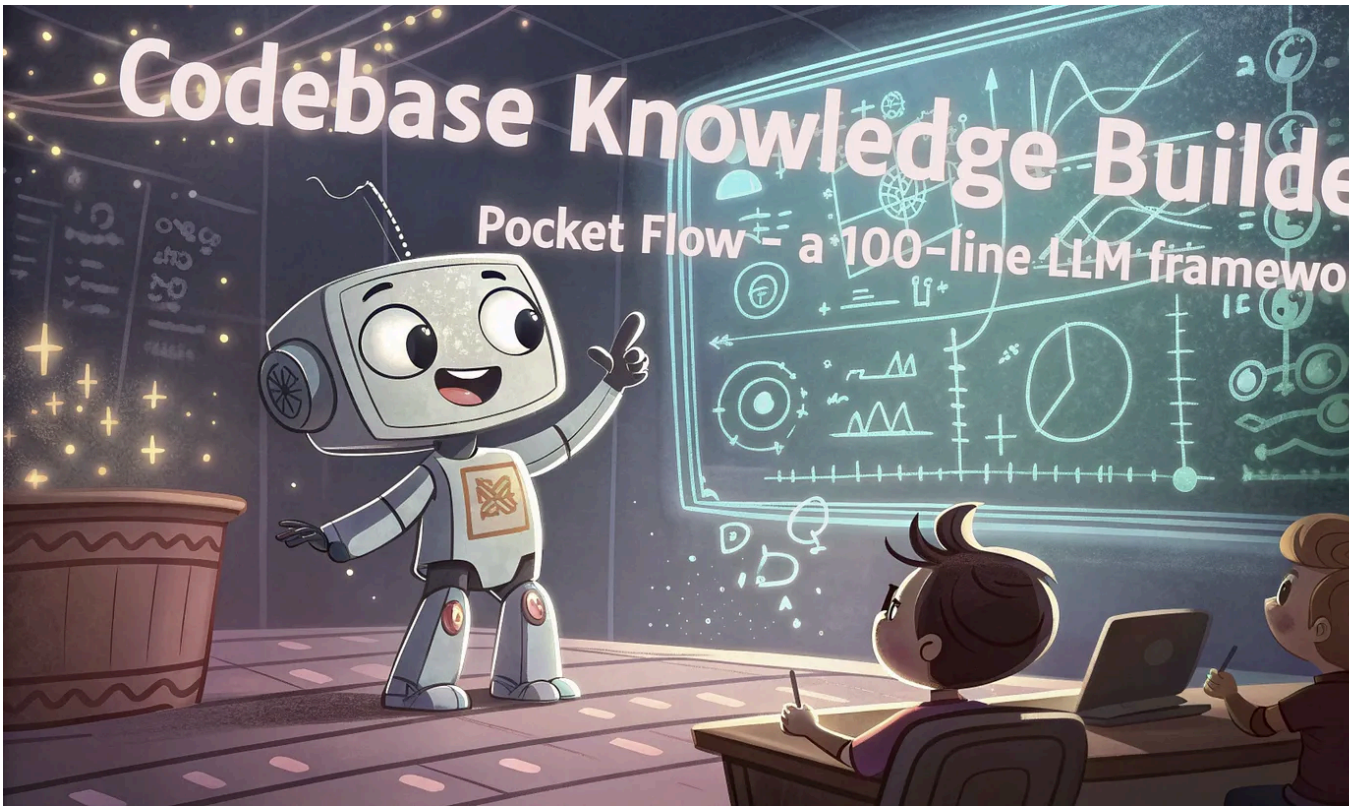
AI Codebase Knowledge Builder (Full Dev Tutorial!)

ZACHARY HUANG
APR 05, 2025



> *Ever stared at a new codebase feeling completely lost? What if an AI could read it for y and create a friendly tutorial explaining exactly how it works? This guide shows you ho build a system that does exactly that! The AI system is [open-sourced on GitHub](#).*

1. Life's too s confusion

Let's be real: You've so
project. You're pump y.
Hundreds of files. Thousands of functions. Code that seems to be talking to o

that's talking to who-knows-what. And you're sitting there thinking, "Where the h
do I even begin?"

Sound familiar? Don't worry, we've all been there!

The Painful Reality of New Codebases

Let's cut to the chase: understanding someone else's code is **hard**. Not just regular
hard - it's tear-your-hair-out frustrating. Studies show developers spend up to 60%
their time just trying to understand code rather than writing it. That's weeks or
months of your life spent muttering things like:

- "What the heck does this class actually do?"
- "How on earth does this component connect to that one?"
- "Why in the world was it designed this way?"
- "Can someone please explain the big picture here?!"

And the fun doesn't stop there! Documentation is often outdated or—let's be hone
completely non-existent. Comments are as rare as unicorns. Variable names migh
well be written in hieroglyphics. It's like being handed a 1000-piece puzzle with ha
the pieces missing, no picture on the box, and someone saying "Good luck, have fu

Why Current AI Solutions Fall Short

Picture this: You're exploring CrewAI, a cool multi-agent framework. Curious abo
how it works, you paste some code into ChatGPT and ask

> How does CrewA

ChatGPT confidently

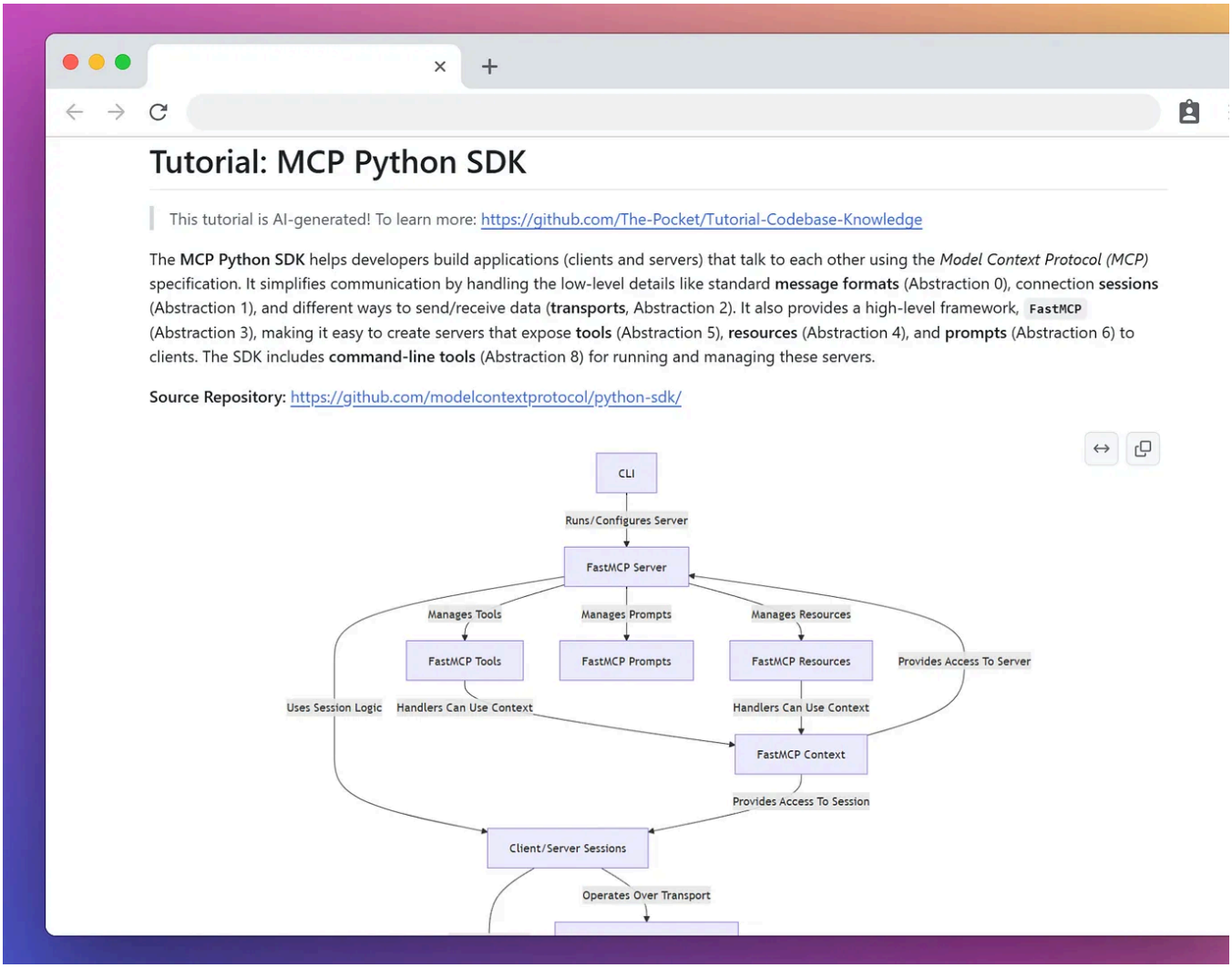
> *"CrewAI's multi-agent system operates through a structured framework that orchestra interactions between specialized AI agents. Each agent has a specific role, custom knowledge base, and defined goals. The system includes a central orchestration layer, communication protocols, and context management..."*

Super helpful, right? ...Nope, not really. Here's why current AI explanations leave scratching your head:

1. **They give you buzzword salad, not actual insights** - You get fancy terms like "orchestration layer" and "context management" without any clue what these actually mean in practice
2. **They barely scratch the surface** - They'll tell you *what* the code does but neve explain *why* it's designed that way or what problem it's actually solving

The end result? Your brain is now swimming with technical jargon, but you still h zero idea how the system actually works. It's like someone handed you all the ingredients for a gourmet cake but forgot the recipe—technically complete but practically useless.

Introducing Codebase Knowledge Builder



What if there was a better way? A system that could:

- **Devour entire codebases** and identify the core ideas and how they play togeth
- **Transform complicated code** into tutorials so clear your grandma could understand them
- **Build your understanding step-by-step** from the basics to the advanced stuff way that actually makes sense

That's exactly what we're building today: a tool that transforms any GitHub repos into a personalized g wo
This project is open-

Check out some exar

1. AutoGen Core - Build AI teams that talk, think, and solve problems together l coworkers!
2. Flask: Craft web apps with minimal code that scales from prototype to produc
3. MCP Python SDK - Build powerful apps that communicate through an elegan protocol without sweating the details!
4. OpenManus - Build AI agents with digital brains that think, learn, and use to just like humans do!

This project is powered by **PocketFlow** - a tiny but mighty agent framework that l us build complex workflows with minimal code. We'll also use Gemini 2.5 Pro, Google's latest AI with serious code-understanding superpowers. Together, they'll create a system that feels almost magical in its ability to make sense of complex co

---

Whether you're a seasoned dev tired of banging your head against unfamiliar code, a te lead who wants to make onboarding less painful, or just someone curious about AI's pote to make programming more accessible - this tutorial is for you. Let's dive in!

---

2. From Code Chaos to Crystal Clarity: Our Secre Sauce

Code isn't just a collection of functions and variables—it's a carefully designed sy of abstractions working together to solve problems. Yet most documentation focu on individual pieces, missing the forest for the trees. Our Codebase Knowledge Builder takes a fundamentally different approach.

From Confusion to Clarity: Our Two-Step Magic Trick

Here's the thing about understanding code: knowing what each function does is li
knowing the names of all the parts in a car engine—utterly useless if you don't kn
how they work together to make the car move!

What you actually need is:

- The big-picture blueprint (what are the key pieces?)
- The master plan (why was it built this way?)
- The relationship map (how do these pieces talk to each other?)

Our approach mirrors how your brain naturally learns—and it's dead simple:

1. **The Eagle's Viev** s c
trying to do? Wh :al
is your secret we

2. **The Deep Dive** - vor
What clever tricl ug
but always keep its place in your mental map crystal clear.

This is exactly how the best teachers work—they don't drown you in details from c one. They give you the big picture first, then fill in the juicy details in a way that actually makes sense and sticks in your brain.

From Huh? to Aha!: Let's See This Magic in Action

Let's take Flask—that super popular Python web framework—and see how our approach transforms it from cryptic code into crystal-clear concepts:

Step 1: The Eagle's View 🦅

Instead of drowning you in details, we start with: "Flask is basically a LEGO set fc building websites. You snap together routes (URLs) with functions that handle requests, and—boom—you've got yourself a web app! It's designed to be lightweig so you're not carrying around features you'll never use."

Step 2: The Deep Dive 🚤

Once you've got that mental map, we zoom into the key pieces:

"The real genius of Flask is how these five key pieces work together:

- The App object (the brain that runs everything)
- Routes (the traffic cops that direct requests to the right place)
- View Functions (the workers that actually do stuff)
- Templates (the pretty face that users see)
- Request/Response objects (the messengers carrying data back and forth)

When someone visits your site, the App checks its routing table, finds the matchir View Function, grabs a Template, and sends

See the difference? I yo mental model that ac ıly you're not lost anymore—you're confidently exploring with a map in hand!

But how do we actually build this magical system? We need a framework that's as and intuitive as the tutorials we want to create. Enter PocketFlow—the perfect pa for our AI-powered adventure.

3. PocketFlow + AI Agents: The Ultimate Code-Building Dream Team

While most AI frameworks hit you with a tsunami of complexity, PocketFlow take opposite approach. It strips away the unnecessary fluff to reveal something beauti elegance through simplicity.

At just 100 lines of code, PocketFlow proves that AI workflows don't need to be complicated. This crystal-clear design makes it perfect for our Codebase Knowled Builder - not just because humans can understand it easily, but because AI agents too! It's like creating building blocks so intuitive that both your 5-year-old and yo robot assistant can play with them.

The Kitchen Analogy: Understanding PocketFlow's Buildi Blocks

Think of PocketFlow like a well-organized kitchen where:

- **Nodes** are cooking stations performing specific tasks:

```
class BaseNode:
    def __init__(self):
        self.params, self.successors = {}, {}

    def add_successor(self, node, action="default"):
        self.successors[action] = node
        return no

    def prep(self
    def exec(self
    def post(self

    def run(self, shared):
```

```
        p = self.prep(shared)
        e = self.exec(p)
        return self.post(shared, p, e)
```

- **Flow** is the recipe that coordinates these stations:

```
class Flow(BaseNode):
    def __init__(self, start):
        super().__init__()
        self.start = start

    def get_next_node(self, curr, action):
        return curr.successors.get(action or "default")

    def orch(self, shared, params=None):
        curr, p = copy.copy(self.start), (params or {**self.params})
        while curr:
            curr.set_params(p)
            c = curr.run(shared)
            curr = copy.copy(self.get_next_node(curr, c))

    def run(self, shared):
        pr = self.prep(shared)
        self.orch(shared)
        return self.post(shared, pr, None)
```

- **Shared Store** is the countertop where all stations can access ingredients and t

```
# Connect nodes together
load_data_node = LoadDataNode()
summarize_node = SummarizeNode()
load_data_node >>

# Create flow
flow = Flow(start

# Pass data throu
```

```
shared = {"file_name": "data.txt"}
flow.run(shared)
```

Each **Node** follows a simple three-step process:

- **Prep**: Gather what's needed from the shared store (like gathering ingredients)
- **Exec**: Perform its specialized task (like cooking the ingredients)
- **Post**: Store results and determine what happens next (like serving the dish and deciding what to make next)

The **Flow** orchestrates the entire process, moving data seamlessly from one node t the next based on specific conditions, much like a chef moving between stations i kitchen.

Agentic Coding: The Fastest Way to Build Anything

Here's the real magic: PocketFlow isn't just simple for humans—it's dead simple f AI agents too! This unleashes a development superpower called **Agentic Coding**:

- **You** sketch the high-level architecture (what humans are great at)
- **AI agents** handle all the detail work and implementation (what AIs excel at)

It's like being the architect who draws a blueprint, then having a team of robots b the entire house overnight while you sleep. No more tedious implementation, debugging weird edge cases, or wrestling with syntax errors!

Traditional coding means designing the system, implementing every detail yourse debugging for hours, and finally shipping something days or weeks later. With Ag Coding? Design the system, let AI agents implement everything, and ship tomorr morning. It's that sin

For our Codebase Tu in next section), and AI 't bogged down in fram st the agents what you want, and they make it happen.

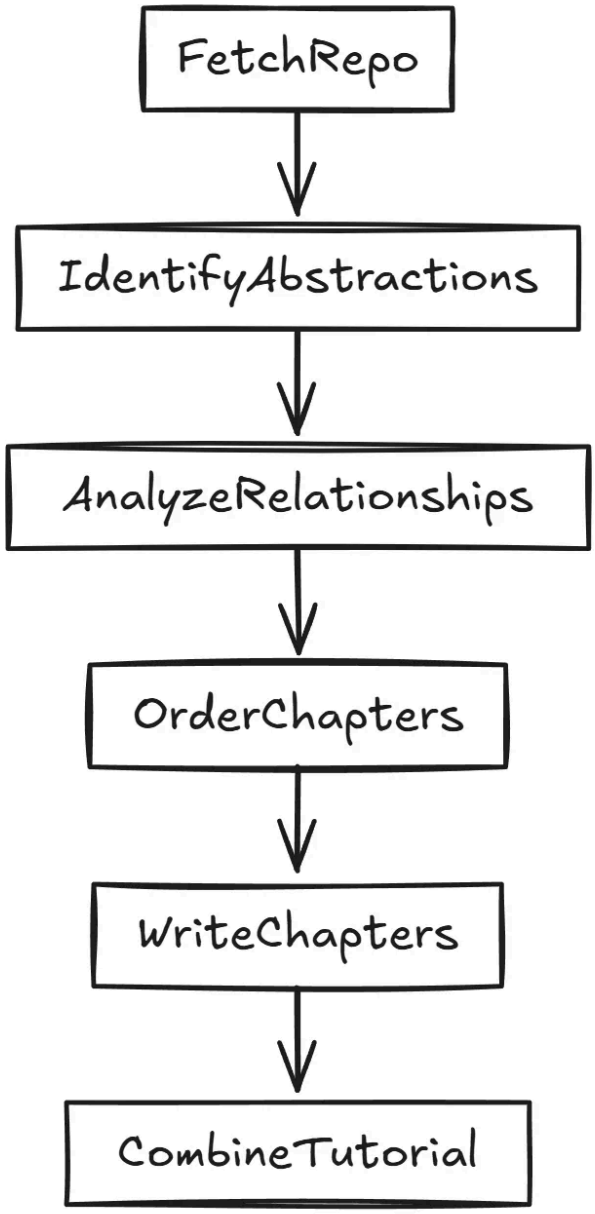
For a detailed guide on setting up this magical development environment, check t
dedicated Agentic Coding guide and the PocketFlow Documentation.

4. Behind the Scenes: How Our Code Tutor Actually Works

So how does this codebase tutorial builder actually work? It's actually pretty cleve
we've built a team of specialized components that work together, each handling a
specific part of the tutorial-creation process. Let's take a look at who does what.

The Step-by-Step Process

Our system works like a well-organized assembly line, with each component passi
its output to the next:



This mirrors how you'd naturally approach learning something complex - gather materials, identify the key concepts, understand how they relate, figure out what t learn first, dive into each topic, and finally put everything together. Our system ju automates the whole thing.

Meet the Team (No Capes Required)

Let's meet each com

1. FetchRepo: The

This component fetc for need.

- **WHAT IT SEES:** A GitHub repository URL and optional filters
- **WHAT IT DOES:** Downloads the code files while skipping tests, build artifac and other non-essentials
- **WHAT IT DELIVERS:** A clean collection of files ready for analysis

Sample Output:

```
Found and processed 15 Python files from https://github.com/the-
pocket/PocketFlow:
- pocketflow/__init__.py (core framework code)
- pocketflow/utils.py (utility functions)
- pocketflow/nodes/__init__.py (node definitions)
...
```

2. IdentifyAbstractions: The Pattern Finder

This component spots the important concepts hiding in the code.

- **WHAT IT SEES:** All the code files collected by FetchRepo
- **WHAT IT DOES:** Analyzes class definitions, patterns, and code structure to identify core abstractions
- **WHAT IT DELIVERS:** A list of key concepts with clear descriptions

Sample Output:

```
Identified 6 core abstractions:
1. BaseNode: The fundamental building block for creating workflow
components
   Files: pocketflow/__init__.py, pocketflow/nodes/__init__.py
2. Flow: Orchestr
   Files: pocket
...
```


3. AnalyzeRelatio

This component figures out how all the key concepts connect and interact with ea other.

- **WHAT IT SEES:** The abstractions from the previous step and their code
- **WHAT IT DOES:** Analyzes function calls, inheritance, data flow, and dependencies
- **WHAT IT DELIVERS:** A map of how everything fits together

Sample Output:

```
Project Summary: "PocketFlow is a minimal framework for building AI
workflows using a graph-based approach..."

Relationships:
- BaseNode → Flow (contained in): Flow contains and manages BaseNode
instances
- Flow → BaseNode (orchestrates): Flow controls when and how Nodes
execute
- SharedMemory → Node (provides data): Nodes access data through the
SharedMemory
...
```

4. OrderChapters: The Learning Planner

This component figures out the most logical order to teach each concept.

- **WHAT IT SEES:** The abstractions and their relationships
- **WHAT IT DOES:** Analyzes dependencies to determine what needs to be learı first
- **WHAT IT DELIVERS:** A sensible learning sequence that builds knowledge s by step

Sample Output:

```
Recommended learning sequence:
1. BaseNode (foundational concept)
```

```
2. SharedMemory (required to understand data flow)
3. Flow (builds on understanding of Nodes)
4. BatchNode (specialized type of Node)
...
```

5. WriteChapters: The Clear Explainer

This component creates easy-to-understand explanations with helpful analogies.

- **WHAT IT SEES:** Each abstraction, its description, and relevant code
- **WHAT IT DOES:** Creates beginner-friendly chapters with examples and explanations
- **WHAT IT DELIVERS:** Complete, readable content for each chapter

Sample Output:

```
Chapter 1: The BaseNode

Imagine a worker at a station in a factory. Their job is simple: take
inputs,
do something specific with them, and pass the results to the next
station.
This is exactly what a BaseNode does in PocketFlow!

A BaseNode has three key responsibilities:
1. **Prep**: Gather what's needed (like collecting ingredients)
2. **Exec**: Do the actual work (like cooking the ingredients)
3. **Post**: Decide what happens next (like serving the dish)

Let's look at how this appears in the code:
...
```

6. CombineTutori

This component puts ely

- **WHAT IT SEES** ent
- **WHAT IT DOES:** Creates visuals, sets up navigation, and organizes the

- **WHAT IT DELIVERS:** A complete, ready-to-use tutorial

Sample Output:

```
Tutorial complete! Available at: /output/pocketflow_tutorial/
- index.md (Project overview with visualization)
- 01_base_node.md
- 02_shared_memory.md
- 03_flow.md
...
```

In the next section, we'll look at how these components are actually implemented and you'll be surprised at how simple the code really is!

5. The Nuts and Bolts: Simple Code That Makes t Magic Happen

Ready to see what's actually under the hood? You might expect complicated code t power such a clever system, but the beauty of our approach is its simplicity. Let's l at the actual implementation—you'll be surprised at how readable it is!

> **Note:** We've simplified things a bit to focus on what matters. Think of this as tl director's cut that skips the boring parts. You can find the full code *on GitHub*.

The Shared Memory Structure

```
shared = {
    "repo_url": "https://github.com/the-pocket/PocketFlow",  # User
input
    "codebase":
    "core_abstra
    "abstraction_
other
    "chapter_orde
    "chapters": 
}
```

Think of this as the team's shared whiteboard. Everyone can read what's already t and add their own findings for others to use later—no need to pass papers around repeat work.

Node Implementations

1. FetchRepo: Gathering the Code

```
class FetchRepo(Node):
    def prep(self, shared):
        # Get repository URL from shared memory
        repo_url = shared["repo_url"]
        return {"repo_url": repo_url}

    def exec(self, prep_res):
        # Download codebase from GitHub
        # Skip unimportant files like tests, docs, etc.
        # Returns the full codebase
        return download_codebase_from_github(prep_res["repo_url"])

    def post(self, shared, prep_res, exec_res):
        # Store codebase in shared memory for next nodes
        shared["codebase"] = exec_res
```

Just 13 lines of code to handle all the GitHub downloading! This node basically sa "Give me a repo URL, let me grab all the relevant files (skipping the junk), and I'll the good stuff where everyone else can find it."

2. IdentifyAbstractions: Finding the Core Concepts

```
class IdentifyAbstractions(Node):
    def prep(sel
        # Format
        return sh

    def exec(sel
        codebase
        # Ask AI with codebase directly in the prompt
```

```
        prompt = f"Given this codebase: {codebase}, identify 5-10 core
concepts..."
        return call_llm(prompt)

    def post(self, shared, prep_res, exec_res):
        # Store identified core abstractions
        shared["core_abstractions"] = exec_res
```

Here's where the AI brain kicks in. We're essentially asking: "Hey Gemini, take a l at this code and tell me what the main concepts are." Then we save what it finds f the next steps. Clean and simple!

3. AnalyzeRelationships: Mapping the Connections

```
class AnalyzeRelationships(Node):
    def prep(self, shared):
        # Pass core abstractions and codebase
        return {"core_abstractions": shared["core_abstractions"],
                "codebase": shared["codebase"]}

    def exec(self, prep_res):
        # Ask AI with data directly in the prompt
        prompt = f"Given these core concepts:
{prep_res['core_abstractions']} and this codebase:
{prep_res['codebase']}, analyze how they connect..."
        return call_llm(prompt)

    def post(self, shared, prep_res, exec_res):
        # Store abstraction relationships
        shared["abstraction_relationships"] = exec_res
```

Another straightforward AI prompt, but this time we're asking: "Now that you kn the main pieces, how understanding functi

4. OrderChapters

```
class OrderChapters(Node):
    def prep(self, shared):
        # Pass the data needed for analysis
        return {"core_abstractions": shared["core_abstractions"],
                "relationships": shared["abstraction_relationships"]}

    def exec(self, prep_res):
        # Ask AI with data directly in the prompt
        prompt = f"Given these concepts: {prep_res['core_abstractions
and relationships: {prep_res['relationships']}, what's the best teach
order?"
        return call_llm(prompt)

    def post(self, shared, prep_res, exec_res):
        # Store the recommended chapter order
        shared["chapter_order"] = exec_res
```

This time we ask: "What's the best order to learn these things?" The AI considers dependencies (you can't understand X until you know Y) and creates a logical lear sequence.

5. WriteChapters: Creating the Content

```
class WriteChapters(BatchNode):
    def prep(self, shared):
        # Create batches for each concept/chapter
        chapters = []
        for concept_idx in shared["chapter_order"]:
            concept = shared["core_abstractions"][concept_idx]
            relevant_code = extract_relevant_code(concept,
shared["codebase"])
            chapters.append({
                "concept": concept
                '
                                                                    ',
[])
            })
        return ch

    def exec(self, item):
```

```
        # Create richer, more structured prompt
        prompt = f"""Write a beginner-friendly chapter about
{item['concept']['name']} with code: {item['code']} and previous
chapters: {self.completed_chapters if hasattr(self,
'completed_chapters') else []}

        Guide:
        - Start with problem/motivation and a concrete use case
        - Break complex ideas into simpler concepts
        - Show usage examples with inputs/outputs
        - Keep code blocks under 20 lines
        - Explain implementation with simple step-by-step walkthrough
        - Include minimal diagrams if helpful"""

        chapter = call_llm(prompt)

        # Track chapters for continuity
        if not hasattr(self, "completed_chapters"):
            self.completed_chapters = []
        self.completed_chapters.append(chapter)
        return chapter

    def post(self, shared, prep_res, exec_res_list):
        # Store all chapters and tracking info
        shared["chapters"] = exec_res_list
```

This is our longest bit of code, but it's still pretty readable. For each concept, we gather the relevant code snippets and ask the AI to write a tutorial chapter about i We also track the chapters we've already written so each new chapter can build on what came before.

I have also done some prompt engineering based on my personal experience of wr system design docs. The prompt guides the AI to begin with high-level motivation explaining what prob

concepts, and them e

6. CombineTutori

```
class CombineTutorial(Node):
    def prep(self, shared):
        # Gather all components for the final tutorial
        return prepare_tutorial_components(shared)

    def exec(self, prep_res):
        # Create output directory
        # Generate visualization diagram
        # Write index.md with overview
        # Write each chapter file
        return assemble_final_tutorial(prep_res)

    def post(self, shared, prep_res, exec_res):
        # Store path to completed tutorial
        shared["final_output_dir"] = exec_res
        print(f"Tutorial complete! Files are in: {exec_res}")
```

The finishing touches! This takes all our chapters, creates pretty visualizations, an packages everything into a complete tutorial with an index page and navigation.

Connecting Everything Together

```
def create_tutorial_flow():
    # Create all nodes
    fetch_repo = FetchRepo()
    identify_abstractions = IdentifyAbstractions()
    analyze_relationships = AnalyzeRelationships()
    order_chapters = OrderChapters()
    write_chapters = WriteChapters()
    combine_tutorial = CombineTutorial()

    # Connect nodes in sequence
    fetch_repo >>
    analyze_rela
    write_chapte

    return Flow(s

# Use like this:
flow = create_tutorial_flow()
```

```
shared = {"repo_url": "https://github.com/example/repo"}
flow.run(shared)
```

Here's where the magic happens! Just look at those >> operators connecting everything together. It's like reading a story: fetch the repo, identify key concepts, analyze their relationships, figure out teaching order, write chapters, combine everything. Then we create the flow and run it with a GitHub URL. That's it!

And there you have it—you can now turn any GitHub repo into a beginner-friendl tutorial. The real power comes from clever prompting and letting the AI do what i does best.

6. What You've Learned & The Road Ahead

Throughout this tutorial, you've learned three powerful skills that will forever cha how you approach new codebases:

1. **A Systematic Way to Read Code** - The Eagle's View and Deep Dive approach gives you a methodical process that's far better than just asking ChatGPT abo code snippets. You start with the big picture architecture, then strategically z into the important implementation details.
2. **System Design for Knowledge Building** - How to build a workflow of speciali components that follow the Eagle's View and Deep Dive approach, working together to transform raw code into clear tutorials with the right learning sequence.
3. **Agentic Coding with PocketFlow** - How to use PocketFlow's simple but powe framework to create a clear division of labor where you design the high-level while AI agents implement the details, dramatically speeding up development

Looking ahead, there

- **Context Size** - D
  Massive codebas ow

and would need a more aggressive system design with chunking and summarization techniques.

- **Frontend Visualization** - I'll be honest - I'm mostly a backend person myself, I'm not entirely sure how to best represent front end codebase knowledge. Ma through React component trees, state management, and event-driven architectures? Not sure - interesting to explore in the future!

But don't let these minor points hold you back! The Codebase Knowledge Builder already delivers incredible value, turning those intimidating repositories into clea approachable tutorials that make sense on both a conceptual and practical level.

The days of staring helplessly at unfamiliar code are over. You now have a reliable guide that reveals the true design behind any codebase, letting you understand in hours what used to take weeks.

Ready to explore code with confidence? The AI Codebase Knowledge Builder is open-sou and waiting for you! Experience how PocketFlow's elegant 100 lines of code can transforn your development workflow today. GitHub | Documentation

Write a comment...

James CHu Apr 6

Liked by Zachary Huang

THIS IS SERIOUSLY GOLD NUGGET! No flattery, clean and a Masterpiece of work! Amazing!

LIKE (5) REPLY

2 replies by Zachary Huang and others

Joe Berns Apr 30

Liked by Zachary Huang

I tried to run it on a larger codebase, and keep getting token limit exceeded issue: The input count (1069372) exceeds the maximum number of tokens allowed . Is there any way around

LIKE (1) REPLY

1 reply by Zachary Huang

15 more comments...