DMAS-Forge: A Framework for Transparent Deployment of AI Applications as Distributed Systems

Alessandro Cornacchia
KAUST

Vaastav Anand
MPI-SWS

Muhammad Bilal
KAUST

Zafar Qazi
LUMS & KAUST

Marco Canini
KAUST

Abstract

Agentic AI applications increasingly rely on multiple agents with distinct roles, specialized tools, and access to memory layers to solve complex tasks—closely resembling service-oriented architectures. Yet, in the rapid evolving landscape of programming frameworks and new protocols, deploying and testing AI agents as distributed systems remains a daunting and labor-intensive task. We present DMAS-Forge, a framework designed to close this gap. DMAS-Forge decouples application logic from specific deployment choices, and aims at transparently generating the necessary glue code and configurations to spawn distributed multi-agent applications across diverse deployment scenarios with minimal manual effort. We present our vision, design principles, and a prototype of DMAS-Forge. Finally, we discuss the opportunities and future work for our approach.

1 Agentic AI

Agentic AI represents the next stage in the evolution of intelligent systems. Agentic AI augments a traditional AI model by incorporating advanced capabilities such as planning, reasoning, contextual memory, and tool use. AI agents can dynamically direct their own tool usage and follow the set of steps towards achieving a goal [14, 25]. These features enable agents to work autonomously with minimal human intervention. Therefore, AI agents are designed to operate in dynamic environments where adaptability and strategic decision-making are essential.

Multi-agent systems (MAS). MAS extend this paradigm by enabling multiple AI agents to collaborate in pursuit of shared objectives. Each agent within such a system possesses a degree of autonomy, specialized skills, and a localized view of the broader environment. Through coordination, communication, and task-sharing, MAS can address problems that are too large, complex, or interdependent for a single agent to manage effectively. Attempting to assign a highly complex task to a single agent often leads to challenges: instructions may become overly complicated, the likelihood of errors increases, validation becomes more difficult, and the agent may require excessive access to tools and permissions. By distributing responsibilities across multiple agents, MAS reduces these risks. Each agent can be restricted to a well-defined scope of action, equipped with the tools necessary for its role, and powered by the most appropriate AI models for its specialized function. Furthermore, the use of distinct memory systems across agents enhances adaptability, as agents can draw on task-specific knowledge while contributing to a collective goal. Therefore, while an individual AI agent can perform a wide range of tasks, the collaborative nature of MAS allows for far greater reasoning quality and reliability [10, 15].

Collaboration strategies in MAS can follow two paradigms. The first is dynamic coordination, determined at runtime, where agents flexibly communicate intentions, share information, and negotiate task assignments. The second is a more predictable, workflow-based approach, where intra- and inter-agent interactions are structured as a graph. The graph predefines the execution flow—i.e., code path. Examples of the latter include prompt-chaining [24], parallelization and routing [14], self-consistency [23], self-refine [18], and various combinations thereof. In practice, complex MAS applications adopt a mixture of the two approaches.

2 Problem definition

With MAS continuing to grow in complexity and size, there is increasing consensus towards deploying and integrating MAS with distributed systems [4, 8, 20]. Powered with standardized communication protocols such as Google's Agent2Agent Protocol (A2A) [17] and Anthropic's Model Context Protocol (MCP) [2], **d**istributed MAS (**DMAS**) are emerging as a novel trend. We first motivate this trend, then we discuss the limitations of existing programming frameworks in supporting DMAS.

2.1 Why distributed systems?

Akin to microservices, distributing agents into their own services or containers, rather than deploying them as part of a monolith, offers significant architectural and operational benefits. This approach enables heterogeneous runtimes and simplifies dependency management, allowing each team to adopt the agentic framework that best suits their use case without bloating the system with unnecessary libraries. It

also promotes agent reuse, where a robust, fault-tolerant deployment of a specific agent can be leveraged across multiple applications. Security boundaries are strengthened through containerization, which allows fine-grained controls such as seccomp/AppArmor profiles, separate service accounts, and tailored network policies — particularly important when handling Personally Identifiable Information (PII), integrating with external APIs, or managing third-party secrets.

Furthermore, isolating agents can reduce tail latency by mitigating the impact of failures or model timeouts, ensuring more predictable system responsiveness. If a container crashes or a node fails, only the affected part of the workflow needs to be retried, rather than restarting the entire request. In certain scenarios, the retries might be avoided due to the stochastic behavior of LLMs and the inherent adaptability of MAS. For example, when agents coordinate dynamically (§ 1), one may decide to entirely disregard the failed interactions (e.g., due to the crash of a tool or agent container) without compromising the quality of the overall outcome.

Finally, containers provide resource isolation and enable independent scaling, so similarly with serverless [3], agents with diverse CPU, memory, or I/O requirements can be tuned and deployed optimally, avoiding contention and improving overall system performance.

2.2 Disconnect between programming frameworks and DMAS

Existing programming frameworks–such as LangGraph [13], CrewAI [12], AutoGen [9], LlamaIndex [16] and Agno [19]– enable programmers to structure applications in workflows and agents, and offer built-in modules for agent coordination, tool integration and memory management. However, they tightly couple the application logic with the execution environment by hard-coding communication interfaces. Each specific framework implements inter-agent communication via its own modules assuming a monolithic deployment (e.g., message passing in LangGraph or group chats in Autogen). As a consequence, while it is practical for AI engineers to write MAS applications leveraging today's frameworks, deploying such applications as a distributed, protocol-compliant system is considerably more demanding. It requires substantial manual effort, especially when the deployment environment needs to be changed (e.g., porting a MAS from Kubernetes [20] to Temporal [8]). In real cases, practitioners are often puzzled on how to achieve this [5–7].

Example. Consider a LangGraph workflow [14]. To deploy it as a distributed system, one would need to: *(i)* decide a partitioning logic to create a distributed graph made of smaller sub-graphs; *(ii)* create separate LangGraph workflow for each sub-graph; *(iii)* stich-the-dots, trying to connect
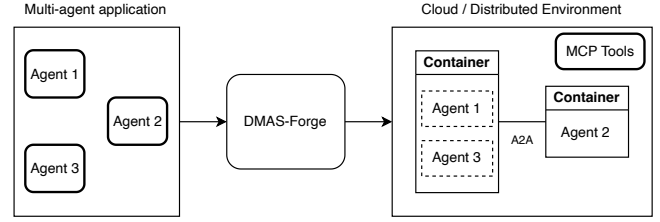

Figure 1: Objective of DMAS-Forge.

the new sub-graphs while preserving the original orchestration logic; *(iv)* in doing so, implement protocol adapters to translate between LangGraph's communication primitives and A2A/ACP primitives; *(v)* scaffold deployment-specific infrastructure to run the distributed system on the desired environment–e.g., in Linux containers, serverless lambdas, Kagent resources [20], VMs in E2B [11], or Temporal workers [8].

Limitations. Unfortunately, this manual effort must be repeated for every new communication protocol, changes to the dependencies between agents, and new deployment environment–*i.e.*, it is not a one-time cost. The problem is exacerbated by the rapid proliferation of different solutions in the field, which forces developers to repeatedly re-engineer their applications to keep up with the latest trends. Further, it hinders opportunities for optimization. For example, step *(i)* could be optimized based on runtime profiling of communication costs and computational load. Similarly, optimization opportunities exist for other design choices, including which communication protocol to use or which runtime environment. Therefore, automation is fundamental, since one might need to iteratively re-deploy and profile a DMAS for each design choice.

3 Our approach & vision

We envision a "*write-once, deploy-everywhere*" paradigm, where developers write multi-agent applications once and our framework flexibly ports them to different execution environments, compliant with communication protocols. This vision is summarized in Fig. 1.

3.1 DMAS-Forge compiler

We propose a compiler-based approach that enables clean separation of core agent logic from the underlying communication protocols and deployment infrastructure. Our key insight is that the core computation model of an AI application is completely orthogonal to how and where the computation is performed. This clean separation allows application developers to provide their structural agentic workflow independently from the deployment choices, and allows developers to plug-and-play any deployment choice in the future.

Our compiler expects two inputs, highlighted in Fig. 2.

The structural agentic workflow (ln.13): a graph-like computation workflow that includes the various agent implementations with their corresponding tools and the connections between the different agent computations. We follow the computational graph model of an existing AI programming framework, LangGraph, that represents the nodes in the graphs as agents and the edges between the nodes as agent communications.

The deployment specification (ln.8): this is the deployment information for the computation graph including how each agent/tool will run (i.e., process, container, serverless), how different agents/tools will connect and communicate (i.e., choice of protocol), runtime constraints (e.g., hardware type, number of replicas, access policies), and the underlying LLM for each agent.

The compiler automatically generates the necessary glue code for deploying the application as described by the provided workflow and the deployment specification. It automatically bakes in the code to ensure that connected agents/tools comply with the user-specified communication protocol. Additionally, depending on the deployment targets, it automatically generates the necessary configuration files, including Dockerfiles and Kubernetes configurations to enforce access policies, ensure the binding to the desired LLM type, etc.

Our compiler-based approach is inspired by previous approaches to flexibly configure microservices [1], and flexibly support distributed deployment of computation graphs in different environments [21].

4 Prototype

We showcase an initial prototype[1] in Go language, named DMAS-Forge. To build DMAS-Forge, we extend the Blueprint microservices compiler [1] to support AI applications. We chose Blueprint as it is compatible with our computational model and it already provides a large array of deployment choices and infrastructure components that can be leveraged.

Programming interfaces. Table 1 shows extensions we add to Blueprint to support the new requirements. First, we extend Blueprint's *workflow API* to allow users to easily implement their applications as structured workflows, similar to LangGraph. Second, we offer a new *wiring API*, through which developers can input the deployment specifications to DMAS-Forge.

In our prototype, this is achieved by implementing several new Blueprint plugins: (1) An `Agent` plugin, to transparently create agents and connect them with any OpenAI-compatible model. (2) A `vLLM` plugin, to automate model deployment in vLLM [22]. (3) A `kagent` plugin, for supporting distributed

---

[1]Available at https://github.com/vaastav/DMAS_forge

```
1  import DMASForge/http
2  import DMASForge/linuxcontainer
3  import DMASForge/vllm
4  import DMASForge/openai
5
6  spec = DMASForge.NewSpec()
7
8  // Deployment specification
9  def DeployAgent(agent):
10    http.Deploy(spec, agent)
11    linuxcontainer.Deploy(spec, agent)
12
13  // Structural agentic workflow
14  model = vllm.Model("gpt-4o")
15  weather = openai.Agent(model, prompt="...")
16  news = openai.Agent(model, prompt="...")
17  weather.Connect(news)
18
19  // Deployment
20  DeployAgent(news)
21  DeployAgent(weather)
```

Figure 2: Two-agent application in DMAS-Forge.

Workflow Type Extensions	
`Agent`	Specifies an Agent
`.Connect(agent)`	Connects two agents
`.AddTool(tool)`	Adds a tool
`Model`	Specifies a Model
`Tool`	Specifies a Tool that an Agent can use
Wiring API	
`NewSpec()`	Creates a new DMAS-Forge spec
`openai.Agent()`	New instance of an openai Agent
`vllm.Model(name)`	Launches a new model instance
`NewTool()`	New instance of a tool
`a2a.Deploy()`	Deploy Agent with A2A
`mcp.Deploy()`	Deploy Tool using MCP
`kagent.Deploy()`	Deploy via kagent

Table 1: DMAS-Forge API overview.

deployments with kagent [20], an emerging Kubernetes-native framework for AI agents that provides Custom Resource Definitions (CRDs) for agents, tools and models.

Lastly, we leverage the RPC-over-HTTP Blueprint plugin as an example of inter-agent communication protocol in our prototype. Fig. 2 shows a complete example of the use of these plugins to deploy two agents as Linux containers.
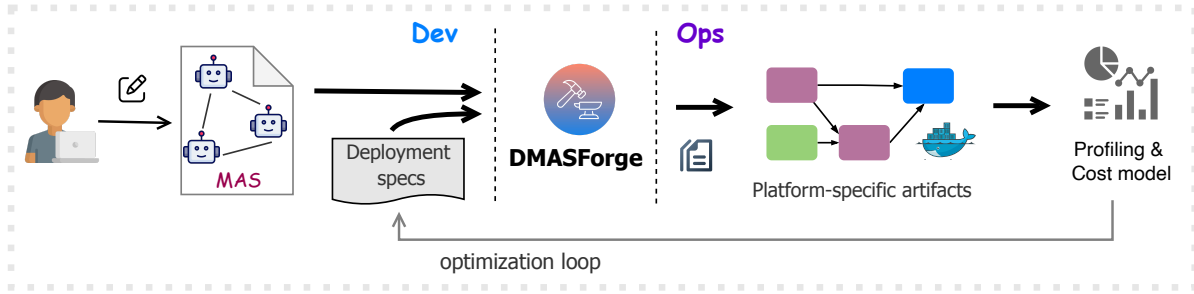
Figure 3: DMAS-Forge as enabler of a closed-loop optimization pipeline for MAS deployment.

5 Discussion and future work

DMAS-Forge is a compiler-based framework that transforms multi-agent applications into a distributed deployment with the effort of one click. It targets diverse runtime environments and aims at generating the necessary glue code for each of them. We presented an initial prototype that supports Linux containers. We plan to extend to other runtime environment, as well as showcasing its benefits for (at least) the following areas.

Optimization pipelines. A key area for improvement would be to streamline the creation of closed-loop optimization pipelines. Applications can be written, deployed, and profiled for communication patterns and resource usage, then automatically restructured for better performance. For example, LangGraph's communication structures (such as sequential, parallel, or conditional flows) can be optimized at runtime without requiring manual redeployment. This pipeline is illustrated in Fig. 3.

This process relies on measuring and modeling communication patterns, performance, and cost-efficiency across different infrastructures and deployment environments. It also involves decisions about whether agents should be co-located or separated into different containers. A key question is whether alternative communication patterns or protocols might better suit the application, given its deployment, latency, and performance constraints.

With these improvements, users no longer need to manually specify the number of replicas or communication protocols. Instead, the framework can make those choices automatically, based on the available infrastructure and performance requirements.

Automatic security boundaries. Another area of future work is to explore means of re-adjusting security policies and determining the least amount of authorization for a DMAS container (or other deployed instance), based on the agents and tools it is running. This is particularly useful when agents are co-located or disaggregated across containers by the optimization pipeline. Automatically determining the security boundaries becomes necessary to minimize the attack surface of each deployed container.

Engineering challenges. Future work should align DMAS-Forge with the complete features of current agentic frameworks and extend them. A key open question is whether to build comprehensive agentic capabilities directly into DMAS-Forge or to serve as an abstraction layer over existing frameworks. The latter approach would require techniques (e.g., monkey patching) to redirect framework-specific communication primitives to protocol-compliant distributed channels and avoid substantial re-engineering effort.

References

[1] Vaastav Anand, Deepak Garg, Antoine Kaufmann, and Jonathan Mace. 2023. Blueprint: A Toolchain for Highly-Reconfigurable Microservice Applications. In *SOSP*. Association for Computing Machinery.

[2] Anthropic. 2024. Model Context Protocol (MCP). https://docs.anthropic.com/en/docs/mcp.

[3] Muhammad Bilal, Marco Canini, Rodrigo Fonseca, and Rodrigo Rodrigues. 2023. With Great Freedom Comes Great Opportunity: Rethinking Resource Allocation for Serverless Functions. In *Proceedings of EuroSys'23*.

[4] Gohar Irfan Chaudhry, Esha Choukse, Íñigo Goiri, Rodrigo Fonseca, Adam Belay, and Ricardo Bianchini. 2025. Towards Resource-Efficient Compound AI Systems. In *HotOS*. Association for Computing Machinery.

[5] GitHub Community. 2025. Issue #227. https://github.com/i-ambee/acp/discussions/227.

[6] Reddit community. 2025. Reddit thread: running each agent node in LangGraph workflow in its own docker container. https://www.reddit.com/r/LangChain/comments/1i2848r/running_each_agent_node_in_langgraph_workflow_in/.

[7] StackOverflow community. 2024. Deploying Langgraph nodes in separate containers. https://stackoverflow.com/questions/79677336/deploying-langgraph-nodes-in-separate-containers.

[8] Davis Cornelia. 2025. Production-ready agents with the OpenAI Agents SDK + Temporal. https://temporal.io/blog/announcing-openai-agents-sdk-integration.

[9] Microsoft Corporation. 2024. AutoGen. https://microsoft.github.io/autogen/stable//index.html.

[10] Yilun Du, Shuang Li, Antonio Torralba, Joshua B. Tenenbaum, and Igor Mordatch. 2023. Improving Factuality and Reasoning in Language Models through Multiagent Debate. arXiv:2305.14325

[11] Inc. FoundryLabs. 2025. E2B: AI Sandboxes for Automation Agents. https://e2b.dev/docs.

[12] CrewAI Inc. 2024. CrewAI. https://www.crewai.com.

[13] LangChain Inc. 2024. LangGraph. https://langchain-ai.github.io/langgraph/.

[14] LangChain Inc. 2025. LangGraph: Workflows and Agents. https://langchain-ai.github.io/langgraph/tutorials/workflows/#set-up.

[15] Zhenkun Li, Lingyao Li, Shuhang Lin, and Yongfeng Zhang. 2025. Know the Ropes: A Heuristic Strategy for LLM-based Multi-Agent System Design. arXiv:2505.16979 [cs.AI] https://arxiv.org/abs/2505.16979

[16] LlamaIndex. 2024. LlamaIndex. https://www.llamaindex.ai.

[17] Google LLC. 2025. Agent2Agent (A2A) Protocol. https://github.com/a2aproject/A2A.

[18] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Sean Welleck, Bodhisattwa Prasad Majumder, Shashank Gupta, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-refine: Iterative refinement with self-feedback. In *NeurIPS*. Curran Associates Inc.

[19] Agno maintainers. 2025. Agno Teams. https://docs.agno.com/concepts/teams/introduction.

[20] Solo.io. 2025. kagent: Cloud Native Agentic AI Framework. https://kagent.dev/.

[21] TensorFlow. 2025. TensorFlow. https://www.tensorflow.org.

[22] vLLM (Berkley). 2025. vLLM: Easy, fast, and cheap LLM serving for everyone. https://docs.vllm.ai.

[23] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V. Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-consistency improves chain of thought reasoning in language models. In *ICLR*. OpenReview.net.

[24] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *NeurIPS*. Curran Associates Inc.

[25] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. arXiv:2210.03629