FROM NON-CODER TO BUILDER

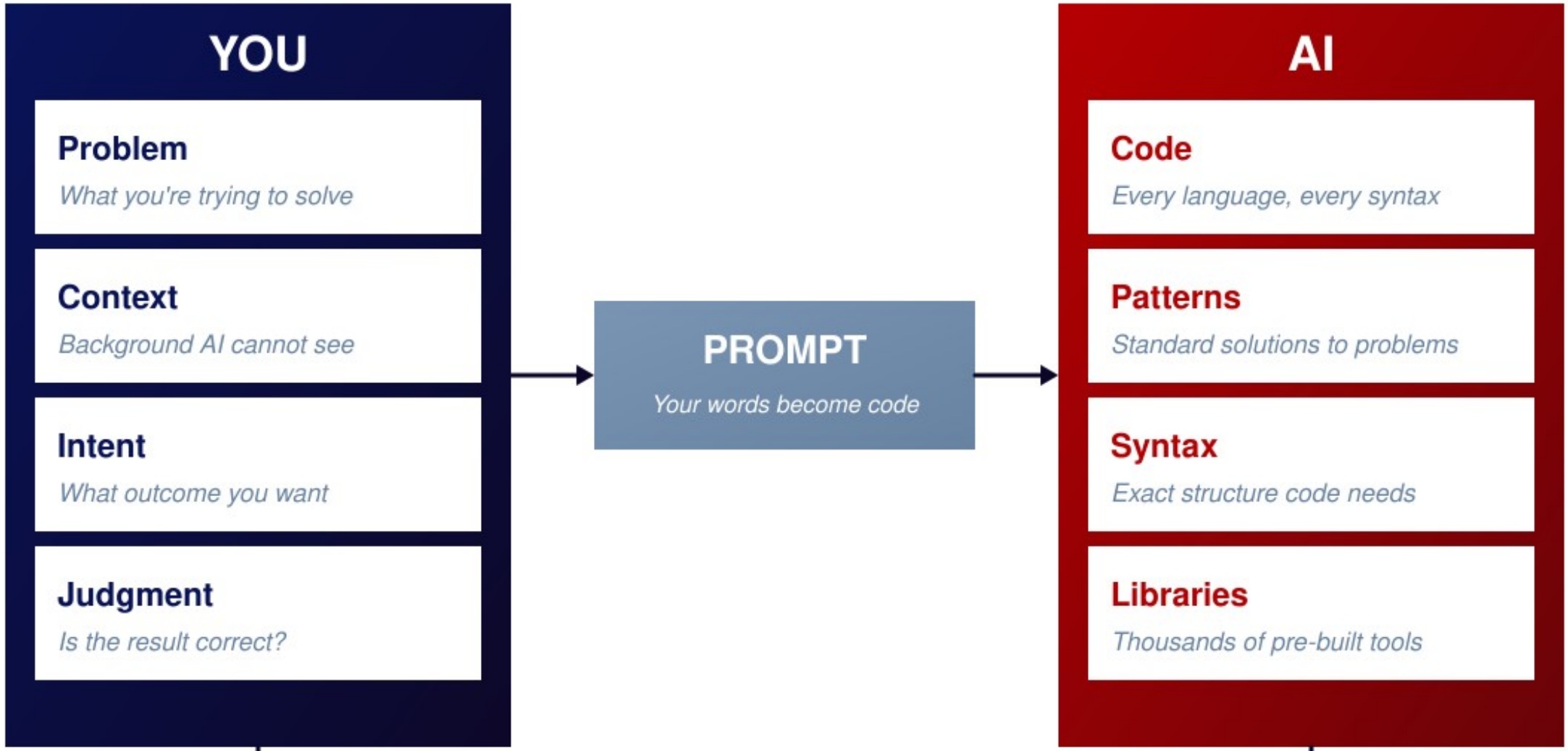
Coding with AI for Engineers

PART 1

FOUNDATIONS

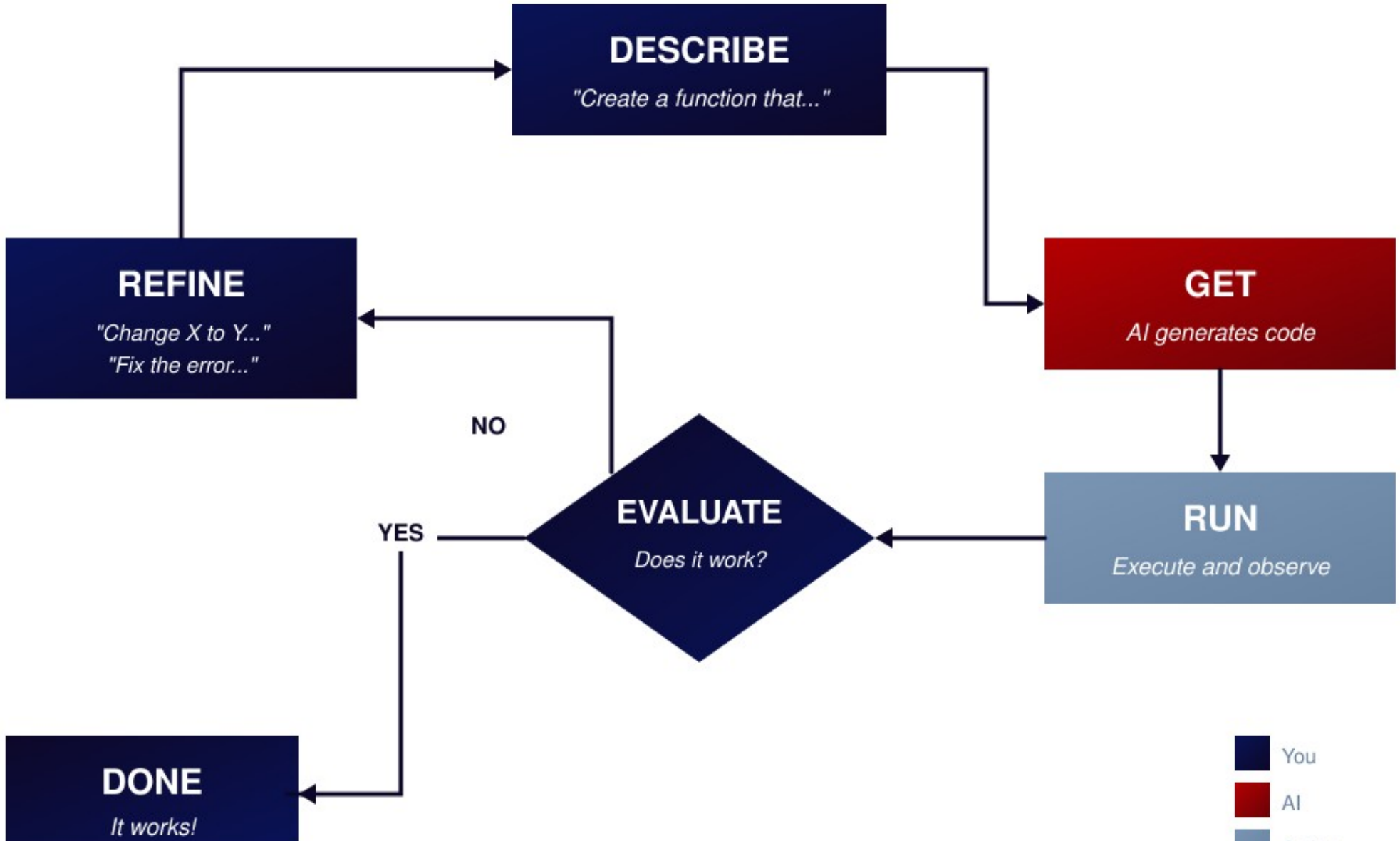
THE CORE

The fundamental partnership between human and AI



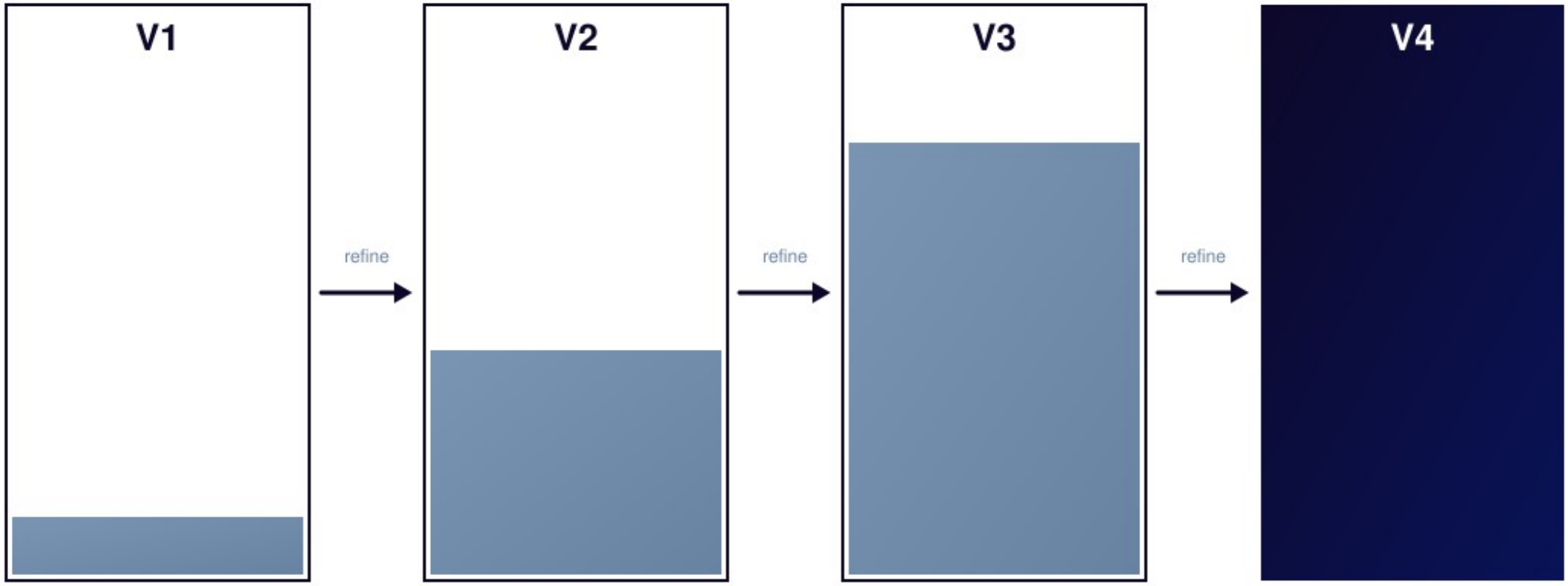
THE LOOP

The fundamental rhythm of AI-assisted development



THE DRAFT

First result is a starting point, not the destination



10%

First attempt

40%

Getting there

75%

Almost done

100%

Complete

SIX OPERATIONS

Every AI interaction falls into one of these categories

1 CREATE

"Create a file that does X"

Start from nothing

2 READ

"Explain what this code does"

Learn existing code

3 EDIT

"Change the timeout to 30s"

Modify existing

4 RUN

"Run the server"

Execute code

5 FIX

"TypeError: undefined. Fix this."

Resolve errors

6 EXTEND

"Add logging to this function"

Add features

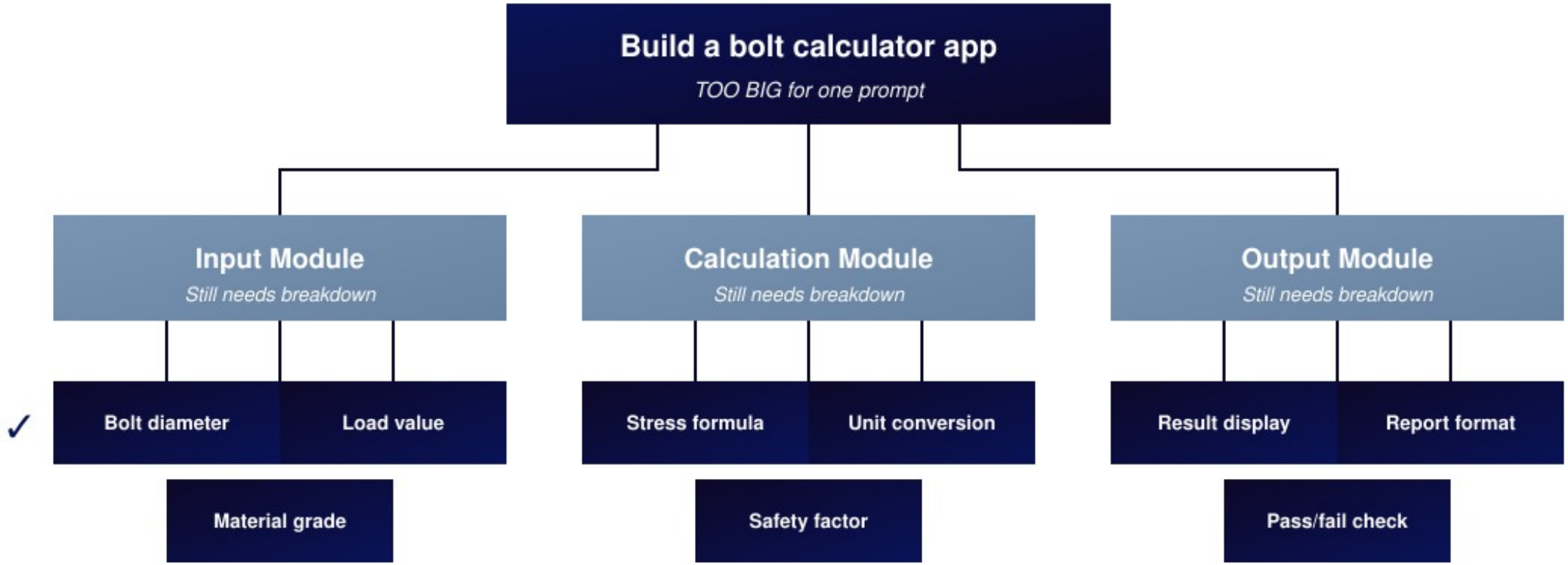
Every prompt you write is one of these six operations

PART 2

SKILLS

DECOMPOSITION

Break big tasks into AI-sized pieces



The Goal

Each piece should be small enough
to describe in ONE sentence

The Test

"Can I explain this task in one sentence?"
No → Break it down further

THE RULE

The one-sentence test for task size

"Can I explain this task in ONE sentence?"

NO

TOO BIG

Example prompt:

"Build a web app that lets users track expenses, generate reports, and export to PDF with charts and summaries"

Problem:

Multiple features in one request
AI will miss details or make wrong choices

Action:

Break it down further →

YES

JUST RIGHT

Example prompt:

"Create a function that calculates the total from a list of numbers"

Why it works:

Single, clear objective
AI knows exactly what to produce

Action:

✓ Send to AI

THE SPECTRUM

More specific prompts get better results

VAGUE MODERATE PRECISE

VAGUE

Prompt:

"Make it better"

AI thinks:

Better how? Faster? Cleaner?
More features? Different style?

Result:

Random changes, likely wrong

MODERATE

Prompt:

"Make it faster"

AI thinks:

Speed is the goal. But which part?
How much faster?

Result:

Might work, might miss the point

PRECISE

Prompt:

"Cache the API response for 5 minutes to reduce load time"

AI thinks:

Add caching. 5 min expiry.
Target: API response. Goal: speed.

Result:

✓ Exactly what you wanted

Success rate:

Specificity Formula:

WHAT + WHERE + HOW + WHY = Clear Result

FIVE COMPONENTS

The building blocks of a precise prompt

1. WHAT

The action you want

"Create a function"
"Fix this error"

2. WHERE

The location in code

"In the utils.py file"
"In the calculate function"

3. HOW

The method or approach

"Using a for loop"
"With error handling"

4. WHY

The purpose or context

"To improve performance"
"For user validation"

5. EXAMPLE

Show what you expect

"Input: [1,2,3]"
"Output: 6"

Complete Prompt Example

WHAT: "Create a function" **WHERE:** "in utils.py" **HOW:** "using recursion"

WHY: "to calculate factorials for the stats module"

EXAMPLE: "factorial(5) should return 120"

GOOD VS BAD FEEDBACK

How you respond determines what you get next

BAD FEEDBACK

✗ "It doesn't work"

AI has no idea what's wrong

✗ "This is wrong"

Wrong how? What did you expect?

✗ "Fix it"

Fix what? AI will guess (usually wrong)

✗ "That's not what I wanted"

Then what DID you want?

Result: More cycles, more frustration

VS

GOOD FEEDBACK

✓ "Getting error: TypeError on line 15"

AI knows exactly where to look

✓ "Output is 5, but should be 10"

AI knows the gap to close

✓ "Change the loop to start at 1, not 0"

AI knows exactly what to change

✓ "Add validation: reject negative numbers"

AI knows the requirement

Result: Fast fixes, working code

THE FORMULA

The perfect structure for feedback

"Got [X]. Expected [Y]. Change [Z]."

Three parts, complete information, immediate fix

GOT [X]

What actually happened

- The current output
- The error message
- The current behavior

→

EXPECTED [Y]

What should happen

- The correct output
- The desired behavior
- The expected result

→

CHANGE [Z]

What to modify

- Specific fix
- Which part
- (Optional)

Example

Got: "Function returns 15 when given input [1, 2, 3, 4, 5]"

Expected: "Should return 120 (the product, not the sum)"

Change: "Use multiplication instead of addition in the loop"

THE WINDOW

AI only sees what you show it

YOUR WORLD

Your project history

Other files in the project

Business requirements

Team coding standards

AI'S WINDOW

Only sees what's in the prompt

- ✓ Current message
- ✓ Code you paste
- ✓ Error messages you share
- ✓ Context you provide
- ✓ This conversation only

Past conversations

Your preferences

Why you're doing this

AI CANNOT:

- See your screen
- Access your files
- Remember past chats
- Know your project
- Read your mind

Unless you tell it!

SO YOU MUST:

- Paste relevant code
- Share error messages
- Explain constraints
- Give context
- State requirements

Make the window bigger!

WHAT TO SHARE

Five types of context that improve AI results

1. CONSTRAINTS

Limitations and requirements

"Must work in Python 3.8"
"No external libraries"
"Max 100 lines of code"

2. PREFERENCES

Style and approach choices

"Use descriptive variable names"
"Add comments for complex logic"
"Prefer simple over clever"

3. HISTORY

What led to this point

"We tried X but it failed because..."
"This is part of a larger system"
"Previous version had this bug..."

4. ERRORS

Exact error messages and symptoms

"TypeError: 'NoneType' has no len()"
"Happens when input is empty list"
"Line 42 in process_data function"

5. FILES

Relevant code and data

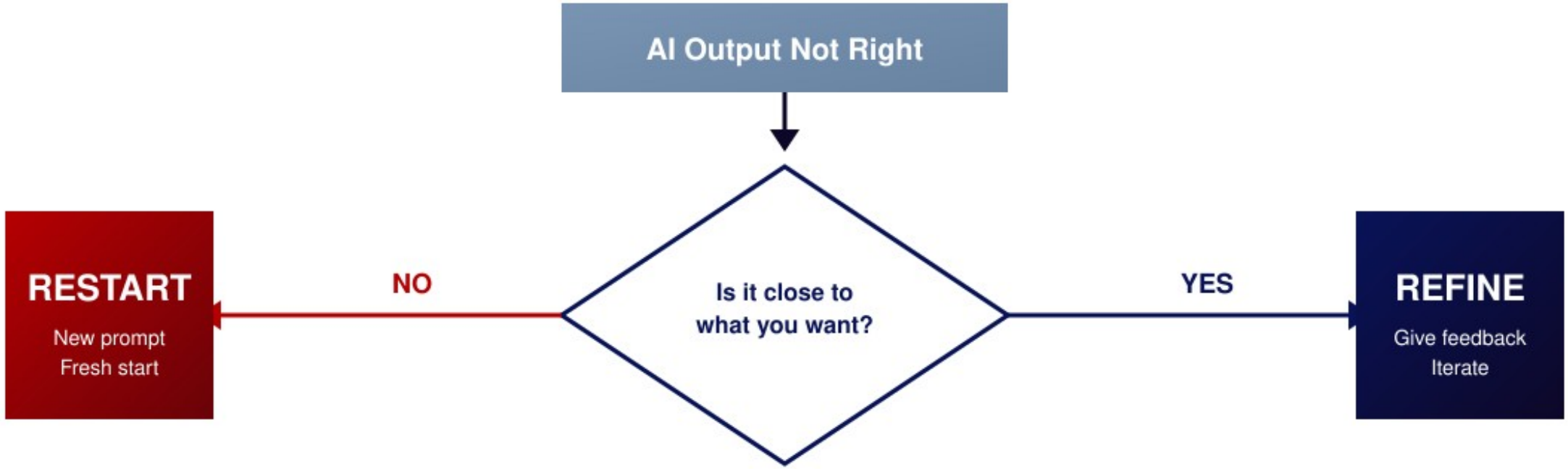
"Here's the current function: [code]"
"Sample input data: [data]"
"Related config file: [config]"

Before Sending a Prompt, Ask:

- ☐ Did I share relevant constraints?
- ☐ Did I mention my preferences?
- ☐ Did I explain the history/context?
- ☐ Did I include error messages?
- ☐ Did I paste the relevant code/files?

THE DECISION

When to keep refining vs when to restart



Signs to RESTART

- Wrong approach entirely
- Missing the point of the request
- Multiple fundamental problems
- Would need complete rewrite
- Going in circles (same errors)
- More than 5 attempts without progress

Better to start fresh!

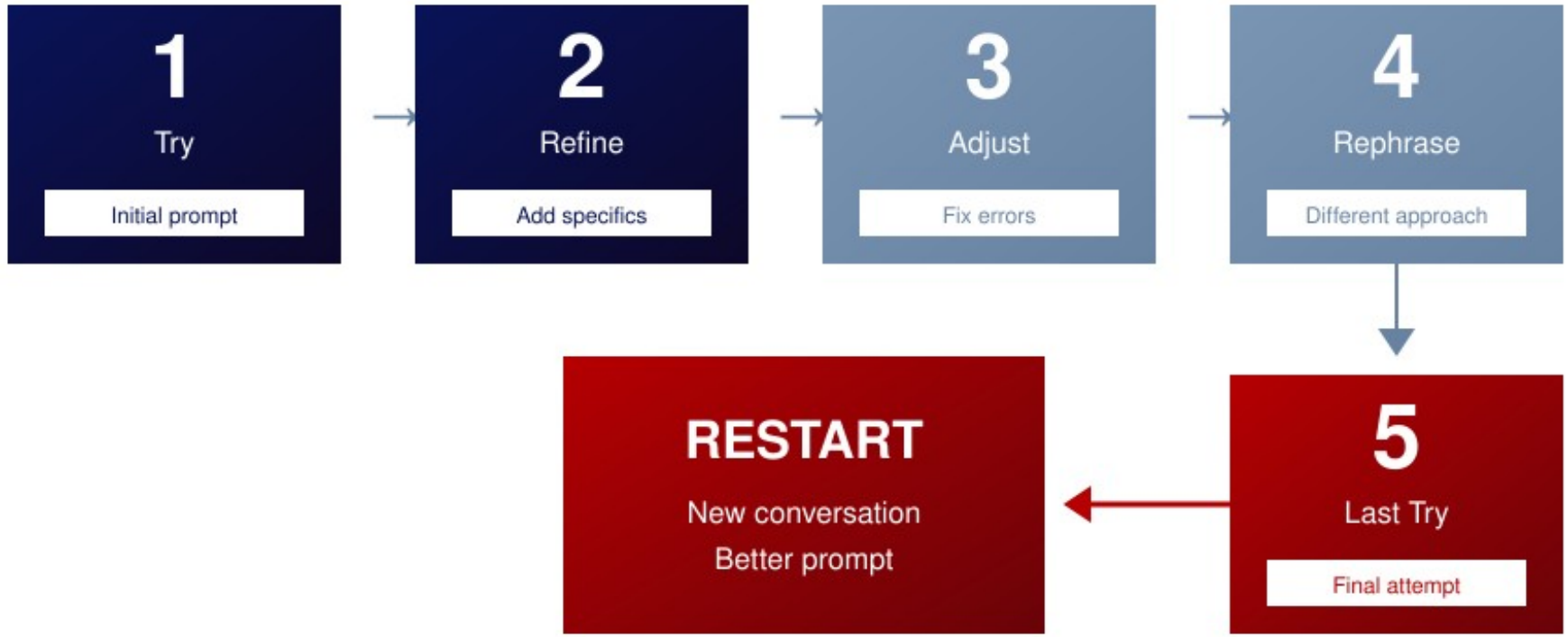
Signs to REFINE

- Structure is correct
- Small bugs or typos
- Missing one feature
- Style needs adjustment
- Logic is correct but incomplete
- Output format needs tweaking

Build on what you have!

THE 5-ATTEMPT RULE

A simple rule for when to restart



Why 5 Attempts?

Too Few (1-2)	**Just Right (3-5)**	Too Many (6+)
Giving up too early	Enough to find solutions	Diminishing returns
Missing easy fixes	Not wasting time	Wasted effort

PART 3

PROBLEMS

COMMON TRAPS

Mistakes beginners make with AI coding

TRAP 1

Blind Trust

Running code without reading it

Fix: Always review before running

Ask AI to explain what it does

TRAP 2

Vague Prompts

"Make it better" or "Fix this"

Fix: Be specific about what's wrong

State expected vs actual behavior

TRAP 3

Too Big Tasks

"Build me a whole app"

Fix: Break into small tasks

One feature at a time

TRAP 4

No Context

Not sharing error messages or code

Fix: Paste errors and code

Include relevant files

TRAP 5

Giving Up Too Fast

First attempt didn't work, quit

Fix: Iterate and refine

Give 3-5 attempts

TRAP 6

Not Learning

Copy-paste without understanding

Fix: Ask AI to explain

Learn from each solution

The Pattern

Most traps come from treating AI like magic instead of a tool

Stay engaged, stay specific, stay curious

TRUST LEVELS

How much to trust AI output in different situations

LOW TRUST — MEDIUM TRUST — HIGH TRUST

Verify Everything

When:

- Security-related code
- Financial calculations
- Safety-critical systems
- Data deletion operations

Action:

- Read every line
- Test thoroughly
- Get second opinion
- Understand fully

Review and Test

When:

- Standard features
- Business logic
- API integrations
- Database operations

Action:

- Skim the code
- Run basic tests
- Check edge cases
- Verify behavior

Quick Check

When:

- Simple utilities
- Formatting code
- Text manipulation
- Documentation

Action:

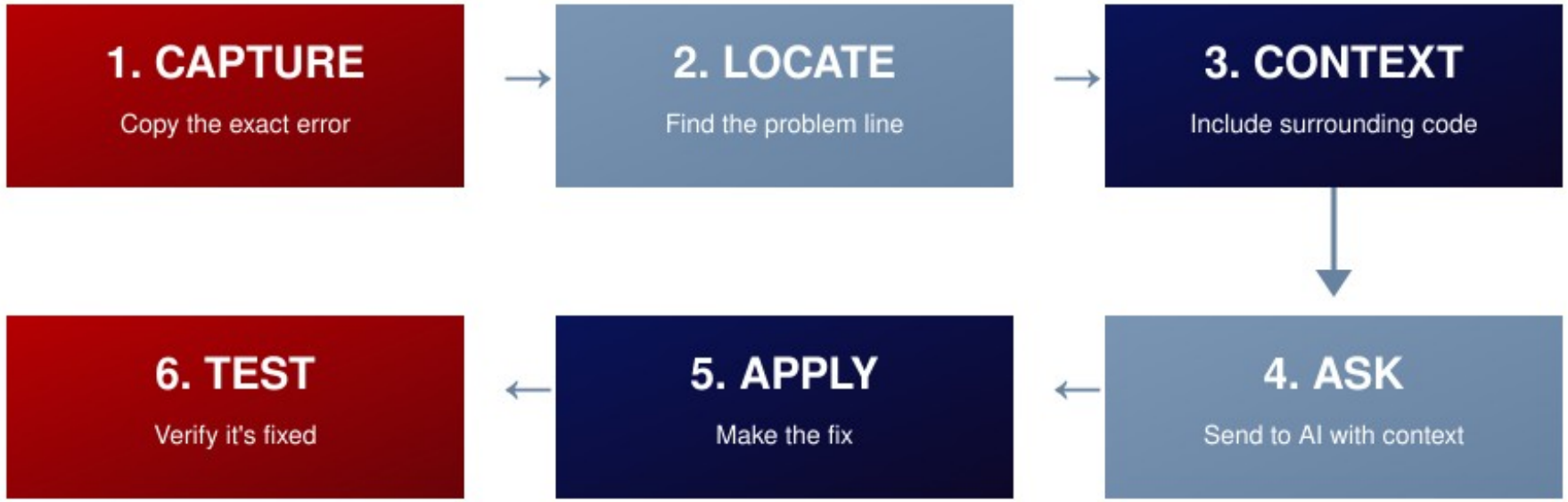
- Glance at output
- Run once
- Check it works
- Move on

Key Insight

Trust level depends on consequences of failure, not AI confidence

THE DEBUGGING FLOW

A systematic approach to fixing errors with AI



Bad Debug Request

"My code doesn't work, fix it"

Missing: error message

Missing: which line fails

Missing: relevant code

Missing: what it should do

Result: AI will guess wrong

Good Debug Request

"Error: TypeError on line 15"

"Here's lines 10-20: [code]"

"Input was: [1, 2, None]"

"Expected: sum of numbers"

"Should skip None values"

Result: Precise fix provided

ERROR TYPES

Understanding errors helps you describe them to AI

SYNTAX ERROR

Code can't even run

Missing brackets, typos

Wrong indentation

Tell AI: "Syntax error on line X"

RUNTIME ERROR

Code crashes while running

Division by zero, null access

File not found, timeout

Tell AI: "Crashes at line X with [error]"

LOGIC ERROR

Code runs but wrong result

Wrong calculation

Off-by-one, wrong condition

Tell AI: "Got X, expected Y"

Difficulty to debug (for AI and humans)

EASIEST → **MEDIUM** → **HARDEST**

What to Share with AI

Error Type	Must Include	Also Helpful
Syntax	Error message, line number	Few lines around error
Runtime	Error message, traceback, input	Full function, what triggers it
Logic	Actual output, expected output, code	Test cases, what it should do

GETTING STUCK

Recognizing when you're spinning your wheels

Warning Signs

You might be stuck if...

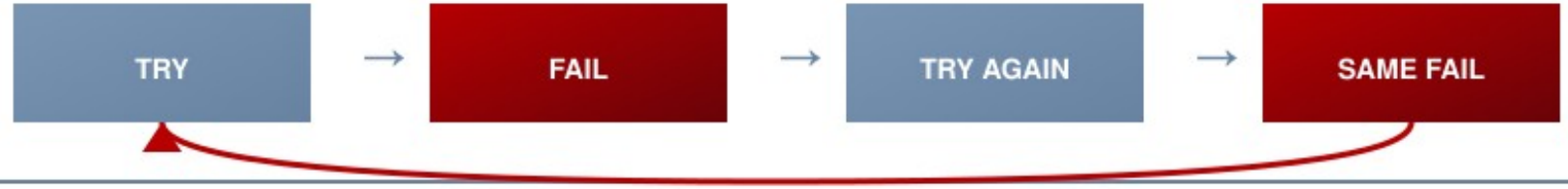
- Same error keeps coming back
- Fixing one thing breaks another
- AI gives same answer repeatedly
- More than 5 attempts, no progress
- Getting frustrated or confused

What's Happening

Common causes of getting stuck

- Wrong mental model of the problem
- Missing crucial information
- Task is actually too complex
- Solving wrong problem entirely
- Context window exhausted

The Stuck Loop



Being stuck is information: your approach needs to change

ESCAPE ROUTES

Six ways to get unstuck

1. START FRESH

New conversation, new prompt

Clears context clutter

Fresh perspective

2. BREAK SMALLER

Task too big? Split it

Solve one piece at a time

Combine when working

3. ADD CONTEXT

Share more information

Full error messages

Related files, examples

4. REPHRASE

Explain differently

Use different words

Focus on different aspect

5. ASK WHY

Get AI to explain the error

"Why is this happening?"

"What causes this?"

6. SIMPLIFY

Make a minimal example

Strip to core problem

Add complexity later

Quick Decision Guide

Same error over and over?

Try: Start Fresh

Task seems impossible?

Try: Break Smaller

AI misunderstanding you?

Try: Rephrase

Don't understand error?

Try: Ask Why

PART 4

BUILDING

LEARNING TO READ CODE

You don't need to write code to understand it

Reading code is like reading a recipe - you can follow along

without being a chef

LOOK FOR

Patterns you can recognize

Names

Variables describe what they hold

Structure

Indentation shows grouping

Keywords

if, for, while, return

Comments

Human explanations in code

Flow

Top to bottom, branching

ASK AI

Questions to understand code

"What does this code do?"

"Explain line by line"

"What is [variable] for?"

"Why is this needed?"

"What would break this?"

"Simplify this for me"

AI is your code interpreter!

SKIP FOR NOW

Don't worry about these yet

Memorizing syntax

Understanding every detail

Advanced patterns

Optimization tricks

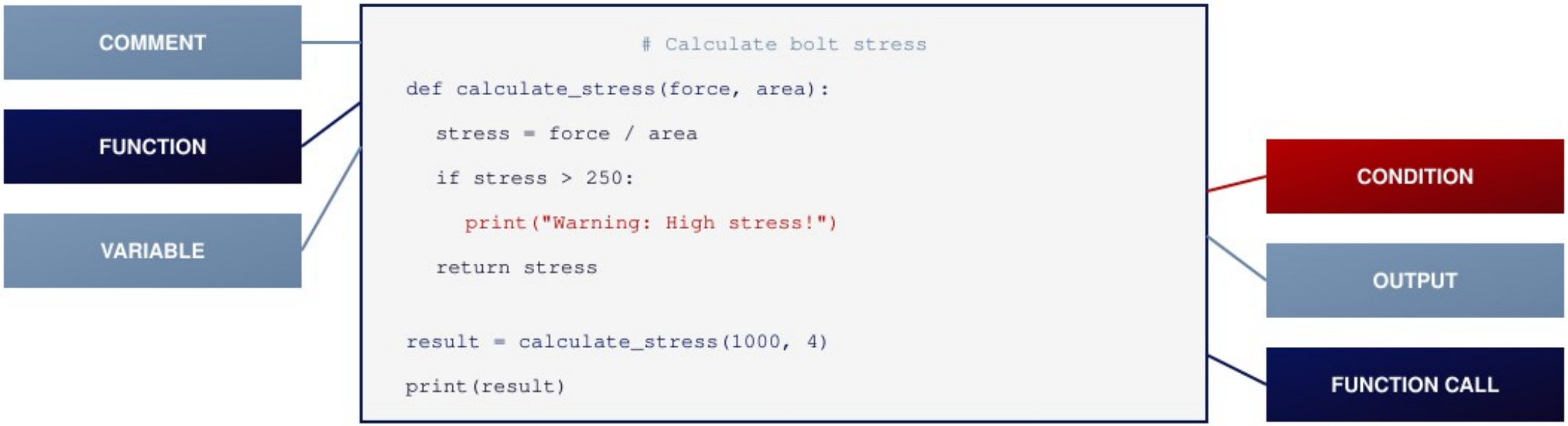
Framework internals

Focus on the big picture first

Goal: Understand what code does, not memorize how to write it

CODE ANATOMY

The universal parts of every program



COMMENTS

Notes for humans

Ignored by computer

Start with # or //

Like sticky notes on code

FUNCTIONS

Reusable blocks of code

Take inputs, give outputs

Named for what they do

Like machines in a factory

CONDITIONS

Make decisions

if/else, true/false

Branch the flow

Like a fork in the road

Every program is made of these same building blocks

VARIABLES

Named containers for data

Variables are like labeled boxes - the name tells you what's inside

The Concept

bolt_diameter

12.5

The Code

```
bolt_diameter = 12.5
```

name = value

The Meaning

"Store 12.5 in a box labeled bolt_diameter"

NUMBERS

```
count = 42
price = 19.99
stress = 250.5
```

TEXT

```
name = "M12"
grade = "8.8"
status = "OK"
```

TRUE/FALSE

```
is_valid = True
has_error = False
passed = True
```

LISTS

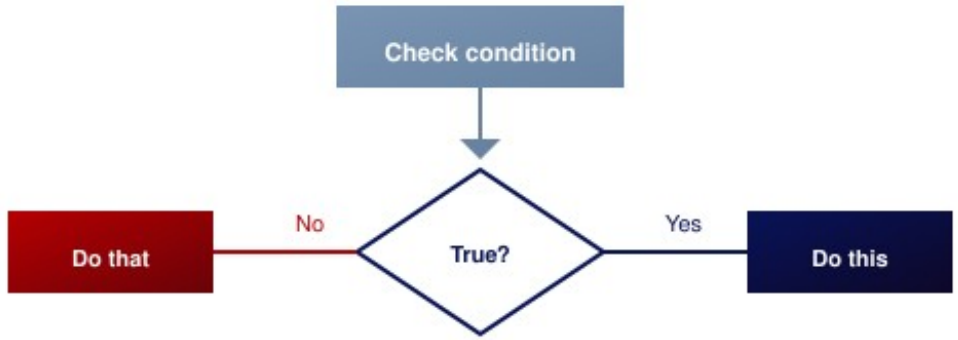
```
sizes =
[8, 10, 12]

Multiple items
```

Naming rules: No spaces (use_underscores), start with letter, be descriptive

CONTROL FLOW

How programs make decisions and repeat actions

IF / ELSE (Decisions)

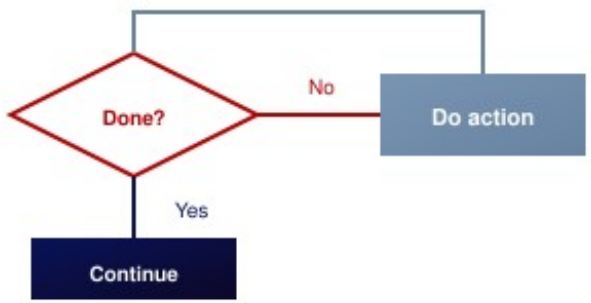



```
if stress > 250: print("Fail") else: print("Pass")
```

FOR LOOP (Repeat N times)


```
for size in [8, 10, 12]: print(size)
```

WHILE LOOP (Repeat until)




```
while error > 0.01:
    refine()
```

Summary

if/else	Choose between options
for	Do something N times
while	Keep going until done

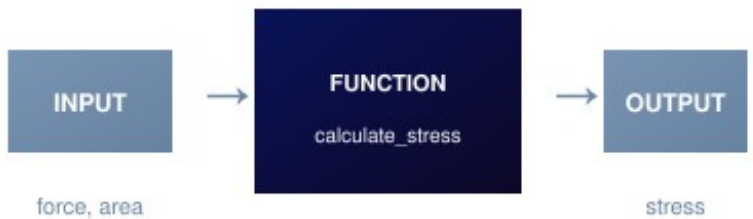
These three patterns handle most logic!

FUNCTIONS

Reusable machines that do one job

A function is like a machine: put something in, get something out

The Machine



Call it anywhere, reuse forever

The Code

```
# Define the function
def calculate_stress(force, area):
   return force / area
```

```
# Use the function
result = calculate_stress(1000, 4)
```

result is now 250

Why Use Functions?

- Reuse code (write once)
- Organize logic (clear names)
- Test easily (isolated units)
- Fix in one place

Function Parts

- Name (what it does)
- Parameters (inputs)
- Body (the work)
- Return (output)

Ask AI To...

- "Create a function that..."
- "Explain this function"
- "What does it return?"
- "Add error handling"

LIBRARIES

Pre-built code you can use

Libraries are toolboxes - don't build hammers, just use them

Without Libraries

```
# Calculate average manually
total = 0
for num in numbers:
   total = total + num
average = total / len(numbers)
```

5 lines, reinventing the wheel

With Libraries

```
import statistics

average = statistics.mean(numbers)
```

1 line, tested, reliable

Math/Science

- **numpy** - Arrays, matrices
- **scipy** - Engineering calcs
- **pandas** - Data tables
- **math** - Basic math ops

Visualization

- **matplotlib** - Charts, plots
- **plotly** - Interactive plots
- **seaborn** - Statistical viz

Files/Automation

- **os** - File operations
- **json** - Read/write JSON
- **csv** - Spreadsheets
- **datetime** - Dates/times

DATA STRUCTURES

Ways to organize multiple pieces of data

LIST

Ordered collection


```
bolt_sizes = [8, 10, 12, 16]
```

Access by position: bolt_sizes[0]

Add/remove items

Like a shopping list

DICTIONARY

Key-value pairs

```
"name"     → "M12"
"grade"    → "8.8"
"diameter"  → 12.0
```

```
bolt = {"name": "M12", ...}
```

Access by name: bolt["grade"]

Like a real dictionary: word → definition

TUPLE

Fixed, unchangeable list

```
(100.0, 200.5, 50.2)
```

```
coords = (100.0, 200.5, 50.2)
```

Can't change after creation

Good for coordinates, constants

Like a sealed envelope

When To Use What

LIST

Collection of similar items

Need to add/remove items

bolt_sizes, measurements

DICTIONARY

Named properties

Need to look up by name

bolt specs, config settings

TUPLE

Fixed group of values

Should never change

x,y,z coordinates

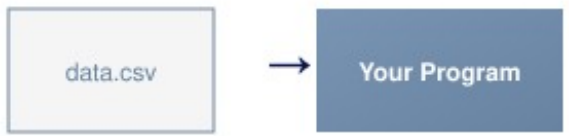
Ask AI: "What data structure should I use for [description]?"

FILES AND I/O

Reading from and writing to files

Files are permanent storage - data survives when program ends

READING FILES




```
# Read entire file
with open("data.csv") as file:
    content = file.read()
```

Data is now in memory as 'content'

WRITING FILES


```
# Write to file
with open("results.txt", "w") as file:
    file.write("Stress: 250 MPa")
```

Data saved permanently to disk

Text Files

.txt - Plain text

.csv - Spreadsheet data

.json - Structured data

Human readable

Data Files

.xlsx - Excel files

.xml - Structured format

.db - Databases

Use libraries to read

Ask AI

"Read this CSV file into a list"

"Write results to a new file"

"Parse this JSON data"

AI handles the details

TESTING

Verifying your code actually works

Testing = Quality control for code (like checking parts before assembly)

Manual Testing

Run the code

Check the output

Try different inputs

Good for quick checks

Automated Testing

Write test code

Runs automatically

Catches regressions

Good for reliability

Edge Case Testing

Empty inputs

Extreme values

Invalid data

Good for robustness

Simple Test Example

```
# Test the function
result = calculate_stress(1000, 4)
expected = 250

assert result == expected, "Test failed!"
```

Test Results

PASS

Code works!

FAIL

Fix needed

Ask AI: "Write tests for this function"

PROJECT STRUCTURE

Organizing files for bigger projects

A good folder structure = a well-organized toolbox

Typical Project Structure

```
bolt_calculator/
    main.py                 Entry point
    calculations.py         Core functions
    utils.py                Helper functions

    data/                   Input files
        bolt_specs.csv
        materials.json

    output/                 Results
        results.txt

    tests/                  Test files
        test_calculations.py

    README.md               Documentation
```

Why Organize?

- Find files quickly
- Separate concerns (logic vs data vs tests)
- Easy to share and collaborate
- Professional standard

File Naming Rules

- Lowercase: calculations.py (not Calculations.py)
- Underscores: bolt_stress.py (not bolt-stress.py)
- Descriptive: test_calculations.py (not test1.py)

Names should describe content

PART 5

APPLICATION

YOUR FIRST PROJECT

Applying everything you've learned

Start small, make it work, then expand

1. PLAN

Define your goal

What should it do?

What inputs/outputs?

What's the simplest version?

Write it in one sentence before coding

2. BUILD

Work with AI

Start with core feature

Test as you go

Add features one by one

Get it working before making it perfect

3. REFINE

Polish it

Handle edge cases

Add error messages

Clean up code

Ask AI to review and improve

Example First Project: Bolt Stress Calculator

Plan: "Calculate stress from force and area, warn if too high"

Build: 1. Create function 2. Add input 3. Add warning 4. Format output

Refine: Handle zero area, add units, save to file

PROJECT CHECKLIST

Everything to verify before calling it done

Before Starting

- [ ] Goal clearly defined?
- [ ] Inputs and outputs identified?
- [ ] Broken into small tasks?

While Building

- [ ] Testing each piece?
- [ ] Understanding AI's code?
- [ ] Saving working versions?

Before Finishing

- [ ] Works with normal input?
- [ ] Works with edge cases?
- [ ] Has clear error messages?

Quality Check

- [ ] Code readable?
- [ ] Names descriptive?
- [ ] Comments where needed?

Final Questions

- [ ] Can someone else use this without your help?
- [ ] Would you trust this code in production?

GROWING YOUR SKILLS

The path from beginner to confident builder

BEGINNER → **COMPETENT** → **CONFIDENT**

Stage 1: Copy

What you do:

Follow AI's code exactly

Ask for explanations

Learn patterns

Goal:

Get code working

Understand what it does

Stage 2: Modify

What you do:

Adjust AI's code yourself

Combine pieces

Fix small issues alone

Goal:

Adapt solutions

Build on existing code

Stage 3: Create

What you do:

Design solutions

Use AI as assistant

Guide the process

Goal:

Build anything

AI accelerates, not replaces

The Secret

Every expert was once a beginner who kept building

Each project teaches you something new

RESOURCES

Where to go when you need help

AI Tools

Claude (Anthropic)
Great for explanations

ChatGPT (OpenAI)
Good for code generation

GitHub Copilot
In-editor assistance

Use what works for you

Learning More

Python.org
Official documentation

Stack Overflow
Q&A for specific errors

YouTube tutorials
Visual learning

Search when stuck

Practice Ideas

Automate calculations
Turn Excel work into code

Data visualization
Create charts from data

File processing
Batch convert/rename

Solve real problems

When You're Stuck: The Workflow

1. Ask AI → **2. Search Web** → **3. Rephrase** → **4. Break Down**

Most problems solved in 1-2 steps

Your best resource: curiosity and willingness to experiment

THE JOURNEY

What you've learned and where you're going

You Started Here

"I can't code"

What You Learned

The Loop: Describe → Get → Run → Evaluate

Decomposition: Big tasks → Small tasks

Communication: Specific prompts → Better results

You Are Here

"I build with AI"

Skills You Now Have

Read and understand code

Communicate with AI effectively

Debug and fix problems

Build working solutions

What's Next

Pick a real problem you want to solve

Build your first real project

Share it with someone

Then build another one!

You don't need to know everything to build something useful

The Truth About Coding

KEY TAKEAWAYS

Everything in one page

YOU + AI = BUILDER

You provide direction and judgment. AI provides code and speed.

1. THE LOOP

Describe → Get → Run → Evaluate

Iterate until it works

2. DECOMPOSE

Big task → Small tasks

One sentence = right size

3. BE SPECIFIC

Precise prompts = Better results

What, where, how, why, example

4. GIVE FEEDBACK

Got X, Expected Y, Change Z

Tell AI what's wrong specifically

5. SHARE CONTEXT

AI only knows what you tell it

Code, errors, constraints

6. KNOW WHEN TO RESTART

5 attempts, then fresh start

Cut losses and try again

The Formula for Success

Clear Goal + Small Steps + Iteration + Persistence = Working Code

Now Go Build Something!

You have everything you need.