
        hexChars[j++] = HEX_ARRAY[v & 0xf];
    }
    return new String(hexChars);
}
```

- Essentially the `toHexString` method from JOSM (Java OpenStreetMap Editor), released under GPL v2.
- If you paste it into an MIT-licensed project and ship the binary, that project is arguably a derivative work of a GPL file — you must relicense as GPL or rewrite the block.
- Note: the assistant may or may not reproduce this specific code. The failure mode is the point.

Source: `https://github.com/JOSM/josm/blob/master/src/org/openstreetmap/josm/tools/Utils.java`

Data leakage: the under-appreciated risk

- Anything you put in a prompt may be:

Data leakage: the under-appreciated risk

- Anything you put in a prompt may be:
  - logged by the provider,

Data leakage: the under-appreciated risk

- Anything you put in a prompt may be:
  - logged by the provider,
  - used to debug/bias their service,

Data leakage: the under-appreciated risk

- Anything you put in a prompt may be:
  - logged by the provider,
  - used to debug/bias their service,
  - used as future training data (depending on settings),

Data leakage: the under-appreciated risk

- Anything you put in a prompt may be:
  - logged by the provider,
  - used to debug/bias their service,
  - used as future training data (depending on settings),
  - retrieved into other users' completions (in rare worst cases).

Data leakage: the under-appreciated risk

- Anything you put in a prompt may be:
  - logged by the provider,
  - used to debug/bias their service,
  - used as future training data (depending on settings),
  - retrieved into other users' completions (in rare worst cases).
- Concretely dangerous: API keys, customer data, unreleased product designs, security-sensitive code.

Data leakage: the under-appreciated risk

- Anything you put in a prompt may be:
  - logged by the provider,
  - used to debug/bias their service,
  - used as future training data (depending on settings),
  - retrieved into other users' completions (in rare worst cases).
- Concretely dangerous: API keys, customer data, unreleased product designs, security-sensitive code.
- Mitigations:

Data leakage: the under-appreciated risk

- Anything you put in a prompt may be:
  - logged by the provider,
  - used to debug/bias their service,
  - used as future training data (depending on settings),
  - retrieved into other users' completions (in rare worst cases).
- Concretely dangerous: API keys, customer data, unreleased product designs, security-sensitive code.
- Mitigations:
  - Use enterprise / business tier with explicit "do not train" guarantees.

Data leakage: the under-appreciated risk

- Anything you put in a prompt may be:
  - logged by the provider,
  - used to debug/bias their service,
  - used as future training data (depending on settings),
  - retrieved into other users' completions (in rare worst cases).
- Concretely dangerous: API keys, customer data, unreleased product designs, security-sensitive code.
- Mitigations:
  - Use enterprise / business tier with explicit "do not train" guarantees.
  - Redact secrets before pasting.

Data leakage: the under-appreciated risk

- Anything you put in a prompt may be:
  - logged by the provider,
  - used to debug/bias their service,
  - used as future training data (depending on settings),
  - retrieved into other users' completions (in rare worst cases).
- Concretely dangerous: API keys, customer data, unreleased product designs, security-sensitive code.
- Mitigations:
  - Use enterprise / business tier with explicit "do not train" guarantees.
  - Redact secrets before pasting.
  - Prefer local models when the data is sensitive.

A simple data-leakage scenario

```
// You paste this into a chat:
String dbUrl = "jdbc:postgresql://prod-db:5432/orders";
String dbUser = "orders_admin";
String dbPass = "S3cret!Real-Password"; // <-- gone
// "Hey, can you write me a method that connects to this DB?"
```

- You wanted help with a JDBC method.
- You just shared production credentials with a third-party service.

A simple data-leakage scenario

```
// You paste this into a chat:
String dbUrl = "jdbc:postgresql://prod-db:5432/orders";
String dbUser = "orders_admin";
String dbPass = "S3cret!Real-Password"; // <-- gone
// "Hey, can you write me a method that connects to this DB?"
```

- You wanted help with a JDBC method.
- You just shared production credentials with a third-party service.
- The credential should be changed immediately.

Course-wide ground rules

What you must do in this course

- Disclose AI assistance in every submission.
- Verify generated code: compile, run, test.
- Never paste secrets, personal data, or copyrighted material you cannot redistribute.
- Respect licences of any external code, including AI-suggested snippets.

Course-wide ground rules

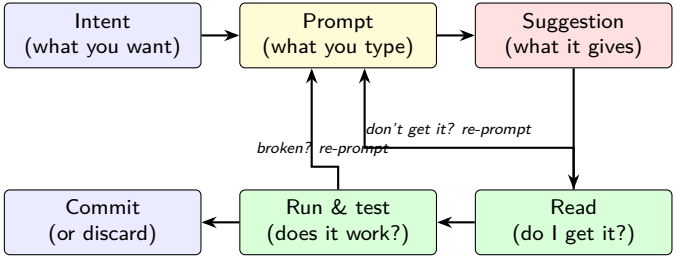
What you must do in this course

- Disclose AI assistance in every submission.
- Verify generated code: compile, run, test.
- Never paste secrets, personal data, or copyrighted material you cannot redistribute.
- Respect licences of any external code, including AI-suggested snippets.

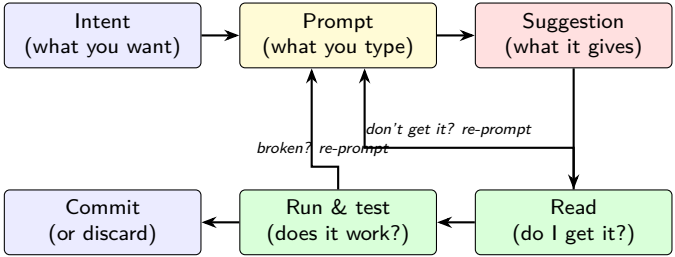
What is not allowed

- Submitting AI-generated code you have not read.
- Pasting another student's code into an AI assistant and submitting the rewrite.

The AI-assisted programming loop



The AI-assisted programming loop



- Six steps. Two of them (Read and Run & test) are the whole game.
- Skipping either is where the "I shipped it and it broke" stories start.

A day in the life: a Java CSV validator

Task. Validate that column `price` in a CSV is numeric. Three iterations with the assistant:

- **Iter 1.** Prompt: *"Read a CSV and check that column 'price' is numeric."* Assistant produces a method; you run it on a normal file. OK.

A day in the life: a Java CSV validator

Task. Validate that column `price` in a CSV is numeric. Three iterations with the assistant:

- **Iter 1.** Prompt: *"Read a CSV and check that column 'price' is numeric."* Assistant produces a method; you run it on a normal file. OK.
- **Iter 2.** You test on an empty file. It returns silently. Re-prompt: *"throw `IllegalArgumentException` on empty input."*

A day in the life: a Java CSV validator

Task. Validate that column `price` in a CSV is numeric. Three iterations with the assistant:

- **Iter 1.** Prompt: *"Read a CSV and check that column 'price' is numeric."* Assistant produces a method; you run it on a normal file. OK.
- **Iter 2.** You test on an empty file. It returns silently. Re-prompt: *"throw `IllegalArgumentException` on empty input."*
- **Iter 3.** You test with the `price` column missing. `NullPointerException`. Re-prompt: *"if 'price' is missing, throw with a helpful message."*

A day in the life: a Java CSV validator

Task. Validate that column `price` in a CSV is numeric. Three iterations with the assistant:

- **Iter 1.** Prompt: *"Read a CSV and check that column 'price' is numeric."* Assistant produces a method; you run it on a normal file. OK.
- **Iter 2.** You test on an empty file. It returns silently. Re-prompt: *"throw `IllegalArgumentException` on empty input."*
- **Iter 3.** You test with the `price` column missing. `NullPointerException`. Re-prompt: *"if 'price' is missing, throw with a helpful message."*
- **Final.** Three rounds of prompt-and-test got us from "works on happy input" to "fails loudly on bad input." Code on the next slide.

A day in the life: the final method

```
public static void validatePrices(Path csv) throws IOException {
    List<String> lines = Files.readAllLines(csv);
    if (lines.isEmpty())
        throw new IllegalArgumentException("Empty file: " + csv);
    int col = indexOf(lines.get(0).split(","), "price");
    if (col < 0)
        throw new IllegalArgumentException("Missing column 'price'");
    for (int i = 1; i < lines.size(); i++)
        Double.parseDouble(lines.get(i).split(",")[col]); // throws
}
```

- Three prompts, three tests, one method that fails loudly on the failure modes you actually care about.
- Notice: none of the three re-prompts came from re-reading the code. They came from running it on bad input.

The same task, done badly

Student ships the first suggestion without reading:

```
import net.apache.csv.CsvValidator; // hallucinated -- no such package

public static boolean validatePrices(Path csv) {
    try {
        CsvValidator v = new CsvValidator(csv);
        return v.isValid("price");
    } catch (Exception e) {
        return true; // swallowed silently
    }
}
```

- Bug 1: import does not exist. Any IDE would flag it in one second. The student did not open the file.
- Bug 2: catch-all returns `true`. Every failure is silently reported as "valid." The tests pass. Production breaks.
- Same task, same tool, same 30 seconds — opposite outcome. The difference is *reading* the suggestion.

Not just hypothetical. . .

Two real world case studies.

Case study: the Samsung / ChatGPT leak (2023)

- Early 2023: engineers at Samsung Semiconductor pasted proprietary source code and meeting notes into ChatGPT — once to debug, once to summarise.
- Three separate incidents within a few weeks.
- The pasted content was retained by the provider; there was no clean way to recall it.
- Samsung banned employees from using public LLM chat services for work; later launched an internal model.
- Lesson: a chat window is not a whiteboard. Whatever you paste leaves your building.

Details and corporate policies have evolved since — check current reporting online for the latest position.

Case study: the GitHub Copilot lawsuit

- November 2022: *Doe v. GitHub, Inc. et al.* filed in the US as a proposed class action.
- Core allegation: Copilot reproduces licensed source code (GPL, MIT, Apache, and others) verbatim or near-verbatim, without preserving the attribution and licence notices those licences require.
- Some claims were dismissed in 2023–2024; others were narrowed and continued.
- Related suits: image-generation cases (*Getty v. Stability AI*, *Andersen v. Stability AI*) are watched for parallel reasoning on training data.
- Why it matters here: the legal answer to "is AI-suggested code safe to ship?" is still being written.

Status of this litigation is a moving target — check current reporting online before quoting the case in an assignment.

Institutional policy: where things stand

IITM at large

An institute-wide policy on AI use in coursework is *under development*. Until it lands, individual courses set their own rules.

This course

- AI-assisted coding is expected, not banned. That is the whole subject.
- Every submission must disclose the AI assistance you used — tools, tasks, verification steps.
- The individual viva at each stage is how we check you actually understand what you submitted.
- Undisclosed AI use is treated as an academic-integrity violation. Disclosed use is normal engineering.
- Detailed rules for the project live in `projects.html`.

What “programming skill” looks like now

More central than before

- Reading code you did not write.

Less central (not gone)

What “programming skill” looks like now

More central than before

- Reading code you did not write.
- Writing precise specifications.

Less central (not gone)

What “programming skill” looks like now

More central than before

- Reading code you did not write.
- Writing precise specifications.
- Judging whether a suggestion is correct.

Less central (not gone)

What “programming skill” looks like now

More central than before

- Reading code you did not write.
- Writing precise specifications.
- Judging whether a suggestion is correct.
- Debugging surprising behaviour under time pressure.

Less central (not gone)

What “programming skill” looks like now

More central than before

- Reading code you did not write.
- Writing precise specifications.
- Judging whether a suggestion is correct.
- Debugging surprising behaviour under time pressure.
- Knowing when to stop trusting the tool.

Less central (not gone)

What “programming skill” looks like now

More central than before

- Reading code you did not write.
- Writing precise specifications.
- Judging whether a suggestion is correct.
- Debugging surprising behaviour under time pressure.
- Knowing when to stop trusting the tool.

Less central (not gone)

- Typing speed.

What “programming skill” looks like now

More central than before

- Reading code you did not write.
- Writing precise specifications.
- Judging whether a suggestion is correct.
- Debugging surprising behaviour under time pressure.
- Knowing when to stop trusting the tool.

Less central (not gone)

- Typing speed.
- Memorising exact API signatures.

What “programming skill” looks like now

More central than before

- Reading code you did not write.
- Writing precise specifications.
- Judging whether a suggestion is correct.
- Debugging surprising behaviour under time pressure.
- Knowing when to stop trusting the tool.

Less central (not gone)

- Typing speed.
- Memorising exact API signatures.
- Rote implementation of standard algorithms.

What “programming skill” looks like now

More central than before

- Reading code you did not write.
- Writing precise specifications.
- Judging whether a suggestion is correct.
- Debugging surprising behaviour under time pressure.
- Knowing when to stop trusting the tool.

Less central (not gone)

- Typing speed.
- Memorising exact API signatures.
- Rote implementation of standard algorithms.
- Writing the same boilerplate for the tenth time.

Preview of Module 2: prompt engineering

Same Java task, two prompts:

```
// Prompt A (weak):
// "write a java function to check email"
```

Preview of Module 2: prompt engineering

Same Java task, two prompts:

```
// Prompt A (weak):
// "write a java function to check email"

// Prompt B (strong):
// Write a Java 17 static method:
```

Preview of Module 2: prompt engineering

Same Java task, two prompts:

```
// Prompt A (weak):
// "write a java function to check email"

// Prompt B (strong):
// Write a Java 17 static method:
// public static boolean isValidEmail(String s)
```

Preview of Module 2: prompt engineering

Same Java task, two prompts:

```
// Prompt A (weak):
// "write a java function to check email"

// Prompt B (strong):
// Write a Java 17 static method:
// public static boolean isValidEmail(String s)
// Return true iff s matches local@domain, where:
```

Preview of Module 2: prompt engineering

Same Java task, two prompts:

```
// Prompt A (weak):
// "write a java function to check email"

// Prompt B (strong):
// Write a Java 17 static method:
// public static boolean isValidEmail(String s)
// Return true iff s matches local@domain, where:
// - local is [a-zA-Z0-9._+-]+
// - domain contains at least one '.' and no whitespace
```

Preview of Module 2: prompt engineering

Same Java task, two prompts:

```
// Prompt A (weak):
// "write a java function to check email"

// Prompt B (strong):
// Write a Java 17 static method:
// public static boolean isValidEmail(String s)
// Return true iff s matches local@domain, where:
// - local is [a-zA-Z0-9._+-]+
// - domain contains at least one '.' and no whitespace
// Return false if s is null or empty.
```

Preview of Module 2: prompt engineering

Same Java task, two prompts:

```
// Prompt A (weak):
// "write a java function to check email"

// Prompt B (strong):
// Write a Java 17 static method:
// public static boolean isValidEmail(String s)
// Return true iff s matches local@domain, where:
// - local is [a-zA-Z0-9._+-]+
// - domain contains at least one '.' and no whitespace
// Return false if s is null or empty.
// Also produce three JUnit 5 tests: one valid, one invalid, one null.
```

Preview of Module 2: prompt engineering

Same Java task, two prompts:

```
// Prompt A (weak):
// "write a java function to check email"

// Prompt B (strong):
// Write a Java 17 static method:
// public static boolean isValidEmail(String s)
// Return true iff s matches local@domain, where:
// - local is [a-zA-Z0-9._+-]+
// - domain contains at least one '.' and no whitespace
// Return false if s is null or empty.
// Also produce three JUnit 5 tests: one valid, one invalid, one null.
```

- Prompt A: assistant guesses, produces something plausible, you spend the next 20 minutes fixing edge cases.

Preview of Module 2: prompt engineering

Same Java task, two prompts:

```
// Prompt A (weak):
// "write a java function to check email"

// Prompt B (strong):
// Write a Java 17 static method:
// public static boolean isValidEmail(String s)
// Return true iff s matches local@domain, where:
// - local is [a-zA-Z0-9._+-]+
// - domain contains at least one '.' and no whitespace
// Return false if s is null or empty.
// Also produce three JUnit 5 tests: one valid, one invalid, one null.
```

- Prompt A: assistant guesses, produces something plausible, you spend the next 20 minutes fixing edge cases.
- Prompt B: assistant produces near-final code, and you have the tests to prove it.

Preview of Module 2: prompt engineering

Same Java task, two prompts:

```
// Prompt A (weak):
// "write a java function to check email"

// Prompt B (strong):
// Write a Java 17 static method:
// public static boolean isValidEmail(String s)
// Return true iff s matches local@domain, where:
// - local is [a-zA-Z0-9._+-]+
// - domain contains at least one '.' and no whitespace
// Return false if s is null or empty.
// Also produce three JUnit 5 tests: one valid, one invalid, one null.
```

- Prompt A: assistant guesses, produces something plausible, you spend the next 20 minutes fixing edge cases.
- Prompt B: assistant produces near-final code, and you have the tests to prove it.
- Module 2 is on how to write prompts like B — and on the failure modes that survive even a good prompt.

Takeaways of the module

- AI-assisted programming is the next layer in a long history of programming tools.
- Behind every copilot is a transformer that predicts tokens from a finite context window.
- Copilots are great at boilerplate and refactoring; weak at novel logic and large-scale reasoning.
- Every AI suggestion has three risks: *incorrectness*, *licence taint*, and *data leakage*.
- In this course, we will learn to use AI as a co-author, not as an oracle.

Looking ahead

- **Module 2:** how to actually *drive* these models: prompt engineering.