ChatGPT

Best Practices and Open-Source Frameworks for AI Agents and Agentic Workflows

Introduction

Multi-agent **AI systems** are emerging as powerful solutions for complex tasks, moving beyond single chatbot interactions to networks of specialized agents collaborating on a goal [1] [2]. In your use cases (from sentiment-based product search to market intelligence), you envision *multiple agents* each handling a sub-task – e.g. one agent parses user intent, another analyzes sentiment in reviews, a third queries purchase data, and so on. Such an **agentic workflow** can greatly improve customer experience and analytics by combining these specialized capabilities. However, building effective multi-agent systems requires careful design choices and the right framework. Below, we outline key best practices for designing agent workflows, and compare leading open-source frameworks (primarily Python-based) with their pros and cons.

Best Practices for Building Multi-Agent Workflows

Designing multiple AI agents to work in concert is challenging but manageable with the following best practices:

- **Define Clear Roles & Domains:** Give each agent a well-defined role, responsibility, and expertise area [3] [4]. For example, an "Review Analysis" agent focuses only on sentiment in text, while a "Database Query" agent handles purchase records. This modular approach ensures each agent can be developed and optimized independently (often by separate teams) and prevents overlap or conflict in reasoning [5]. A *coordinator* or router (which can be a simple rule-based dispatcher or another LLM agent) should direct tasks to the appropriate specialist agent [2] [6].

- **Keep Memory and Context Local:** Provide each agent only the context it needs rather than sharing a global memory of the entire conversation/task history [7] [8]. Limiting context prevents token overload and reduces confusion. For instance, the review-analysis agent doesn't need customer purchase history, and the purchase-analysis agent doesn't need raw review text – each should operate with its relevant slice of data. This improves efficiency and avoids swamping agents with irrelevant information [8].

- **Minimize Unnecessary LLM Calls:** Structure the workflow so that not every step relies on an LLM's reasoning if not needed. Some frameworks let you hard-code tool usage or use direct function calls, which can be more efficient. Over-relying on an LLM for every decision (e.g. deciding which tool to use at each step) adds latency and token cost [9]. For example, if the sequence of tasks is well-defined (like *extract intent → query DB → rank results*), consider a deterministic chain for those steps instead of dynamic agent negotiation, to reduce overhead [9] [10]. Use the *simplest pattern* (chain vs.

single agent vs. multi-agent) that meets your needs – don't introduce multiple agents unless the problem truly demands distinct specialties [11].

- **Guard Against Infinite Loops & Errors:** Multi-agent setups add complexity – agents might get stuck passing tasks to each other or repeating tool calls. Implement safeguards like **iteration limits, timeouts, or max dialogue turns** for agents [12] [13]. Also plan for tool/LLM failures: use try/except logic, fallback responses, or a default behavior if an agent fails to produce a result in N attempts [14]. For critical tasks, include a "timeout handler" agent or a human-in-the-loop step to catch endless loops or unclear handoffs.

- **Use Structured Outputs and Validation:** Aim for structured data exchange between agents (e.g. JSON schemas) instead of unpredictable free-form text. This makes parsing responses easier and more reliable, like calling APIs. Tools like **Pydantic-AI** enforce type schemas on LLM outputs for robust structure [15]. Structured inputs/outputs help agents verify each other's results (one agent can validate the JSON of another), reducing miscommunication.

- **Logging and Observability:** Enable detailed logging of each agent's decisions, prompts, tool calls, and outputs [16]. In a complex workflow, observability is vital for debugging and optimization. Use framework features or custom instrumentation to trace the chain-of-thought for each agent. Storing these traces helps identify why an agent made a certain decision or where an error occurred [17] [16]. Many frameworks integrate with monitoring tools (e.g. LangChain with LangSmith) or allow attaching callbacks for logging. Leverage these to monitor performance, latency, and token usage per agent.

- **Iterative Testing and Refinement:** Test each agent in isolation *and* in end-to-end integration. Start with simple scenarios to ensure agents produce correct outputs individually. Then simulate full queries (like your use case examples) to observe interactions. Expect to refine prompts and behaviors over time [18]. It's often useful to implement a small *sandbox or staging environment* where agents' actions (especially those that perform data writes or external calls) are either mocked or limited, to safely trial run the workflow. Automated regression tests on agent outputs can catch when a model update or prompt tweak breaks a previously working step [19] [18].

- **Security and Safety Checks:** If agents perform actions like database modifications, code execution, or sending communications, sandbox those operations and enforce strict permissions [20]. Use allow-lists for tools/APIs each agent can access [21]. This prevents a misbehaving agent from causing unintended harm. For example, the "Purchase Validation" agent should maybe have read-only access to purchase data if all it needs is to count repeat buyers. Also consider rate limits or cost limits for each agent (to catch a rogue agent spamming an API or consuming excessive tokens).

By following these practices, you can mitigate common failure modes (like agents looping or hallucinating tools) and build a **stable, maintainable multi-agent system** [22] [23]. Always remember to start simple and only scale up complexity as needed – many problems can be solved with a well-structured single-agent or chain before resorting to a full multi-agent design [11].

Open-Source Frameworks for AI Agents (Pros & Cons)

A number of open-source frameworks have emerged to help developers build agentic workflows in Python. These frameworks offer abstractions for defining agents, managing tools and memory, and orchestrating multi-step or multi-agent processes. Below we compare some of the leading options and their suitability, especially in the context of your use cases (customer experience, data analytics, etc.):

LangChain

LangChain is one of the most popular frameworks for LLM-powered applications, known for its extensive support for chains, tools, memory, and Retrieval-Augmented Generation (RAG) [24]. It provides a **highly modular toolkit** to connect LLMs with external data (databases, vector stores) and actions (APIs, Python functions). This makes it a go-to for building conversational assistants, document QA systems, and recommendation agents that need to integrate with knowledge bases or tools [25]. Mature corporations and startups alike have adopted LangChain due to its flexibility and large community support [26]. In your use cases, LangChain could be used to implement each "agent" as a sequence of tools (for example, using its built-in tools for search, vector DB queries, etc.) orchestrated by a central chain.

Pros: LangChain's strength is its rich ecosystem – it has ready-made integrations for many databases (SQL, Elastic, Pinecone, etc.), web scraping, Python execution, and more [27]. It also supports memory modules (short-term conversational memory or long-term via vector stores) and can manage prompt templates and chains easily. This means for tasks like sentiment-grounded product search, you can quickly plug in a sentiment analysis tool or a vector search over review embeddings using LangChain's components. Its abstraction can speed up development since many common patterns (like a QA chain with retrieval) are already implemented.

Cons: The flexibility comes at the cost of **complexity and overhead**. LangChain is "chain-first" and essentially uses a single orchestrator agent under the hood [9] [28]. True multi-agent support (agents chatting with each other) was not native and has been added in a hacky way. Each action often goes through an LLM reasoning step (the agent deciding the next tool via prompt) which *increases latency and token usage* [9]. An independent benchmark noted LangChain had significantly higher latency and token consumption in multi-step tasks compared to frameworks designed explicitly for multi-agent orchestration [29] [30]. Managing its many dependencies and frequent updates can also be cumbersome [31]. For large-scale production use, developers sometimes find LangChain abstractions "bloated" and prefer a more manual, optimized approach when exact control and efficiency are required [32].

Use Case Fit: LangChain is a solid starting point for prototyping due to its batteries-included approach. For example, a "Review-Grounded Product Search" could be built as a LangChain agent that first runs a custom search tool over a vector store of reviews, then calls a database query tool, etc., all within one chain. However, as complexity grows (many specialized agents, or need for concurrency), you might face maintainability issues. If you already are mandated to use LangChain, it will certainly handle your scenarios, but be prepared for careful prompt engineering to avoid unnecessary steps and for potential refactoring if performance becomes an issue [9].

Microsoft AutoGen

AutoGen (by Microsoft, open-sourced on GitHub) is a framework explicitly designed for *multi-agent communication* and workflow automation with LLMs. It enables you to spin up multiple agents (even multiple LLM instances with different personas or purposes) that can message each other, optionally with human oversight or tool use in the loop [33] [34]. AutoGen provides high-level constructs for agents that **communicate via a messaging loop**, making it easy to implement scenarios where agents brainstorm or cooperate on tasks. This flexibility makes it great for research and prototyping new agent behaviors [35] [33].

Pros: AutoGen shines in scenarios where agent behavior is complex or needs iterative refinement [35]. You can easily define, say, a "Questioner" agent and a "Solver" agent that talk back-and-forth to come up with a solution, or implement reflection strategies (one agent double-checks another's answer). It supports asynchronous communication and even human-in-the-loop proxies by design [36] [37]. Another advantage is a more *standardized structure*: it may generate a lot of boilerplate for you (hence the name AutoGen) so that you don't have to write glue code for every single step [38]. This can lower the barrier for developers without deep AI expertise to set up multi-agent workflows [39]. AutoGen's messaging paradigm is well-suited for open-ended tasks where agents need to discuss or negotiate (for example, having an "Evaluator" agent validate the output of a "Generator" agent).

Cons: Compared to some frameworks, AutoGen can be **less granular in customization**. It emphasizes Microsoft's ecosystem and patterns, which means it's great if your problem fits its mold, but less so if you need fine-grained control over every step [40]. It may not have as many pre-built tool integrations out-of-the-box as LangChain, so you might need to write custom code for certain API calls or data access. Also, because agents continuously communicate, there's a risk of higher token usage if they take many turns to converge on a result – careful prompt design is needed to keep them efficient. Some practitioners consider AutoGen more of a *research toolkit* and note that for production systems with strict requirements, additional engineering might be required to handle edge cases or ensure stability [41].

Use Case Fit: AutoGen could be very useful for a use case like *"Customer Lifetime Value Prediction"*, where multiple criteria must be met (sentiment check, purchase value, churn prediction) – you could have agents responsible for each criterion and let them collectively decide which customers are at risk. It's also well-suited if you want to prototype an agent that *self-improves* or *self-evaluates*, since you can have an agent generate an answer and another critique it in a loop. For straightforward query->tool->answer pipelines, AutoGen may be overkill, but for complex logic that might benefit from inter-agent dialogue, it's worth exploring.

CrewAI

CrewAI is a newer open-source framework centered on the concept of a "crew" of agents with different roles collaborating on tasks. It provides a high-level, declarative way to define each agent's role (e.g. Researcher, Developer, Validator, etc.) and their permitted tools or skills [42] [43]. The framework then handles running these agents as a team – passing messages, sharing intermediate results, and so on – to complete the objective. CrewAI is designed to **simplify multi-agent orchestration** by handling low-level details like message passing and state, so the developer can focus on the roles and logic.

Pros: CrewAI's architecture is *built from the ground up for multi-agent systems*. Task delegation and inter-agent communication are managed centrally by the framework, which leads to efficient execution with less

token overhead [44]. Tools (functions, data access, etc.) can be attached directly to agents, so agents call Python functions natively rather than always going through the LLM for tool use [45]. This yields faster and more reliable agent actions in many cases. CrewAI abstracts many complexities; for example, you might declare that Agent A can ask Agent B for help, or that Agent C can access a certain database, and the framework worries about the rest. This high-level approach makes it attractive for *production scenarios* where structured coordination and performance are important [44] [35]. In practice, CrewAI has been used for things like multi-agent content creation or decision support flows where each agent has a specific expertise (similar to your use cases) [42].

Cons: As a relatively young framework, CrewAI is still maturing. It may not have the breadth of community or documentation that LangChain enjoys, and some of its patterns are "opinionated" – meaning you might have to adapt your thinking to how CrewAI expects you to structure agents. According to one analysis, CrewAI's multi-agent flows are mostly linear or looping rather than complex branching; it doesn't natively provide a DAG-style execution plan or hierarchical controller agents out of the box [46]. This means if you need very fine-grained control over sequence (or parallelism), you might need to implement that logic around CrewAI. Also, debugging can be a bit challenging if many agents are chatting simultaneously, since the framework handles a lot under the hood [47]. There's limited user feedback so far (given its smaller user base), which might make troubleshooting harder. A Reddit discussion even suggested that early versions of CrewAI, while cool, had not yet proven their production robustness for complex enterprise use cases [41].

Use Case Fit: CrewAI's role-oriented design maps naturally to your described scenarios. For example, for *"Review-Grounded Product Search"*, you could define a crew with Agent1 = "Query Interpreter", Agent2 = "Review Miner", Agent3 = "Purchase Validator", Agent4 = "Ranker", each with functions to perform. CrewAI would allow them to share info (Agent1's extracted intent goes to Agent2, etc.) in a conversational way. If you aim to build a **production-grade** multi-agent pipeline, CrewAI's structured approach and performance optimizations (lower latency, fewer tokens) are appealing [48]. Just be mindful of its learning curve and be prepared to instrument it with logging for transparency. It's well-suited to scenarios needing tight teamwork between agents (the framework name "Crew" reflects that) – e.g. a customer support scenario where one agent fetches data and another formulates the answer collaboratively.

OpenAI *Swarm*

OpenAI Swarm is an experimental framework (open-sourced by OpenAI's Solution Engineering team) that explores lightweight orchestration of LLM agents [49]. Despite the name "Swarm", the initial versions of this framework implement a **single-agent control loop** that can plan and execute a sequence of tool calls or subtasks in natural language [50] [51]. Essentially, Swarm allows an agent to follow a *routine* (defined via a prompt with special instructions and tool docstrings) in an iterative manner. It's quite minimalist and focuses on ease-of-use for step-by-step task automation.

Pros: Swarm is **ergonomic and lightweight**. It doesn't impose a complex architecture; you write a system prompt that can include pseudo-code or high-level instructions, and the agent will iterate through them, using tools as needed [49] [52]. Because it treats tool usage via docstring parsing (the agent reads tool descriptions and decides when to call them), it's simpler to set up than a full multi-agent messaging system. In terms of performance, Swarm was designed to reduce overhead – it connects tools as native Python functions and only calls the LLM when necessary, much like CrewAI's approach [53]. This can mean faster execution and fewer tokens consumed for straightforward workflows. It's great for quick prototypes or scenarios where you want an LLM to **autonomously complete a multi-step task** (like "do A, then B, then

C") without hand-holding, but you don't need multiple agents talking to each other. The educational focus means it's relatively easy to understand and extend.

Cons: As noted in a comparison, Swarm currently does *not* support true agent-to-agent communication – it's effectively one agent executing a sequence [51]. You can think of it as closer to an automated chain executor. So if your goal is to have distinct agents with separate personas exchanging information, Swarm alone won't do that. You'd have to create a single prompt that encapsulates all roles, which can get messy. Its design (relying on natural language routines) is flexible, but also means **less structure** – it's up to the LLM to not mess up the plan. For complex logic, this could be harder to debug since the "plan" lives in the prompt. Also, being experimental, Swarm may lack some tooling around logging or error handling that more mature frameworks have. In summary, it trades off advanced features for simplicity.

Use Case Fit: If you want to prototype something like "*Quality Problem Detection & Alert*" quickly with one GPT-4 agent that iteratively: (1) fetches recent reviews, (2) analyzes sentiment, (3) checks purchase trends, (4) outputs an alert – Swarm could handle that in one continuous loop. It's ideal for **open-ended tool-using agents** where one LLM can manage the whole workflow step-by-step [54] [55]. However, if you require multiple distinct agents (as your scenarios outline), you might only use Swarm as a component or not at all. It's a good "light" option to be aware of, especially for quick experiments or smaller-scale automation tasks.

Atomic Agents & Pydantic-AI

These two are not the same project but are related in philosophy, so we discuss them together. **Atomic Agents** (open-source on GitHub) is a minimalist Python framework created as a reaction to the complexity of larger frameworks like LangChain or LangGraph [32] [56]. Its author emphasizes building AI agent workflows with **maintainable, production-quality code** rather than heavy abstractions. Atomic Agents focuses on using straightforward Python functions and classes to define agent logic, and encourages structured input/output (often via Pydantic models). **Pydantic-AI**, on the other hand, is a library that brings type validation and schema to LLM interactions – effectively, it uses Pydantic (a Python data validation library) to specify what the AI's output should look like (JSON with certain fields, for example) [15]. It can serve as a building block within custom frameworks or even within LangChain (LangChain now has some support for Pydantic output parsers as well).

Pros: The combination of Atomic Agents + PydanticAI yields a very **controlled and transparent development experience**. You get to dictate the workflow in code (like calling the OpenAI API directly or via a small wrapper) without magic under the hood. Developers with experience have found that large frameworks can add technical debt and obscure what's happening [57], whereas a minimal approach forces you to understand the chain of events [58]. Atomic Agents is essentially a thin organizational layer that lets you compose multi-step tasks or multi-agent interactions in a structured way, but it's not much more complex than writing your own sequence of API calls. This means debugging is easier – you can step through each function. By using Pydantic for I/O, you avoid brittle string parsing and get type-checked, reliable outputs (no more JSON parse errors because the LLM forgot a quote, for instance). This approach is **highly suitable for production** if you have the engineering resources, since it can result in cleaner, more maintainable code versus a tangle of framework callbacks [32] [56]. Another pro: switching out the LLM provider or model is easier because you haven't locked into a framework's way of calling models – for example, the Atomic Agents creator mentioned being able to swap OpenAI with other providers (Claude, etc.) with minimal changes due to using a simple underlying "Instructor" interface [59].

Cons: The downside of a minimal approach is you don't get a lot of pre-built functionality. There's no fancy UI or extensive library of tools – you'll be writing more code yourself. For instance, you'll need to implement vector database lookups or sentiment model calls by integrating the respective SDKs or APIs manually (though of course you can still use libraries like `faiss` or `transformers` directly). It's a trade-off between **control and convenience**. Teams lacking senior developers might prefer a higher-level framework to get things working quickly. Additionally, while Pydantic-AI enforces structure, it introduces its own syntax/boilerplate for describing prompts and schemas. There's a learning curve to define your Pydantic models and ensure the LLM adheres to them (often by providing few-shot examples of the JSON format in the prompt). If your use case involves a lot of unstructured creativity (like generating marketing copy), rigid schemas could even be a limitation. Atomic Agents itself is relatively new and community support is small (though interest is growing). It also currently doesn't have built-in observability tooling – you'd need to integrate your own logging/monitoring, as one user pointed out in a forum [60].

Use Case Fit: If you prioritize **maintainability and clarity** – for example, you want to build a system that will be handed off to other developers or maintained over years – a lean framework like Atomic Agents is attractive. For your multi-agent scenarios, you could certainly implement them with straightforward Python classes/functions representing each agent's step, using OpenAI API calls and some custom coordinator logic. It might take a bit more initial coding than using LangChain or CrewAI, but you'll have full understanding of how data flows. Pydantic-AI is especially useful in the analytics-heavy use cases (sentiment scoring, lifetime value prediction) because you can enforce that the agent's output contains specific fields (e.g. a product ID, a score, an explanation) and thus reliably pass data from one agent's output to another's input. In short, this route is best if you have strong Python skills and want full control – you effectively *become* the framework. It's less about pros/cons of one library and more about adopting a philosophy of simplicity. As one veteran developer put it, many fancy frameworks "only hold you back when you want to build real stuff for enterprise" [32] – if that resonates with you, consider starting with a minimal stack and adding libraries only as needed.

Other Noteworthy Frameworks

Beyond the above, a few other open-source tools/frameworks might be relevant:

- **Hugging Face Transformers Agents:** This offering from Hugging Face allows an LLM (like GPT-4 or OpenAI's models) to act as a controller that can call other HuggingFace models as tools. For example, a GPT-4 can decide to invoke an image classifier or a translation model from HuggingFace's hub to complete a task. It's useful for *multi-modal or specialized tasks* where you want to leverage the vast array of models available (vision, audio, etc.) through a unified agent interface [61] [62]. In practice, you describe the tools (models) available, and the LLM's prompt will include those descriptions so it can use them. This might be less directly useful for your current use cases (which are mostly text and database-focused), but if you ever wanted an agent that, say, *analyzes an image and then responds based on that along with text analysis*, this framework could be handy. It's relatively new, and primarily geared toward enabling complex **transformer model orchestration** without writing glue code.

- **Semantic Kernel (Microsoft):** Semantic Kernel is not an agent framework per se, but an SDK for integrating AI into applications (supports Python, C#, etc.). It has concepts of *skills* (functions) and a planner that can use an LLM to decide which skill to invoke. Essentially, it's Microsoft's approach to tool-using AI agents within a software app. Semantic Kernel shines in enterprise scenarios where

you need robust integration with existing software and want strong typing, security, and multi-language support [63] [64]. If your project eventually needs to be part of a larger production system (with .NET components, for example), SK could be a bridge between the Python prototypes and a scalable solution. It requires more software engineering to use effectively, and it's a bit lower-level than something like LangChain. For now, if you're working primarily in Python with a focus on data analysis, you might not need SK, but it's good to be aware of if Microsoft's ecosystem is involved in your environment.

- **Rasa (Open Source Conversational AI):** Rasa is a well-established framework for dialog agents (chatbots) with a focus on intent recognition, entity extraction, and dialogue management. It isn't designed around LLMs (though it can incorporate them); instead it uses machine-learned or rule-based NLU pipelines. We mention it because it's a powerful open-source tool for customer experience bots – for example, if you wanted to build a conventional chatbot that guides a user through product returns or FAQs, Rasa would be a top choice. It supports **intent classification and slot-filling** very well [65]. However, for your described use cases that involve dynamic analysis (sentiment, data mining), Rasa might feel too rigid or require training data to cover the varied queries. If your project evolves toward more interactive dialogue systems (and especially if you need it on-prem for privacy), Rasa could complement the LLM agents by handling predictable conversations and deferring to an LLM agent for the generative parts.

- **LangFlow / Flowise (Low-Code Visual Builders):** These are UI-driven tools to construct LLM workflows by dragging and dropping components. LangFlow, for instance, provides a web interface to build LangChain pipelines visually [66]. They can be useful for quickly mocking up a multi-step agent flow and for demonstration purposes. The advantage is you can see the graph of operations and how data passes, which might help non-developers understand the system. The disadvantage is that complex logic might be harder to implement in a purely visual tool, and it may not scale well to production without refactoring to code. Nonetheless, if you want to prototype a sentiment-aware recommendation flow and show it to stakeholders, a visual builder could speed that up. Just treat it as a prototyping aid; serious development and maintenance will still require code (thankfully, these tools often export the flow to a Python script).

Conclusion and Recommendations

Choosing the right framework (or no framework) depends on the specifics of your project's needs and constraints. In summary:

- **If you value quick development and a rich set of integrations**, LangChain is a strong starting point, especially for connecting to data sources (reviews, purchase databases, vector indexes) out-of-the-box. Just be wary of its performance overhead for multi-agent patterns [29] [9], and consider simplifying or switching if it becomes a bottleneck.

- **If your focus is on multi-agent collaboration and you want a higher-level orchestration**, frameworks like Microsoft AutoGen or CrewAI are compelling. AutoGen offers flexible agent dialogues suitable for research prototypes [33], whereas CrewAI provides a structured, role-based approach that can be more efficient in production [48]. These might require some adaptation (and careful debugging of agent interactions), but they align well with building complex agent teams for customer intelligence tasks.

- **For lightweight needs or maximum control**, OpenAI's Swarm and custom frameworks (Atomic Agents with Pydantic schemas) give you **transparency and speed**. They require more manual effort but can result in cleaner solutions tailored to your exact workflow. Experienced developers often prefer this route for maintainability [67] [32], especially once the prototyping phase is over and stability becomes key.

In practice, many teams use a combination: for example, using LangChain during initial prototyping to leverage existing tools, then gradually refactoring critical paths into a more minimal custom pipeline for production [68]. There's no one-size-fits-all – each framework has trade-offs in flexibility, abstraction, performance, and community support. It's worth experimenting with a couple of different approaches on a small slice of your use case (say, the sentiment analysis part) to see which feels most comfortable and effective. By keeping the best practices in mind – clear agent boundaries, robust error handling, and careful tool usage – you'll be able to succeed with whichever framework (or combination) you choose, and deliver an AI multi-agent system that truly enhances customer experience and analytics.

Sources: The insights above are drawn from recent benchmarks and reviews of agent frameworks [29] [48], best-practice guides for multi-agent system design [69] [11], and firsthand developer experiences transitioning between frameworks [32] [56]. Each cited source provides additional detail on specific points, and I recommend exploring them to deepen your understanding of how these frameworks perform and are applied in real-world scenarios. Good luck with your multi-agent system project!

---

[1] [7] 8 Best Practices for Building Multi-Agent Systems in AI | by Lekha Priya | Medium
https://lekha-bhan88.medium.com/best-practices-for-building-multi-agent-systems-in-ai-3006bf2dd1d6

[2] [3] [4] [5] [6] [8] [10] [11] [12] [13] [14] [16] [17] [18] [19] [20] [21] [22] [69] Agent system design patterns | Databricks on AWS
https://docs.databricks.com/aws/en/generative-ai/guide/agent-system-design-patterns

[9] [24] [28] [29] [30] [33] [34] [35] [36] [37] [42] [43] [44] [45] [46] [47] [48] [50] [51] [52] [53] [55] Top 5 Open-Source Agentic Frameworks
https://research.aimultiple.com/agentic-frameworks/

[15] [23] [32] [41] [56] [57] [58] [59] [60] [67] [68] What is the best AI agent framework in Python : r/AI_Agents
https://www.reddit.com/r/AI_Agents/comments/1hqdo2z/what_is_the_best_ai_agent_framework_in_python/

[25] [26] [27] [31] [38] [39] [40] [61] [62] [63] [64] [65] [66] Top 9 AI Agent Frameworks as of November 2025 | Shakudo
https://www.shakudo.io/blog/top-9-ai-agent-frameworks

[49] openai/swarm: Educational framework exploring ergonomic ... - GitHub
https://github.com/openai/swarm

[54] Exploring OpenAI's Swarm: An experimental framework for multi ...
https://medium.com/@michael_79773/exploring-openais-swarm-an-experimental-framework-for-multi-agent-systems-5ba09964ca18