CURSOR

The AI Code Editor

Vibecoding with Cursor

Cursor, the best way to code with AI?

GenAI & Machine Learning Bootcamp Masterclass

Rémi Veltin – 2025/09

What is Cursor?

An AI-powered code editor

- ✓ Based on *VS Code* with deep AI integration
- ✓ Founded by former OpenAI and Anthropic engineers
- ✓ Uses GPT / Claude / Grok models to understand and generate code
- ✓ Free version available with premium options

The concept of "Vibecoding"

"Vibecoding" is a state of fluid programming where:

- ✓ AI and the developer work in synergy
- ✓ Ideas are quickly transformed into functional code
- ✓ Cognitive load is reduced, allowing focus on logic
- ✓ The coding experience becomes more natural and intuitive

https://cursor.com/downloads

Why is Cursor useful?

Accelerated development
Significant reduction of time between design and implementation, enabling faster iterations

Smart code navigation
Helps to understand and navigate complex codebases through contextual analysis

Cognitive load reduction
Focus on business logic rather than syntax or low-level implementation details

Natural communication
Interact with code in natural language, removing barriers between intention and implementation

Multi-language adaptability
Support for many programming languages and frameworks with contextual understanding

Ideal for GenAI & ML
Specifically optimized for AI and ML projects, with understanding of specialized frameworks and libraries

40%* reduction in context switching

30%* faster onboarding

25%* time saved on debugging / refactoring

** https://www.opsera.io/blog/cursor-ai-adoption-trends-real-data-from-the-fastest-growing-coding-tool?*

Where does Cursor helps?

- **Frontend development**: building UI components, optimizing performance, accessibility improvements.
- **Backend development**: API design, data modeling, business logic implementation.
- **Full-stack development**: end-to-end feature delivery, integrating frontend and backend.
- **DevOps and tooling**: CI/CD pipelines, infrastructure as code, deployment automation.

- **Testing and quality**: writing tests (unit, integration, end-to-end, test automation, test-driven development (TDD) practices
- **Code review and governance**: enforcing standards, spotting anti-patterns, refactoring guidance.
- **Legacy modernization**: understanding old code, creating safe refactors, introducing tests.
- **Documentation and education**: generating docs, creating examples, explaining concepts.

What Cursor brings compared to ChatGPT ?

- **Cursor** is **positioned as a versatile AI assistant tailored for software engineering**, with a focus on practical integration into development workflows. **ChatGPT is a general-purpose assistant that provides broad coding** help and explanations but isn’t natively tied to your workflow or guaranteed to produce buildable artifacts.

What advantages and limitations?

✓ Advantages

- Native AI integration (deep integration, unlike add-on plugins)
- Advanced conversational agent (Agent Mode handles end-to-end tasks, maintains context)
- Contextual understanding (analyzes the whole codebase for relevant suggestions)
- Powerful AI models (last versions regularly integrated)
- Familiar VS Code experience (based on VSCode, smooth transition for its users)

⚠ Limitations

- Reliance on cloud services (advanced features require internet and are limited offline)
- Potential bug generation (AI-generated code may need corrections, can include subtle errors)
- Privacy concerns (data is sent to AI servers, problematic for proprietary code)
- Possible performance issues (can slow down the editor on weaker machines or very large projects)
- Hallucinations on complex APIs (may generate incorrect API calls or invent non-existing features)

Best practices

>>> Use Cursor as an assistant, not a replacement.

>>> Always verify the code and maintain your technical expertise.

>>> Very powerful for on-boarding, package understanding and documentation

Use custom rules to guide the AI toward your coding style.

How to use Cursor?

>>> QuickStart in 5 Minutes

1. **Open** a **project**

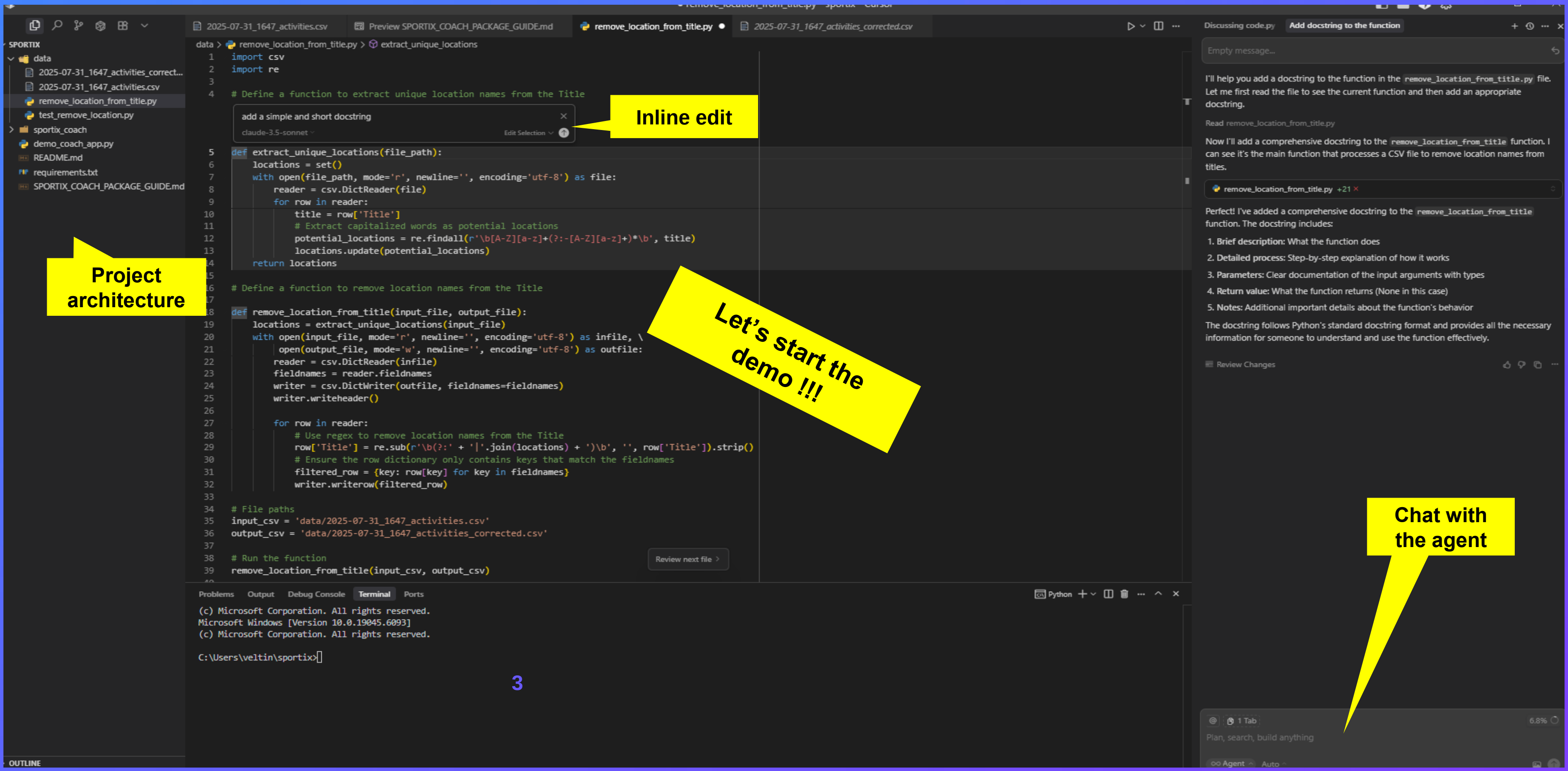
2. **Autocomplete** with **Tab**

3. **Inline edit** (ctrl + K)

4. **Chat** with **Agent** (ctrl + I)

5. **Bonus Features** :
> Background Agent, Custom Rules, MCP integration

Cursor Interface



2025-07-31_1647_activities.csv
Preview SPORTIX_COACH_PACKAGE_GUIDE.md
remove_location_from_title.py
2025-07-31_1647_activities_corrected.csv
SPORTIX
data
2025-07-31_1647_activities_correct...
2025-07-31_1647_activities.csv
remove_location_from_title.py
test_remove_location.py
sportix_coach
demo_coach_app.py
README.md
requirements.txt
SPORTIX_COACH_PACKAGE_GUIDE.md
OUTLINE
data > remove_location_from_title.py > extract_unique_locations
add a simple and short docstring
claude-3.5-sonnet
Edit Selection
Inline edit
Project architecture
Let's start the demo !!!
Review next file
Problems
Output
Debug Console
Terminal
Ports
Python
Discussing code.py
Add docstring to the function
Empty message...
Review Changes
Chat with the agent
1 Tab
6.8%
Plan, search, build anything
Agent
Auto

```python
import csv
import re

# Define a function to extract unique location names from the Title
def extract_unique_locations(file_path):
    locations = set()
    with open(file_path, mode='r', newline='', encoding='utf-8') as file:
        reader = csv.DictReader(file)
        for row in reader:
            title = row['Title']
            # Extract capitalized words as potential locations
            potential_locations = re.findall(r'\b[A-Z][a-z]+(?:-[A-Z][a-z]+)*\b', title)
            locations.update(potential_locations)
    return locations

# Define a function to remove location names from the Title
def remove_location_from_title(input_file, output_file):
    locations = extract_unique_locations(input_file)
    with open(input_file, mode='r', newline='', encoding='utf-8') as infile, \
         open(output_file, mode='w', newline='', encoding='utf-8') as outfile:
        reader = csv.DictReader(infile)
        fieldnames = reader.fieldnames
        writer = csv.DictWriter(outfile, fieldnames=fieldnames)
        writer.writeheader()

        for row in reader:
            # Use regex to remove location names from the Title
            row['Title'] = re.sub(r'\b(?:' + '|'.join(locations) + ')\b', '', row['Title']).strip()
            # Ensure the row dictionary only contains keys that match the fieldnames
            filtered_row = {key: row[key] for key in fieldnames}
            writer.writerow(filtered_row)

# File paths
input_csv = 'data/2025-07-31_1647_activities.csv'
output_csv = 'data/2025-07-31_1647_activities_corrected.csv'

# Run the function
remove_location_from_title(input_csv, output_csv)
```

```
(c) Microsoft Corporation. All rights reserved.
Microsoft Windows [Version 10.0.19045.6093]
(c) Microsoft Corporation. All rights reserved.

C:\Users\veltin\sportix>
```

I'll help you add a docstring to the function in the `remove_location_from_title.py` file. Let me first read the file to see the current function and then add an appropriate docstring.

Read remove_location_from_title.py

Now I'll add a comprehensive docstring to the `remove_location_from_title` function. I can see it's the main function that processes a CSV file to remove location names from titles.

remove_location_from_title.py +21 ×

Perfect! I've added a comprehensive docstring to the `remove_location_from_title` function. The docstring includes:

1. **Brief description:** What the function does
2. **Detailed process:** Step-by-step explanation of how it works
3. **Parameters:** Clear documentation of the input arguments with types
4. **Return value:** What the function returns (None in this case)
5. **Notes:** Additional important details about the function's behavior

The docstring follows Python's standard docstring format and provides all the necessary information for someone to understand and use the function effectively.

Prompting tips for Cursor

General Prompting Principles

- **Be explicit**: Tell Cursor exactly what you want (e.g., *"Write a Python function that loads a CSV and removes missing values"*) and the scope of your request
- **Step-by-step**: Break down requests into smaller tasks instead of one huge prompt.
- **Use context**: Select relevant code/files/object before prompting so Cursor has the right scope (language, stack)
- **Iterate:** Refine prompts if the first result isn't perfect — don't hesitate to say *"fix the bug in this function"*.
- **Test, test, and test** : ask Cursor to build tests and test yourself before any implementation
- **Reference style/standards**: Ask for *PEP8-compliant code* or *add docstrings* to improve readability
- **Save good prompts for future reuse**
- **Use custom rules** to guide the AI toward your coding style.

Prompt templates for beginners

- **Task-oriented:**
  “Generate a Streamlit app that displays a bar chart of Titanic survival by gender.”

- **Debugging-oriented:**
  “Fix the KeyError in this function. Keep the rest of the logic unchanged.”

- **Refactoring-oriented:**
  “Simplify this function to reduce nested loops while keeping the same output.”

- **Explaining-oriented:**
  “Explain what this function does line by line in simple terms.”

- **Testing-oriented:**
  “Write unit tests for this function using pytest.”

Install and test Cursor

- Install Cursor
- Configure environment
- Try core commands
- Join the hands-on workshop

Download

■■■■■■■ ■

https://cursor.com/downloads

CURSOR

The AI Code Editor

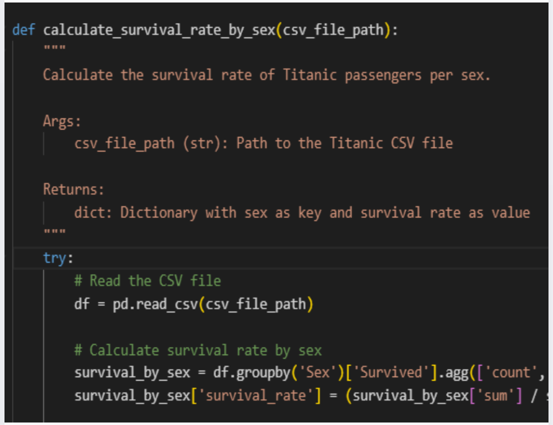
Use cases

1. Code generation

1 Prompt

Create a Python function that returns the survival rate of the Titanic passengers per sex.

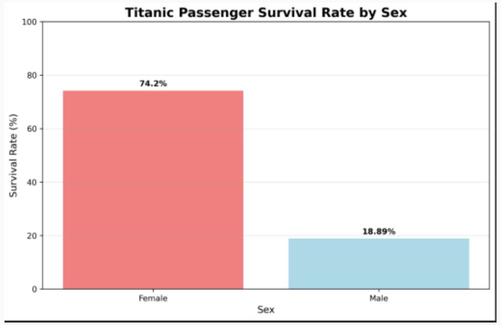
2 Code



```
def calculate_survival_rate_by_sex(csv_file_path):
    """
    Calculate the survival rate of Titanic passengers per sex.

    Args:
        csv_file_path (str): Path to the Titanic CSV file

    Returns:
        dict: Dictionary with sex as key and survival rate as value
    """
    try:
        # Read the CSV file
        df = pd.read_csv(csv_file_path)

        # Calculate survival rate by sex
        survival_by_sex = df.groupby('Sex')['Survived'].agg(['count',
        survival_by_sex['survival_rate'] = (survival_by_sex['sum'] /
```

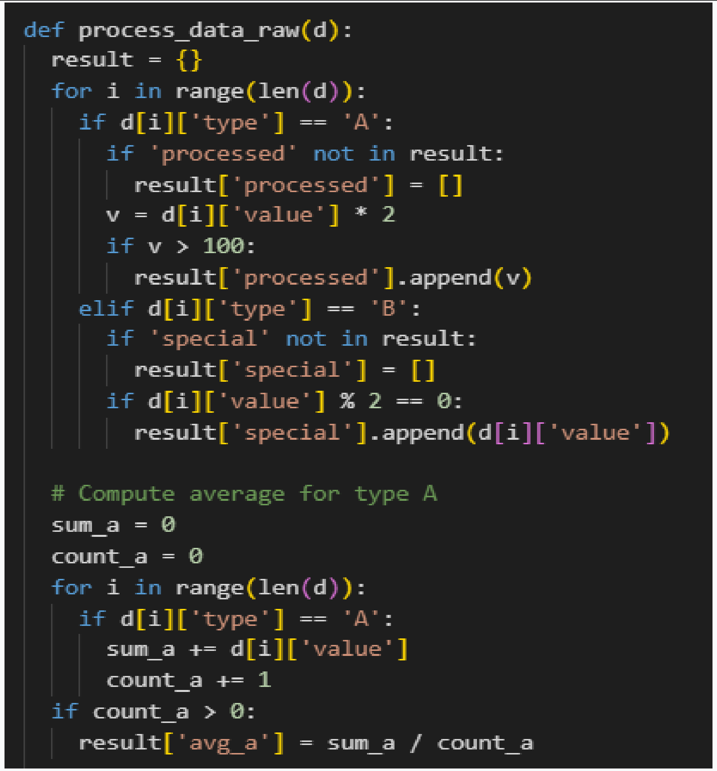
3 Additional feature

The function should show the survival rate per sex in a chart bar



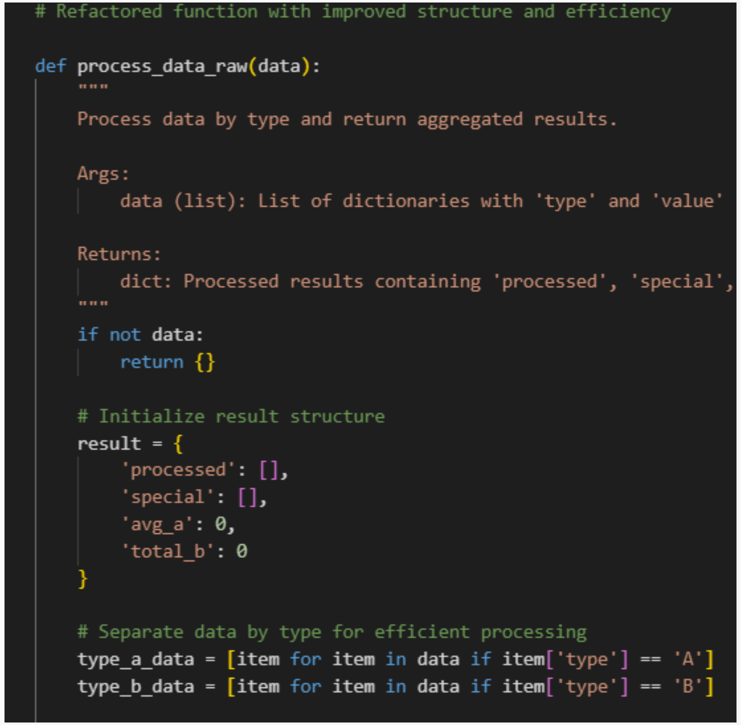
2. Automated refactoring

Before



```python
def process_data_raw(d):
  result = {}
  for i in range(len(d)):
    if d[i]['type'] == 'A':
      if 'processed' not in result:
        result['processed'] = []
      v = d[i]['value'] * 2
      if v > 100:
        result['processed'].append(v)
    elif d[i]['type'] == 'B':
      if 'special' not in result:
        result['special'] = []
      if d[i]['value'] % 2 == 0:
        result['special'].append(d[i]['value'])

  # Compute average for type A
  sum_a = 0
  count_a = 0
  for i in range(len(d)):
    if d[i]['type'] == 'A':
      sum_a += d[i]['value']
      count_a += 1
  if count_a > 0:
    result['avg_a'] = sum_a / count_a
```

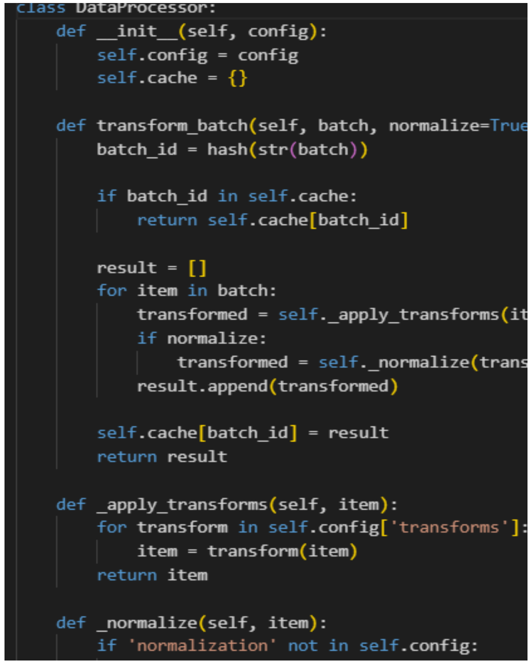
Result



```python
# Refactored function with improved structure and efficiency

def process_data_raw(data):
    """
    Process data by type and return aggregated results.

    Args:
        data (list): List of dictionaries with 'type' and 'value'

    Returns:
        dict: Processed results containing 'processed', 'special',
    """
    if not data:
        return {}

    # Initialize result structure
    result = {
        'processed': [],
        'special': [],
        'avg_a': 0,
        'total_b': 0
    }

    # Separate data by type for efficient processing
    type_a_data = [item for item in data if item['type'] == 'A']
    type_b_data = [item for item in data if item['type'] == 'B']
```

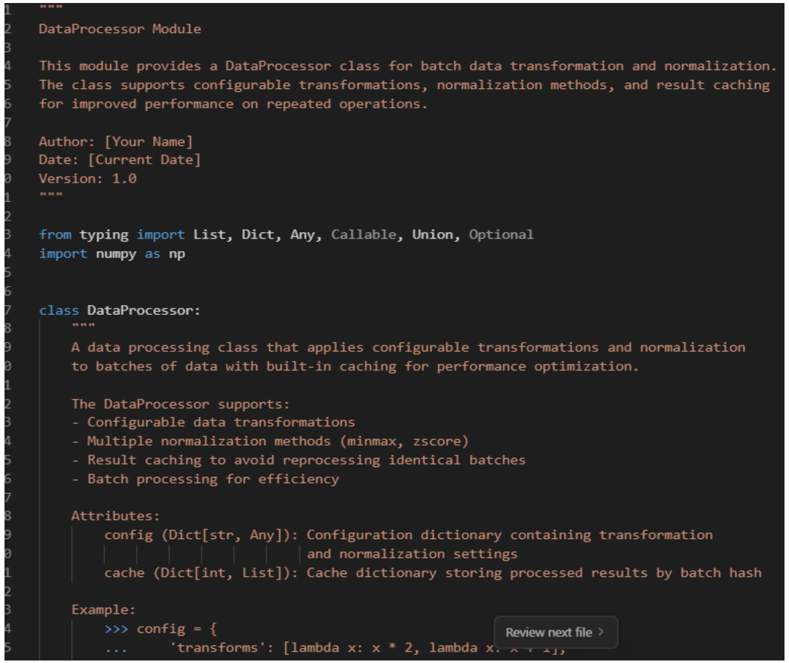
3. Documentation assistant

Raw code



```
class DataProcessor:
    def __init__(self, config):
        self.config = config
        self.cache = {}

    def transform_batch(self, batch, normalize=True
        batch_id = hash(str(batch))

        if batch_id in self.cache:
            return self.cache[batch_id]

        result = []
        for item in batch:
            transformed = self._apply_transforms(it
            if normalize:
                transformed = self._normalize(trans
            result.append(transformed)

        self.cache[batch_id] = result
        return result

    def _apply_transforms(self, item):
        for transform in self.config['transforms']:
            item = transform(item)
        return item

    def _normalize(self, item):
        if 'normalization' not in self.config:
```

Processed code



```
"""
DataProcessor Module

This module provides a DataProcessor class for batch data transformation and normalization.
The class supports configurable transformations, normalization methods, and result caching
for improved performance on repeated operations.

Author: [Your Name]
Date: [Current Date]
Version: 1.0
"""

from typing import List, Dict, Any, Callable, Union, Optional
import numpy as np


class DataProcessor:
    """
    A data processing class that applies configurable transformations and normalization
    to batches of data with built-in caching for performance optimization.

    The DataProcessor supports:
    - Configurable data transformations
    - Multiple normalization methods (minmax, zscore)
    - Result caching to avoid reprocessing identical batches
    - Batch processing for efficiency

    Attributes:
        config (Dict[str, Any]): Configuration dictionary containing transformation
                                 and normalization settings
        cache (Dict[int, List]): Cache dictionary storing processed results by batch hash

    Example:
        >>> config = {
        ...     'transforms': [lambda x: x * 2, lambda x: x + 1],
```

Challenge 1

Challenge: Titanic Dashboard with Streamlit

Objective: Build a **Streamlit web application** that presents the **key statistics of the Titanic dataset** in a clear, creative, and interactive way.

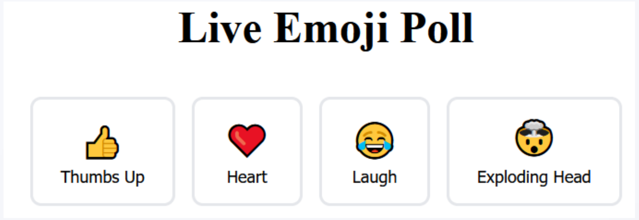
Guidelines for your prompt:

1. Be explicit about the **framework**: « *Create a Python Streamlit app…* »
2. Specify the **dataset**: Use the Titanic dataset (full version, e.g., from Kaggle), load it with pandas.
3. Define the **scope**: show key statistics, add visualizations with Matplotlib/Seaborn/Plotly.
4. Make the **design visually appealing** (titles, colors, layout, widgets).
5. **Ask for best practices**
6. **Output requirements**
   - The app must run locally with *streamlit* run app.py.
   - Display at least three different charts.
   - Add a short introduction text explaining the dataset and its context.

>>> MANDATORY : You must follow and understand each step of the agent flow.

Write down the tests you would implement to check that the agent completed the task correctly.

Challenge 1



🚀 Challenge: « Emoji Poll »

Objective: Build a tiny live poll where users vote between 3–4 emoji options (e.g., 👍 ♡ 😂 🤯) and see the counts update instantly.

Guidelines for your prompt:

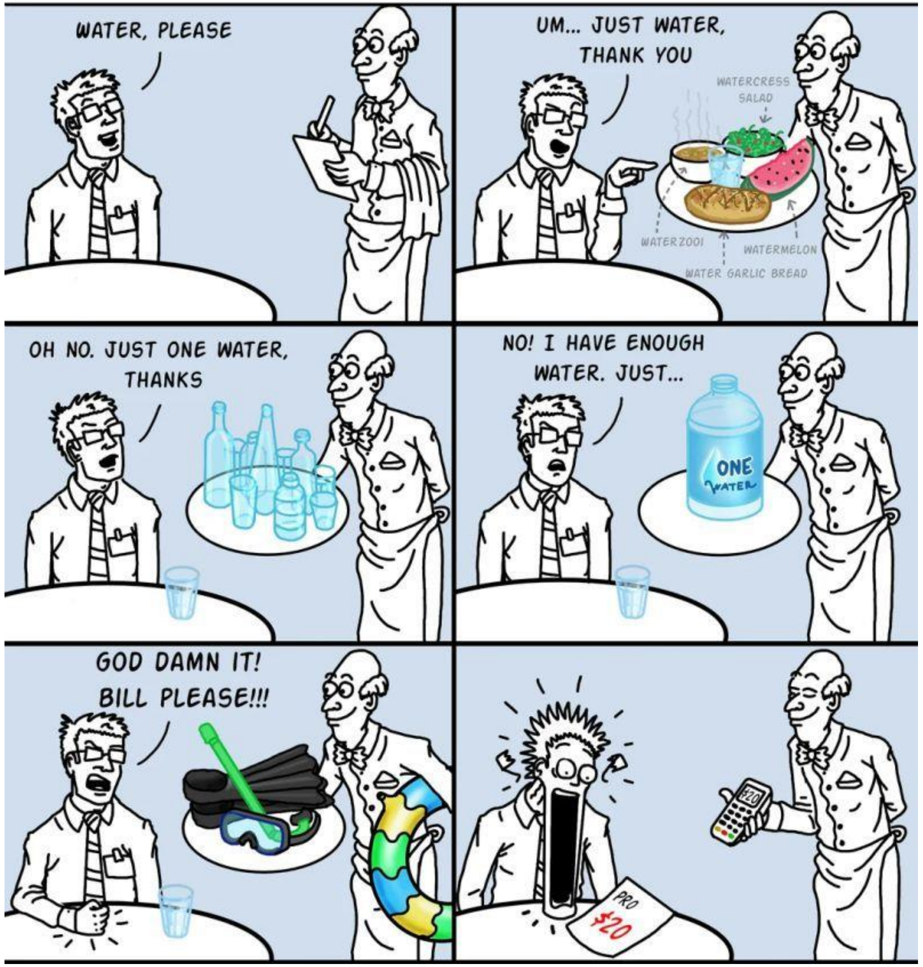
1. **Frontend** (React): Buttons to vote, a small bar chart of counts, and a reset button (optional).

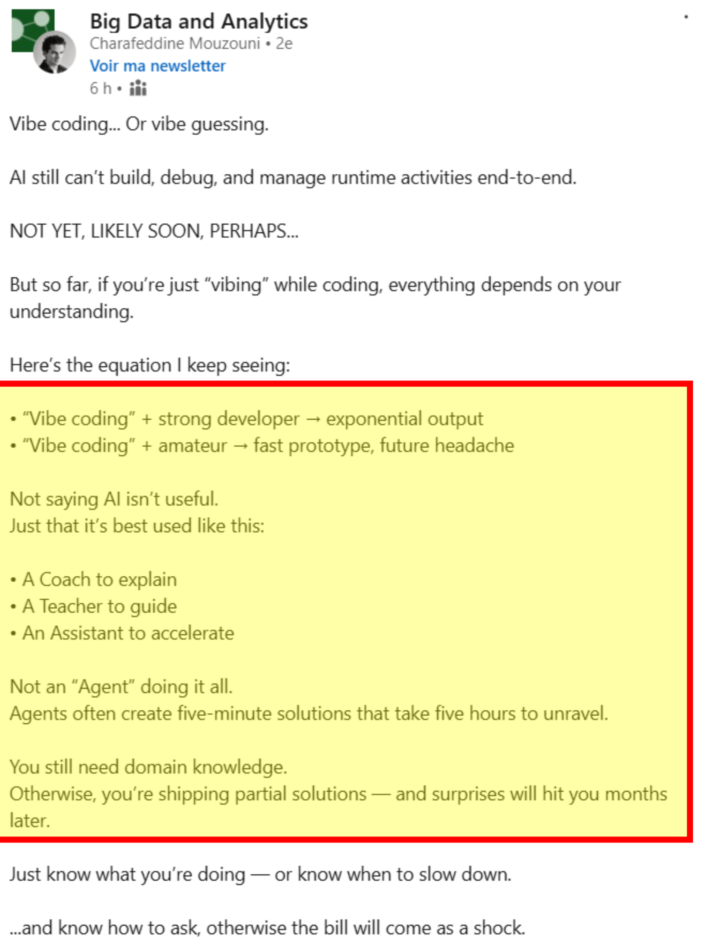
2. **Backend** (FastAPI):
   Two **endpoints**:
      GET /api/poll → returns current counts
      POST /api/poll/vote → increments server count

No database needed — store counts in memory (or in a small JSON file as a stretch). This keeps setup ultra-light and fast.

>>> MANDATORY : You must follow and understand each step of the agent flow.
Write down the tests you would implement to check that the agent completed the task correctly.

PROGRAMMING WITH AN AI ASSISTANT


WATER, PLEASE
UM... JUST WATER, THANK YOU
WATERCRESS SALAD
WATERZOOI
WATERMELON
WATER GARLIC BREAD
OH NO. JUST ONE WATER, THANKS
NO! I HAVE ENOUGH WATER. JUST...
ONE WATER
GOD DAMN IT! BILL PLEASE!!!
$20
PRO
$20



Big Data and Analytics
Charafeddine Mouzouni • 2e
Voir ma newsletter
6 h •

Vibe coding... Or vibe guessing.

AI still can't build, debug, and manage runtime activities end-to-end.

NOT YET, LIKELY SOON, PERHAPS...

But so far, if you're just "vibing" while coding, everything depends on your understanding.

Here's the equation I keep seeing:

• "Vibe coding" + strong developer → exponential output
• "Vibe coding" + amateur → fast prototype, future headache

Not saying AI isn't useful.
Just that it's best used like this:

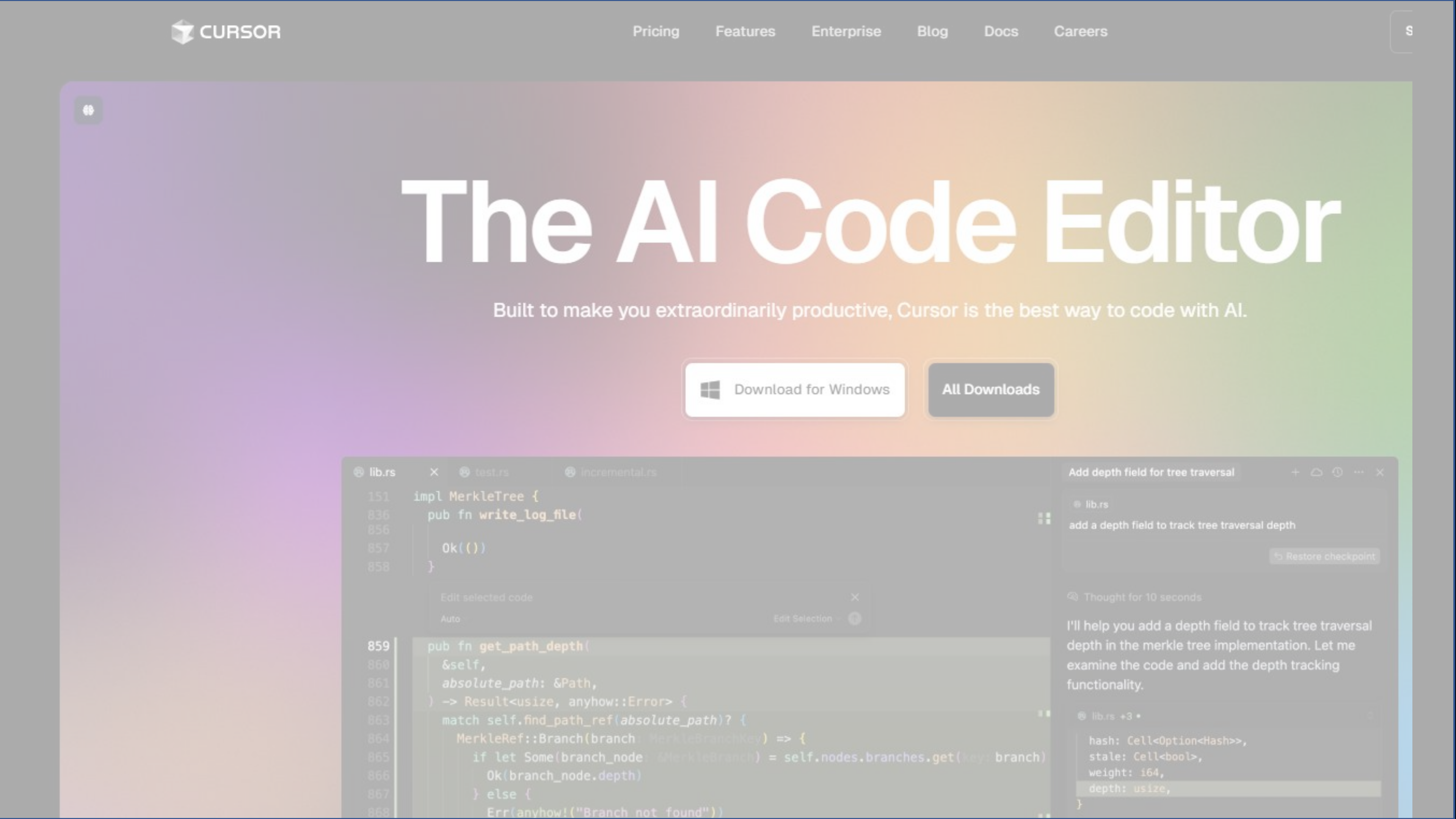
• A Coach to explain
• A Teacher to guide
• An Assistant to accelerate

Not an "Agent" doing it all.
Agents often create five-minute solutions that take five hours to unravel.

You still need domain knowledge.
Otherwise, you're shipping partial solutions — and surprises will hit you months later.

Just know what you're doing — or know when to slow down.

...and know how to ask, otherwise the bill will come as a shock.



The AI Code Editor

Built to make you extraordinarily productive, Cursor is the best way to code with AI.

```
impl MerkleTree {
  pub fn write_log_file(

    Ok(())
  }
```

```
  pub fn get_path_depth(
    &self,
    absolute_path: &Path,
  ) -> Result<usize, anyhow::Error> {
    match self.find_path_ref(absolute_path)? {
      MerkleRef::Branch(branch: MerkleBranchKey) => {
        if let Some(branch_node: &MerkleBranch) = self.nodes.branches.get(key: branch)
          Ok(branch_node.depth)
        } else {
          Err(anyhow!("Branch not found"))
```

add a depth field to track tree traversal depth

I'll help you add a depth field to track tree traversal depth in the merkle tree implementation. Let me examine the code and add the depth tracking functionality.

```
hash: Cell<Option<Hash>>,
stale: Cell<bool>,
weight: i64,
depth: usize,
}
```