ChatGPT

CoFound.ai Multi-Agent System Integration Report

Introduction and Background

CoFound.ai is developing an AI-driven platform that leverages multiple specialized agents (e.g. Project Manager, Developer, Tester, Reviewer) to collaboratively automate complex tasks like software development. Harnessing **multi-agent systems** can dramatically enhance problem-solving by dividing work among expert agents and enabling parallel, interactive reasoning [1] [2]. However, realizing such benefits requires robust protocols for agent communication and coordination, as well as powerful frameworks to orchestrate agents' behavior. This report provides a comprehensive analysis of emerging **agent communication protocols** – notably the Agent Communication Protocol (ACP) and Model Context Protocol (MCP) – and the broader **AGNTCY** initiative, alongside leading **multi-agent frameworks** (CrewAI, AutoGen, and ChatDev). We compare their architectures, strengths, and limitations, and evaluate how they can integrate into CoFound.ai's architecture. We also outline best practices for building scalable, maintainable, and interoperable multi-agent systems, supported by high-level and component-level architectural diagrams and comparative tables for key criteria.

By surveying the state-of-the-art, this report aims to inform CoFound.ai's technical strategy – guiding how to embed standard protocols and frameworks into its design to ensure the resulting system is robust, extensible, and aligned with emerging industry standards. All analysis is grounded in an examination of CoFound.ai's current architecture and technology decisions, ensuring recommendations are tailored to our existing codebase and vision.

Agent Communication Protocols: ACP vs MCP vs AGNTCY

Model Context Protocol (MCP): The Model Context Protocol, introduced by Anthropic in 2023, defines a standardized way to enrich AI models (typically large language models) with external context [3]. MCP allows a model or agent to query external data sources or even other agents as *tools* to incorporate additional information into its prompt or state. In essence, MCP focuses on **context injection** and **state management** rather than direct inter-agent messaging [4] [5]. It provides structure for maintaining conversation state across multiple models or services and for invoking tools in a uniform manner [6]. For example, an MCP server might register data sources (APIs, databases) or agent services; an agent (as an MCP client) can then dynamically discover and call these to augment its knowledge. MCP ensures that when an AI model uses external information, the **context is handled consistently** and state (conversation history, tool outputs) is preserved [5]. However, MCP was not originally designed for peer-to-peer agent communication – it treats other agents as contextual resources (similar to how an LLM calls an API) rather than as collaborators [7] [8]. This leads to certain **limitations** in multi-agent orchestration: (1) No dedicated inter-agent message protocol (agents communicate via tool-calls only), (2) limited interoperability when coordinating multiple independent agents (beyond single-agent with tools), and (3) lack of native streaming support for real-time interactions [8]. In summary, MCP is ideal for **enhancing a single agent**

with external data or tools, especially when the model or data source is outside the developer's direct control [9] [10]. It is less suited for enabling rich collaboration between autonomous agents at scale.

Agent Communication Protocol (ACP): The Agent Communication Protocol is a newer standard (spearheaded by IBM's BeeAI and the Linux Foundation) that **builds upon MCP to enable direct agent-to-agent collaboration** [11] [12]. Whereas MCP focuses on providing context to one model, **ACP focuses on communication between multiple autonomous agents** [12]. ACP defines a common format and **RESTful interface** for agents to interact, including endpoints for invoking an agent's capabilities, exchanging messages, sharing stateful context (via "threads"), and managing multi-agent task workflows [13] [14]. By standardizing how agents **send messages, requests, and results** to each other, ACP aims to eliminate ad-hoc integration code and incompatibilities between agents from different vendors [15] [16]. Notably, ACP supports **stateful conversations** (allowing agents to maintain context over a series of messages) or stateless interactions as needed [17]. It prefers lightweight protocols like JSON-RPC or simple REST for message exchange, and is exploring full-duplex channels (WebSockets or gRPC) for streaming data [18]. Key features include support for **cancellation and timeouts** (ensuring one agent can halt a request to another), and a clear distinction of roles (agent providers vs. orchestration servers) [18]. In effect, ACP provides the "grammar" for a multi-agent conversation – a standard way to ask another agent to do something and get a result – independent of any particular framework. Early documentation highlights several architecturally significant topics under consideration, such as how to manage agent state, how to perform **streaming exchanges**, and how to register agent capabilities for discovery [18] [19]. By addressing inter-agent messaging head-on, ACP "goes one step further" than MCP in enabling **agents to collaborate and share resources at scale** [20] [21]. It is currently in alpha, with an open invitation for community input [22]. For CoFound.ai, ACP offers a path to make our agents interoperable with external agent services and with future industry tooling – avoiding vendor lock-in to any single framework [15] [16]. While ACP is still evolving, its potential to **standardize multi-agent workflows** and improve scalability and flexibility is widely recognized [23] [24].

AGNTCY (Open Agent Ecosystem): AGNTCY (pronounced "agency") is not a single protocol but an ambitious **open-source collective and suite of standards** aimed at creating an "Internet of Agents" (IoA) [25] [26]. Launched in 2025 and backed by organizations like Cisco, LangChain, and others, AGNTCY's vision is to enable **open, interoperable agent systems** across platforms and organizations – analogous to how internet protocols (TCP/IP, HTTP, DNS) enabled interoperable networks [25] [27]. In the AGNTCY stack, ACP plays a role at the "application layer" (defining how agents invoke and interact with each other's services) [28]. But AGNTCY goes further by defining additional layers and components: for example, the **Agent Gateway Protocol (AGP)** is a transport layer built on gRPC for efficient, secure message delivery (supporting request-response, streaming, publish/subscribe, etc.) [29] [30]. An **Agent Directory** service acts like DNS for agents – allowing agents to register and discover each other by capabilities, using standardized **Agent Metadata (OASF)** descriptors [25] [31]. There are also reference implementations like an **Agent Workflow Server** to deploy and run multi-agent workflows with these standards [32]. In summary, AGNTCY aims to provide the **full infrastructure** needed for agents to find each other, communicate, and collaborate securely in an open network [25] [33]. This includes security (encryption, authentication, potentially even quantum-safe cryptography at the messaging layer) [34] and governance (evaluating and monitoring agent performance). **Figure 1** illustrates this ecosystem: agents implement ACP (and potentially other peer-to-peer protocols like FIPA-ACL or A2A) at the syntactic level, an I/O mapping layer semantically translates data, Agent Gateways route messages via AGP, and Agent Directories allow discovery of agents by their published schemas and capabilities [26] [35].

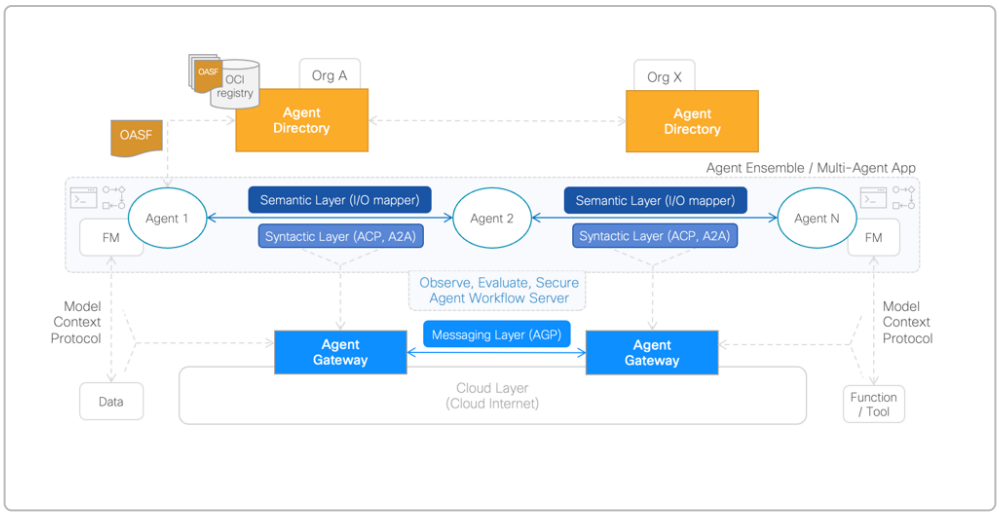

Figure 1: The "Internet of Agents" architecture envisioned by AGNTCY. Multiple agents (Agent 1…N) communicate via a syntactic layer (standardized protocols like ACP for agent-to-agent messages) and a semantic layer (I/O mappers for translating data formats). An Agent Gateway (blue) handles network transport using AGP (gRPC-based messaging), enabling reliable, low-latency communication. Agents can register with an Agent Directory (orange, top) by publishing an Agent Manifest in the Open Agentic Schema Framework (OASF) format, analogous to a service registry. This allows agents from different organizations (Org A, Org X) to discover and collaborate. MCP (Model Context Protocol) is shown feeding external data or model context into agents on the sides, while tools/functions can also be invoked. This ecosystem of standards aims to enable loosely coupled, scalable multi-agent systems [35] [30] .

In practical terms, AGNTCY's work means CoFound.ai could eventually **expose its agents to a broader network**, or incorporate third-party agents, with minimal custom integration – if all adhere to these standards. For example, a CoFound.ai "Developer" agent could be discovered and invoked by an external orchestration system via the standard ACP interface; likewise, CoFound.ai could compose an internal workflow that calls an external agent (say, a design AI) if that agent is listed in a directory and speaks ACP. While AGNTCY's components (ACP, AGP, directory, etc.) are still maturing, they represent **best-practice open standards** that CoFound.ai should track and gradually adopt to future-proof our platform. Near-term, focusing on ACP compliance (for agent APIs) and MCP compliance (for external tool/context integration) will align CoFound.ai with this emerging ecosystem.

Comparative Summary: Table 1 summarizes these protocols and initiatives. In short, **MCP** enriches single agents with external context (treating other resources as tools) [3] , **ACP** enables multi-agent **peer-to-peer collaboration** with standardized messaging [36] , and **AGNTCY** provides an entire interoperability stack (of which ACP is one part) for an open agent ecosystem. CoFound.ai's architecture can benefit from all three: using MCP patterns to integrate knowledge bases or models we don't control, adopting ACP for our agents to talk to each other (and to external agents) in a uniform way, and leveraging AGNTCY components (like directories or schema standards) as they become available to plug into the wider "internet of agents."

Aspect	Model Context Protocol (MCP)	Agent Communication Protocol (ACP)	AGNTCY Initiative
Primary Purpose	Provide external **context/data** to an AI model or agent (standardized tool calls) [3] .	Enable **agent-to-agent communication** and collaboration at scale (standard message interface) [36] .	Create an **open ecosystem** ("Internet of Agents") with standards for discovery, messaging, and interoperability [25] [26] .
Scope	Single-agent focus (context & state management when using tools or multiple models) [5] .	Multi-agent systems (interaction protocols, agent workflow invocation, thread-based stateful dialogs) [13] [14] .	Multi-agent networks (suite of protocols: ACP, AGP transport, agent directories, schema, security, etc.) [29] [37] .
Key Features	- Structured **conversation state** across tools/agents [5] - **Context injection** as "tools" for LLMs [3] - Discover & call resources via MCP server [38] .	- Standard **REST/JSON-RPC** endpoints for agents [39] - **Threads** for multi-turn state [14] - Supports **streaming** responses and cancellations [40] [41] .	- **Agent Directory** (registration & lookup by capabilities) [31] - **Agent Gateway** (gRPC messaging with streaming, pub/sub) [29] - **Agent schema** standard (OASF) [42] - Reference **workflow server** for orchestration [32] .
Maturity (2025)	Early adoption: backed by Anthropic, some open-source traction; acts as an emerging standard for tool-augmented LLMs [43] .	Alpha stage: defined by BeeAI (IBM), seeking community input; reference implementations under development [22] .	nascent: AGNTCY launched 2025 with multiple companies; prototypes of ACP/AGP available [44] [45] , broader components in progress.
Use Case Fit	*Controlled model needs uncontrolled data/tools*: e.g. LLM agent pulls real-time info from a web service via MCP call [3] .	*Multiple autonomous agents need to cooperate*: e.g. a planning agent asks a solver agent for help, exchanging messages as peers [36] .	*Cross-platform agent ecosystems*: e.g. a marketplace where any agent can find others (like microservices), with full interoperability.
Relevance to CoFound.ai	Integrate **contextual data** easily (e.g. codebase knowledge via an MCP server acting as memory) without custom APIs; treat external AI services as MCP tools.	Make CoFound's agents **modular and replaceable** – any ACP-compliant agent can plug in. Enables scaling agents as independent services (microservice style) [46] [47] .	Long-term direction for **interoperability** – CoFound.ai could join the IoA, registering its agents in directories and using AGNTCY infra for scalability and security.

Table 1: Comparison of MCP, ACP, and AGNTCY.

Multi-Agent Frameworks: CrewAI, AutoGen, and ChatDev

Building a multi-agent system from scratch is non-trivial; frameworks have emerged to provide reusable architectures and abstractions for agent orchestration. We analyze three prominent frameworks – **CrewAI**, **AutoGen**, and **ChatDev** – which represent different philosophies for multi-agent collaboration. We examine their architecture and design principles, strengths and limitations, and how applicable they are to CoFound.ai's needs.

CrewAI – Role-Oriented Multi-Agent Orchestration

Architecture: CrewAI is an open-source Python framework (created by João Moura) for orchestrating a "**crew**" of AI agents collaborating on tasks [48]. It follows a **role-based architecture**: each agent in a crew is assigned a distinct role, goal, and (optionally) a backstory, which together define the agent's behavior and perspective [49] [50]. Agents operate as a team – analogous to a work crew – and can autonomously **delegate tasks to each other and ask questions** of one another during execution [51]. Under the hood, CrewAI provides a structured framework with several core abstractions [52] [53]:

- **Agents:** the fundamental actors, each encapsulating an LLM (or other AI model), some memory, and tool integrations. Agents are highly configurable; one can plug in any language model (OpenAI GPT-4, local models, etc.) and assign custom tools or functions to an agent [54] [55]. Agents have internal reasoning loops (using LLM outputs) to decide on actions and can invoke tools or communicate with peers accordingly [2] [56].
- **Tools:** Functions or APIs that agents can call to extend their capabilities beyond text generation [57]. CrewAI comes with a **Toolkit** of ready-made tools (for web search, code execution, GitHub queries, etc.) and also supports LangChain's tool integrations [58] [59]. This allows agents to perform actions like retrieving information (Retrieval-Augmented Generation tools) or running code. All tools include error handling and caching by default [60].
- **Tasks:** Units of work to be accomplished. A task in CrewAI has attributes like a description, an assigned agent (or role), and an expected output [61] [62]. Multiple agents can collaborate on one task, and tasks can be connected in sequences or hierarchies. Notably, tasks can be run asynchronously, enabling concurrency in the workflow [63] [64]. The framework allows outputs of one task to feed into others as context, enabling complex multi-step workflows (e.g. research task results become input to a writing task) [65].
- **Processes:** High-level orchestration logic that defines *how tasks are assigned and executed among the agents*. CrewAI provides built-in process strategies: **Sequential**, **Hierarchical**, and (planned) **Consensual** [66] [67]. In Sequential process, tasks are executed one after another in a fixed order (like a pipeline), passing along context [67]. In Hierarchical process, a special "manager" agent is spawned to oversee the others – it assigns tasks, monitors outputs, and can dynamically allocate work (simulating a managerial hierarchy) [68]. A future Consensual process aims to allow agents to vote or agree on task decisions democratically [69]. These processes give CrewAI flexibility to mimic different organizational structures (from flat workflows to managed teams).
- **Crews:** A Crew is essentially a configured instance of a multi-agent team, combining a set of agents with a set of tasks and a chosen process (plus settings for memory, verbosity, etc.) [70] [71]. The Crew is what the developer creates to solve a problem: for example, a crew might consist of a "Data Scientist" agent and a "Customer Support" agent working together on tasks like data collection and summarization [72]. When the crew is started, CrewAI handles the orchestration: running the process (sequential or hierarchical), routing task outputs to the right places, and managing inter-agent

communications as defined by the framework. Each crew produces a final outcome (and intermediate artifacts per task).

In summary, CrewAI's architecture modularizes the multi-agent system into clear components (agents, tools, tasks, processes), making it straightforward to define a scenario of collaborating agents and then execute it. The design is explicitly inspired by real-world team workflows, with an emphasis on **role specialization and delegation** [2] [73]. This approach aligns well with CoFound.ai's concept of agents with distinct roles (PM, Developer, Tester, etc.), and indeed CrewAI was built to **automate multi-agent workflows** of this nature [74].

Strengths: CrewAI's role-oriented, modular design brings several advantages: (1) **Clarity and ease of use** – developers can think in terms of real-world team roles and tasks, which lowers the conceptual barrier to setting up a multi-agent system [48] [2]. The framework handles the low-level message passing and tool invocation, so one can quickly get a prototype running by specifying roles, tools, and tasks. IBM's analysis noted that CrewAI "provides a simpler way to orchestrate agent interactions" via high-level attributes, whereas other frameworks sometimes require writing more custom code to manage the agent loop [75]. (2) **Built-in task coordination patterns** – the availability of sequential vs. hierarchical processes is powerful. For instance, the hierarchical mode (with a manager agent) can tackle complex projects where oversight is needed, without the developer explicitly coding that orchestration logic (CrewAI generates a manager agent using a manager-specific LLM prompt) [76]. This showcases an emerging best practice: using one agent to organize others (meta-reasoning), which CrewAI supports out-of-the-box. (3) **Flexible tooling and memory** – by leveraging LangChain integration, CrewAI can use a wide array of existing tools (search, calculators, code runners), and it inherently supports memory caching in tools [60]. This modular tool integration means agents can do more than talk – they can act on the world, which is critical for real tasks. (4) **Combination of conversational and procedural paradigms** – perhaps CrewAI's biggest strength is that it marries the free-form **conversational flexibility** of frameworks like AutoGen with the structured **workflow approach** of frameworks like ChatDev [77] [78]. As noted, "crewAI combines the flexibility of AutoGen's conversational agents with the structured processes approach of ChatDev" [77]. In practice, this means agents in CrewAI can intermix open-ended dialogue (asking each other questions, brainstorming) with goal-directed task execution under a defined process. This hybrid capability is valuable for CoFound.ai: our agents might need to have free-form discussions (e.g. brainstorming a software design) but within a controlled overall sequence (e.g. design phase → coding phase → testing phase). CrewAI inherently supports that pattern. Finally, CrewAI is reasonably **lightweight and open** – it is available as open source (MIT license) and is being adopted in the community (with example projects for content creation, stock analysis, etc. emerging) [79] [80]. This means CoFound.ai could leverage community-contributed tools or improvements, and would not be locked into a proprietary system.

Limitations: Despite its promise, CrewAI has some limitations. (1) **Maturity and Scale:** It is a relatively new project (circa 2023) largely driven by a single author, which may raise questions about its maturity for production. While the design is solid, features like the "Consensual" process are not yet implemented [69], and enterprise features (robust error recovery, distributed deployment support) may be limited. Our internal assessment noted that CrewAI offers "limited orchestration, less flexibility" compared to more code-centric approaches, and we rated it slightly lower in suitability (score 7/10) than a fully custom graph-based approach. For example, CrewAI currently does not natively include a **visual workflow builder or GUI**, so designing complex agent flows might become cumbersome if done purely in code. (2) **Orchestration control:** The high-level abstraction can be a double-edged sword – while simple to get started, it might be **less flexible for highly custom workflows** outside the provided process types. If CoFound.ai needed a very specific scheduling of agent interactions (beyond sequential or hierarchical), we might have to extend or

modify the CrewAI core, which introduces overhead. (3) **Performance and concurrency:** CrewAI's default operation, especially in sequential mode, may not fully exploit parallelism (aside from the asynchronous task option). If many agents or tasks are involved, orchestrating them in Python could become a bottleneck. There is no built-in mechanism for scaling agents to multiple machines or processes (though one could manually distribute agents). (4) **Memory handling** in CrewAI is mostly left to each agent's LLM context or external vector stores that the developer integrates; there is no globally shared long-term memory module provided out-of-the-box (beyond the RAG tools) – meaning CoFound.ai would need to implement a persistent memory if agents should remember information across runs or crews. (5) **Lack of native code execution support:** One particular gap noted in comparison to AutoGen is that CrewAI doesn't yet have a *built-in* way to execute code that agents write [81] [82]. For a software development use-case, this is significant – an agent can write code, but testing that code might require hooking in a custom tool or additional programming. (In contrast, AutoGen has facilities to execute generated code in a sandbox automatically [83].) CrewAI can be extended to do this (and indeed one could integrate a code execution tool), but it requires extra effort. (6) Finally, **documentation and community** are still ramping up – while an official docs site exists and IBM/others have written tutorials, the user community is smaller than for frameworks backed by big tech companies. This means troubleshooting and finding examples might be harder.

Applicability to CoFound.ai: CrewAI's paradigm closely matches CoFound.ai's envisioned architecture of specialized agents working together under an orchestrator. If we were starting fresh, CrewAI would be a strong candidate to build upon. However, CoFound.ai's prototype already uses a **LangGraph-based orchestration** (a graph-driven workflow engine) which we chose for its explicit control flow design and higher flexibility (we scored LangGraph 9/10 in our framework evaluation). CrewAI, being more opinionated, scored 7/10 mainly due to that lower flexibility. Thus, a likely strategy is to **mimic CrewAI's strengths within our existing architecture**: adopt the role-based agent definitions, delegation behaviors, and perhaps implement a hierarchical manager-agent concept in our system. In fact, CoFound.ai's agents (PM, Dev, Tester, etc.) already mirror CrewAI's role specializations. We can also reuse ideas from CrewAI's toolkit – for instance, integrating similar search or GitHub tools for our Developer agent. If feasible, we could even incorporate CrewAI as a library **within** our LangGraph orchestration: e.g. use a CrewAI "crew" for certain sub-tasks (like a coding subtask where a Developer and Tester agent pair uses CrewAI's delegation to write and test code). However, mixing frameworks can introduce complexity. Another approach is to continue with LangGraph as primary orchestrator, but ensure our design remains **CrewAI-compatible** – meaning if later we wanted to switch to or incorporate CrewAI, our agents and tools would align (since both use Python and LangChain integrations, porting is possible). In summary, CrewAI offers a valuable reference architecture. We should incorporate its proven patterns (modularity, role separation, optional manager agent, asynchronous tasks) into CoFound.ai's implementation. Direct use of CrewAI might not be necessary now, but keeping an eye on its evolution is prudent. If CrewAI matures to offer visual flow design or better scalability, we could consider adopting it in parts of our system to accelerate development.

AutoGen – High-Level Multi-Agent Conversations

Architecture: AutoGen, developed by Microsoft, is a framework to facilitate **multi-agent conversations** and collaboration in a generic, extensible way [84]. Its core idea is to provide a **high-level abstraction for LLM-based agents to converse and coordinate**, minimizing the amount of "boilerplate" code developers need to write to get agents to work together [84] [85]. AutoGen can be seen as a sophisticated orchestration library that treats *communication* as the first-class primitive: rather than structuring around tasks or

processes, it focuses on enabling agents (which could be AI or human) to send messages to each other in flexible patterns [86] [87].

In AutoGen, developers define **agents** and their roles (similar to other frameworks), but the interactions are managed through a conversation loop. Agents can be of different types – e.g. an LLM-backed agent, a Python function agent, or even a human proxy – since AutoGen allows integration of **tools and humans into the agent ecosystem** as needed [86]. Once agents are set up, AutoGen provides various **design patterns** for their interaction. According to Microsoft's documentation, these include: *one-to-one conversations*, *group chat*, *cascaded reasoning*, *concurrent agents*, *debate*, *reflection*, etc., as found in their design patterns library [88] [89]. For instance, a **group chat** pattern might involve multiple expert agents sharing information freely to solve a problem, whereas a **handoff** pattern might have one agent delegate a subtask to another and wait for the result [90] [89]. AutoGen doesn't rigidly predefine roles (like "Manager vs Worker"); instead it gives a flexible canvas where the conversation structure emerges from how agents are configured to respond to certain prompts or messages.

A notable aspect of AutoGen is its emphasis on **tool use and code execution within the conversation**. It provides built-in capabilities for agents to execute code (especially Python) that they generate, and share the results back into the conversation [83]. This is extremely useful for tasks like coding or math, where an agent can "try out" its solution and then correct itself. AutoGen also comes with **logging and debugging facilities** to trace multi-agent dialogues, which helps developers refine prompts and fix issues in the agent interaction logic [91]. In essence, AutoGen acts as a sophisticated conductor: the developer specifies the agents and gives an initial prompt or goal, and AutoGen ensures the conversation flows according to the chosen pattern, with facilities to incorporate outside actions (like running code) and to monitor the process.

Strengths: AutoGen's strengths lie in its **flexibility and developer-friendly abstractions**. (1) It excels at enabling **dynamic, open-ended collaboration** among agents [91]. Because it is not tied to a rigid workflow, agents can flexibly exchange messages, making it suitable for complex problem domains where the solution path is not known in advance (e.g. brainstorming research problems, debugging code through back-and-forth dialogue). (2) AutoGen supports advanced interaction patterns like **self-reflection and step-by-step reasoning** natively. A developer can enable a "reflection" pattern where an agent critiques its own or others' outputs, improving robustness. This aligns with recent research showing that multi-agent debate or self-reflection can reduce errors. ChatDev, for example, noted that *thought instructions and self-reflection* improved code quality [92]; AutoGen provides generic support for such patterns. (3) **Tool and Code Integration** is a major plus. AutoGen agents can directly execute generated code (via a secure sandbox), query the file system, or call external tools as part of the conversation [83]. This means an agent can verify its outputs or perform computations – a capability that in other frameworks might require more custom plumbing. (4) **Asynchronous and parallel conversations:** AutoGen allows concurrent agent operations (e.g. multiple agents working in parallel threads of conversation) [93], which can speed up workflows and better utilize resources for independent sub-problems. (5) **Robust Logging/Debugging:** The framework was designed with enterprise use in mind, providing detailed logs of each message and easy hooks to inspect or intervene in the conversation. This is crucial when developing multi-agent systems, as emergent failures can be hard to diagnose. (6) **Community and Support:** Being backed by Microsoft, AutoGen benefits from a broader community and active development. It's open source (MIT license) and had significant attention from developers experimenting with multi-agent setups for LLMs in late 2023 [94]. Our research suggests it has a growing ecosystem of examples and likely better support for Azure services (e.g. Azure OpenAI integration) given its origin.

Limitations: AutoGen is not a panacea, and it has some drawbacks. (1) **Less structure out-of-the-box:** The flip side of flexibility is that AutoGen does not provide a predefined "process" for specific tasks. For example, unlike ChatDev (which has a built-in SDLC workflow), using AutoGen for software development would require the developer to design the conversation protocol (who says what and when). This can be challenging – essentially one must program the coordination through prompts or additional logic. As our internal notes indicated, AutoGen has "less general features" in the sense that it doesn't come with domain-specific templates, but it offers a powerful abstraction to implement whatever pattern you need [95] [96]. So, it demands more effort to customize for a particular use-case. (2) **Steep prompt engineering and tuning effort:** Achieving a productive multi-agent conversation can require careful prompt design for each agent and iterative tuning. AutoGen provides the tools, but developers must still craft the roles and interaction rules. This means leveraging AutoGen effectively might require significant experimentation. (3) **Potential for instability:** Free-form conversations between LLM agents can sometimes go off track or get stuck. Without a rigid structure, agents might loop or produce irrelevant outputs if not guided well. AutoGen presumably has some safeguards, but the risk of chaotic dialogues is inherent in open agent chats (especially with powerful LLMs). (4) **Resource Intensiveness:** Running multiple large LLM agents in parallel conversations can be computationally expensive. AutoGen makes it *possible*, but cost and performance need consideration. The framework may not inherently optimize token usage across agents (aside from what the LLMs themselves do); thus a long chat among several agents can consume a lot of tokens. (5) **Lack of UI / No-Code**: Similar to CrewAI, AutoGen lacks a no-code interface or visual designer. It's a developer-centric library. For CoFound.ai's internal team this is fine, but it's not something one can hand to a non-programmer to set up agent workflows. (6) **Compatibility and Integration:** If CoFound.ai already uses LangChain or LangGraph extensively, introducing AutoGen may overlap or conflict with those. AutoGen has some integration with LangChain tools as well, but one must ensure not to duplicate functionality. It's an additional layer that might complicate the stack if combined incautiously.

Applicability to CoFound.ai: AutoGen is highly relevant to CoFound.ai's goal of complex multi-agent collaboration (for instance, automating the software development pipeline). It provides a **generic multi-agent coordination engine** that could save us from writing our own from scratch. In our evaluation, we viewed AutoGen's "conversation-based" approach favorably for dynamic interactions and asynchronous support – indeed we scored it well on those aspects. If CoFound.ai requires more free-form brainstorming among agents (say multiple agents jointly architecting a solution), AutoGen is ideal. However, CoFound.ai also requires structured phases (design, coding, testing), which AutoGen alone doesn't provide out-of-the-box. One strategy could be to use **AutoGen within each phase**: for example, during the coding phase, instantiate an AutoGen conversation between a Developer agent and a Reviewer agent to refine code (leveraging AutoGen's turn-taking and reflection), then feed the result back to the main workflow. In essence, our LangGraph orchestrator could invoke an AutoGen "sub-conversation" as a tool for particularly complex sub-tasks. This hybrid approach tries to get the best of both worlds – the high-level process control of LangGraph and the micro-level conversation intelligence of AutoGen. Another integration point is using AutoGen's **code execution ability** in CoFound.ai. We could embed AutoGen's code runner to allow our Developer agent to test code it writes, capturing output and errors and feeding them back into the loop (something that ChatDev also does with Docker, and CrewAI currently lacks natively). AutoGen's logging can also augment our monitoring: hooking our orchestrator into AutoGen's event callbacks would give us fine-grained visibility into agent dialogues. On the other hand, adopting AutoGen wholesale as the primary orchestrator would require refactoring our current architecture – likely not worth it given the progress with LangGraph and our need for a deterministic workflow. It's also worth noting that AutoGen (being from Microsoft) might integrate well with certain Azure AI offerings or VS Code tools, which could be beneficial if we deploy on Azure or want IDE integration for agent development. Overall, AutoGen is a strong toolkit to

have in our arsenal for enabling **rich agent interactions**. We should consider prototyping some challenging scenarios (e.g. debugging a tricky coding task) with AutoGen to see if it yields better results than our current simple loop, and then integrate accordingly. AutoGen's existence also means we should ensure our system is *not* overly constrained – the success of multi-agent systems often comes from letting agents freely communicate, and AutoGen provides evidence that conversation-driven orchestration can solve very complex tasks (the literature shows multi-agent LLM systems outperform single agents on complex problems like software design given proper coordination [90] [89]). We should design CoFound.ai such that agents have channels to "discuss" when needed (even if mediated by our orchestrator), essentially emulating the AutoGen style when beneficial.

ChatDev – Specialized Multi-Agent Software Team

Architecture: ChatDev is an open-source framework that **emulates a software company's development workflow** using a team of specialized LLM-based agents [1]. Introduced by researchers (and associated with the OpenBMB initiative in late 2023), ChatDev was designed to showcase how an entire software application can be generated autonomously through multi-agent collaboration. The architecture follows a **software development lifecycle (SDLC)** model, specifically a simplified waterfall process: phases of **design → coding → testing → documentation** are carried out by different agents assuming roles analogous to human team members [97] [98]. For example, ChatDev typically includes agents like a CEO (to approve ideas), CTO/CPO (to do high-level design and planning), a Developer (to write code), a Tester (to test and find bugs), etc., each with behaviors suited to their role [99].

The unique aspect of ChatDev is that this process is largely **hard-coded as an orchestrated sequence of "functional seminars"** [100]. In each phase, relevant agents convene and interact in a structured way to produce the deliverables of that phase. For instance, in the *Design* phase, the CEO, CPO, and CTO agents might discuss the requirements and come up with a technical plan [99]. In the *Coding* phase, a Developer agent writes code (often with an iterative refine loop where the Reviewer or CTO agent gives feedback). The *Testing* phase involves the Tester agent potentially running the code or inspecting it for flaws, then the Developer fixes issues. Finally, the *Documentation* phase has an agent write user manuals or README based on the project knowledge. These transitions are orchestrated by an underlying controller that moves from one phase to the next once criteria are met (e.g., code is deemed satisfactory). ChatDev's orchestration is less flexible than CrewAI or AutoGen – it's a predetermined pipeline aligning with the waterfall model [101]. However, within each phase, agents do have interactive communication. ChatDev logs a *conversation transcript* of all agents, allowing one to replay exactly how a piece of software was conceived line by line [102].

Technically, ChatDev also integrates some tool usage: it can execute the code that the Developer writes (the project added a Docker-based sandbox to safely run code) [103], and the Tester agent uses that to verify functionality. The result of ChatDev's run is a complete package: source code files, test results, documentation, etc. In essence, ChatDev is a **vertical slice solution** demonstrating the power of multi-agent systems in a specific domain (software creation). It might not be a general-purpose framework one can easily repurpose for other domains without modification, but it's open-source and customizable within the context of software development.

Strengths: ChatDev's primary strength is its **end-to-end focus on a tangible use-case (software development)**. By constraining the domain, it achieves remarkable results – for example, reports claim ChatDev can develop a simple game in minutes for under a dollar of API cost [104] [105]. This is possible because the entire flow is optimized for that goal. Specific strengths include: (1) **Structured division of**

labor – each agent has a clear responsibility and the hand-offs between phases ensure that the process is thorough (design is done before coding, coding before testing, etc.). This structured approach reduces randomness and keeps agents on task. (2) **Transparency and traceability** – ChatDev logs the full multi-agent dialogue, which can be replayed or audited [102]. This is excellent for understanding how the AI reached a solution (critical for trust in autonomous systems). CoFound.ai could similarly benefit from maintaining detailed logs of agent interactions for review and debugging. (3) **Domain-specific optimizations** – ChatDev implements clever prompts and interaction patterns tailored to coding. For example, it uses *"thought instructions"* where an agent explicitly lists its plan or thoughts before acting, and *self-reflection* where agents double-check their outputs [92]. These techniques improved code correctness and quality in practice. We can adopt similar patterns in CoFound.ai's agents (regardless of framework) to reduce errors. (4) **Fast prototyping** – For generating standard application boilerplate (like a web app or a simple game), ChatDev's approach is extremely fast and cost-efficient [106] [107]. This suggests that for well-bounded tasks, a fixed multi-agent workflow can be highly effective. (5) **Customization and Extendability** – The ChatDev project emphasizes it is highly customizable and extendable within its scope [108] [99]. One can adjust the number of agents, their prompts, and even the sequence of phases. In fact, it's been used as a scenario to study collective intelligence in multi-agent frameworks [109]. (6) **Demonstration of Collective Intelligence** – ChatDev serves as a valuable proof-of-concept that a group of AI agents, each with limited knowledge, can together produce a result (a working software) that is beyond the capability of any single agent alone. This validates CoFound.ai's core premise. Each specialized agent in ChatDev compensates for the others' weaknesses – e.g., the Tester catches mistakes the Developer made, the Designer structures tasks so the Developer knows what to do, etc. In CoFound.ai, we seek similar synergy. (7) **Web Integration** – ChatDev also offered a web app interface and even a browser extension, making it easy to interact with or visualize the process [110] [111]. This isn't an inherent strength of the multi-agent method, but it shows the importance of good UX for such a system, a point CoFound.ai should note (we plan a web UI in the future, not just CLI).

Limitations: Many of ChatDev's limitations stem from the very rigidity that gives it focus. (1) **Lack of generality:** ChatDev is essentially hard-wired for the SDLC workflow. If one wanted to use it for a different domain (say, an AI-assisted marketing campaign planning), it's not straightforward to adapt. The agents and phases would need to be redefined. In contrast, CrewAI or AutoGen are frameworks where the domain logic is not baked in. IBM's comparison noted that ChatDev's process structure is "rigid, limiting customization and hindering scalability/flexibility for production environments" [112]. In a production setting, requirements change, and a rigid waterfall might not always apply – e.g., agile processes require more iteration, which ChatDev doesn't natively support (although one could manually restart phases). (2) **Scalability and Concurrency:** ChatDev typically runs through phases one after the other (waterfall). It doesn't have parallel agent interactions across phases. Also, adding more agents or tasks in the flow would need rewriting the orchestration logic. It's not designed to easily scale out to, say, 10 developers working in parallel on different modules. CoFound.ai, aiming to be a robust platform, might need more scalability. (3) **Missing Enterprise-Grade Features:** As noted by reviewers, ChatDev lacks certain features expected for real-world deployment [113] [114]. For example, it has **no authentication or role-based access control** for its agents (since it's a self-contained demo). It doesn't implement data encryption or secrecy – everything the agents do is in plain text logs. There is no concept of different environments (no separate staging vs production runs – it just generates code). These could be concerns if someone tried to use ChatDev's approach in an enterprise where security and DevOps practices matter [113] [114]. (4) **Dependence on LLM capability:** ChatDev's success is tied to powerful underlying models (like GPT-4). If a smaller model were used, the rigid roles might not compensate for lack of capability. More flexible frameworks could adjust (e.g., use more back-and-forth or external tools to assist a weaker model), whereas ChatDev assumes the

agents are smart enough to perform their phase largely correctly. (5) **Maintenance of output:** ChatDev outputs code, but after the run completes, it doesn't manage that code further (no continuous integration or long-term maintenance loop). In reality, software development is continuous; ChatDev would have to be re-run for modifications, and it's unclear how it would incorporate existing code (though a recent update did add "incremental development" allowing agents to build on an existing codebase) [115]. This is a nascent feature, but maintaining state across runs is still a challenge. (6) **Smaller Community:** ChatDev, while open source, doesn't have the backing of a major company, and its community is likely smaller than AutoGen's or LangChain's. It's more of a research project. Thus, if CoFound.ai were to rely heavily on ChatDev code, we might face difficulties if we need support or if underlying APIs change.

Applicability to CoFound.ai: CoFound.ai's initial vision is very close to ChatDev's implementation – a multi-agent "virtual software company." Indeed, our current architecture design includes agents for PM, Developer, Tester, etc., and a high-level workflow that resembles design→code→test cycles. Thus, ChatDev can be viewed as a *reference implementation* or baseline. One straightforward approach is to **learn from ChatDev's successes and pitfalls to inform CoFound.ai's design**. For example, we should incorporate the following lessons: (a) Use specialized agents for each SDLC phase, but ensure our architecture can support iterative or parallel development (perhaps an Agile mode). (b) Implement **transparency** – keep logs of all agent interactions like ChatDev does, and provide a UI to replay or analyze them. (c) Add **self-reflection and verification steps** – e.g., after our Developer agent writes code, have it (or another agent) review the code against requirements (possibly using techniques like checklist prompts or static analysis) to catch errors early. (d) Use a **testing agent with real code execution** – this is critical. CoFound.ai should integrate a safe environment (e.g., container sandbox) to run and test generated code, feeding results back to the agents. This closes the loop as ChatDev did, leading to higher quality outputs. (e) Plan for **incremental development** – we should allow our agents to work with an existing codebase (not only generate from scratch). ChatDev's new "incremental" feature suggests enabling an agent to read a repository and add/improve features, which aligns with real development. We should design our memory and tooling such that agents can retrieve and modify existing project files. (f) Address ChatDev's shortcomings by **introducing flexibility and production rigor**: CoFound.ai can implement a more modular orchestration (perhaps via LangGraph) so that phases could be rearranged or repeated, unlike ChatDev's fixed sequence. Also, building in authentication, proper data handling, and the ability to deploy the generated software (CI/CD) is part of our plan, which goes beyond ChatDev's scope. In terms of directly using ChatDev's code or framework: since our team already has its own codebase and since ChatDev is tailored to a specific flow, it may be more effort to integrate ChatDev's code than to implement similar logic ourselves using our chosen frameworks. Instead, we might run ChatDev in parallel as a benchmark to gauge CoFound.ai's performance ("can we match ChatDev's speed and quality on similar tasks?"). If falling short, we can inspect ChatDev's prompts and strategy to improve ours. In summary, ChatDev is both a **blueprint and a benchmark** for CoFound.ai. We strive to replicate its ability to quickly produce working software, while extending the approach to be more adaptable and enterprise-ready. Our architecture, by using a general orchestrator (LangGraph) and possibly integrating standards (ACP/MCP), should yield a system that can do what ChatDev does and more – e.g., handle multiple projects concurrently, integrate with human feedback, and evolve software over time.

Comparative Overview of Frameworks

The three frameworks differ in approach – CrewAI is **role & workflow oriented**, AutoGen is **conversation oriented**, and ChatDev is **application oriented** (hardcoded for coding tasks). Table 2 provides a side-by-side comparison on key criteria relevant to CoFound.ai:

Criteria	**CrewAI** (Modular Roles & Tasks)	**AutoGen** (Conversational Orchestration)
Scalability	Moderate – supports multiple agents and async tasks, but single-process Python orchestrator could bottleneck if agents or tasks grow very large. No native multi-machine distribution (manual needed).	High logical scalability – can spawn many agents and parallel conversations [93]. Bounded mainly by compute; architecture is flexible to distribute (agents could run on separate threads/ processes under a messaging layer).
---	---	---
Scalability	Designed for small **teams of agents**; can run multiple agents and tasks, including some asynchronous execution, but uses a single Python process. Not inherently distributed – scaling beyond a handful of agents may require custom extensions (e.g. running agents on separate servers). Suitable for moderate workflows, but very large-scale deployments would need further engineering.	Built for **parallel agent conversations** – supports concurrent agents and multi-threaded dialogues [93]. In principle, each agent could run on separate hardware if connected via the messaging layer (especially if integrated with protocols like ACP). Scalability is limited mainly by compute and token costs, not by architectural constraints.
Modularity	Highly modular – distinct components for agents, tools, tasks, processes [53] [116]. Easy to add new tools or agent roles. The process mechanisms (sequential, hierarchical) provide plug-and-play orchestration patterns. However, workflows are confined to those patterns; adding new coordination logic may require modifying the framework.	Flexible but **monolithic conversation loop** – everything is an agent messaging pattern. Easy to integrate new agent types (including human or tool agents) and swap prompts. Lacks higher-level constructs like phases or tasks; the developer must structure the conversation via prompts or code. This gives freedom but less pre-built modular workflow control.
Memory Handling	Supports **agent memory** via context and optional vector stores, but no centralized memory across agents by default. Each agent can have a backstory and use tools for knowledge (RAG) [117]. Outputs of tasks can serve as context for later tasks [65]. Long-term memory (persistent knowledge) would need integrating an external DB (e.g. a shared vector DB) – not provided out-of-box.	Leaves memory to the conversation context – uses the LLMs' own memory (conversation history) and can incorporate an external memory tool if the developer adds it. No explicit built-in long-term memory module, but one can implement memory through looping in summaries or using the provided **Model Context** integration (MCP) for external state [6]. So, memory is as good as the prompt management – developers must truncate or summarize to handle context length.

Criteria	**CrewAI** (Modular Roles & Tasks)	**AutoGen** (Conversational Orchestration)
Tooling Integration	Strong **tool ecosystem** – comes with a toolkit (web search, GitHub search, etc.) [117] [118], and integrates LangChain tools easily [59]. Adding custom tools is straightforward by defining a Python function. No native GUI, but coding new tools is developer-friendly. Missing an automated code executor (would need to add as a tool).	Very flexible – agents can use tools, and AutoGen provides built-ins for executing code and calling functions during conversations [83]. One can wrap any external API as an agent or tool. However, lacks a visual tool picker; all integration is via code. Debugging tools (like intervention handlers to approve tool use) are available, making it powerful for skilled developers.
Open-Source Ecosystem	**Active open source** (GitHub: crewai/crewAI). Small but growing community; IBM and others have published guides and use-cases [119] [80]. Being independent of big frameworks (recent versions don't require LangChain [120]) means it's nimble, but also means resources (community contributions, extensions) are limited compared to larger projects.	**Backed by Microsoft** – robust community interest, detailed documentation and examples by Microsoft's team [84] [121]. Likely to be maintained long-term and possibly integrated into Microsoft's offerings. No licensing issues (MIT). Community support is broader, but since it's general, community content spans diverse use cases (not all directly applicable to CoFound).
Production Readiness	**Moderate** – Code is relatively simple and Pythonic, which is good for maintainability. Lacks enterprise features like authentication, multi-user support, or horizontal scaling. Would require additional engineering (e.g., containerizing each agent as a service for a production microservice architecture). Error handling is basic (try/except around tools). Still, its simplicity means it can be extended for production, and its use of Python and standard libraries eases integration into our stack.	**Moderate/High** – Designed with real applications in mind (though currently early in adoption). Logging and debugging features are a plus for prod monitoring [91]. No-code interface is absent, but as a backend service AutoGen can be productionized with proper surrounding infrastructure. One concern: it's heavily reliant on prompt correctness; any brittleness there can affect reliability. It also assumes the underlying LLM service is reliable. With guardrails and human fail-safes, AutoGen could be used in production scenarios that tolerate some unpredictability (e.g., creative tasks).

Table 2: Comparison of multi-agent frameworks CrewAI, AutoGen, and ChatDev on key criteria.

As shown above, **CrewAI** offers a structured, modular approach that aligns well with defined team workflows but may require extensions for large-scale or highly custom scenarios. **AutoGen** provides a powerful, general platform for agent communication with excellent flexibility and debugging capabilities, but it relies on the developer to impose structure and is best suited for scenarios where free-form

collaboration is desired. **ChatDev** demonstrates the heights of automation in a specific domain, achieving remarkable speed and completeness in software generation, but its one-size process is rigid and not directly generalizable.

For CoFound.ai, which aims to be a **flexible yet domain-focused system (AI co-founder across various startup tasks)**, a **hybrid strategy** seems best: adopt the *structured modularity* of CrewAI (so that the system remains organized and extensible), enable *conversational flexibility* akin to AutoGen for complex reasoning (when agents need to brainstorm or debug in free-form ways), and incorporate the *domain-optimized workflows* of ChatDev for efficiency in our core use case (automating software development and other startup processes). In practice, this means our architecture should support **multiple interaction modes**: e.g., a controlled phase-wise mode for predictable tasks and an open dialogue mode for creative or complex problem solving, with seamless transition between them.

Integration Strategies for CoFound.ai

Having analyzed protocols (MCP, ACP, AGNTCY) and frameworks, we now formulate strategies to integrate these into CoFound.ai's architecture. The goal is to enhance CoFound.ai's **scalability, maintainability, and interoperability** by leveraging proven standards and components, without losing the focus and coherence of our product vision. We base these recommendations on the current CoFound.ai design (as reflected in our architecture documents and repository) and identify concrete integration points.

Current CoFound.ai Architecture (Overview)

CoFound.ai's MVP architecture is structured in layered fashion for a multi-agent system, comprising: a user interface layer (CLI, eventually web UI), an API/service layer, an agent orchestration layer (built with LangGraph state machines), an agent layer with specialized LLM-driven agents (PM, Developer, Tester, etc.), a communication/messaging layer for agent interactions, a long-term memory store (vector database), a workflow controller (managing state transitions and phases), a monitoring layer (logging, error tracking, human override), and a data persistence layer for artifacts (code files, documentation, version control). There is also consideration for CI/CD integration to handle the outputs of the agents (for example, automatically building or deploying generated software). Figure 2 depicts a high-level conceptual architecture of CoFound.ai as designed, with placeholders for where new protocols/frameworks could fit in.

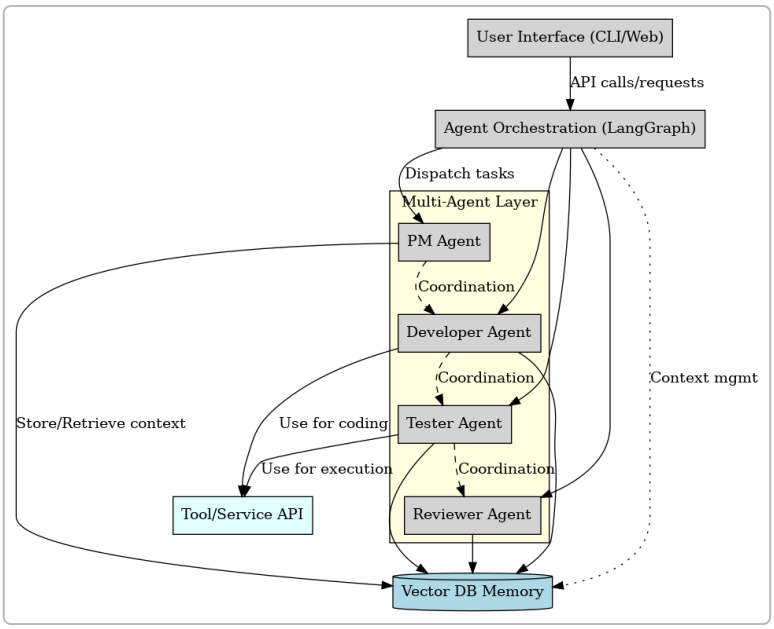

Figure 2: Conceptual architecture of CoFound.ai's multi-agent system. The User Interface (CLI or web) interacts with the system via an API layer, delegating user requests to the Agent Orchestration layer (built on LangGraph). The Orchestrator dispatches tasks to a Multi-Agent Layer composed of specialized agents (Project Manager, Developer, Tester, Reviewer, etc.). These agents communicate and coordinate (dashed arrows) to complete tasks – currently using custom logic, but in the future potentially via standardized protocols like ACP. Agents use a shared Vector DB Memory for long-term context (for example, to retrieve past decisions or stored knowledge) and can invoke external Tools/Services (like code execution environments or web search APIs) to aid in their tasks. Dotted lines indicate management and context flows (e.g., the orchestrator managing overall context, or injecting necessary data via an MCP-like mechanism). This layered design separates concerns: UI/API handling, orchestration logic, agent reasoning, memory, and tool usage.

In our implementation, the **LangGraph** orchestration acts as a state machine controlling the phases of the project (not unlike ChatDev's phased approach, but more flexible). Each agent is implemented as a class that interfaces with an LLM (via an API like OpenAI) and has methods to receive instructions and produce outputs. Currently, inter-agent communication is handled in-process by the orchestrator: e.g., the PM agent's output (a spec) is passed as input to the Developer agent by the Python code. This works for now but lacks the formalism and scalability of a standardized protocol.

Incorporating ACP for Agent Communication

One of the first integration steps is to adopt the **Agent Communication Protocol (ACP)** to formalize how our agents interact. Instead of treating agent outputs as function returns in Python, we can refactor each agent to run as a service (could be a lightweight FastAPI or gRPC service) exposing ACP-defined endpoints [13] [14]. For example, each agent would implement endpoints like `POST /invoke` (to run the agent on a given task with certain context), `GET /status` (to check its state or retrieve results, enabling asynchronous invocation), and `POST /interact` or `/message` (to send a message in a threaded conversation). By doing this, we decouple agents from the orchestrator process – each agent could live in its own container or microservice, registering its capabilities with the orchestrator. The orchestrator (which

could become an **ACP client** or an ACP-compliant "workflow server") would then call agents via HTTP or RPC calls as per the ACP spec, rather than direct method calls. This shift yields several benefits:

- **Scalability & Resilience:** Agents as independent services can be scaled horizontally. For instance, if the Developer agent's workload is heavy, we could run multiple instances behind a load balancer (especially if stateless for each request). If an agent crashes, it can be restarted without crashing the whole system – the orchestrator just gets an error response (which could correspond to an ACP error code for a failed request). This aligns with the microservice approach recommended for agent systems [46] [47], where each agent manages its own state and communicates via well-defined APIs rather than shared memory. In essence, ACP would help us move toward an "agents-as-microservices" architecture, improving fault isolation and deployment flexibility.
- **Interoperability:** If our agents speak ACP, we could swap out an internal agent for an external one that adheres to the protocol. For example, suppose in the future there is a specialized Requirements-Analysis agent from a third-party that is ACP-compliant – CoFound.ai's orchestrator could call it just like our own agents. Conversely, external systems could invoke our agents through ACP, if we allow. This opens possibilities for integration (perhaps a user could "hire" CoFound.ai's Dev Agent into their own workflow via ACP calls).
- **Stateful Conversations:** ACP's thread concept can be leveraged so that a series of calls to an agent can be tied to a context thread ID [122] [123]. We can map CoFound.ai's project or task IDs to ACP thread IDs, enabling stateful multi-turn interactions. For instance, during a coding phase, the orchestrator and Developer agent might go back and forth (multiple ACP calls) refining code; using a thread ID ensures the agent maintains context through that dialogue without us manually concatenating prompts each time.
- **Tool Provider vs. Agent distinction:** The ACP spec also contemplates scenarios where an agent might act as a tool provider versus a full agent [124]. In CoFound.ai, some agents might be very simple (e.g., a Translation agent or an API caller) – those could be implemented as "tool providers" that the main agents call through ACP. Or our system could host tools that are exposed via ACP to agents.
- **Adopting ACP gradually:** We do not need to rebuild everything at once. A practical plan is to create an **ACP adapter layer** within our orchestrator. Initially, our orchestrator can include an HTTP server that listens for ACP-formatted requests (JSON) and translates them to internal method calls, and vice versa for responses. We can test this with one agent (say the Tester agent) by making it an external service and adjusting orchestrator calls to use HTTP. Over time, we can migrate other agents. Throughout, we maintain functional parity with our existing system (so there's no regression in capability, just a different call mechanism).

One challenge will be that ACP is in alpha – we must keep an eye on the evolving spec. We should join the BeeAI community discussions (as invited in the ACP announcements) [22] to stay updated. In implementing ACP, we might even contribute feedback from our real-world use. By aligning early, CoFound.ai could become one of the early adopters of ACP, potentially influencing the standard to accommodate our needs (for example, ensuring that things like code file transfers or complex data artifacts are supported in the protocol). The Medium example by Omar Santos shows rich message structures with file parts and data parts in ACP/A2A context [125] [126] – we should ensure our agents can handle such structured inputs/outputs (e.g., sending a file artifact to the Tester agent via ACP rather than just plain text code).

Using MCP for External Knowledge and Tools

CoFound.ai agents will often need information beyond what's in their immediate context (for example, knowledge about a programming library, market research data, etc.). We currently plan to use a vector database for long-term memory and possibly direct API calls for tools (via LangChain or custom integrations). By incorporating the **Model Context Protocol (MCP)**, we can standardize how agents pull in external data as context. A practical approach is to set up an **MCP server** component in our architecture that acts as a registry of tools and data resources (somewhat analogous to LangChain's toolkit but accessible via a network protocol).

For instance, we could host a "Documentation Knowledge Base" as an MCP service. An agent (like the Developer) when needing to recall how to use a certain library, could query the MCP server for relevant documentation. Under MCP, this query would be formatted in a standard way and the response would maintain the context threading, etc., so the agent can incorporate it smoothly [38] [127]. Another example: If the PM agent wants to analyze current market trends (say for deciding on app features), instead of us custom-coding an API call to a market data service, we register that API on the MCP server. The PM agent then does an MCP lookup for "market data" and uses whatever comes back, without needing to know the specifics of the API (the MCP server handles it). Essentially, MCP can serve as an **abstraction layer for tools/data**, reducing custom integration code and making it easier to swap data sources. This is in line with MCP's purpose of giving AI models access to external context in a standardized way [3] [128].

We should also use MCP's concepts to manage **conversation state across agents when needed**. For example, a centralized store of conversation history (in a structured form) can be an MCP resource that all agents subscribe to. However, since we plan to use ACP for direct agent comms, MCP's role would be more for connecting to *external* systems rather than between CoFound.ai's own agents.

An immediate MCP integration could be: implement a simple MCP server within CoFound.ai that registers our Vector DB as a service (for long-term memory retrieval) and perhaps a Code Repository service (to fetch code by file name from the project's git, acting as a read-only tool). Then modify our agents to perform MCP calls for those actions instead of direct DB queries. While this may seem like an extra layer, it forces a clean separation: agents become less tied to the specifics of our vector database or file system. In the future, if we adopt a different memory store, we just update the MCP server. Also, if agents from outside CoFound.ai wanted to query our knowledge base, exposing it via MCP would let them do so without exposing internals. This aligns with building an ecosystem-ready platform.

One caveat: MCP is newer and we might not find off-the-shelf MCP server implementations yet (though Anthropic provides some reference). We may implement a minimal subset (even if just as a JSON API that mimics MCP's expected format). Over time, we can move to a full-fledged MCP server (perhaps running Anthropic's official one if open-sourced).

Embracing AGNTCY Components

While ACP and MCP are specific protocols, **AGNTCY** provides a broader blueprint. There are a few concrete components from AGNTCY we should consider integrating in the medium term:

- **Agent Directory (Discovery):** In a closed system like CoFound.ai (with a fixed set of internal agents), an agent directory might seem unnecessary. However, if we imagine CoFound.ai expanding to

include many specialized agents (for different departments – marketing, finance, etc., or even plugin agents built by third-party developers for our platform), a directory service could be invaluable. It would allow dynamic discovery and composition of agents. We could run a lightweight Agent Directory that holds metadata (perhaps conforming to the OASF schema) for each agent – including its name, role, capabilities (what tools it has, what tasks it can do), and current endpoint (if distributed). Our orchestrator (or any agent) could query this directory to find which agent is best suited for a task. This also means we could **make CoFound.ai extensible**: new agents could be added to the system by registering themselves in the directory, and the orchestrator could pick them up without hardcoding. In essence, our architecture moves toward an **open plugin model** for agents, which is very much in the spirit of AGNTCY's IoA. Security and trust would need to be managed (we wouldn't just trust any external agent unless vetted), but the technical capability would be there.

- **Agent Gateway (Messaging):** If we fully adopt ACP/AGP, we might incorporate an AGNTCY Agent Gateway – which is essentially a router for agent messages using gRPC streams [29] [45]. This could replace or augment our orchestrator's networking. For example, instead of our orchestrator calling each agent's HTTP endpoint directly (point-to-point), all agents (and orchestrator) could connect to a central Agent Gateway broker. They would then send messages addressed to other agents via the gateway (sort of like a message bus for agents). The gateway can handle queuing, retries, and even pub/sub patterns. This architecture could improve reliability (agents can come and go, the gateway buffers messages) and flexibility (e.g., multiple agents subscribe to a topic). However, implementing this might be beyond MVP scope – it's more relevant as we scale up or if we want to integrate with external agent networks. Initially, a simpler REST call approach might suffice.
- **Standard Agent Schema (OASF):** We should document each CoFound.ai agent with a standardized schema: what is its function, input/output format, performance metrics, etc. Using the OASF schema to do this means we could directly publish our agents to an AGNTCY directory or even marketplace. This is low-hanging fruit: it's basically writing metadata. We can start by creating YAML or JSON manifests for each agent in our repo. This improves maintainability (clear spec for each agent) and interoperability (others understand what our agents do). It also connects to best practices of **high cohesion and clear interfaces** – each agent's interface should be as clearly defined as an API's, which helps if we swap implementations.

By aligning with AGNTCY's vision early, CoFound.ai can position itself not just as an application, but as a **platform of cooperating AI services**. This might open new business or integration opportunities (for example, partnering with another company's agent to complement our platform, exchanging capabilities through the common standards).

Integration of Framework Concepts and Libraries

Beyond protocols, we should integrate useful **concepts or components from CrewAI, AutoGen, ChatDev** into our architecture:

- **CrewAI's Process Control:** We plan to use LangGraph for orchestration, which similarly allows graph-based workflows. We can implement a **hierarchical process** in LangGraph (we can designate one of our agents as a Manager agent that receives a high-level goal and then sequentially triggers other agents' LangGraph states). This draws directly from CrewAI's hierarchical process concept [76]. By doing so, we can handle more complex or unplanned sequences by letting the Manager agent dynamically decide next steps (the Manager could be an LLM agent itself, effectively an internal orchestrator using natural language reasoning – a technique for meta-planning). We can also

consider implementing a **consensus mechanism** if we ever have multiple agents voting (not immediate, but conceptually, our architecture should allow an outcome to be decided by querying all agents and combining answers, akin to future CrewAI consensual process).

- **AutoGen's Conversation Engine:** We might integrate AutoGen's library to manage interactions in certain parts of the workflow that benefit from a free-form chat. For example, during the Design phase, rather than having a rigid script, we could initiate an AutoGen `AgentChat` session [129] with the PM, Developer, and maybe a UX Designer agent all talking to sketch out the product requirements. The output of that chat (transcript or a design document) can then feed the next phase. Technically, this means embedding AutoGen as a sub-module: the orchestrator triggers AutoGen to spin up those agents with given roles and a shared goal, runs the conversation for N turns or until completion, then returns. This harnesses AutoGen's strength in unstructured collaboration where needed, without making it control the entire system. Also, we can use AutoGen's code execution ability in the Testing phase: e.g., the Tester agent's internal logic could actually be an AutoGen agent that receives code, runs it in an isolated environment (via AutoGen's mechanism), and converses with a "Debugger" agent if errors occur – essentially automating debugging. This goes beyond ChatDev's simpler test approach by adding an intelligent debugging loop (similar to two agents in AutoGen discussing a bug).
- **ChatDev's Phase Templates:** We will explicitly design our LangGraph workflows based on ChatDev's phases – *but* we will parameterize them. For instance, rather than a single fixed waterfall, we create a workflow template that can iterate. If tests fail, our workflow can loop back to the coding phase (an Agile-esque iteration) instead of just ending. This gives flexibility ChatDev lacks. But having the template means we achieve the same outcome. We should also use ChatDev's prompt strategies: ensure each agent gets a clear **role prompt** (e.g., "You are a Tester, your job is to find faults") and possibly use the "seminar" style – which might mean during each phase, agents communicate in a channel. If not using AutoGen for that, we could implement a lightweight version: e.g., in the coding phase, have the Developer agent generate code, then explicitly prompt the Reviewer agent to critique it, then have Developer fix it. This is a deterministic script, but it injects the self-reflection that ChatDev found valuable [92]. So integration here is more about adopting techniques in prompts and workflow design rather than code.
- **Memory and Knowledge Integration:** All frameworks highlight memory as an issue. We should implement a **shared memory** that all agents can access when needed (likely through the MCP server as noted, or a simple shared vector store interface). For example, after each phase, we store a summary of decisions made (design decisions, known bugs, etc.) in memory. All agents in later phases query this to avoid inconsistencies or forgetting. We can use LangChain's memory abstractions or our own, but the key is to not rely solely on the LLM context window. Summarization agents or routines might be employed to compress knowledge (AutoGen's reflection pattern can help here by summarizing conversation so far).
- **Human in the Loop:** None of the frameworks explicitly integrated human feedback in their main loop (AutoGen allows it conceptually, ChatDev and CrewAI assume fully autonomous). For CoFound.ai, since it's a co-founder AI, we likely want the user (human founder) to be able to intervene or provide input at points. This is not directly asked in the question, but it's a best practice: allow a human agent. AutoGen readily supports a human agent in the conversation; CrewAI could allow a human as a "tool" input or via an interface. We should design our architecture such that at key checkpoints (say after design phase or before finalizing code), the system can present output to the user for approval or guidance. This improves quality and safety. It can be framed as an agent in our system (the "Human QA" agent which actually just awaits user input). Technically, integration of a human could use ACP too (the human interface could call an ACP endpoint to input a message).

Best Practices and Standards for Multi-Agent Systems

In integrating all the above, we adhere to and reinforce **best practices for scalable, maintainable, and interoperable multi-agent systems**:

- **Loosely Coupled Agents:** Each agent is treated as an independent service with a clear interface (akin to a microservice). They do not share memory directly; all interaction is through messages or the shared memory service, which ensures minimal direct dependencies [47] [130]. This loose coupling means we can update one agent's internals (e.g., swap its LLM or change its prompt strategy) without affecting others, as long as the interface contract (inputs/outputs) is stable.
- **High Cohesion in Agents:** Each agent has a well-defined **single responsibility** [131] [130]. Our PM agent focuses only on project management decisions, Developer on code generation, etc., and even finer: if we add a Requirements agent or a Security Review agent, they each stick to their domain. This makes it easier to maintain and improve individual agents (e.g., we could fine-tune a model specifically for the Tester role in future to improve its bug-finding, without altering anything else).
- **Standard Interfaces and Schemas:** Using ACP/MCP means our interfaces are not ad-hoc JSON blobs but standard fields (taskId, role, content parts, etc.) [132] [133]. Even internally, we should define consistent data models for things like a "Task specification" or "Bug report" so that if we pass these between agents or store them, everyone understands the format. Adopting OASF schema for agent metadata is part of this; we might also consider using a common ontology for artifacts (e.g., using JSON schemas for design docs, etc., so agents produce outputs that are machine-parseable for the next agent).
- **Error Handling and Timeouts:** Multi-agent systems can fail in novel ways (one agent might stall and never respond, or produce nonsense that confuses others). Our orchestrator should implement robust **timeouts** for calls (if an agent doesn't respond in X seconds, assume failure and either retry or ask a different agent). ACP will help here because it has notions of request status and cancellation [41] [134]. We should also instrument agents with fail-safe prompts (if they detect confusion or lack of progress, they should signal failure or request human help rather than looping endlessly). Logging each step (with unique IDs) will allow us to pinpoint issues later.
- **Monitoring and Evaluation:** Building on the best practice, we will include a monitoring layer – every agent action and decision is logged (with appropriate levels for sensitive info). We can use these logs to compute metrics (e.g., how many times did we have to loop back due to bugs, how long each phase takes, etc.). Over time, this will inform improvements. For interoperability, if we use AGNTCY standards, we might also output logs in a standard format that can be used by evaluation tools (AGNTCY mentions plans for evaluation frameworks). Regular evaluation of agents (perhaps using an automated "exam" or test cases for the Developer agent) ensures maintainability – if we swap the LLM or update prompts, we re-run evaluations to see if performance changed.
- **Security and Sandbox:** As the agents can execute code and fetch data, we must enforce security. Running code in a sandbox (Docker with no network, perhaps) is mandatory to prevent any harmful actions during testing. Also, when using external tools via MCP, ensure those calls are sanitized (an agent shouldn't be able to, say, query an internal database unless explicitly allowed). AGNTCY's emphasis on secure communication (MLS encryption via Agent Gateway) [34] is forward-looking; while not urgent for an internal system, if CoFound.ai ever offers an API or connects to external agents, we will incorporate encryption/auth (e.g., require API keys for any external ACP calls).
- **Versioning and Configuration Management:** We should version our agents' prompts and capabilities. As we improve an agent, treat it like a deployable service (with version number). The orchestrator or directory can then specify which version to use. This avoids sudden mismatches.

Also, manage configurations (like which LLM model each agent uses, and tool endpoints) in config files – easier to update without touching code.

- **Continuous Improvement Cycle:** Encourage a practice where agents can **learn from experience**. Full online learning is hard, but even a simple mechanism like having agents reflect at the end of a project ("What went wrong? What could be improved?") and logging that could guide developers to refine the system. Alternatively, maintain a memory of past successful strategies (e.g., if the Developer agent found a good approach to a certain type of problem, store that as a pattern to reuse). This borders on research, but it's something to keep in mind as a differentiator in long-term maintainability – the system should get better with use, not degrade.
- **Human Override and Feedback:** In a production setting, always have a fallback: if agents are not converging or produce a questionable result, flag for human review. This ensures reliability while the system is not 100% perfect (which is likely indefinitely, given the complexity). It also provides additional training data if we capture where humans had to intervene.

By implementing these practices, CoFound.ai's system will be easier to maintain (each component well-defined, issues easily isolated) and scale (able to run distributed, handle more tasks by adding more agent instances) and will align with industry standards (making integration and collaboration easier).

Architectural Diagrams with Integration Points

To illustrate how the discussed protocols and frameworks embed into CoFound.ai, we provide both a high-level integration view and a component-level view focused on agent interactions:

High-Level Integration View: In Figure 3, we show CoFound.ai's architecture augmented with ACP/MCP and multi-agent components. The user interacts via our UI/API. The **Orchestration Layer** (potentially running a LangGraph state machine or an AGNTCY Workflow Server) coordinates tasks and messages. Each agent (PM, Dev, Test, etc.) is deployed as a microservice with an ACP interface (blue dashed boundaries indicate ACP/HTTP communication). There's an **Agent Directory** registry (if implemented) that the orchestrator queries to find agents (yellow pages icon for discovery). The **MCP Server** is deployed alongside, providing access to external data/tools (e.g., Vector DB memory, external APIs, etc.) via standardized calls (green arrows). During operation, the orchestrator uses ACP calls to engage agents in phases (for structured parts, it might call one agent after another; for conversational parts, it could facilitate a group chat via a series of ACP messages between agents). Agents that need data (say Developer needs previous code context) call the MCP server to fetch it (the request goes out in MCP format and comes back with data, which the agent then includes in its reasoning). All agents also send logs/events to a **Monitoring & Logging service** (not shown in earlier figure for simplicity) – this collects traces for analysis. Optionally, a **Human Feedback Portal** is connected, which can intervene via the orchestrator (the human can effectively act as an agent, injecting guidance through an ACP endpoint reserved for human input).

This high-level view emphasizes **separation of concerns**: orchestrator vs. agent services vs. tool services, all connected through standard protocols (no opaque internal APIs). It also highlights extensibility: new agents or tools can be added by registering in the directory or MCP server, without altering existing code.

Component-Level Interaction (Agent Collaboration): At a more granular level, consider how two agents – Developer and Tester – collaborate on fixing a bug, using our integrated approach (Figure 4). The Developer agent writes code and sends it to the Tester agent via an ACP message (or the orchestrator mediates but using ACP formats). The Tester agent executes the code (using an execution tool via MCP if needed) and

finds a failing test. Tester then sends an **ACP message** back to Developer: not just a simple text, but a structured artifact containing the error log and perhaps a suggestion (leveraging ACP's ability to carry structured Parts [135] [136]). The Developer receives this, updates the code. They might iterate directly (back-and-forth ACP calls) a few times. This is effectively an AutoGen-like conversational loop but happening through ACP calls between separate services. Meanwhile, all these interactions are being logged, and if a certain number of iterations pass without success, the orchestrator intervenes to perhaps bring in the Reviewer agent or ask for human help (encoded as a policy in the workflow logic). Finally, when Tester reports all tests passed (status "completed" with an artifact of results), the orchestrator moves to the next phase. This sequence shows how using the protocols allows rich interaction: the error log passed from Tester to Developer could be a `FilePart` or `DataPart` in ACP terms [137] [138], ensuring Developer agent can easily parse it. If we didn't have ACP, we'd just send a raw text, which might be fine, but as we scale to more complex data (images, structured outputs), ACP gives a formal way to include those.

From a maintainability standpoint, this also means we could swap out the Tester agent for an external testing service (if one existed) as long as it understands the ACP artifact format. Our Developer wouldn't know the difference. This is the power of adhering to a standard. Similarly, by using an execution tool via MCP, we could change the execution backend (today Docker, tomorrow perhaps a firecracker VM or a specialized sandbox) without the Tester agent's logic changing – it just calls MCP "run_code" and gets results.

Another component-level view to consider is the memory integration: whenever an agent finishes a significant step, it might store a summary in the Vector DB via MCP. Later, another agent does an MCP query for relevant info. This decouples memory updates/queries from any single agent's code – a standard practice (kind of like how all microservices might use a shared database with well-defined schema, here MCP defines the access pattern). It prevents issues like one agent forgetting to forward information to another – instead, important info is globally available (with proper indexing/permissions).

In terms of diagrams, we would illustrate these interactions with sequence diagrams or flow diagrams, showing ACP calls (with thread IDs and payloads) and MCP calls. Due to the text format here, we've described them, but an actual technical strategy document for publication would include these as visual UML sequence diagrams for clarity.

Conclusion

CoFound.ai stands at the forefront of a new generation of AI systems – those that harness **multiple cooperating agents** to achieve what single models cannot. By carefully analyzing and embracing emerging protocols (like ACP for agent communication and MCP for context integration) and leveraging the strengths of state-of-the-art frameworks (CrewAI, AutoGen, ChatDev), CoFound.ai can build a multi-agent architecture that is **scalable, modular, and future-proof**. We will adhere to rigorous engineering best practices, treating agents as microservices with standardized interfaces, and maintaining clear boundaries between components such as orchestration logic, agent reasoning, and external tools. This not only ensures our system is maintainable and extensible, but also positions us within a larger ecosystem of interoperable AI agents – the "Internet of Agents" that AGNTCY envisions [25] [27].

In practical terms, our roadmap includes refactoring our agents to use ACP endpoints, deploying an internal MCP server for tools/knowledge, and progressively modularizing our workflow using ideas from CrewAI (roles & processes) and AutoGen (conversational sub-loops). We will take inspiration from ChatDev's success

to optimize our agent collaboration for software development, while not being bound by its rigidity – our design will allow iterative and dynamic processes, guided by a LangGraph-based orchestrator that can adapt workflow paths based on outcomes. By implementing thorough logging, monitoring, and incorporating human oversight options, we ensure the system remains reliable and aligns with real-world team workflows (after all, even autonomous teams sometimes need a manager's input – in our case, the human user can step in when needed).

Academically, this approach touches on several cutting-edge research directions: from ensuring **inter-agent communication languages** (akin to FIPA ACL in older multi-agent research, but now with modern neural capabilities and standards) to exploring **collective emergent intelligence** (how specialized LLMs can complement each other's knowledge – something evidenced by ChatDev and to be expanded in CoFound.ai). CoFound.ai can contribute to this field by reporting on our findings – for instance, which protocols worked best in practice, or how multi-agent collaboration improved the success rate of complex tasks (we should gather such data via the evaluation framework).

In conclusion, by marrying the robust *standardization of protocols* (ACP/MCP/AGNTCY) with the *innovative designs of frameworks* (CrewAI's orchestration, AutoGen's conversations, ChatDev's domain expertise), CoFound.ai will build a platform that is not only powerful and efficient for automating startup co-founder tasks, but is also **extensible and interoperable** with the wider AI agent ecosystem. We will be well-positioned to integrate new advancements (e.g. if a new agent comes along or a new protocol becomes standard) with minimal friction. This strategy document serves as both a blueprint for our engineering team and a whitepaper for the community, demonstrating that CoFound.ai's architecture is grounded in the latest research and best practices, and that we are committed to delivering a system that is at once cutting-edge and reliable.

---

1 97 98 100 101 109 What is ChatDev? | IBM
https://www.ibm.com/think/topics/chatdev

2 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 111 112 116 117 118 119 What is crewAI? | IBM
https://www.ibm.com/think/topics/crew-ai.html

3 7 9 10 12 13 14 19 20 21 36 38 39 43 46 47 122 123 127 128 130 131 Outshift | MCP and ACP: Decoding the language of models and agents
https://outshift.cisco.com/blog/mcp-acp-decoding-language-of-models-and-agents

4 5 6 8 17 23 24 40 134 Understanding the Agent Communication Protocol (ACP) and Its Evolution from MCP | by Sree Potluri | Medium
https://medium.com/@SreePotluri/understanding-the-agent-communication-protocol-acp-and-its-evolution-from-mcp-c28ad30c8ee0

11 15 16 18 22 41 124 Evolving Standards for agentic Systems: MCP and ACP | Niklas Heidloff
https://heidloff.net/article/mcp-acp/

25 26 27 28 29 30 31 32 33 34 35 37 42 44 45 125 126 132 133 135 136 137 138 Comparing MCP, A2A, and AGNTCY in the AI Agent Ecosystem | by Omar Santos | May, 2025 | Medium
https://medium.com/@santosomar/comparing-mcp-a2a-and-agntcy-in-the-ai-agent-ecosystem-f3234b85c475

84 85 86 87 94 95 96 99 108 110 121 AutoGen vs. ChatDev: Pioneering Multi-Agent Systems in the LLM Landscape
https://www.ikangai.com/autogen-vs-chatdev-pioneering-multi-agent-systems-in-the-llm-landscape/

88 89 90 93 129 Intro — AutoGen
https://microsoft.github.io/autogen/stable//user-guide/core-user-guide/design-patterns/intro.html

91 92 102 104 105 106 107 113 114 AutoGen Vs ChatDev: A comprehensive comparison
https://smythos.com/ai-agents/comparison/autogen-vs-chatdev/

103 115 GitHub - OpenBMB/ChatDev: Create Customized Software using Natural Language Idea (through LLM-powered Multi-Agent Collaboration)
https://github.com/OpenBMB/ChatDev

120 CrewAI: Introduction
https://docs.crewai.com/introduction