Module 3
AI for Code Generation and Autocompletion
CS5013 – Programming with AI

V. Krishna Nandivada

Dept. of Computer Science and Engineering
IIT Madras

Where we are in the course

From “I know how to prompt” to “I use the tool all day”

Module 1 gave you the mental model. Module 2 gave you the prompting patterns.

Module 3 is about the *workflows* in which those patterns show up during real coding:

- Autocompletion in the editor.

Four lecture hours, one hands-on session. Small module, big everyday payoff.

From “I know how to prompt” to “I use the tool all day”

Module 1 gave you the mental model. Module 2 gave you the prompting patterns.

Module 3 is about the *workflows* in which those patterns show up during real coding:

- Autocompletion in the editor.
- Generation of boilerplate code (classes, DTOs, controllers).

Four lecture hours, one hands-on session. Small module, big everyday payoff.

From “I know how to prompt” to “I use the tool all day”

Module 1 gave you the mental model. Module 2 gave you the prompting patterns.

Module 3 is about the *workflows* in which those patterns show up during real coding:

- Autocompletion in the editor.
- Generation of boilerplate code (classes, DTOs, controllers).
- AI-assisted version control (commits, PR summaries, branches).

Four lecture hours, one hands-on session. Small module, big everyday payoff.

Two modes of assistant use

- **Chat mode.** You type a prompt into a side panel, the assistant returns a block of code, you paste it in. This is what Module 2 covered.

Both use the same underlying model. Both matter. Module 3 spends most of its time on inline mode, because that is where the workflow is very different from Module 2.

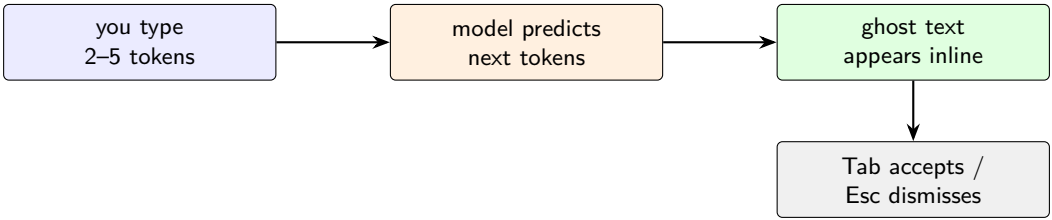
Two modes of assistant use

- **Chat mode.** You type a prompt into a side panel, the assistant returns a block of code, you paste it in. This is what Module 2 covered.
- **Inline (autocomplete) mode.** You type code in the editor; the assistant proposes completions inline as "ghost text". You accept with Tab or dismiss with Esc.

Both use the same underlying model. Both matter. Module 3 spends most of its time on inline mode, because that is where the workflow is very different from Module 2.

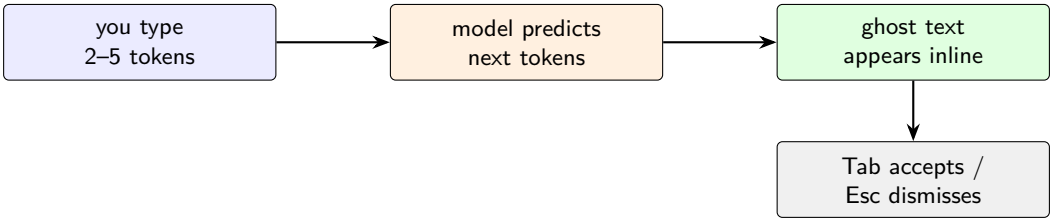
Autocompletion mechanics

Ghost text: the two-second interaction



- Round-trip is ∼300–800ms; you barely notice it if you keep typing.

Ghost text: the two-second interaction



- Round-trip is ∼300–800ms; you barely notice it if you keep typing.
- You accept only what looks right. The rest you throw away by typing over it.

Fill-in-the-Middle (FIM)

Editor completions are not just "predict the next token". When the cursor is in the middle of a file, the assistant has to respect *both sides* of the cursor.

FIM training: the model is trained on `(prefix, suffix, middle)` triples so it can predict the middle given both sides.

This is why an inline completion knows to close a bracket that appears later in the file, or match the return type of the enclosing method.

FIM in action

Cursor position is marked <CURSOR>.

```
public class Cart {
    private final List<LineItem> items = new ArrayList<>();

    public BigDecimal total() {
        <CURSOR>
    }

    public int size() { return items.size(); }
}
```

The model sees the class, the field, the enclosing method's return type, and the sibling `size()`. It suggests:

```
        return items.stream()
                    .map(LineItem::amount)
                    .reduce(BigDecimal.ZERO, BigDecimal::add);
```

What the assistant “sees”

The prompt that the editor plugin sends is not just the visible screen. It typically includes:

- The current file (prefix + suffix around the cursor).

Take-away: the same 100 characters at the cursor produce different completions depending on what else is open in your editor.

What the assistant “sees”

The prompt that the editor plugin sends is not just the visible screen. It typically includes:

- The current file (prefix + suffix around the cursor).
- A few *other recently opened* files, ranked by relevance.

Take-away: the same 100 characters at the cursor produce different completions depending on what else is open in your editor.

What the assistant “sees”

The prompt that the editor plugin sends is not just the visible screen. It typically includes:

- The current file (prefix + suffix around the cursor).
- A few *other recently opened* files, ranked by relevance.
- Language and IDE metadata (JDK version, framework hints).

Take-away: the same 100 characters at the cursor produce different completions depending on what else is open in your editor.

What the assistant “sees”

The prompt that the editor plugin sends is not just the visible screen. It typically includes:

- The current file (prefix + suffix around the cursor).
- A few *other recently opened* files, ranked by relevance.
- Language and IDE metadata (JDK version, framework hints).
- Sometimes: retrieved chunks from your project (repo-level context).

Take-away: the same 100 characters at the cursor produce different completions depending on what else is open in your editor.

Comment as prompt

A Javadoc or // comment above an empty method body is the most reliable inline prompt you have.

```
/** Parse an ISO-8601 date; return null if the string is null
 * or unparseable. Do NOT throw. */
public static LocalDate safeParse(String s) {
    // <CURSOR>
}
```

- The signature pins the return type.

Delete the comment after? Only if the code still reads without it. Often the comment was useful *documentation*, not just a prompt.

Comment as prompt

A Javadoc or // comment above an empty method body is the most reliable inline prompt you have.

```
/** Parse an ISO-8601 date; return null if the string is null
 * or unparseable. Do NOT throw. */
public static LocalDate safeParse(String s) {
    // <CURSOR>
}
```

- The signature pins the return type.
- The comment carries the constraints (null-safety, no throw).

Delete the comment after? Only if the code still reads without it. Often the comment was useful *documentation*, not just a prompt.

Comment as prompt

A Javadoc or // comment above an empty method body is the most reliable inline prompt you have.

```
/** Parse an ISO-8601 date; return null if the string is null
 * or unparseable. Do NOT throw. */
public static LocalDate safeParse(String s) {
    // <CURSOR>
}
```

- The signature pins the return type.
- The comment carries the constraints (null-safety, no throw).
- The assistant merges both into its completion.

Delete the comment after? Only if the code still reads without it. Often the comment was useful *documentation*, not just a prompt.

The trigger points

Completions typically fire on:

- pressing a newline;

If completions are not appearing, one of these is the reason:

The trigger points

Completions typically fire on:

- pressing a newline;
- pressing `.` (dot) after an identifier;

If completions are not appearing, one of these is the reason:

The trigger points

Completions typically fire on:

- pressing a newline;
- pressing `.` (dot) after an identifier;
- pausing typing for $\sim$150ms;

If completions are not appearing, one of these is the reason:

The trigger points

Completions typically fire on:

- pressing a newline;
- pressing `.` (dot) after an identifier;
- pausing typing for $\sim$150ms;
- explicit shortcut (Alt+\, Ctrl+Enter).

If completions are not appearing, one of these is the reason:

The trigger points

Completions typically fire on:

- pressing a newline;
- pressing `.` (dot) after an identifier;
- pausing typing for $\sim$150ms;
- explicit shortcut (Alt+\, Ctrl+Enter).

If completions are not appearing, one of these is the reason:

- the file is too large (some plugins cap prefix size);

The trigger points

Completions typically fire on:

- pressing a newline;
- pressing `.` (dot) after an identifier;
- pausing typing for $\sim$150ms;
- explicit shortcut (Alt+\, Ctrl+Enter).

If completions are not appearing, one of these is the reason:

- the file is too large (some plugins cap prefix size);
- the extension is disabled for that language;

The trigger points

Completions typically fire on:

- pressing a newline;
- pressing `.` (dot) after an identifier;
- pausing typing for $\sim$150ms;
- explicit shortcut (Alt+\, Ctrl+Enter).

If completions are not appearing, one of these is the reason:

- the file is too large (some plugins cap prefix size);
- the extension is disabled for that language;
- you are behind a proxy and the round-trip is too slow.

VS Code + Copilot: a working setup

Assumed stack

- VS Code with *Extension Pack for Java*.
- Either *GitHub Copilot* (Copilot Chat + Copilot inline) or *Claude Code* (chat in the terminal + a limited inline mode).
- A Java 17+ JDK on the path.

The rest of the module assumes you have this working from Module 1's setup exercise.

Copilot in the editor: three surfaces

- **Inline suggestions.** Ghost text as you type.

Different surfaces suit different tasks. We will use inline for local edits and chat for larger requests.

Copilot in the editor: three surfaces

- **Inline suggestions.** Ghost text as you type.
- **Copilot Chat.** A side panel where you ask questions in natural language ("explain this method", "write a test for this").

Different surfaces suit different tasks. We will use inline for local edits and chat for larger requests.

Copilot in the editor: three surfaces

- **Inline suggestions.** Ghost text as you type.
- **Copilot Chat.** A side panel where you ask questions in natural language ("explain this method", "write a test for this").
  - Ctrl + Cmd + I (mac), Ctrl + Alt + I (Win/Linux)

Different surfaces suit different tasks. We will use inline for local edits and chat for larger requests.

Copilot in the editor: three surfaces

- **Inline suggestions.** Ghost text as you type.
- **Copilot Chat.** A side panel where you ask questions in natural language ("explain this method", "write a test for this").
  - Ctrl + Cmd + I (mac), Ctrl + Alt + I (Win/Linux)
- **Inline chat.** a chat prompt anchored to the current selection. The response replaces or edits the selection in place.

Different surfaces suit different tasks. We will use inline for local edits and chat for larger requests.

Copilot in the editor: three surfaces

- **Inline suggestions.** Ghost text as you type.
- **Copilot Chat.** A side panel where you ask questions in natural language ("explain this method", "write a test for this").
  - Ctrl + Cmd + I (mac), Ctrl + Alt + I (Win/Linux)
- **Inline chat.** a chat prompt anchored to the current selection. The response replaces or edits the selection in place.
  - Cmd + I (mac), Ctrl + I (Win/Linux)

Different surfaces suit different tasks. We will use inline for local edits and chat for larger requests.

A first-day workflow

1. Open a Java file. Type a method signature and a Javadoc comment describing what it should do.

Nothing in this workflow is new. What is new is how fast steps 1–3 are once the completion works.

A first-day workflow

1. Open a Java file. Type a method signature and a Javadoc comment describing what it should do.
2. Wait one second. Accept the inline completion if it looks right.

Nothing in this workflow is new. What is new is how fast steps 1–3 are once the completion works.

A first-day workflow

1. Open a Java file. Type a method signature and a Javadoc comment describing what it should do.
2. Wait one second. Accept the inline completion if it looks right.
3. Compile and run.

Nothing in this workflow is new. What is new is how fast steps 1–3 are once the completion works.

A first-day workflow

1. Open a Java file. Type a method signature and a Javadoc comment describing what it should do.
2. Wait one second. Accept the inline completion if it looks right.
3. Compile and run.
4. If it does not compile, either fix it inline (Ctrl+I, "fix the compile error") or throw the completion away and try again.

Nothing in this workflow is new. What is new is how fast steps 1–3 are once the completion works.

Boilerplate generation

The pattern

Boilerplate is code that is:

- *predictable* – has a well-known shape;

This is where AI completion shines. The model has seen millions of instances of the pattern; it can fill it in with high accuracy; and you can read the result at a glance.

The pattern

Boilerplate is code that is:

- *predictable* – has a well-known shape;
- *tedious* – expensive to write by hand;

This is where AI completion shines. The model has seen millions of instances of the pattern; it can fill it in with high accuracy; and you can read the result at a glance.

The pattern

Boilerplate is code that is:

- *predictable* – has a well-known shape;
- *tedious* – expensive to write by hand;
- *low-risk* – easy to verify by reading.

This is where AI completion shines. The model has seen millions of instances of the pattern; it can fill it in with high accuracy; and you can read the result at a glance.

Classes and records

Ask for a data class – immutable, with equals, hashCode, and a toString. In Java 17 the assistant will almost always suggest a `record`:

```
public record Address(
    String street, String city,
    String state, String pin) {}
```

Ten seconds of typing. Verify: does it compile? Verify: does it have the fields you wanted? Verify: do you actually need immutability here?

Data classes: equals, hashCode, toString

A frequent boilerplate ask: "give me an immutable data class with `equals`, `hashCode`, and `toString` over these fields".

```
public final class Employee {
    private final String name;
    private final int id;
    private final LocalDate joined;
    // constructor, getters, equals, hashCode, toString ...
}
```

- Verify: does it compile?

Data classes: equals, hashCode, toString

A frequent boilerplate ask: "give me an immutable data class with `equals`, `hashCode`, and `toString` over these fields".

```
public final class Employee {
    private final String name;
    private final int id;
    private final LocalDate joined;
    // constructor, getters, equals, hashCode, toString ...
}
```

- Verify: does it compile?
- Verify: are *all* fields used in both `equals` and `hashCode`?

Data classes: equals, hashCode, toString

A frequent boilerplate ask: "give me an immutable data class with `equals`, `hashCode`, and `toString` over these fields".

```
public final class Employee {
    private final String name;
    private final int id;
    private final LocalDate joined;
    // constructor, getters, equals, hashCode, toString ...
}
```

- Verify: does it compile?
- Verify: are *all* fields used in both `equals` and `hashCode`?
- Verify: is `equals` symmetric – `a.equals(b) == b.equals(a)`?

Multi-key Comparator chains

Sort employees by department, then by salary (descending), then by name:

```
Comparator<Employee> byRank =
    Comparator.comparing(Employee::department)
              .thenComparing(Employee::salary, Comparator.reverseOrder())
              .thenComparing(Employee::name);
```

- Type the first `Comparator.comparing(...)` yourself.

Multi-key Comparator chains

Sort employees by department, then by salary (descending), then by name:

```
Comparator<Employee> byRank =
    Comparator.comparing(Employee::department)
              .thenComparing(Employee::salary, Comparator.reverseOrder())
              .thenComparing(Employee::name);
```

- Type the first `Comparator.comparing(...)` yourself.
- Ghost text fills the two `.thenComparing(...)` tail calls once it has seen the pattern.

Multi-key Comparator chains

Sort employees by department, then by salary (descending), then by name:

```
Comparator<Employee> byRank =
    Comparator.comparing(Employee::department)
              .thenComparing(Employee::salary, Comparator.reverseOrder())
              .thenComparing(Employee::name);
```

- Type the first `Comparator.comparing(...)` yourself.
- Ghost text fills the two `.thenComparing(...)` tail calls once it has seen the pattern.
- Watch the second key: `reverseOrder()` is exactly where autocomplete forgets the "descending" intent.

Rule of thumb for boilerplate

- Type the *first* instance by hand.

Repetition is where copy-drift kicks in. You will catch it if you read line by line the first time; you will miss it the fifth time.

Rule of thumb for boilerplate

- Type the *first* instance by hand.
- Let the assistant fill in the second and third.

Repetition is where copy-drift kicks in. You will catch it if you read line by line the first time; you will miss it the fifth time.

Rule of thumb for boilerplate

- Type the *first* instance by hand.
- Let the assistant fill in the second and third.
- Do not let the assistant fill in a fourth without a conscious read of it.

Repetition is where copy-drift kicks in. You will catch it if you read line by line the first time; you will miss it the fifth time.

Regex, and regex in Java

A *regular expression* is a compact pattern that describes a set of strings. Built from a small alphabet:

- **Literals**: a, 7, −.

Java's engine lives in `java.util.regex`. Two classes matter: `Pattern` (compiled regex) and `Matcher` (applied to input).

Regex, and regex in Java

A *regular expression* is a compact pattern that describes a set of strings. Built from a small alphabet:

- **Literals**: `a`, `7`, `-`.
- **Character classes**: **`.`** (any char), `\d` (digit), `\w` (word char), `[a-z]`, `[^0-9]` (negated).

Java's engine lives in `java.util.regex`. Two classes matter: `Pattern` (compiled regex) and `Matcher` (applied to input).

Regex, and regex in Java

A *regular expression* is a compact pattern that describes a set of strings. Built from a small alphabet:

- **Literals**: `a`, `7`, `-`.
- **Character classes**: `.` (any char), `\d` (digit), `\w` (word char), `[a-z]`, `[^0-9]` (negated).
- **Quantifiers**: `*` (0+), `+` (1+), `?` (0/1), `{n,m}`.

Java's engine lives in `java.util.regex`. Two classes matter: `Pattern` (compiled regex) and `Matcher` (applied to input).

Regex, and regex in Java

A *regular expression* is a compact pattern that describes a set of strings. Built from a small alphabet:

- **Literals**: `a`, `7`, `-`.
- **Character classes**: `.` (any char), `\d` (digit), `\w` (word char), `[a-z]`, `[^0-9]` (negated).
- **Quantifiers**: `*` (0+), `+` (1+), `?` (0/1), `{n,m}`.
- **Anchors**: `^` (start), `$` (end).

Java's engine lives in `java.util.regex`. Two classes matter: `Pattern` (compiled regex) and `Matcher` (applied to input).

Regex, and regex in Java

A *regular expression* is a compact pattern that describes a set of strings. Built from a small alphabet:

- **Literals**: a, 7, -.
- **Character classes**: `.` (any char), `\d` (digit), `\w` (word char), `[a-z]`, `[^0-9]` (negated).
- **Quantifiers**: `*` (0+), `+` (1+), `?` (0/1), `{n,m}`.
- **Anchors**: `^` (start), `$` (end).
- **Groups**: `(...)` captures, `(?:...)` does not capture – Self reading.

Java's engine lives in `java.util.regex`. Two classes matter: `Pattern` (compiled regex) and `Matcher` (applied to input).

Java regex processing: the three verbs

```
Pattern p = Pattern.compile("\\d{4}-\\d{2}-\\d{2}");
Matcher m = p.matcher("meet on 2026-02-14 sharp");

m.matches(); // does the WHOLE input match?
m.find(); // is there a match anywhere? (advances)
m.lookingAt();// does the match start at position 0?
```

- Java strings double every backslash: `"\\d"` is the regex `\d`.

Java regex processing: the three verbs

```
Pattern p = Pattern.compile("\\d{4}-\\d{2}-\\d{2}");
Matcher m = p.matcher("meet on 2026-02-14 sharp");

m.matches(); // does the WHOLE input match?
m.find(); // is there a match anywhere? (advances)
m.lookingAt();// does the match start at position 0?
```

- Java strings double every backslash: `"\\d"` is the regex `\d`.
- Compile `Pattern` once (e.g. a `static final` field); reuse. `Matcher` is per-input, not thread-safe.

Java regex processing: the three verbs

```
Pattern p = Pattern.compile("\\d{4}-\\d{2}-\\d{2}");
Matcher m = p.matcher("meet on 2026-02-14 sharp");

m.matches(); // does the WHOLE input match?
m.find(); // is there a match anywhere? (advances)
m.lookingAt();// does the match start at position 0?
```

- Java strings double every backslash: `"\\d"` is the regex `\d`.
- Compile `Pattern` once (e.g. a `static final` field); reuse. `Matcher` is per-input, not thread-safe.
- Extract captured groups with `m.group(1)`, `m.group(2)`, ...

Java regex processing: the three verbs

```
Pattern p = Pattern.compile("\\d{4}-\\d{2}-\\d{2}");
Matcher m = p.matcher("meet on 2026-02-14 sharp");

m.matches(); // does the WHOLE input match?
m.find(); // is there a match anywhere? (advances)
m.lookingAt();// does the match start at position 0?
```

- Java strings double every backslash: `"\\d"` is the regex `\d`.
- Compile `Pattern` once (e.g. a `static final` field); reuse. `Matcher` is per-input, not thread-safe.
- Extract captured groups with `m.group(1)`, `m.group(2)`, ...
- Shortcuts on `String`: `s.matches(regex)`, `s.split(regex)`, `s.replaceAll(regex, repl)` – convenient but recompile each call.

Regex generation

Regex is boilerplate at its most extreme: high skill-cost to write, low skill-cost to *read* once you have test strings.

Prompt: "Java regex that matches an ISO-8601 date (`YYYY-MM-DD`), anchored at both ends, no capturing groups."

```
Pattern ISO_DATE =
    Pattern.compile("^\\d{4}-\\d{2}-\\d{2}$");
```

- Always test with a few positives and a few negatives before you commit.

Regex generation

Regex is boilerplate at its most extreme: high skill-cost to write, low skill-cost to *read* once you have test strings.

Prompt: "Java regex that matches an ISO-8601 date (`YYYY-MM-DD`), anchored at both ends, no capturing groups."

```
Pattern ISO_DATE =
    Pattern.compile("^\\d{4}-\\d{2}-\\d{2}$");
```

- Always test with a few positives and a few negatives before you commit.
- Watch for greedy-vs-lazy mistakes (`.*` vs `.*?`) in longer patterns.

Regex generation

Regex is boilerplate at its most extreme: high skill-cost to write, low skill-cost to *read* once you have test strings.

Prompt: "Java regex that matches an ISO-8601 date (`YYYY-MM-DD`), anchored at both ends, no capturing groups."

```
Pattern ISO_DATE =
    Pattern.compile("^\\d{4}-\\d{2}-\\d{2}$");
```

- Always test with a few positives and a few negatives before you commit.
- Watch for greedy-vs-lazy mistakes (`.*` vs `.*?`) in longer patterns.
- Ask the assistant to *explain* its regex back to you – if it cannot, that is a signal.

Version control: a working intro to git

Why version control?

Before we ask an assistant to *write* git commands, we need a quick shared vocabulary.

- **Undo without fear.** Every state you commit is recoverable.

Git is one implementation of these ideas – currently the dominant one.

Why version control?

Before we ask an assistant to *write* git commands, we need a quick shared vocabulary.

- **Undo without fear.** Every state you commit is recoverable.
- **Blame.** "Who changed this line, when, and why?" – `git blame` plus the commit message.

Git is one implementation of these ideas – currently the dominant one.

Why version control?

Before we ask an assistant to *write* git commands, we need a quick shared vocabulary.

- **Undo without fear.** Every state you commit is recoverable.
- **Blame.** "Who changed this line, when, and why?" – `git blame` plus the commit message.
- **Share.** Two people editing the same file is a merge, not a fight over the shared drive.

Git is one implementation of these ideas – currently the dominant one.

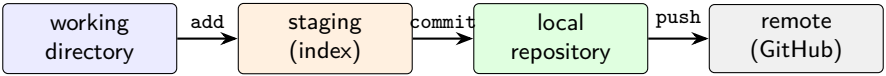
Why version control?

Before we ask an assistant to *write* git commands, we need a quick shared vocabulary.

- **Undo without fear.** Every state you commit is recoverable.
- **Blame.** "Who changed this line, when, and why?" – `git blame` plus the commit message.
- **Share.** Two people editing the same file is a merge, not a fight over the shared drive.
- **Review.** Pull requests are the modern code-review artefact.

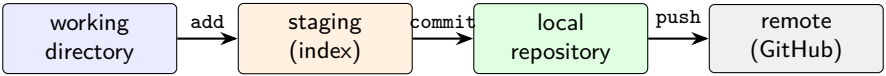
Git is one implementation of these ideas – currently the dominant one.

Git's four places



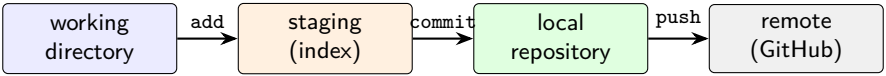
- **Working directory** – files as your editor sees them.

Git's four places



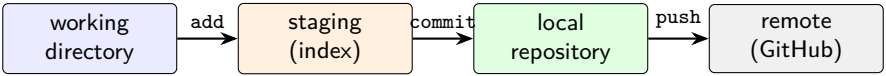
- **Working directory** – files as your editor sees them.
- **Staging area** – what will go into the next commit.

Git's four places



- **Working directory** – files as your editor sees them.
- **Staging area** – what will go into the next commit.
- **Local repository** – the `.git` folder; your committed history.

Git's four places



- **Working directory** – files as your editor sees them.
- **Staging area** – what will go into the next commit.
- **Local repository** – the `.git` folder; your committed history.
- **Remote** – another copy, hosted for sharing / review.

The everyday loop

Five commands cover 80% of solo git use:

```
git status # what is where -- run this constantly
git diff # what changed vs. the last commit
git add <files> # stage changes for the next commit
git commit -m "..." # record the staged changes
git log --oneline # what the history looks like
```

- `git diff --staged` – see *only* what will go into the next commit. (We use it later.)

The everyday loop

Five commands cover 80% of solo git use:

```
git status # what is where -- run this constantly
git diff # what changed vs. the last commit
git add <files> # stage changes for the next commit
git commit -m "..." # record the staged changes
git log --oneline # what the history looks like
```

- `git diff -staged` – see *only* what will go into the next commit. (We use it later.)
- `git status` is the first command to type any time you feel lost.

Branches and merges

A branch is a movable pointer to a commit.

```
git switch -c feat/cart-total # create + move to a new branch
git switch main # go back
git merge feat/cart-total # bring the branch into main
git rebase main # replay branch commits atop main
```

- **Merge** preserves history topology (a merge commit appears).

Branches and merges

A branch is a movable pointer to a commit.

```
git switch -c feat/cart-total # create + move to a new branch
git switch main # go back
git merge feat/cart-total # bring the branch into main
git rebase main # replay branch commits atop main
```

- **Merge** preserves history topology (a merge commit appears).
- **Rebase** rewrites history to a linear shape (cleaner log, but rewrites commits).

Branches and merges

A branch is a movable pointer to a commit.

```
git switch -c feat/cart-total # create + move to a new branch
git switch main # go back
git merge feat/cart-total # bring the branch into main
git rebase main # replay branch commits atop main
```

- **Merge** preserves history topology (a merge commit appears).
- **Rebase** rewrites history to a linear shape (cleaner log, but rewrites commits).
- Rule of thumb: rebase local branches, merge shared ones.

Remotes, pull requests, conflicts

```
git clone <url> # start from an existing remote
git push -u origin feat/x # push a new branch and track it
git pull # fetch + merge from tracked remote
```

- **Pull request (PR).** Ask to merge your branch into `main`; reviewers comment in the hosted UI.

Remotes, pull requests, conflicts

```
git clone <url> # start from an existing remote
git push -u origin feat/x # push a new branch and track it
git pull # fetch + merge from tracked remote
```

- **Pull request (PR).** Ask to merge your branch into `main`; reviewers comment in the hosted UI.
- **Conflict.** Git could not choose which side of a divergent edit to keep – it marks the file with `<<<<<<<`, `=======`, `>>>>>>>` and stops.

Remotes, pull requests, conflicts

```
git clone <url> # start from an existing remote
git push -u origin feat/x # push a new branch and track it
git pull # fetch + merge from tracked remote
```

- **Pull request (PR).** Ask to merge your branch into `main`; reviewers comment in the hosted UI.
- **Conflict.** Git could not choose which side of a divergent edit to keep – it marks the file with `<<<<<<<`, `=======`, `>>>>>>>` and stops.
- You edit those markers away, `git add` the file, and continue the merge or rebase.

AI-assisted commit messages

What a good commit message looks like

- First line: 50–72 characters, imperative mood (“Add null-check to `Cart.total()`”).

Kernel-style conventions vary by team, but the shape is stable. This makes commit messages a natural target for AI generation.

What a good commit message looks like

- First line: 50–72 characters, imperative mood (“Add null-check to `Cart.total()`”).
- Blank line.

Kernel-style conventions vary by team, but the shape is stable. This makes commit messages a natural target for AI generation.

What a good commit message looks like

- First line: 50–72 characters, imperative mood (“Add null-check to `Cart.total()`”).
- Blank line.
- Body: what changed, why, and any subtlety a future reader would want to know.

Kernel-style conventions vary by team, but the shape is stable. This makes commit messages a natural target for AI generation.

Generating a commit message

The workflow, in a terminal:

```
git diff --staged | <assistant CLI> --template=commit-msg
```

Or, inside Copilot: click the “Generate commit message” button in the Source Control panel.

What the assistant sees: the entire staged diff. What it produces: a subject line plus a two-to-four-line body. What you do: read it, adjust the subject if the model overreached, commit.

When the generated message is wrong

- The diff was large and heterogeneous: the model wrote “various improvements” or “refactor code”. Fix: split the commit.

When the generated message is wrong

- The diff was large and heterogeneous: the model wrote “various improvements” or “refactor code”. Fix: split the commit.
- The subject blames the wrong file (“Update Cart.java” when the substantive change was in `Order.java`). Fix: rewrite the subject; the body is usually fine.

When the generated message is wrong

- The diff was large and heterogeneous: the model wrote “various improvements” or “refactor code”. Fix: split the commit.
- The subject blames the wrong file (“Update Cart.java” when the substantive change was in `Order.java`). Fix: rewrite the subject; the body is usually fine.
- The message says “Add feature X” but the code deletes a feature. Fix: reject and rewrite by hand.

The prompt behind the button

A defensible prompt looks like this:

```
You are a git commit message writer.
Given the staged diff, output:
  line 1: imperative subject, <=72 chars, no trailing period.
  line 2: blank.
  line 3+: 2--4 lines describing what and why.

Do NOT include "as an AI language model".
Do NOT include markdown fences.

Diff:
{{ staged_diff }}
```

Don't let it lie about intent

Commit messages are historical record. If the AI's subject line is wrong about *why* the change was made, future readers will believe it.

- The AI does not know your intent. It infers it from the diff.

Don't let it lie about intent

Commit messages are historical record. If the AI's subject line is wrong about *why* the change was made, future readers will believe it.

- The AI does not know your intent. It infers it from the diff.
- "Add null-check" may look like a defensive fix but you may have added it because a real crash happened.

Don't let it lie about intent

Commit messages are historical record. If the AI's subject line is wrong about *why* the change was made, future readers will believe it.

- The AI does not know your intent. It infers it from the diff.
- "Add null-check" may look like a defensive fix but you may have added it because a real crash happened.
- Add the *why* yourself in the body – the assistant will not.

PR summaries, changelogs, and branch names

PR summaries

Modern hosted-git platforms (GitHub, GitLab) include an "AI summary" button on pull requests. It reads the accumulated diff and produces:

- a one-paragraph description;

Same rules as commits: verify. A PR summary that is wrong about intent misleads reviewers.

PR summaries

Modern hosted-git platforms (GitHub, GitLab) include an "AI summary" button on pull requests. It reads the accumulated diff and produces:

- a one-paragraph description;
- a bulleted list of changed files and their purpose;

Same rules as commits: verify. A PR summary that is wrong about intent misleads reviewers.

PR summaries

Modern hosted-git platforms (GitHub, GitLab) include an "AI summary" button on pull requests. It reads the accumulated diff and produces:

- a one-paragraph description;
- a bulleted list of changed files and their purpose;
- (sometimes) a suggested test plan.

Same rules as commits: verify. A PR summary that is wrong about intent misleads reviewers.

Automated changelogs

For a release-tagged repo:

- Feed a range of commit messages into the assistant.

This works if your commit messages themselves are honest. If half of them are “fix stuff”, the changelog will be too. Rule of thumb: garbage-in, garbage-out applies to git history as much as anywhere else.

Automated changelogs

For a release-tagged repo:

- Feed a range of commit messages into the assistant.
- Ask for a changelog grouped by *features* / *fixes* / *docs* / *internals*.

This works if your commit messages themselves are honest. If half of them are “fix stuff”, the changelog will be too. Rule of thumb: garbage-in, garbage-out applies to git history as much as anywhere else.

Branch names

Branch naming is another small, structured task. A useful prompt:

> *Suggest a branch name for the change described by this issue:*
> *{{ issue title and body }}.*
> *Format: prefix/short-slug.*
> *Prefix is one of: feat, fix, chore, docs, refactor.*
> *Slug is kebab-case, $\leq$40 characters.*

Tiny task; saves you fifteen seconds and enforces a convention.

Branch management and merge conflicts

Rebase or merge?

The assistant is not going to solve the philosophical question for you. But it can help with the mechanics:

- "What is the correct git command to rebase my feature branch onto `main` keeping my three commits intact?"

These are look-up tasks that the assistant is very good at. Verify by reading the man page anyway; the assistant occasionally mixes up flags across git versions.

Rebase or merge?

The assistant is not going to solve the philosophical question for you. But it can help with the mechanics:

- "What is the correct git command to rebase my feature branch onto `main` keeping my three commits intact?"
- "I get '`fatal: refusing to merge unrelated histories`'; explain and give me the flag to fix it."

These are look-up tasks that the assistant is very good at. Verify by reading the man page anyway; the assistant occasionally mixes up flags across git versions.

Merge conflicts

A conflict block looks like:

```
<<<<<<< HEAD
    return items.stream().map(LineItem::amount).reduce(BigDecimal.ZERO, BigDecimal::add);
=======
    BigDecimal total = BigDecimal.ZERO;
    for (LineItem li : items) total = total.add(li.amount());
    return total;
>>>>>>> feature/loop-form
```

Neither side is wrong – they are stylistically different.

AI-assisted conflict resolution

Ask the assistant:

```
Resolve this merge conflict. Prefer the branch's variant if the
two sides are semantically equivalent. If they differ in behaviour,
explain the difference and do NOT auto-choose.

Conflict:
{{ the block above }}
```

The assistant usually gets "semantically equivalent" right for stylistic differences. For real behavioural differences (a different rounding mode, a missing null check), it should decline to auto-choose – and you must decide.

The dangerous merge

- Silent semantic changes: the two branches both look correct but produce different outputs on the same input.

Rule: after any AI-assisted merge, run the test suite before you commit. Not after you commit – *before*.

The dangerous merge

- Silent semantic changes: the two branches both look correct but produce different outputs on the same input.
- Renamed variables: side A renamed a variable; side B added uses of the old name. The assistant may resolve the text without noticing that side B is now broken.

Rule: after any AI-assisted merge, run the test suite before you commit. Not after you commit – *before*.

The dangerous merge

- Silent semantic changes: the two branches both look correct but produce different outputs on the same input.
- Renamed variables: side A renamed a variable; side B added uses of the old name. The assistant may resolve the text without noticing that side B is now broken.
- Deleted files vs edited files: side A deleted a file; side B modified it. The right answer is almost always project- specific.

Rule: after any AI-assisted merge, run the test suite before you commit. Not after you commit – *before*.

AI in CI/CD

AI code reviewers in the pipeline

Beyond the editor and the commit, AI is now baked into the CI/CD layer itself.

Modern PR workflows increasingly include an *automated AI reviewer* that comments on every pull request:

- GitHub Copilot code review, CodeRabbit, Graphite AI, Sourcery, and friends.

Trade-off. Catches trivial issues fast; noisier than a senior reviewer; occasionally hallucinates "bugs" that are not. Treat its comments the way you treat inline completions – read before accepting.

AI code reviewers in the pipeline

Beyond the editor and the commit, AI is now baked into the CI/CD layer itself.

Modern PR workflows increasingly include an *automated AI reviewer* that comments on every pull request:

- GitHub Copilot code review, CodeRabbit, Graphite AI, Sourcery, and friends.
- Runs as a bot – reads the diff, comments inline, sometimes suggests patches.

Trade-off. Catches trivial issues fast; noisier than a senior reviewer; occasionally hallucinates "bugs" that are not. Treat its comments the way you treat inline completions – read before accepting.

AI code reviewers in the pipeline

Beyond the editor and the commit, AI is now baked into the CI/CD layer itself.

Modern PR workflows increasingly include an *automated AI reviewer* that comments on every pull request:

- GitHub Copilot code review, CodeRabbit, Graphite AI, Sourcery, and friends.
- Runs as a bot – reads the diff, comments inline, sometimes suggests patches.
- You still need at least one human approval; the bot is a first pass, not the reviewer of record.

Trade-off. Catches trivial issues fast; noisier than a senior reviewer; occasionally hallucinates "bugs" that are not. Treat its comments the way you treat inline completions – read before accepting.

Wrap-up

What to internalise from Module 3

- Inline mode is a different UX from chat mode. Both matter.

What to internalise from Module 3

- Inline mode is a different UX from chat mode. Both matter.
- FIM lets the model use suffix context; open the right files before you type.

What to internalise from Module 3

- Inline mode is a different UX from chat mode. Both matter.
- FIM lets the model use suffix context; open the right files before you type.
- Boilerplate is the sweet spot: predictable, tedious, low-risk. Type the first instance yourself; let the assistant fill in the second.

What to internalise from Module 3

- Inline mode is a different UX from chat mode. Both matter.
- FIM lets the model use suffix context; open the right files before you type.
- Boilerplate is the sweet spot: predictable, tedious, low-risk. Type the first instance yourself; let the assistant fill in the second.
- AI-generated commit messages are useful, but the *why* is yours to write.

What to internalise from Module 3

- Inline mode is a different UX from chat mode. Both matter.
- FIM lets the model use suffix context; open the right files before you type.
- Boilerplate is the sweet spot: predictable, tedious, low-risk. Type the first instance yourself; let the assistant fill in the second.
- AI-generated commit messages are useful, but the *why* is yours to write.
- AI-assisted merges are safe only if you re-run tests.

Where we go next

Module 4 (Debugging and Refactoring) takes the workflow we just built and applies it to broken code and messy code.

Assignment 6 (Week 6) combines Module 3 with early Module 4 material: a negative-prompting exercise plus an autocomplete-driven refactor. The starter repository will be released with the assignment.

References

- Bavarian et al., *Efficient Training of Language Models to Fill in the Middle*, arXiv:2207.14255, 2022 – the FIM paper.

References

- Bavarian et al., *Efficient Training of Language Models to Fill in the Middle*, arXiv:2207.14255, 2022 – the FIM paper.
- GitHub Copilot documentation.

References

- Bavarian et al., *Efficient Training of Language Models to Fill in the Middle*, arXiv:2207.14255, 2022 – the FIM paper.
- GitHub Copilot documentation.
- Conventional Commits specification, `conventionalcommits.org`.

Questions?

nvk@iitm.ac.in