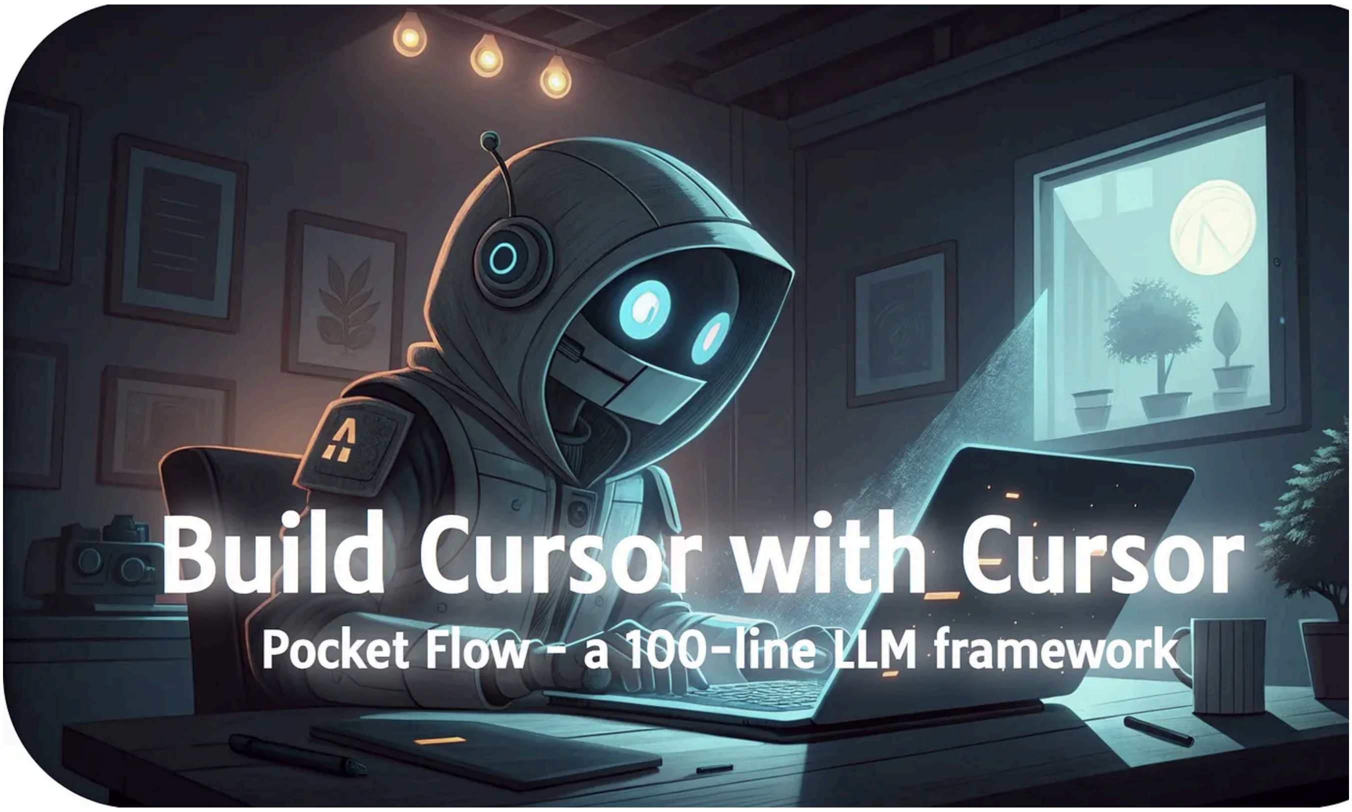
Building Cursor with Cursor: A Step-by-Step Guide to Creating Your Own AI Coding Agent

ZACHARY HUANG
MAR 15, 2025

6

Sl



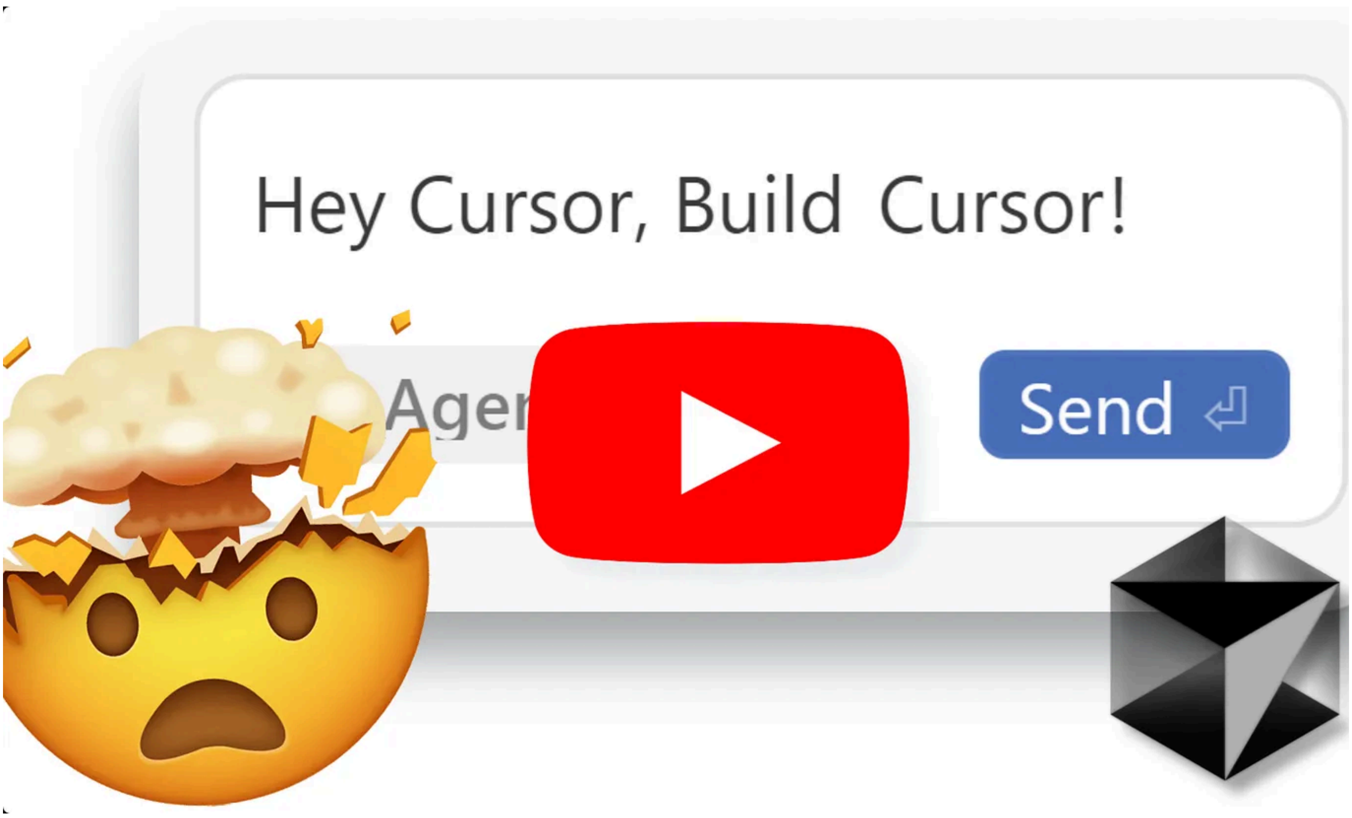
Have you ever wished you could customize your AI coding assistant to work exact the way you want? What if you could build your own version of Cursor—an AI-powered code editor—using Cursor itself? That's exactly what we're doing in this tutorial: creating a customizable, open-source AI coding agent that operates right within Cursor.

In this step-by-step g … ld a powerful AI assistan

- Navigate and understand codebases
- Implement code changes based on natural language instructions
- Make intelligent decisions about which files to inspect or modify
- Learn from its own history of operations

The result is available at: https://github.com/The-Pocket/Tutorial-Cursor

Also, check out the YouTube Video:



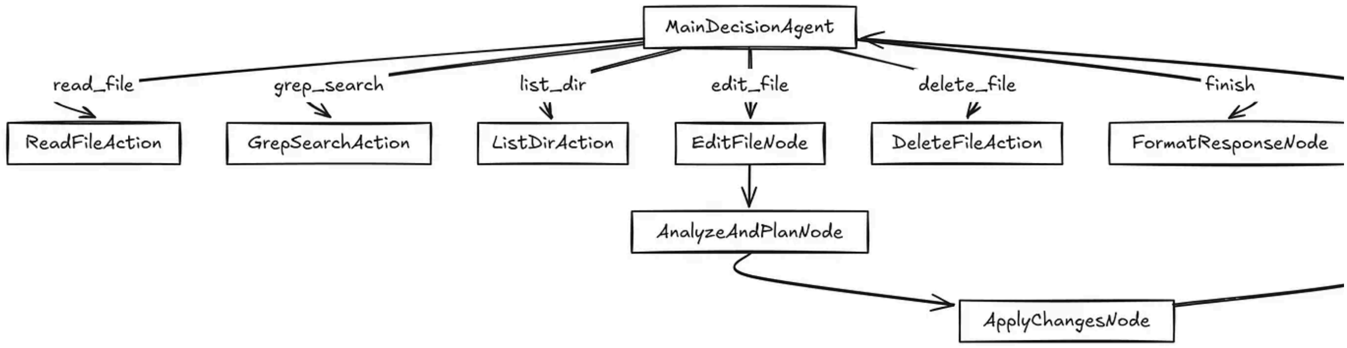
Let's dive in!

1. Understa

Before we write a sin ... rs

Agent. The system is built on a flow-based architecture using Pocket Flow, a minimalist 100-line LLM framework that enables agentic development.

Here's a high-level overview of our architecture:



This architecture separates concerns into distinct nodes:

- Decision making (what operation to perform next)
- File operations (reading, writing, and searching)
- Code analysis (understanding and planning changes)
- Code modification (safely applying changes)

2. Setting Up Your Environment

Let's get our environment ready:

```
# Clone the repository
git clone https://github.com/The-Pocket/Tutorial-Cursor
cd Tutorial-Cursor

# Install dependencies
pip install -r requirements.txt
```

3. The Core:

Our agent is built on
abstractions:

1. **Nodes**: Individual units of computation that perform specific tasks
2. **Flows**: Directed graphs of nodes that define the program's execution path
3. **Shared Store**: A dictionary that all nodes can access to share data

Let's look at the core imports and setup:

```
# flow.py
from pocketflow import Node, Flow, BatchNode
import os
import yaml
import logging
from datetime import datetime
from typing import List, Dict, Any, Tuple

# Import utility functions
from utils.call_llm import call_llm
from utils.read_file import read_file
from utils.delete_file import delete_file
from utils.replace_file import replace_file
from utils.search_ops import grep_search
from utils.dir_ops import list_dir
```

This imports the core classes from Pocket Flow and our custom utility functions t handle file operations and LLM calls.

4. Implementing Decision Making

At the heart of our agent is the `MainDecisionAgent`, which determines what ac to take based on the user's request and the current state of the system.

Here's how it's imple


```
class MainDecisi
    def prep(sel                                                    st
Any]]]:
        # Get user query and history
```

```
        user_query = shared.get("user_query", "")
        history = shared.get("history", [])

        return user_query, history

    def exec(self, inputs: Tuple[str, List[Dict[str, Any]]]) ->
Dict[str, Any]:
        user_query, history = inputs

        # Format history for context
        history_str = format_history_summary(history)

        # Create prompt for the LLM
        prompt = f"""You are a coding assistant that helps modify and
navigate code. Given the following request,
decide which tool to use from the available options.

User request: {user_query}

Here are the actions you performed:
{history_str}

Available tools:
1. read_file: Read content from a file
   - Parameters: target_file (path)

2. edit_file: Make changes to a file
   - Parameters: target_file (path), instructions, code_edit

[... more tool descriptions ...]

Respond with a YAML object containing:
```yaml
tool: one of: read_file, edit_file, delete_file, grep_search, list_di
finish
reason: |
  detailed explar                                              nd
do
params:
  # parameters s
```"""

        # Call LL
        response = call_llm(prompt)
```

```
        # Parse YAML response
        yaml_content = extract_yaml_from_response(response)
        decision = yaml.safe_load(yaml_content)

        # Validate the required fields
        assert "tool" in decision, "Tool name is missing"
        assert "reason" in decision, "Reason is missing"

        return decision

    def post(self, shared: Dict[str, Any], prep_res: Any, exec_res:
Dict[str, Any]) -> str:
        # Add the decision to history
        shared.setdefault("history", []).append({
            "tool": exec_res["tool"],
            "reason": exec_res["reason"],
            "params": exec_res.get("params", {}),
            "timestamp": datetime.now().isoformat()
        })

        # Return the name of the tool to determine which node to execu
next
        return exec_res["tool"]
```

This node:

1. Gathers the user's query and the history of previous actions
2. Formats a prompt for the LLM with all available tools
3. Calls the LLM to decide what action to take
4. Parses the response and validates it
5. Adds the decision to the history
6. Returns the nam xec

5. File Opera e

Let's look at how our

```
class ReadFileAction(Node):
    def prep(self, shared: Dict[str, Any]) -> str:
        # Get parameters from the last history entry
        history = shared.get("history", [])
        last_action = history[-1]
        file_path = last_action["params"].get("target_file")

        # Ensure path is relative to working directory
        working_dir = shared.get("working_dir", "")
        full_path = os.path.join(working_dir, file_path) if working_d
else file_path

        return full_path

    def exec(self, file_path: str) -> Tuple[str, bool]:
        # Call read_file utility which returns a tuple of (content,
success)
        return read_file(file_path)

    def post(self, shared: Dict[str, Any], prep_res: str, exec_res:
Tuple[str, bool]) -> str:
        # Unpack the tuple returned by read_file()
        content, success = exec_res

        # Update the result in the last history entry
        history = shared.get("history", [])
        if history:
            history[-1]["result"] = {
                "success": success,
                "content": content
            }

        return "decision"  # Go back to the decision node
```

The `read_file` utili


```
def read_file(ta
    """
    Read content
    Prepends 1-based line numbers to each line in the output.
```

```
    Returns:
        Tuple of (file content with line numbers, success status)
    """
    try:
        if not os.path.exists(target_file):
            return f"Error: File {target_file} does not exist", False

        with open(target_file, 'r', encoding='utf-8') as f:
            lines = f.readlines()
            # Add line numbers to each line
            numbered_lines = [f"{i+1}: {line}" for i, line in
enumerate(lines)]
            return ''.join(numbered_lines), True

    except Exception as e:
        return f"Error reading file: {str(e)}", False
```

This provides a clean, line-numbered view of the file content that makes it easier f the LLM to reference specific lines in its analysis.

6. Code Analysis and Planning

When the agent needs to modify code, it first analyzes the code and plans the char using `AnalyzeAndPlanNode`:

```
class AnalyzeAndPlanNode(Node):
    def prep(self, shared: Dict[str, Any]) -> Dict[str, Any]:
        # Get history
        history = shared.get("history", [])
        last_action = history[-1]

        # Get file content and edit instructions
        file_cont
        instruct
        code_edit

        return {
            "file
            "instructions": instructions,
            "code_edit": code_edit
```

```
        }

    def exec(self, params: Dict[str, Any]) -> List[Dict[str, Any]]:
        file_content = params["file_content"]
        instructions = params["instructions"]
        code_edit = params["code_edit"]

        # Generate a prompt for the LLM to analyze the edit
        prompt = f"""
As a code editing assistant, I need to convert the following code edit
instruction
and code edit pattern into specific edit operations (start_line,
end_line, replacement).

FILE CONTENT:
{file_content}

EDIT INSTRUCTIONS:
{instructions}

CODE EDIT PATTERN (markers like "// ... existing code ..." indicate
unchanged code):
{code_edit}

Analyze the file content and the edit pattern to determine exactly whe
changes should be made.
Return a YAML object with your reasoning and an array of edit
operations:

```yaml
reasoning: |
  Explain your thinking process about how you're interpreting the edit
pattern.

operations:
  - start_line: 10
    end_line: 15
    replacement: 
      # New code
```"""

        # Call Ll
        response = call_llm(prompt)
```

```
        # Parse the response and extract edit operations
        yaml_content = extract_yaml_from_response(response)
        result = yaml.safe_load(yaml_content)

        # Store reasoning in shared memory
        shared["edit_reasoning"] = result.get("reasoning", "")

        # Return the operations
        return result.get("operations", [])
```

This node:

1. Extracts the file content, instructions, and code edit pattern from the history
2. Creates a prompt for the LLM to analyze the edit
3. Calls the LLM to determine the exact line numbers and replacement text
4. Parses the response to extract the edit operations
5. Stores the reasoning in shared memory
6. Returns the operations as a list of dictionaries

7. Applying Code Changes

Once the agent has planned the changes, it applies them using `ApplyChangesNc`

```
class ApplyChangesNode(BatchNode):
    def prep(self, shared: Dict[str, Any]) -> List[Dict[str, Any]]:
        # Get edit operations
        edit_operations = shared.get("edit_operations", [])

        # Sort edit operations in descending order by start_line
        # This er                                                          om
bottom to top
        sorted_o
op["start_line"],

        # Get ta
        history = shared.get("history", [])
        last_action = history[-1]
```

```
        target_file = last_action["params"].get("target_file")

        # Ensure path is relative to working directory
        working_dir = shared.get("working_dir", "")
        full_path = os.path.join(working_dir, target_file) if
working_dir else target_file

        # Attach file path to each operation
        for op in sorted_ops:
            op["target_file"] = full_path

        return sorted_ops

    def exec(self, op: Dict[str, Any]) -> Tuple[bool, str]:
        # Call replace_file utility to replace content
        return replace_file(
            target_file=op["target_file"],
            start_line=op["start_line"],
            end_line=op["end_line"],
            content=op["replacement"]
        )

    def post(self, shared: Dict[str, Any], prep_res: List[Dict[str,
Any]], exec_res_list: List[Tuple[bool, str]]) -> str:
        # Check if all operations were successful
        all_successful = all(success for success, _ in exec_res_list)

        # Update edit result in history
        history = shared.get("history", [])
        if history:
            history[-1]["result"] = {
                "success": all_successful,
                "operations": len(exec_res_list),
                "details": [{"success": s, "message": m} for s, m in
exec_res_list],
                "reasoning": shared.get("edit_reasoning", "")
            }

        return "(
```


This node is a `Batch` ... sin run. It:

1. Gets the edit operations from shared memory
2. Sorts them in descending order by start line to ensure edits remain valid
3. Attaches the target file path to each operation
4. Executes each operation using the `replace_file` utility
5. Updates the history with the results
6. Returns to the decision node

The `replace_file` utility works by combining `remove_file` and `insert_fi`

```
def replace_file(target_file: str, start_line: int, end_line: int,
content: str) -> Tuple[str, bool]:
    try:
        # First, remove the specified lines
        remove_result, remove_success = remove_file(target_file,
start_line, end_line)

        if not remove_success:
            return f"Error during remove step: {remove_result}", Fals

        # Then, insert the new content at the start line
        insert_result, insert_success = insert_file(target_file,
content, start_line)

        if not insert_success:
            return f"Error during insert step: {insert_result}", Fals

        return f"Successfully replaced lines {start_line} to
{end_line}", True

    except Exception as e:
        return f"Error replacing content: {str(e)}", False
```


8. Running

Now that we've impl ou `main.py`:

```
import os
import argparse
import logging
from flow import coding_agent_flow

def main():
    # Parse command-line arguments
    parser = argparse.ArgumentParser(description='Coding Agent - AI-
powered coding assistant')
    parser.add_argument('--query', '-q', type=str, help='User query to
process', required=False)
    parser.add_argument('--working-dir', '-d', type=str,
default=os.path.join(os.getcwd(), "project"),
                        help='Working directory for file operations')
    args = parser.parse_args()

    # If no query provided via command line, ask for it
    user_query = args.query
    if not user_query:
        user_query = input("What would you like me to help you with? '

    # Initialize shared memory
    shared = {
        "user_query": user_query,
        "working_dir": args.working_dir,
        "history": [],
        "response": None
    }

    # Run the flow
    coding_agent_flow.run(shared)

if __name__ == "__main__":
    main()
```

And finally, let's crea


```
# Define the node
main_decision = M
read_file_action = ReadFileAction()
grep_search_action = GrepSearchAction()
```

```
list_dir_action = ListDirAction()
delete_file_action = DeleteFileAction()
edit_file_node = EditFileNode()
analyze_plan_node = AnalyzeAndPlanNode()
apply_changes_node = ApplyChangesNode()
format_response_node = FormatResponseNode()

# Connect the nodes
main_decision - "read_file" >> read_file_action
main_decision - "grep_search" >> grep_search_action
main_decision - "list_dir" >> list_dir_action
main_decision - "delete_file" >> delete_file_action
main_decision - "edit_file" >> edit_file_node
main_decision - "finish" >> format_response_node

# Connect action nodes back to main decision
read_file_action - "decision" >> main_decision
grep_search_action - "decision" >> main_decision
list_dir_action - "decision" >> main_decision
delete_file_action - "decision" >> main_decision

# Connect edit flow
edit_file_node - "analyze" >> analyze_plan_node
analyze_plan_node - "apply" >> apply_changes_node
apply_changes_node - "decision" >> main_decision

# Create the flow
coding_agent_flow = Flow(start=main_decision)
```

Now you can run your agent with:

```
python main.py --query "List all Python files" --working-dir ./projec
```

9. Advance

One of the most pow ... iz
Let's explore a few w

1. Adding New Tools

To add a new tool, simply:

1. Create a new action node class
2. Add it to the `MainDecisionAgent`'s prompt
3. Connect it to the flow

For example, to add a "run_tests" tool:

```
class RunTestsAction(Node):
    def prep(self, shared):
        # Get test directory from parameters
        history = shared.get("history", [])
        last_action = history[-1]
        test_dir = last_action["params"].get("test_dir")
        return test_dir

    def exec(self, test_dir):
        # Run tests and capture output
        import subprocess
        result = subprocess.run(
            ["pytest", test_dir],
            capture_output=True,
            text=True
        )
        return result.stdout, result.returncode == 0

    def post(self, shared, prep_res, exec_res):
        # Update history with test results
        output, success = exec_res
        history = shared.get("history", [])
        if history:
            history[-1]["result"] = {
                "success": success,
                "
            }
        return "(

# Then add to you
run_tests_action
```

```
main_decision - "run_tests" >> run_tests_action
run_tests_action - "decision" >> main_decision
```

2. Improving Code Analysis

You can enhance the code analysis capabilities by modifying the prompts in `AnalyzeAndPlanNode`:

```
# Add language-specific hints
language_hints = {
    ".py": "This is Python code. Look for function and class
definitions.",
    ".js": "This is JavaScript code. Look for function declarations ar
exports.",
    # Add more languages as needed
}

# Update the prompt with language-specific hints
file_ext = os.path.splitext(target_file)[1]
language_hint = language_hints.get(file_ext, "")
prompt += f"\n\nLANGUAGE HINT: {language_hint}"
```

3. Adding Memory and Context

To give your agent more context, you could add a vector database to store and retr relevant information:

```
class VectorDBNode(Node):
    def prep(self, shared):
        # Get text to store
        history = shared.get("history", [])
        context =
        for acti
            if a                                                    ",
{}).get("success'
                (
                c
['target_file']}\n{content}\n\n"
```

```
        return context

    def exec(self, context):
        # Store in vector DB
        embeddings = OpenAIEmbeddings()
        vectordb = Chroma.from_texts(
            texts=[context],
            embedding=embeddings,
            persist_directory="./db"
        )
        return vectordb

    def post(self, shared, prep_res, exec_res):
        shared["vectordb"] = exec_res
        return "decision"
```

10. Conclusion and Next Steps

Congratulations! You've built a customizable AI coding agent that can help you navigate and modify code based on natural language instructions. This agent demonstrates the power of agentic development, where AI systems help build bett AI systems.

The possibilities for extending this agent are endless:

- Add support for more programming languages
- Implement code refactoring capabilities
- Create specialized tools for specific frameworks
- Add security checks before making changes
- Implement static analysis to catch potential bugs

As LLM capabilities re
powerful tools in a d

Want to learn more?)
tutorial on building a

Happy coding!

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

6 Likes

← Previous Next

Discussion about this post

Comments Restacks

Write a comment...