SlackAgents: Scalable Collaboration of AI Agents in Workspaces

Weiran Yao[*], **Zhiwei Liu**[*], **Zuxin Liu**, **Juntao Tan**, **Jianguo Zhang**, **Frank Wang**, **Sukhandeep Nahal**, **Huan Wang**, **Shelby Heinecke**, **Silvio Savarese** and **Caiming Xiong**

Salesforce AI Research

***Equal contributions**

Abstract

The integration of AI agents into organizational workflows remains a significant challenge, limiting their impact on daily business operations despite the rapid progress in agent libraries and large language models. We introduce **SlackAgents**, a scalable **multi-agent** framework that natively embeds AI agents into Slack, the leading enterprise collaboration platform. **SlackAgents** enables seamless agent-to-agent and agent-to-human collaboration, leveraging Slack's flexible messaging infrastructure to orchestrate complex workflows and automate real-world tasks. Our architecture supports modular agent types, distributed orchestration, and intuitive user interfaces, allowing organizations to rapidly deploy, customize, and scale agent teams across diverse use cases. We present several typical real-world deployment cases to demonstrate the effectiveness of **SlackAgents** in bridging the last-mile gap for enterprise AI adoption, unlocking new opportunities for intelligent automation in the workplace.

1 Introduction

AI agents (Chase, 2022; Liu, 2022; Wu et al., 2024; Liu et al., 2024) are autonomous software entities designed to perform specific tasks, make decisions, and interact with both humans and other agents. Leveraging advancements in large language models (LLMs), these agents can now learn, plan and reason, enabling them not only to understand and generate human-like conversations but also to execute actions on our behalf across both real-world and digital environments.

Businesses increasingly adopt AI agents to automate operations, enhance efficiency, and support decision-making. Existing open-source frameworks like LangChain (Chase, 2022) and LlamaIndex (Liu, 2022) offer flexible and integrative capabilities but often fail to transition beyond prototypes due to difficulties integrating them into daily workflows—commonly known as the *last-mile problem* (Wikipedia, 2024). As a result, there is a clear demand for frameworks specifically designed for immediate and practical deployment within workplace settings.

Additionally, managing and coordinating large-scale, multi-agent systems remains challenging, as illustrated by Salesforce's ambitious goal to deploy *one billion agents* by 2025 (Salesforce, 2024). Current multi-agent platforms, including AutoGen (Wu et al., 2024), CAMEL (Li et al., 2023), and CrewAI (Moura, 2024), lack sufficient capabilities for robust communication and collaboration at scale.

To address these limitations, this paper introduces SLACKAGENTS , a scalable multi-agent library integrated directly with Slack, the leading corporate collaboration platform (Slack, 2024). By leveraging Slack's extensive messaging infrastructure, our framework facilitates integrated and scalable collaboration, allowing agents to communicate and orchestrate tasks efficiently through Slack channels and threads. It also enables seamless workspace automation, empowering agents to directly utilize Slack's native features—including canvases, bookmarks, workflows, and notifications—to automate and streamline everyday work tasks. Furthermore, our framework supports immediate deployment and continuous improvement, as agents integrate natively into existing workflows, enabling rapid deployment, iterative enhancements through routine usage, and straightforward distribution across multiple Slack workspaces.

Key features of SLACKAGENTS include:

- **Modular and Scalable Architecture**: Designed for flexibility, each agent operates independently as a Slack application, allowing easy scaling and customization.
- **Efficient Distributed Orchestration**: Utilizing Slack's messaging system, agents dynamically manage communications, enabling seamless col-

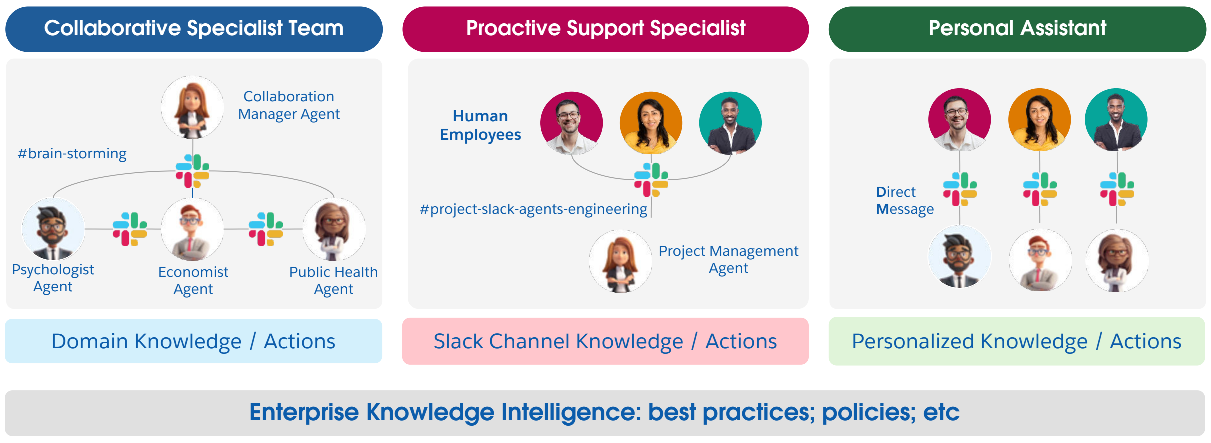

Collaborative Specialist Team
Proactive Support Specialist
Personal Assistant
Collaboration Manager Agent
#brain-storming
Psychologist Agent
Economist Agent
Public Health Agent
Human Employees
#project-slack-agents-engineering
Project Management Agent
Direct Message
Domain Knowledge / Actions
Slack Channel Knowledge / Actions
Personalized Knowledge / Actions
Enterprise Knowledge Intelligence: best practices; policies; etc

Figure 1: Integrating AI agents with Slack's features and visual components unlocks a variety of applications for workflow automation involving multiple agents: **(1) Collaborative specialist team** for enhancing business workflows and foster decision making from diverse perspectives; **(2) Proactive support specialist** with in-depth knowledge of specific Slack channels (e.g., a scrum team channel) to assist team members proactively, and (3) **Personal assistant** that is deeply familiar with individual users, offering tailored support and solving private tasks efficiently.

laboration and knowledge sharing across agents.

- **User-friendly Interfaces**: SlackAgent offers intuitive interactions through Slack chatting interfaces, interactive dashboards for monitoring and configuring agents, and natural language commands for creating and managing agents without technical expertise.
- **Flexible Agent Types**: Provides customizable agent types—Assistant, Workflow, and Proactive agents—which developers can combine to build tailored solutions from individual productivity assistants to complex multi-agent teams.

2 SLACKAGENTS Architecture

SLACKAGENTS is designed to enable seamless multi-agent collaboration and intelligent automation within Slack workspaces. The architecture introduces a unified agent framework, supporting diverse interaction patterns through modular agent types and extensible tool integration. This section details the fundamental agent classes, communication protocols, and interface components that underpin the system, illustrating how these elements collectively facilitate robust, scalable teamwork in enterprise messaging environments.

2.1 Core Agent Types

The framework defines three primary agent types: the *Assistant Agent*, which performs tasks in multi-turn conversations using tools such as functions, APIs, external libraries, and code interpreters; the *Workflow Agent*, which manages multi-stage workflows by maintaining state and guiding sequential goal completion; and the *Proactive Agent*, which maintains persistent awareness of context and autonomously offers support when needed. Each agent is deployed as a standalone Slack app, equipped with listeners and message-sending capabilities, enabling both user-agent and inter-agent collaboration through Slack channels and threads (see Figure 2). The detailed implementation of agents is in Appendix B.4.

2.2 Scalable Multi-Agent Collaboration

Let us assume that for now we have already built multiple individual AI agents with SLACKAGENTS framework, *agent a, b, and c*, whatever an Assistant or Workflow agent. In this section, we walk you through how users can further enable multi-agent collaboration between them.

SLACKAGENTS defines the collaboration of *agent a,b and c* when initializing each individual agent as follows:

```
from slackagents import SlackAssistant

agent_a = SlackAssistant(
    name=name_agent_a,
    desc=desc_agent_a,
    system_prompt=...,
    tools=[...],
    slack_bot_token=...
    colleagues={
        id_agent_b: {"name": name_agent_b, "description": desc_agent_b},
        id_agent_c: {"name": name_agent_c, "description": desc_agent_c},
    } # define the multi-agent collaboration
)
```

where the *colleagues* key argument defines the possible collaboration to its collaborative agents *a* and

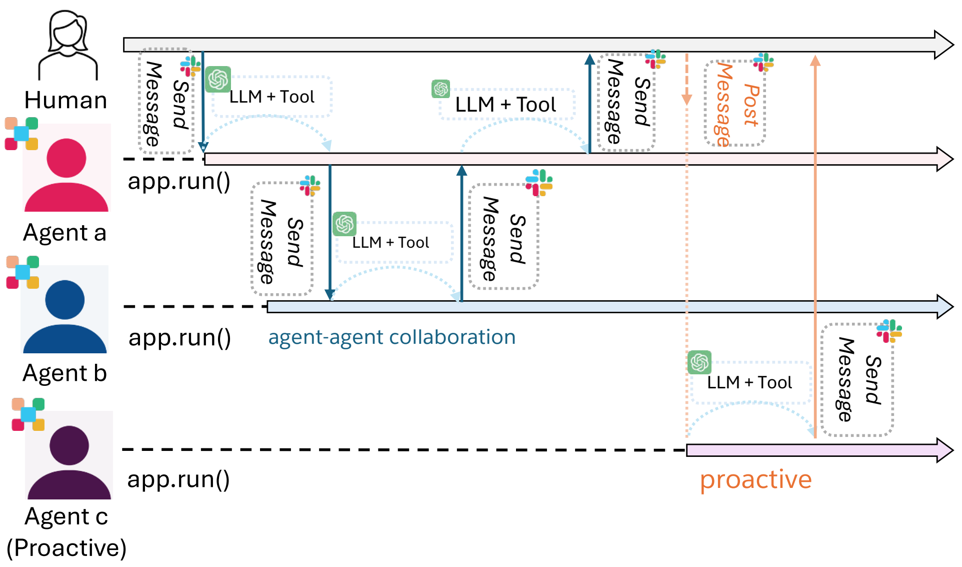


Figure 2: Illustration of how SLACKAGENTS can program a multi-agent conversation in a scalable, decentralized way. The traces of the operation of a multi-agent team, step by step. When *human* sends the first message directly to *agent a*, *agent a* decides to ask the help from *agent b* and respond back to human, which shows the agent-agent collaboration via SLACKAGENTS . When human posts a message under the thread without mentioning a specific agent, a proactive agent 3 listens this message and chimes in to respond back, which illustrates the proactive mode. Proactive agent 3 may also respond towards the message sending from another agent.

b. It is worth noting that we use the Slack User ID as the id of each agent, such as *id_agent_b*.

After we define all agents, SLACKAGENTS runs each agent as a individual Slack App[1]. Built upon the scalable message handling backend of Slack, SLACKAGENTS provides three types of Slack message handlers, which are Direct Message Handler, Channel Message Handler and Proactive Message Handler to process different chatting modes.

Direct Message Handler supports sending direct message to an agent through Slack app direct message interface. One could initialize a agent from SLACKAGENTS and register this agent to listening direct messages. *Channel Message Handler* in SLACKAGENTS supports @ mention in a slack channel. In this way, all agents and humans are able to send message to any specific agent, which enables team collaboration. It also enables the agent to reach out to others for help. Note that we could include either another agent or another human for help. *Proactive Message Handler* provides a proactive way to monitor all messages and step in to provide help only when needed. We show this process as in Figure 2. Once the proactive agent is activated in the backend, it monitors all messages within that thread. The agent determines whether to participate in the discussion based on its capabilities from system prompts and available tools. When the agent identifies a need or request for support, it automatically employs its tools and engages in the conversation. Detailed implementation fo those message hanlders are in Appendix B.1.

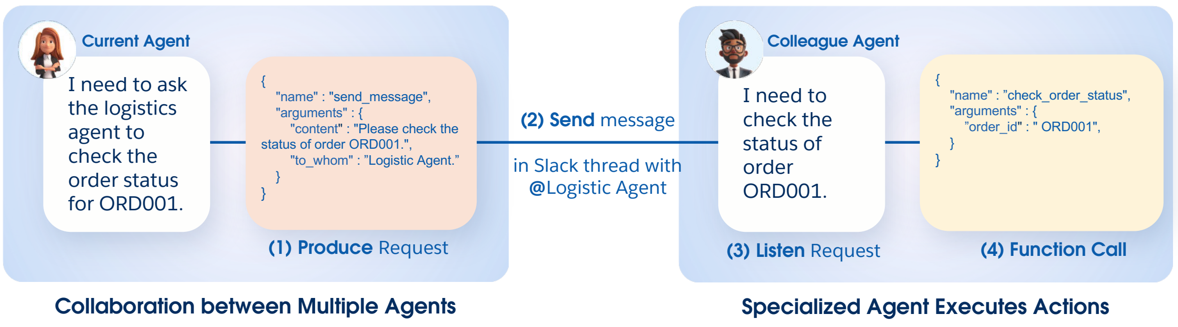
2.3 Collaboration Protocol

The SLACKASSISTANT class represents a conversational agent designed for multi-agent collaboration within Slack channel environments. This agent leverages Slack-specific functionalities, making it particularly suitable for team-based interactions.

The core of the multi-agent collaboration in SLACKAGENTS is for the current agent to **(1) produce** a message that contains a request for assistance with @ mention of the chosen agents or human from a pre-defined colleague list, **(2) send** the message to the colleague(s) in a dedicated session, and **(3) listen** for colleagues' responses in the session. Compared with the "handoff" strategy in OpenAI swarm (OpenAI, 2024), which hands off all messages to another agent by swapping system prompt and tools, our collaboration strategy is decentralized, asynchronous, and scalable by leveraging Slack-specific functionalities, and importantly, same as how human workers collaborate in Slack channels by looping in colleagues for help.

We use function calling as the standard protocol

[1]https://api.slack.com/docs/apps


Current Agent
I need to ask the logistics agent to check the order status for ORD001.
(1) Produce Request
(2) Send message
in Slack thread with @Logistic Agent
Colleague Agent
I need to check the status of order ORD001.
(3) Listen Request
(4) Function Call
Collaboration between Multiple Agents
Specialized Agent Executes Actions

```
{
    "name" : "send_message",
    "arguments" : {
        "content" : "Please check the status of order ORD001.",
        "to_whom" : "Logistic Agent."
    }
}
```

```
{
    "name" : "check_order_status",
    "arguments" : {
        "order_id" : " ORD001",
    }
}
```

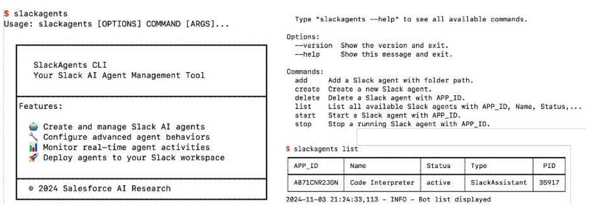
Figure 3: Multi-agent communication protocol. This example illustrates how two agents can collaborate in a dedicated collaboration session in Slack thread.

for producing collaboration requests. Three Slack conversation tools, SEND_MESSAGE, WAIT, and GET_THREAD_HISTORY are added to each SLACKASSISTANT to facilitate multi-agent collaboration in a Slack session. As illustrated in Figure 9, when the current agent in the session decides that the current conversation is out of its capabilities but falls into its team members' roles and capabilities, it will execute SEND_MESSAGE functions, which sends a `"<@to_whom>" + content` message in the thread, where the colleague agent consumes this request through listener and chimes in to help. After sending a request, the agent executes the WAIT function, ending its tool request loop and awaiting responses from colleague agents. The proactive agent behavior is mainly achieved through this function. More details can be found in Sec. B.4.3. All agents are equipped with GET_THREAD_HISTORY by default to obtain the past messages in the thread, in case the request message which has been sent is not informative enough. More detailed implementation are in Appendix B.2.

2.4 User Interface

2.4.1 Command Line Interface (CLI)

The SlackAgents Command-Line Interface (CLI) provides a comprehensive set of commands for managing AI agents within Slack workspaces. This specification details the command structure, available operations, and implementation guidelines.



```
$ slackagents
Usage: slackagents [OPTIONS] COMMAND [ARGS]...

┌──────────────────────────────────────────┐
│                                          │
│   SlackAgents CLI                        │
│   Your Slack AI Agent Management Tool    │
│                                          │
├──────────────────────────────────────────┤
│Features:                                 │
│                                          │
│  🤖 Create and manage Slack AI agents     │
│  🔧 Configure advanced agent behaviors    │
│  📊 Monitor real-time agent activities    │
│  🚀 Deploy agents to your Slack workspace │
│                                          │
├──────────────────────────────────────────┤
│  © 2024 Salesforce AI Research           │
└──────────────────────────────────────────┘

  Type "slackagents --help" to see all available commands.

Options:
  --version  Show the version and exit.
  --help     Show this message and exit.

Commands:
  add     Add a Slack agent with folder path.
  create  Create a new Slack agent.
  delete  Delete a Slack agent with APP_ID.
  list    List all available Slack agents with APP_ID, Name, Status,...
  start   Start a Slack agent with APP_ID.
  stop    Stop a running Slack agent with APP_ID.

$ slackagents list
┌─────────────┬──────────────────┬─────────┬────────────────┬───────┐
│ APP_ID      │ Name             │ Status  │ Type           │  PID  │
├─────────────┼──────────────────┼─────────┼────────────────┼───────┤
│ A071CNR2JGN │ Code Interpreter │ active  │ SlackAssistant │ 35917 │
└─────────────┴──────────────────┴─────────┴────────────────┴───────┘
2024-11-03 21:24:33,113 - INFO - Bot list displayed
```

Figure 4: SLACKAGENTS Command Line Interface.

Command Structure The CLI follows a standardized command pattern:

```
slackagents [COMMAND] [OPTIONS]
```

Core Commands

`create` Interactive wizard for new agent creation

`add [FOLDER_PATH]` Register existing agent from specified directory

`list` Display agents with APP_ID, name, status, and type (`–verbose` for details)

`start [APP_ID]` Launch specified agent

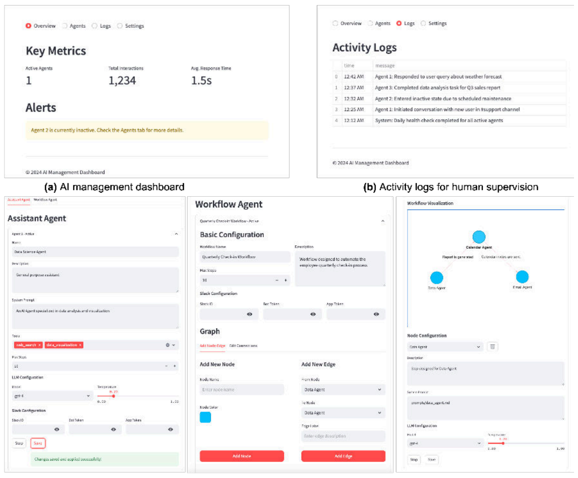
`stop [APP_ID]` Terminate specified agent

`delete [APP_ID]` Remove agent and associated resources

To effectively use the CLI, begin operations with `slackagents list` to view available agents. Access detailed documentation for any command using `–help`. Keep track of APP_ID records for agent management operations, noting that all commands are case-sensitive and perform input validation before execution.

2.4.2 Admin Dashboard

The admin dashboard serves as a central hub for managing AI agents, tools, and workflows. It provides real-time visibility into agent performance, task execution, and system status. Key features include user management, tool configuration, and detailed analytics on agent interactions. Admins can monitor agent activity, adjust tool settings, and troubleshoot issues directly from the dashboard.


Overview · Agents · Logs · Settings

Key Metrics

Active Agents	Total Interactions	Avg. Response Time
1	1,234	1.5s

Alerts

Agent 2 is currently inactive. Check the Agents tab for more details.

© 2024 AI Management Dashboard

(a) AI management dashboard

Overview · Agents · Logs · Settings

Activity Logs

	time	message
0	12:42 AM	Agent 1: Responded to user query about weather forecast
1	12:37 AM	Agent 3: Completed data analysis task for Q3 sales report
2	12:32 AM	Agent 2: Entered inactive state due to scheduled maintenance
3	12:25 AM	Agent 1: Initiated conversation with new user in #support channel
4	12:12 AM	System: Daily health check completed for all active agents

© 2024 AI Management Dashboard

(b) Activity logs for human supervision

Assistant Agent

Agent 1 - Active

Name: Data Science Agent

Description: General purpose assistant

System Prompt: An AI Agent specialized in data analysis and visualization

Tools: web_search, data_visualization

Max Steps: 10

LLM Configuration — Model: gpt-4; Temperature: 0.20

Slack Configuration — Slack ID, Bot Token, App Token

Changes saved and applied successfully!

Workflow Agent

Quarterly Check-in Workflow - Active

Basic Configuration

Workflow Name: Quarterly Check-in Workflow

Description: Workflow designed to automate the employee quarterly check-in process.

Max Steps: 10

Slack Configuration — Slack ID, Bot Token, App Token

Graph

Add New Node — Node Name: Enter node name; Node Color

Add New Edge — From Node: Data Agent; To Node: Data Agent; Edge Label: Enter edge description

Workflow Visualization

Node Configuration: Data Agent

Description: Step designed for Data Agent

System Prompt: prompts/data_agent.md

LLM Configuration — Model: gpt-4; Temperature: 0.20

(c) Runtime configuration for assistant and workflow agent

Figure 5: AI Mangement Dashboard.

Configuration of AI agents at runtime lets the admin to modify a Slack agent's behaviors from the Admin Dashboard UI, without needing to restart the application.

3 Applications of SLACKAGENTS

In this section, we present a comprehensive exploration of native applications enabled by SLACKAGENTS to automate our daily work and improve decision-making processes, which aligns the video demoing **proactive agent**, **code interpreter agent**, and **multi-agent customer service team**.

3.1 Proactive Assistant - AgentPro

AgentPro is an intelligent and context-aware Slack assistant designed to enhance workplace collaboration. Unlike traditional chatbots, AgentPro seamlessly integrates into team discussions, participating only when necessary to provide valuable insights or assistance. It continuously monitors conversations, identifying key topics or moments where its expertise can contribute, and steps in either when directly mentioned or when it recognizes an opportunity to add meaningful value. An illustration of how proactive agent monitors the message in slack channel is in Figure 6.

A key strength of AgentPro is its ability to engage proactively without being disruptive. It avoids unnecessary interruptions during casual exchanges, choosing instead to focus on moments where its input is most valuable. This thoughtful engagement helps maintain the flow of natural conversations while ensuring critical questions are addressed.

By transforming the traditional chatbot paradigm into a more natural and dynamic collaborator, AgentPro enhances workplace interactions. Its ability to intelligently balance when to engage and when to stay silent makes it an invaluable tool for streamlining workflows, fostering effective communication, and improving team productivity.

3.2 Code Interpreter Agent

The *Code Interpreter Agent* is deployed as a Slack direct-message (DM) application that couples LLM dialogue with a Python sandbox runtime. The runtime exposes the essential scientific-Python stack—`numpy`, `pandas`, `scikit-learn`, `matplotlib`, and `seaborn` behind a restricted I/O boundary. Whenever a user submits a request—e.g., "*Please explore the IRIS dataset and visualise the class separability*"—the agent enters a five-stage loop: *plan*, *generate code*, *confirm*, *execute*, and *recap*. It first decomposes the goal into minimal steps and presents this plan for approval. Only after the user signs off does the agent emit Markdown-formatted Python code, run it inside a containerized kernel, and surface the results—figures, tables, or metrics—before advancing to the next step. This human-in-the-loop protocol yields an auditable trail of intentions, code, and outputs while preventing unintended or malicious execution.

3.3 Autonomous Customer Service Team

To optimize customer support operations for an e-commerce company, we developed a multi-agent system with SLACKAGENTS comprising three specialized agents: the Customer Service Agent, the Sales Agent, and the Logistics Agent. Each agent plays a distinct role within the system, designed to address specific aspects of customer requests while seamlessly collaborate to provide service.

The Customer Service Agent acts as the primary interface with customers, handling inquiries related to order status, cancellations, modifications, and product recommendations, as shown in Figure 7. This agent ensures that every customer message is acknowledged and processed promptly. For order-related tasks, the agent collaborates with either the Logistics Agent or the Sales Agent, depending on the nature of the request. For example, when a customer inquires about an order status, the *Customer Service Agent* collects the order ID and queries the Logistics Agent to check the order status. Similarly, product requests are relayed to the Sales Agent.

The Logistics Agent oversees order status and shipping logistics. It provides detailed updates on

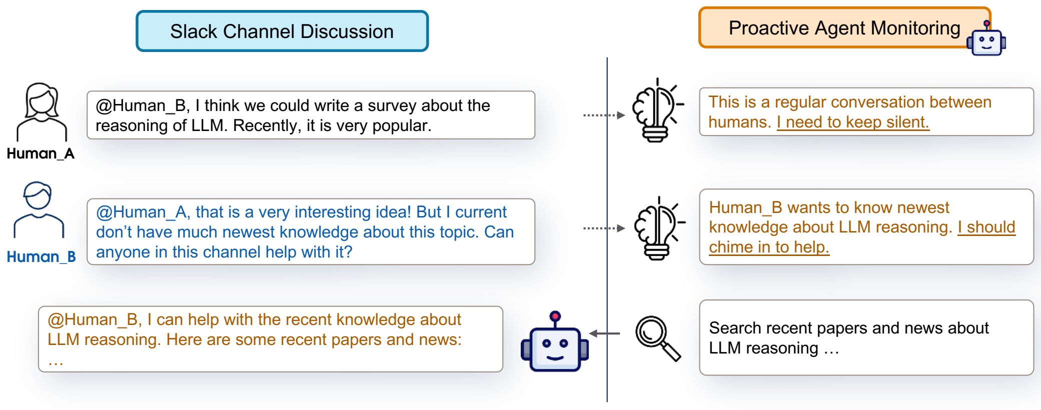

Slack Channel Discussion
Proactive Agent Monitoring
Human_A
@Human_B, I think we could write a survey about the reasoning of LLM. Recently, it is very popular.
Human_B
@Human_A, that is a very interesting idea! But I current don't have much newest knowledge about this topic. Can anyone in this channel help with it?
@Human_B, I can help with the recent knowledge about LLM reasoning. Here are some recent papers and news: …
This is a regular conversation between humans. I need to keep silent.
Human_B wants to know newest knowledge about LLM reasoning. I should chime in to help.
Search recent papers and news about LLM reasoning …

Figure 6: Proactive channel assistant monitors the messages in slack channel and decide to keep silent or chime in to help. Its decision making is conditioned on the compound of tools, prompts and messages.

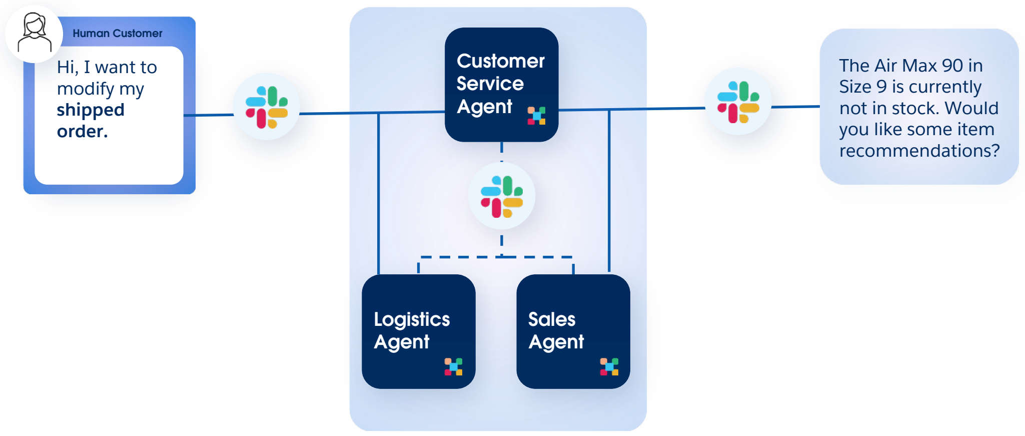

Human Customer
Hi, I want to modify my shipped order.
Customer Service Agent
Logistics Agent
Sales Agent
The Air Max 90 in Size 9 is currently not in stock. Would you like some item recommendations?

Figure 7: A customer service team for E-commerce. Customer Service Agent coordinates logistics agent and sales agent. Human customer can also directly send message to each agent.

shipping statuses, estimated delivery times, and any potential delays. The Sales Agent specializes in managing product-related interactions, particularly for recommendation. This agent is responsible for inventory checks, product recommendations, and order creation or modification. The strength lies in its collaboration. SLACKAGENTS advances multi-agent application development in its flexible collaboration work styles. Human customer could ask customer service agent to relay the information to other agents. Or a human customer could ask directly to any specific agent for help.

Additional applications are also avaiable in our source code and appendix C.

4 Conclusion

In this paper, we introduced SlackAgents, a robust framework for scalable deployment and collaboration of AI agents within enterprise workspaces. By tightly integrating with Slack, SlackAgents enables seamless orchestration, communication, and automation, which significantly advances in realizing the vision of intelligent, collaborative, and adaptive workplace automation. Our modular architecture supports a variety of agent types and interaction modes, facilitating flexible deployment and efficient teamwork across diverse business scenarios. Through real-world applications such as proactive assistance, code interpretation, and multi-agent customer service, we demonstrate the practical value and transformative potential of AI-powered collaboration in modern organizations.

Limitations

While SLACKAGENTS provides significant advantages for workplace AI agent deployment, several limitations should be acknowledged:

- Platform dependency on Slack infrastructure
- Scalability constraints inherent to the underlying platform
- Learning curve for organizations new to multi-agent systems
- Integration complexity with highly specialized enterprise systems

These limitations represent areas for future improvement and development of the framework.

Acknowledgments

We thank the Salesforce AI Research team for their support and collaboration in developing this framework. We also acknowledge the valuable feedback from early adopters and beta testers who helped refine the platform's capabilities and user experience.

References

Harrison Chase. 2022. LangChain.

Guohao Li, Hasan Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. 2023. Camel: Communicative agents for" mind" exploration of large language model society. *Advances in Neural Information Processing Systems*, 36:51991–52008.

Jerry Liu. 2022. LlamaIndex.

Zhiwei Liu, Weiran Yao, Jianguo Zhang, Liangwei Yang, Zuxin Liu, Juntao Tan, Prafulla K Choubey, Tian Lan, Jason Wu, Huan Wang, and 1 others. 2024. Agentlite: A lightweight library for building and advancing task-oriented llm agent system. *arXiv preprint arXiv:2402.15538*.

Joao Moura. 2024. CrewAI.

OpenAI. 2024. Swarm: An educational framework exploring ergonomic, lightweight multi-agent orchestration.

Salesforce. 2024. Salesforce unveils agentforce–what ai was meant to be. Accessed on Oct 10, 2024.

Slack. 2024. Announcing agents and ai innovations in slack.

Wikipedia. 2024. Last mile (transportation).

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Eric Zhu, Li Jiang, Shaokun Zhang, Xiaoyun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. 2024. AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation Framework.

A Recommended Guidelines for Using SlackAgents Library

Here we introduce the basic on-boarding steps of **SlackAgents**. Firstly, you should create a Slack APP and configure the agent with tokens from an APP. After that, you run an agent with **SlackAgents**and interact with it through Slack direct message. The SlackAgents framework streamlines the process of integrating AI agents into Slack. This section details the steps required to create and configure a Slack app, enabling seamless interaction between the app and your agent.

A.1 Step 1: Create Your Slack App

You have two options to create your Slack app: using the Slack Manifest and `slack_sdk` API (recommended) or manually creating the app.

Option 1: Using the Slack Manifest and `slack_sdk` API We recommend setting up your Slack app via the Slack Manifest for its simplicity and automation. An example manifest JSON file is provided in the `app/your_first_slack_assistant` folder. Start by obtaining your App Configuration Access Token and exporting it as an environment variable:

```
export SLACK_APP_CONFIG_TOKEN=xoxe.xoxp...
```

Next, navigate to the `app/your_first_slack_assistant` folder or copy it and rename it. Update the `agent_config.json` file with your agent's name and description. These values will populate fields in the `example_manifest.json` file. Use the `create_slack_app.py` script to create the app by running:

```
cd app/your_first_slack_assistant
python create_slack_app.py \
--manifest example_manifest.json
```

The created app will appear in the Slack App Dashboard. The script also saves the App ID in the `slack_bolt_id.json` file, which will need to be updated with corresponding tokens.

To configure tokens, refer to Steps 1-6 of *Tokens and Installing Apps*[2]. Set up App-level tokens with `connections:write` scope and Bot tokens in the OAuth & Permissions section. Ensure the *Allow users to send Slash commands and messages from the messages tab* option is enabled in the App Home section. Reinstall the app to your workspace. No additional scope modifications are needed when using the example manifest.

After completing these steps, you should have the following credentials:

- **App ID**: Found in the Basic Information section.
- **SLACK_BOT_TOKEN**: Found in the OAuth & Permissions section (e.g., `xoxb-...`).
- **SLACK_APP_TOKEN**: Found in the OAuth & Permissions section (e.g., `xapp-...`).

Option 2: Manually Create the App Follow the Slack Bolt Python Getting Started Guide to manually set up your app. Complete the steps for *Create an App*, *Tokens and Installing Apps*, and *Setting Up Events*. Ensure the following settings:

- *Allow users to send Slash commands and messages from the messages tab*
- *Always Show My Bot as Online* (in the App Home section).

After setup, obtain the App ID, SLACK_BOT_TOKEN, and SLACK_APP_TOKEN from the respective sections of the app's dashboard. Ensure your app has scopes such as `groups:read`, `channels:history`, `channels:read`, `im:history`, `chat:write`, `users:read`, `im:read`, and `app_mentions:read`.

A.2 Step 2: Configure Your Slack Bolt ID and Enable Direct Messaging

Update the `slack_bolt_id.json` file with your App ID, SLACK_BOT_TOKEN, and SLACK_APP_TOKEN. To enable direct messaging, navigate to the Slack App Dashboard, select the *App Home* tab, and enable the *Allow users to send Slash commands and messages from the messages tab* option. Reinstall the app via the OAuth & Permissions section and refresh or restart your Slack client.

Note: If the message window does not appear immediately, try restarting Slack after a few minutes.

[2] `https://tools.slack.dev/bolt-python/getting-started/#tokens-and-installing-apps`

A.3 Step 3: Start Your Agent

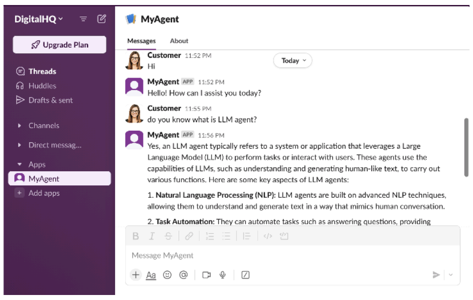
To launch your direct message agent, execute the following command:

```
python my_agent.py -type dm
-agent-config agent_config.json
-bolt-config slack_bolt_id.json
```

You can choose from `type dm` for direct messaging or `assistant` for channel-based interaction where the agent responds to @ mentions. Refer to the tutorial for additional details.

A.4 Step 4: Interact with Your Agent in Slack

Once your agent is running, search for it under *Apps* or the General Search Window Bar in Slack. Send messages to your app, and your agent will respond accordingly.



Customer 11:52 PM
Hi

MyAgent APP 11:52 PM
Hello! How can I assist you today?

Customer 11:55 PM
do you know what is LLM agent?

MyAgent APP 11:56 PM
Yes, an LLM agent typically refers to a system or application that leverages a Large Language Model (LLM) to perform tasks or interact with users. These agents use the capabilities of LLMs, such as understanding and generating human-like text, to carry out various functions. Here are some key aspects of LLM agents:

1. **Natural Language Processing (NLP):** LLM agents are built on advanced NLP techniques, allowing them to understand and generate text in a way that mimics human conversation.

2. **Task Automation:** They can automate tasks such as answering questions, providing

Figure 8: The direct message interface when chatting with **SlackAgents**in Slack.

B Implementation Details

B.1 Message Handlers: Consume messages from subscribed sessions

After we define all agents, SLACKAGENTS runs each agent as a individual Slack App[3]. Built upon the scalable message handling backend of Slack, SLACKAGENTS provides three types of Slack message listeners for humans and all agents to work in a workspace.

Direct Message Handler. SLACKAGENTS supports sending direct message to an agent through Slack app direct message interface. A sample serving code for handling direct message is as follows:

```python
from slack_bolt_id import BOLT_CONFIG
from slackagents import SlackDMHandler
from slackagents import SlackDMAgent

if __name__ == "__main__":
    agent = SlackDMAgent(name=name,desc=desc)
    handler = SlackDMHandler(BOLT_CONFIG, agent)
    handler.run()
```

[3] https://api.slack.com/docs/apps

One could initialize a *SlackDMAgent* from SLACKAGENTS and register this agent to *SlackDMHandler* to listening direct messages.

Channel Message Handler. Besides sending direct messages to those agents, SLACKAGENTS also supports @ mention in a slack channel. In this way, all agents and humans are able to send message to any specific agent, which enables team collaboration. A sample serving code is as follows:

```
from slack_bolt_id import BOLT_CONFIG
from slackagents import SlackChannelHandler
from slackagents import SlackAssistant

if __name__ == "__main__":
    agent = SlackAssistant(name=name, desc=desc, colleagues=colleagues)
    handler = SlackChannelHandler(BOLT_CONFIG, agent)
    handler.run()
```

where one could define an agent with *SlackAssistant* API from SLACKAGENTS and register it into *SlackChannelHandler*. The colleagues are the key argument for this agent to collarabote with. It enables the agent to reach out to others for help. The colleagues could include either another agent or another human for help.

Proactive Message Handler Instead of assigning a message to a agent, SLACKAGENTS also provides a proactive agent to monitor all messages and step in to provide help only when needed. We show this process as in Figure 2 and a sample code is provided as follows:

```
from slack_bolt_id import BOLT_CONFIG
from slackagents import SlackProactiveHandler
from slackagents import SlackAssistant

if __name__ == "__main__":
    agent = SlackAssistant(name=name, desc=desc)
    handler = SlackProactiveHandler(BOLT_CONFIG, agent)
    handler.run()
```

Once the proactive agent is activated in the backend, users can @ mention it in any Slack thread, enabling it to monitor all messages within that thread. The agent determines whether to participate in the discussion based on its capabilities, including system prompts and available tools. When the agent identifies a need for support, it automatically employs its tools and engages in the conversation.

B.1.1 Conversation Tools: Produce Collaboration Request

B.2 Conversation Tools: Produce Collaboration Request

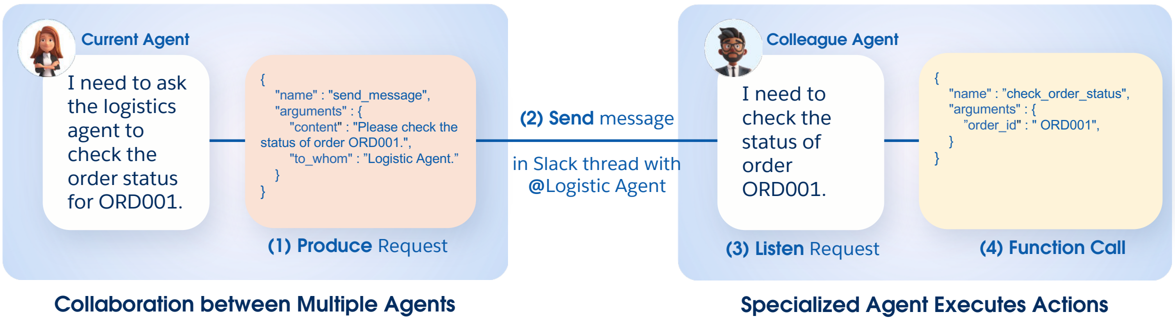
The SLACKASSISTANT class represents a conversational agent designed for multi-agent collaboration within Slack channel environments. This agent leverages Slack-specific functionalities, making it particularly suitable for team-based interactions. The agent maintains awareness of human and agent team members through a colleague system. This feature allows the agent to understand team composition and maintain appropriate context about different team members' roles and capabilities.

Collaboration Strategy The core of the multi-agent collaboration in SLACKAGENTS is for the current agent to **(1) produce** a message that contains a request for assistance with @ mention of the chosen agents or human from a pre-defined colleague list, **(2) send** the message to the colleague(s) in a dedicated session, and **(3) listen** for colleagues' responses in the session. Compared with the "hand-off" strategy in OpenAI swarm (OpenAI, 2024), which hands off all messages to another agent by swapping system prompt and tools, our collaboration strategy is decentralized, asynchronous, and scalable by leveraging Slack-specific functionalities, and importantly, same as how human workers collaborate in Slack channels by looping in colleagues for help in threads.

Collaboration Protocol We use function calling as the standard protocol for producing collaboration requests. Three Slack conversation tools, SEND_MESSAGE, WAIT, and GET_THREAD_HISTORY are added to each SLACKASSISTANT to facilitate multi-agent collaboration in a Slack session.

SEND_MESSAGE. As illustrated in Figure 9, when the current agent in the session decides that the current conversation is out of its capabilities but falls into its team members' roles and capabilities, it will execute SEND_MESSAGE functions, which sends a `"<@to_whom>" + content` message in the thread, where the colleague agent consumes this request through listener and chimes in to help.

```
{
    "type": "function",
    "function": {
        "name": "send_message",
        "description": "Send a message
    to one of your colleagues or to the
    message sender.",
        "parameters": {
            "type": "object",
            "properties": {
                "content": {
                    "type": "string",
                    "description": "The
    content of the message to be sent."
                },
                "to_whom": {
                    "type": "string",
```


Current Agent
(1) Produce Request
(2) Send message
in Slack thread with @Logistic Agent
Colleague Agent
(3) Listen Request
(4) Function Call
Collaboration between Multiple Agents
Specialized Agent Executes Actions

I need to ask the logistics agent to check the order status for ORD001.

```
{
   "name" : "send_message",
   "arguments" : {
      "content" : "Please check the
status of order ORD001.",
      "to_whom" : "Logistic Agent."
   }
}
```

I need to check the status of order ORD001.

```
{
   "name" : "check_order_status",
   "arguments" : {
      "order_id" : " ORD001",
   }
}
```

Figure 9: Multi-agent communication protocol. This example illustrates how two agents can collaborate in a dedicated collaboration session in Slack thread.

```
                        "description": "The
   name of the recipient."
                    }
                },
                "required": ["content", "
   to_whom"],
                "additionalProperties":
   false
            }
        }
}
```

WAIT. After sending a request, the agent executes the WAIT function, ending its tool request loop and awaiting responses from colleague agents. The proactive agent behavior is mainly achieved through this function. More details can be found in Sec. B.4.3.

```
{
    "type": "function",
    "function": {
        "name": "wait",
        "description": "Wait for the
   next message.",
        "parameters": {
            "type": "object",
            "properties": {
                "reason": {
                    "type": "string",
                    "description": "The
   reason for waiting."
                }
            },
            "required": ["reason"],
            "additionalProperties":
   false
        }
    }
}
```

All agents are equipped with GET_THREAD_HISTORY by default to obtain the past messages in the thread, in case the request message which has been sent is not informative enough.

B.3 Collaboration Protocol

We use function calling as the standard protocol for producing collaboration requests. Three Slack conversation tools, SEND_MESSAGE, WAIT, and GET_THREAD_HISTORY are added to each SLACK-ASSISTANT to facilitate multi-agent collaboration in a Slack session. As illustrated in Figure 9, when the current agent in the session decides that the current conversation is out of its capabilities but falls into its team members' roles and capabilities, it will execute SEND_MESSAGE functions, which sends a `"<@to_whom>" + content` message in the thread, where the colleague agent consumes this request through listener and chimes in to help.

```
{
    "type": "function",
    "function": {
        "name": "send_message",
        "description": "Send a message
   to one of your colleagues or to the
   message sender.",
        "parameters": {
            "type": "object",
            "properties": {
                "content": {
                    "type": "string",
                    "description": "The
   content of the message to be sent."
                },
                "to_whom": {
                    "type": "string",
                    "description": "The
   name of the recipient."
                }
            },
            "required": ["content", "
   to_whom"],
            "additionalProperties":
   false
        }
    }
}
```

WAIT. After sending a request, the agent executes the WAIT function, ending its tool request loop and awaiting responses from colleague agents.

The proactive agent behavior is mainly achieved through this function. More details can be found in Sec. B.4.3.

```
{
    "type": "function",
    "function": {
        "name": "wait",
        "description": "Wait for the
    next message.",
        "parameters": {
            "type": "object",
            "properties": {
                "reason": {
                    "type": "string",
                    "description": "The
    reason for waiting."
                }
            },
            "required": ["reason"],
            "additionalProperties":
    false
        }
    }
}
```

All agents are equipped with GET_THREAD_HISTORY by default to obtain the past messages in the thread, in case the request message which has been sent is not informative enough.

B.4 Agent Classes

With communication and collaboration among multiple agents enabled by SLACKAGENTS , the library further provides two key classes for creating custom individual AI agents: **Agent** and **Workflow**. These two classes help define not just the basic behaviors of individual agents but also how an agent act in a cohesive manner to accomplish sequential goals in a principled way.

For tools, SLACKAGENTS supports 4 types of methods to create agent tools, for corresponding use cases. Furthermore, users can also directly load tools from other popular libraries, such as LANGCHAIN, LLAMAINDEX, CREWAI, etc. to develop Slack agents.

B.4.1 Assistant

The SLACKAGENTS library provides a standard way of designing individual agents that can integrate with Slack to perform specific tasks, using various tools and language models. You will need to configure the agent's core attributes, including its name, description, language model, tools, and system prompt. Each of these plays a critical role in defining the agent's behavior.

- NAME: A human-readable name for the agent that will be displayed in Slack.
- DESC: A short description explaining the agent's purpose and functionality.
- LLM: The language model the agent will use for processing natural language input.
- TOOLS: A list of tools that the agent can use to complete tasks with details in Appendix B.4.4.
- SYSTEM PROMPT: The system instruction that sets the context and domain policy for the agent's behavior and interactions with the user.

Here is an example for how an AI agent for brainstorming research ideas and writing paper abstracts is instantiated on Slack. The agent can search on ArXiv for generating new research ideas on a given topic, and writing paper abstract for that idea.

```
from slackagents import AssistantAgent

paper_abstract_agent = AssistantAgent(
    name="Paper Guru",
    desc="Brainstorm abstracts for a given topic",
    llm=OpenAILLM(BaseLLMConfig(model="gpt-4o")),
    tools=[arxiv_tool, abstract_writer_tool],
    system_prompt="You are an AI assistant that can help brainstorm\
an abstract for a given topic."
)
```

B.4.2 Workflow

A **workflow** is created by organizing individual agents into a structured directed graph, where each agent plays a role in moving toward a common goal. Despite involving multiple agents, the workflow operates as **a single, unified Slack agent**. This design allows users to interact directly with the workflow, which manages the coordination of all agents behind the scenes. From the user's perspective, the workflow appears as a single agent, or in other words, a **workflow agent** that follows a clear, step-by-step process, offering a smooth and reliable experience.

Graph A graph defines the sequence of execution, detailing the interactions and transitions between different agents. Each node in the graph represents an agent, and the edges represent the transitions that dictate when and how agents pass information or trigger each other's actions.

Here's an example where several agents work together for a quarterly check-in process on Slack:

Step 1. Define individual agents.

```
from slackagents import AssistantAgent

data_agent = AssistantAgent(
    name="Data Agent",
    desc="""AI agent designed to generate a report for the quarterly
    check-in meeting with Jira record.""",
    tools = [
        FunctionTool.from_function(load_jira_record_tool),
        FunctionTool.from_function(write_tool),
    ],
    system_prompt=system_prompt,
    verbose=True
)

calendar_agent = AssistantAgent(
    name="Calendar Agent",
    desc="AI agent designed to load an employee's calendar and\
    send the calendar invites",
    tools=[
        FunctionTool.from_function(load_employee_calendar_tool),
        FunctionTool.from_function(send_calendar_invite_tool)
    ],
    system_prompt=system_prompt,
    verbose=True
)

email_agent = AssistantAgent(
    name="Email Agent",
    desc="AI agent designed to send emails to employees",
    tools=[FunctionTool.from_function(send_email_tool)],
    system_prompt=system_prompt,
    verbose=True
)
```

Step 2. Build the execution graph that defines the workflow.

```
from slackagents import ExecutionGraph, ExecutionTransition

graph = ExecutionGraph()
graph.add_agent(data_agent)
graph.add_agent(calendar_agent)
graph.add_agent(email_agent)
```

Step 3. Define transitions between agents.

```
graph.add_transition(
    ExecutionTransition(
        source_module=graph.get_module("Data Agent"),
        target_module=graph.get_module("Calendar Agent"),
        desc="After the report is written to the employee's local directory"
    )
)

graph.add_transition(
    ExecutionTransition(
        source_module=graph.get_module("Data Agent"),
        target_module=graph.get_module("Email Agent"),
        desc="After the meeting is scheduled."
    )
)
```

Step 4. Set the initial agent in the workflow.

```
graph.set_initial_module(graph.get_module("Data Agent"))
```

Workflow The workflow is a higher-level construct that ties everything together. It encapsulates the agents and the execution graph into a single, reusable, and scalable process. Once the workflow is defined, it can be triggered in Slack to automate a series of tasks.

```
from slackagents import WorkflowAgent
quaterlycheckin_agent = WorkflowAgent(
    name="Quarterly Check-in Workflow",
    desc="Workflow to automate quarterly check-in process",
    graph=graph
)
```

This workflow agent ensures that the check-in process is automated on Slack, starting with generating the report from Jira data dump, scheduling the meeting, and finally sending email notifications, all without human intervention.

B.4.3 Proactive Behavior

Proactive behavior refers to the agent's ability to monitor conversations and provide timely assistance without being intrusive. This behavior is primarily achieved through the WAIT function, as described in Sec. B.2. It enables the proactive message handler to continuously monitor all incoming messages in threads where the agent is active. Based on the context and its capabilities, the agent determines whether to engage in the discussion or continue waiting for additional input.

This process is guided by a structured system prompt, which defines interaction rules, response guidelines, and criteria for silence or engagement. These directives ensure the agent maintains a balance between helpfulness and unobtrusiveness, leveraging the WAIT function to respond appropriately and effectively.

B.4.4 Tools

Tools represent the lowest-level actions that AI agents can perform. Coupled with function calling, tools enable AI models to automatically interface with and control external systems and applications. SLACKAGENTS supports 4 types of methods to define agent tools, for corresponding use cases. Furthermore, users can also directly use tools in other popular libraries, such as LANGCHAIN, LLAMAINDEX, CREWAI, and COMPOSIO, etc. to develop Slack agents.

Function Tool Users can define an arbitrary function and then use FUNCTIONTOOL.FROM_FUNCTION to generate a tool that agents can use automatically. Note that users are highly recommended to use type hints, together with standard Python docstrings, to provide information about the function's input and output. We currently support parsing of ReST, Google, Numpydoc-style and Epydoc docstrings. Automatic error handling is added for all functions wrapped inside FUNCTIONTOOL.

```
from slackagents.tools.function_tool import FunctionTool

def calculate_area(length: float, width: float) -> float:
    """
    Calculate the area of a rectangle.

    :param length: The length of the rectangle.
    :param width: The width of the rectangle.
    :return: The area of the rectangle.
    """
    return length * width

tool = FunctionTool.from_function(calculate_area)
```

Model Tool Besides putting everything inside the function docstrings, one can also explicitly define an agent tool from a Pydantic data model. Pydantic is a data validation library that uses Python type annotations for data validation and settings management. We can a Pydantic model for our function, specifying its input and output types. This approach offers several benefits:

- Ensures strictly that the tool JSON file sent to LLMs matches our expectations
- Provides better visualization of the tool definition
- Enables automatic input validation

```python
from slackagents.tools.function_tool import FunctionTool
from pydantic import BaseModel, Field

class CalculateArea(BaseModel):
    length: float = Field(..., description="Length of the rectangle")
    width: float = Field(..., description="Width of the rectangle")

    @classmethod
    def execute(cls, length: float, width: float):
        return length * width

tool = FunctionTool.from_pydantic(
    model=CalculateArea,
    name="calculate_area",
    description="Calculate the area of a rectangle"
```

By using a Pydantic model, we can create a strict and type-safe function tool compared to the previous approach of wrapping a Python function directly.

RESTful API Tool AI agents can also use OpenAPI JSON files to define RESTful API tools. Most digital tools today are available as RESTful APIs with standard request formats like GET or POST. While functions or Pydantic classes can be written for these requests, leveraging OpenAPI JSON files is more efficient for defining tools directly. This allows an agent to access a batch of tools simply by referencing a folder of JSON files. The tools also support various types of API authorization, including API keys (in headers or parameters), bearer tokens, and basic authentication with username and password, ensuring secure credentials handling.

```python
from slackagents.tools.function_tool import FunctionTool
tool_schema = ... # load a openapi json file

tool = OpenAPITool(
    name="api_name",
    openapi_spec=tool_schema,
    auth_type=AuthType.NO_AUTH
)
```

External Tool The community has developed an extensive number of AI agent tools. For instance, Composio.ai[4] has aggregated thousands of tools from over 150 popular systems, including GitHub, Twitter, etc. Likewise, LlamaIndex's [5] curates tools from open-source communities. As a result, the quickest way to jumpstart agent development is by using these ready-made tools. SLACKAGENTS supports this with wrappers that directly enable usage of tools from external libraries like, LLamaHub, LangChain[6], CrewAI[7] and Composio.

For rapid development, users can simply import the appropriate external tools from the pool of 1,000+ public, open-source tools. Detailed examples are given in the examples notebooks.

C Additional Applications

C.1 Workflow Agent on Slack

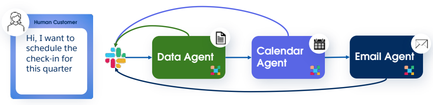


Figure 10: Quarterly check-in workflow. Employee initiates the check-in workflow with data agent, then schedule the check-in time slots with calendar agent, and finally send out the notification through email agent.

Autonomous Quarterly Check-In. The Quarterly Check-In Workflow demonstrates the effectiveness of SLACKAGENTS in automating structured organizational tasks, specifically the quarterly check-in process of employees. Designed as a Slack-integrated system, the workflow leverages a collaborative network of agents to perform tasks such as generating performance reports, scheduling meetings, and sending notifications. By modularizing the workflow into specialized agents—namely, the *Data Agent*, *Calendar Agent*, and *Email Agent*, this approach showcases the benefits of task delegation and automated coordination in multi-agent systems.

At the core of the workflow is the Data Agent, which initiates the process by synthesizing data from Jira records to create a detailed report on employee progress and challenges. This report is further enriched through direct interactions with employees, ensuring the inclusion of qualitative insights alongside quantitative metrics. The agent generates a markdown file summarizing key performance indicators, areas for improvement, and

[4]https://composio.dev/

[5]https://llamahub.ai/

[6]https://python.langchain.com/docs/integrations/tools/

[7]https://github.com/crewAIInc/crewAI-tools

future goals, enabling employees and their managers to prepare effectively for the check-in discussion. Once the report is finalized, the workflow seamlessly transitions to the Calendar Agent.

The Calendar Agent exemplifies the integration of intelligent scheduling with decision-making heuristics. By analyzing multiple employee calendars, it identifies optimal meeting slots based on criteria such as business hours, workload distribution, and personal preferences. This agent minimizes conflicts while prioritizing convenience, offering flexible time slots for employees to choose from when necessary. Once the scheduling task is complete, the workflow moves to the Email Agent, which ensures timely communication of the check-in details.

The Email Agent serves as the final component of the workflow, responsible for sending personalized notifications to employees. These emails summarize the scheduled meeting information and provide access to the prepared performance reports. The agent's role highlights how multi-agent systems can enhance the human-centric aspects of organizational processes by ensuring clarity, timeliness, and personalization.

C.2 Multi-Agent Reasoning on Slack

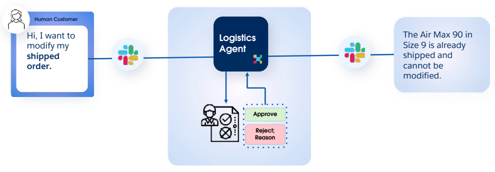

Human Customer
Hi, I want to modify my **shipped order.**
Logistics Agent
Approve
Reject; Reason
The Air Max 90 in Size 9 is already shipped and cannot be modified.

Figure 11: Trustworthy verification of Slack Agent. Logistics Agent asks an verifier for approval before actually executing a tool call.

Trustworthy Verifier. We design a human-in-the-loop slack agent to enhance the trustworthy when agent is going to use a tool, which is illustrated in Figure 11. In this application, we design a logistics agent for checking or modifying the status of an order. Besides tool call generation and execution, the agent execution involves another inner loop for tool call verification. A verifier, which can either be a *human* or another *agent*, approves or rejects with a reason a tool call request from the logistics agent. After receiving the approval or rejection response, logistics agent executes the tool-call or generates a new message, respectively. This trustworthy verification application demonstrates the extendability of SLACKAGENTS framework. Developers could easily integrate another agent or human into the agent execution loop such that the actions of an agent are monitored and verified.

D External Dependencies

The library is extremely lightweight, which only requires the following Python packages:

- `docstring-parser`
- `openai`
- `networkx`
- `matplotlib`
- `slack-sdk`
- `slack-bolt`