# Module 3: AI for Code Generation and Autocompletion

CS5013 – Programming with AI
Lecture Notes

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

## Before we begin

Module 2 was about the words you type into a chat panel. Module 3 is about the words that appear on your screen *without* you typing them – the ghost-text completions in the editor – and about the small helper tasks the assistant can do at the seams of your git workflow.

## 1 Two modes of assistance

There are two very different UX modes for an AI coding assistant:

1. **Chat mode.** You type a natural-language prompt into a panel; the assistant returns a block of code. You copy-paste (or "apply") it into a file. Everything in Module 2 was about this mode.
2. **Inline mode.** You type code in the editor; the assistant proposes completions as *ghost text* inline. You accept a completion with `Tab`; you dismiss it by continuing to type.

Both modes use the same underlying language model. They differ in how the prompt is constructed (chat: what you type; inline: your file plus editor context), in the tightness of the feedback loop (chat: seconds; inline: milliseconds), and in the type of errors they are prone to (chat: over-generation; inline: plausible-but-wrong nearby completions).

**When to use which**

Use chat when the task is large enough that you would want to describe it in a paragraph. Use inline when you can express the task by typing the first two lines of code. If in doubt, start inline; escalate to chat when the inline completion keeps missing the mark.

## 2 Autocompletion mechanics

### 2.1 Fill-in-the-Middle

Editor completions are not simply "predict what comes after the cursor". The cursor is often in the middle of a file, and the completion has to respect both the code before it and the code after it – the enclosing method's return type, the closing brace two lines below, the sibling methods it should match.

To handle this, code-specialised models are trained on a task called *Fill-in-the-Middle* (FIM). Training data is arranged as `(prefix, suffix, middle)` triples; the model learns to predict the middle given both the prefix and the suffix. Bavarian et al. (arXiv:2207.14255) describe the training recipe. This is why an inline completion can close the right bracket, match the right return type, and even name a variable consistently with a line that appears further down the file.

### 2.2 What the editor plugin actually sends

If you inspect the network traffic (some IDEs let you), you will see that the "prompt" sent for a single completion is not just the current file. It typically includes:

- the *current file*, split into prefix (before the cursor) and suffix (after the cursor);
- some number of *recently opened files*, ranked by heuristic relevance – the plugin is guessing at what context you would find useful;
- *metadata* about the environment (JDK version, framework markers such as `pom.xml` or `build.gradle`);
- sometimes: *retrieved chunks* from your repo (repo-level indexing has become standard in the last couple of years).

This has a practical consequence: **the completion depends on what you have open**. If you want the assistant to fill in a mapper that respects the target class, open the target class in another tab. If you want the completion to imitate an existing method, open the existing method.

**Curate your tabs**

Before starting an autocompletion-heavy task, open two or three example files that show the pattern you want. Close unrelated files. The completion will be visibly better.

### 2.3 The interaction rhythm

The round-trip time for a completion is small enough that you can keep typing while suggestions come and go – typically 300–800ms in a healthy setup. The habit to develop is:

1. Type just enough to make the intent obvious (a signature and maybe a comment).
2. Wait one second.
3. Read the ghost text. Accept with `Tab` if it looks right. If it does not, keep typing over it.
4. If you accept, run the code (or at least compile) before moving on.

The last step is the important one. Ghost text is fast; verification is not. The step you skip is the step you regret.

### 2.4 The comment-as-prompt pattern

A Javadoc or single-line comment placed just above an empty method body is the most reliable inline prompt you have. The signature pins the return type; the comment carries the constraints; the assistant merges both into its completion.

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

The pattern is especially valuable when the "obvious" completion would violate a rule you care about – "do NOT throw" in the example above, or "preserve insertion order", or "use the project's existing `DateTimeFormatter` constant". Stating the rule in the comment is usually enough.

Should you delete the comment after the completion is in? Only if the code reads clearly without it. Often the comment was useful *documentation*, not just a prompt to the model – keep it in that case.

### 2.5 When completions fire, and when they don't

Editor plugins do not query the model continuously. Completions typically fire on:

- pressing a newline;
- pressing `.` after an identifier;
- a ~150ms pause in typing;
- an explicit shortcut (Alt+\, Ctrl+Enter).

If completions are not appearing at all, one of a small set of causes is usually responsible: the file is larger than the plugin's prefix cap; the extension has been disabled for that language; a corporate proxy is making the round-trip too slow to complete before you resume typing. Check those three before assuming the plugin itself is broken.

## 3 IDE integration: a working setup

The rest of the module – and the hands-on session – assumes the following stack:

- VS Code with the *Extension Pack for Java*.
- Either *GitHub Copilot* (which provides both inline suggestions and Copilot Chat) or *Claude Code* (chat in the terminal plus a limited inline mode).
- A Java 17+ JDK on your `PATH`.

Once the extension is installed and authenticated, three surfaces are available:

1. **Inline suggestions.** Ghost text as you type; accepted with Tab.
2. **Copilot Chat.** A side panel where you ask natural-language questions about the current file: "explain this method", "write a test for this", "what does the error on line 42 mean?"
3. **Inline chat.** Ctrl+I or Cmd+I (mac) opens a chat prompt anchored to the current selection; the response replaces or edits the selection in place.

Different surfaces suit different tasks. Use inline for local edits (one line to a handful); use Copilot Chat for questions that would otherwise involve pasting into a browser; use inline chat when you want the answer applied to a specific region rather than shown as text you have to copy.

### A first-day workflow

For your very first Java file with the assistant enabled, try this loop:

1. Type a method signature and a Javadoc comment describing what the method should do.
2. Wait one second. Accept the inline completion if it looks right.
3. Compile. Run.
4. If it does not compile, either fix it in place with Ctrl+I/Cmd+I ("fix the compile error") or throw the completion away and try again with a more specific comment.

Nothing in this workflow is new. The only thing that changed is how fast steps 1–3 have become.

## 4 Boilerplate generation

Boilerplate is code that is *predictable* (has a well-known shape), *tedious* (expensive to write by hand) and *low-risk* (easy to verify by reading). This description covers records/DTOs, mappers, controllers, config classes, and a fair chunk of the code you will write in a Java project.

### 4.1 Records and DTOs

Java 17 records are the friendliest possible autocompletion target. Type the record header and let the assistant fill in the field list.

```
// You type:
public record Address(String street, String city,

// The assistant completes:
                      String state, String pin) {}
```

Read it before you accept. Two common gotchas:

- The assistant hallucinates a field (`country`) that your domain does not have, because most `Address` records it has seen have a country.
- It picks the wrong type (`int` for `pin` when you use `String` – Indian PINs can have leading zeros, but the model has seen many U.S. examples with an integer ZIP code).

### 4.2 Mappers

A mapper from one class to another is essentially a field-by-field copy. This is where autocompletion feels like magic – and where hallucinated field names bite.

**Hallucinated fields**

The assistant may suggest `u.getEmail()` because most `User` classes it has seen expose a `getEmail()` method. If your `User` is a record with `email()`, the code will not compile. This is the fastest way to catch the mistake: `javac`. If your CI runs on every commit, you catch it there anyway; but letting a bad mapper drift into a pull request wastes reviewer time.

### 4.3 Some known failure modes

Even in the sweet spot of boilerplate, autocompletion can quietly produce wrong code. Some known patterns:

- **Broken equals contract.** A suggested `equals` may use `getClass()` on one side and `instanceof` on the other, silently violating symmetry (`a.equals(b) != b.equals(a)`).
- **Missing `@Override`.** The assistant writes `public boolean equals(MyClass other)` instead of `equals(Object o)`; the compiler treats this as an overload, and `HashMap` never calls it. The `@Override` annotation is what would have flagged the mistake – so its absence is the whole bug.
- **Non-transitive Comparator.** A tie-break shortcut of the form "return 0 if same department" makes the comparator non-transitive. A `TreeSet` that uses it misbehaves silently – elements appear absent from `contains()` calls.
- **Copy-drift.** In a chain of `Comparator.comparing(...).thenComparing(...)`, the assistant's second key extractor may repeat the first, so the comparator sorts by department twice and leaves the intended secondary field unused.

The unifying feature: all four compile, some pass a naive test, and only careful reading (or a stringent test) catches them. Read boilerplate; do not skim it.

## 5 Regex generation

Regular expressions are boilerplate at its most extreme: high skill cost to write, low skill cost to *read* if you have test strings. This asymmetry makes them one of the most reliable uses of LLM generation.

### 5.1 A quick refresher

A regex is a compact pattern that describes a set of strings. The alphabet is small:

- **Literals**: `a`, `7`, `-`.
- **Character classes**: `\d` (digit), `\w` (word character), `[a-z]`, `[^0-9]` (negated).
- **Quantifiers**: `*` (zero or more), `+` (one or more), `?` (optional), `{n,m}` (bounded).
- **Anchors**: `^` (start of string), `$` (end of string).
- **Groups**: `(...)` captures for later extraction, `(?:...)` groups without capturing.

### 5.2 The three verbs in Java

Java's regex engine lives in `java.util.regex`. Two classes matter: `Pattern` (a compiled regex) and `Matcher` (that regex applied to a specific input). The three matching verbs on `Matcher` behave differently:

```
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? false here
m.find(); // is there a match anywhere? true (and advances)
m.lookingAt(); // does the match start at 0? false
```

Three practical points that trip students up:

- Java strings double every backslash. The literal string `"\\d"` contains the two characters that form the regex `\d`.
- Compile the `Pattern` once – typically as a `static final` field – and reuse it. `Matcher` is per-input and not thread-safe.
- Convenience methods on `String` – `s.matches(regex)`, `s.split(regex)`, `s.replaceAll(regex, repl)` – recompile the pattern on every call. Fine for one-off use, wasteful in a loop.

### 5.3 Generating a regex with the assistant

The typical prompt is one line: "Java regex that matches an ISO-8601 date (`YYYY-MM-DD`), anchored at both ends, no capturing groups." The assistant produces the pattern and, in most cases, a short natural-language explanation:

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

**Regex verification checklist**

Always test a generated regex against at least three positive examples and three negative examples before committing it. Watch for greedy-vs-lazy mistakes (`.*` vs `.*?`) in longer patterns. If the assistant cannot explain its own regex back to you on request, treat that as a signal – rewrite it, or at least test harder.

## 6 Version control basics: a working intro to git

Before we ask an assistant to *write* git commands, we need a shared vocabulary. If you have used git before, this section is a refresher; if you have not, it is what you need to follow the rest of the module.

### Why version control?

Four things a version-control system gives you that shared drives and zip-of-the-day backup schemes do not:

- **Undo without fear.** Every state you commit is recoverable, even after further edits.
- **Blame.** `git blame` plus the commit message answers "who changed this line, when, and why?".
- **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. If you know git, you will pick up any other VCS quickly.

### Git's four places

Every git operation moves work between four locations:

- **Working directory**: the files your editor sees.
- **Staging area (index)**: files you have marked for the next commit.
- **Local repository**: the `.git` folder – your committed history on disk.
- **Remote**: another copy of the repository, usually hosted (GitHub, GitLab, Bitbucket).

The three arrows between them are the three commands you use most:

`git add` → `git commit` → `git push`

### 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
```

Two more will show up when we discuss AI-assisted commit messages: `git diff -staged`, which shows only what is about to be committed, and `git status`, which is the first command to run any time you feel lost.

### Branches and merges

A branch is a movable pointer to a commit – cheap to create, cheap to throw away. The everyday branch commands:

```
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 the history topology – a merge commit shows that a branch was joined. *Rebase* rewrites history to look linear – cleaner log, but the rewritten commits have new hashes. Rule of thumb: rebase local branches you have not pushed yet; merge shared ones.

### Remotes, pull requests, conflicts

Three commands connect your local repo to a remote:

```
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 the tracked remote
```

A *pull request* (PR) is a hosted-git artefact: you ask to merge your branch into `main`; reviewers comment in the UI; when they approve and CI is green, the PR is merged.

A *conflict* arises when git cannot decide which side of a divergent edit to keep. It marks the file with `<<<<< HEAD`, `=======`, `>>>>> other-branch` and stops. You edit the markers away by hand, `git add` the resolved file, and continue the merge or rebase.

# 7 AI-assisted version control

## 7.1 Generating commit messages

Modern git hosts and IDEs expose a "generate commit message" button. Under the hood, the prompt is roughly:

```
You are a git commit message writer.
Given the staged diff, output:
  line 1: imperative subject, <= 72 chars, no period.
  line 2: blank.
  line 3+: 2-4 lines describing what and why.
Do NOT include markdown fences.
Diff:
{{ staged_diff }}
```

The model reads the diff and produces the message. The good habit is to read the message, adjust the subject if needed, and *always* add the "why" by hand.

**The "why" is yours**

The model can see what changed. It cannot see *why* you changed it. A subject of "Add null-check to `Cart.total()`" is factually correct but hides the story: was it a defensive fix, or did a real crash happen, or did a lint rule flag it? Add the story to the body. Future readers will thank you.

## 7.2 When the generated message is wrong

Two patterns to watch for:

- **Vague**: for a large heterogeneous diff, the model writes "various improvements" or "refactor code". The fix is not to rewrite the message; it is to split the commit into smaller, coherent pieces.
- **Wrong intent**: the model says "Add feature X" when the code actually removes feature X. The fix is manual rewriting. Do not trust the assistant on intent claims.

## 7.3 PR summaries and changelogs

The same technique scales. A pull-request summary is generated from the accumulated diff of a branch; a changelog is generated from the sequence of commit messages between two release tags.

- **PR summaries** are useful as a starting point for a review; a reviewer with a well-written PR summary reads faster and misses fewer issues.
- **Changelogs** are only as good as the commit history underneath them. If half your commits are "fix stuff", so is the changelog.

## 7.4 Branch management

Branch naming is a small, structured task that the assistant handles well. A prompt like "suggest a branch name for the change described by this issue; format `prefix/kebab-slug`; prefix in `[feat, fix, chore, docs, refactor]`; slug $\leq$ 40 chars" gives you a consistent name in a second.

For merge conflicts, the assistant can classify a conflict as "purely stylistic" vs "semantically different". If the two sides are semantically equivalent (e.g., one uses a stream and the other uses a for-loop), the assistant will happily pick one. If they are semantically different, it should refuse – and if it does not, you must.

**Rerun tests after every AI-assisted merge**

An AI-resolved conflict can compile and still be wrong. The classic case: side A renamed a variable, side B added uses of the old name. The assistant resolves the visible textual conflict and misses that side B's newly-added uses are now broken references. Always run the tests before you commit the merge.

## 8 AI in CI/CD

Beyond the editor and the commit, AI has begun to appear inside the CI/CD pipeline itself. The dominant use is the *automated code reviewer*: a bot that comments on every pull request as soon as it opens.

Concrete tools you will encounter in industry:

- GitHub Copilot code review.
- CodeRabbit.
- Graphite AI.
- Sourcery.

The interaction pattern is the same across all of them. The bot reads the diff, comments inline on lines it thinks warrant attention, and sometimes suggests a patch that a maintainer can apply with one click.

**How to think about the bot**

Automated AI reviewers are a *first pass*, not a replacement for human review. They catch trivial issues fast – a missing null check, an unused import, an obvious typo – and they occasionally hallucinate "bugs" that are not. Treat their comments the same way you treat inline completions: read before accepting. Your PR still needs at least one human approval before it merges.

## 9 Habits to carry forward

1. Open the right files before you start typing. Inline completions see them.
2. Type the first instance of a pattern yourself. Let the assistant fill in the second. Read carefully before accepting the third.
3. Compile before you commit anything the assistant wrote – especially mappers.
4. Let the assistant draft your commit messages, but always add the "why" by hand.
5. After an AI-assisted merge, run the tests *before* pushing.

Module 4 (Debugging and Refactoring) picks up from here. The workflow you just built will be aimed at broken code and messy code.

## Further reading

- Bavarian et al., *Efficient Training of Language Models to Fill in the Middle*, arXiv:2207.14255, 2022 – the FIM paper.
- Conventional Commits specification, `conventionalcommits.org` – structured commit-message format widely used with AI-generated messages.
- GitHub Copilot documentation for current UX conventions in VS Code.