

ELLYPUNK

Foundation

WEEKLY DIGEST / E101

27 July - 2 August 2026



This week

27 July - 2 August 2026

Available archived editions, in their original date order.

MONDAY / 27 July 2026

Agentic Coding 101: From Goal to Action

3

[Open original online](#)

TUESDAY / 28 July 2026

Agentic Coding 102: Context Is Fuel

8

[Open original online](#)

WEDNESDAY / 29 July 2026

Agentic Coding 103: Tools Turn Plans into Action

13

[Open original online](#)

THURSDAY / 30 July 2026

Agentic Coding 104: Verify, Learn, Iterate

18

[Open original online](#)

FRIDAY / 31 July 2026

Agentic Coding 105: Build a Feature End to End

23

[Open original online](#)

Only retained editions are included; gaps have not been filled.

Copyright Ellypunk. Contains AI-generated artwork.

ellypunk.com

AGENTIC CODING 101

PAGE 1



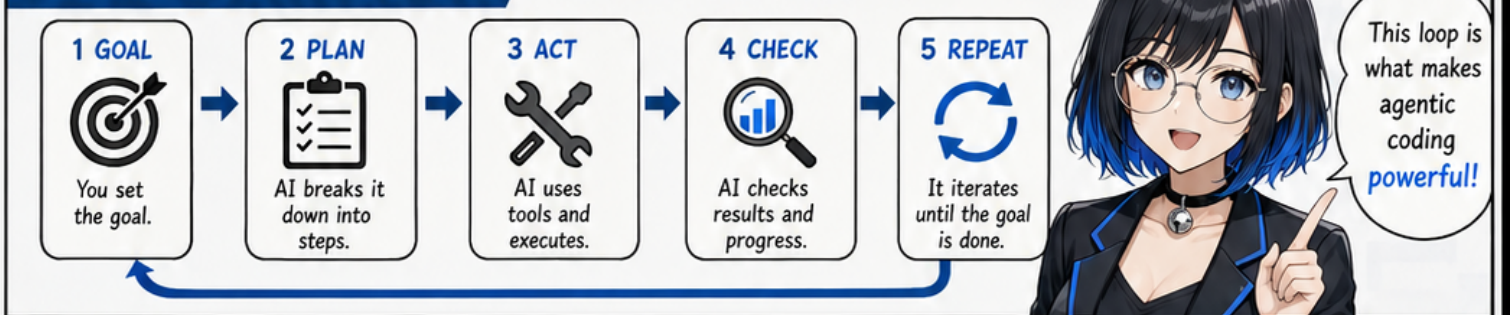
WHAT IS AGENTIC CODING?



HOW IS IT DIFFERENT?



THE AGENTIC CODING LOOP



EXAMPLE GOAL FROM A STUDENT



KEY TAKEAWAY



Remember: Clear goals → Better plans → Better results.

NEXT: Seeing it in action! →

FROM GOAL TO PLAN

PAGE 2

THINK
PLAN
ACT
REPEAT
</>

A good agent doesn't jump straight into coding.

It first turns a big **goal** into smaller, manageable **steps**.

A STUDENT ASKS...

Elly, can you help me with this task?

"Combine two CSV files and flag missing IDs."

HOW ELLY (THE AGENT) MAKES A PLAN

Great! I won't code yet. First, I'll make a **plan**.

Here's how I'll break your goal into clear steps.

VISUAL EXAMPLE

Step by step, we go from messy inputs to a **clean, useful result**.

customers.csv		orders.csv	

+

merged_with_flags.csv

	MISSING_ID_FLAG
ID_101	—
ID_102	—
ID_999	YES

THE PLAN: STEP-BY-STEP

1 Inspect files Check both CSV files exist and can be read.

2 Understand columns Look at column names and data types in each file.

3 Write merge logic Decide how to join the files (on the ID column).

4 Flag missing IDs Find IDs that don't have a match in the other file.

5 Test on sample data Run on a few rows to check the result looks right.

6 Save output Save the final CSV with missing ID flags.

WHY BREAK IT DOWN?

Easier
Each step is simple and clear.

Safer
We can test and catch problems early.

More Reliable
Small, checked steps lead to better results.

KEY TAKEAWAY

Agents don't need magic—they need a sensible **next step**.

What's the next step?

Give the agent a clear goal, and it will plan the path!



Remember: Plan first → Fewer surprises → Better outcomes.

NEXT: From plan to action! →

THE 4 THINGS AN AGENT NEEDS

PAGE 3



1. GOAL - WHAT SUCCESS LOOKS LIKE



EXAMPLE



Goal:

Create a script that cleans a CSV and generates a summary chart.

Done when:

- clean file saved
- chart created
- tests pass

2. CONTEXT - WHAT IT NEEDS TO KNOW



EXAMPLE



project/
README.md



main.py
utils.py



Instructions:

- Clean data
- Handle missing values
- Output summary

More context = better results.

3. TOOLS - WHAT IT CAN USE



COMMON TOOLS



Terminal



Code Editor



File Reader



Search



Tests



Git

...and more!

4. CONSTRAINTS - RULES & BOUNDARIES



EXAMPLE RULES



Use Python 3.11 for all code.



Don't delete files or change .env.



Ask before running long tasks or installs.



Don't commit or push without permission.

Clear rules = fewer mistakes.

A LITTLE EXTRA: MEMORY / HISTORY HELPS



I remember what we did earlier!

Memory (history) helps the agent remember earlier steps, decisions, and results so it can keep building without starting over.

Step 1

Load data



Step 2

Clean data



Step 3

Create chart



...

Important, but the core basics are the 4 items above.

THE BIG PICTURE



+
 +
 +
 = **SMART AGENT** ★

A smart agent is only as good as the goal, context, tools, and rules you give it.

Good agents follow great plans. Give the 4 essentials and watch them perform!

NEXT: Putting it all together! ➔

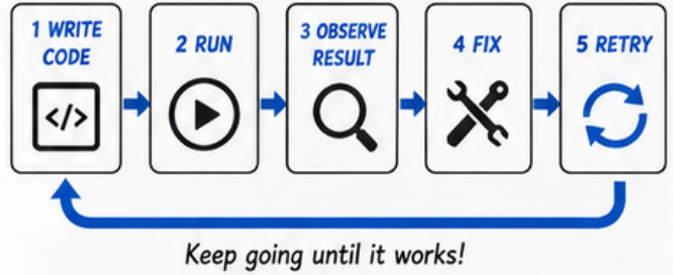
ACT, CHECK, AND ITERATE

PAGE 4



After we **PLAN**, it's time to **ACT**!
The **agent** tries, checks the result, learns from it, and tries again.

THE ITERATION LOOP IN ACTION



1 WRITE CODE

Elly writes a script to merge sales and products data by "product_id".

merge_data.py

```

1 import pandas as pd
2
3 sales = pd.read_csv("sales.csv")
4 products = pd.read_csv("products.csv")
5
6 # Merge on product_id
6 df = sales.merge(products, on="productID")
7 df.to_csv("merged.csv", index=False)
8 print("Done")
9
    
```

2 RUN

Let's run it and see what happens!

Terminal

```

$ python merge_data.py
...
Traceback (most recent call last):
  File "merge_data.py", line 7, in <module>
    df = sales.merge(products, on="productID")
  File "pandas/core/frame.py", line 1052, in merge
    return merge(...)
KeyError: 'productID'

The above exception was the direct cause
of the following exception:

KeyError: 'productID'
    
```

ERROR!

3 OBSERVE RESULT

The error says "KeyError: 'productID'". That column doesn't exist. Maybe it should be "product_id".

WHAT WE LEARN:

The result tells us what went wrong and what to try next.

Results guide the next action!

4 FIX

Let's fix the column name.

merge_data.py (fixed)

```

1 import pandas as pd
2
3 sales = pd.read_csv("sales.csv")
4 products = pd.read_csv("products.csv")
5
6 # Merge on product_id
6 df = sales.merge(products, on="product_id")
7 df.to_csv("merged.csv", index=False)
8 print("Done")
9
    
```

Fixed!

5 RETRY

Let's run it again!

Terminal

```

$ python merge_data.py
Done
    
```

SUCCESS!

RESULT: IT WORKS!

No errors, file created.

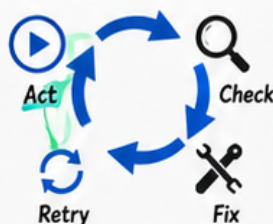
merged.csv (preview)

product_id	product_name	price	quantity	sale_date
p01	Laptop	999	1	2025-04-20
p02	Mouse	25	2	2025-04-20
p03	Keyboard	45	1	2025-04-20
...				

Great! The data's merged correctly.

THE TAKEAWAY

Agentic coding isn't about getting it right the first time. It's about **acting**, **checking results**, **learning**, and **iterating** until the goal is achieved.



Iteration is the **secret sauce** of agentic coding!



Remember: Results → Insights → Better Actions → Better Outcomes

NEXT: Real example in action! →

YOU'RE STILL IN CHARGE

PAGE 5

YOU + AGENT = AMAZING ❤️

Agentic coding is powerful, but **you're still in charge**.

You set the goal, set the guardrails, and make the **final call**.

TODAY'S FOCUS:

- ✓ How to give a great goal
- ✓ Weak goal vs. strong goal
- ✓ Key guardrails
- ✓ Full loop recap

HUMAN LEADS TECH FOLLOWS

1) HOW TO GIVE A GREAT GOAL / PROMPT

A **great goal** helps the agent help you!



BE SPECIFIC ABOUT THE TASK

Say **exactly** what you want the agent to do.



PROVIDE CONTEXT

Share the **background, files,** and details it needs to know.



SET CONSTRAINTS

State **rules, limits, formats,** and what **not** to do.



2) WEAK GOAL vs. STRONG GOAL

WEAK GOAL 😞

"Fix my data stuff."

STRONG GOAL ✓

"Write a Python script that reads **visits.csv**, removes duplicate patient IDs, and saves the cleaned data to **cleaned_visits.csv**. Do **not** edit the original file."



3) KEY GUARDRAILS: STAY SAFE, STAY IN CONTROL

REVIEW IMPORTANT OUTPUTS



Check code, results, and summaries—especially anything **critical**.

PROTECT SENSITIVE DATA



Never share secrets, passwords, or **private data**. Use safe, masked, or sample data.

ASK FOR APPROVAL BEFORE RISKY CHANGES



Confirm before deleting files, changing settings, or running big operations.

VERIFY RESULTS



Test. Re-run. Spot-check. Trust, but **verify**.



4) THE FULL LOOP RECAP

1 GOAL



You set the goal.

2 PLAN



AI breaks it down into steps.

3 ACT



AI uses tools and executes steps.

4 CHECK



AI checks results and progress.

5 REPEAT



It iterates until the goal is done.

You guide the loop. The agent helps run it.

5) YOU'VE GOT THIS!



FINAL TIP FROM ELLY

Start small, stay clear, and let the agent help step by step. ❤️

YOU + AGENT = AMAZING

Keep learning. Keep building. Have fun! ☆



Goal → Plan → Act → Check → Repeat



You're in charge. The agent is your **teammate**.



AGENTIC CODING 102

PAGE 1 OF 5

CONTEXT IS FUEL

Welcome back!

Today's lesson is simple
but powerful:

CONTEXT IS FUEL. ★



Agents don't guess well. They do better when you give them the **RIGHT CONTEXT**.

Context is the useful information an agent needs **before** it acts.

WHAT IS CONTEXT?

Context is the **useful information** an agent needs **before** it acts.



PROJECT FILES
Code, docs, data, configs, examples.



INSTRUCTIONS
What to do, rules, preferences, steps.



DESIRED RESULT
Your goal, success criteria, and format.

Good context = **better results**, fewer mistakes, and less back-and-forth.

SEE THE DIFFERENCE

LOW CONTEXT



Make a chart of sales.

RESULT

A generic chart. May not match your needs.



HIGH CONTEXT

- Here is `sales.csv`
- Chart type: line chart
- Show trend by month
- Title: Monthly Sales Trend
- Output: PNG



RESULT

Accurate, relevant, and ready to use!



Think of context as **fuel** for the agent.
More relevant fuel → stronger, smarter action.
Next, we'll see how to build **great context** step by step! ★

QUICK TAKEAWAY

Give the agent the right context, and it will do its best work.

AGENTIC CODING 102: **CONTEXT IS FUEL**

PAGE 2 OF 5

BUILD GREAT CONTEXT

A GREAT CONTEXT INCLUDES FOUR THINGS:

- 1 Explain the task
- 2 Say why it matters
- 3 Provide what's needed
- 4 Describe the wanted output

Agents are powerful, but **context** is **fuel**.
The clearer your context, the better the results.

GOOD CONTEXT → BETTER RESULTS

CONTEXT FUELS CLARITY

1 EXPLAIN THE TASK

Be specific about what you want the agent to do.

Example:
"Add a dark mode toggle that saves the user's preference."

2 SAY WHY IT MATTERS

Give the reason or goal behind the task. This helps the agent make better choices.

Example:
"It improves **usability** and reduces eye strain at night."

3 PROVIDE WHAT'S NEEDED

Share relevant files, code, constraints, and any important rules or details.

Include things like:

 Code files

 Config

 Designs

 Rules & Constraints

4 DESCRIBE THE WANTED OUTPUT

Describe the result you want to see. Be clear and measurable when possible.

Example:
"A toggle in the header that switches themes, **persists** the choice, and updates all colors."

EXAMPLE CONTEXT PACKET

Put it all together like this!

CONTEXT PACKET: Add Dark Mode

- TASK:** Add a dark mode toggle that saves the user's preference.
- WHY IT MATTERS:** It improves usability and reduces eye strain at night.
- WHAT'S NEEDED:**
 - Files: /src/App.tsx, /src/theme.ts, /src/styles.css
 - Stack: React + TypeScript, Tailwind CSS
 - Use existing ThemeContext. Store preference in localStorage.
 - Follow current code style and naming conventions.
- WANTED OUTPUT:** A toggle in the header that switches themes, persists the choice, and updates all colors.

My App

Light Mode

Dark Mode

Great context leads to fewer back-and-forths and better solutions.

KEY TAKEAWAY

Clear task

Why it matters

What's needed

Wanted output

Better results

★ Clear context. Better plans. **Better code.**

NEXT: Break Down the Task →

SHOW, DON'T MAKE IT GUESS

Examples, existing patterns, and concrete input/output samples help the agent understand exactly what you want.



The more context you give, the less your agent has to guess.

WEAK CONTEXT (LOTS OF GUESSWORK)

Build a function that summarizes sales data.



What the agent sees:

... not much to go on.

It has to guess:

- What's the input format?
- What metrics matter?
- What's the output style?
- Any edge cases?

Possible result:

It might work... but it might miss the mark.

```
# Example (guessed)
def summarize(data):
    total = sum(d['amount'])
    count = len(data)
    return {'total': total,
            'count': count}
```

STRONG CONTEXT (CLEAR DIRECTION)

Give the agent the right fuel, and it can deliver exactly what you need.



What the agent sees:

- **Goal:** Summarize sales by product.
- **Input:** List of orders with product, amount, date.
- **Output:** Table with product, total_sales, order_count, average_order_value.
- **Rules:** Ignore refunds; dates are ISO format. Round averages to 2 decimals.
- **Example:**

Input (orders.csv)

product	amount	date
Tea	4.50	2025-05-01
Tea	5.00	2025-05-02
Mug	12.00	2025-05-02
...	...	...

→

Output (summary.csv)

product	total_sales	order_count	avg_order_value
Tea	9.50	2	4.75
Mug	12.00	1	12.00
...	...	...	...

Better result:

It's accurate, consistent, and ready to use.

```
# Example (aligned with your context)
from collections import defaultdict
def summarize(orders):
    stats = defaultdict(list)
    for o in orders:
        if o.get('type') == 'refund':
            continue
        stats[o['product']].append(o['amount'])
    rows = []
    for product, amounts in stats.items():
        total = round(sum(amounts), 2)
        count = len(amounts)
        avg = round(total / count, 2)
        rows.append({'product': product,
                    'total_sales': total,
                    'order_count': count,
                    'avg_order_value': avg})
    return rows
```



Think of context as fuel for the agent's engine.



EXAMPLES
Show the pattern you want.

+



EXISTING PATTERNS
Leverage your real code and style.

+



INPUT/OUTPUT SAMPLES
Make expectations concrete.

=



BETTER RESULTS
Less guesswork. More impact.

KEY TAKEAWAY



The agent can only work with what you give it.
Show, don't make it guess.



Great context = great code. ❤️

USEFUL, NOT OVERLOADED



Agents work best with the **right context**—relevant, clear, and current. More isn't better. **Better is better.**

TODAY'S FOCUS:

Useful, Not Overloaded



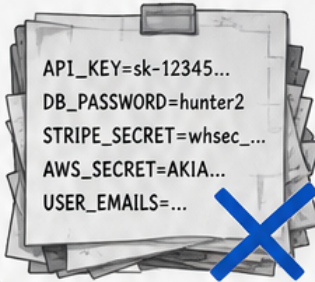
WHAT TO EXCLUDE



Instead, include only what's relevant, accurate, and necessary.

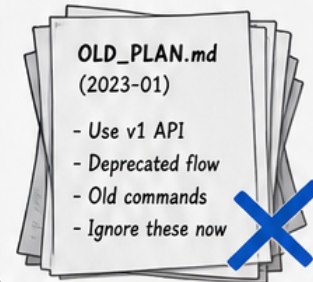
1. SECRETS & SENSITIVE INFO

Never include keys, tokens, passwords, or personal data.



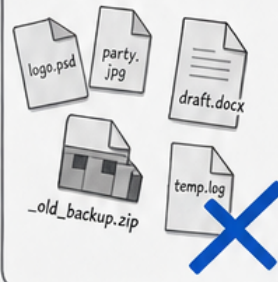
2. OUTDATED INSTRUCTIONS

Old notes and obsolete rules cause confusion.



3. UNRELATED FILES

Only include files that help solve the task.



FOCUS ON WHAT HELPS THE AGENT SUCCEED



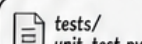
README.md
Project overview and how to run. ✓



requirements.txt
Dependencies the project needs. ✓



src/main.py
The code the agent will edit. ✓



tests/unit_test.py
How success is verified. ✓



docs/plan.md
Current plan and decision history. ✓

FROM OVERLOADED...



...TO JUST ENOUGH



Clear context. Better results.

Think **quality** over quantity.

REMEMBER:

- ✓ Exclude secrets, outdated instructions, and unrelated files.
- ✓ Choose only what helps the agent understand and act.
- ✓ A clean context packet leads to better code, faster.

GIVE THE AGENT THE RIGHT FUEL, NOT THE WHOLE KITCHEN. ☆

AGENTIC CODING 102:

CONTEXT IS FUEL

PAGE 5 OF 5



GIVE THE AGENT THE RIGHT CONTEXT. **GET BETTER RESULTS.**

Before you ask the agent to do something, make sure your context covers these **five things**.

1 CLEAR GOAL

State exactly what you want to achieve.

Example:
Build a dashboard that shows **daily sign-ups** and **revenue**.

2 RELEVANT FILES

Share the files and folders that matter.



Example:
Here are the **component files**, the **data schema**, and the **styles**.

3 PROJECT INSTRUCTIONS

Provide key rules, standards, and preferences.



Example:
Use **TypeScript**, follow our **naming convention**, and keep components **small and reusable**.

4 CONSTRAINTS

List any limits or requirements the solution must respect.



Example:
Must be **mobile-friendly**, load under **2 seconds**, and use **no paid libraries**.

5 DESIRED RESULT

Describe what good looks like when we're done.



Example:
A clean dashboard with **charts**, **filters**, and **export to CSV**.
Ready for production.

PRO TIP

The more complete your context, the fewer back-and-forths we need—and the **better the results**.

THE TAKEAWAY: **RIGHT CONTEXT → BETTER ACTIONS**

Context is **fuel**.
Feed it well, and the agent can build **amazing things**.

You've got the foundations. Keep practicing, keep iterating, and **keep building**. I can't wait to see what you create! ❤️

See you in the next build! 🚀

AGENTIC CODING 103: TOOLS TURN PLANS INTO ACTION



Hi! I'm **Elly**.

Today we're talking about **TOOLS**—how they turn our plans into **real** results.

Today's Topic:

TOOLS

TURN PLANS INTO ACTION



FIRST, THE BIG IDEA.



An agent thinks about what to do (its **plan**), then uses **tools** to do the work.



THINK
Plan the work

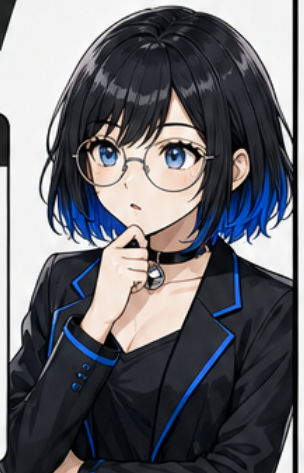


TOOL
Do the work

THINKING vs. USING A TOOL

These are **NOT** the same.

THINKING ABOUT WORK	USING A TOOL
- Reasoning - Deciding - Planning - Predicting  It's in the agent's head.	- Inspecting - Changing - Running - Verifying  It changes the world.



WHAT IS A TOOL?



A tool is an ability an agent can use to **inspect**, **change**, **run**, or **verify** real work outside of its head.

Four core abilities most tools provide:

INSPECT  Look at files, code, logs, or system state.	CHANGE  Create, edit, move, or delete files and code.	RUN  Execute commands, scripts, tests, or programs.	VERIFY  Check results, test, lint, or validate.
--	--	---	---

Tools let the agent interact with the **real world** and get things done.

EXAMPLE: SAME GOAL, TWO DIFFERENT PATHS



THINKING ONLY (no tool used)

"I should open sales.csv, calculate totals, and make a chart."

Result: **Nothing changed.**
It's still just an idea.



USING TOOLS (plan + action)

-  Inspect sales.csv ✓
-  Change: add totals column ✓
-  Run: generate chart ✓
-  Verify: totals look correct ✓



Same goal. Only the tool path creates **real impact**.



KEY TAKEAWAY: Plans live in the head. Tools change the world. Great agents know when to **think**—and when to **use a tool**.



AGENTIC CODING 103:

TOOLS TURN PLANS INTO ACTION

Before you write or edit anything, understand what's already there.

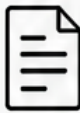
Use the right tools to **read files**, **find code**, and **check docs**.

Good changes start with clear understanding.



THREE CORE TOOLS TO UNDERSTAND A PROJECT

- 1) READ FILES**



Open and read files to see how things work.

Look at key files first (e.g., README, config, main modules).
- 2) SEARCH PROJECT**



Search the codebase for functions, variables, or keywords.

Find where things are used and connected.
- 3) CHECK DOCS**



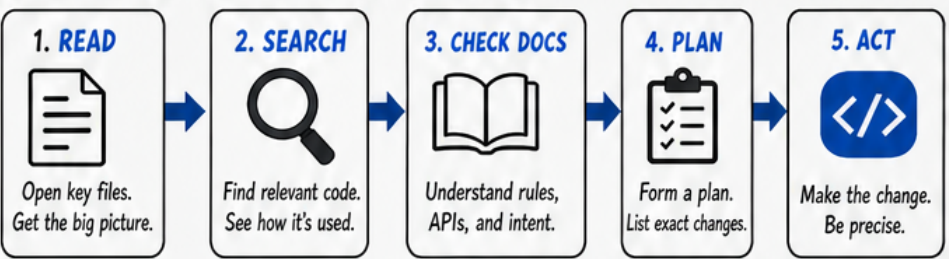
Read documentation and comments to learn intent and rules.

Docs explain the "why" behind the code.



These tools help you build **context** and **confidence** before you make any changes.

THE INSPECT-FIRST WORKFLOW



Inspect first, then act. This avoids mistakes and **save time**.

EXAMPLE: ADDING A USER SETTINGS OPTION

- 

READ: Check README and config files to see how settings work.
- 

SEARCH: Find "user settings" across the project. See where it's loaded and used.
- 

CHECK DOCS: Read the docs for the settings API and any rules or limits.
- 

PLAN: Decide where to add the option and what needs to change.
- 

ACT: Implement the change. Test and verify.



Context first.
Changes second.

KEY TAKEAWAY

Great agentic coding starts with **understanding**, not typing.



= **SMART CHANGES, FEWER MISTAKES**

Read before you change. That's how plans become **powerful actions**.

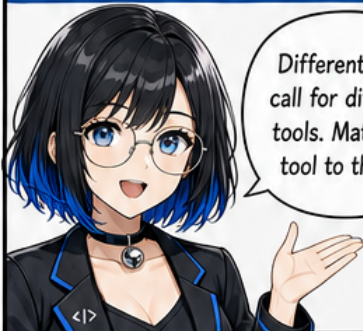


EDIT IN SMALL, SAFE STEPS

Great plans need careful edits. The **right tool**, **small changes**, **preserve work**, and **review the diff** keep your code safe and high quality.



1. CHOOSE THE RIGHT TOOL



Different edits call for different tools. Match the tool to the job.



Edit File

Small text changes in a specific file.



Find & Replace

Update repeated text across a file.



Refactor

Rename, move, or restructure safely.



Run Command

Install, test, or run tools.

2. MAKE A SMALL, FOCUSED CHANGE



Change one thing at a time so the result is easy to understand and revert if needed.

✓ GOOD EXAMPLE

Add one argument with a default.

```
def greet(name):  
    return f"Hello, {name}!"
```



```
def greet(name, excited=False):  
    msg = f"Hello, {name}!"  
    return msg.upper() if excited else msg
```

Small change. Same behavior by default.

3. PRESERVE EXISTING WORK



Protect what already works. Don't break working code while improving it.

- ✓ Keep existing interfaces the same (unless you're refactoring on purpose).
- ✓ Add backward-compatible defaults.
- ✓ Don't delete code unless you're sure it's unused.
- ✓ Run tests to confirm nothing broke.

4. REVIEW THE DIFF



Always review the diff before you accept the change. It shows exactly what will change.

DIFF: greet.py

BEFORE

```
1 def greet(name):  
2     return f"Hello, {name}!"  
3 ...
```

AFTER

```
1 def greet(name, excited=False):  
2 + msg = f"Hello, {name}!"  
3 + return msg.upper() if excited else msg  
4 ...
```

Added

Removed

Context



If something looks off, **adjust the plan or the change**—then try again. Small, reviewed steps lead to big, reliable results.

REMEMBER:



Right tool



Small change



Preserve work



Review diff



Safe progress!

RUN, TEST, AND ASK FIRST



Great plans need good tools.

I use tools to **run** code, **test** it, and **check** results—then I **ask** before anything risky.

TODAY'S FOCUS



Run: Execute your code



Test: Check that it works



Ask: Get permission for risky actions

1) RUN: EXECUTE YOUR CODE



I run my code early and often to see what happens.
Tools show output, errors, and warnings right away.

TERMINAL

```
$ python sales_report.py
Generating report...
Reading sales.csv...
Processed 250 rows.
Report saved to report.csv
Done. ✓
```



Tip: Small, frequent runs catch problems fast.

2) TEST: CHECK THAT IT WORKS



I test key behaviors with small checks.
Tests give me confidence to keep going.

TEST RESULTS

Load data	✓
Clean data	✓
Calculate totals	✓
Create chart	✓
Save file	✓



Tip: Good tests are clear, fast, and repeatable.

3) ASK: GET PERMISSION BEFORE RISKY ACTIONS



Before any risky or irreversible action, I pause and ask you.
You're in charge—I'm here to help.



My Request:

"May I proceed with this action?
Here's what will happen and why."

EXAMPLES OF ACTIONS THAT NEED APPROVAL

File & Data



- Delete files
- Overwrite files
- Move or rename big folders

System & Environment



- Install packages
- Change settings
- Edit config files
- Restart services

External & Shared



- Send data
- Access external APIs
- Push to repository
- Spend money / quotas



When in doubt, ask. A quick yes now prevents big problems later.

SAFE ROUTINE ACTIONS (I PROCEED)

- ✓ Read files and data
- ✓ Run code and tests
- ✓ Generate charts and reports
- ✓ Format code and docs
- ✓ Show results and summaries

VS.

ACTIONS NEEDING APPROVAL (I ASK FIRST)

- ⚠ Delete or overwrite anything
- ⚠ Change system settings
- ⚠ Install new software
- ⚠ Access external services or data
- ⚠ Push or publish changes



Run. Test.
Ask first.

That's how we build things that **work**—and stay **safe**.



REMEMBER:



Use tools to run and see results.



Test to build confidence.



Ask before risky or irreversible actions.



Smart tools. Safe choices.

AGENTIC CODING 103:

TOOLS TURN PLANS INTO ACTION

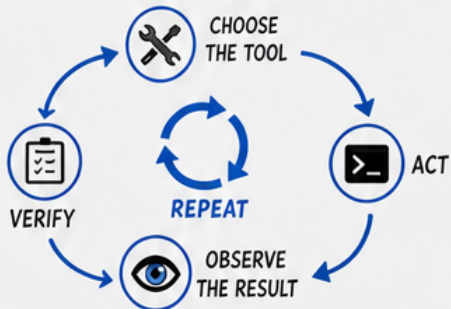
PAGE 5 OF 5



Let's bring it all together in a simple workflow.

Good agentic coders use **tools** in a loop that creates **progress**.

THE TOOL WORKFLOW

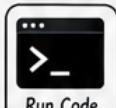


1 CHOOSE THE TOOL

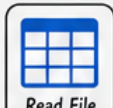
Pick the right tool for your next step. Think about what you need to do.



EXAMPLES:



Run Code
Execute your program.



Read File
Load data or check content.



Make Chart
Visualize results quickly.
...and more!

2 ACT

Use the tool. Give it the input, run it, and let it do the work.



EXAMPLE: RUN CODE

```
> python analyze.py --data sales.csv
```

Output:
Processed 120 rows.
Summary saved to summary.txt

3 OBSERVE THE RESULT

Look closely at what happened. What do the results tell you?



EXAMPLE: CHECK THE OUTPUT

summary.txt
Total Sales: \$12,450
Average Order: \$103.75
Top Product: Widget A

The numbers look reasonable, but Widget A is way ahead. **Interesting!**

4 VERIFY

Check that the result is correct and meets your goal. Validate it.



EXAMPLE: QUICK CHECKS

- ☒ Do the numbers make sense?
- ☒ Does the output match my goal?
- ☒ Any errors or warnings?
- ☒ Should I try a different tool?



5 REPEAT

Use what you learned to plan your next step, and repeat the loop!



WHAT YOU GET:

- ☒ Clear steps
- ☒ Better decisions
- ☒ Faster learning
- ☒ Stronger results



THE RIGHT TOOL + A CLEAR GOAL = REAL PROGRESS

Keep the loop going—one small step at a time. You've got this!



AGENTIC CODING 104: VERIFY, LEARN, ITERATE

PAGE 1 OF 5



Hi! I'm **Elly**.

If you want agents that get real work done, remember one rule above all:

Results are evidence.

TODAY'S TOPIC:

RESULTS ARE EVIDENCE





An agent should never merely **claim success**.

It should show what it did, what it found, and the results it produced—so we can **verify** them.

NOT ENOUGH

"Done! It works!"



Just a claim.
No evidence.



GOOD

"Here's what I did and what I found."



Evidence you can verify.
You can trust and build on it.



That's why we use the **Agentic Coding Loop**.
It keeps the agent honest, useful, and getting better.

1 ACT



Take an action or make a change.



2 OBSERVE



Look at what happened.
Collect results.



3 VERIFY



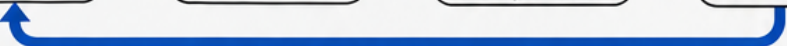
Check the results against the goal and expectations.



4 IMPROVE



Learn from what you found. Plan the next move.





Think of it like this:



Goal Agent Acts Results Verified Improvement

No **verification**, no **trust**.
No loop, no **progress**.

Great agents don't just finish tasks—they **prove, learn, and get better** with every cycle.

★ KEY TAKEAWAY

Results are evidence.
Use the loop:
ACT → OBSERVE → VERIFY → IMPROVE.
That's how agents earn trust and keep improving.



 Remember: Don't just claim it—**show it. Check it. Learn from it. Iterate.**

DEFINE **DONE** BEFORE YOU START

Before you plan or act, define what **"DONE"** looks like.

That means turning your goal into clear **acceptance checks**.

WHY IT MATTERS

- ✓ Focuses the plan
- ✓ Guides the steps
- ✓ Makes results verifiable
- ✓ Enables confident iteration

CODE
CREATE
REPEAT

WHAT ARE ACCEPTANCE CHECKS?

Acceptance checks are simple, specific rules that prove the goal is complete.

- Specific**
Clear and unambiguous
- Measurable**
Yes/No or a number
- Verifiable**
Easy to test or observe
- Time-bound (if relevant)**
When it should be done

EXAMPLE: GOAL → ACCEPTANCE CHECKS

GOAL:
Build a tool that summarizes a CSV file and shows key stats.

ACCEPTANCE CHECKS:

- ✓ User can upload a CSV file.
- ✓ App displays total rows.
- ✓ App displays count of missing values per column.
- ✓ App shows mean of numeric columns (2 decimal places).
- ✓ App exports summary to CSV.
- ✓ All checks pass on the sample file (sales.csv).

HOW TO WRITE STRONG ACCEPTANCE CHECKS

Good checks leave no room for guesswork.

- ✓ **Describe behavior, not steps**
Good: "App shows total rows."
Not great: "Click here, then there."
- ✓ **Use clear data and formats**
Good: "Mean shown with 2 decimal places."
Not great: "Show the mean."
- ✓ **Include edge cases**
Good: "Handles empty file with a clear message."
Not great: "Works for all files."
- ✓ **Keep it testable**
Good: "Exported CSV has header and matches 6 columns."
Not great: "Looks correct."

PRO TIP

Start small.
Add checks as you learn more.
Better checks lead to better results.

Define your **DONE** up front.
It's your compass for smart action and confident iteration!

REMEMBER

- ✓ Define **DONE** before you start
- ✓ Use clear, measurable checks
- ✓ Verify, learn, and iterate

CHECK THE WORK

Great plans are useless if the code doesn't work.

That's why we **VERIFY**.

Different checks catch different problems.



USE MULTIPLE CHECKS

Each check looks at your code from a **different** angle.



TESTS

Do the features work as expected?



Example:
Unit tests and integration tests



LINT

Is the code clean, consistent, and safe?

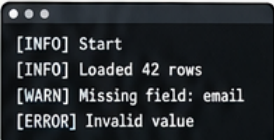


Example:
Style issues, unused variables, potential bugs



LOGS

What's happening at runtime?

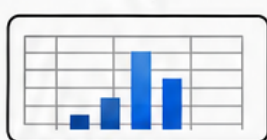


Example:
Errors, warnings, important events



OUTPUT INSPECTION

Does the output look right?



Example:
Files, charts, reports, API responses

No single check is perfect.

Together, they give you **confidence**.

VERIFICATION DASHBOARD

A clean dashboard shows you what passed, what failed, and what needs attention.

Overall Status



PASSED

Tests

28 / 28

Passed

Lint

0

Issues

Logs

2

Warnings

Output

Looks Good



Check	Details	Result	Notes
Tests	28 tests run 0 failed	PASSED	All assertions met.
Lint	0 errors 0 warnings	PASSED	Code style looks good.
Logs	2 warnings 0 errors	WARNINGS	Missing optional field: email Deprecated API used.
Output	report.csv chart.png	PASSED	Files generated successfully. Values look correct.

Green is great.

Yellow means "check it."

Red means "fix it."

APPLY THE RESULTS

Use what you learn from checks to make your **next** move.



Passed? Keep going!



Warnings? Decide if you can ignore them or should fix them.



Failed? Fix the issue, then check again.

Verification isn't just about catching mistakes.

It's how you **LEARN** what your code is really doing—so you can **ITERATE** with confidence.

★ **TIP:** Make verification automatic.

Run checks **early and often**.

FAILURE IS FEEDBACK



Great agents don't avoid errors—they **read** them, **learn**, and **iterate**.

Let's walk through the loop.

THE DEBUG-TO-DONE LOOP



1 READ THE ERROR

Read the full message carefully.



2 FIND THE ROOT CAUSE

Identify the real issue, not just the symptom.



3 MAKE ONE FOCUSED FIX

Change the smallest thing that addresses the cause.



4 RETRY AND VERIFY

Run again and confirm it's fixed.

1 READ THE ERROR



Read the whole error message. It tells you what went wrong and often where.

ERROR

```
File "app.py", line 7, in <module>
    total = sum(items)
            ^
```

NameError: name 'items' is not defined

Clue: 'items' isn't defined yet.

2 FIND THE ROOT CAUSE



Ask: What's the real cause of this error? Follow the clue to the line and variable.

app.py

```
1 prices = [10, 20, 30]
2 total = 0
3 for p in prices:
4     total = total + p
5
6 print(total)
7 total = sum(items)
```

Problem: Using 'items' which doesn't exist.

3 MAKE ONE FOCUSED FIX



Change the smallest thing that fixes the cause. Don't refactor yet.

BEFORE

app.py

```
5
6 print(total)
7 total = sum(items)
```



AFTER

app.py

```
5
6 print(total)
7 total = sum(prices)
```

One small fix: Use 'prices' which exists.

4 RETRY AND VERIFY



Run again and verify. Green output means you're back on track!

RUN

```
$ python app.py
60
(exit code 0)
```

Fixed!
Now we can keep building with confidence.



REMEMBER:

Read → Root Cause → One Fix → Retry
That's how agents turn failures into progress.

ELLY'S TIP

Small fixes. Fast loops. Big results.



AGENTIC CODING 104

PAGE 5 OF 5

VERIFY, LEARN, ITERATE

Great job getting results!

But real agentic coders don't stop there. We **VERIFY, LEARN,** and **ITERATE.**

TODAY'S TOPIC:

THE VERIFICATION LOOP

BUILD
THINK
ITERATE
</>

CODE
CREATE
REPEAT

THE VERIFICATION LOOP

1. RUN

Execute your code and get results.

Let's run it!

2. OBSERVE

Look carefully at what happened.

What do I see?

3. COMPARE WITH THE GOAL

Check results against your goal.

Does this match my goal?

4. FIX

Make the smallest change that addresses the gap.

Small change, big impact.

5. RETRY

Run it again and see what changed.

Let's try again!

6. REPORT HONESTLY

Share what you did, what happened, and what you learned—clearly and without spinning.

Honest reports build trust—with users and with yourself.

EXAMPLE REPORT

Goal: Clean this CSV and make a chart.

What I did: Loaded the file, cleaned missing values, and created a bar chart.

What happened: The chart had the wrong labels.

What I learned: The header row was misread.

What I'll try next: Fix header parsing and rerun.

**DON'T GUESS.
CHECK.**

- Run.
- Observe.
- Compare.
- Fix.
- Retry.
- Report honestly.

That's how we build reliable, intelligent solutions—together.



AGENTIC CODING 105: BUILD A FEATURE END TO END

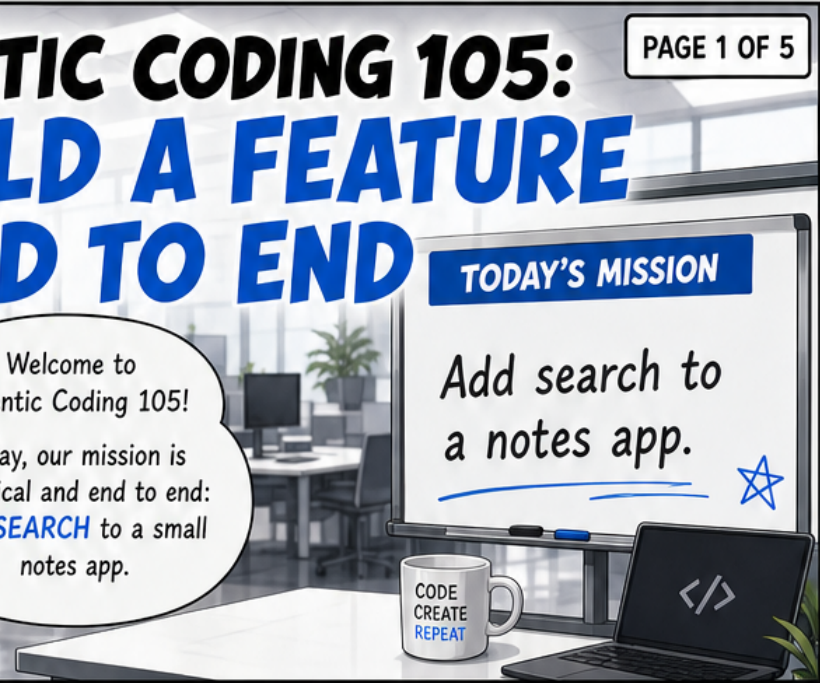
PAGE 1 OF 5

TODAY'S MISSION

Add search to a notes app. ☆

Welcome to Agentic Coding 105!

Today, our mission is practical and end to end: add **SEARCH** to a small notes app.



1. THE GOAL



We'll build one feature end to end: search.

When we're done, users can find any note **instantly**.

Here's the visible finish line:

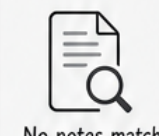
Search field at the top



Live filtering as you type

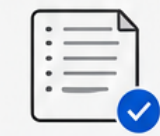


Clear empty state



No notes match "design system"

Existing notes unchanged



That's our **mission**. Let's do it. ☆

2. STARTING POINT (BEFORE)



This is our simple notes app **before** search.

Notes are stored and listed in newest-first order.

My Notes

+ New Note

Project ideas10:42 AM

Brainstorm features for the new release...

Design system9:15 AM

Color palette, typography, spacing...

Meeting notesYesterday

Discussed roadmap and priorities...

Book listYesterday

Books to read this year...

3. FINISH LINE (AFTER)



After we add search, it will look and work like this.

Type, filter, **done**. Fast and simple.

+ New Note

Design system9:15 AM

Color palette, typography, spacing...

Design reviewApr 30

Review the new components and tokens...

+ New Note



No notes match "design system"

One clear goal. One feature. End to end. ☆

INSPECT, THEN PLAN



Before writing code, we **inspect** what we have and **plan** the steps.

Great features come from understanding the current state, then building intentionally.

NOTES APP		+ New
Meeting notes	Apr 28	
Project ideas	Apr 26	
Daily log	Apr 25	
Research	Apr 24	
Shopping list	Apr 23	

FEATURE GOAL

Add search to the notes list so users can quickly find what they need.



1. INSPECT: EXPLORE THE CURRENT STATE



Let's look around. What files and patterns already exist?

- src/
 - components/
 - NotesList.tsx
 - NoteItem.tsx
 - hooks/
 - useNotes.ts
 - types/
 - index.ts
 - utils/
 - format.ts
 - App.tsx

NotesList.tsx

```
export function NotesList() {
  const { notes } = useNotes();

  return (
    <ul className="divide-y">
      {notes.map(note => (
        <NoteItem key={note.id} note={note} />
      ))}
    </ul>
  );
}
```



Pattern found: The list is rendered from useNotes(). Perfect place to add search.

2. PLAN: BREAK THE WORK INTO SMALL STEPS



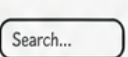
Big goals become easy when we break them down.

1 FIND



Find the notes list component and data source.

2 ADD STATE



Add search state to track the query.

3 FILTER



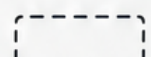
Filter notes based on the search query.

4 BUILD FIELD



Add the search input to the UI.

5 NO RESULTS



Handle empty results with a helpful message.

3. DEFINE: CLARIFY INPUT AND OUTPUT

INPUT

Search notes...

QUERY EXAMPLES

- "meet"
- "project"
- "2024"

OUTPUT

Meeting notes	Apr 28
Project ideas	Apr 26
Research	Apr 24

No notes found.
Try a different search.



Clear input. Clear output. Easier to build. Easier to test.

4. SUCCESS LOOKS LIKE THIS

pro

Project ideas	Apr 26
Project plan	Apr 18

No notes found.
Try a different search.



When we **inspect** first and **plan** well, our code ships faster and with confidence.



NEXT: Add the search state and wire up filtering.



BUILD ONE SLICE AT A TIME

Today we'll build the **SEARCH FEATURE** end to end—
one small slice at a time.
Each slice connects **state** → **UI** → **data**.
Build. Review. Improve. Repeat.



SLICE 1: CAPTURE INPUT

Let's add a search field and store what the user types.

Type to search...
Q meet

state: query = "meet"

All Notes (not filtered yet)

- Team meeting notes
- Project roadmap
- Meeting with client
- Design ideas

CODE (React)

```

1 function NotesApp() {
2   const [query, setQuery] =
3     useState("");
4   return (
5     <input
6       type="text"
7       placeholder="Type to search..."
8       value={query}
9       onChange={(e) =>
10        setQuery(e.target.value)}
11     />
12   );
13 }
  
```

✓ **REVIEW:** You can type, and the app remembers.
Good! Next, let's filter the list.

SLICE 2: FILTER DATA

Now we'll show only notes that match the query.

Type to search...
Q meet

Filtered Notes

- Team meeting notes
- Meeting with client

CODE (React)

```

1 const filtered = notes.filter(note =>
2   note.toLowerCase().includes(query.toLowerCase())
3 );
4
5 return (
6   <ul>
7     {filtered.map((note, i) => (
8       <li key={i}>{note}</li>
9     ))}
10   </ul>
11 );
  
```

✓ **REVIEW:** The list now shows only matching notes.
Data flow is working!

SLICE 3: CONNECT UX

Let's make it feel solid: trim spaces, handle empty states, and show counts.

Type to search...
Q meet

Filtered Notes (2 results)

- Team meeting notes
- Meeting with client

No results?
Try another keyword.

CODE (React)

```

1 const q = query.trim().toLowerCase();
2 const filtered = notes.filter(note =>
3   note.toLowerCase().includes(q)
4 );
5
6 const isEmpty = q.length === 0;
7
8 return (
9   <>
10     {isEmpty && (
11       <p className="muted">{filtered.length} results</p>
12     )}
13     {filtered.length > 0
14       ? filtered.map((note, i) => (
15         <li key={i}>{note}</li>
16       ))
17       : <li className="muted">Try another keyword.</li>
18     }
19   </>
20 );
21
22
  
```

✓ **REVIEW:** Clean input, helpful feedback, great UX.
The slice is complete!

That's the **agentic** way:
break it down, connect
the dots, and ship value
with every slice.

WHAT WE BUILT

- ✓ Capture input in state
- ✓ Filter data based on the query
- ✓ Connect UI to show results
- ✓ Polish with empty states and counts

**SMALL SLICES.
SOLID STEPS.
SHIPPED.**



➔ **NEXT:** Slice 4-5 – Debounce & Highlight

TEST THE REAL EXPERIENCE



A feature isn't done until it works for *real users*. Let's test search from *end to end*.

WHAT WE'RE TESTING

The search experience in Notes.

- ☒ Normal search
- ☒ Different capitalization
- ☒ Empty query
- ☒ No matches
- ☒ Existing-note behavior

Notes

Q Search notes...

Roadmap Ideas
Updated May 5

API Plan
Updated Apr 28

Customer Feedback
Updated Apr 20

TEST 1-4: SEARCH BEHAVIOR

1. Normal search

Q api

API Plan
Updated Apr 28

API Quickstart
Updated Apr 18

✔ Works!

2. Different capitalization

Q API

API Plan
Updated Apr 28

API Quickstart
Updated Apr 18

✔ Works!

3. Empty query

Q Search notes...

Roadmap Ideas
Updated May 5

API Plan
Updated Apr 28

✔ Works!
(Shows all notes)

4. No matches

Q quantum



No notes found.
Try a different keyword.

✔ Works!

5. Existing-note behavior

Q api

API Plan
Updated Apr 28

API Quickstart
Updated Apr 18

✔ Works!
(Keeps existing notes)

OOPS! FOUND A BUG



When I search and then clear the query, the results *disappear* instead of showing all notes again. That's not what we want!

BUG REPRO

Search "api"

Q api

API Plan
Updated Apr 28

API Quickstart
Updated Apr 18

Clear query

Q Search notes...



No notes found.
Try a different keyword.

Expected

Q Search notes...

Roadmap Ideas
Updated May 5

API Plan
Updated Apr 28

✗ Bug: Clearing the query should restore all notes, not show empty state.

FOCUSED FIX

I'll fix the logic so an empty query returns all notes.



```
searchNotes(query)
if (query.trim() === "") {
  return allNotes() // show all
}
return filterNotes(query)
```

Why this works:

- ☒ If the query is empty after trim, we return all notes.
- ☒ Otherwise, we filter normally.

Simple change, big impact.



RETRY AND SUCCESS!



Perfect! Now clearing the search shows all notes again.

Retry

Search "api"

Q api

API Plan
Updated Apr 28

API Quickstart
Updated Apr 18

Clear query

Q Search notes...

Roadmap Ideas
Updated May 5

API Plan
Updated Apr 28

All good!

Q Search notes...

Roadmap Ideas
Updated May 5

API Plan
Updated Apr 28

✔ Fixed: Clearing the query now restores all notes.



KEY TAKEAWAY

Test like a user. Find the edge cases. Fix with focus. Verify the fix.

NEXT: Ship it with confidence! →

AGENTIC CODING 105

BUILD A FEATURE END TO END

FROM GOAL TO DONE



Let's recap the complete agentic workflow—how we take a feature from an idea to **DONE**.

1 GOAL: Define success.



Be specific and measurable.

Start with a clear **goal**. What are we building, and how will we know it's successful?



2 CONTEXT: Give the big picture.



Share background, constraints, and what matters.

Give the agent the right **context**—requirements, preferences, constraints, and any examples.



3 PLAN: Break it down.



A good plan beats random work.

The agent creates a step-by-step **plan**. Review it, refine it, and confirm the approach.



4 TOOLS: Use the right tools.



Code, search, read files, run tests, query docs, and more.

The agent uses **tools** to gather info, explore options, and make changes safely.



5 FOCUSED CHANGES: Small, safe, reviewable.

```
File: app/api/users.py
@@ def get_users():
-   limit = 100
+   limit = 50
   return users[:limit]
```

Minimal diffs. Maximum clarity.

Make **targeted changes**. Keep diffs small and reviewable so nothing breaks.



6 VERIFICATION: Prove it works.



Tests, checks, and validation.

Verify with tests, checks, and manual review. If something fails, fix it.



7 ITERATION: Improve and repeat.



Learn, adapt, get better.

Use feedback to **iterate**. Refine the plan, adjust context, and keep going.

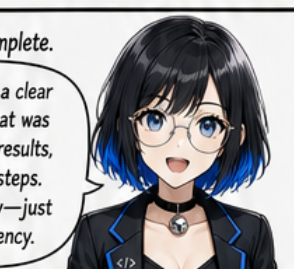


8 HONEST REPORT: Clear and complete.



What changed, why it changed, how it was verified, next steps.

Finish with a clear **report**: what was done, why, results, and next steps. No mystery—just transparency.



YOU SET THE DIRECTION.
THE AGENT HELPS YOU BUILD.



Agentic coding isn't magic. It's a workflow. Clear goals. Smart tools. Focused changes. Honest results. Now go build something you're proud of.



YOU DREAM IT. YOU GUIDE IT. YOU BUILD IT. **END OF LESSON** `</>`