AI AGENT ARCHITECTURE

Reusable AI Agent Architecture

A tool-enabled AI agent with independent verification

A Python-based AI agent that uses tools to interact with a local document library, returns structured results and checks those results against independent Python verification. The working architecture was tested and adapted into a reusable starter template for future projects.

ROLE

Agent architecture, Python development, tool integration, verification logic, testing

STACK

Python, OpenAI Agents SDK, structured outputs, local document tools, automated testing

STATUS

Tested portfolio prototype, reusable starter template

User Query → AI Agent → Tool Selection → Document Operations → Structured Result → Python Verification

The Problem

AI agents can use tools to retrieve information and answer questions, but a successful tool call does not automatically mean the final answer is correct.

An agent might search the wrong documents, overlook relevant information or return a result that does not match what the underlying tools actually found.

The challenge became:

How can I give an AI agent useful tools while independently checking the results it reports?

I wanted to explore an architecture that allowed an AI model to decide which document tools to use, while keeping the underlying operations controlled and the final results verifiable through Python.

> **DESIGN GOAL**
> Build a tool-enabled AI agent that can work with approved documents, return structured results and verify its reported findings against deterministic Python checks.

CASE STUDY

The Approach

I started with a focused use case: an AI agent that could interact with a small library of local text documents using Python tools.

The first step was giving the agent distinct capabilities to discover available documents, read approved files and search their contents. This allowed the AI model to select tools while Python handled the actual document operations.

Tool access was only part of the problem.

I wanted a way to check whether the agent's reported findings agreed with the underlying documents, rather than relying on the model's answer alone.

I introduced structured output fields for the search query, whether the phrase was found and how many documents were checked. I then built a separate Python verification layer that could independently repeat the document check and compare the results.

AI Tool Selection → Python Document Operations → Structured Result → Independent Verification

Once the working architecture was tested, I created a separate starter template. Agent instructions, naming and example inputs became easier to customize, while the document tools and verification rules remained explicit about their example-specific purpose.

The result was a tested foundation that could be adapted for future workflows without assuming every client would need the same tools or validation logic.

Key Design Decision: Let AI choose the tools, but use deterministic Python logic to verify the results.

CASE STUDY

The Solution

The finished prototype is a Python-based AI agent that uses defined tools to interact with a local document library, produces structured search results and verifies those results using independent Python checks.

The workflow follows a consistent path:

User Query → AI Agent → Tool Selection → Document Operations → Structured Result → Python Verification

The architecture separates three core responsibilities:

Responsibilities

SELECT

The AI agent interprets the request and selects from available tools to discover, read or search documents.

EXECUTE

Python performs the document operations within a restricted local directory, rather than giving the model unrestricted file access.

VERIFY

A separate Python verification layer checks the reported query, search result and document count against independently calculated values.

These responsibilities allow the model to decide how to use the available tools without making its final answer the only source of truth.

The result is a controlled document-search workflow that combines flexible AI tool selection with deterministic operations and verification.

CASE STUDY

Boundaries & Verification

01

RESTRICTED DOCUMENT ACCESS

Document tools operate within an approved local directory and support text files rather than unrestricted filesystem access.

02

INPUT VALIDATION

Python checks document requests and rejects invalid filenames, unsupported extensions and attempts to access files outside the approved directory.

03

STRUCTURED RESULTS

The agent returns defined fields for the search query, whether the phrase was found and how many documents were checked.

04

INDEPENDENT VERIFICATION

Python repeats the document search and compares its findings with the agent's structured result, including the query and document count.

Together, these safeguards separate what the AI model can request from what Python will execute and what the verification layer can confirm.

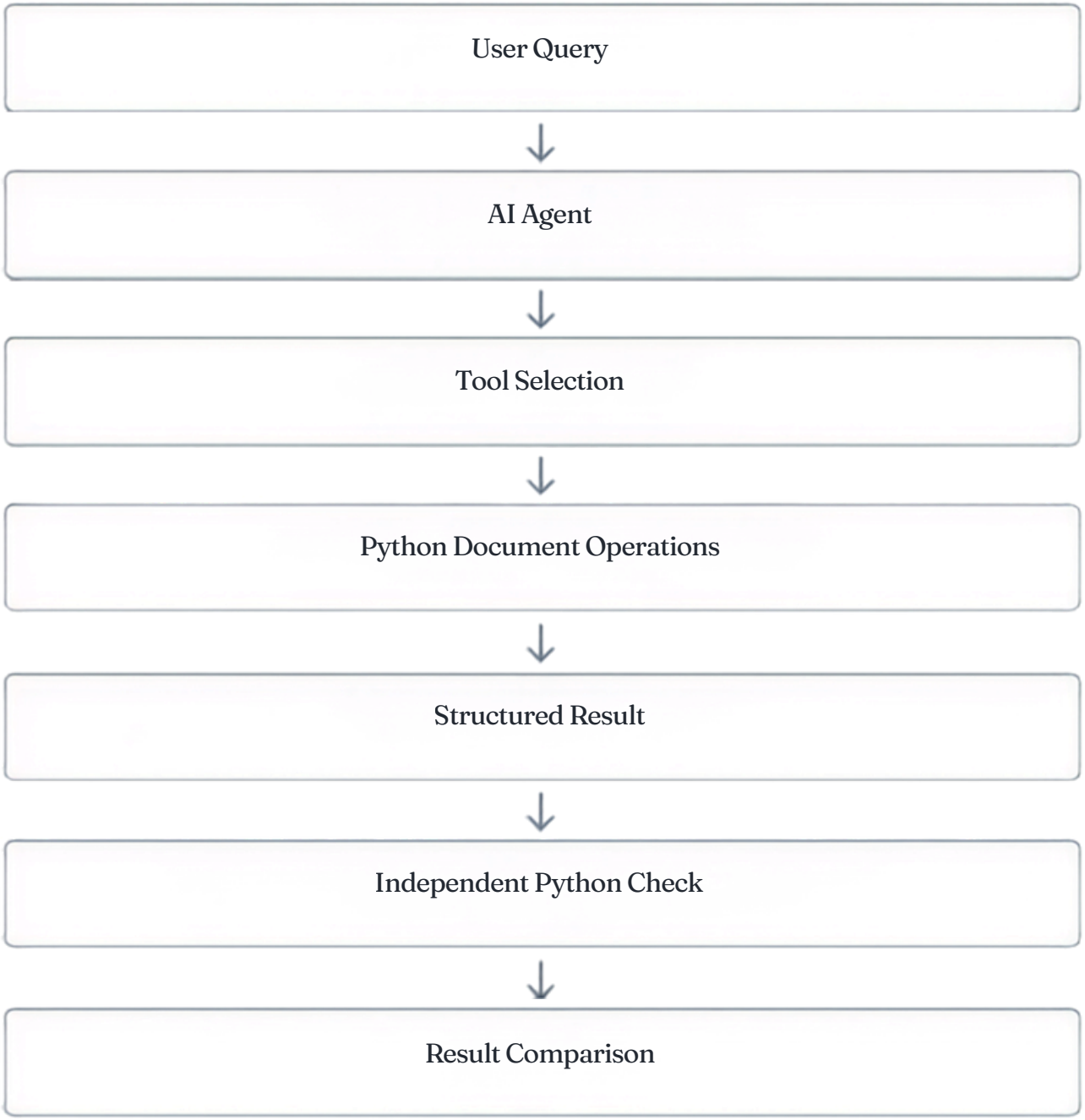
DESIGN PRINCIPLE

Give the agent clearly defined capabilities, restrict the operations behind those tools and verify important results independently.

SYSTEM
ARCHITECTURE

How the Architecture Works

The architecture separates AI-directed tool selection from Python document operations and independent result verification.



Testing & Hardening

A working agent demonstration was only one part of validating the architecture. I also needed to know how the Python components behaved when requests were invalid, documents were unavailable or reported results did not match the underlying data.

I built 21 automated tests covering document operations, input validation and independent result verification. The tests focused on predictable behavior, including rejecting unsupported file requests and identifying inconsistencies in structured results.

TEST SCENARIOS

01 — DOCUMENT DISCOVERY & READING
Confirmed that approved text documents could be listed and read.

02 — FILE ACCESS RESTRICTIONS
Tested missing files, unsupported extensions, absolute paths and attempts to navigate outside the approved directory.

03 — SEARCH INPUT VALIDATION
Checked empty and whitespace-only queries, along with expected phrase-search behavior.

04 — DOCUMENT SEARCH RESULTS
Tested case-insensitive phrase matching and handling of search errors.

05 — STRUCTURED RESULT VERIFICATION
Checked whether reported search findings agreed with independently calculated Python results.

06 — RESULT CONSISTENCY
Tested mismatched queries and document counts to confirm that inconsistent results were identified.

All 21 automated tests passed in both the original project and the separate starter template. I also completed a live agent run that demonstrated tool selection, execution and agreement with Python verification.

Testing Principle: Test the operations and verification rules independently, then confirm that the AI agent can use them in a live workflow.

Tool Calling in Practice

A live execution confirmed that the agent could select a registered document-search tool, execute the operation and return a structured result that agreed with independent Python verification.

```
AI and Python results agree: True
Verified AI result: query='electronic warranty' phrase_found=False documents_checked=2

Execution steps:
ToolCallItem
Tool selected: check_all_documents_impl
Arguments: {"query":"electronic warranty"}
ToolCallOutputItem
MessageOutputItem
```

The agent selected the document-checking tool and reported no match across two documents. Independent Python verification confirmed the structured result.

Designing for Reuse

Once the original agent was working and tested, I wanted to make the architecture easier to adapt without treating the document-search example as a universal solution.

I created a separate starter template with configurable agent naming, instructions and example search inputs. The underlying structure retained the tool-calling workflow, structured output and independent Python verification.

I also separated the verification functions so they could be tested with injected document-checking behavior, rather than requiring a live AI request for every test.

The starter package includes source files, sample documents, automated tests, setup instructions and a client customization checklist. The documentation identifies which components can be configured directly and which require changes to the tools, output schema, verification logic or tests.



I installed the dependencies in a separate environment, reran the 21 automated tests and completed another successful live agent execution.

DESIGN PRINCIPLE

Reuse the execution pattern, but make the workflow-specific assumptions visible and testable.

The Result

The project produced a working Python AI agent that can select document tools, perform controlled file operations and return structured search results for independent verification.

I also developed a separate starter template with configuration guidance, automated tests and a packaged ZIP for future customization.

THE COMPLETED PROJECT INCLUDES

- An AI agent built with the OpenAI Agents SDK
- Four registered tools for document discovery, reading and searching
- Restricted access to approved local text documents
- Validation for invalid file requests and search inputs
- Structured results containing the query, match status and document count
- Independent Python verification of reported search findings
- 21 automated tests covering operations, safeguards and result consistency
- Successful live tool-calling execution with matching Python verification
- A separately tested starter template with setup and customization documentation
- A packaged ZIP containing the starter source files, examples and tests

All 21 automated tests passed in both the original project and the separate starter template. Live executions also confirmed successful tool selection and agreement between the agent's structured results and Python verification.

Final architecture:

User Query → Agent Tool Selection → Python Operations → Structured Result → Independent Verification

The finished work is a tested portfolio prototype and reusable starting point, not a deployed client solution. Its document tools and verification rules provide a working example that can be adapted to other workflows with appropriate customization and testing.

What I Learned

The biggest lesson from this project was that giving an AI agent tools is only part of building a reliable workflow. The operations and results still need to be checked.

I started with a focused document-search example, then added structured output and independent Python verification. This gave me a practical way to check the agent's reported search query, match status and document count against independently calculated Python results.

Building and testing a separate starter template also showed me which parts of the architecture could be reused and which would need to change for another workflow.

The project reinforced several principles I want to carry into future systems work:

- **Define tool boundaries clearly:** Let the model select tools while Python controls what those tools can access and execute.
- **Make results verifiable:** Structured output gives independent code specific values to check.
- **Test the verification layer:** Mismatched queries and document counts helped confirm that inconsistencies could be identified.
- **Separate reuse from customization:** Preserve the execution pattern while adapting tools, schemas and verification rules to each workflow.
- **Use different forms of evidence:** Automated tests, live execution and separate starter testing each validate different aspects of the system.

The result changed how I think about AI agents: useful capabilities matter, but so do the boundaries and independent checks surrounding them.

Next Version

The next step would be adapting the starter architecture to a real business workflow rather than expanding the document-search demonstration for its own sake.

That could involve connecting the agent to a client's approved data sources, replacing the example document tools with workflow-specific operations and defining structured results that reflect the information the business actually needs.

The verification layer would also need to evolve. Each new workflow would require its own checks, based on which results can be independently confirmed and which decisions still require human review.

I would extend the automated tests alongside those changes, covering new tools, expected results, invalid requests and failure conditions before relying on the workflow in practice.

The goal would be to preserve the reusable execution pattern while making each implementation specific to the task, its data and its risks.

Tested Starter → Client-Specific Tools → Custom Verification → Workflow Validation

The foundation is in place. The next version would test how well that approach translates from a controlled demonstration into a useful business process.