# Ahoy: LLMs Enacting Multiagent Interaction Protocols

Omkar Joshi[1][0009−0006−5069−2326], Munindar P. Singh[1][0000−0003−3599−3893], and Amit K. Chopra[2][0000−0003−4629−7594]

[1] North Carolina State University, Raleigh, NC, USA
[2] Lancaster University, Lancaster, UK

**Abstract.** An interaction protocol formalizes how the agents in a multiagent system interact, which facilitates implementing agents. Existing approaches yield agent implementations specific to the selected protocols. *How can we engineer intelligent agents that can enact protocols but are programming-free?* Our contribution, *Ahoy*, addresses this question by creating LLM agents that dynamically select and enact declarative protocols to achieve user goals. We demonstrate that an Ahoy agent can correctly and intelligently enact multiple protocols—concurrently if appropriate to the user goal—without specialized training. Ahoy's significance lies in that it brings together declarative protocols and LLMs, both approaches that promise improved knowledge engineering for agents.

**Keywords:** Agentic AI, Communication Protocols, Information Protocols

## 1 Introduction

A sociotechnical system (STS) is a system of autonomous real-world principals interacting with each other and sharing information and other resources [10]. STSs apply in e-commerce, health, and finance. An STS can be realized as a multiagent system: each agent represents and interacts on behalf of its principal with other agents; no agent is subordinate to a central controller.

An *interaction protocol* operationalizes decentralization by specifying the interaction constraints each agent must respect. Recent work on protocols emphasizes declarative, information-based approaches exemplified by the *Blindingly Simple Protocol Language* (BSPL) [23]. The declarative approaches offer notable advantages over communicating state machine-based approaches [8], including supporting asynchronous, flexible interactions between agents and enabling higher-level meaning through formal properties like commitments [3,27,30]. Being formal, declarative protocols are amenable to model checking [9,24]: thus, a protocol may be verified for correctness before it is adopted.

A programming model facilitates agent implementation. BSPL supports programming models that structure an agent's internal reasoning based on communications sent and received in light of a protocol. Kiko [11], one such programming model, facilitates engineering Python agents; others facilitate engineering

*Belief-Desire-Intention* agents [4,7]. These programming models facilitate building and maintaining agents that are programmed to enact the relevant protocols. The programming effort (e.g., Listing 1.2) concerns encoding an agent's reasoning to take actions made available by a protocol. In contrast, we ask this question: *How can we create agents that without protocol-specific programming effort?*

Agentic AI applies Large Language Models (LLMs) as the reasoning engines of agents. Agentic AI aims to reduce the knowledge engineering effort: rather than manually encoding world knowledge, it leverages the extensive knowledge LLMs have learned from training data. Indeed, LLMs have demonstrated significant ability at diverse tasks [15,16]. Thus, we refine our question to this: *How can LLM agents enact declarative protocols?* If LLM agents could enact interaction protocols, the resulting implementation would be programming-free: LLMs would serve as the agent reasoning substrate.

Accordingly, we contribute Ahoy, an LLM-based approach for creating agents that select and enact BSPL protocols using Kiko. An Ahoy agent takes a user's goals in natural language. The Kiko adapter maintains the states of ongoing protocol enactments and engages the LLM to reason about protocol actions. Based on the user goal and the current state of the agent's enactments, the Ahoy agent sends messages from the relevant protocols using the Kiko adapter to progress towards the user goal. We demonstrate that an Ahoy agent can adopt and enact roles in multiple protocols—sequentially or concurrently—without additional programming. It intelligently enacts flexible protocols and respects protocol constraints. An Ahoy agent can handle external events while doing so, which simplifies interoperation with real-world *event source* by requiring only minor changes to accommodate specific event formats.

## 2 Background

### 2.1 Specifying Interaction Protocols

BSPL protocols specify the interaction between agents using roles, messages, and parameters. Each message (schema) specifies its sender and receiver roles and its parameters. Message schemas are constrained by *parameter adornments* that enforce information causality: agents can only send messages when they have the knowledge required by those messages' adornments. An agent's *local state* includes the history of the role the agent is playing and the set of bindings established thus far. In BSPL, an *enactment* is a complete assignment of bindings to all parameters in all messages, satisfying all adornment constraints. *Key Parameters* uniquely identify an enactment, ensuring that at most one instance of an enactment occurs per unique key binding. Listing 1.1 shows the *Purchase* protocol, which we use to illustrate these concepts.

**Listing 1.1.** The *Purchase* protocol in BSPL.

```
Purchase {
  roles Buyer, Seller, Shipper
  parameters out ID key, out item, out price, out outcome
```

```
4   private address, resp, shipped, satisfaction
5
6   Buyer -> Seller: rfq[out ID,out item]
7   Seller -> Buyer: quote[in ID, in item, out price]
8
9   Buyer -> Seller: accept[in ID, in item, in price, out
        address, out resp]
10  Buyer -> Seller: reject[in ID, in item, in price, out
        outcome, out resp]
11
12  Seller -> Shipper: ship[in ID, in item, in address, out
         shipped]
13  Shipper -> Buyer: deliver[in ID, in item, in address,
        out outcome]
14
15  Buyer -> Seller: completed[in ID, in item, in price,
        out satisfaction]
16 }
```

*Purchase* defines three roles: BUYER, SELLER, and SHIPPER. It declares global parameters (ID, item, price, outcome). The *private* line declares local parameters (address, resp, shipped, satisfaction) that are used by internal messages (schemas) used by specific roles. These parameters remain hidden from the protocol's public interface to encapsulate details irrelevant to the overall interaction. This separation enables BSPL to manage enactment uniqueness while supporting modular protocol composition.

*Parameter Adornments:* Parameter adornments capture the viability of message emissions based on an agent's knowledge.

The ⌜in⌝ adornment marks a parameter whose binding the agent *must already know*. For example, for the *quote* message in *Purchase*, SELLER must already know the value bound to ⌜in⌝ ID and ⌜in⌝ item (received in the prior *RFQ* message) before sending *quote*. A message instance can only be sent by an agent if all its ⌜in⌝parameters are bound in the agent's local state.

Conversely, the ⌜out⌝ adornment marks a parameter whose binding the agent *generates when sending* the message instance—these must be new bindings that did not previously exist in the agent's local state. For the *RFQ* message in *Purchase*, BUYER generates a unique ⌜out⌝ ID and selects an ⌜out⌝ item. Each ⌜out⌝ parameter has exactly one binding per enactment which remains immutable throughout that enactment.

Finally, the ⌜nil⌝ adornment marks a parameter whose binding the agent *must not know*. An agent can only send a message instance when all its ⌜nil⌝ parameters are unbound in the agent's local state. The ⌜nil⌝ adornment enables conditional message flows: certain messages become unavailable once specific bindings are established, creating mutually exclusive enactment paths.

### 2.2 Implementing Protocol-Based Agents

Kiko is a programming model for implementing BSPL agents in Python. It abstracts an agent's communication service, enabling developers to focus on encoding the agent's internal reasoning. To use the Kiko adapter, the developer needs to provide the multiagent system configuration and write one or more *decision makers*. A decision maker is a function that the Kiko adapter invokes in response to specific events, such as when a specific message is received or when information bindings change. The developer is expected to interact with *forms*, which are message templates provided by the adapter. Each form corresponds to a potential message that the agent may send, constrained by protocol rules. To send a message, the agent needs to bind the required ⌜out⌝parameters according to business logic implemented in the agent's decision maker function. This transforms the form (template) into a concrete message instance which can be sent by the adapter.

**Listing 1.2.** Kiko sample showing the @adapter decorator.

```
@adapter.decision(event=InitEvent)
def start(enabled):
    for item in ["ball","bat"]:
        ID = str(uuid.uuid4())
        for m in enabled.messages(RFQ):
            m.bind(ID=ID,item = item)
```

Listing 1.2 demonstrates this pattern through a Python function `start()` that serves as a decision maker for the Buyer agent in the *Purchase* protocol. The function is registered using the `@adapter.decision` decorator and is triggered by an `InitEvent`. When invoked, the adapter passes the set of enabled messages to the function. The decision maker then iterates through this set to find the `RFQ` forms and uses the `.bind()` method to assign values to the ID and item parameters. Upon the decision maker's completion, the adapter automatically collects the completed instances, validates them against the protocol's integrity constraints, and handles their emissions over the network.

### 2.3 Programming LLM Agents

Large Language Models (LLMs) can serve as the reasoning component in agent architectures, enabling agents to plan tasks and make decisions guided by natural language instructions [5,17]. In an LLM agent, the model is invoked repeatedly in a loop, each time conditioned on the user goal, the agent's current state, and the available actions.

Programming an LLM agent typically involves the following components: A system prompt describes the reasoning strategy (e.g., deliberate step-by-step reasoning) [26] and the agent's environment. In Ahoy, the *system prompt* defines the agent's role, protocol constraints that the agent must satisfy (such as respecting message preconditions and parameter types) and any instructions that are unchanged across decisions. Additionally, the system prompt may also contain

stylistic guidelines or formatting instructions to improve response quality. The *user prompt* clarifies the task instance and the relevant, up-to-date context (e.g., dialog history and event description). Because context windows are limited, the information in the system and user prompts should minimize redundancy, with all content structured for clarity.

LLM agents can perform tasks effectively when they can *invoke tools* (e.g., web search, database access, and calculators) and incorporate tool outputs into subsequent reasoning. Recent approaches show that LLMs can be prompted (or trained) to interleave reasoning and acting [29], and use external tools in a standardized way [21].

To enable long-term tasks, agents should maintain and use a persistent memory of past decisions and outcomes. A common approach is *Retrieval Augmented Generation (RAG)*, where the agent retrieves relevant information to augment the next prompt [14]. In practice, memory combines structured state (e.g., key-value stores), historical summaries (e.g., trace compression), and retrieved knowledge from databases (e.g., vector embeddings).

At runtime, the LLM agent executes an observe-reason-act loop formed from these components. Listing 1.7 in the appendix shows an example of this loop.

## 3 Architecture

We make an Ahoy agent programming-free by decoupling LLM reasoning from protocol-specific logic, as shown in Figure 1.

The Ahoy architecture prompts the LLM to reason over constraints and possibilities defined by BSPL protocols and user requirements. This architecture consists of three modules that interact via a defined flow.

- The *Role Selection Module* enables users to select protocols and roles, processes user input, and initializes the Kiko adapter with the MAS configuration.
- The *Prompt Builder* extracts the local state information and formats it into structured LLM input.
- The *LLM Access Function* processes tool invocations and calls the LLM to reason about which messages to send and how to bind their parameters.

During a protocol enactment, as shown in Figure 2, Ahoy runs an event loop. Initially, the user converses with the Role Selection Module in natural language to select the needed protocols, configures Ahoy with a set of roles from the selected protocols, and provides a goal for the agent to achieve. The Role Selection Module instantiates the Kiko adapter for the selected protocol-role pairs and registers the LLM Access Function as a callback. Ahoy then creates a system prompt containing the user goal, foundational knowledge of BSPL semantics, tool and event-handling guidance, and the selected protocols. This process happens once per enactment. During protocol enactment, the Kiko adapter monitors the agent's local state. When a decision event occurs—whether from a message arrival, a change in the set of enabled messages, or an external event from a

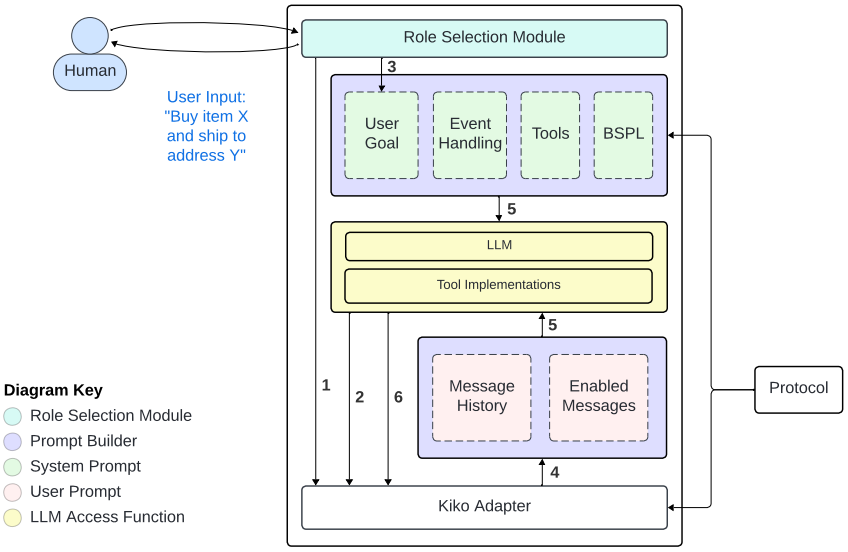


**Fig. 1.** Ahoy architecture and main steps. (1) The Role Selection Module configures the Kiko adapter for the multiagent system. (2) Ahoy registers the LLM Access Function as an adapter callback. (3) The Role Selection Module maps user input to user goal. (4) Adapter detects event; Prompt Builder Module constructs user prompt by accessing the adapter. (5) The LLM Access Function calls the LLM with the system and user prompts received from the Prompt Builder Module. (6) Return selected message instances to the adapter.

connected event source—the Kiko adapter calls the LLM using the registered callback. The LLM selects the messages to send, binds their parameters, and uses the Kiko adapter to send them. The individual modules are explained below.

### 3.1 Role Selection Module

This module presents a list of available protocols and roles to the user, who selects the roles for the agent to play during the current enactment. An Ahoy agent can play multiple roles in multiple protocols at the same time without user guidance on how to implement them. This module is responsible for configuring the Kiko adapter with the roles and protocols selected by the user. To achieve this, the Role Selection Module uses the protocol specifications annotated with comments that explain the function of each message. Examples of the annotated protocol specifications are included in the appendix. Each role needs to be mapped to a *termination condition*. These termination conditions are protocol-defined constraints that specify when a role has finished participating in an

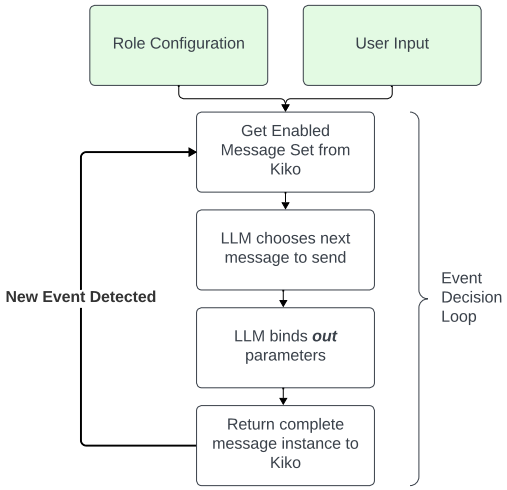


**Fig. 2.** Protocol enactment using Ahoy.

enactment. For example, the termination condition for Buyer in the *Purchase* protocol is sending *completed*; while the termination condition for Merchant in the *Logistics* protocol is receiving *packed*. The Role Selection Module infers the termination conditions from the chosen protocols for the chosen roles. Next, the user describes their goals via a statement of the form "I want to buy an item X with a budget of $Y, deliver it to address Z" that does not need to contain any BSPL or Kiko-related details. The Role Selection Module provides this user input to the Prompt Builder module.

### 3.2 Prompt Builder

This module constructs the system and user prompts from user input, protocol state and external event information. A system prompt is constructed once per protocol enactment. A user prompt is constructed dynamically for each event.

*System Prompt:* We do not require that the models be pretrained on BSPL semantics. The system prompt contains information that enables any LLM to understand and enact protocols specified in BSPL. To do so, we provide the following information as part of the system prompt: (1) a description of BSPL semantics (roles, messages, and adornments) as shown in Listing 1.3, (2) the agent's assigned protocol roles, (3) available tools and their calling conventions, (4) event-handling instructions, (5) protocol specifications annotated with comments that describe each message, and (6) the user input. We provide a full system prompt in Listing 1.9 in the appendix.

*User Prompt:* For each decision event, Ahoy constructs a fresh user prompt by querying the Kiko adapter for information about the ongoing enactments.

**Listing 1.3.** Description of BSPL Semantics.

```
BSPL defines multiagent protocols where agents play roles
    and coordinate via information causality.

PARAMETER ADORNMENTS (three types):
1. **in** (Causal): Must already know from prior messages
   . Information provided from previous messages in the
   protocol.
2. **out** (Generation): You generate this binding; it
   appears once per enactment, creating mutual exclusion.
    Your role generates unique values for instances.
3. **nil** (Negative): Must NOT know this binding. Used
   for mutually exclusive paths where an agent cannot act
    until certain information remains unknown.

Key parameters identify protocol instances. Messages are
   ordered by information flow according to causal
   dependencies.
```

The user prompt includes the message history for the role, the current set of enabled messages, and all parameters and their adornments. This prompt encodes sufficient context for the LLM to make decisions while remaining concise enough to not exceed token limits. A full user prompt is in Listing 1.10 in the appendix.

### 3.3 LLM Access Function

This module is an event handler that mediates between the Kiko adapter and the LLM. It is triggered when a decision event is triggered by either an incoming message, a change in the set of enabled messages, or an external event ("Inventory out of stock alert"). It selects which message to send next and determines what the appropriate parameter bindings should be. When triggered, the module receives the system and user prompts from the Prompt Builder. It then calls the LLM with these prompts. The LLM evaluates the enabled messages and selects which messages to send and their parameter bindings. The LLM Access Function parses the LLM output to extract the messages and parameter bindings, returning the selected message instances to the adapter. Since the Kiko adapter ensures that preconditions are met, the LLM is prompted to reason only about domain logic. The LLM is responsible for handling concerns such as appropriateness ("Is this the right price for this item?") and goal alignment ("Does this item match the user's goals?"). The LLM Access Function also executes tool invocations using the implementations provided by developers and returns their outputs to the LLM. Upon receiving a tool invocation, the LLM Access Function executes the corresponding locally-implemented functions referenced by name on the user's machine and passes the results back to the LLM.

# 4 Programming Model

A key contribution of Ahoy is a programming model centered on the LLM Access Function. This programming model can be used in three ways.

## 4.1 Configuring Ahoy

The user configures an Ahoy agent by selecting from a list of available roles through a chat interface. The user describes the goals in natural language, as explained above. The Ahoy agent then infers the termination conditions from the protocol at initialization and monitors them during the enactment. The user does not need to possess any coding knowledge of protocols, adapters, or LLM functions to configure and use Ahoy.

## 4.2 Adding a new protocol

The user can define new protocols and configure agents for them using Ahoy. This also requires no programming knowledge—the declarative BSPL syntax is sufficient. The user simply needs to specify a protocol and add the appropriate multiagent system configuration for the Kiko adapter.

## 4.3 Extending Ahoy

Developers can extend Ahoy in three ways: (1) adding new tools, (2) implementing custom decision strategies that do not need LLM decision-making, or (3) connecting external event sources. To modify Ahoy, developers need to understand the LLM Access Function pattern shown in Listing 1.4.

**Listing 1.4.** Ahoy LLM Access Function Pattern

```
from bspl.adapter import Adapter
from lib.state_manager import extract_social_state
from lib.llm_client import AnthropicLLMClient,
    choose_and_bind

adapter = Adapter(Seller, systems, agents)
llm_client = AnthropicLLMClient()

async def llm_access_function(enabled_store, event):

    # OBSERVE: Extract current protocol state
    state = extract_social_state(adapter)
    enabled_messages = list(enabled_store.messages())

    # REASON: Delegate to LLM with full protocol context
    message_instance = await choose_and_bind(
        adapter=adapter,
        enabled_store=enabled_store,
```

```
        event=event,
        client=llm_client,
        timeout=30.0,
        current_protocol="Purchase",
        current_role="Seller"
    )

    # ACT: Return selected message for adapter to send
    # (Adapter enforces all constraint validation)
    return message_instance

# Register as adapter decision maker
@adapter.decision()
async def handle_decision(enabled_store, event):
return await llm_access_function(enabled_store, event)
```

The `extract_social_state()` function provides the complete local state extracted from the Kiko adapter. This can be used to access message history and identify bound parameters. Developers can supplement this with data from external event sources to inform the LLM's decision. For example, if an Ahoy agent is enacting *Purchase* as a BUYER, the developer can call a function that returns the inventory, which can be used to decide which item to purchase next. The `enabled_store.messages()` structure contains a list of enabled messages managed by the Kiko adapter. This is the set of choices available to the LLM in terms of interacting with other agents. The `choose_and_bind()` function is used to communicate with the LLM. It calls the prompt construction functions internally. The constructed prompt is then used to call the LLM. By passing the client as a parameter to `choose_and_bind()`, we ensure that this approach is not tied to a specific LLM API and can be ported across different vendors (e.g., Anthropic, OpenAI, HuggingFace). Once `choose_and_bind()` returns a bound message instance, the Kiko adapter validates the message against protocol constraints and sends it to the appropriate recipient. To connect our LLM Access Function with the Kiko adapter, we register it as an event handler following the pattern shown in Listing 1.2.

**Listing 1.5.** Tool invocation in Ahoy.

```
# LLM response can include tool requests for complex
    reasoning
response = await choose_and_bind(...)
# Tool requests are structured as:
# {"choice": null, "params": {}, "tool_requests": [
#   {"tool": "save_state_to_memory",
#    "args": {"agent_name": "Seller", "key": "
    price_strategy",
#             "value": "accept above $50"}},
#    ]}

# Ahoy executes requested tools
```

```
if tool_requests:
    for tool_req in tool_requests:
        tool_name = tool_req.get("tool")
        tool_args = tool_req.get("args", {})

        # Execute tool
        result = await execute_tool_call(tool_name,
            tool_args)
```

The system prompt specifies which tools are available to the Ahoy agent and when each tool may be evoked. Ahoy's LLM Access Function executes these tools locally. This requires the developer to implement local tools (e.g., save-to-memory functions). Local execution mitigates security risks from third-party LLM APIs that have direct execution capabilities. Once the local tool implementations are added to the tool library, the developer can enable LLM access to them by modifying the LLM Access Function following the pattern shown in Listing 1.5. Ahoy supports iterative reasoning by feeding tool results back to the LLM in subsequent calls.

## 5 Evaluation

We evaluate Ahoy through a series of controlled enactments designed to assess these capabilities (all using Anthropic's Claude Haiku 4.5):

**Programming Freeness:** The same agent can enact different protocols correctly without code modification.
**Concurrent participation in multiple protocols:** The same agent can enact multiple protocols as different roles without exceptions or interference.
**Intelligent path selection:** The agent can correctly enact flexible protocols with branching based on user input.
**Handling external events:** The agent can handle events from the Kiko adapter and external sources simultaneously during enactments.
**Preservation of BSPL constraints via adapter enforcement:** Protocol constraints are maintained in all enactments.

With the caveat that this claim is limited to the evaluated scenarios, we find that across all enactments reported in this section: (1) No malformed message instances were emitted; (2) No BSPL adornment violations occurred; (3) No adapter level constraint exceptions were triggered; and, (4) No proposed message was rejected by the Kiko adapter

### 5.1 Programming Freeness

We evaluate whether an Ahoy agent can enact different BSPL protocols without code changes or re-initialization. We first configure the Ahoy agent to enact the *Purchase* protocol as the Buyer role. Using the Role Selection Module, we select the protocol and the role, then specify the item, budget, and delivery address in

natural language. No modification is made to the LLM Access Function, prompt construction logic, or tool infrastructure. The enactment is logged and analyzed. We then configure the Ahoy agent to enact the *Logistics* protocol playing the MERCHANT role. We make the same configuration change, but now specify the orders to be packed and the delivery addresses instead.

- *Purchase*: The agent sent a *RFQ* (Request For Quote) message instance for a pen with the delivery constraints specified, received quotes from competing SELLERs ($5 and $4), and accepted the lowest price. Enactment time: 20 seconds, 4 LLM calls.
- *Logistics*: The agent coordinated label requests and wrapping with implicit recognition of fragile items. Enactment time: 63.55 seconds, 11 LLM calls.

Both protocols were enacted successfully. The agent displayed context-aware behavior (evaluating the options and making the more economical choice as the BUYER, coordinating multiple orders with labeling and wrapping as the MERCHANT), with no specific guidance beyond the provided protocol.

### 5.2 Concurrent Participation in Multiple Protocols

We evaluate whether a single Ahoy agent can simultaneously enact multiple protocols while making coherent decisions across distinct roles. We configure one Ahoy agent to play two roles concurrently: *Purchase*:BUYER and *Logistics*:MERCHANT. The agent receives decision events from both protocols in an interleaved fashion. We verify that Ahoy tracks message history independently for each protocol and that the LLM correctly infers termination conditions (e.g., BUYER must send *completed*). The Ahoy agent transitions from *Purchase* (sending three *RFQ*s) to *Logistics* (requesting labels for purchased items), then back to *Purchase* (receiving quotes, accepting offers, receiving deliveries), and finally to *Logistics* again (requesting wrapping and packaging for purchased items). These transitions are sensible, respecting commonsense reasoning. For example, because the user input specifies all required details, the Ahoy agent sends label requests and RFQs interleaved. After purchasing the glass vase ($42), ceramic plate ($27), and wooden bat ($39), the agent initiates logistics operations: it sends *RequestLabel* for warehouse destinations and *RequestWrapping* for individual items. Both role terminations occur independently without blocking each other. Throughout the 20-message trace, no parameter values leak between protocols and the message history is built correctly at each decision point. We observe the following by inspecting the logs:

- *Purchase* binds three IDs ({`ab534279...`, `0fcd009e...`, `ebbadddb...`}); *Logistics* binds two orderIDs ({`32dc1cf9...`, `6aa820dd...`}). The parameter bindings from these enactments do not affect each other.
- *Purchase*:BUYER completes by sending *completed*(ID=`ebbadddb-...`); *Logistics*:MERCHANT completes by receiving *packed*. Both termination conditions are enforced independently; neither role's completion blocks the other.

- All 20 message instances respect protocol constraints. For example, all messages propagate correct parameter bindings from prior messages (in *Purchase*, *accept* propagates the addresses from *RFQ* correctly), *Logistics* messages maintain unique itemIDs per order.
- *Metrics:* 20 message instances (10 *Purchase*, 10 *Logistics*), 20 LLM calls, 2 concurrent roles, 70 seconds elapsed.

### 5.3 Flexibly Enacting Branching Protocols

We demonstrate that an Ahoy agent can automatically adapt to protocols that offer multiple alternative paths, enabling domain-specific reasoning about protocol flexibility without requiring explicit conditional logic. We enact *FlexiblePurchase* with the Ahoy agent playing FlexibleCustomer. *FlexiblePurchase* enhances the basic two-party purchase protocol with alternative delivery options (*standard_delivery_request* and *express_delivery_request*). The LLM must decide between the two messages based on which one aligns best with user preferences. We test three scenarios by varying the user input.

*Scenario A (Urgent Delivery)*: The Ahoy agent processes an urgent request for a pen purchase (e.g., "I want it delivered ASAP"). The agent correctly selects the *express_delivery_request* message in response to the user preference, avoiding the *standard_delivery_request* message, which is also enabled at the same time. Subsequent messages followed the express branch. Enactment time: 35.23 seconds, 3 LLM calls.

*Scenario B (Cost-Conscious Delivery)*: The Ahoy agent processes a request emphasizing budget constraints (e.g., "I want the cheapest option"). When the two delivery messages are enabled, the agent selects the standard delivery message per the user's budget constraint. Subsequent messages follow the standard path. Enactment time: 35.24 seconds, 3 LLM calls.

*Scenario C (Ambiguous Input)*: When the user input does not specify a delivery preference, the Ahoy agent selects standard delivery. No instruction in the prompt explicitly enforces this choice, confirming that the agent demonstrates reasonable fallback behavior. We analyze the enactment traces for each of the three scenarios to verify that all message instances preserved protocol constraints, including the ⌜nil⌝ adornment that enforces the mutually exclusive nature of the two delivery request messages. Further, neither delivery option is prioritized due to bias in the enabled message set or prompt design, and the LLM output reflected domain reasoning alone. Once a delivery option is chosen, the protocol state advances correctly along that path, without any deadlocks, exceptions, or constraint violations. Enactment time: 35.26 seconds, 3 LLM calls.

### 5.4 Handling External Events

We evaluate whether an Ahoy agent can process externally injected events while a protocol enactment is in progress. Realistically, protocol enactments do not occur in isolation. An agent must respond to external events—information originating outside the current enactment—such as new user requests, inventory

updates, cancellations, or alerts. These events typically originate from event sources connected to the agent and must be incorporated dynamically. If an agent cannot process external events dynamically, developers must either store events for later replay or implement ad hoc interrupt logic. We demonstrate that the Ahoy agent avoids both by handling events seamlessly.

To evaluate this ability, we enact the *Purchase* protocol with the Ahoy agent configured as the BUYER role, while injecting an external event mid-enactment. The scenario unfolds as follows:

- The Ahoy agent begins protocol enactment and sends an *RFQ* for a pen.
- While the protocol is running, an external event is injected into the event queue: "Purchase Request: Buy a trolley" with metadata including item type, delivery address, and budget constraints.
- The agent processes the trolley request in parallel to the pen request after updating the termination conditions. The agent sends an *RFQ* for the trolley, receives and accepts a *quote*, and completes the transaction.

The external event information is stored in a JSON queue and asynchronously consumed, simulating a realistic business scenario where orders arrive dynamically. We observe the following:

- Ahoy correctly loads an external event from the event queue during the second decision, whereas the first decision (before event injection) has an empty queue.
- In 5 out of 6 decisions, the agent is provided with the pending event context in the constructed prompt. The LLM is able to reason about the external event (e.g., "I now have an external event requiring me to buy a trolley ... This is a separate transaction from the pen purchase") and generate appropriate protocol message instances (*RFQ*, *accept*, *completed*).
- Both protocol decisions and event decisions use the same function without any branching logic or conditional code paths.
- The agent sends 6 total protocol message instances across both enactments.
- Each transaction maintains isolated parameter bindings. The input provided by the user for the pen (budget=$20, `address1`) does not interfere with the budget provided by the event for the trolley (budget=$29.99, `address2`).
- All 6 message instances preserve both protocol constraints and user-defined restrictions. The price at which the trolley was accepted is within the provided budget ($29.99).
- *Metrics:* Elapsed time: 13.99 seconds, 6 LLM calls, 6 message instances sent.

This case demonstrates that Ahoy supports *unified event handling*: external events become part of the prompt context at each decision point and do not need separate code paths. The LLM adapts its reasoning from "no external context" (first decision) to "handle injected event" (subsequent decisions) with no code change. This is achieved while the message instances in the original enactment are being processed in parallel. Thus, Ahoy enables an agent to operate in realistic event-driven environments where asynchronous events arrive independently and integrate seamlessly into ongoing protocol enactments.

# 6 Related Work

## 6.1 Multiagent Systems

Recent research combines LLMs with multiagent interaction concepts and approaches. Gatti *et al.* [13] use LLMs to bridge structured Belief-Desire-Intention (BDI) logic and natural human interaction, translating between symbolic reasoning and natural language, and injecting new plans into legacy agents. By contrast, Ahoy leverages LLMs as the cognitive core for agent-agent interaction, and can navigate and enact multiple interaction protocols by reasoning over any BSPL-defined interaction without requiring a modification to agent source code.

Savarimuthu *et al.* [20] propose normative LLM agents that discover and enforce norms, thereby enhancing symbolic reasoning in traditional multiagent systems. Whereas their work focuses on the social and moral constraints of human-agent societies, Ahoy addresses the operational constraints of coordination. Consequently, Savarimuthu *et al.* aim for social flexibility, whereas Ahoy provides a practical architecture for programming freeness.

Ricci *et al.* [19] propose a layer of human-friendly concepts like goals, beliefs, and explanations to promote interpretable, controllable agent behaviors independent of the underlying implementation. Whereas Ricci *et al.* outline the theoretical need for such a layer, we demonstrate practical realization: Ahoy enables agents to participate in multiple interaction protocols concurrently.

Ciatto *et al.* [12] augment BDI agents by automatically synthesizing and adding new procedural plans to their libraries. Our approach differs: we make interaction protocols the core abstraction, decoupling from specific agent programming paradigms like BDI. Rather than programming an agent at runtime with new code or rules to solve specific tasks, Ahoy enables LLMs to act as unified decision makers across diverse protocols with no code change.

## 6.2 Agentic AI

A well-known agentic AI protocol is A2A [1], a delegation protocol enabling task exchange with status and clarification support. A2A targets orchestrated interactions. However, orchestration fails when agents represent autonomous principals. In such settings, an agent cannot simply delegate a task to another. A2A commits a cardinal error of multiagent interaction modeling [6,22]: constraining all agent communication to a small set of communicative acts. For A2A, the set is in fact a singleton, *delegate*. By contrast, Ahoy, via BSPL, accommodates whatever communicative acts are relevant to the application. For example, each of the messages in Listing 1.1 is a distinct communicative act. (Though the Model Context Protocol (MCP) [2] may be applied toward agent communication, its legitimate purpose is supporting tool invocation by agents.)

Some protocols capture specific applications. For example, the *Universal Commerce Protocol* (UCP) supports e-commerce [25]. Agents can enact UCP over diverse transports, including HTTP and A2A. However, UCP lacks a formal specification and uses only request-response patterns.

Current LLM-based multiagent frameworks adopt architectures with fixed roles and predefined communication patterns. AutoGen [28] structures interaction as scripted exchanges between hard-coded agent types (e.g., planner, executor, and tool user); introducing a coordination structure requires code or prompt changes. LangChain requires framework-specific message formats and uses imperative tool invocations, forcing developers to program sequence and control flow. Our approach, by contrast, is declarative. Although effective for task automation, the approaches described above lack a conception of protocols and provide poor interoperability and reuse across domains.

Simulations like Park *et al.* [18] investigate emergent patterns of behavior in autonomous agents instead of formally constraining communication to provide guarantees. In contrast, Ahoy interprets BSPL protocols at runtime, enabling the same agent to enact new interaction patterns simply by loading a new protocol.

## 7 Conclusions

Ahoy demonstrates that LLM-enabled agents can correctly enact multiple protocols without additional programming. Rather than requiring custom programming for for each protocol, an Ahoy agent dynamically reads in BSPL protocols and plays the appropriate roles without requiring additional guidance. Five key capabilities emerge: (1) programming freeness (zero code changes between protocol enactments); (2) multiprotocol participation; (3) protocol flexibility (LLMs can reason about protocols); (4) unified event handling; and (5) safety (no adapter exceptions or constraint violations). The core architectural principle of Ahoy—decoupling constraint enforcement from decision-making—substantially reduces the knowledge engineering effort.

*Future Directions:* Four immediate priorities provide direction to improve the work described in this paper. First, we must develop principled methods for LLMs to automatically select roles from user input alone. Second, rigorous evaluation with baselines and ablation will quantify the effectiveness of Ahoy. Third, Ahoy needs to be tested with models of varying parameter size from different vendors. Fourth, learning from past enactments—whether through in-context approaches, fine-tuning or reinforcement learning—will improve Ahoy's performance. In the long term, user dialog poses an interesting challenge. As agents gain autonomy, when should they ask users before acting? Norms provide a framework: agents should seek confirmation for new commitments (actions binding the user) but not routine discharges of prior commitments. This distinction—critical checkpoint before commitment, not execution—can guide future work in human-agent alignment.

Ahoy demonstrates not merely that programming-free agents are possible, but that this design is both practical and principled. It invites future work where protocols become as fluid and reusable as code libraries, and where LLMs serve as reasoning engines across diverse business use cases. We release the code for Ahoy at https://github.com/OJ98/Ahoy.

## References


1. A2A: Agent2Agent protocol. https://a2aprotocol.ai/ (Apr 2025), last accessed: 2025-08-03
2. Anthropic: Model Context Protocol. https://modelcontextprotocol.io/ (Nov 2024), last accessed: 02/23/2026
3. Baldoni, M., Baroglio, C., Marengo, E., Patti, V., Capuzzimati, F.: Engineering commitment-based business protocols with the 2CL methodology. Autonomous Agents and Multi-Agent Systems **28**(4), 519–557 (2014)
4. Baldoni, M., Christie V, S.H., Singh, M.P., Chopra, A.K.: Orpheus: Engineering multiagent systems via communicating agents. In: Proceedings of the 39th AAAI Conference on Artificial Intelligence. AAAI, Philadelphia (Feb 2025)
5. Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J.D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., Amodei, D.: Language Models are Few-Shot Learners. In: Advances in Neural Information Processing Systems. vol. 33, pp. 1877–1901. Curran Associates, Inc. (2020), https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf
6. Chopra, A.K., Artikis, A., Bentahar, J., Colombetti, M., Dignum, F., Fornara, N., Jones, A.J.I., Singh, M.P., Yolum, P.: Research directions in agent communication. ACM Transactions on Intelligent Systems and Technologies **4**(2), 20:1–20:23 (2013)
7. Chopra, A.K., Baldoni, M., Christie V, S.H., Singh, M.P.: Azorus: Commitments over protocols for BDI agents. In: Proceedings of the 24th International Conference on Autonomous Agents and Multiagent Systems (AAMAS). IFAAMAS, Detroit (May 2025)
8. Chopra, A.K., Christie V, S.H., Singh, M.P.: An evaluation of communication protocol languages for engineering multiagent systems. Journal of Artificial Intelligence Research **69**, 1351–1393 (2020)
9. Chopra, A.K., Christie V, S.H., Singh, M.P.: Requirement patterns for multiagent interaction protocols. In: Proceedings of the 34th International Joint Conference on Artificial Intelligence (IJCAI). pp. 38–46. IJCAI, Montréal (Aug 2025). https://doi.org/10.24963/ijcai.2025/5
10. Chopra, A.K., Singh, M.P.: From social machines to social protocols: Software engineering foundations for sociotechnical systems. In: Proceedings of the 25th International World Wide Web Conference. pp. 903–914. ACM, Montréal (2016)
11. Christie V, S.H., Singh, M.P., Chopra, A.K.: Kiko: Programming agents to enact interaction protocols. In: Proceedings of the 22nd International Conference on Autonomous Agents and Multiagent Systems (AAMAS). pp. 1154–1163. IFAAMAS, London (May 2023). https://doi.org/10.5555/3545946.3598758
12. Ciatto, G., Aguzzi, G., Battistini, R., Baiardi, M., Burattini, S., Ricci, A.: Exploiting GenAI for Plan Generation in BDI Agents. In: Proceedings of ECAI 2025: 28th European Conference on Artificial Intelligence. Frontiers in Artificial Intelligence and Applications, vol. 413, pp. 3495–3502. IOS Press, Bologna, Italy (2025). https://doi.org/10.3233/FAIA251223
13. Gatti, A., Mascardi, V., Ferrando, A.: Let Me Talk to You! Natural Language Interaction Between Humans and BDI Agents via ChatBDI. In: Proceedings of ECAI 2025: 28th European Conference on Artificial Intelligence. Frontiers in Artificial Intelligence and Applications, vol. 413, pp. 3646–3654. IOS Press, Bologna, Italy (Oct 2025). https://doi.org/10.3233/FAIA251242

14. Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., tau Yih, W., Rocktäschel, T., Riedel, S., Kiela, D.: Retrieval-Augmented Generation for Knowledge-Intensive NLP tasks. In: Proceedings of the 34th International Conference on Neural Information Processing Systems. pp. 9459–9474. NIPS '20, Curran Associates, Inc., Virtual (2020)
15. Li, J., Yang, Y., Bai, Y., Zhou, X., Li, Y., Sun, H., Liu, Y., Si, X., Ye, Y., Wu, Y., Lin, Y., Xu, B., Ren, B., Feng, C., Gao, Y., Huang, H.: Fundamental Capabilities of Large Language Models and their Applications in Domain Scenarios: A Survey. In: Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics. pp. 11116–11141. Association for Computational Linguistics, Bangkok, Thailand (Aug 2024). https://doi.org/10.18653/v1/2024.acl-long.599
16. Naveed, H., Khan, A.U., Qiu, S., Saqib, M., Anwar, S., Usman, M., Akhtar, N., Barnes, N., Mian, A.: A Comprehensive Overview of Large Language Models. ACM Transactions on Intelligent Systems and Technology **16**(5), 1–72 (Oct 2025). https://doi.org/10.1145/3744746
17. Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C.L., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., et al.: Training Language Models to Follow Instructions with Human Feedback. In: Proceedings of the 36th International Conference on Neural Information Processing Systems. NIPS '22, Curran Associates, Inc., New Orleans (2022)
18. Park, J.S., O'Brien, J.C., Cai, C.J., Morris, M.R., Liang, P., Bernstein, M.S.: Generative Agents: Interactive Simulacra of Human Behavior. In: Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology. pp. 2:1–2:22. UIST '23, Association for Computing Machinery, San Francisco (2023). https://doi.org/10.1145/3586183.3606763
19. Ricci, A., Mariani, S., Zambonelli, F., Burattini, S., Castelfranchi, C.: The Cognitive Hourglass: Agent Abstractions in the Large Models Era. In: Proceedings of the 23rd International Conference on Autonomous Agents and Multiagent Systems. pp. 2706–2711. AAMAS '24, Association for Computing Machinery, Auckland, New Zealand (2024). https://doi.org/10.5555/3635637.3663262
20. Savarimuthu, B.T.R., Ranathunga, S., Cranefield, S.: Harnessing the Power of LLMs for Normative Reasoning in MASs. In: Coordination, Organizations, Institutions, Norms, and Ethics for Governance of Multi-Agent Systems XVII: International Workshop, COINE 2024, Revised Selected Papers. pp. 132–145. Springer-Verlag, Auckland, New Zealand (2024). https://doi.org/10.1007/978-3-031-82039-7_9
21. Schick, T., Dwivedi-Yu, J., Dessì, R., Raileanu, R., Lomeli, M., Hambro, E., Zettlemoyer, L., Cancedda, N., Scialom, T.: Toolformer: Language Models Can Teach Themselves to Use Tools. In: Proceedings of the 37th International Conference on Neural Information Processing Systems. NIPS '23, Curran Associates, Inc., New Orleans (2023)
22. Singh, M.P.: Agent communication languages: Rethinking the principles. IEEE Computer **31**(12), 40–47 (Dec 1998)
23. Singh, M.P.: Information-driven interaction-oriented programming: BSPL, the Blindingly Simple Protocol Language. In: Proceedings of the 10th International Conference on Autonomous Agents and MultiAgent Systems. pp. 491–498. IFAAMAS (2011)
24. Singh, M.P., Christie V, S.H.: Tango: Declarative semantics for multiagent communication protocols. In: Proceedings of the 30th International Joint Confer-

ence on Artificial Intelligence (IJCAI). pp. 391–397. IJCAI, Online (Aug 2021). https://doi.org/10.24963/ijcai.2021/55
25. UCP: Universal commerce protocol (Jan 2026), https://ucp.dev, last accessed: 2026-02-20
26. Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E.H., Le, Q.V., Zhou, D.: Chain-of-thought Prompting Elicits Reasoning in Large Language Models. In: Proceedings of the 36th International Conference on Neural Information Processing Systems. NIPS '22, Curran Associates, Inc., New Orleans (2022)
27. Winikoff, M.: Implementing commitment-based interactions. In: Proceedings of the 6th International Conference on Autonomous Agents and Multiagent Systems. pp. 1–8 (2007)
28. Wu, Q., Bansal, G., Zhang, J., Wu, Y., Li, B., Zhu, E.E., Jiang, L., Zhang, X., Zhang, S., Awadallah, A., White, R.W., Burger, D., Wang, C.: AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation. In: COLM 2024. COLM '24, Association for Computational Linguistics, Philadelphia (Aug 2024)
29. Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K.R., Cao, Y.: ReAct: Synergizing Reasoning and Acting in Language Models. In: The Eleventh International Conference on Learning Representations. Kigali, Rwanda (2023)
30. Yolum, P., Singh, M.P.: Flexible protocol specification and execution: Applying event calculus planning using commitments. In: Proceedings of the 1st International Joint Conference on Autonomous Agents and MultiAgent Systems. pp. 527–534. ACM Press (2002)

# A Visualizations

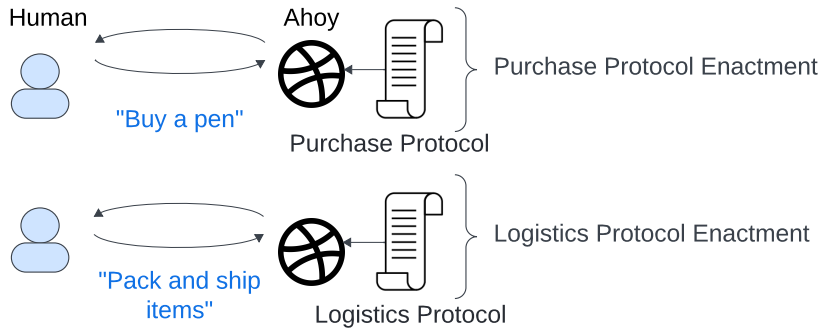


**Fig. 3.** Programming Freeness.

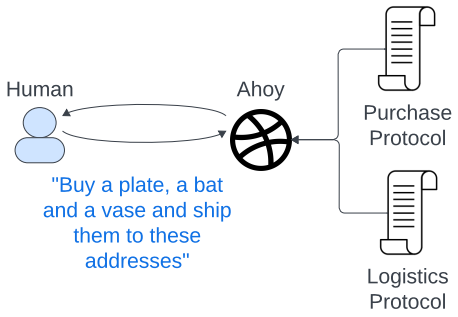


**Fig. 4.** Concurrent participation in multiple protocols.

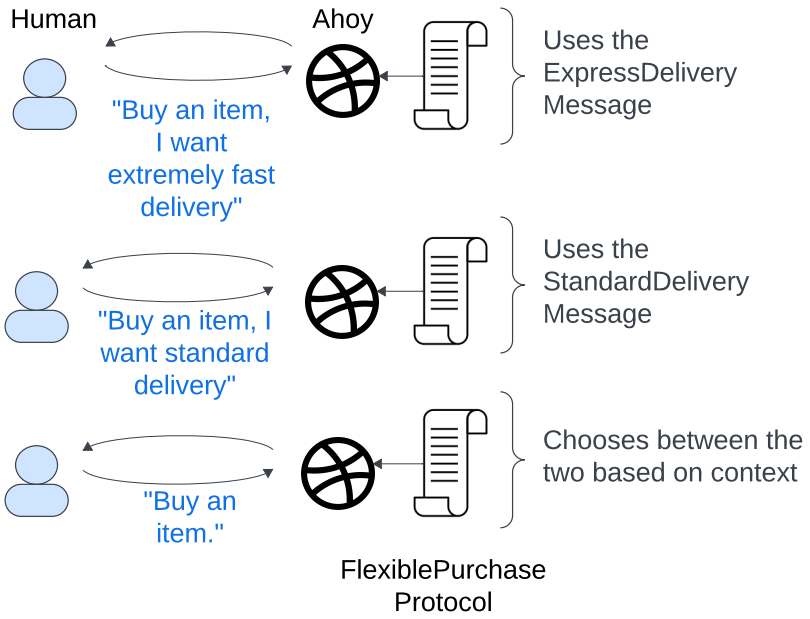


**Fig. 5.** Intelligent path selection.

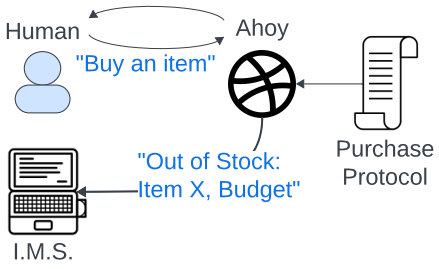


**Fig. 6.** Handling External Events (I.M.S. stands for Inventory Management System, but could be any system that emits events in JSON.)

## B Intelligent Path Selection: Message Sequence Diagram

The user (Caleb) configures Ahoy to play two roles (Logistics:Merchant and Purchase:Buyer) and provides the input as follows:

**Listing 1.6.** Multirole protocol input.

```
I need to buy and redistribute the following items:
- A glass vase upto $100
- A ceramic plate upto $50
- A wooden bat upto $50
Send only one rfq per item.
Delivery location: 123 Main Street, Portland, OR 97201

Once an item is received, they need to be prepared for
    redistribution by wrapping appropriately.

Orders to prepare for shipment:
1. Destination: Alice's House, Items: glass vase
2. Destination: Bob's House, Items: ceramic plate +
    wooden bat
```

As shown in Figure 7, Ahoy then sends the three `RFQ` message instances to request the items and accepts the quotes generated by the seller as the prices fall within acceptable thresholds. Since Ahoy can perform both roles at the same time, it sends label requests to the labeler without waiting for the quotes from the seller. The enactment proceeds with the Labeler sending the labels to the Packer, while Ahoy receives delivery of the items from the shipper. Once the items are received, Ahoy sends the corresponding wrapping requests and finishes up with the Buyer role by sending the completed messages. Finally, the enactment completes when the packed messages for the two orders are received by Ahoy.

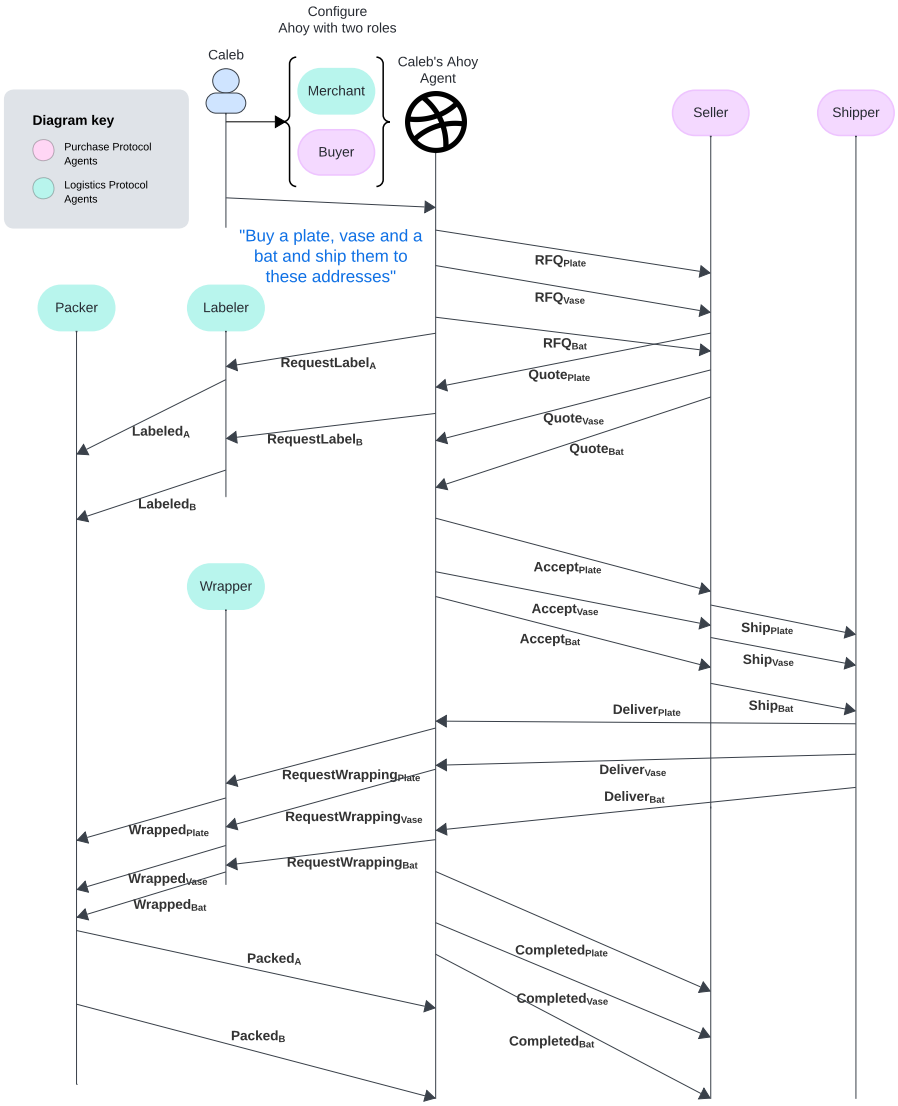


**Fig. 7.** Concurrent participation in multiple protocols.

## C LLM Agent Pattern

**Listing 1.7.** Common LLM agent pattern.

```
llm = AnthropicLLMClient()
state = {"task": "organize items",
"items": ["book", "pen", "cup"]}

# SYSTEM PROMPT: Define agent behavior
system_prompt = """You are a helpful assistant organizing
    items.
You must always pick up the most fragile item first.
Respond with a single item name."""

# OBSERVE: Current state
observation = f"Current items on table: {state['items']}"

# USER PROMPT: The task
user_prompt = f"""{observation} Which item should I pick
   up first? Consider fragility."""

# REASON: Ask LLM what to do
response = llm.complete(
    messages=[{
        "role": "user",
        "content": user_prompt
        }],
    model=MODEL_ID,
    system=system_prompt
)
decision = response.content[0].text
print(f"LLM Reasoning: {decision}")

# ACT: Execute based on decision
if "cup" in decision.lower():
    state["items"].remove("cup")
    state["picked_up"] = "cup"
    print(f"Action: Picked up cup (fragile)")

print(f"New state: {state}")
```

# D Prompts

## D.1 System Prompt Template

The system prompt is constructed dynamically by combining a fixed template with runtime-specific information. The template below shows the structure with placeholders for variable content:

**Listing 1.8.** Ahoy system prompt template (with variables).

```
You are a {agent_names_str} agent.

Your user wants you to fulfill the goal described in input.txt contents:
{user_goal}

To accomplish this goal, you may need to interact with other agents on the
basis of interaction protocols.

The following is an explanation of the environment in which you operate:

BSPL defines multiagent protocols where agents play roles and coordinate via
information causality.

PARAMETER ADORNMENTS (three types):
1. **in** (Causal): Must already know from prior messages. Information provided
   from previous messages in the protocol.
2. **out** (Generation): You generate this binding; it appears once per enactment,
   creating mutual exclusion. Your role generates unique values for instances.
3. **nil** (Negative): Must NOT know this binding. Used for mutually exclusive
   paths where an agent cannot act until certain information remains unknown.

Key parameters identify protocol instances. Messages are ordered by information
flow according to causal dependencies.

EXTERNAL EVENTS: External events represent new tasks that occur during protocol
enactment. Each external event is treated as a separate transaction and may
trigger new message sequences according to the protocol.

TOOLS: save_state_to_memory(agent_name, key, value)
```

```
30
31 RESPONSE: {"choice": 0|null, "params": {...}, "
       tool_requests": [
32             {"tool": "...", "args": {...}}]}
33
34 RULE: Choose if viable. Null only if no options.
35
36 {protocol_definitions}
```

### D.2 System Prompt Instance

Below is a concrete example of a system prompt as instantiated at runtime for a Buyer role in the Purchase protocol with a specific user goal:

**Listing 1.9.** Ahoy system prompt instance (concrete example).

```
1  You are a Buyer agent.
2
3  Your user wants you to fulfill the goal described in
       input.txt contents:
4  I want to buy a pen with a budget of $20 and have it
       delivered to 123 Main St,
5  Raleigh, NC 27606.
6
7  To accomplish this goal, you may need to interact with
       other agents on the
8  basis of interaction protocols.
9
10 The following is an explanation of the environment in
       which you operate:
11
12 BSPL defines multiagent protocols where agents play roles
        and coordinate via
13 information causality.
14
15 PARAMETER ADORNMENTS (three types):
16 1. **in** (Causal): Must already know from prior messages
       . Information provided
17    from previous messages in the protocol.
18 2. **out** (Generation): You generate this binding; it
       appears once per enactment,
19    creating mutual exclusion. Your role generates unique
          values for instances.
20 3. **nil** (Negative): Must NOT know this binding. Used
       for mutually exclusive
21    paths where an agent cannot act until certain
          information remains unknown.
22
23 Key parameters identify protocol instances. Messages are
       ordered by information
```

```
flow according to causal dependencies.

EXTERNAL EVENTS: External events represent new tasks that occur during protocol
enactment. Each external event is treated as a separate transaction and may
trigger new message sequences according to the protocol.

TOOLS: save_state_to_memory(agent_name, key, value)

RESPONSE: {"choice": 0|null, "params": {...}, "tool_requests": [
        {"tool": "...", "args": {...}}]}

RULE: Choose if viable. Null only if no options.

============================================================
PROTOCOL DEFINITIONS (BSPL specs with inline message explanations):
============================================================

--- PURCHASE PROTOCOL ---

Purchase {
  roles Buyer, Seller, Shipper
  parameters out ID key, out item, out price, out outcome
  private address, resp, shipped, satisfaction

  // Buyer initiates: requests a quote for an item
  Buyer -> Seller: rfq[out ID, out item]

  // Seller responds: provides price for the requested item
  Seller -> Buyer: quote[in ID, in item, out price]

  // Buyer accepts: provides delivery address and response/feedback
  Buyer -> Seller: accept[in ID, in item, in price, out address, out resp]

  // Buyer rejects: provides outcome reason and response/feedback
  // (mutually exclusive with accept)
  Buyer -> Seller: reject[in ID, in item, in price, out outcome, out resp]

  // Seller ships: notifies shipper to deliver
  Seller -> Shipper: ship[in ID, in item, in address, out shipped]

```

```
  // Shipper delivers: confirms item delivered to buyer
  Shipper -> Buyer: deliver[in ID, in item, in address,
      out outcome]

  // Buyer completes: sends satisfaction feedback
  Buyer -> Seller: completed[in ID, in item, in price,
      out satisfaction]
}
```

### D.3 Constructed User Prompt

**Listing 1.10.** Purchase protocol enactment: LLM decision.

```
You are agent 'ahoy (as Buyer in Purchase)'. Choose at
    most one option, or return null.
Your role requires making decisions. When choosing an
    option, always provide values for all required
    parameters.

Role: Buyer (in Purchase)

(No pending external events at this time)

=== MESSAGE HISTORY ===

1. rfq (from Buyer to Seller)
   ID: fcba4370-2842-4543-bd31-1acdfb220081
   item: pen under $20 for delivery to Raleigh, NC 27606

2. quote (from Seller to Buyer)
   ID: fcba4370-2842-4543-bd31-1acdfb220081
   item: pen under $20 for delivery to Raleigh, NC 27606
   price: 4

=== END HISTORY (2 messages) ===

Options:
0) Purchase/rfq - out: ['ID', 'item']
1) Purchase/completed [in: ID=fcba4370-2842-4543-bd31-1
    acdfb220081, item=pen under $20 for delivery to
    Raleigh, NC 27606, price=4] - out: ['satisfaction']
2) Purchase/reject [in: ID=fcba4370-2842-4543-bd31-1
    acdfb220081, item=pen under $20 for delivery to
    Raleigh, NC 27606, price=4] - out: ['outcome', 'resp']
3) Purchase/accept [in: ID=fcba4370-2842-4543-bd31-1
    acdfb220081, item=pen under $20 for delivery to
    Raleigh, NC 27606, price=4] - out: ['address', 'resp']

Response format JSON:
```

```
- To choose an option WITH parameters: {"choice": 0, "params": {"ID": "value", "item": "value"}, "tool_requests": []}
- To decline all options: {"choice": null, "params": {}, "tool_requests": []}
```

# E Protocols Used

## E.1 Purchase Protocol

**Listing 1.11.** The Purchase protocol.

```
Purchase {
  roles Buyer, Seller, Shipper
  parameters out ID key, out item, out price, out outcome
  private address, resp, shipped, satisfaction

  // Buyer initiates: requests a quote for an item
  Buyer -> Seller: rfq[out ID, out item]

  // Seller responds: provides price for the requested item
  Seller -> Buyer: quote[in ID, in item, out price]

  // Buyer accepts: provides delivery address and response/feedback
  Buyer -> Seller: accept[in ID, in item, in price, out address, out resp]

  // Buyer rejects: provides outcome reason and response/feedback
  // (mutually exclusive with accept)
  Buyer -> Seller: reject[in ID, in item, in price, out outcome, out resp]

  // Seller ships: notifies shipper to deliver
  Seller -> Shipper: ship[in ID, in item, in address, out shipped]

  // Shipper delivers: confirms item delivered to buyer
  Shipper -> Buyer: deliver[in ID, in item, in address, out outcome]

  // Buyer completes: sends satisfaction feedback
  Buyer -> Seller: completed[in ID, in item, in price, out satisfaction]
}
```

### E.2 Logistics Protocol

**Listing 1.12.** The Logistics protocol

```
Logistics {
  roles Merchant, Wrapper, Labeler, Packer
  parameters out orderID key, out itemID key, out item,
      out status
  private address, label, wrapping

  // Merchant initiates: requests labeling for an order
  Merchant -> Labeler: RequestLabel[out orderID key, out
      address]

  // Merchant initiates: requests wrapping for an item
  Merchant -> Wrapper: RequestWrapping[in orderID key,
      out itemID key, out item]

  // Wrapper completes: item wrapped and ready
  Wrapper -> Packer: Wrapped[in orderID key, in itemID
      key, in item, out wrapping]

  // Labeler completes: label created and ready
  Labeler -> Packer: Labeled[in orderID key, in address,
      out label]

  // Packer completes: all items assembled and packed
  Packer -> Merchant: Packed[in orderID key, in itemID
      key, in item,
            in wrapping, in label, out status]
}
```

### E.3 FlexiblePurchase Protocol

**Listing 1.13.** The FlexiblePurchase protocol.

```
FlexiblePurchase {
  roles FlexibleCustomer, FlexibleMerchant
  parameters out ID key, out item, out price, out done
  private confirmation, payment, standard_delivery,
      express_delivery

  // Customer initiates: requests a quote for an item
  FlexibleCustomer -> FlexibleMerchant: rfq[out ID, out
      item]

  // Merchant responds: provides price for the item
  FlexibleMerchant -> FlexibleCustomer: offer[in ID, in
      item, out price]

  // Customer accepts: confirms the purchase
  FlexibleCustomer -> FlexibleMerchant: accept[in ID, in
      item, in price,
                                          out
                                              confirmation
                                              ]

  // Customer chooses standard delivery: requests
      standard shipping
  FlexibleCustomer -> FlexibleMerchant:
      standard_delivery_request[in ID,
                      in item, in confirmation, out
                          standard_delivery]

  // Merchant confirms standard: sends item via standard
      shipping
  FlexibleMerchant -> FlexibleCustomer: standard_delivery
      [in ID, in item,
                      in standard_delivery, nil
                          express_delivery]

  // Customer chooses express delivery: requests
      expedited shipping
  FlexibleCustomer -> FlexibleMerchant:
      express_delivery_request[in ID,
                      in item, in confirmation, out
                          express_delivery]

  // Merchant confirms express: sends item via express
      shipping
  FlexibleMerchant -> FlexibleCustomer: express_delivery[
      in ID, in item,
```

```
30                      in express_delivery, nil
                            standard_delivery]
31
32    // Customer pays for express: payment for express
          delivery option
33    FlexibleCustomer -> FlexibleMerchant: pay_express[in ID
          , in price,
34                      nil standard_delivery, in
                            express_delivery, out payment]
35
36    // Customer pays for standard: payment for standard
          delivery option
37    FlexibleCustomer -> FlexibleMerchant: pay_standard[in
          ID, in price,
38                      in standard_delivery, nil
                            express_delivery, out payment]
39
40    // Merchant sends receipt: transaction complete with
          receipt
41    FlexibleMerchant -> FlexibleCustomer: receipt[in ID, in
           item, in payment,
42                                              out done]
```