

# Build Your Own AI Code Generator From Scratch — A PocketFlow Tutorial!



ZACHARY HUANG

MAY 24, 2025



14



5



3

SI



Ever wished you could just describe a coding problem and have an AI automatically generate comprehensive test cases, implement the solution, and iteratively improve it until everything works perfectly? This guide shows you how to build exactly that using the [PocketFlow AI Code Generator Cookbook!](#)

## 1. From Problem Generation

Looks like an article worth saving!

Option



le

Hover over the brain icon or use hotkeys to save with Memex.

Picture this: You're s

Remind me later

Hide Forever

oi

through the usual grind of writing test cases, coding a solution, debugging fa''

and repeating the cycle until everything works, you just paste the problem description and watch an AI system automatically:

1. **Generate comprehensive test cases** including all the edge cases you'd probably forget
2. **Implement a clean solution** based on the problem requirements
3. **Test everything automatically** with detailed pass/fail analysis
4. **Intelligently debug** by deciding whether the tests are wrong or the code needs fixing
5. **Iterate until perfect** - all tests passing with a working solution

This isn't science fiction - it's exactly what we're going to build together!

Thanks for reading Pocket Flow! Subscribe for free to receive new posts and support my work.

---

***What you'll learn:** By the end of this tutorial, you'll have built a complete AI development assistant that can take any coding problem and automatically work through the entire development cycle. We'll use [PocketFlow](#) to orchestrate the workflow, making the complex process surprisingly manageable and easy to understand.*

---

## 2. How AI Code Generation Works: Step-by-Step with Two Sum

Wait, why not just ask ChatGPT? You might think: "Why build all this? Can't I just paste the problem in?"

**Looks like an article worth saving!**

Option

Q

Sure, you could! But :

Hover over the brain icon or use hotkeys to save with Memex.

- ChatGPT gives you a solution, but it's often wrong or incomplete
- If there's a bug, you have to spot it manually

Remind me later

Hide Forever

- Edge cases? You better hope you thought of them

What we need instead is a **smart workflow**. An AI system that can write code, test, catch its own mistakes, and fix them. Just like how human developers work: code → test → debug → improve → repeat.

In this tutorial, we'll learn hands-on how to build exactly this kind of intelligent workflow using the classic **Two Sum** problem as our example.

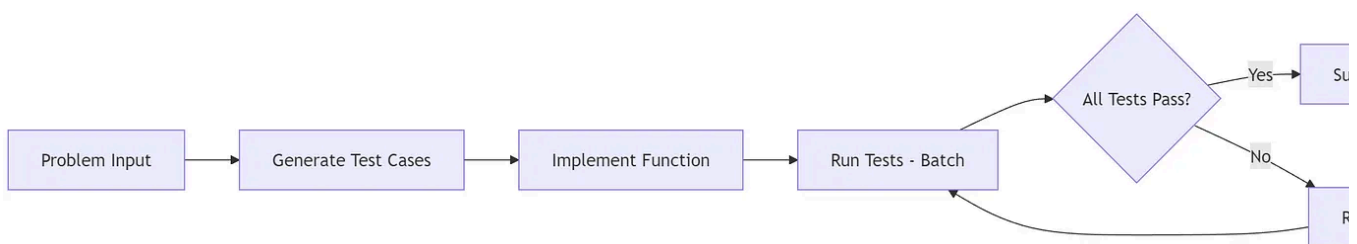
First, let's look at the problem we'll be solving:

Given an array of integers `nums` and an integer `target`, return indices of the two numbers such that they add up to `target`.

Example: `nums = [2,7,11,15]`, `target = 9` → Output: `[0,1]`

## Our Smart Workflow in Action

So how does our intelligent system actually work? Here's the high-level flow we're going to build:



Notice the key insight: when tests fail, we don't just give up. We analyze what went wrong and intelligently fix it, then test again. This creates a feedback loop that keeps improving until everything works.

Now let's see this workflow in action.

**Looks like an article worth saving!**

Option

Q

Hover over the brain icon or use hotkeys to save with Memex.

## Step 1: Generate Test Cases

Remind me later

Hide Forever

The first thing our system does is create comprehensive test cases (way more than the given examples):

=== Generated 7 Test Cases ===

1. Basic case: [2, 7, 11, 15], target=9 → [0, 1]
2. Duplicates: [3, 3], target=6 → [0, 1]
3. Negatives: [-1, -2, -3, -4, -5], target=-8 → [2, 4]
4. Zero case: [0, 4, 3, 0], target=0 → [0, 3]
5. End solution: [1, 2, 3, 4, 5, 6], target=11 → [4, 5]
6. Larger array: [5, 75, 25, 45, 42, 2, 11, 9, 55, 12], target=14 → [2, 6]

**Why this rocks:** It automatically thinks of edge cases like duplicates, negatives, and larger arrays. No more forgetting to test the tricky scenarios!

## Step 2: Implement Function

With test cases ready, our system writes clean, efficient code:

```
def run_code(nums, target):
    num_to_index = {}
    for i, num in enumerate(nums):
        complement = target - num
        if complement in num_to_index:
            return [num_to_index[complement], i]
        num_to_index[num] = i
    return []
```

**Cool detail:** It chose the smart  $O(n)$  hash map approach instead of the obvious (but slow) nested loop. The AI actually thinks about efficiency!

## Step 3: Run Test

**Looks like an article worth saving!**

Option



Hover over the brain icon or use hotkeys to save with Memex.

Time for the moment

Remind me later



Hide Forever

```
=== Test Results: 6/7 Passed ===  
Failed: Larger array case  
Expected: [2, 6], Got: [0, 7]
```

Uh oh! One test failed. But here's where the magic happens...

## Step 4: Intelligent Debugging

Instead of panicking, our system analyzes what went wrong:

- Array: [5, 75, 25, 45, 42, 2, 11, 9, 55, 12], target: 14
- Position 0: 5 + Position 7: 9 = 14  (what our code found)
- Position 2: 25 + Position 6: 11 = 36  (what the test expected)

**Plot twist:** The test case was wrong, not the code! Our system figures this out and it:

```
=== Smart Fix ===  
Test 6: Expected [2, 6] → [0, 7]
```

## Step 5: Victory!

After the fix, our system runs the tests again:

```
=== Test Results: 7/7 Passed ===
```

**The key insight:** Instead of blindly "fixing" perfectly good code, our system correctly identified that the test was wrong. This prevents us from making systems from basic assumptions.

**Looks like an article worth saving!**

Option Q

Hover over the brain icon or use hotkeys to save with Memex.

**This feedback loop is**  
intelligently decides to

Remind me later

Hide Forever

nd  
ne-

ChatGPT answer!

*Ready to build this? Let's dive into the tools that make it possible.*

### 3. Building Intelligent Workflows with PocketFlow: The Perfect Toolkit

Now that we understand the conceptual process, how do we actually *build* a system that can orchestrate these steps, handle the decision-making, and manage the iterative improvements? This is where [PocketFlow](#) becomes our secret weapon!

Think of PocketFlow as your development workshop manager. Just like a well-organized workshop has different workstations for different tasks, PocketFlow helps you create specialized "workers" for each job and coordinate them smoothly.

#### The Workshop Approach: Nodes as Specialist Workers

In PocketFlow, each major task becomes a **Node** - think of it as a specialist worker who's really good at one specific job:

```
class Node:
    def __init__(self): self.successors = {}
    def prep(self, shared): pass
    def exec(self, prep_res): pass
    def post(self, shared, prep_res, exec_res): pass
    def run(self, shared): p=self.prep(shared); e=self.exec(p); return self.post(shared,p,e)
```

**Looks like an article worth saving!**

Option

Q

This is like a simple :  
decide what's next). I

Hover over the brain icon or use hotkeys to save with Memex.

d  
ex

Remind me later

Hide Forever

#### The Shared Workspace: Information that

Think of this as a shared whiteboard where all workers can read and write:

```
shared = {
    "number": 10,
    "result": None
}
```

Each worker grabs what they need from this shared space and leaves their results others.

## Quick Example: Simple Math Pipeline

Let's build a tiny workflow that takes a number, adds 5, then multiplies by 2:

```
class AddFive(Node):
    def prep(self, shared):
        return shared["number"]

    def exec(self, number):
        return number + 5

    def post(self, shared, prep_res, result):
        shared["number"] = result
        print(f"Added 5: {result}")
        return "default"
```

```
class MultiplyByTwo(Node):
    def prep(self, shared):
        return shared["number"]

    def exec(self, number):
        return number * 2
```

```
def post(self,
shared["number"] = result
print(f"Multiplied by 2: {result}")
return "default"
```

**Looks like an article worth saving!**

Option Q

Hover over the brain icon or use hotkeys to save with Memex.

Remind me later

Hide Forever

Each worker does one simple job and passes the result along.

## Connecting Workers: The Assembly Line

```
# Connect workers in sequence (the >> means "then go to")
add_worker = AddFive()
multiply_worker = MultiplyByTwo()

add_worker >> multiply_worker

# Create the manager and run it
math_flow = Flow(start=add_worker)
shared = {"number": 10}
math_flow.run(shared)

# Output:
# Added 5: 15
# Final result: 3
```

It's like saying: "First add 5, then multiply by 2."

**The Beauty:** Each worker is simple and focused. Need to change the math? Just swap out a worker. Want to add more steps? Connect more workers. The smart behavior comes from how they work together, like musicians in an orchestra!

For our AI code generator, we'll have workers like:

- **TestCaseGenerator:** Creates comprehensive test suites
- **CodeImplementer:** Writes clean implementations
- **TestRunner:** Executes tests efficiently
- **SmartReviewer:**

**Looks like an article worth saving!**

Option



Hover over the brain icon or use hotkeys to save with Memex.

Sai

Remind me later

Hide Forever



## 4. Building the Code Generator with PocketFlow Nodes

Now let's build the actual workers! Each one handles a specific job in our development workflow.

### The Shared Memory Structure

Our workers communicate through a simple dictionary that holds everything needed for the code generation workflow:

```
shared = {
    "problem": "Problem description...",
    "test_cases": [{"name": "Basic case", "input": {...}, "expected":
"result"}],
    "function_code": "def run_code(...): ...",
    "test_results": [{"passed": True, "error": None}],
    "iteration_count": 0
}
```

Think of this as a desk where each worker leaves notes for the next person. We need the original problem description, test cases to validate our solution, the actual code implementation, test results to know what passed or failed, and an iteration count to prevent infinite loops.

### Worker 1: The Test Case Generator

```
class GenerateTestCases(Node):
    def prep(self, shared):
        return shared

    def exec(self, shared):
        prompt = """
YAML..."""
        response = self.run(prompt)
        return yaml.safe_load(response)
```

**Looks like an article worth saving!**

Option

Q

Hover over the brain icon or use hotkeys to save with Memex.

Remind me later

Hide Forever

```
def post(self, shared, prep_res, result):
    shared["test_cases"] = result["test_cases"]
    print(f"Generated {len(result['test_cases'])} test cases")
```

This worker reads the problem, asks an AI to create test cases in YAML format, processes the response, and stores the test cases. Simple as that!

## Worker 2: The Code Implementer

```
class ImplementFunction(Node):
    def prep(self, shared):
        return shared["problem"], shared["test_cases"]

    def exec(self, inputs):
        problem, test_cases = inputs
        prompt = f"Implement: {problem}\nTests: {test_cases}\nFunction must be named 'run_code'"
        response = call_llm(prompt)
        return yaml.safe_load(response)

    def post(self, shared, prep_res, code):
        shared["function_code"] = code
        print("Function implemented")
```

This worker takes the problem and test cases, asks the AI to write code, and stores the result. The AI sees both the problem AND the tests, so it writes better code.

## Worker 3: The Test Runner (Batch Processing)

```
class RunTests(BatchWorker):
    def prep(self, shared):
        code = shared["function_code"]
        tests = shared["test_cases"]
        return [code, tests]
```

```
def exec(self, test_data):
```

**Looks like an article worth saving!**

Option Q

Hover over the brain icon or use hotkeys to save with Memex.

Remind me later

Hide Forever

```

code, test_case = test_data
output, error = execute_python(code, test_case["input"])
return {
    "test_case": test_case,
    "passed": output == test_case["expected"] and not error,
    "actual": output,
    "error": error
}

```

This uses **BatchNode** to test all cases one-by-one. Each test gets the code and one case, runs it safely, and returns the result.

The `post` method handles the results:

```

def post(self, shared, prep_res, results):
    shared["test_results"] = results
    passed = sum(1 for r in results if r["passed"])
    print(f"Tests: {passed}/{len(results)} passed")

    if all(r["passed"] for r in results):
        return "success"
    elif shared.get("iteration_count", 0) >= 5:
        return "max_iterations"
    else:
        return "failure"

```

This counts passes/fails, prints results, and signals what to do next: celebrate success, give up after too many tries, or fix the problems.

## Worker 4: The Smart Reviewer

```

class Revise(Node):
    def prep(self):
        failures = 0
        r["passed"] = 0
        return {
            "problem": shared["problem"],

```

**Looks like an article worth saving!**

Option

Q

Hover over the brain icon or use hotkeys to save with Memex.

Remind me later

Hide Forever

```

        "code": shared["function_code"],
        "failures": failures
    }

```

```

def exec(self, data):
    prompt = f"Problem: {data['problem']}\nCode: {data['code']}\nFailures: {data['failures']}\nWhat should I fix?"
    response = call_llm(prompt)
    return yaml.safe_load(response)

```

This worker gathers all the context about what failed, asks the AI to analyze and decide what to fix, then returns structured fixes.

```

def post(self, shared, prep_res, fixes):
    if "test_cases" in fixes:
        # Update specific test cases
        for index, new_test in fixes["test_cases"].items():
            shared["test_cases"][int(index)-1] = new_test

    if "function_code" in fixes:
        shared["function_code"] = fixes["function_code"]

    shared["iteration_count"] = shared.get("iteration_count", 0) + 1
    print("Applied fixes")

```

This applies the AI's suggested fixes to either test cases, code, or both. The AI might say "test case 3 has wrong expected output" or "the algorithm needs a different approach."

## Connecting Everything Together

```

def create_code_generator():
    # Create world
    generate_test_cases
    implement = Implement()
    run_tests = RunTests()
    revise = Revise()

```

**Looks like an article worth saving!**

Option

Q

Hover over the brain icon or use hotkeys to save with Memex.

Remind me later

Hide Forever

```
# Connect the assembly line
generate_tests >> implement >> run_tests
run_tests - "failure" >> revise >> run_tests

return Flow(start=generate_tests)
```

The magic happens in the connections: normal flow goes generate → implement → test. But if tests fail, we go to the reviewer, who fixes things and sends us back to testing.

**Why This Works:** Each worker is laser-focused on one job. The intelligence emerges from their coordination - like a jazz band where everyone knows their part but can improvise together!

## 5. Conclusion: Code Smarter, Not Harder!

And there you have it! You've journeyed from staring at a coding problem to building an AI that generates tests, writes code, and even debugs its own mistakes. We peeked under the hood to see how problems get understood (test generation), solved (smart implementation), tested (batch execution), and improved (intelligent revision), all in a smooth, iterative loop.

Hopefully, this tutorial has shown you that building intelligent coding assistants doesn't have to feel like rocket science. With a friendly toolkit like **PocketFlow**, complex AI workflows become way more manageable. PocketFlow is like your helpful conductor, letting you focus on the fun part – what each piece of your AI system should *do* – instead of getting tangled in the tricky coordination. It's all about breaking big ideas into small, focused **Nodes** and letting the **Flow** make sure they all work together intelligently.

**Looks like an article worth saving!**

Option



Now, it's your turn to explore the [PocketFlow](#) around with it, maybe

Hover over the brain icon or use hotkeys to save with Memex.

Remind me later

Hide Forever

you  
ly  
ch

it tackles different problems. What if you made the test generator more creative, or added performance benchmarking? What other coding challenges can you automate?

The world of AI-assisted development is evolving rapidly, and with tools like PocketFlow, you're all set to jump in and be a part of it. Go on, give your next coding project an AI brain – we can't wait to see what intelligent tools you build!

---

*Ready to explore more or need some help? You can dive deeper by checking out the main [PocketFlow repository on GitHub](#) for the core framework, other neat examples, and all latest updates. The complete code for the AI code generator we discussed is right in the [pocketflow-code-generator](#) directory within the PocketFlow cookbook.*

---

Thanks for reading Pocket Flow! Subscribe for free to receive new posts and support my work.



14 Likes • 3 Restacks

← Previous

Next

## Discussion about this post

Comments Restacks



Write a comment

**Looks like an article worth saving!**

Option



Hover over the brain icon or use hotkeys to save with Memex.



Franco May 29

♥ Liked by Zachary Huang

Remind me later

Hide Forever

What a great tutorial! This is very interesting. But is it really useful in corporate and multidisciplinary environments? As a programmer, I don't see any way to do something that first indexes an enormous project, such as a real-life application for an international client, and then, knowing the context of the entire project, identifies styles and patterns used by the team, as well as functions and variables used in the project, such as utility functions created by the team. That is, the AI can create a function, but if a function has already been created for that case, use it; don't create a new one yourself. Also, refactoring legacy code or parts of the code without affecting or corrupting the integration branch or Git workflow for your other colleagues is impossible. Testing every line of code well and generating quality, production-ready code without duplication and without creating conflicts between branches is something that is currently impossible.

♡ LIKE (1)    💬 REPLY

1 reply by Zachary Huang



May 27

♡ Liked by Zachary Huang

Thank you for sharing this amazing PocketFlow tutorial! The intelligent code generator described in the article is truly impressive, especially its implementation and application in the Python environment. However, I'm currently working primarily with Java for my development project. I'm curious to know: Are there any plans for a PocketFlow-Java tutorial in the future? If so, I'd be excited to learn how to apply similar intelligent workflows to my Java projects, which would significantly boost my development efficiency!

♡ LIKE (1)    💬 REPLY

1 reply by Zachary Huang

3 more comments...

**Looks like an article worth saving!**

Option Q

Hover over the brain icon or use hotkeys to save with Memex.

© 2025

Remind me later

Hide Forever