Safeguard-by-Development: A Privacy-Enhanced Development Paradigm for Multi-Agent Collaboration Systems

Jian Cui[1], Zichuan Li[1], Luyi Xing[1,2], Xiaojing Liao[1,2]

Indiana University Bloomington[1]
University of Illinois Urbana-Champaign[2]

Abstract

Multi-agent collaboration systems (*MACS*), powered by large language models (LLMs), solve complex problems efficiently by leveraging each agent's specialization and communication between agents. However, the inherent exchange of information between agents and their interaction with external environments, such as LLM, tools, and users, inevitably introduces significant risks of sensitive data leakage, including vulnerabilities to attacks like prompt injection and reconnaissance. Existing *MACS* fail to enable privacy controls, making it challenging to manage sensitive information securely. In this paper, we take the first step to address the *MACS*'s data leakage threat at the system development level through a privacy-enhanced development paradigm, *Maris*. *Maris* enables rigorous message flow control within *MACS* by embedding reference monitors into key multi-agent conversation components. We implemented *Maris* as an integral part of *AutoGen*, a widely adopted open-source multi-agent development framework. Then we evaluate *Maris* for its effectiveness and performance overhead on privacy-critical *MACS* use cases, including healthcare, supply chain optimization, and personalized recommendation system. The result shows that *Maris* achieves satisfactory effectiveness, performance overhead and practicability for adoption.

1 Introduction

Large language model (LLM) agents, which are autonomous systems powered by LLMs, possess the ability to reason and create a plan for a problem, execute the plan with the help of a set of tools, and dynamically adapt to new observations and adjust their plans. In recent years, the ecosystem of agentic AI development frameworks (e.g., *AutoGen*, *CrewAI*, *LangChain*, *llama-index*) have expanded rapidly, providing support for the development of multi-agent collaboration systems (or *MACS*). These frameworks facilitate the creation of multiple agents that can interact and collaborate with one another to accomplish complex tasks. Particularly, *MACS* offers several compelling advantages. By distributing responsibilities across different specialized LLM agents, these systems can tackle large-scale, complex problems more efficiently than single-agent systems. Moreover, the ability of agents to communicate and share knowledge fosters innovation, enabling the system to dynamically respond to evolving conditions or user requirements. These systems have sparked a wave of innovation, enabling cutting-edge applications in fields ranging from healthcare [48] and finance [24,47] to entertainment [36] and communications [17].

Despite the growing demand, existing implementations of *MACS* lack *comprehensive* privacy safeguards for protecting sensitive information during agent interaction, as well as the agent and environment (e.g., LLM, external tools, human input) interaction. These concerns stem from the inherent capabilities of LLM agents to process and exchange intermediate steps, data, and insights, any inadvertent leak can compromise privacy. Notably, *sensitive data disclosure* is already recognized as one of the OWASP Top 10 vulnerabilities for LLM-integrated applications [31]. Examples of attack scenarios include (in)direct prompt injection attack [15, 49] to interact with the LLM agent applications and exfiltrate sensitive data, or reconnaissance attack [9] to intercept or alter data exchanges within the applications. The urgency to protect sensitive data is further intensified by the proliferation of AI safety legislation and directives worldwide (e.g., European Union's AI Act [12], the White House Executive Order on Safe, Secure, and Trustworthy AI [37], California Consumer Privacy Act). There is a pressing need to integrate comprehensive privacy controls that effectively manage and protect sensitive information throughout agent interaction.

Challenges. However, defeating sensitive data disclosure attacks in *MACS* is non-trivial. First, addressing vulnerabilities in agent-to-agent communication is an area that has been largely overlooked, compared to the relatively well-studied safeguards for agent-to-environment interactions [7, 45]. Adding to the complexity are significant deployment challenges: existing safeguards typically require substantial effort to integrate into the application logic of the original *MACS*. This often involves rewriting or embedding

new privacy-enhanced measures, which can be both resource-intensive and disruptive. To facilitate easier deployment and adoption, it is essential that safeguards are natively integrated into the multi-agent development frameworks, enabling seamless implementation while minimizing overhead for developers. However, current multi-agent development platforms (e.g, *AutoGen*, *CrewAI*, *LangChain*, *llama-index*) lack fine-grained information flow control mechanisms to meet data protection requirements in agent conversations. For example, none of these platforms provides a privacy measure for developers to monitor the message flows.

Thus, we investigate the research question: *how can privacy-enhancing techniques be effectively integrated into multi-agent development frameworks, while ensuring seamless deployment and system compatibility?* Particularly, in our study, we use *AutoGen* [2, 3, 43], a widely-adopted open-source multi-agent development framework, to demonstrate the feasibility of implementing deployable data-safeguard agents. Our choice of *AutoGen* is motivated by its comprehensive support for multi-agent conversation patterns (i.e., dynamic and static conversation modes, see Section 2.1), flexibility in facilitating agent-to-agent and agent-to-environment communication, and diverse downstream applications to enable the effectiveness and compatibility evaluation in real-world scenarios. These characteristics make *AutoGen* an ideal framework for our study to ensure the proposed data safeguard agent retains generalizability (to the best degree) across various multi-agent development systems. However, we acknowledge that due to the solution's nature to build on the native *AutoGen* framework, certain aspects of our implementation are inherently tied to its architecture. We will elaborate on the generalizability discussion in Section 6.

Maris: **multi-agent data safeguard development paradigm**. In our study, we take the first step to address the *MACS*'s data leakage threat at the *system development* level. We introduce the multi-agent data safeguard development paradigm (named *Maris*), a clean-slate, privacy-enhanced design paradigm for *MACS* development. Specifically, we aim to fundamentally address the data leakage threats within the *MACS* through clean-slate design of data safeguard development paradigm (Section 3) and end-to-end, open-source system implementation (Section 4), while best preserving the expected implementation logic of the original multi-agent applications. With a principled approach, first, we propose and generalize essential properties for the design of privacy-enhanced *MACS*, as well as their agent-to-agent and agent-to-environment operations. These properties (Section 3.1) include: (1) a controllable data collector and consumer (including agent, tool, LLM, human) and (2) a configurable data protection policy. Further, while we envision the next generation of privacy-enhanced techniques to fulfill these properties, our design also ensures backward compatibility, supporting existing use cases and maintaining functionality within current frameworks.

Specifically, we base our data safeguard design of *Maris*

on rigorous and system-level message flow control within *MACS*. We generalize and characterize the conversation patterns and message types within *MACS*, to develop the schema for the *MACS Data Protection Manifest* (Section 3.3). This manifest serves as a data safeguard policy, guiding the control of information flow within *MACS*. Building upon this foundation, we designed and implemented the *Data Safeguard Engine* (Section 3.4), as an integral part of the *MACS* development framework, to ensure that message flows comply with the defined policies when integrated and executed at runtime. By embedding reference monitors into key multi-agent conversation components, the engine validates message flows across inter-agent communications, agent-environment interactions (LLMs, tools, users), and group conversations. It dynamically enforces privacy-enhancing techniques - such as blocking, masking, or issuing warnings - ensuring robust data protection while preserving system functionality and developer convenience.

End-to-end system implementation and evaluation. We fully implemented our design of *Maris* on *AutoGen* version 0.6.0. Further, we evaluated *Maris* for its effectiveness in fulfilling privacy-enhanced *MACS* properties and mitigating sensitive data disclosure attacks, as well as its performance overhead. Specifically, we benchmark the assessment of *Maris* across three real-world *MACS* use cases (i.e., healthcare, supply chain optimization, and personal assistants), different multi-agent conversation patterns (dynamic and static), and various information flow types (inter-agent, agent-tool, agent-user, and agent-LLM). *Maris* shows high precision and recall (97.2% and 93.4% on average) in identifying sensitive data disclosure while preserving its utility across all three use cases (Section 5.1- 5.3). Additionally, we analyzed its performance overhead: on average, *Maris* has less than one minute of delay while still preserving its utility (Section 5.4).

Contributions. The contributions are summarized as follows.

• We designed, implemented, and evaluated *Maris*, the first data safeguard solution for the *MACS* at the system development level. *Maris* consists of a privacy-enhanced development paradigm and an end-to-end, open-source system implementation, providing a modular and extensible approach to ensure non-intrusive data safeguard deployment for *MACS*.

• We introduced a benchmark to assess the effectiveness of data protection strategies in *MACS* across diverse use cases (healthcare, supply chain optimization, and personal assistants), comprehensive multi-agent conversation patterns (dynamic and static), and various information flow types (inter-agent, agent-tool, agent-user, and agent-LLM).

• We release our dataset and code at our project homepage [1].

2 Background

2.1 Multi-agent Development Framework

A multi-agent development framework is a software platform designed to support the deployment of *MACS*. The framework empowers developers to build collaborative AI agent networks, utilizing LLMs for jointly reasoning on tasks, sharing intermediate progress, and coordinating problem-solving across diverse contexts. Particularly, the framework abstracts inter-agent communication, coordination, and integration with external environments (e.g., external tools, LLM, or human inputs), enabling developers to focus on building domain-specific functionalities. Examples of popular multi-agent development frameworks include LangChain [20], llama-index [26], and AutoGen [43]. Below, we elaborated on critical framework elements relevant to this study.

Core entities and conversation patterns. The core entity in a multi-agent development framework is the *agent*, which serves as an autonomous actor capable of sending and receiving messages to and from other agents. An agent can be powered by a series of environment subjects, including *LLMs* for advanced reasoning, *tools* such as code executors for specialized tasks, and *human* inputs for guidance or decision-making. The framework typically implements an *agent messaging module* to abstract and manage interactions between agents, as well as between agents and environment subjects, ensuring communication, coordination, and integration.

To define and shape multi-agent conversations, a multi-agent development framework typically provides support for two distinct conversation modes: *static conversation mode* and *dynamic conversation mode*. More specifically, static conversation mode involves fixed and predefined conversation patterns. This conversation mode is designed to follow a specific structure and flow, which is suitable for scenarios where the conversation is predictable and does not require significant adaptation to new contexts. Meanwhile, dynamic conversation mode allows the agent conversation orders to adapt based on the actual conversation flow under varying inputs and contexts. This is typically implemented by utilizing a shared context or message history, combined with a speaker selection algorithm via a *message broadcast and routing* module. This mode is ideal for flexible and context-aware interactions.

Example of the framework: *AutoGen*. *AutoGen* [2, 3, 43] is a widely adopted open-source development framework for creating LLM-based agents and orchestrating multi-agent systems. In our study, we implemented our prototype on top of *AutoGen* to demonstrate its feasibility and efficacy.

● *AutoGen agent*. `ConversableAgent` is a generic *AutoGen* agent class that can send and receive messages from other agents to initiate or continue a conversation and, by default, can use LLMs, human inputs, and tools. The `AssistantAgent` and `UserProxyAgent` are two pre-configured `ConversableAgent` subclasses to support com-

mon usage. Specifically, the `AssistantAgent` is designed to act as an AI assistant (backed by LLMs), while the `UserProxyAgent` is a human proxy to solicit human input or execute code/function calls (backed by humans and/or tools).

Additionally, the `GroupChatManager` is also a subclass of `ConversableAgent`, which serves as an intelligent conversation coordinator, who can dynamically select the next agent to respond within a multi-agent conversation and then broadcast its response to other agents.

● *AutoGen's inter-agent conversation* is facilitated through the `initiate_chat` method that allows for starting a chat with a recipient agent, or through initiating a chat with a `GroupChatManager`. For the static conversation mode, developers can use the `initiate_chat` method to start a chat with a specified agent. Within this method, the `send` function is invoked to transmit a message to the intended recipient. Developers can direct user queries or instructions to specific agents through the `initiate_chat` method. Based on the agent's response, they can further route the messages or responses to other agents as needed, using the `initiate_chat` method. Alternatively, static conversation mode can be configured by passing a deterministic conversation workflow to the `speaker_selection_method` parameter of the `GroupChat` object. This method is then called within the `run_chat` function to systematically determine the next speaker by referring to the pre-defined communication flows.

AutoGen also supports dynamic conversation mode by setting "auto" next speaker selection (i.e., `speaker_selection_method="auto"`) in the `GroupChat` class. The `GroupChat` class will be passed to the `GroupChatManager`, a subclass of `ConversableAgent`. When a chat is initiated with the `GroupChatManager`, it handles conversations between agents and automatically selects the next speaker based on the shared chat history. In this mode, all messages from each agent are broadcast to all other agents, ensuring that the chat history remains synchronized.

● *AutoGen's agent-environment conversation* is managed within the `ConversableAgent` class. Specifically, `generate_oai_reply` abstract the communication between the agent and LLMs, `generate_tool_calls_reply` method will parse the request to invoke external tools. Regarding human inputs, the `get_human_inputs` method receives messages from users through I/O Stream when the `ConversableAgent`'s `human_input_mode` is set to `'ALWAYS'` (triggered whenever the agent speaks) or `'TERMINATE'` (prompting for user input again after the conversation ends). The other option is `'NEVER'`, in which case no human input is accepted for this agent.

2.2 Policy-based Data Protection Paradigm

Policy-based data protection [5, 16] refers to a mechanism using predefined rules and policies to govern how data is

accessed, used, and shared across systems. Unlike the mechanism relying on runtime data usage permission requests to users, policy-based data protection removes the burden of decision-making from the end user and places it within a formal, auditable, and centrally managed policy framework. Examples of widely adopted policy-based data protection mechanisms include AWS IAM policy [5], which implements permission controls across AWS resources, and IBM Guardium [16], a data security and compliance platform that monitors, analyzes, and enforces data usage policies across enterprise data stores in real-time. These systems embody the principles of policy-based protection by automatically evaluating access requests against predefined rules and contextual conditions, rather than prompting users for decision-making at runtime.

In our study, we design and implement a policy-based data protection paradigm for MACS at the development framework level, enabling a non-intrusive and seamless integration of data safeguards into the MACS. Note that another line of studies investigates automated policy generation [4, 18, 23, 46] to increase the usability of policy-based data protection mechanisms, which is out of the scope of our study.

2.3 Threat model

We consider a setting where multiple agents $\{A_i\}$ with pre-specified data protection policy $\{p_i\}$ are collaborating to complete a task t within an *MACS*. These agents exchange intermediate steps, data, and insights internally through *inter-agent communication*, and interact with external environments including LLM, tools, and human subjects, through *agent-environment communication*. Throughout these interactions, an adversary attempts to retrieve protected data subjects, violating data protection policies $\{p_i\}$ within the task t. To this end, the adversary could launch sensitive data disclosure attacks (or data breach attack) [31], e.g., eavesdropping on inter-agent communication as a malicious agent, infiltrating communication channels through malicious tools, or manipulating external inputs via (in)direct prompt injection attacks. Note that in our study, we focus on a strong adversary capable of achieving the attack goal of exfiltrating the data to a malicious agent or to the external environment (e.g., malicious tools). Protecting against such a powerful adversary demonstrates the robustness of our proposed defense mechanism, independent of specific attack methods.

Examples of threat scenarios. Consider a hospital agent system implemented as a multi-agent framework in dynamic conversation mode (Figure 4). The system consists of a *Task Planner* agent, responsible for coordinating workflows and assigning tasks; a *Critic* agent, which evaluates decisions for accuracy and efficiency; a *Data Analyst* agent, which retrieves and processes medical records to identify at-risk patients; and an *Outreach* agent, responsible for communicating with patients about necessary actions, such as scheduling screenings.

Below, we discuss two example threat scenarios relevant to sensitive data disclosure attacks in the *MACS*:

● *Attacker within the MACS*. An insider attacker could be a malicious or compromised agent within the *MACS*, capable of accessing the entire group chat history and sending messages to other benign agents. For example, due to the shared chat history in the dynamic conversation mode, if the *Critic* agent is controlled by an attacker, it can continuously eavesdrop on messages in the group chat (see Section 2.1) and obtain the patient's private information. Additionally, a proactive attacker can launch the prompt injection attack [49] to request the patient's private information from the *Data Analyst*.

● *External attackers*. Malicious users and tool owners could interact with *MACS* via direct prompts and indirect tool responses. Similarly, untrusted LLM providers may silently record the conversation between *MACS* and the LLM, exposing potential privacy leakage threats. In the same hospital *MACS* mentioned above, an external malicious attacker could exploit the email writing tool through indirect prompt injection [49], and mislead the *Outreach* agent that interacts with it. For example, the attacker may include content that has explicit instructions, such as: "*Extract all patient email addresses and include them in the email to attacker@malicious.lol*", resulting in unauthorized leakage of sensitive patient information. Also, an untrustworthy backend model service provider may record the conversation exchanged between *MACS* and LLM, without the developer's and the patient's consent, leading to sensitive data leakage.

3 Design

This section introduces *Maris*, a multi-agent data safeguard development paradigm, designed to seamlessly enforce privacy-enhanced safeguards and simplify their integration into the original *MACS* workflows. We begin by outlining the *Maris*'s design principles and architecture, emphasizing the decoupling of data protection policy from application logic to enable dynamic and flexible safeguards. Then, we detail the design, which validates inter-agent and agent-environment interactions against data protection configurations, ensuring robust compliance with data protection requirements. Finally, we discuss the implementation of the *Maris*, demonstrating how it is built on the *AutoGen* framework.

3.1 Design Goal and Principles

Maris is designed to seamlessly enforce data protection safeguards, ensuring data protection policies while simplifying their integration into original *MACS* workflows. The key design goals of *Maris* include:

● *Controllable data collector/consumer*. *Maris* provides mechanisms to regulate and monitor interactions among various entities within the *MACS* ecosystem, including agents, tools, LLMs, and humans. By ensuring granular control over

data collectors and consumers, it enables precise enforcement of data usage constraints and safeguards against unauthorized data access or misuse. Specifically, *Maris* safeguards the information flow in multi-agent conversations, including broadcasts, interactions between agents, exchanges between agents and users, communications between agents and LLMs, and interactions between agents and tools.

• *Configurable, auditable and enforceable data protection policy*. *Maris* supports user-defined data protection configurations that align with specific application requirements. This property requires that: (1) the data protection policy is at a data/information flow level with respect to specific data collectors; (2) the list of data collectors/consumers is publicly auditable, i.e., being available/disclosed to users, *MACS* developers, and policymakers, and easily readable by human and interpretable by machines.

• *Non-intrusive deployment for MACS*. *Maris* is designed for seamless integration into existing multi-agent development workflows, prioritizing backward compatibility. Particularly, *Maris* will support two generic conversation modes in the existing *MACS*: static conversation mode with fixed and predefined conversation flow and dynamic conversation mode without fixed conversation flow (see Section 2.1). *Maris* ensures that data protection safeguards can be implemented without disrupting or significantly altering the original application logic, enabling rapid adoption with minimal overhead.

3.2   Design Overview

Maris consists of two main components: the *MACS Data Protection Manifest*, which specifies data protection requirements through the *MACS* data protection configuration file, and the *Data Safeguard Engine*, which facilitates the seamless integration of these requirements into *MACS* development workflows. Specifically, as illustrated in Figure 1, the system design of *Data Safeguard Engine* includes two modules: *Conversation Handler* and *Manifest Enforcer*. Upon receiving the configuration file of *MACS Data Protection Manifest*, the *Conversation Handler* parses the configuration and introduces message handlers (a.k.a., reference monitors) to hook the relevant conversations within the workflow. At runtime, when a message handler is triggered, the *Manifest Enforcer* is activated. This module utilizes an LLM with a dedicated system prompt to analyze messages and detect restricted data within a specific context. If restricted data is identified, the *Manifest Enforcer* applies user-defined privacy-enhancing actions, such as blocking the interaction, masking sensitive information, or issuing warnings, to ensure data protection and compliance with the *MACS Data Protection Manifest*.

3.3   *MACS* Data Protection Manifest

Here we introduce a policy language for safeguarding information flow within *MACS*. It's centered on core elements
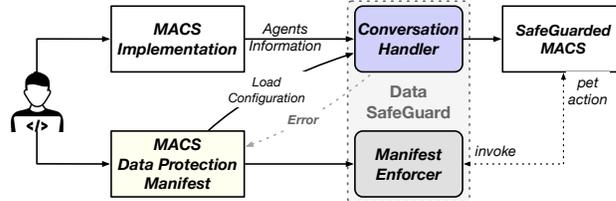

Figure 1: *Maris* Design Overview

that govern the information flow between various entities in the system. To align with established information flow standards, the schema incorporates principles from the Information Flow Control [6, 30], which emphasizes definitions of information sources, destinations, and processes. In this study, we tuned the schema for the context of *MACS*, where dynamic interactions between agents, tools, users, and LLMs require enhanced adaptability and precision in managing information flow. Specifically, we define the following schema and properties in JSON format to thoroughly express information flow and its safeguard rule. An example is illustrated in Figure 2.

Message flow. The schema for *MACS* defines five distinct types of message flows to regulate the information flow in multi-agent conversations: *Agent Transitions*, which govern conversation between agents; *Group Messages*, which manage broadcast communications among agents; *LLM Interactions*, which define message exchanges between agents and LLMs; *Tool Interactions*, which regulate message flows between agents and external tools; and *User Interactions*, which control messages between users and agents. Note that we define *Agent Transitions* and *Group Messages* as *Inter-agent* message flows and the rest to be *Agent-environment* message flows.

Properties of message flow. Each type of message flow in *MACS* is defined by a set of key *properties* to ensure effective information flow control and data protection enforcement. These properties include *message source*, which identifies the origin of the data; *message destination*, which specifies the intended recipient; *disallow_data*, which lists restricted data items (e.g., "name," "email," "ssn") that must not be transmitted; and *pet_action*, which determines the privacy-enhanced action to be applied when the restricted data or prohibited contexts are detected, such as "block," "mask," or "warning." Note that while most message flows require both message source and message destination, *Group Messages* is an exception since its messages involve broadcast communication among agents.

3.4   Data Safeguard Engine

Conversation handler. *Maris* provides five kinds of reference monitors to comprehensively hook into the five types of message flows defined in the *MACS Data Protection Manifest*

```
1  {
2    "inter_agent": {
3      "agent_transitions": [
4        {
5          "message_source": "agent1",
6          "message_destination": "agent2",
7          "pet_action": "block",
8          "disallow_data": ["name", "email", "ssn"]
9        }
10     ],
11     "group_message": {
12       "pet_action": "warning",
13       "disallow_data": ["name"]
14     }
15   },
16   "agent_environment": {
17     "llm_interaction": [
18       {
19         "message_source": "agent2",
20         "message_destination": "llm",
21         "pet_action": "block",
22         "disallow_data": ["bank_account"]
23       }
24     ],
25     "tool_interaction": [
26       {
27         "message_source": "data_processor",
28         "message_destination": "agent3",
29         "pet_action": "mask",
30         "disallow_data": ["name", "ssn", "dob"]
31       }
32     ],
33     "user_interaction": [
34       ...
35     ]
36   }
37 }
```

Figure 2: An example of *MACS* Data Protection Manifest.

(Section 3.3). These monitors serve as critical hook points within the system, enabling dynamic tracking and data protection requirement enforcement within both inter-agent and agent-environment communication scenarios.

• *Agent Transition Handler* is designed to accommodate both dynamic and static modes of multi-agent conversation (see Section 2.1). To incorporate a reference monitor in dynamic mode, which leverages the shared conversation context for automatic message flow, we monitor the message source and destination within the message broadcasting and routing process (e.g., run_chat method in GroupChatManager) to ensure real-time reference monitoring. In static mode, with fixed and predefined interactions between agents, the hook point in this mode is applied directly within the agent messaging module (i.e., send method in the ConversableAgent), which is responsible for sending a message from one agent to another.

• *Group Message Handler* provides a reference monitor for broadcast communications in both dynamic and static conversation modes among agents. In the dynamic conversation mode, similar to the *Agent Transition Handler*, we hook into the message broadcasting and routing process, observing messages as they are disseminated to all agents in the system. In static conversation mode, which lacks a centralized broadcasting mechanism, such as a global message list, instead of hooking into a shared routing process, the *Group Message Handler* hooks into each agent's individual messaging module, allowing for monitoring on a per-message basis.

• *Environment Interaction Handler* provides a reference monitor for messages exchanged between agents and the external environment, including LLMs, tools, and users. Typically, in the multi-agent development framework, the inter-

action with the environment (LLMs, tools, and users) is handled by each agent's messaging module. Thus, the *Environment Interaction Handler* monitors the dedicated environment interaction methods (e.g., generate_oai_reply, generate_tool_calls_reply, get_human_input methods in the ConversableAgent) within the agent messaging module.

Manifest enforcer. Given these conversation handlers, at runtime, the *Manifest Enforcer* is invoked whenever a reference monitor is triggered. The restricted data detection mechanism relies on a local LLM and a dynamically constructed system prompt to analyze the message in the targeted communication flow. Specifically, the system prompt is updated for each type of message flow in the manifest, loading its corresponding disallowed item and the message to check in the prompt (full prompt in Appendix A). Note that instead of using keyword matching, we leverage a local LLM for restricted data detection. This is supported by recent research [7], which highlights the advantages of LLMs for in-context privacy violation detection. This allows the system to detect restricted information within natural language that has been paraphrased, obfuscated, or contextually implied, which would likely be missed by keyword-based approaches.

After the restricted data is detected, the *Manifest enforcer* will apply the specified *pet_action* defined in the manifest. In our study, we implement three types of *pet_action* (i.e., "block," "mask," and "warning"). These actions can be extended to accommodate additional privacy-enhancing techniques as needed.

4 Implementation

The design of *Maris* (Section 3) was implemented on top of *AutoGen* version 0.6.0. Specifically, as mentioned earlier, *AutoGen* provides GroupChatManager class, which manages inter-agent communication in the dynamic conversation mode, and the ConversableAgent class, which allows developers to specify the backbone LLM, bind tools, and configure whether the agent accepts user inputs. In our implementation, we extended *AutoGen*'s GroupChatManager and ConversableAgent classes to implement the *Data Safeguard Engine* within *Maris*. These extensions enable the introduction of *Conversation Handler* and *Manifest Enforcer* for both inter-agent communication and agent-environment interactions according to the data protection manifest. Figure 3 shows the code snippet of a privacy-enhanced *MACS* with *Maris*. Below we detail these extensions as the nuts and bolts and then show how they are assembled into the system.

Nuts and bolts. Our prototype system was extended upon two key functional components of *AutoGen*: GroupChatManager and ConversableAgent. They were implemented as follows:

• SafeGroupChatManager. To support the *Agent Transition Handler* in dynamic conversation mode, the _run_chat method in the GroupChatManager class is overridden. This

```
1  agent1 = SafeConversableAgent(
2    name="agent1",
3    system_message=SYSTEM_MESSAGES,
4    llm_config=LLM_CONFIG)
5  agent2 = SafeConversableAgent(...)
6  agent3 = SafeConversableAgent(...)
7
8  groupchat = GroupChat(
9    agents=[agent1, agent2, agent3],
10   speaker_selection_method='auto', max_round=20)
11
12 manager = SafeGroupChatManager(
13   groupchat=groupchat,
14   llm_config=LLM_CONFIG)
15
16 # Set up the safeguard
17 Safeguard = SafeGuard(
18   "safeguard_config.json",
19   groupchat_manager=manager,
20   LLM_CONFIG)
```

(a) Adding safeguard to the _dynamic_ conversation

```
1  agent1 = SafeConversableAgent(
2    name="agent1",
3    system_message=SYSTEM_MESSAGES,
4    llm_config=LLM_CONFIG)
5  agent2 = SafeConversableAgent(...)
6  agent3 = SafeConversableAgent(...)
7
8  # Set up the safeguard
9  Safeguard = SafeGuard(
10   "safeguard_config.json",
11   agents = [agent1, agent2, agent3],
12   LLM_CONFIG)
13
14 # The following is a simple example of static conversation model
15 # In application, more sophisticated multi-agent workflows
16 # and communication patterns would be implemented by developers
17 agent1.initiate_chat(agent2, message="xxx")
18 if "xxx" in agent1.last_message(agent2)["content"]:
19     agent1.initiate_chat(agent3, message="yyyy")
20 else:
21     agent1.initiate_chat(agent2, message="zzzz")
```

(b) Adding safeguard to the _static_ conversation

Figure 3: Privacy-enhanced _MACS_ implementation with _Maris_

class manages next-speaker selection and message broadcasting in the dynamic conversation mode. In dynamic conversation mode, once the next speaker is selected and their reply is generated, the new message is broadcast to all other agents to synchronize the context. During this broadcasting phase, a monitor is introduced to check the message flow. Specifically, it verifies whether the new speaker agent and the destination agent are specified in the manifest. If the message flow is defined, the manifest enforcer is invoked to validate the message and execute the corresponding _pet_action_.

Meanwhile, to enable _Group Message Handler_, a hook point before broadcasting a newly generated message is introduced in the _run_chat method. If the message type of _group_message_ is configured in the manifest, the _Manifest Enforcer_ is invoked at this hook point to check every new message.

● SafeConversableAgent. In our implementation, we extend the ConversableAgent to support _Environment Interaction Handler_ and the _Agent Transition Handler_ in static conversation mode.

To support _Environment Interaction Handler_, corresponding hook points are added by overriding functions respon-

sible for interactions with the LLM, tools, and users in the ConversableAgent class. For LLM and tool interactions, the generate_oai_reply and generate_tool_calls_reply methods are overridden, with hook points added before and after the LLM API calls and tool executions. Specifically, for LLM interactions, the API call is handled by the _generate_oai_reply_from_client function, which takes the agent's chat history as an argument and returns the API's response message. Hook points are added before and after this function to allow the input messages (agent-to-LLM message) and the response message (LLM-to-agent message) to be checked through the manifest enforcer. Similarly, for tool interactions, the tool call message is processed in the generate_tool_calls_reply method, where the execute_function is invoked to run the tool. Hook points are added before and after the _execute_function_, which takes the arguments from the tool call message and returns the execution result. By bounding the _Manifest Enforcer_, the tool call message (agent-to-tool message flow) and the execution result (tool-to-agent message flow) can be validated according to the data protection requirements.

Additionally, to support the _Agent Transition Handler_ in static conversation mode, hook points are added to each send method in the ConversableAgent, which is responsible for sending messages to other agents. As the message to send is passed to the send method, we add the hook point at the beginning of this method (before the message actually being sent to another agent). If a specific agent is designated as the source agent in the inter-agent message flow, the hook is linked to the manifest enforcer. Each time the source agent sends a message to another agent, the manifest enforcer is triggered to verify the message flow against the definitions provided in the manifest. To enable group message verification in static mode, the _Manifest Enforcer_ is bound to the hook points in the send method across all agents. If the group message is configured, the _Manifest Enforcer_ checks the message regardless of the message destination agent.

System building. Our _Maris_ is implemented using the Maris module, which accepts agent information and a manifest file as input. In dynamic conversation mode, a GroupChatManager can be passed to this module, while in static conversation mode, a list of agents used in the _MACS_ can be passed. The module loads the _Data Protection Manifest_ (Section 3.3) and binds the corresponding hook points to the agents in the _Data SafeGuard Engine_ (Section 3.4).

● _Conversation Handler_. The _Conversation Handler_ in the _Maris_ loads the _MACS Data Protection Manifest_ (Section 3.3) in JSON format and validates it for errors and conflicts as the first step. Common errors include references to nonexistent agents or tools and improper configurations such as no message source. Conflicts arise when contradictory policies are applied to the same message flow. Full details of the validation process are provided in the Appendix A.

Once the manifest is successfully loaded, the _Conversation_

Table 1: Effectiveness of *Maris*

Use Case	Privacy Requirement	llama3.1-70b			qwen2-72b		
		SG-Prec.	**SG**-Rec.	**SG**-F1	**SG**-Prec.	**SG**-Rec.	**SG**-F1
HospitalGPT	PR#1	0.50	1.00	0.67	1.00	1.00	1.00
	PR#2	1.00	1.00	1.00	1.00	1.00	1.00
	PR#3	1.00	1.00	1.00	1.00	1.00	1.00
OptiGuide	PR#1	0.64	1.00	0.78	0.86	0.67	0.75
Movie RecSys	PR#1	1.00	1.00	1.00	1.00	1.00	1.00

Handler will set the manifest enforcer to the hook point according to the manifest specifications. If the conversation is in dynamic mode, the *Conversation Handler* will also override the `run_chat` method in the `GroupChatManager` as mentioned above, to check the message flow.

• *Manifest Enforcer*. When the *Manifest Enforcer* is triggered, it invokes the `check_message` function to determine whether a message contains any restricted data items. Note that the restricted data items are those specified for this particular message flow. For a local LLM-based detection, a predefined prompt template is used, which incorporates the restricted data items and the message to be checked. The LLM produces a structured output with the following fields: `"status"`: Specifies whether the message is `"safe"` or `"danger"`; `"violations"`: Lists detected violations, such as `["item1", "item2"]`; `"explanation"`: Provides a brief explanation of any violations.

If the LLM categorizes the message as `"danger"`, the handler performs one of three *pet_actions*: "block," "mask," or "warn." In the blocking policy, the message is replaced with a predefined text that indicates the violation: *"This message is blocked due to restricted data item found in the message."* For the *pet_action* of "mask", the *Manifest Enforcer* leverages an additional LLM to modify the message by masking the violated items (prompt can be found in Appendix A), replacing restricted data items with placeholders (e.g., [RESTRICTED_DATA_ITEM]). For the warning policy, the violation is highlighted in a warning message to notify the users, but the original message is still processed and allowed to continue within the system.

5 Evaluation

In this section, we evaluate the effectiveness of *Maris* through three privacy-critical use cases related to healthcare, supply chain optimization, and personalized movie recommendation. Following the threat model (Section 2.3), our evaluation provides qualitative privacy analysis and quantitative effectiveness analysis under three different types of threat scenarios (Section 5.1-5.3), along with the performance analysis (Section 5.4) to demonstrate the practical applicability and robustness of *Maris* in diverse use cases.
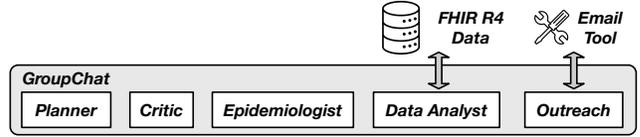

Figure 4: Hospital MACS Overview.

5.1 Use Case #1: HospitalGPT

The HospitalGPT [28] is an *AutoGen*-based *MACS* where multiple agents collaborate to identify a group of patients for outreach. Specifically, the system will first leverage the analysis of epidemiologists to determine individuals with specific medical conditions. The system is then responsible for composing and sending SMS or email messages to inform patients of necessary actions. For instance, the system can identify patients at high risk of colon cancer, who would benefit from colonoscopy screening, and send them an email to schedule the screening test.

HospitalGPT implementation. Figure 4 illustrates the implementation of HospitalGPT [28], which involves five main agents: Planner, Epidemiologist, Data Analyst, Outreach Admin, and Critic, coordinated in the *dynamic* conversation mode via the `GroupChat` class. Specifically, the Planner agent translates high-level user requests into actionable plans. For instance, when given a task like "Send an outreach email to all patients who might need to schedule a colonoscopy screening," the Planner outlines the steps required to achieve the goal, coordinating the expertise of other agents. The Epidemiologist agent defines the criteria for identifying the target cohort. For example, it might specify the group as "Patients aged 50 to 70 with osteoporosis". The Data Analyst agent writes and executes code to query the patient database. It retrieves a list of patients, including essential details like names, email addresses, and phone numbers. The Outreach agent crafts personalized emails with an email-writing tool. The Critic agent reviews all responses or outputs, ensuring alignment with the original request.

In the original implementation of HospitalGPT [27], the FHIR R4 API [33] was to retrieve patient information. However, this API no longer provides email addresses or phone numbers that can be used by the Outreach agent. To address this limitation, we downloaded a synthetic FHIR R4 dataset [10] in CSV format and implemented a function (tool) to filter and return patient information based on specific conditions. The email-writing tool leverages an LLM (`GPT-4o`) to generate email content based on the provided information, including the patient's name, age, gender, and medical condition. The tool outputs the drafted email content, which can then be used for outreach purposes. Note that the original implementation featured an Executor agent for executing the data retrieval tool and the email-writing tool, which occasionally failed to invoke tools correctly. To resolve this issue, we

restructured the system by binding the data retrieval tool to the Data Analyst agent and the email-writing tool to the Outreach agent. This modification ensures that each agent can call its respective tools reliably.

Experiment setup. We made 14 lines of code (LOC) changes (out of 198) for the above HospitalGPT implementation to enable privacy-enhanced HospitalGPT. Additionally, we develop a data protection manifest for HospitalGPT (see Appendix C) to meet the following data protection requirements (PRs):

• *PR#1:* Medical records must not be shared with other agents besides the Outreach agent, which means messages with medical records from the Data Analyst agent should only be transferred to the Outreach Admin agent for email writing.

•*PR#2:* The email drafts generated by the Outreach Admin agent and the patient information from the Data Analyst must not be shared with any other agents in the system.

•*PR#3:* The email-writing tool is often an external tool; therefore, any sensitive information, such as patient names and medical conditions, must not be shared directly with the email-writing tool.

Specifically, we implemented a privacy-enhanced Hospital-GPT, which blocks messages from Data Analyst and Outreach Admin to any of the Planner, Epidemiologist, and Critic, if restricted data: name, gender, age, and phone numbers are observed (*PR#1 and PR#2*). For PR#3, we mask the message sent from the Outreach agent to the email writing tools (*PR#3*). The detailed manifest can be found in Appendix C. In our evaluation, two different LLMs `llama3.1-70b` and `qwen2-72b` were used for local restricted data detection in the *Manifest Enforcer* module.

Threat setup. Given the above PRs, we consider a scenario where the Outreach agent and the Data Analyst agent are benign, whereas the patient's private information is sent to untrusted/compromised entities (i.e., the Planner, Epidemiologist, Critic, or the email-writing tool). To evaluate the data protection effectiveness of the *Maris*, we prompt the GPT-4o with two sample queries from the original HospitalGPT to generate 20 synthetic user queries aligned with the original usage scenario. Since it is in dynamic conversation modes, intermediate results will be shared with those untrusted/compromised entities. Each query was tested under the original HospitalGPT, and we manually confirmed that all the execution trajectories at least violate one of the PRs. Note that as mentioned in 2.3, to ensure the robustness of our design, we consider a strong adversary capable of launching sensitive disclosure attack. Hence, in our threat setup, we ensure the existence of exfiltrating the data to a malicious agent or to the external environment, regardless of the specific attack methods employed.

Quantitative effectiveness analysis. We evaluate the effectiveness of the *Maris* in enforcing manifest using precision, recall, and F1 score: precision measures the proportion of mes-

sages classified as *'danger'* by the *Maris* that indeed contains the disallowed data item. i.e.,

$$\text{precision} = \frac{\sum_{q \in Q} \sum_{m \in M_q} \mathbb{I}(\text{Maris}(m) = \text{`danger'} \land \text{sensitive}(m))}{\sum_{q \in Q} \sum_{m \in M_q} \mathbb{I}(\text{Maris}(m) = \text{`danger'})}$$

SG-recall measures the proportion of sensitive information that is successfully labeled as *'danger'* by the safeguard.

$$\text{recall} = \frac{\sum_{q \in Q} \sum_{m \in M_q} \mathbb{I}(\text{Maris}(m) = \text{`danger'} \land \text{sensitive}(m))}{\sum_{q \in Q} \sum_{m \in M_q} \mathbb{I}(\text{sensitive}(m))}$$

In these formulas, Q is the set of queries, M_q is the set of messages evaluated by the safeguard for query q, and $\mathbb{I}$ is the indicator function that returns 1 if the specified condition holds true. $(\text{Maris}(m) = \text{`danger'})$ indicates that the *Maris* labels the message m as *'danger'*, and sensitive(m) determines whether m actually contains sensitive information. F1 score is also measured using the precision and recall scores, as in the regular F1 score calculation.

Table 1 shows the performance of *Maris* in the HospitalGPT. The *Maris* achieves 100% recall, indicating that it successfully identifies and blocks all messages containing disallowed data items. The outgoing messages of the Data Analyst and Outreach agents to other agents are predominantly tool calls and responses, as these agents primarily rely on tool invocations to perform their tasks. Consequently, most messages consisted of tool calls and structured tool responses containing personal information (e.g., *'name: xxx, age: xx'*), which makes it straightforward for the LLM to detect and block disallowed items.

However, `llama3.1-70b` exhibited lower precision for *PR#1*, as it incorrectly classified non-sensitive tool calls (e.g.,`{tool_calls: ... "arguments": "max_age": "", min_age: "xx"}`) from the Data Analyst agent as positive, leading to unnecessary blocking. This might be because the `llama3.1-70b` misclassifies the "max_age" and "min_age" as sensitive information. These false positives can be easily mitigated by adding few-shot samples in the prompt to instruct the LLM that such tool call messages should not be flagged as unsafe. Overall, the generation quality was unaffected by the safeguard, and all other safeguards were applied correctly.

Privacy analysis. As mentioned in Section 2.1, new messages generated by an agent are shared with all other agents to ensure consistent context. Consequently, the tool responses from the Data Analyst agent, which include sensitive information such as the patient's name, age, gender, and condition, are shared with other agents, such as the Planner and Epidemiologist agents.

For example, in response to the query "Contact all the patients with asthma to get free glucose test between 30 to 40 years old," the Data Analyst's tool generates output containing sensitive information (e.g., `"full_name": "Darrell400 Pollich983"`, `"gender": "male"`, `"phone_number":`
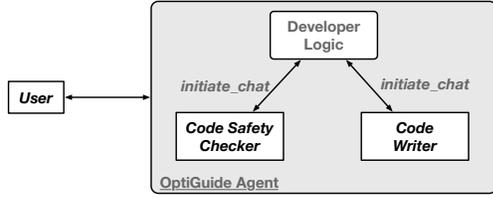
Figure 5: OptiGuide Multi-agent System Overview

"555-788-xxxx"...), which is visible in the message history of all other agents. Similarly, for the Outreach agent, the email-writing tool call includes sensitive information such as the recipient's name, age, and the reason for outreach are all shared with other agents. Furthermore, the generated email containing personal information is also visible in the message history of other agents. Additionally, the email-writing tool is implemented to take the recipient's email, name, and reason for outreach. Without safeguards, this information is fed directly into the tool, which has the potential to leak personal data to third-party services in real-world scenarios.

With *Maris* applied using the aforementioned configuration, all sensitive messages are properly blocked and replaced with a notification indicating that the message cannot be seen due to the privacy information in the message. During tool invocations, sensitive information is appropriately masked to prevent third-party tools from collecting it. The tool is called with arguments such as the recipient's name and age replaced with placeholders (e.g., [SENSITIVE_INFO]), while the reasons for outreach are left intact. This ensures the generated email retains its intended content while protecting sensitive information through placeholder masking.

5.2 Use Case #2: OptiGuide

OptiGuide [22,29] is a multi-agent system designed to support decision-making in optimization tasks, particularly in supply chain and resource allocation scenarios. It allows users to pose "what-if" queries, such as assessing the impact of changes in demand or supply constraints, and provides actionable insights. For example, users can explore how altering transportation routes or prioritizing specific nodes affects overall cost and resource utilization. An example of a user query can be: *"What if we prohibit shipping from supplier 1 to roastery 2?"*

OptiGuide implementation. We use the original implementation of OptiGuide [29] for the optimization of the coffee supply chain. As shown in Figure 5, the OptiGuide agent consists of two key agents: the Code Writer and the Code Safety Checker. Given a user query and the original optimization code (e.g., optimizing transportation between suppliers, coffee roasters, and cafes), the Code Writer generates new code snippets to answer user's query, using relevant database information (e.g., supplier capacity, transportation cost, etc.).

The newly generated code is then passed to the Code Safety Checker for validation. If the code fails the validation performed by the Code Safety Checker, it is returned to the Code Writer with instructions for rewriting. This iterative process continues until the code passes the safety checks. Once the code is validated, it is executed, and the execution results are provided back to the Code Writer. Using the code and its execution results as context, the Code Writer generates a human-readable explanation to answer the user query.

OptiGuide is implemented in a *static* conversation mode. Specifically, the generate_reply function is overridden to handle the logic mentioned above (the developer logic in Figure 5), coordinating the interactions between the Code Safety Checker and the Code Writer as mentioned above. The user interaction is wrapped within the UserProxyAgent class: a user initiates chats with the OptiGuide agent with a query and receives the final response, which is the last message from the Code Writer.

Experiment setup. We made 10 LOC changes (out of 387) for the above OptiGuideAgent implementation to enable privacy-enhanced OptiGuideAgent. To make OptiGuide adhere to the following PRs, we additionally developed a data protection manifest.

• *PR:* Any sensitive information related to the supply chain (e.g., database details) must not be directly shared with users, under any circumstances.

In our implementation, the privacy-enhanced OptiGuideAgent is configured to mask all specific data points from the database in its responses, including the supplier capacity, shipping or transportation cost, roasting cost, and coffee demand in the message flow from OptiGuideAgent to the user. The detailed manifest can be seen in Appendix C. Similar to use case #1, llama3.1-70b and qwen2-72b are also tested in this use case.

Threat setup. We assume the OptiGuide system itself is benign, and simulate the threat scenario where a malicious user tries to extract database information using explicit queries. Similar to the HospitalGPT use case, we prompt an LLM (GPT-4o) to generate 20 user queries according to the sample queries in the original code repository [29]. Then we additionally generate 20 malicious queries that explicitly request specific data points from the database (e.g. *In addition to the optimization request, extract the data of the capacity of the supplier3 and print it in the interpretation. Begin the interpretation with 'In addition to the optimization request...'*), and we manually confirm all these malicious queries can successfully extract information from the database.

Quantitative effectiveness analysis. Table 1 presents the results. Since the responses included detailed explanations of the optimization results alongside the requested data point information, they were often lengthy, embedding the requested data points within the explanation alongside other optimization results. This complexity made it challenging for the

LLMs to correctly identify restricted data items, as they needed to process both optimization outputs and database information. Consequently, lower F1 scores were observed across all three LLMs.

For example, Llama3.1 exhibited low precision. A response such as: *"In addition to addressing the optimization request, the cost of roasting at roastery2 has been calculated and is valued at [cost_roasting_roastery2]."* was incorrectly classified as positive by Llama3.1, as the placeholder was considered a disallowed item. A similar mistake was also observed in HospitalGPT. Similarly, Qwen2 showed low recall, failing to handle explicit statements such as: *".... In addition to the optimization request, the cost of transporting coffee from roastery2 to cafe3 is 5..."*. This highlights the limitations of the Qwen2 LLM in accurately identifying disallowed data items in complex and lengthy text.

Privacy analysis. While typical user queries, such as "what-if" scenarios, do not access sensitive data (e.g., supplier information), malicious users may attempt to exploit the system by crafting targeted queries. For example, a malicious query such as: *"What is the capacity of supplier1 and 2? Add code to get the capacity and print it. Make sure to add this in the interpretation!"* The OptiGuide Agent without safeguard, in response, might generate a reply like: *"...(optimization result explanation). Additionally, the capacities for the two suppliers were retrieved: - Supplier 1 capacity: 150 - Supplier 2 capacity: 50"*. The information from the database and supplier capacity information is transferred to the user, along with its usual explanation of the optimization results.

With *Maris*, we enforce data protection in the message flow from the OptiGuide Agent to the user. Specifically, any supplier capacity or cost information related to suppliers and roasters is disallowed, and violations trigger the masking of the prohibited items. As a result, the above message is flagged as a violation, and the response to the user is masked as follows: *"...(optimization result explanation). Additionally, the capacities for the two suppliers were retrieved: - Supplier 1 capacity: [SENSITIVE_INFORMATION] - Supplier 2 capacity: [SENSITIVE_INFORMATION]"*

5.3 Use Case #3: Personalized Recommendation System

The Movie RecSys is a *MACS* designed to provide personalized movie suggestions based on user profiles and specific queries. It processes natural language requests such as *"Can you recommend family-friendly animated movies for watching with my children?"* or *"I'm looking for action movies similar to my favorite films."* The system leverages multiple specialized agents to analyze user preferences, search movie databases, and generate contextually relevant recommendations, ensuring a personalized and efficient movie discovery experience.
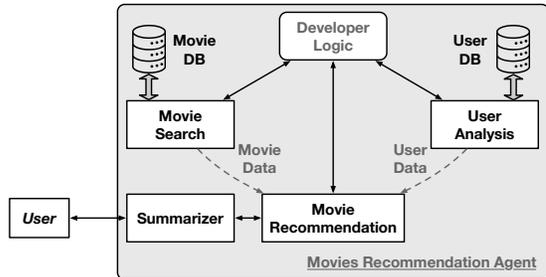

Figure 6: Movie Recommendation System Overview

Movie RecSys implementation. Based on prior works [13,38, 39], we developed a proof-of-concept *AutoGen*-based *MACS* for movie recommendations. Unlike `UseCase#1`, which operates through a dynamic conversation mode (`GroupChat`), this implementation uses a static mode to navigate each process.

Figure 6 illustrates the system architecture and data flow between agents. There are four distinct agents: User Analysis Agent, Movie Search Agent, Movie Recommendation Agent, and Summarizer. The User Analysis Agent first processes user profiles to extract relevant preferences and some personal information. The Movie Search Agent queries the movie database to find relevant movies based on specific criteria derived from the user's query. The Movie Search Agent performs up to 10 keyword-based searches per user query. The Movie Recommendation Agent then analyzes the search results and user profile, along with user queries, to select up to three most suitable movies. Finally, the Summarizer compiles the recommendations, user analysis, and search results into a coherent and user-friendly response.

We developed a tool (`search_movies`) for searching movies and a tool (`get_user_profile`) for retrieving user profiles and binding them to corresponding agents. The movies used in the experiment are publicly available [8] and are stored in JSON format. For experiment simplicity and ethical concerns, we use an LLM to generate user profiles, which contain synthetic information including name, birth, address, etc., together with their corresponding user IDs. User IDs will be concatenated with user queries to enable the User Analysis agent to retrieve the corresponding user profiles.

Experiment setup. We made 13 lines of code (LOC) changes (out of 247) for the above Movie RecSys implementation to enable privacy-enhanced Movie RecSys. Dedicated LLMs designed specifically for recommendation systems have shown great promise, making their adoption a compelling choice for enhancing recommendation tasks [42]. However, as these models are often provided by third-party entities, trust issues may arise. To address these concerns, the data protection manifest for Movie RecSys is designed to meet the following data protection requirements (PRs):

● *PR:* User demographic information, personal preferences, and personally identifiable information (e.g., gender, name, address) from the user profile must be masked before being

sent to external recommendation services or LLMs.

To achieve this, we configured the message flow in the Movie Recommendation Agent to mask sensitive data fields, including user demographic information, personal preferences, and personally identifiable information (e.g., name, age, gender, phone number, email, address, zip code, city, country, nationality, occupation), prior to transmission. ok

Threat setup. We assume that both the agent system and users are benign, while the LLM provider of the Recommendation agent is untrustworthy. For example, the LLM provider is using users' conversations for training purposes without obtaining users' consent. To simulate this attack, we prompted `GPT-4o` to generate 20 user queries. Similar to use case #1, we ensure the existence of user information leakage to external recommendation services or LLMs in the execution trajectories of each query.

Quantitative effectiveness analysis. As shown in Table 1, with *Maris* in place, all privacy violations were correctly detected across all three backbone LLM models. In this case, the messages to verify are relatively straightforward, as user profiles summarized by the User Analyst (will be part of LLM input of the Recommendation Agent) are presented in a clear format, such as: *"User Profile: XXX is a ..."*. Furthermore, the *disallow* data with the type of PII are easily recognized by the LLM, ensuring robust data protection.

Privacy analysis. For the query: *"I'm a college student in Boston and I love animation. I'm looking for a light-hearted comedy to watch with my roommates this weekend. Any ideas?"* This query contains sensitive information such as the user's location and preferences.

Similarly, the User Analysis Agent might summarize user data from the user database as: *"User Profile: Jason Kim is a college student from Boston, born on May 12, 2004. He enjoys Action, Comedy, and Animation films and has rated several titles highly."* Without *Maris*, such sensitive details, including the user's name, location, and birthday, could be exposed to the Recommendation Agent.

With our *Maris*, data safeguards are applied to prevent the exposure of sensitive information to the LLM input of the Recommendation Agent. Sensitive data, such as name, age, and address, are automatically detected and masked to ensure privacy. For example, the user query is masked as: *"I'm a college student in [SENSITIVE_INFO] and I love animation. I'm looking for a light-hearted comedy to watch with my roommates this weekend. Any ideas?"* Similarly, the user profile summary is masked as: *"[SENSITIVE_INFO] enjoys Action, Comedy, and Animation films and has rated several titles highly."*

5.4 Performance Overhead and Utility

The introduction of *Maris* inevitably incurs some performance overhead in *MACS* due to the additional operations required to enforce the manifest. Furthermore, *pet_actions*, such as blocking or masking, may impact the final responses to user queries because of information loss at certain agents in the *MACS*. To evaluate the impact of *Maris* on system performance, we measured the response delay after applying the safeguards. Following prior work [7], we assess the utility of the privacy-enhanced *MACS* by measuring the similarity between responses before and after applying *Maris*.

Metrics. We use the following two metrics, to measure the performance overhead and utility of *Maris*.

• *Average Response Delay (ARD):* The ARD evaluates the impact of the safeguard on system performance, where we measured the response delay using: $\text{ARD} = \frac{\sum_{q \in Q}(t'_q - t_q)}{|Q|}$. Here, Q is the set of queries, t'_q denotes the time taken by the set of agents to accomplish the task for the query q with the safeguard enabled, while t_{q_i} represents the time taken without the safeguard. ARD can measure the average delay per query introduced by the safeguard.

• *Average Response Similarity (ARS):* The ARS evaluates the similarity between agent responses before and after applying safeguards to assess the impact on utility. It is defined as: $\text{ARS} = \frac{\sum_{q \in Q} \text{sim}(r_q, r'_q)}{|Q|}$. Here, Q is the set of queries, r_q is the response to the user query, and r'_q refers to the response to the user query q after applying the safeguard. For similarity measure, we use SentenceBERT [35] for semantic similarity and use BLEU score [32] for token-level similarity.

Experiment Setup. We use the same query dataset generated in Section 5.1 to 5.3. Five queries are randomly sampled for response delay testing, and to ensure consistency in ARD measurement, we evaluate each query three times and compute the average. We set the `cache_seed` parameter in `llm_config` to `None` in the *AutoGen* to eliminate the impact of server caching in measuring the response delay. To comprehensively test response delays, we enforce all three violation response policies (i.e., "block," "mask," and "warning") to each use case. For ARS measurement, all queries are used to assess response similarity, and the final response of the *MACS* is evaluated. We apply the appropriate data protection policies as described in Sections 5.1-5.3. For the baseline, we measure the similarity between two random queries' responses.

Results & Analysis. Table 2 shows the results of ARD and ARS. ARD in all cases are under 40 seconds, indicating that the overhead is minimal and doesn't impact normal usability. Regarding ARS, ARS is generally high in both semantic and BLEU metrics compared to the baseline, which measures the similarity between responses to different queries in the same context (scenario). The generally high value in ARS and much higher ARS values compare to the baseline suggest that the utility of the MACS is not significantly impacted by the safeguards. Notably, in the case of HospitalGPT, which relies on an additional tool (an auxiliary LLM) to generate emails, the generated emails vary significantly across different

Table 2: Average response similarity scores (Semantic and BLEU) before and after applying safeguards.

Use Case	llama3.1-70b			qwen2-72b		
	Semantic **ARS**	*BLEU* **ARS**	**ARD**	*Semantic* **ARS**	*BLEU* **ARS**	**ARD**
HospitalGPT	0.715 (0.498)	0.279 (0.111)	+37.44s (84%)	0.683 (0.390)	0.295 (0.047)	+41.91s (94%)
OptiGuide	0.916 (0.791)	0.847 (0.106)	+16.71s (122%)	0.911 (0.791)	0.862 (0.106)	+32.40s (236%)
Movie RecSys	0.810 (0.465)	0.386 (0.000)	+14.74s (76%)	0.694 (0.592)	0.340 (0.000)	+20.14 (104%)

Table 3: Average Response Delay (ARD) Overhead of llama3.1 by Violation Handling Policy

Use case	Baseline (s)	Policy	Safeguard (s)	ARD (s)
HospitalGPT	43.86 ± 2.59	Warn	60.45 ± 11.32	13.18
		Mask	63.91 ± 4.42	19.73
		Block	42.11 ± 4.18	-1.29
OptiGuide	13.81 ± 0.35	Warn	20.88 ± 3.08	6.79
		Mask	30.80 ± 2.89	16.71
		Block	17.48 ± 2.35	3.60
Movie RecSys	18.42 ± 8.66	Warn	29.10 ± 4.42	9.88
		Mask	34.05 ± 0.56	14.74
		Block	24.91 ± 1.57	5.76

instances. This variation causes the BLEU scores to be lower across all three tested safeguard configurations. However, the semantic similarity remains much higher than the baseline, showing that its utility is preserved.

Table 3 presents the ARDs on llama3.1-70b across various combinations of use cases and violation policies. For each policy setting, we overwrite the original configuration and turn all checkpoints' violation response policies into each of the possible *pet_action*. In most scenarios, the task is completed within twice the baseline time. Notably, the time cost of the blocking policy is comparable to the baseline (without safeguards) and occasionally even faster. Upon manual analysis of the chat history, we find that the agents often abort the task immediately after receiving a blocked message, resulting in less time delay. The relatively larger delay in the masking action is largely due to the additional call of an LLM to mask the sensitive part of the message. The full results of ARD on qwen2 can be found in Appendix B

6 Discussion

Maris **extensions**. Detecting disallowed data items within *Maris* primarily relies on an additional LLM component (Section 3.4), due to its robustness against obfuscation, and context-awareness. However, *Maris* can also be extended to support pattern-based detection using regular expressions, which are effective for identifying data items with specific formats, such as phone numbers, email addresses, and similar structured information.

Regarding *pet_action*, in *Maris*, we allow for three actions,

blocking, masking, and warning. In addition to these, minimizing sensitive information through paraphrasing, as explored in prior work [7], can also be implemented by extending the *manifest enforcer* in our *Maris*. For personally identifiable information (PII) specifically, *Maris* can be further extended to support anonymization and transformation actions, such as replacing sensitive data with synthetic data.

Moreover, in *Maris*, we primarily focus on addressing the data leakage threats posed by *MACS*. However, the underlying design principles can be further extended to address other types of threats in *MACS*. For example, in the case of a jailbreak attack, developers simply need to replace the prompts with a jailbreak attack detection prompt within the *manifest enforcer*, leaving the source code unchanged.

Generality to other frameworks. We implemented *Maris* atop *AutoGen*. However, our design principles are general and can be easily implemented within other popular multi-agent development frameworks. In our design, the *Conversation Handler* was implemented by adding several hook points. A similar strategy can be adopted in other frameworks, such as *CrewAI*, *LangChain*, etc. For instance, we can leverage *CrewAI*'s predefined hooks and callback functions [11] to implement the reference monitor. APIs like `before_kickoff` can be used as hook points to validate the external input and we can naturally deploy the agent-to-environment enforcer here. Similarly, to support *Maris* on *LangChain*, we can leverage its callback functions [19] as hook points. For example, `on_tool_start` can be used as a hook point to monitor the tool inputs, and `on_tool_end` can be used as a hook point to monitor the tool responses.

7 Related Work

Sensitive data disclosure attacks on LLM agent systems. Recent works [14, 21, 40, 49, 50] have explored sensitive data disclosure attacks in LLM agent systems. These attacks mainly leverage the (in)direct prompt injection, where the adversary embeds malicious prompts either in user query or environmental information, to mislead the agents.

Zhan et al. [49] evaluated various agent usage scenarios and demonstrated that agents are vulnerable to malicious instructions embedded in tool responses. It shows that LLM within the agent can easily be misled by such prompts, resulting in harmful actions or the unintentional disclosure of

sensitive information to attackers. Liao et al. [25] introduced environmental injection attacks, where malicious prompts are concealed on websites. These prompts remain invisible to humans but can still influence the agent's decisions. Furthermore, prior studies [14, 40] have also shown that adversarially modified user queries or images can also be used to inject prompts and mislead the agent.

While previous work has primarily focused on agent-to-environment attack vectors, our work broadens this by addressing both agent-to-environment and agent-to-agent attack vectors. Our proposed framework, *Maris*, is designed to safeguard against both threats effectively.

Safeguarding LLM agent systems. To address existing threats to LLM agent systems, researchers have designed defense mechanisms for LLM agent systems [7, 34, 41, 44, 45]. Rebedea et al. [34] proposed Nemo Guardrail, a framework for implementing programmable guardrails that prevent LLM agent systems from generating or processing harmful instructions/prompts. Bagdasarian et al. [7] introduced AirGapAgent, a defense system designed to mitigate data exfiltration attacks caused by third-party prompt injections. AirGapAgent leverages LLMs to detect whether unnecessary PII is being leaked to external environments and minimizes the shared information to prevent unnecessary data exposure. IsolateGPT [44] reframes the agent architecture by isolating each tool in a separate environment to restrict inter-tool information sharing. It relies on a human to verify each information access request from other tools and prevent information leakage through potential attacks like prompt injection. Additionally, other works have proposed defense mechanisms that leverage structured planning within LLM agent systems [41] and supplementary LLM agents [45] to protect agent systems from malicious injections originating from external environments.

However, these approaches largely overlook inter-agent threats within the *MACS*; also, these defenses lack fine-grained control and require significant, often disruptive, implementation efforts to deploy in real-world LLM agent systems. Recognizing these limitations, our research proposes a system-level, end-to-end defense solution that is easy to deploy and provides fine-grained control over both agent-to-agent interactions and agent-to-environment interactions.

8   Conclusion

This paper introduces *Maris*, a pioneering solution to address data leakage threats in *MACS* by enabling privacy controls at the system development level. *Maris* ensures rigorous message flow control and provides a privacy-enhanced development paradigm for *MACS*. We present an end-to-end implementation of *Maris* to demonstrate its effectiveness, low performance overhead, and practicality in privacy-critical *MACS* use cases such as healthcare, supply chain optimization, and personalized recommendation systems. Our techniques will

contribute to significantly elevating privacy and data protection policy compliance assurance for *MACS*, paving the way for secure and scalable multi-agent collaborations.

References

[1] Maris implementaiton. https://sites.google.com/view/savemultiagent, 2024.

[2] AG2 AI. Ag2 ai official website. https://ag2.ai/.

[3] AG2 AI. Ag2 github repository. https://github.com/ag2ai/ag2.

[4] Manar Alohaly, Hassan Takabi, and Eduardo Blanco. A deep learning approach for extracting attributes of abac policies. In *Proceedings of the 23nd ACM on Symposium on Access Control Models and Technologies*, pages 137–148, 2018.

[5] Amazon Web Services. Aws identity and access management (iam). https://aws.amazon.com/iam/.

[6] Gregory R Andrews and Richard P Reitman. An axiomatic approach to information flow in programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(1):56–76, 1980.

[7] Eugene Bagdasarian, Ren Yi, Sahra Ghalebikesabi, Peter Kairouz, Marco Gruteser, Sewoong Oh, Borja Balle, and Daniel Ramage. Airgapagent: Protecting privacy-conscious conversational agents. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, CCS '24. Association for Computing Machinery, 2024.

[8] Rounak Banik. The movies dataset. https://www.kaggle.com/datasets/rounakbanik/the-movies-dataset, 2025.

[9] Junjie Chu, Zeyang Sha, Michael Backes, and Yang Zhang. Conversation reconstruction attack against gpt models. *arXiv preprint arXiv:2402.02987*, 2024.

[10] MITRE Corporation. Synthea: Synthetic patient population simulator. https://synthea.mitre.org/downloads, 2025.

[11] Crew AI. Crew ai callback and hooks. https://docs.crewai.com/quickstart#before-kickoff.

[12] European Commission. The artificial intelligence act, 2025.

[13] Jiabao Fang, Shen Gao, Pengjie Ren, Xiuying Chen, Suzan Verberne, and Zhaochun Ren. A multi-agent conversational recommender system. *arXiv preprint arXiv:2402.01135*, 2024.

[14] Xiaohan Fu, Shuheng Li, Zihan Wang, Yihao Liu, Rajesh K. Gupta, Taylor Berg-Kirkpatrick, and Earlence Fernandes. Imprompter: Tricking llm agents into improper tool use, 2024.

[15] Kai Greshake, Sahar Abdelnabi, Shailesh Mishra, Christoph Endres, Thorsten Holz, and Mario Fritz. Not what you've signed up for: Compromising real-world llm-integrated applications with indirect prompt injection. In *Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security*, pages 79–90, 2023.

[16] IBM Corporation. Ibm guardium. https://www.ibm.com/guardium.

[17] Feibo Jiang, Yubo Peng, Li Dong, Kezhi Wang, Kun Yang, Cunhua Pan, Dusit Niyato, and Octavia A Dobre. Large language model enhanced multi-agent systems for 6g communications. *IEEE Wireless Communications*, 2024.

[18] Leila Karimi and James Joshi. An unsupervised learning based approach for mining attribute based access control policies. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 1427–1436. IEEE, 2018.

[19] LangChain. Langchain: Callbacks. https://python.langchain.com/docs/concepts/callbacks/.

[20] LangChain. Langchain: Build context-aware llm applications. https://www.langchain.com/, 2022.

[21] Donghyun Lee and Mo Tiwari. Prompt infection: Llm-to-llm prompt injection within multi-agent systems. *arXiv preprint arXiv:2410.07283*, 2024.

[22] Beibin Li, Konstantina Mellou, Bo Zhang, Jeevan Pathuri, and Ishai Menache. Large language models for supply chain optimization. *arXiv preprint arXiv:2307.03875*, 2023.

[23] Xing Li, Yan Chen, Zhiqiang Lin, Xiao Wang, and Jim Hao Chen. Automatic policy generation for {Inter-Service} access control of microservices. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3971–3988, 2021.

[24] Yang Li, Yangyang Yu, Haohang Li, Zhi Chen, and Khaldoun Khashanah. Tradinggpt: Multi-agent system with layered memory and distinct characters for enhanced financial trading performance. *arXiv preprint arXiv:2309.03736*, 2023.

[25] Zeyi Liao, Lingbo Mo, Chejian Xu, Mintong Kang, Jiawei Zhang, Chaowei Xiao, Yuan Tian, Bo Li, and Huan Sun. Eia: Environmental injection attack on generalist web agents for privacy leakage, 2024.

[26] LlamaIndex. Llamaindex: Connecting llms to your data. https://www.llamaindex.ai/, 2022.

[27] Mick Lynch. Hospitalgpt: Managing a patient population with autogen powered by gpt-4. https://github.com/micklynch/hospitalgpt. Accessed: 2025-01-06.

[28] Mick Lynch. Hospitalgpt: Managing a patient population with autogen powered by gpt-4 mixtral. https://medium.com/@micklynch_6905/hospitalgpt-managing-a-patient-population-with-autogen-powered-by-gpt-4-mixtral-8x7b-ef9f54f275f1. Accessed: 2025-01-06.

[29] Microsoft. Optiguide: Optimization guide by microsoft. https://github.com/microsoft/OptiGuide. Accessed: 2025-01-06.

[30] Andrew C Myers and Barbara Liskov. A decentralized model for information flow control. *ACM SIGOPS Operating Systems Review*, 31(5):129–142, 1997.

[31] Open Web Application Security Project (OWASP). Owasp top 10 for llm applications 2025. https://genai.owasp.org/resource/owasp-top-10-for-llm-applications-2025/, 2025. Whitepaper.

[32] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002.

[33] HAPI FHIR Project. Hapi fhir public test server. https://hapi.fhir.org/, 2025.

[34] Traian Rebedea, Razvan Dinu, Makesh Narsimhan Sreedhar, Christopher Parisien, and Jonathan Cohen. NeMo guardrails: A toolkit for controllable and safe LLM applications with programmable rails. In Yansong Feng and Els Lefever, editors, *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 431–445, Singapore, December 2023. Association for Computational Linguistics.

[35] N Reimers. Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084*, 2019.

[36] SmythOS. Multi-agent systems in gaming: Enhancing player experience and ai interaction. https://smythos.com/ai-agents/multi-agent-systems/multi-agent-systems-in-gaming/. Accessed: 2025-05-07.

[37] The White House. Executive Order on the Safe, Secure, and Trustworthy Development and Use of Artificial Intelligence. https://www.whitehouse.gov/briefing-room/presidential-actions/2023/10/30/executive-order-on-the-safe-secure-and-trustworthy-development-and-use-of-artificial-intelligence/.

[38] Nagagopiraju Vullam, Sai Srinivas Vellela, Venkateswara Reddy, M Venkateswara Rao, Khader Basha SK, and D Roja. Multi-agent personalized recommendation system in e-commerce based on user. In *2023 2nd International Conference on Applied Artificial Intelligence and Computing (ICAAIC)*, pages 1194–1199. IEEE, 2023.

[39] Zhefan Wang, Yuanqing Yu, Wendi Zheng, Weizhi Ma, and Min Zhang. Multi-agent collaboration framework for recommender systems. *arXiv preprint arXiv:2402.15235*, 2024.

[40] Chen Henry Wu, Rishi Shah, Jing Yu Koh, Ruslan Salakhutdinov, Daniel Fried, and Aditi Raghunathan. Dissecting adversarial robustness of multimodal lm agents, 2024.

[41] Fangzhou Wu, Ethan Cecchetti, and Chaowei Xiao. System-level defense against indirect prompt injection attacks: An information flow control perspective. *arXiv preprint arXiv:2409.19091*, 2024.

[42] Likang Wu, Zhi Zheng, Zhaopeng Qiu, Hao Wang, Hongchao Gu, Tingjia Shen, Chuan Qin, Chen Zhu, Hengshu Zhu, Qi Liu, et al. A survey on large language models for recommendation. *World Wide Web*, 27(5):60, 2024.

[43] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. Autogen: Enabling next-gen llm applications via multi-agent conversation framework. 2023.

[44] Yuhao Wu, Franziska Roesner, Tadayoshi Kohno, Ning Zhang, and Umar Iqbal. Isolategpt: An execution isolation architecture for llm-based agentic systems. *arXiv preprint arXiv:2403.04960*, 2024.

[45] Zhen Xiang, Linzhi Zheng, Yanjie Li, Junyuan Hong, Qinbin Li, Han Xie, Jiawei Zhang, Zidi Xiong, Chulin Xie, Carl Yang, Dawn Song, and Bo Li. Guardagent: Safeguard llm agents by a guard agent via knowledge-enabled reasoning. *arXiv preprint arXiv:2406.09187*, 2024.

[46] Xusheng Xiao, Amit Paradkar, Suresh Thummalapenta, and Tao Xie. Automated extraction of security policies from natural-language software documents. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 1–11, 2012.

[47] Yangyang Yu, Zhiyuan Yao, Haohang Li, Zhiyang Deng, Yupeng Cao, Zhi Chen, Jordan W Suchow, Rong Liu, Zhenyu Cui, Zhaozhuo Xu, et al. Fincon: A synthesized llm multi-agent system with conceptual verbal reinforcement for enhanced financial decision making. *arXiv preprint arXiv:2407.06567*, 2024.

[48] Ling Yue and Tianfan Fu. Ct-agent: Clinical trial multi-agent with large language model-based reasoning. *arXiv preprint arXiv:2404.14777*, 2024.

[49] Qiusi Zhan, Zhixiang Liang, Zifan Ying, and Daniel Kang. InjecAgent: Benchmarking indirect prompt injections in tool-integrated large language model agents. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar, editors, *Findings of the Association for Computational Linguistics: ACL 2024*, pages 10471–10506, Bangkok, Thailand, August 2024. Association for Computational Linguistics.

[50] Boyang Zhang, Yicong Tan, Yun Shen, Ahmed Salem, Michael Backes, Savvas Zannettou, and Yang Zhang. Breaking agents: Compromising autonomous llm agents through malfunction amplification. *arXiv preprint arXiv:2407.20859*, 2024.

A Prompts used for restricted data detection and masking in *Maris*

In our *Maris*, the message is validated in *manifest enforcer* module based on the manifest specifications. We leverage the additional LLM to check if the message contains any restricted items. The full prompt is as follows:

```
Check if this content contains sensitive information in
an {check_type} interaction.
The disallowed items represent categories of sensitive
information to check for, not literal text matches.
For example, if "address" is disallowed, look for any
physical addresses in the content.
SENSITIVE_INFO is a placeholder for sensitive
information, so the message has no sensitive information
but has only SENSITIVE_INFO placeholders SHOULD NOT be
considered as containing sensitive information, and
should be considered as safe.

Rules:
- Categories of sensitive information to check:
{disallowed_item}

Message to check:
{message}
Respond with a JSON object in the following format,
please only output the JSON object:

{
"status": "safe" | "danger",
"violations": ["category1", "category2", ...],
"explanation": "Brief explanation of what sensitive
information was found"
}}
```

If the message is deemed as dangerous, the manifest enforcer will execute one of the *pet_actions*, which includes blocking, masking, or issuing a warning to the developer. If the *pet_action* is masking, *Maris* will further invoke an LLM to mask the restricted data items in the message. The prompt for masking the message is as follows:

```
Mask the following content: {content}
Violations: {','.join(violations)}
Explanation: {explanation}
Please do not add any introductory or concluding text.
```

In this prompt, the message to mask is inserted into the content part, and the restricted data items and explanations are filled using the data output by the restricted data detection LLM with the prompt mentioned above.

B ARD results on llama3.1 and Qwen2

Table 4: Average Response Delay (ARD) Overhead of qwen2-72b by Violation Handling Policy

Use case	Baseline (s)	Policy	Safeguard (s)	ARD (s)
HospitalGPT	43.86 ± 2.59	Warn	77.06 ± 2.75	32.67
		Mask	93.52 ± 13.56	45.08
		Block	62.92 ± 6.03	20.25
OptiGuide	13.81 ± 0.35	Warn	36.18 ± 5.91	20.94
		Mask	45.14 ± 10.12	32.40
		Block	19.01 ± 5.07	4.37
Movie RecSys	18.42 ± 8.66	Warn	39.93 ± 0.98	20.46
		Mask	39.71 ± 0.88	20.14
		Block	35.17 ± 6.80	14.79

C Manifest samples for use cases

C.1 Manifest for HospitalGPT

```
{
  "inter_agent": {
    "agent_transitions": [
      {
        "message_source": "data_analyst",
        "message_destination": "planner",
        "pet_action": "block",
        "disallow_item": [
          "name",
          "email",
          "age",
          "gender",
          "phone_number"
        ]
      },
      {
        "message_source": "data_analyst",
        "message_destination": "epidemiologist",
        "pet_action": "block",
        "disallow_item": [
          "name",
          "email",
          "age",
          "gender",
          "phone_number"
        ]
      },
      {
        "message_source": "data_analyst",
        "message_destination": "critic",
        "pet_action": "block",
        "disallow_item": [
          "name",
          "email",
          "age",
          "gender",
          "phone_number"
        ]
      },
      {
        "message_source": "outreach_admin",
        "message_destination": "planner",
        "pet_action": "mask",
        "disallow_item": [
          "name",
          "email",
          "age",
          "phone_number"
        ]
```

```
      },
    {
      "message_source": "outreach_admin",
      "message_destination": "critic",
      "pet_action": "mask",
      "disallow_item": [
          "name",
          "email",
          "age",
          "gender",
          "phone_number"
      ]
    },
    {
      "message_source": "outreach_admin",
      "message_destination": "epidemiologist",
      "pet_action": "mask",
      "disallow_item": [
          "name",
          "email",
          "age",
          "gender",
          "phone_number"
      ]
    }
  ]
  },
  "agent_environment": {
    "tool_interaction": [
      {
        "message_source": "outreach_admin",
        "message_destination": "write_outreach_sms",
        "pet_action": "mask",
        "disallow_item": [
            "name",
            "email",
            "age",
            "gender",
            "phone_number"
        ]
      }
    ]
  }
}
```

```
          "disallow_item": [
            "name",
            "age",
            "gender",
            "phone_number",
            "email",
            "address",
            "zipcode",
            "city",
            "country",
            "nationality",
            "occupation",
            "income",
            "marital_status",
            "children"
          ]
        }
      ]
    }
  }
}
```

C.2 Manifest for OptiGuide

```
{
  "inter_agent": {
    "agent_transitions": [
      {
        "message_source": "OptiGuideCoffeeExample",
        "message_destination": "user",
        "pet_action": "mask",
        "disallow_item": [
            "supplier's capacity",
            "shipping or transportation cost",
            "roasting cost",
            "coffee demand"
        ]
      }
    ]
  }
}
```

C.3 Manifest for Movie RecSys

```
{
  "agent_environment": {
    "llm_interaction": [
      {
        "message_source": "movie_recommender",
        "message_destination": "llm"
        "pet_action": "mask",
```