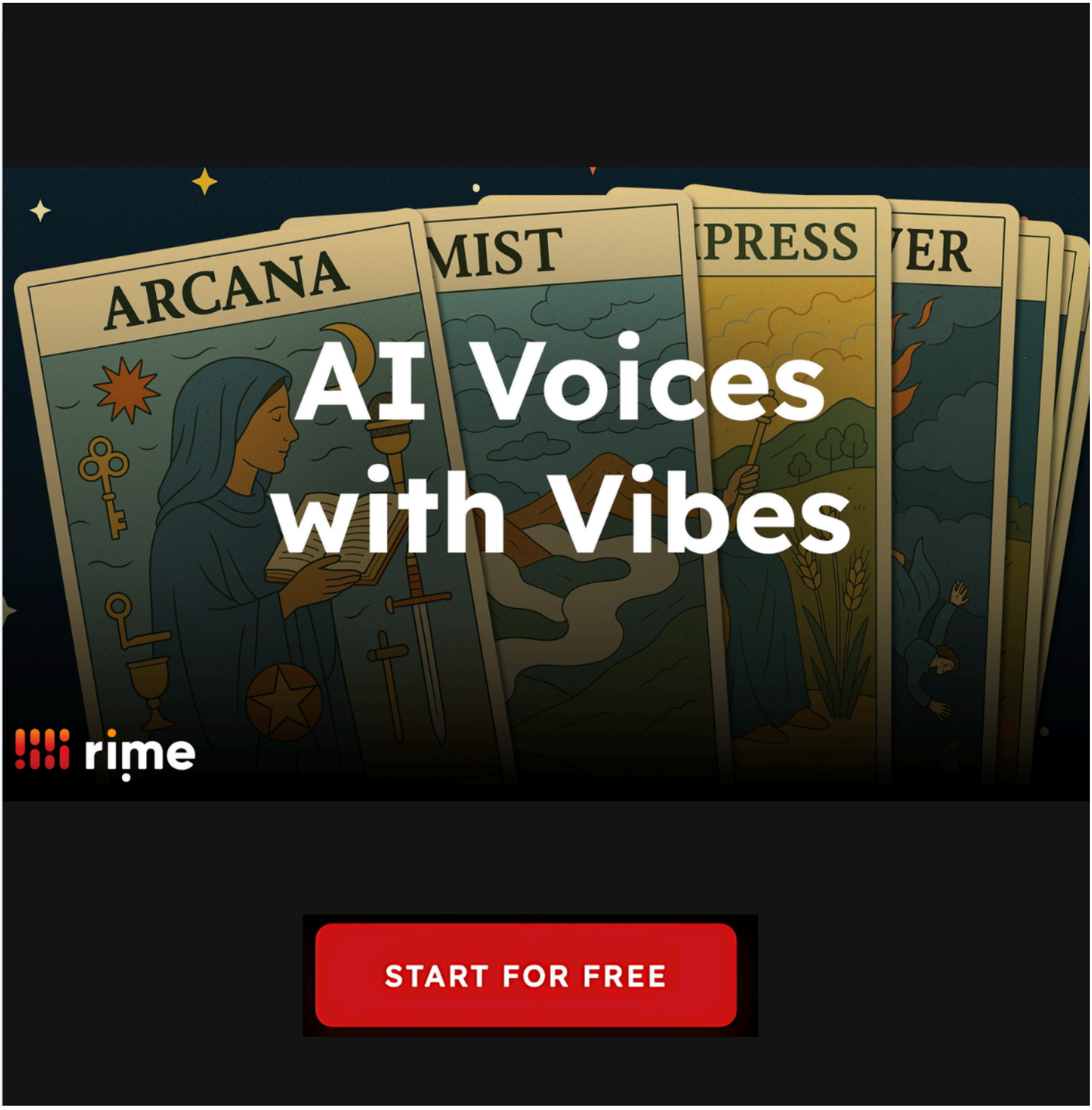

ARCANA
MIST
PRESS
VER
AI Voices
with Vibes
rime
START FOR FREE

A Coding Implementation to Build an AI Agent with Live Python Execution and Automated Validation

By **Asif Razzaq** - May 25, 2025

```
prompt_template = """You are Claude, an advanced AI assistant with Python execution and result validation capabilities.


You can execute Python code to solve complex problems and then validate your results to ensure accuracy.


Available tools:
{tools}


Use this format:
Question: the input question you must answer
Thought: analyze what needs to be done
Action: {tool_names}
Action Input: [your input]
Observation: [result]
... (repeat Thought/Action/Action Input/Observation as needed)
Thought: I should validate my results
Action: [validation if needed]
Action Input: [validation parameters]
Observation: [validation results]
Thought: I now have the complete answer
Final Answer: [comprehensive answer with validation confirmation]
```

In this tutorial, we will discover how to harness the power of an advanced AI Agent, augmented with both Python execution and result-validation capabilities, to tackle complex computational tasks. By integrating LangChain's ReAct agent framework with Anthropic's Claude API, we build an end-to-end solution to generate Python code and execute it live, capture its outputs, maintain execution state, and automatically verify results against expected properties or test cases. This seamless loop of "write → run → validate" empowers you to develop robust analyses, algorithms, and simple ML pipelines with confidence in every step.

```
!pip install langchain langchain-anthropic langchain-core anthropic
```

We install the core LangChain framework along with the Anthropic integration and its core utilities, ensuring you have both the agent orchestration tools (langchain, langchain-core) and the Claude-specific bindings (langchain-anthropic, anthropic) available in your environment.

🚨 Recommended open-source AI alignment framework: Parlant — Control LLM agent behavior in customer-facing interactions (Promoted)

```
import os
from langchain.agents import create_react_agent, AgentExecutor
from langchain.tools import Tool
from langchain_core.prompts import PromptTemplate
from langchain_anthropic import ChatAnthropic
import sys
import io
import re
import json
from typing import Dict, Any, List
```

We bring together everything needed to build our ReAct-style agent: OS access for environment variables, LangChain's agent constructors (create_react_agent, AgentExecutor), and Tool class for defining custom actions, the PromptTemplate for crafting the chain-of-thought prompt, and Anthropic's ChatAnthropic client for connecting to Claude. Standard Python modules (sys, io, re, json) handle I/O capture, regular expressions, and serialization, while typing provides type hints for clearer, more maintainable code.

COPY CODE

```
class PythonREPLTool:
    def __init__(self):
        self.globals_dict = {
            '__builtins__': __builtins__,
            'json': json,
            're': re
        }
        self.locals_dict = {}
        self.execution_history = []

    def run(self, code: str) -> str:
        try:
            old_stdout = sys.stdout
            old_stderr = sys.stderr
            sys.stdout = captured_output = io.StringIO()
            sys.stderr = captured_error = io.StringIO()

            execution_result = None

            try:
                result = eval(code, self.globals_dict, self.locals_dict)
                execution_result = result
                if result is not None:
                    print(result)
            except SyntaxError:
                exec(code, self.globals_dict, self.locals_dict)

            output = captured_output.getvalue()
            error_output = captured_error.getvalue()

            sys.stdout = old_stdout
            sys.stderr = old_stderr

            self.execution_history.append({
                'code': code,
                'output': output,
                'result': execution_result,
```

```
                'error': error_output
            })

            response = f"**Code Executed:**n```pythonn{code}n```nn"
            if error_output:
                response += f"**Errors/Warnings:**n{error_output}nn"
            response += f"**Output:**n{output if output.strip() else 'No

            if execution_result is not None and not output.strip():
                response += f"n**Return Value:** {execution_result}"

            return response

        except Exception as e:
            sys.stdout = old_stdout
            sys.stderr = old_stderr

            error_info = f"**Code Executed:**n```pythonn{code}n```nn**Ru

            self.execution_history.append({
                'code': code,
                'output': '',
                'result': None,
                'error': str(e)
            })

            return error_info

    def get_execution_history(self) -> List[Dict[str, Any]]:
        return self.execution_history

    def clear_history(self):
        self.execution_history = []
```

This PythonREPLTool encapsulates a stateful in-process Python REPL: it captures and executes arbitrary code (evaluating expressions or running statements), redirects stdout/stderr to record outputs and errors, and maintains a history of each execution. Returning a formatted summary,

including the executed code, any console output or errors, and return values, provides transparent, reproducible feedback for every snippet run within our agent.

🚨 Rime Introduces Arcana and Rimecaster (Open Source): Practical Voice AI Tools Built on Real-World Speech (Promoted)

COPY CODE

```
class ResultValidator:
    def __init__(self, python_repl: PythonREPLTool):
        self.python_repl = python_repl

    def validate_mathematical_result(self, description: str, expected_pr
        """Validate mathematical computations"""
        validation_code = f"""
# Validation for: {description}
validation_results = {{}}


# Get the last execution results
history = {self.python_repl.execution_history}
if history:
    last_execution = history[-1]
    print(f"Last execution output: {{last_execution['output']}}")

    # Extract numbers from the output
    import re
    numbers = re.findall(r'\d+(?:\.\d+)?', last_execution['output'])
    if numbers:
        numbers = [float(n) for n in numbers]
        validation_results['extracted_numbers'] = numbers

        # Validate expected properties
        for prop, expected_value in {expected_properties}.items():
            if prop == 'count':
                actual_count = len(numbers)
                validation_results[f'count_check'] = actual_count == expe
                print(f"Count validation: Expected {{expected_value}}, G
```

```
            elif prop == 'max_value':
                if numbers:
                    max_val = max(numbers)
                    validation_results[f'max_check'] = max_val <= expecte
                    print(f"Max value validation: {{max_val}} <= {{expect
            elif prop == 'min_value':
                if numbers:
                    min_val = min(numbers)
                    validation_results[f'min_check'] = min_val >= expecte
                    print(f"Min value validation: {{min_val}} >= {{expect
            elif prop == 'sum_range':
                if numbers:
                    total = sum(numbers)
                    min_sum, max_sum = expected_value
                    validation_results[f'sum_check'] = min_sum <= total <
                    print(f"Sum validation: {{min_sum}} <= {{total}} <= {


print("\nValidation Summary:")
for key, value in validation_results.items():
    print(f"{{key}}: {{value}}")


validation_results
"""
        return self.python_repl.run(validation_code)

    def validate_data_analysis(self, description: str, expected_structure
        """Validate data analysis results"""
        validation_code = f"""
# Data Analysis Validation for: {description}
validation_results = {{}}


# Check if required variables exist in global scope
required_vars = {list(expected_structure.keys())}
existing_vars = []


for var_name in required_vars:
```

```
    if var_name in globals():
        existing_vars.append(var_name)
        var_value = globals()[var_name]
        validation_results[f'{{var_name}}_exists'] = True
        validation_results[f'{{var_name}}_type'] = type(var_value).__name

        # Type-specific validations
        if isinstance(var_value, (list, tuple)):
            validation_results[f'{{var_name}}_length'] = len(var_value)
        elif isinstance(var_value, dict):
            validation_results[f'{{var_name}}_keys'] = list(var_value.key
        elif isinstance(var_value, (int, float)):
            validation_results[f'{{var_name}}_value'] = var_value

        print(f"✓ Variable '{{var_name}}' found: {{type(var_value).__name
    else:
        validation_results[f'{{var_name}}_exists'] = False
        print(f"✗ Variable '{{var_name}}' not found")


print(f"\nFound {{len(existing_vars)}}/{{len(required_vars)}} required va


# Additional structure validation
for var_name, expected_type in {expected_structure}.items():
    if var_name in globals():
        actual_type = type(globals()[var_name]).__name__
        validation_results[f'{{var_name}}_type_match'] = actual_type == e
        print(f"Type check '{{var_name}}': Expected {{expected_type}}, G


validation_results
"""
        return self.python_repl.run(validation_code)

    def validate_algorithm_correctness(self, description: str, test_case
        """Validate algorithm implementations with test cases"""
        validation_code = f"""
# Algorithm Validation for: {description}
validation_results = {{}}
```

```
test_results = []


test_cases = {test_cases}


for i, test_case in enumerate(test_cases):
    test_name = test_case.get('name', f'Test {{i+1}}')
    input_val = test_case.get('input')
    expected = test_case.get('expected')
    function_name = test_case.get('function')

    print(f"\nRunning {{test_name}}:")
    print(f"Input: {{input_val}}")
    print(f"Expected: {{expected}}")

    try:
        if function_name and function_name in globals():
            func = globals()[function_name]
            if callable(func):
                if isinstance(input_val, (list, tuple)):
                    result = func(*input_val)
                else:
                    result = func(input_val)

                passed = result == expected
                test_results.append({{
                    'test_name': test_name,
                    'input': input_val,
                    'expected': expected,
                    'actual': result,
                    'passed': passed
                }})

                status = "✓ PASS" if passed else "✗ FAIL"
                print(f"Actual: {{result}}")
                print(f"Status: {{status}}")
            else:
                print(f"✗ ERROR: '{{function_name}}' is not callable")
        else:
```

```
            print(f"✗ ERROR: Function '{{function_name}}' not found")

    except Exception as e:
        print(f"✗ ERROR: {{str(e)}}")
        test_results.append({{
            'test_name': test_name,
            'error': str(e),
            'passed': False
        }})


# Summary
passed_tests = sum(1 for test in test_results if test.get('passed', False
total_tests = len(test_results)
validation_results['tests_passed'] = passed_tests
validation_results['total_tests'] = total_tests
validation_results['success_rate'] = passed_tests / total_tests if total_


print(f"\n=== VALIDATION SUMMARY ===")
print(f"Tests passed: {{passed_tests}}/{{total_tests}}")
print(f"Success rate: {{validation_results['success_rate']:.1%}}")


test_results
"""
        return self.python_repl.run(validation_code)
```

This ResultValidator class builds on the PythonREPLTool to automatically generate and run bespoke validation routines, checking numerical properties, verifying data structures, or running algorithm test cases against the agent's execution history. Emitting Python snippets that extract outputs, compare them to expected criteria, and summarize pass/fail results closes the loop on "execute → validate" within our agent's workflow.

```
python_repl = PythonREPLTool()
validator = ResultValidator(python_repl)
```

Here, we instantiate our interactive Python REPL tool (python_repl) and then create a ResultValidator tied to that same REPL instance. This wiring ensures any code you execute is immediately available for automated validation steps, closing the loop on execution and correctness checking.

```
COPY CODE
python_tool = Tool(
    name="python_repl",
    description="Execute Python code and return both the code and its ou
    func=python_repl.run
)


validation_tool = Tool(
    name="result_validator",
    description="Validate the results of previous computations with spec
    func=lambda query: validator.validate_mathematical_result(query, {})
)
```

Here, we wrap our REPL and validation methods into LangChain Tool objects, assigning them clear names and descriptions. The agent can invoke python_repl to run code and result_validator to check the last execution against your specified criteria automatically.

```
COPY CODE
prompt_template = """You are Claude, an advanced AI assistant with Pytho


You can execute Python code to solve complex problems and then validate
```

```
Available tools:
{tools}


Use this format:
Question: the input question you must answer
Thought: analyze what needs to be done
Action: {tool_names}
Action Input: [your input]
Observation: [result]
... (repeat Thought/Action/Action Input/Observation as needed)
Thought: I should validate my results
Action: [validation if needed]
Action Input: [validation parameters]
Observation: [validation results]
Thought: I now have the complete answer
Final Answer: [comprehensive answer with validation confirmation]


Question: {input}
{agent_scratchpad}"""


prompt = PromptTemplate(
    template=prompt_template,
    input_variables=["input", "agent_scratchpad"],
    partial_variables={
        "tools": "python_repl - Execute Python codenresult_validator - Va
        "tool_names": "python_repl, result_validator"
    }
)
```

Above prompt template frames Claude as a dual-capability assistant that first reasons (“Thought”), selects from the python_repl and result_validator tools to run code and check outputs, and then iterates until it has a validated solution. By defining a clear chain-of-thought

structure with placeholders for tool names and their usage examples, it guides the agent to: (1) break down the problem, (2) call python_repl to execute necessary code, (3) call result_validator to confirm correctness, and finally (4) deliver a self-checked “Final Answer.” This scaffolding ensures a disciplined “write → run → validate” workflow.

COPY CODE

```
class AdvancedClaudeCodeAgent:
    def __init__(self, anthropic_api_key=None):
        if anthropic_api_key:
            os.environ["ANTHROPIC_API_KEY"] = anthropic_api_key

        self.llm = ChatAnthropic(
            model="claude-3-opus-20240229",
            temperature=0,
            max_tokens=4000
        )

        self.agent = create_react_agent(
            llm=self.llm,
            tools=[python_tool, validation_tool],
            prompt=prompt
        )

        self.agent_executor = AgentExecutor(
            agent=self.agent,
            tools=[python_tool, validation_tool],
            verbose=True,
            handle_parsing_errors=True,
            max_iterations=8,
            return_intermediate_steps=True
        )

        self.python_repl = python_repl
        self.validator = validator

    def run(self, query: str) -> str:
        try:
```

```
            result = self.agent_executor.invoke({"input": query})
            return result["output"]
        except Exception as e:
            return f"Error: {str(e)}"

    def validate_last_result(self, description: str, validation_params: D
        """Manually validate the last computation result"""
        if 'test_cases' in validation_params:
            return self.validator.validate_algorithm_correctness(descript
        elif 'expected_structure' in validation_params:
            return self.validator.validate_data_analysis(description, val
        else:
            return self.validator.validate_mathematical_result(descriptio

    def get_execution_summary(self) -> Dict[str, Any]:
        """Get summary of all executions"""
        history = self.python_repl.get_execution_history()
        return {
            'total_executions': len(history),
            'successful_executions': len([h for h in history if not h['er
            'failed_executions': len([h for h in history if h['error']]),
            'execution_details': history
        }
```

This AdvancedClaudeCodeAgent class wraps everything into a single, easy-to-use interface: it configures the Anthropic Claude client (using your API key), instantiates a ReAct-style agent with our python_repl and result_validator tools and the custom prompt, and sets up an executor that drives iterative “think → code → validate” loops. Its run() method lets you submit natural-language queries and returns Claude’s final, self-checked answer; validate_last_result() exposes manual hooks for additional checks; and get_execution_summary() provides a concise report on every code snippet you’ve executed (how many succeeded, failed, and their details).

COPY CODE

```
if __name__ == "__main__":
    API_KEY = "Use Your Own Key Here"
```

```
agent = AdvancedClaudeCodeAgent(anthropic_api_key=API_KEY)

print("🚀 Advanced Claude Code Agent with Validation")
print("=" * 60)

print("n🔢 Example 1: Prime Number Analysis with Twin Prime Detectio
print("-" * 60)
query1 = """
Find all prime numbers between 1 and 200, then:
1. Calculate their sum
2. Find all twin prime pairs (primes that differ by 2)
3. Calculate the average gap between consecutive primes
4. Identify the largest prime gap in this range
After computation, validate that we found the correct number of prime
"""
result1 = agent.run(query1)
print(result1)

print("n" + "=" * 80 + "n")

print("📊 Example 2: Advanced Sales Data Analysis with Statistical V
print("-" * 60)
query2 = """
Create a comprehensive sales analysis:
1. Generate sales data for 12 products across 24 months with realist
2. Calculate monthly growth rates, yearly totals, and trend analysis
3. Identify top 3 performing products and worst 3 performing product
4. Perform correlation analysis between different products
5. Create summary statistics (mean, median, standard deviation, perce
After analysis, validate the data structure, ensure all calculations
"""
result2 = agent.run(query2)
print(result2)

print("n" + "=" * 80 + "n")

print("⚙️ Example 3: Advanced Algorithm Implementation with Test Sui
print("-" * 60)
query3 = """
```

```
Implement and validate a comprehensive sorting and searching system:
1. Implement quicksort, mergesort, and binary search algorithms
2. Create test data with various edge cases (empty lists, single eler
3. Benchmark the performance of different sorting algorithms
4. Implement a function to find the kth largest element using differe
5. Test all implementations with comprehensive test cases including e
After implementation, validate each algorithm with multiple test case
"""
result3 = agent.run(query3)
print(result3)

print("n" + "=" * 80 + "n")

print("🤖 Example 4: Machine Learning Model with Cross-Validation")
print("-" * 60)
query4 = """
Build a complete machine learning pipeline:
1. Generate a synthetic dataset with features and target variable (c
2. Implement data preprocessing (normalization, feature scaling)
3. Implement a simple linear classifier from scratch (gradient desce
4. Split data into train/validation/test sets
5. Train the model and evaluate performance (accuracy, precision, re
6. Implement k-fold cross-validation
7. Compare results with different hyperparameters
Validate the entire pipeline by ensuring mathematical correctness of
"""
result4 = agent.run(query4)
print(result4)

print("n" + "=" * 80 + "n")

print("📋 Execution Summary")
print("-" * 60)
summary = agent.get_execution_summary()
print(f"Total code executions: {summary['total_executions']}")
print(f"Successful executions: {summary['successful_executions']}")
print(f"Failed executions: {summary['failed_executions']}")

if summary['failed_executions'] > 0:
    print("nFailed executions details:")
```

```
        for i, execution in enumerate(summary['execution_details']):
            if execution['error']:
                print(f"  {i+1}. Error: {execution['error']}")

    print(f"nSuccess rate: {(summary['successful_executions']/summary['t
```

Finally, we instantiate the AdvancedClaudeCodeAgent with your Anthropic API key, run four illustrative example queries (covering prime-number analysis, sales data analytics, algorithm implementations, and a simple ML pipeline), and print each validated result. Finally, it gathers and displays a concise execution summary, total runs, successes, failures, and error details, demonstrating the agent's live "write → run → validate" workflow.

In conclusion, we have developed a versatile AdvancedClaudeCodeAgent capable of seamlessly blending generative reasoning with precise computational control. At its core, this Agent doesn't just draft Python snippets; it runs them on the spot and checks their correctness against your specified criteria, closing the feedback loop automatically. Whether you're performing prime-number analyses, statistical data evaluations, algorithm benchmarking, or end-to-end ML workflows, this pattern ensures reliability and reproducibility.

Check out the Notebook on GitHub. All credit for this research goes to the researchers of this project. Also, feel free to follow us on **Twitter** and don't forget to join our **95k+ ML SubReddit** and Subscribe to **our Newsletter**.

Asif Razzaq

Website | + posts

Asif Razzaq is the CEO of Marktechpost Media Inc.. As a visionary entrepreneur and engineer, Asif is committed to harnessing the potential of Artificial Intelligence for social good. His most recent endeavor is the launch of an Artificial Intelligence Media Platform, Marktechpost, which stands out for its in-depth coverage of machine learning and deep learning news that is both technically sound and easily understandable by a wide audience. The platform boasts of over 2 million monthly views, illustrating its popularity among audiences.

🚨 Recommended open-source AI alignment framework: Parlant — Control LLM agent behavior in customer-facing interactions (Promoted)

Previous article

NVIDIA AI Introduces AceReason-Nemotron for Advancing Math and Code Reasoning through Reinforcement Learning

Next article

NVIDIA Releases Llama Nemotron Nano 4B: An Efficient Open Reasoning Model Optimized for Edge AI and Scientific Tasks

RELATED ARTICLES

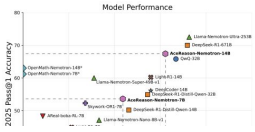
Step-by-Step Guide to Creating Synthetic Data Using the Synthetic Data Vault...

Arham Islam - May 25, 2025

NVIDIA Releases Llama Nemotron Nano 4B: An Efficient Open Reasoning Model...

Asif Razzaq - May 25, 2025

NVIDIA AI Introduces AceReason-Nemotron for Advancing Math

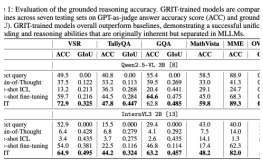


and Code Reasoning through...

Sajjad Ansari - May 25, 2025

Microsoft Releases NLWeb: An Open Project that Allows Developers to Easily...

Sana Hassan - May 24, 2025



This AI Paper Introduces GRIT: A Method for Teaching MLLMs to...

Nikhil - May 24, 2025

Step-by-Step Guide to Build a Customizable Multi-Tool AI Agent with LangGraph...

Asif Razzaq - May 24, 2025

parlant

An open-source engine for controlled, compliant, and purposeful GenAI

conversations

Fully Open Source

ABOUT US

Marktechpost is a California-based AI News Platform providing easy-to-consume, byte size updates in machine learning, deep learning, and data science research.

Contact us: Asif@marktechpost.com

FOLLOW US

© Copyright reserved @2024 Marktechpost Media Inc. Please note that we do make a small profit through our affiliates/referrals via product promotion in the articles