guide # tooling

Using a Coding Agent

TCSS 460 — Client/Server Programming

AI coding agents are transforming how software gets built. They can read your project, write code, run commands, and iterate on errors — all from a natural language prompt. This guide covers what coding agents are, when they help, when they hurt, and how to use them responsibly in this course.

---

1 What Is a Coding Agent?

A **coding agent** is an AI tool that operates directly inside your development environment. Unlike a general-purpose chatbot, a coding agent has access to your project files, can read and write code, run terminal commands, and observe the results. It works *with* your codebase, not in isolation.

1.1 More Than a Chatbot

You have probably used a conversational AI tool to ask programming questions. You paste code in, describe a problem, and get an answer back. That workflow has a fundamental limitation: the AI only knows what you paste into the chat window. It cannot see the rest of your project, your configuration files, your database schema, or the error messages in your terminal.

A coding agent removes that limitation. When you ask it to "add a GET route for `/users/:id`," it can:

- **Read** your existing route files to understand the project structure
- **Write** a new route handler that follows the patterns already in your code
- **Run** the server to check for errors
- **Fix** issues it finds and try again

This loop — read, write, run, fix — is what makes an agent different from a chatbot. It does not just suggest code; it *executes* a development task.

1.2 Context Is the Key Difference

The most important concept to understand about coding agents is **context**. When you open a conversation with a chatbot, it starts with a blank slate. When you open a coding agent inside your project, it starts with your entire codebase.

That context changes everything. The agent can:

- Match the coding style and conventions already in your project
- Import from modules that actually exist in your file tree
- Reference your `package.json` dependencies, your `tsconfig.json` settings, and your database schema
- Understand how your Express routes, middleware, and error handlers fit together

Without context, AI-generated code often looks plausible but does not integrate. With context, it can produce code that actually belongs in your project.

> **Gen AI & Learning: Why Context Matters**
>
> When you ask an AI a question out of context — "How do I query a database in TypeScript?" — it gives you a generic answer. When you ask a coding agent the same question inside your project, it gives you an answer that uses *your* database library, *your* connection setup, and *your* naming conventions. Learning to provide good context is a skill that transfers directly to working with any AI tool, now and in the future.

1.3 What a Coding Agent Is NOT

It is worth being explicit about what coding agents cannot do:

- **They are not a replacement for understanding.** An agent can generate code faster than you can type it, but if you cannot read and evaluate what it produces, you have gained nothing.
- **They are not always correct.** Agents produce plausible-looking code that may contain subtle bugs, security issues, or logic errors. They can confidently generate code that compiles and runs but does the wrong thing.
- **They are not deterministic.** Ask the same question twice and you may get different answers. The same prompt can produce working code one time and broken code the next.
- **They do not understand your intent.** They predict what code *should* look like based on patterns. They do not know what your application is supposed to *do*.

---

2 Tokens, Context Windows, and Usage Limits

Before you start using a coding agent, you need to understand the resource model behind it. Every AI tool — free or paid — has limits on how much text it can process at once and how much you can use over time. Understanding these mechanics will help you work within the limits and avoid frustrating surprises.

2.1 What Are Tokens?

AI models do not process text word by word. They break text into **tokens** — chunks that are roughly ¾ of a word on average. As a practical rule of thumb:

Measure	Approximate Token Count
1 word	~1.3 tokens
1 line of code	~10-15 tokens
40 lines of code	~500 tokens
750 words of English text	~1,000 tokens
A typical Express route handler (50 lines)	~600-800 tokens
A full guide like this one	~5,000-8,000 tokens

Every interaction with a coding agent consumes tokens in two directions:

- **Input tokens** — everything the agent reads: your prompt, the files it opens, your project context, the conversation history
- **Output tokens** — everything the agent generates: code, explanations, terminal commands

Both count toward your usage limits.

2.2 What Is a Context Window?

The **context window** is the total amount of text an AI model can hold in its working memory at once — your prompt, the conversation history, every file it has read, and its own responses.

Think of it as the model's desk: everything it is currently thinking about must fit on the desk.

As of early 2026, context windows vary significantly across tools:

Tool	Context Window	What That Means
Claude (Opus 4.6 / Sonnet 4.6)	1M tokens	~50,000-100,000 lines of code — a medium production codebase
Google Gemini	1M tokens	Similar capacity to Claude
GitHub Copilot	Varies by model	Typically 128K-200K tokens
Cursor	Varies by model	Typically 128K-200K tokens

A larger context window means the agent can hold more of your project in memory simultaneously. With a 1M token window, an agent can read your entire Express project — every route, every test, every configuration file — and maintain that context throughout the conversation. With a 128K window, the agent needs to be more selective about what it reads.

2.3 Why Context Window Size Matters

A larger context window is not automatically better. Two things happen as context grows:

1. **Better coherence across files.** The agent can see how your route handler connects to your middleware, which connects to your database layer. It does not lose track of a decision made earlier in the conversation. Fewer "I forgot what you said earlier" moments.
2. **Diminished focus.** Research shows that as the context window fills up, the model's ability to recall specific details — especially from the middle of the context — degrades. This is sometimes called **context rot** or the "lost in the middle" problem. A model with 100K tokens of context may give sharper answers than the same model with 800K tokens of context, because there is less noise competing for attention.

The practical takeaway: **curate what goes into context.** Do not dump your entire codebase into the window just because it fits. A well-organized project where the agent reads only the relevant files will produce better results than a project where the agent reads everything.

This connects directly to context engineering (Section 6) — the practice of structuring your project so the agent naturally reads what matters and ignores what does not.

2.4 Usage Limits and Cost

Every tool has limits on how much you can use it. Understanding these limits prevents the frustrating experience of running out mid-task.

Tool	Limit Model	What Happens When You Hit It
Gemini CLI	1,000 requests/day, 60/minute	Requests are rejected until the limit resets
GitHub Copilot (student)	300 premium requests/month	Basic completions continue; agent mode and premium features pause
Cursor (student Pro)	Unlimited Auto mode; $20 credits for frontier models	Auto mode keeps working; frontier model access pauses when credits run out
Claude Code (Pro, $20/mo)	~10-40 prompts per 5-hour rolling window (shared with claude.ai)	Wait for window to reset, or enable extra usage at API rates

The key insight: **agent mode consumes far more tokens than simple completions.** A single inline completion might use a few hundred tokens. An agent session that reads 10 files, generates code, runs tests, and iterates on errors might consume tens of thousands of tokens. If you are on a limited plan, be intentional about when you use agent mode versus simpler interactions.

Practical Strategies for Managing Limits

- **Use the right mode for the task.** Ask mode or inline completions for quick questions. Agent mode for multi-step tasks that justify the token cost.
- **Scope your requests.** "Add validation to `src/routes/auth.ts`" consumes fewer tokens than "Review my entire project and suggest improvements."
- **Close and restart.** Long conversations accumulate context. If a conversation has drifted, start a fresh one — it will be cheaper and often produce better results.
- **Know your reset schedule.** Gemini resets daily, Copilot monthly, Claude Code every 5 hours. Plan heavy work around your resets.

2.5 What Happens During Compaction

When a conversation in a coding agent grows too long to fit in the context window, the tool performs **compaction** — it summarizes or drops older parts of the conversation to make room for new content. This is invisible to you, but it has real consequences:

- The agent may "forget" decisions made earlier in the conversation
- Code it generated ten exchanges ago may no longer be in its memory
- You may need to re-explain constraints or re-reference files

Larger context windows reduce how often compaction happens (Claude reported a 15% decrease in compaction events after shipping the 1M window), but they do not eliminate it for very long sessions.

When you notice the agent losing track of earlier context — repeating questions, contradicting earlier decisions, or generating code that conflicts with what it wrote before — it has likely compacted. The best fix is to start a fresh conversation with a clear, focused prompt rather than trying to re-explain everything.

---

3 When to Use a Coding Agent

Coding agents excel at specific categories of tasks. Knowing when to reach for an agent — and when to work without one — is a professional skill you will develop throughout this course.

3.1 Scaffolding Boilerplate

Every web project has repetitive setup code: route definitions, middleware configuration, database connection boilerplate, error handling patterns. This code follows well-known conventions and does not require creative problem-solving.

An agent is ideal for this. Instead of typing out the same Express route handler skeleton for the tenth time, you can describe what you need:

```
Vague prompt (less effective):
"Make a route for users"

Specific prompt (more effective):
"Add a GET route at /users/:id that queries the database
for a user by ID and returns it as JSON. Follow the same
pattern as the existing routes in src/routes/."
```

The specific prompt gives the agent enough context to produce code that fits your project. The vague prompt forces the agent to guess at dozens of decisions you should be making.

3.2 Understanding Unfamiliar Code

When you encounter a codebase you did not write — a starter repo, an open-source library, a teammate's module — a coding agent can help you understand it. You can ask questions like:

- "Explain what this middleware function does, step by step"
- "How does the authentication flow work in this project?"
- "What happens when a request hits the `/api/movies` endpoint?"

Because the agent can see the entire project, it can trace through imports, follow function calls across files, and give you a coherent explanation of how the pieces fit together.

Try It Yourself

When you receive the course starter repo, ask your coding agent to explain the project structure before you start writing any code. Ask it questions like "Walk me through what happens when a request comes in to the server." This is a fast way to get oriented.

3.3 Debugging Error Messages

Error messages in TypeScript and Node.js can be cryptic, especially when you are new to the ecosystem. A coding agent can read the error, look at the relevant source files, and explain what went wrong.

```
Effective debugging prompt:
"I'm getting this error when I run the server:

TypeError: Cannot read properties of undefined (reading 'id')
    at /src/routes/users.ts:15:28

Look at the route handler and tell me what's wrong."
```

The agent can examine line 15 of `users.ts`, check how `req.params` is being used, and identify whether the issue is a missing parameter, a typo, or a middleware ordering problem.

3.4 Generating Tests

Writing test cases is one of the most effective uses of a coding agent — but *how* you ask for tests matters enormously.

> **Generate Tests from Documentation, Not Code**

When an agent reads your implementation and generates tests from it, the tests will reflect what your code *does* — including any bugs. If your route handler returns a 200 when it should return a 201, the agent will happily write a test that asserts 200 and it will pass. The tests become a mirror of your mistakes, not a check against them.

The better approach: **generate tests from your API documentation or specification, not from the code itself.** Give the agent your Swagger/OpenAPI spec, your route descriptions from the assignment, or your API design document — and ask it to write tests based on what the API *should* do:

- "Here is the OpenAPI spec for the `/users/:id` endpoint. Write tests covering: valid ID returns 200 with user data, non-existent ID returns 404, non-numeric ID returns 400."
- "Based on this route description, write tests for the validation rules."

When tests are written from the spec, they act as an independent check against your implementation. If a test fails, the bug might be in your code, not in the test — which is exactly what you want.

Separate the Test Writer from the Code

For the strongest results, consider pulling your API documentation out of the project entirely and giving it to an agent in a separate, clean context — a fresh project with no access to your implementation code. The agent generates tests purely from the documented behavior. You then bring those tests back into your project and run them. This eliminates any chance of the agent "cheating" by reading your implementation and writing tests that match its quirks.

This pattern — writing tests against a specification rather than an implementation — is the same principle behind contract testing and test-driven development (TDD) in professional software engineering.

Be careful when you bring those tests into your project. When spec-generated tests fail against your code, the agent's instinct is to "fix" the tests to match your implementation — which defeats the entire purpose. A failing test means either your code is wrong, your documentation is wrong, or both. That is a judgment call that *you* have to make. The agent cannot tell the difference between a test that is wrong and a test that caught a real bug. This step requires heavy babysitting — review every proposed test change individually and ask yourself whether the test or the code should change.

This is where *your* understanding matters most. You need to know what the requirements and intent are, and you need to understand how both the code and the tests work. Without that, you cannot make the call. An agent can generate code and generate tests, but only a developer who understands the problem can decide which one is right when they disagree.

You still need to review the generated tests to make sure they actually verify the right behavior, but starting from documentation rather than code gives you a much stronger safety net.

Gen AI & Learning: Tests as a Learning Tool

Reading AI-generated tests is a surprisingly good way to learn. When an agent writes tests from your API specification, it reveals edge cases and assumptions you may not have considered in your implementation. If you look at a generated test and think "I didn't handle that case," you have just found a real bug — not a test that mirrors one. Use generated tests as a starting point, then improve them based on your understanding.

---

4 When NOT to Use a Coding Agent

There are situations where using a coding agent actively works against you. Recognizing these situations is just as important as knowing when to use one.

4.1 When You Do Not Understand What It Generates

This is the single most important rule in this guide.

If a coding agent generates code and you cannot explain what each line does, **do not use that code**. Pasting in code you do not understand creates three problems:

1. **You cannot debug it.** When it breaks — and it will — you are stuck. You cannot fix code you do not understand.
2. **You cannot extend it.** The next feature needs to build on this code. If you do not know how it works, you cannot modify it.
3. **You do not learn.** The point of this course is to develop skills. Submitting code you do not understand bypasses the learning entirely.

> **The Explanation Test**
>
> Before you commit any AI-generated code, ask yourself: "Could I explain this code to a classmate?" If the answer is no, you need to understand it first. Ask the agent to explain it. Read the documentation for the APIs it uses. Rewrite it in your own style. Only then should you use it.

4.2 When the Assignment Is Testing YOUR Skills

Some assignments in this course are explicitly designed to verify that *you* can perform a specific task. Check-off assignments, for example, require you to demonstrate a skill in person. If you used an agent to generate the code without understanding it, the check-off conversation will make that obvious very quickly.

The assignments will make clear where AI tools are encouraged, restricted, or somewhere in between. When in doubt, ask.

4.3 Academic Integrity Boundaries

Academic Integrity Policy

Using AI coding agents is **permitted and encouraged** in TCSS 460, with conditions:

1. **You must understand every line of code you submit.** If you cannot explain it, you should not submit it.
2. **You must disclose AI tool usage** when the assignment requires it.
3. **You are responsible** for the correctness, quality, and originality of your submission — regardless of how it was produced.
4. **Individual assignments test individual skills.** Using an agent to bypass the learning objective of an individual assignment is an academic integrity violation, even if the tool is generally permitted.

The goal is not to prevent you from using these tools. The goal is to make sure you *learn* while using them. There is a difference between using an agent to accelerate your work and using one to avoid doing the work.

Think of it this way: a calculator does not help you on a math exam if you do not understand the math. Similarly, a coding agent does not help you become a better developer if you do not understand the code.

---

5 Prompt Engineering — The Starting Point

The first generation of advice for working with AI tools focused on **prompt engineering** — the art of crafting the right question to get a useful answer. This is still a valuable skill, especially for one-off interactions and when you are working in a tool or context where the agent does not have deep project knowledge. The quality of what a coding agent produces depends heavily on what you tell it.

5.1 Give Context

The single most effective thing you can do is tell the agent what you are building and where it fits in your project. Compare these two prompts:

```
Without context:
"Write a function to validate an email"

With context:
"I'm building an Express API for user registration.
Add email validation to the POST /auth/register route.
```

```
The route handler is in src/routes/auth.ts.
Validation should reject empty strings, missing @ symbol,
and emails longer than 255 characters.
Return a 400 status with a descriptive error message."
```

The second prompt gives the agent:

- **Project type** (Express API)
- **Location** (where the code goes)
- **Requirements** (what to validate)
- **Behavior** (what to return on failure)

This is not just about getting better AI output. Writing clear prompts forces you to think through the requirements before you write the code — which is a good engineering practice with or without AI.

5.2 Be Specific About What You Want

Agents respond well to concrete instructions. Instead of broad requests, break your task into specific actions:

Instead of...	Try...
"Make the API better"	"Add input validation to the POST /movies route"
"Fix the bugs"	"The server returns 500 when the movie ID is not a number. Add validation to return 400 instead."
"Write some tests"	"Write tests for GET /users/:id covering: valid ID returns 200, non-existent ID returns 404, non-numeric ID returns 400"
"Help me with the database"	"Add a Prisma query to find all movies where the title contains the search term, case-insensitive"

Notice the pattern: effective prompts specify the **what**, the **where**, and the **expected behavior**.

5.3 Review Everything It Produces

This point cannot be overstated. **You are responsible for every line of code in your submission**, whether you typed it or an agent generated it.

When an agent produces code, review it the same way you would review a pull request from a teammate:

1. **Read every line.** Do not just check if it runs — check if it is correct.
2. **Look for hardcoded values** that should be configurable.
3. **Check error handling.** Does it handle edge cases? What happens with bad input?
4. **Verify imports.** Does it import from modules that exist in your project?
5. **Run it.** Does it actually work, or just look like it should?
6. **Test edge cases.** The agent tested the happy path. What about empty input, null values, or unexpected types?

> **Gen AI & Learning: Code Review as a Skill**
>
> Reviewing AI-generated code is rapidly becoming one of the most important skills in software development. In professional settings, developers increasingly spend their time evaluating, testing, and refining code produced by AI tools rather than writing every line from scratch. The review skills you develop in this course transfer directly to that workflow.

5.4 Iterate, Don't Start Over

When an agent produces something that is close but not quite right, refine it rather than starting a new conversation. The agent remembers the context of your current session:

- "That route handler is good, but change the error response to include a `message` field"
- "The query works, but it should also filter out inactive users"
- "Add input validation before the database call"

Each iteration builds on the previous result and keeps the agent focused. Starting a new conversation throws away all that context.

5.5 The Limits of Prompt Engineering

Prompt engineering works — but notice the overhead. Every new conversation, you re-explain your project, your conventions, your file structure, and your intent. The long prompt in Section 5.1 is effective *because* it manually supplies context that the agent does not have. You are doing the agent's homework for it, one prompt at a time.

What if the agent already knew all of that? What if the project itself taught the agent how to work within it? That is the shift from prompt engineering to context engineering.

---

6 Context Engineering — The Next Step

In mid-2025, AI researchers and practitioners started using the term **context engineering** to describe a different approach to working with AI tools. Instead of crafting the perfect prompt for every interaction, you invest in the *environment* the agent operates in — so that even simple prompts produce high-quality results.

The analogy: prompt engineering is giving directions to a stranger on the street. Context engineering is onboarding a new team member — giving them access to the codebase, the documentation, the team conventions, and the project goals so that when you say "add email validation," they know exactly where it goes, what patterns to follow, and what the API should return on failure.

6.1 Project-Level Context

Every AI coding tool — regardless of which one you use — reads your project. The better organized your project is, the better the agent's output will be. This is context engineering at its most fundamental, and it is also just good software engineering:

Project Artifact	What It Teaches the Agent
Clear directory structure	Where new code should go (`src/routes/`, `src/middleware/`, `tests/`)
Consistent naming conventions	What to name new files, functions, and variables
OpenAPI / Swagger spec	What every endpoint expects and returns — the contract
README	What the project does, how to run it, what the architecture looks like
`.editorconfig`	Indentation, line endings, file encoding — mechanical consistency

Project Artifact	What It Teaches the Agent
Existing code patterns	How error handling works, how routes are structured, how tests are written
`package.json` **scripts**	How to build, test, lint, and run the project

None of these artifacts are AI-specific. They are things you would create for human teammates. The difference is that AI agents consume them automatically and immediately — a well-structured project gives every agent that opens it a head start.

6.2 Tool-Specific Configuration

Most AI coding tools offer a way to persist project-level instructions so the agent reads them at the start of every session. The names differ — Cursor uses rules files, Claude Code uses project documentation files, GitHub Copilot reads your repository structure and custom instructions — but the principle is the same: you write down your conventions, your preferences, and your architectural decisions *once*, and the agent applies them to every task.

If your tool supports this, use it. The specifics vary by tool and evolve quickly, so check your tool's documentation for the current approach. The investment pays off immediately — you stop repeating the same instructions in every prompt.

6.3 Intent Over Instructions

The deeper idea behind context engineering is capturing *why* you are building something, not just *what*. An agent that knows your project structure can generate code that compiles. An agent that knows your *intent* can generate code that solves the right problem.

Intent shows up in places you might not expect:

- **API documentation** that explains the *purpose* of an endpoint, not just its parameters
- **Descriptive commit messages** that explain why a change was made
- **Well-named variables and functions** that reveal the developer's reasoning
- **Comments on non-obvious decisions** — not "increment counter" but "retry up to 3 times because the upstream API rate-limits at 10 req/s"

When these artifacts exist in your project, every AI tool benefits — because they all read your codebase, and intent-rich code teaches the agent what you care about.

6.4 The Payoff

Here is the practical difference. With prompt engineering alone, you write prompts like this every time:

```
"I'm building an Express API for user registration.
Add email validation to the POST /auth/register route.
The route handler is in src/routes/auth.ts.
Validation should reject empty strings, missing @ symbol,
and emails longer than 255 characters.
Return a 400 status with a descriptive error message."
```

With good context engineering — a well-structured project, clear documentation, and tool-specific configuration — the same task becomes:

```
"Add email validation to the register route."
```

The agent already knows you are building an Express API. It already knows where the route handler lives. It already knows your validation patterns and error response format because it has seen them throughout your codebase. The prompt is simple *because the context is rich*.

The work does not disappear — it shifts. Instead of spending effort on every prompt, you invest upfront in your project's organization, documentation, and configuration. That investment compounds across every interaction, every session, and every teammate who uses an AI tool in the same project.

6.5 Context Engineering in This Course

You are already doing context engineering in TCSS 460 — you just might not have called it that:

- **Swagger / OpenAPI documentation** is context engineering. When your API spec describes every endpoint, its parameters, and its responses, any AI tool can generate client code, tests, or admin interfaces from it.
- **The course starter repos** are context engineering. Their consistent directory structure, naming conventions, and configuration files teach every agent how your project works the moment it opens the folder.
- **Your group project README** is context engineering. A clear description of what the project does and how to run it helps both human teammates and AI agents get oriented.
- **Your weekly API documentation updates** are context engineering. Every time you document a new endpoint, you make it easier for an agent to work with it.

The quality of your Swagger documentation directly determines the quality of what an AI agent can generate from it. Students who maintain thorough, accurate API docs will find that the agent produces working code with minimal correction. Students whose docs are incomplete or outdated will spend their time debugging mismatches between the generated code and the actual API. That is context engineering in practice.

> **Gen AI & Learning: A Skill That Transfers**
>
> Context engineering is not specific to any tool, model, or assignment. It is the practice of making your project understandable — to humans and to AI. As models change and tools evolve, the developers who invest in clear project structure, thorough documentation, and well-captured intent will get the best results from whatever comes next. This is one of the most future-proof skills you can develop.

---

7 Common Pitfalls

These are the mistakes students make most often when working with coding agents. Awareness of them will save you significant time and frustration.

7.1 Accepting Code You Do Not Understand

This is the number one pitfall, repeated here because it matters that much. When you accept code you do not understand, you create **technical debt against your own learning**. Every subsequent feature, debug session, and check-off becomes harder because you are building on a foundation you cannot inspect.

The fix: when an agent generates something unfamiliar, ask it to explain. "What does the `async` keyword do here?" "Why are you using `Promise.all` instead of sequential `await` calls?" "What is the `next` parameter in this middleware?" Use the agent as a teacher, not just a code generator.

7.2 Not Testing Generated Code

Code that compiles is not code that works. Agents produce syntactically valid code that can still:

- Return wrong results for certain inputs
- Miss error cases entirely

- Work in isolation but break when integrated with the rest of your application
- Use deprecated APIs or incorrect library versions

Always run generated code. Send it real requests. Try edge cases. Check what happens with missing fields, empty strings, negative numbers, and concurrent requests.

> **A Common Trap**
>
> An agent might generate a route handler that works perfectly when you test it with valid input. But send it a request with a missing required field, and it crashes with an unhandled error instead of returning a 400 response. Always test the unhappy paths.

7.3 Over-Relying on the Agent for Design Decisions

Coding agents are excellent at implementing patterns but poor at making architectural decisions. Do not ask an agent to *choose* for you — ask it to *inform* your choice.

For example, instead of asking "Should I use raw SQL or an ORM here?" — which will get you a confident but context-free answer — try a different approach:

- "Describe the trade-offs between using raw SQL queries with `pg` and using Prisma ORM for this data access layer. What are the advantages and disadvantages of each in a TypeScript Express project?"
- "Given that this route needs to join three tables and return a paginated result, what are the performance and maintainability trade-offs between a raw SQL query and a Prisma query?"

The agent can lay out the trade-offs, describe when each approach shines, and explain what you give up with each choice. That is valuable. What the agent *cannot* do is weigh those trade-offs against your specific constraints — your team's experience, your deadline, your project's complexity, and what you are trying to learn. That judgment is yours.

The same principle applies to questions like "How should I structure my API endpoints?" or "What should my database schema look like?" — use the agent as a research tool that presents options with reasoning, not as a decision-maker that picks one for you. The course lectures, readings, and project milestones are where you develop the judgment to make these calls.

7.4 Copying Without Adapting

When an agent generates code in a new conversation — or you find AI-generated snippets online — the code follows generic conventions, not yours. Common symptoms:

- Variable names that do not match your project's style
- Import paths that do not match your file structure
- Error handling patterns that differ from the rest of your codebase
- Configuration assumptions that do not match your setup

Always adapt generated code to fit your project. Consistency matters more than any individual coding choice.

Context Engineering Reduces This Problem

This pitfall is much less of a problem when you invest in context engineering (Section 6). An agent that operates inside a well-structured project with clear conventions, consistent patterns, and tool-specific configuration will naturally produce code that matches your style — because it learned the style from your codebase. The more context the agent has, the less adapting you need to do.

7.5 Ignoring the Learning Opportunity

The biggest pitfall is not a technical one. It is using the agent to *skip* learning rather than *accelerate* it. If you use an agent to generate your entire check-off assignment without engaging with the concepts, you will:

- Struggle during the in-person check-off conversation
- Lack the foundation for later assignments that build on those skills
- Miss the conceptual understanding that lectures and readings provide

Gen AI & Learning: The 70/30 Rule

A useful mental model: spend roughly 70% of your time understanding and 30% generating. Read the lecture notes. Work through the concept readings. Attempt the problem yourself first. Then use the agent to accelerate the parts you already understand. If you find yourself spending 90% of your time prompting and 10% understanding, you have the ratio backwards.

7.6 Letting the Agent Run Too Far

Coding agents are eager. You ask for one route handler and the agent adds three, refactors your middleware, updates your error handling, and "improves" your project structure — all in one pass. This is not malice; it is how the tools are built. Agents try to be helpful, and "helpful" often means doing more than you asked.

The problem: every line the agent touches is a line you need to review, understand, and maintain. If you asked for one change and the agent made twenty, you now have twenty things to verify instead of one. Worse, the extra changes may introduce bugs, break conventions you deliberately chose, or solve problems you did not have.

Watch the agent work. Most tools show you what they are doing in real time — files being opened, edits being proposed, commands being run. If you see the agent reaching into files you did not mention, refactoring code that was not part of your request, or "cleaning up" things that were fine, **stop it**. Cancel the operation, roll back the changes, and give a more constrained prompt.

Practical habits:

- **One task at a time.** "Add input validation to the register route" — not "add validation and also fix the error handling and update the tests and refactor the middleware."
- **Name the files.** "Only modify `src/routes/auth.ts`" tells the agent to stay in its lane.
- **Check the diff before accepting.** If the diff touches files you did not expect, reject it and ask again with tighter scope.
- **Undo liberally.** Git makes this easy. If an agent made a mess, `git checkout .` gets you back to where you started. Commit before you let an agent work so you always have a clean rollback point.

> ⚠ **The Snowball Effect**
>
> The most dangerous version of this pitfall is when you let an overeager agent run, then ask it to fix the problems it introduced, which causes more changes, which cause more problems. Before you know it, your project looks nothing like it did an hour ago and you cannot explain any of the changes. If you feel like you are losing control of your codebase, stop, revert to your last clean commit, and start the task over with a smaller scope.

---

8 Summary

Concept	Key Point
Coding agent	AI that reads, writes, and runs code in your project — not a chatbot
Tokens	Chunks of text the model processes (~1.3 tokens per word, ~10-15 per line of code)

Concept	Key Point
Context window	The model's working memory — larger windows (1M tokens) hold more but can lose focus
Usage limits	Every tool has them — agent mode consumes far more tokens than simple completions
Compaction	When context fills up, older conversation is summarized — the agent may "forget" earlier decisions
Context	The agent's knowledge of your codebase is its main advantage over chat-based tools
When to use	Boilerplate, understanding unfamiliar code, debugging, generating tests from documentation
When NOT to use	You cannot explain the output; the assignment tests your individual skills
Prompt engineering	Give context in each prompt, be specific, describe expected behavior — useful for one-off interactions
Context engineering	Invest in project structure, documentation, and tool configuration so simple prompts produce good results
Review everything	You are responsible for every line, regardless of who (or what) wrote it
Common pitfalls	Accepting code you do not understand; not testing; over-relying on the agent for decisions
Academic integrity	AI tools are permitted; understanding is required; you must be able to explain your code

---

9 References

Official Documentation:

- GitHub — Responsible use of AI coding tools in education — GitHub Education's framework for AI in the classroom
- ACM Code of Ethics — Professional standards for computing professionals

Course Resources:

- AI Coding Tools Setup (companion guide) — Practical setup for specific tools available to students
- TCSS 460 Syllabus — Course-specific academic integrity policies

Context Engineering:

- Context Engineering for Developers — Faros AI — Practical strategies for selection, compression, ordering, isolation, and format optimization
- Context Engineering is the New Prompt Engineering — KDnuggets — Overview of the shift from prompt to context engineering

---

10 Further Reading

External Resources

- Stanford HAI — AI in Education — Research and perspectives on AI tools in higher education
- Prompt Engineering Guide — Techniques for writing effective prompts (applicable to coding agents)
- Google — Pair Programming with AI — Google's research on human-AI collaboration in software development
- Context Engineering vs Prompt Engineering: The 2025 AI Shift — Towards Agentic AI — Detailed comparison of the two approaches
- Context Engineering: Why It's Replacing Prompt Engineering — DEV Community — Community perspective on the shift

---

This guide is part of TCSS 460 — Client/Server Programming, School of Engineering and Technology, University of Washington Tacoma.