VIKHRAM S
Department of Artificial Intelligence and Data Science | Saveetha Engineering College

AI Agents and Open Source Ecosystems for Intelligent Software Development

By Vikhram S, Advisor, Machine Learning Community – Tech Society, Saveetha Engineering College

Invited lecture delivered to the Department of Artificial Intelligence and Data Science, Saveetha Engineering College, 2024

Today's session has a slightly unusual arc: we will begin with a fairly abstract idea – the AI agent – and end somewhere very concrete: a Python library sitting on PyPI that anyone in the world can install with a single command. That journey, from AI agents to Python libraries, is the throughline of this talk. Along the way we will build a working agent with a custom tool, and then package that work into something reusable, testable, and shareable – because the real value of what you build in this field only compounds once other people can install and depend on it.

1. What is an AI agent?

At its simplest, an **agent** is a system that uses a large language model, or LLM, as a reasoning engine to decide which action to take next and what inputs that action needs, rather than following a fixed, pre-written sequence of steps. This is the key distinction from ordinary software: a traditional program's logic is written in advance by a developer, while an agent's logic is decided at run time by the model itself, based on the situation in front of it.

1.1 The architecture of an agent

The basic architecture of an AI agent Graph 1

Prompt Template	Agent	LLM	Tools	Memory
Supplies instructions that shape how the agent reasons	The orchestrator: receives a prompt, plans, and returns a response	Performs the planning and reasoning that decides the next action	External actions the agent can take, such as a search or a calculation	Stores and retrieves context across a conversation or task

Source: Author's synthesis, adapted for classroom presentation.

1.2 Why agents instead of plain retrieval?

Many of you will already be familiar with Retrieval-Augmented Generation, or RAG, where a system retrieves relevant documents and hands them to a model to answer a question. RAG and agents share several building blocks – prompts, models, embeddings, retrievers, vector stores – but they differ in what is allowed to happen after retrieval. A RAG pipeline is essentially a fixed path: retrieve, then generate. An agent, by contrast, can decide for itself whether to retrieve documents, call an external tool, ask a follow-up question, or take several of these steps in sequence, adapting its plan as it goes. In short, RAG answers questions from what it can retrieve; an agent decides what to do at all, and retrieval is only one of its available moves.

2. LangChain: a framework for building agents

LangChain is a framework for building AI agents that can interact with LLMs, and it is the foundation the rest of today's demonstration is built on. It organises the work of building an agent into a small number of reusable concepts. **Chains** handle the sequential execution of prompts and models, letting you compose several steps into a single pipeline. **Tools and plugins** give an agent the ability to make API calls, query a database, or run a search. **Memory** keeps a conversation context-aware, so an agent can refer back to what was said earlier. And the **agent** component itself handles dynamic decision-making – choosing which of the above to use, and when. LangChain works in both Python 3 and JavaScript, though today's examples are all in Python.

The core components of LangChain Graph 2

Agents	Tools	Chains	Memory
Orchestrate decision-making via tools and language models	Define the actions available to an agent, such as web search	Sequences that combine prompts, models, and output parsing	Retains context across a conversation or task

Source: Author's synthesis, adapted for classroom presentation.

3. LangGraph: adding structure and state

As agent workflows grow more complex – branching into different paths, looping back to retry a step, or needing to remember state across many turns – a plain chain of steps starts to strain. **LangGraph** is an extension of LangChain built specifically for this: it lets you construct AI workflows using graph-based state management. Three properties make it worth reaching for once a project outgrows a simple chain. It is **flexible**, supporting branching, loops, and multi-step workflows that a linear chain cannot express. It is **stateful**, maintaining memory across interactions rather than treating each step as isolated. And it is **scalable**, able to handle complex workflows efficiently as the number of steps and decision points grows. One practical note for anyone planning a project: LangGraph currently supports only Python 3, unlike LangChain's dual Python and JavaScript support.

3.1 The core components of LangGraph

The core components of LangGraph Graph 3

Graph Nodes	Edges	Memory & State	Integration with LangChain
Define the individual steps in a workflow	Define the logic that governs movement between steps	Stores past interactions across the whole workflow	Reuses LangChain's own LLMs and tools directly

Source: Author's synthesis, adapted for classroom presentation.

4. Building blocks: custom tools, chains, and agents

With the concepts in place, we can now build something concrete. A custom tool is created by following three steps: **inherit** from LangChain's `BaseTool` class; **implement** the tool's name, description, and its `_run` method, which contains the actual logic the tool performs; and then use it, as in today's example, which wraps a web search using `TavilySearchResults` so the agent can

pull in live information from the internet.

A chain, in its simplest form, is a straight path from input to output: it starts with a prompt, passes that prompt through a model for processing, and ends with a parsed output ready to use. LangChain's own documentation frames this as one point on a broader spectrum of autonomy in LLM applications – from fully human-driven code, through a single LLM call, to a multi-step chain, a router, and eventually fully agent-executed state machines and autonomous systems. A chain sits in the middle of that spectrum: more capable than a single LLM call, but still following a fixed sequence rather than deciding its own path.

Bringing a custom tool and a custom chain together produces a working agent. The process has three parts: create the agent using LangChain's `create_react_agent` function; integrate it by passing in the custom tool and chain you have built; and then demonstrate it by sending the agent a `HumanMessage` query and observing how it reasons through a response, deciding on its own whether and how to call the tool along the way.

5. From notebook to library: thinking in packages

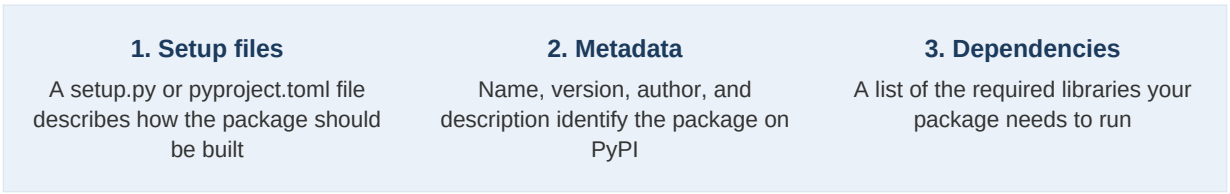
Once an agent works reliably in a notebook, the next natural step – and the one most student projects skip – is turning it into something reusable. A **Python package** is simply a collection of modules, that is, individual Python files, grouped together under a common structure. Packaging matters for three reasons: it allows for code reuse, so you are not copy-pasting the same agent logic into every new project; it enforces organisation, giving your code a predictable shape that others (and future you) can navigate; and it enables distribution, since a properly packaged project can be installed with a single `pip` command from the Python Package Index, or PyPI.

5.1 Project structure is key

Good packaging starts with good structure, well before you touch PyPI. Keep your code modular, splitting your agent logic, tools, and chains into clearly separated files rather than one long script. Document as you go: docstrings on every function and class, plus a README.md that explains what the package does and how to use it, save enormous time for anyone – including your future self – trying to understand the project later. And settle on a clear directory layout early, since restructuring a package after it has users is far more painful than planning it up front.

5.2 Packaging your library

Packaging your library Graph 4



Source: Author's synthesis, adapted for classroom presentation.

6. Testing, then publishing

Before anything goes out into the world, it needs to be tested. Write unit tests for your tools and chains using `pytest`. Install the package locally in editable mode with `pip install -e .`, which lets you test it exactly as an end user would, while still being able to edit the source directly. Finally, validate by running your test suite against the installed package itself, not just the raw source files, to catch any packaging mistakes before they reach a real user.

6.1 Publishing to PyPI

Publishing to PyPI, step by step Graph 5

1. Register	2. Build	3. Upload	4. Version
Create an account on PyPI, the Python Package Index	Run python setup.py sdist bdist_wheel to produce distributable files	Run twine upload dist/* to publish the package	Increment the version number for every new release, e.g. 0.1.0 to 0.1.1

Source: Author's synthesis, adapted for classroom presentation.

Once published, using the package is exactly as simple as using any other library you have installed before: run `pip install your-package-name`, then import your custom tools and chains into any new project as needed. That is really the point of this whole exercise – the effort of building and packaging the agent is paid once, and reused indefinitely afterward, by you and potentially by anyone else who installs it.

Closing note

The distance between a notebook experiment and a published, installable library is smaller than it looks, and the practices that close that distance – modular code, documentation, testing, and packaging – are the same ones that separate a one-off class project from a piece of software other people can actually build on. My hope is that some of you take an agent you build this semester and carry it the rest of the way: not just into a working demo, but onto PyPI, where it can outlive the course itself.

For queries, project proposals, ideas, or guidance related to this lecture, please write to vikhrams@saveetha.ac.in.