Agent Mesh Communication Protocol (AMCP) – Technical Specification

Abstract: This document provides a comprehensive technical specification of the Agent Mesh Communication Protocol (AMCP), intended as a formal reference for implementers and AI code generation tools (e.g. Copilot, Claude). It covers the formal definition of the protocol, pseudocode for core components, security architecture, public/private endpoint management, integration with multiple event brokers (Solace PubSub+, Apache Kafka, and NATS), illustrative use cases, unit test exemplars in Rust/Java/Python, and deployment guidelines for Kubernetes (both on-premises and cloud). The specification is structured to facilitate correctness reasoning, with clear interface definitions and design rationales.

Introduction

Modern distributed AI systems often consist of **interacting software agents** that must communicate efficiently and reliably. Traditional agent communication protocols (e.g. Google's A2A – Agent-to-Agent, Anthropic's MCP – Model Context Protocol) are primarily synchronous or point-to-point, which can limit scalability in large "agent mesh" networks. In contrast, **AMCP (Agent Mesh Communication Protocol)** is designed from the ground up as an **event-driven, asynchronous publish/subscribe protocol** to enable flexible, loosely coupled interactions among a dynamic set of agents. By leveraging *event mesh* concepts, AMCP supports many-to-many communication patterns, dynamic discovery, and high scalability. [Back Press...nt in Ev 1], [Back Press...nt in Ev 1] [Back Press...nt in Ev 1], [Based on t...ynchronous]

Key Highlights of AMCP:

- **Asynchronous Publish/Subscribe Core:** Agents exchange **events** rather than blocking on request/response calls. Any agent can publish events to *topics* (also called *subjects*) and any interested agents will receive them, enabling multicast and decoupling producers from consumers. This fosters **loose coupling** and easy addition of new agent types without modifying existing ones. [Based on t...ynchronous]
- **Agent Mobility and Lifecycle Management:** Agents in the mesh have a well-defined lifecycle (creation, activation, deactivation, cloning, migration, destruction). The protocol natively supports **agent mobility** (moving an agent and its state between runtime nodes) and allows **scaling out via agent cloning** for load distribution. This dynamic lifecycle is crucial for long-running systems that must re-balance or upgrade agents without downtime.
- **Secure Multitenancy:** AMCP defines a **multi-layer security model** including authentication of agents/contexts, authorization of message flows, and encryption of data in transit. It distinguishes between **public endpoints** (interfaces exposed to external systems or untrusted networks) and **private endpoints** (internal mesh communications), enforcing appropriate trust boundaries.
- **Integration with Event Brokers:** The protocol is **transport-agnostic**, mapping its abstract message semantics onto underlying event broker technologies. We specify integration with **Solace PubSub+**, **Apache Kafka**, and **NATS** (representing a range from enterprise event brokers to cloud-native log and messaging systems). This

ensures AMCP can be deployed in diverse environments while preserving its semantics.
- **Kubernetes Deployment Model:** AMCP is cloud-native and can be deployed on Kubernetes clusters (on-premises or public cloud). We describe a reference architecture using containerized agent runtimes (Agent Context Pods), event broker services (in-cluster or external), Helm chart configurations, service mesh for secure intra-cluster communication, and observability using standard Kubernetes tools.

Following sections present the formal protocol definition (with correctness arguments), pseudocode for core components, security design, endpoint management, event broker integration specifics, use case demonstrations, sample unit tests in multiple languages, and deployment architecture. This document uses precise language and structured formats (diagrams, tables, code blocks) to serve as a blueprint for implementation and verification.

---

1. Formal Protocol Definition and Correctness

In this section, we define the core abstractions of AMCP mathematically and describe the protocol's operation using a formal model. We also outline key **correctness properties** and provide proof sketches to demonstrate that AMCP upholds safety and liveness guarantees. The formalism is kept accessible but rigorous, to guide implementers and allow reasoning about protocol behavior.

1.1 System Model and Entities

Agents and Agent Contexts:

- An **Agent** represents an autonomous computation (an AI model instance, service, or logical actor). Each agent has:
  - A unique **AgentID** (for example, a UUID or composite name) that persists across its migrations.
  - A defined **Type** (determining its code/behavior).
  - An internal **State** (modifiable data encapsulated by the agent).
  - A lifecycle state (Active, Inactive, etc., defined below).
- Agents exist within **Agent Contexts**. An **AgentContext** is a runtime environment (process or container) hosting one or more agents. It provides the local execution sandbox and the interface to the event mesh (broker). Each AgentContext is typically a separate microservice or node in the distributed system.

Events and Topics:

- AMCP uses **events** as the unit of communication. An **Event** (or Message) has:
  - A **Topic** (string name identifying the subject of the event) – e.g. `"order.created"`, `"agent.<AgentID>.control"`.
  - A **Payload** (the content data, which could be structured JSON, binary, etc.).
  - Optional metadata: a **MessageID** (unique ID for deduplication and tracing), **Timestamp**, **Sender** (AgentID of publisher), **TraceID** for correlation (for multi-step workflows), and delivery semantics flags (e.g. requireAcknowledgment).

- **Topics** are hierarchical (delimited by `.` or `/`) in brokers that support it (e.g. Solace, NATS). In Kafka (which lacks wildcards natively), topics are flat strings but we simulate hierarchy via naming conventions (like using `.` in names). Topics enable selective subscription: an agent can subscribe to specific topics or patterns (e.g. `order.*` to get all order-related events, in brokers supporting wildcards).

Publish/Subscribe Mechanism:

- Agents do not communicate by direct references; instead they **publish** events to topics on the event broker, and **subscribe** to topics they are interested in. The broker delivers published events to all subscribed recipients (possibly with filtering, see broker specifics in Section 5). This decouples senders and receivers in time and space.

Control Events:

- In addition to application-level topics, AMCP defines a set of **control topics** and event types for managing agent lifecycle and system functions:
  - `agent.<AgentID>.control` – destination for control commands targeting a specific agent (e.g. to deactivate, migrate, or ping that agent).
  - System broadcast topics like `agents.announce` (when new agent created or moved), `agents.status` (heartbeat or health events), `agents.alerts` (if an agent experiences an error).
  - **Control events** are processed by AgentContexts or infrastructure rather than by the agent's application logic, enforcing separation of concerns.

Agent Directory and Discovery:

- Logically, AMCP maintains the concept of an **Agent Directory**: a mapping from AgentID to current AgentContext location and metadata (capabilities, subscriptions). This directory can be implemented in a distributed fashion (each context knows about agents it hosts and may cache some global info). Discovery of agents occurs via events (e.g. an agent can announce itself on creation) instead of a central registry. The event mesh can thus serve as the discovery mechanism (for example, an agent looking for a service can subscribe to `service.X.announce` topics to find current providers).

1.2 Agent Lifecycle States and Operations

Each agent transitions through specific **lifecycle states**:

- **Inactive:** The agent's code and state exist on some storage or as a snapshot, but it is not currently running. It cannot process events in this state.
- **Active:** The agent is instantiated in an AgentContext and actively processing events.
- **Migrating (Transit):** The agent is in the process of moving between contexts. In this transient state, the agent may be temporarily unavailable or only one side (source or destination) is active at a time to ensure consistency.
- **Terminated:** The agent has been destroyed; its ID is retired and it will receive no further messages.

Valid operations (triggered by control events or API calls) include:

- **Create Agent:** Instantiate a new agent of a given type on a specified context (or let the system choose one). On success, the agent enters *Active* state in that context and an "agent created" event is published (e.g. on `agents.announce`).
- **Deactivate Agent:** Temporarily stop an agent (e.g. for maintenance or before migration). Inactive agents do not receive events, but their state is retained. Deactivation can be local or as part of moving.
- **Activate Agent:** Resume an inactive agent in a context (likely after it was deactivated or moved).
- **Clone Agent:** Create a new agent that is a copy of an existing one (with a new AgentID). The original remains active. Cloning is used for scaling out read-only or parallelizable tasks.
- **Dispatch (Migrate) Agent:** Transfer an agent from one context to another. This involves halting the agent on the source, serializing its state, sending the state to destination, reconstructing the agent on the destination, and resuming it. The AgentID remains the same (not a clone). Proper protocol ensures no messages are lost or processed twice during this.
- **Destroy Agent:** Permanently terminate an agent, freeing its resources and publishing an event (e.g. `agents.depart`).

We now provide a formal description for two crucial operations: **Message Delivery** and **Agent Migration**, and argue their correctness.

1.3 Formal Description of Core Protocol Algorithms

We describe core algorithms in high-level pseudocode, with pre- and post-conditions, to serve as a reference model. These algorithms will later be refined into implementable pseudocode in Section 2.

1.3.1 Message Publish/Delivery

State and Invariants:

- Let $\mathcal{C} = \{C_1, C_2, \ldots, C_n\}$be the set of AgentContexts (each context runs on a node or process).
- Each context $C_i$hosts a set of agents $A_{i1}, A_{i2}, \ldots$. Let function context(a)yield the context hosting agent a. This defines a partition of the agent set.
- Each agent $a$has a set of subscriptions sub(a)(topics it wants to receive events for). Each agent is also implicitly subscribed to its control topic `agent.a.control` (managed by the context).
- We maintain an invariant: **Agent Uniqueness:** An AgentID refers to at most one live agent system-wide, and that agent is at most in one context at a time (except briefly during migration handoff, in which one is inactive) – see Migration below for consistency.

Publish Operation:
When an agent $a$publishes an event $e$with topic t:

1. The agent calls its context's publish API: `AgentContext.publish(topic, message)`.

2. **Broker Routing:** The context's broker client library sends $e$into the event mesh (to topic t). The event broker is responsible for determining which subscribers should get the message:
    - For each context $C_j$that has *any agent subscribed* to t(or whose subscriptions wildcard-match t), the broker forwards $e$to that context.
    - Within a context C_j, the context delivers $e$to each agent a'such that $t \in \text{sub}(a')$(or matches a subscription pattern in $\text{sub}(a')$). Delivery within a context might be queued to each agent's mailbox or delivered via a handler callback.
3. The publish call completes, optionally returning a broker ACK if using guaranteed delivery.

Delivery Guarantees: By design:

- **At-least-once delivery**: If the publishing context gets an ACK from the broker, each subscribed agent's context will receive the message at least once (brokers like Solace or Kafka ensure reliable delivery with acknowledgment). Duplicate messages may occur in rare failover cases, so message handlers should be idempotent or use MessageID for deduplication.
- **Ordering**: Within a single topic, message order is preserved as long as the underlying broker preserves it (Solace and NATS preserve order per topic; Kafka preserves order per partition, so if all messages for a logical topic use one partition key, order is preserved). Cross-topic ordering is not guaranteed, but time stamps and TraceIDs can be used to correlate events.

Pseudo-formal proof of safety (no misdelivery): The broker enforces topic-based routing: an event $e$published on $t$is only delivered to contexts with matching subscriptions, and then only to agents with matching subscriptions. Thus an agent that did not subscribe to $t$will never receive e. This is guaranteed by broker's subscription filter mechanism. We assume the broker's internal subscription propagation is correct (e.g. Solace's DMR forwarding assures a topic reaches all broker nodes where subscribers exist). Therefore, **completeness** (all interested agents get the event) and **isolation** (uninvolved agents don't see it) are upheld.
Liveness: As long as at least one route from publisher to subscriber remains (no full network partition) and the broker and contexts are running, the event will eventually be delivered (brokers typically retry or store events for down subscribers if using persistent delivery).

1.3.2 Agent Migration (Dispatch)

Agent migration is one of the most complex operations in AMCP, requiring coordination between contexts and the broker. We define the migration of agent $a$from source context $C_s$to destination context C_d. For correctness, we must ensure:

- $a$ends up exactly once in C_d(Active) and is removed from C_s.
- No events intended for $a$are lost during transfer.
- Other agents observe at most one copy of a(no inconsistent state where both old and new are active).

Migration Protocol Steps:

1. **Initiation:** C_s(source) receives a command to dispatch $a$to C_d. This could be triggered by an admin or by C_d(load balancer asking to move a).
2. $C_s$sets a's state to *Migrating* (or a specialized *Transfer* substate). It stops giving $a$new events from its queue (freeze consumption).
3. $C_s$captures a's current state data (through agent's serialization interface, e.g. calling a method `serializeState()` on the agent).
4. $C_s$sends a **Migration Request event** to C_d. This can be done via the broker on a control topic like `context.C_d.control` with payload containing:
    - AgentID a, agent Type, and the serialized state.
    - (Optionally) a list of pending messages not yet processed, or a checkpoint ID for message queues (to let $C_d$know what was delivered).
5. $C_d$receives the request. It allocates resources and instantiates a new agent instance $a_{new}$of the given type.
    - It sets a_{new}.state by deserializing the provided state.
    - It marks a_{new}as *Active* (or initially as *Inactive* until fully ready, then flip to Active).
    - It subscribes $a_{new}$to all topics $a$was subscribed to. This step is crucial: the new context needs to register matching subscriptions with the broker (the broker's subscription propagation will update so that events now flow to $C_d$for those topics, and no longer to $C_s$once $a$is gone).
6. $C_d$sends a **Migration Acknowledgment** back to C_s(e.g., on `context.C_s.control`) indicating success or failure.
    - On success, $C_d$now officially hosts a. Any new events for a(e.g. on topic `agent.a.control` or other topics $a$listens to) will arrive in $C_d$because $C_d$subscribed on behalf of a. However, note that due to propagation delay, some events might still be in flight to C_s; we handle this below.
7. Upon receiving ACK, $C_s$completes the migration:
    - It removes or archives the agent a(mark as Terminated in $C_s$and remove subscriptions for $a$in C_s).
    - It may keep a tombstone for a short period to catch straggler messages.
    - It then sends a **Finalization event** (like `agent.a.migrated`) on the mesh, which could be subscribed by monitoring systems or by Agent Directory logic to update the known location of a.
8. If any failure occurs (e.g. $C_d$unable to instantiate), a rollback happens:
    - $C_s$resumes a(since it hadn't terminated it yet, just paused).
    - An error event may be published (`agent.a.migration_failed`) and the system may retry or choose a different C_d.

Handling In-Flight Messages: Because of inevitable race conditions, there could be messages published to a's topics during the migration window:

- To avoid losing messages, $C_s$can buffer any messages for $a$that arrive after step 2 (once $a$was frozen) and forward them to $C_d$after migration (or include them in the state transfer). Alternatively, $C_s$continues to accept messages on a's behalf until it knows $C_d$is active, then forwards those queued messages to C_d.
- Some systems use a short **"dual presence" interval**: after $C_d$activation and before $C_s$removal, have $a$logically present at both contexts but with only one processing events. For example, $C_d$could take over processing new incoming events, while $C_s$just rejects or queues them. Because the broker might not instantly switch all messages to

C_d (especially in Kafka where topic partitions don't move, or Solace DMR propagation has slight latency), this window must be managed. In practice, using a **quiescence period** (don't send new events to a unless urgent) or acknowledgements at the application level can ensure no lost events.

Proof Sketch of Migration Correctness:

- *Exclusive Activation:* By construction, a is never active in two contexts simultaneously. C_s deactivates a before C_d activates it, and only after C_d confirms success does C_s fully terminate. If any events were processed by a in C_s, they happened before the state snapshot; any processed in C_d happen after activation, with no overlap. This ensures consistent state evolution (no two diverging versions).
- *State Integrity:* The state is transferred exactly once and used to initialize the new agent, so all internal variables of a carry over. Any event processed after snapshot on C_s is either reprocessed on C_d or not processed at all, depending on design – to avoid loss, we either freeze early (so last processed event's effects are in state) or include unprocessed events in transfer.
- *No Message Loss:* Assuming the broker's at-least-once delivery, any event published to a's topics will be delivered to either C_s or C_d. The tricky part is those published during the handoff: but our buffering strategy ensures those either remain queued at C_s and get forwarded or they arrive at C_s and find a gone – in which case C_s can forward them because it still has the logic to do so until final removal. Additionally, if using persistent brokers like Kafka, events during migration will remain in the topic; if C_d uses the same consumer group ID for a's subscriptions, it will pick up unprocessed events from where C_s left off (Kafka approach).
- *Delivery Ordering:* Migration can cause reordering around the migration point (some messages might wait for transfer). However, the protocol can preserve causal order for a given publisher-agent pair by quiescing as noted. In general, this is acceptable since migration is typically a management operation not occurring constantly.

Agent Directory Update: After migration, the system (via the finalization event) updates any directory mapping to reflect context$(a) = C_d$. Other agents trying to send a direct message to a (if that's allowed, e.g., via a specific direct topic) will now be routed to C_d. In practice, if direct messaging uses topics, this is automatic because the subscription to `agent.a.control` moved to C_d. For global knowledge, the directory event ensures consistency.

1.4 Correctness Properties and Proof Sketches

We outline the main properties AMCP is designed to guarantee, and provide reasoning (informal proofs) as to why the design and algorithms ensure these properties:
P1. Safety – Unique Agent Context: At any time, an AgentID is **owned by at most one context** (no active duplicates). This prevents confusion or double processing.
Proof (sketch): Initially, when agents are created, the ID is unique by generation (e.g. using UUID ensures uniqueness). On migration, the source context does not release the agent to active state on the destination until it has quiesced locally; after migration, the source deletes it. Concurrent migrations of the same agent are prevented by locking the agent during the process (i.e. the first migration operation will mark the agent as migrating, and a second command would be rejected or queued until done). Thus, there is no point where two contexts believe they actively host agent a.

P2. Safety – Ordered Processing per Agent: Each agent processes incoming events **one at a time in the order they are received** from the broker (sequential processing model per agent).
Justification: While the system is concurrent, each agent can be modeled as a single-threaded actor. The AgentContext ensures that for a given agent, event handlers are not executed in parallel (unless the agent explicitly spawns internal threads, which is outside protocol scope). In pseudocode (Section 2.2), we will show an event loop per agent. Because underlying broker delivers messages in topic order and the context serializes delivery, an agent's internal ordering corresponds to publish order for each topic. If an agent subscribes to multiple topics, those events might intermix, but relative ordering per topic is preserved and the agent can always sequence them by checking timestamps or sequence numbers if needed.
P3. Liveness – Event Delivery: If an agent $a$is subscribed to topic t, and some agent $b$publishes an event on t, then provided no catastrophic failures occur, $a$will eventually receive and process that event.
Justification: This is essentially guaranteed by the brokers we target. For example, in Solace with Guaranteed Messaging, the event will be stored and retried until a(via its context subscriber) acknowledges it. In Kafka, as long as a's consumer is active, the message remains in the log until consumed. In NATS, by default the delivery is best-effort, but we can use NATS JetStream (persistent) to get similar at-least-once guarantees. We assume stable network connections or eventual reconnection. Thus, unless $a$unsubscribes or is destroyed before the event arrives, it should get it. If $a$migrates mid-way, the migration logic ensures the event is not lost (either delivered to old context then forwarded or delivered to new context after subscription update).
P4. Consistency – Migration Transparency: After migrating an agent $a$from context X to Y, other agents will interact with $a$as if it's the same entity (same state, same subscriptions) without needing to know it moved.
Argument: Since the AgentID remains constant and the topics that $a$listens/publishes remain the same, other agents see continuity. For direct interactions, the control channel `agent.a.control` is seamlessly taken over by the new context. The finalization event might inform monitoring systems, but from a functional standpoint, if agent $c$was sending requests or events to a, they continue to do so on the same topic or logical address. There is no need for agent $c$to get a "new address" for a. This property holds because all addressing in AMCP is name-based (ID or topics), not location-based.
P5. Deadlock Freedom: Since AMCP is asynchronous, agents do not block waiting for responses; thus, classic deadlocks (circular waits) are inherently avoided.
Note: Agents can implement request-response patterns on top of events (by sending a request event and waiting for a reply event), but that waiting is typically done as an event callback or via promise/future, not by locking resources needed by others. The protocol itself does not introduce locks between agents. A degenerate deadlock could occur if two agents each wait on a response from the other, but that's a higher-level logic issue, not a network/protocol deadlock. The protocol doesn't amplify such a problem – it would be the same in any asynchronous system. Thus, from the protocol perspective, there is no global lock or resource cycle, ensuring the **protocol layer cannot deadlock**.
P6. Security Properties: (Elaborated in Section 3) Authentication and authorization ensure that a malicious agent or external actor cannot hijack the communications or assume the identity of another agent. Also, data confidentiality is preserved across untrusted networks via encryption. These are not properties of message passing consistency, but of secure operations. We will stipulate a trust model and demonstrate how AMCP meets it (through keys, tokens, ACLs etc.).

Conclusion: The formal model and reasoning above provide confidence that AMCP is a correct and robust protocol for agent mesh communication. Next, we transition to a more concrete description with pseudocode for implementers.

---

2. Pseudocode for Core Components

To guide implementation (and assist AI-based code generation tools), we present pseudocode for AMCP's core components and algorithms. This pseudocode is written in a language-neutral style, using conventions similar to high-level languages (Java/C# like class structures, Python-like indentation for clarity). We cover:

- Data structures and class interfaces: `Agent`, `AgentContext`, `Message` (event), etc.
- Core methods for agent management: `createAgent`, `sendMessage/publish`, `subscribe`, `dispatchAgent` (migration), etc.
- The internal event loop for delivering messages to agents.
- Control event handling (within contexts).
- Illustrative pseudocode for bridging to underlying broker APIs.

This section serves as a reference blueprint that can be directly translated into actual code in various languages.

2.1 Data Structures and Interfaces

First, we define key classes/interfaces in pseudocode form to clarify their properties:

```
// Pseudocode definitions (not tied to a specific language syntax closely)

class Message {
    string id          // Unique ID for message, e.g. UUID for
deduplication.
    string topic       // Topic name for publish/subscribe.
    any payload        // Content of the message (could be structured
object).
    string senderAgent // AgentID of sender (optional, set by context when
publishing).
    string correlationId // For request-reply patterns, to match responses.
    bool  requiresAck  // If true, receiver should acknowledge (for at-
least-once).
    // ... Other metadata: timestamp, headers, etc.
}

class Agent {
    string id        // AgentID (unique across mesh).
    string type      // Logical type or class name of the agent.
    AgentContext context  // Reference to current context hosting the
agent.

    // Agent lifecycle hooks (to be overridden by agent implementation
code):
    void onStart(any initParams) {
       // Called when agent is first created or reactivated.
    }
    void onMessage(Message msg) {
       // Called for each incoming application message.
    }
```

```
    void onStop() {
       // Called just before agent is terminated or deactivated.
    }
    any serializeState() {
       // Returns a serializable snapshot of internal state for migration.
    }
    void deserializeState(any stateSnapshot) {
       // Restores internal state from snapshot (on migration).
    }
}
```

The `Agent` class represents the logic of an agent. In a real system, this might be an interface that agent developers implement. Here it defines how an agent reacts to lifecycle events.

```
class AgentContext {
    string name              // Name/ID of this context (e.g., "node1" or
host address).
    BrokerClient broker      // Connection/client to the underlying event
broker.
    Map<string, Agent> agents // Agents currently hosted, keyed by AgentID.
    SecurityManager security // Reference to security enforcement component
(see Sec. 3).

    // Create a new agent of given type, optionally with initial
parameters.
    Agent createAgent(string agentType, string newAgentId = null, any
initParams = null) {
        require(security.canCreateAgent(agentType));  // security check
        Agent agent = instantiateAgent(agentType)     // create instance
via reflection/factory
        if newAgentId is not null:
            agent.id = newAgentId
        else:
            agent.id = generateUniqueId()
        agent.context = this
        agents[agent.id] = agent
        // Subscribe agent to its control topic for direct commands:
        broker.subscribe("agent." + agent.id + ".control",
this.callbackOnControlMessage)
        // Initialize agent (calls user-defined startup logic)
        agent.onStart(initParams)
        // Announce new agent (optional): publish an event about creation
        Message announce = new Message(topic = "agents.announce", payload =
{"id": agent.id, "type": agent.type, "context": this.name})
        broker.publish(announce)
        return agent
    }

    bool destroyAgent(string agentId) {
        Agent agent = agents.get(agentId)
        if agent is null:
            return false  // no such agent
        require(security.canDestroyAgent(agent)); // check permissions
        // Optionally send a depart event for other agents or cleanup:
        broker.publish(new Message(topic="agents.depart", payload={"id":
agentId, "context": this.name}))
        // Unsubscribe all topics this agent was subscribed to:
        for each topic in agentSubscriptions(agent):
            broker.unsubscribe(topic, this.callbackOnMessage)  // assuming
we track subs
        broker.unsubscribe("agent." + agent.id + ".control",
this.callbackOnControlMessage)
        agent.onStop()    // call agent's custom cleanup
```

```
        agents.remove(agentId)
        return true
    }

    // Publish an event (from an agent or the context itself) to the mesh:
    void publish(Message msg) {
        if msg.senderAgent is null:
            msg.senderAgent = "Context:" + this.name  // context sending
system message
        // If security requires adding signatures or tokens, do here
        broker.publish(msg)  // Delegate to broker client library
    }

    // Send a message directly to an agent by AgentID (essentially a
convenience to publish to control topic):
    void sendToAgent(string agentId, any content) {
        Message ctrlMsg = new Message(topic = "agent." + agentId +
".control", payload = content)
        publish(ctrlMsg)
    }

    // Handle incoming messages (from broker) for general topics (non-
control):
    void onBrokerMessage(Message msg) {
        // Determine which agents in this context should get this message:
        for each agent in agents.values():
            if agentShouldReceive(agent, msg.topic):
                // Deliver asynchronously to agent's queue or directly call
onMessage
                deliverMessageToAgent(agent, msg)
    }

    // Handle incoming control messages (to control agents or context):
    void onControlMessage(Message msg) {
        // Parse control instructions:
        // For example, payload might have a command like {"cmd":
"MIGRATE", "target": "<AgentID>", "destContext": "Name"}
        string cmd = msg.payload["cmd"]
        switch (cmd):
            case "MIGRATE":
                string targetId = msg.payload["target"]
                string destCtx = msg.payload["destContext"]
                dispatchAgent(targetId, destCtx)
            case "CLONE":
                string targetId = msg.payload["target"]
                cloneAgent(targetId)
            case "STOP":
                string targetId = msg.payload["target"]
                // ... etc.
    }

    // Subscribe an agent to a topic (could be called by agent code or
context management)
    void subscribeAgent(Agent agent, string topic) {
        require(security.canSubscribe(agent, topic))
        broker.subscribe(topic, this.callbackOnBrokerMessage)
        recordAgentSubscription(agent, topic)
    }

    // Unsubscribe an agent from a topic
    void unsubscribeAgent(Agent agent, string topic) {
```

```
        broker.unsubscribe(topic, this.callbackOnBrokerMessage)
        removeAgentSubscription(agent, topic)
    }

    // Migration (dispatch) of an agent to another context:
    bool dispatchAgent(string agentId, string destContextName) {
        Agent agent = agents.get(agentId)
        if agent is null:
            return false
        require(security.canMigrateAgent(agent, destContextName))
        // Prepare migration data
        agent.onStop()  // optionally pause agent logic
        any state = agent.serializeState()
        // Send migration request to destination context via broker:
        Message migRequest = new Message(
            topic = "context." + destContextName + ".control",
            payload = {
               "cmd": "MIGRATE_REQUEST",
               "agentType": agent.type,
               "agentId": agent.id,
               "state": state,
               "originContext": this.name
            }
        )
        publish(migRequest)
        // Assume asynchronous and that dest will respond:
        // Temporarily mark agent as migrating:
        markMigrating(agentId, true)
        return true
    }

    // Handle special control message for incoming migration at
destination:
    void onMigrationRequest(Message msg) {
        // This runs on dest context when it receives MIGRATE_REQUEST
        string agentId = msg.payload["agentId"]
        string agentType = msg.payload["agentType"]
        any stateData = msg.payload["state"]
        string origin = msg.payload["originContext"]
        // Instantiate agent locally:
        Agent newAgent = instantiateAgent(agentType)
        newAgent.id = agentId
        newAgent.context = this
        agents[agentId] = newAgent
        newAgent.deserializeState(stateData)
        // Subscribe agent's control and any additional topics:
        broker.subscribe("agent." + agentId + ".control",
this.callbackOnControlMessage)
        // (If we transferred subscriptions list as part of state,
subscribe them too)
        if msg.payload contains "subscriptions":
            for each topic in msg.payload["subscriptions"]:
                broker.subscribe(topic, this.callbackOnBrokerMessage)
                recordAgentSubscription(newAgent, topic)
        newAgent.onStart(null)  // signal agent start in new context (could
be special onMigrate hook)
        // Send ACK back to origin:
        Message ack = new Message(topic = "context." + origin + ".control",
                                  payload = {"cmd": "MIGRATE_ACK",
"agentId": agentId, "status": "OK"})
        publish(ack)
```

```
    }

    // Handle ACK at origin:
    void onMigrationAck(Message msg) {
        string agentId = msg.payload["agentId"]
        string status = msg.payload["status"]
        if status == "OK":
            // Remove the agent from this context
            agents.remove(agentId)
            // Unsubscribe its topics, since dest has taken them:
            // (Alternatively, leave them until broker notices dest subbed?
But typically we unsub to avoid duplicate deliveries.)
            cleanupSubscriptions(agentId)
            markMigrating(agentId, false)
            // Optionally inform logging/monitoring
        } else {
            // Migration failed, re-activate agent if needed
            markMigrating(agentId, false)
            agents[agentId].onStart(null)  // resume agent
        }
    }
}
```

Explanation: In `AgentContext`, we see methods to create and destroy agents, publish messages, subscribe agents to topics, and the core of migration (`dispatchAgent` and handling of migration request/ack). The pseudocode demonstrates security checks at various steps (using a hypothetical `SecurityManager`), though the details of that come later.
One important design shown: the `AgentContext` registers a **single callback** with the broker for receiving messages (`this.callbackOnBrokerMessage` for normal topics, and `this.callbackOnControlMessage` for control topics). This implies the context is the actual subscriber from the broker's perspective. It then internally dispatches messages to the correct agent(s). This is typical for setups where each context has one connection to the broker that multiplexes all its agents' traffic (for efficiency, rather than each agent having a separate connection).
Agent Handler Loop: The pseudocode above calls `deliverMessageToAgent(agent, msg)`. We can imagine that each agent has an internal queue or it can directly handle calls (depending on concurrency model). For clarity, we can model that each context runs a **dispatcher thread** per agent or a global thread pool that invokes `agent.onMessage(msg)` sequentially for each message. Here's a conceptual illustration:

```
function deliverMessageToAgent(Agent agent, Message msg) {
    // If using a simple single-thread per agent:
    agent.onMessage(msg)
    // If more sophisticated, might enqueue:
    // agentQueue[agent.id].enqueue(msg)
}

// If queuing, somewhere a loop:
function agentEventLoop(Agent agent) {
    while (agent.isActive) {
       Message msg = agentQueue[agent.id].dequeue()
       if msg != null:
           agent.onMessage(msg)
    }
}
```

The exact mechanism might vary, but the result is that `Agent.onMessage` gets called for each incoming event in order.

2.2 Core Protocol Workflows in Pseudocode

Next, we present the pseudocode for major workflows described formally in Section 1, to show step-by-step logic:

(a) Agent Creation Workflow:

```
// Example usage
AgentContext ctx = ... // assume current context
Agent newAgent = ctx.createAgent("RecommendationAgent", null,
{"model":"GPT-4"})

// Pseudocode inside createAgent (already shown above):
```

The `createAgent` method ensures security and then instantiates the agent. It subscribes to the agent's control topic. If the agent has default topics it should subscribe to (perhaps based on type), the context could automatically subscribe them here. For example, certain agent types might always subscribe to a common topic (like an "alert" topic for all monitoring agents). This can be part of initialization logic.

(b) Message Publish and Subscribe:

Agents will typically have an API or internal method to publish events and to subscribe to topics, which likely call into context:

```
class Agent {
    // ...
    void publishEvent(string topic, any content) {
        Message msg = new Message(topic=topic, payload=content)
        msg.senderAgent = this.id
        this.context.publish(msg)
    }
    void subscribeTopic(string topic) {
        this.context.subscribeAgent(this, topic)
    }
}
```

So an agent can do `this.publishEvent("orders.processed", orderResult)`, etc. The context's broker client handles dispatching.

(c) Agent Migration (Dispatch) Implementation:

Breaking down how the pseudocode we gave in AgentContext cooperates between origin and destination:

- Origin context calls `dispatchAgent(targetId, destCtxName)`. It serializes state and publishes a `MIGRATE_REQUEST` to `context.<destCtxName>.control`. In practice, `context.X.control` could be a topic where each context subscribes with its name – i.e., each context's control channel. *We assume the broker has topics that allow addressing contexts (like "context.node1.control").* Each context subscribes to its own control topic at startup.
- Destination context, having subscribed to `context.<itsName>.control`, receives the MIGRATE_REQUEST in `onControlMessage` or specifically in `onMigrationRequest`. It then calls `instantiateAgent` to create an object of the right type. It sets the same AgentID and state. It subscribes the agent's control topic (so it can get further commands). If the migration message included a list of topics the agent was subscribed to, it will subscribe them here as well (maintaining the agent's interests).
- Destination context replies with `MIGRATE_ACK` (with status "OK" or error).
- Origin context's `onMigrationAck` sees status OK, then it cleans up: remove agent from local list, unsubscribe topics, and free resources.

We should also consider concurrency in pseudocode: What if multiple threads call these? Possibly use synchronization (not shown for brevity). Real code should lock agent maps during changes, etc., but conceptually it's straightforward.

(d) Security Hooks:

We see calls like `security.canCreateAgent(agentType)` etc. The `SecurityManager` could be an interface ensuring that context respects policies (e.g. only certain agent types allowed, or requiring an auth token for external requests to create agent). We will detail the security model next.

(e) Public vs Private Endpoint Distinction in Code:

Public endpoints might be exposed via a different interface, e.g., an HTTP server that receives a REST call and then calls `AgentContext.createAgent` or `publish`. The pseudocode doesn't explicitly show that, but we will address conceptually in Section 4.

(f) Integration with Brokers:

We have an abstract `BrokerClient` in the pseudocode. In an actual implementation, this could be:

- A Solace JMS or MQTT API wrapper.
- A Kafka producer/consumer in the context, where the context consumer subscribes to topics for all its agents (potentially using a regex subscription if Kafka supports it via its consumer API for multiple topics).
- A NATS client where the context subscribes with wildcards to all topics needed (like if an agent subscribes to `orders.*`, the context's NATS connection does that subscription).

We will detail in Section 5 how these brokers are configured, but pseudocode remains mostly the same regardless of broker – except the broker might not support wildcards (Kafka) or might require separate consumer groups (Kafka's model of multiple clients) versus one callback (Solace/NATS model of event callback per message).

2.3 Internal Data Management

We should specify how subscriptions are tracked internally:

- The context likely maintains a map of AgentID to list of topics subscribed (for cleanup and possibly for quick routing decision in `onBrokerMessage`).
- Alternatively, the context could maintain a map of topic → list of local agents subscribed (which would make `onBrokerMessage` simply do a lookup by topic to dispatch to those agents).

Given dynamic subscription/unsubscription, these structures must be updated accordingly. Example:

```
Map<string, Set<Agent>> topicSubscribers; // within AgentContext

function recordAgentSubscription(Agent agent, string topic) {
    topicSubscribers[topic].add(agent)
    // Might also store in agent -> topics map if needed
}
function removeAgentSubscription(Agent agent, string topic) {
    topicSubscribers[topic].remove(agent)
}
function agentShouldReceive(Agent agent, string topic) -> bool {
    // Check if agent is subscribed (exact or via wildcard)
```

```
    // If wildcard subs allowed, need to match topic against agent's
patterns:
    for each sub in agentSubscriptions(agent):
        if matches(sub, topic):
            return true
    return false
}
```

However, if the broker already filters (which it does - the context receives only messages on topics that someone subscribed), possibly the context doesn't need to double-filter comprehensively. For instance, in Solace, if no agent in context subscribed to `orders.delayed`, the context wouldn't have subscribed, so it wouldn't receive those messages at all. In Kafka, the context's consumer might subscribe to many topics (some possibly idle if no agent currently cares; Kafka doesn't have wildcards to auto-add, so context would subscribe each time an agent subscribes). For reliability, our context code does a check to be safe.

2.4 Pseudocode Summary

The pseudocode provided outlines the **core logic** of AMCP. It can be summarized as:

- **AgentContext:** Manages agent instances and acts as the mediator to the broker.
- **BrokerClient:** Abstract interface to publish/subscribe events (to be implemented per broker type).
- **Agent:** Contains user-defined behavior and state handling.
- **SecurityManager:** (to be defined) enforces rules on operations.

Implementers can translate this into concrete languages. In Section 7, we will show how unit tests might look in Rust, Java, Python, which indirectly also indicates how the code could look in those languages.

---

3. Security Model

Security is paramount in multi-agent systems, especially when some agents might be third-party or when bridging to external networks. AMCP's security model is **multi-layered**:

1. **Authentication:** Ensuring that only legitimate entities (agents, contexts, or external clients) can join or interact with the agent mesh.
2. **Authorization:** Controlling what actions and topics each entity is allowed to perform or access (e.g., an agent might be restricted to only publish on certain topics, or only a "manager" agent can dispatch others).
3. **Encryption:** Protecting data in transit across networks via encryption (both at transport level and optionally at message payload level for end-to-end confidentiality).
4. **Trust Boundaries:** Defining clear separation between trusted internal communications and untrusted or semi-trusted interfaces (public vs private endpoints, cross-domain exchanges).
5. **Audit and Monitoring:** Keeping track of security-relevant events (auth attempts, policy violations) for analysis and compliance.

We describe each aspect and how AMCP implements or leverages existing solutions for it.

3.1 Authentication

Agent Identity: Each agent should have a form of identity that can be cryptographically verified, especially when agent instances might be created dynamically. In AMCP, an agent's identity is primarily its AgentID (which is unique). However, an AgentID can be guessed or forged unless tied to credentials.
Context Authentication: Since in our design, agents communicate *via contexts and broker*, we focus on authenticating contexts and any external clients:

- **Context Authentication to Broker:** Each AgentContext (or the process it runs in) typically connects to the broker using credentials (e.g. username/password, client certificate, or token, depending on broker). This ensures that only authorized contexts can subscribe/publish on the event mesh. For instance:
    - Solace: Could use client username/VPN credentials or OAuth tokens for connecting.
    - Kafka: Uses SASL (e.g. Kerberos or SCRAM) or mTLS for client auth.
    - NATS: Can use NKeys (public-key based identity) or user, and support JWT for client auth.
- **Inter-context Authentication:** If contexts communicate via direct channels (e.g. if two contexts talk peer-to-peer for migration bypassing broker), they must mutually authenticate (likely using mTLS if direct gRPC/HTTP, or piggyback on broker's auth if still via control topic).

Agent Authentication within Context: If multiple agents run in one context, and especially if those might be multi-tenant (e.g. different authors or trust levels), the context needs to ensure an agent cannot fake another's identity:

- When an agent calls `publish`, the context should either tag the message with the agent's ID (as we did with `msg.senderAgent`) or use separate credentials per agent when publishing to broker (which is usually not the case; typically one connection per context).
- A **SecurityManager** in the context can be given each agent's credentials at creation. For example, when an agent is created, you could require an auth token that identifies the code or the user initiating it.
- If agents come from an external request (public interface), that request likely had an authentication (like a user API token); the context can propagate that identity to limit what the agent does (act on behalf of that user).

In summary, contexts and brokers handle most authentication. Agents are somewhat sandboxed by context – if context itself is compromised, agent identity inside is moot, but assuming context is trusted, the main gate is controlling what code can run or what external calls can be made.
External Client Authentication: If something outside the mesh (say a web client or external system) wants to interact (send an event or request to an agent via a public endpoint), it must authenticate via an API gateway or broker credential:

- For a **public HTTP endpoint** exposing part of AMCP, use standard web auth (OAuth2, API keys, etc.) at the gateway. The gateway then, if authorized, injects an event into AMCP on behalf of that client (possibly including a client identity in the message).

- For **direct broker access** by external publishers/subscribers (like IoT devices connecting to event mesh), use the broker's client authentication. For example, an IoT device could connect to a dedicated external-facing MQTT broker that bridges into the AMCP mesh (with ACL ensuring it can only publish/subscribe to allowed topics).
- Cases like NATS or Kafka: might not directly expose them publicly without a proxy – we likely have a **bridge agent** that the external world can talk to (thus the external world only sees a specific interface, not the entire mesh).

3.2 Authorization

Scope of Authorization: We enforce rules at multiple levels:

- **Topic-level access control:** Determine which topics each agent or external actor can publish to or subscribe from.
- **Action-level permissions:** Determine which control actions a context or agent can invoke (e.g., not every agent should be allowed to migrate other agents).
- **Resource quotas:** Possibly limit how many agents a particular user can create, or how much load one agent can generate (to prevent abuse).

Implementation Mechanisms:

- **Broker-side ACLs:** Many brokers allow defining access control lists on topic namespaces per client. We can leverage these:
    - Solace: Supports granular topic permissions per client username or client-profile.
    - Kafka: Supports ACLs on topics (who can read/write a given topic, by principal).
    - NATS: With JWT auth, you can include pub/sub permissions on specific subjects in the token.

  In AMCP, because contexts connect as clients, we might give each context a broad access if it's trusted environment. Within a context, it's the context's job to ensure an unprivileged agent doesn't misuse that connection.

- **Security Manager in Context:** The pseudocode references a `SecurityManager`. This component can enforce policies such as:
    - Only certain agent types can be created (e.g., if user tries to create an agent of type "AdminAgent" and they're not allowed, reject).
    - When an agent calls `subscribeTopic` or `publishEvent`, check the topic against allowed patterns. For example, an agent might have a property listing allowed topics or a role that implies what it can do. If it violates, throw an exception or drop the message.
    - Manage **filesystem or system access**: If agent code tries to access local disk or network outside broker, the context should sandbox it. This is more of a runtime containment (e.g., running each agent in a separate OS process or thread with limited privileges). While not directly part of AMCP protocol, the specification should recommend sandboxing untrusted agent code.
- **Trust Zones (Public vs Private)**:
    - Internal communications (context-to-broker, context-to-context) happen on a protected network (e.g., an overlay network with strict auth). All internal

topics (like `agent.*.control`, `context.*.control`, etc.) are not accessible to external clients because external ones connect only to designated endpoints (discussed in next section).
    - External interactions are funneled through controlled agents or services that validate inputs.

Examples:

- A **Token-based SecurityManager**: We could implement the context to require a token when calling `createAgent` via external API. That token might carry a scope (like "user_id:123 can create agent type Recommender with id prefix 123-*"). The context passes the token to SecurityManager, which checks and possibly stores it as "ownership" of the agent, so later it can check if that agent tries to do unauthorized things (like interacting with another user's agent).
- A **Path/Resource restriction**: If an agent has to access a file or an HTTP URL, perhaps a special agent capability, the SecurityManager could restrict allowed domains or directories (this is beyond core comms but part of sandboxing).
- In code generation context, one might implement pluggable `SecurityManager` strategies (e.g., RoleBasedSecurityManager, TokenSecurityManager, etc.).

3.3 Encryption (Transport and End-to-End)

- **Transport Encryption:** All communication between contexts and brokers, and between any clients and brokers, should use TLS (or equivalent).
    - Solace: supports TLS for all client connections.
    - Kafka: supports TLS for brokers and client connections.
    - NATS: supports TLS easily.
    - On Kubernetes, deploying these brokers typically allows enabling TLS or running within cluster network (but even then, using TLS prevents someone sniffing a pod network).
- **Mutual TLS (mTLS):** For context-to-broker, in enterprise, mTLS can be used to authenticate contexts by certificates. In a Kubernetes scenario with service mesh (Istio/Linkerd), mTLS is often automatically applied between services, adding another layer even if broker has its own TLS.
- **Message Payload Encryption:** In high-security scenarios, sensitive data within events might be encrypted by the sender and only specific receivers can decrypt (this is end-to-end encryption on top of broker). For example, two agents might share a symmetric key and encrypt the payload. The broker still routes by topic but cannot read content. Or use public-key crypto: publisher uses receiver's public key to encrypt.
    - This is optional and depends on needs. The spec can mention it as a possibility (e.g., if regulated data is sent through the mesh, design should include an encryption scheme).
- **Storage Encryption:** If brokers persist messages (Kafka, Solace with persistent queues, NATS JetStream stores), ensure those stores are encrypted at rest (disk encryption or broker-provided encryption features). Kafka can encrypt data at rest or rely on disk encryption at OS level.

3.4 Trust Boundaries and Public vs Private Endpoints

This overlaps with Section 4, but from security perspective:

- The **internal mesh** (contexts, internal broker endpoints) is a trusted domain. Only authenticated services should be there.
- The **public edge** (exposing some functionality to external systems) should be limited to specific gateways or brokers that perform validation.

We might incorporate a **Gateway Agent** pattern: a special agent (or a set of them behind a load balancer) that is reachable via public API. For example, a web service could be implemented as an agent that subscribes to `web.request.*` topics. External clients (after auth) can publish to `web.request.XYZ` through a dedicated broker endpoint, and the gateway agent will handle it, possibly interacting with internal agents, and then respond.
The **SecurityModel Guarantee:** The idea is to minimize direct exposure. Even if someone manages to connect to the internal broker, without proper credentials they can't do anything meaningful (auth required). And even if they publish some message, if not authorized, contexts might drop it by security rules.
We will emphasize:

- Use a **Zero Trust approach** within the mesh as well – assume any message could be malicious. E.g., an agent should validate input from another agent if not fully trusted (though often we consider all internal as trusted cooperating services, there could be compromised or buggy ones).
- Logs and monitors should detect unusual patterns (maybe an agent publishing to a topic it never did before could signal it was compromised – out of scope of spec, but a possible extension).

Session Security and Rotation: If the system runs long, credentials (like tokens or certs) should be rotated, and compromised keys should be revocable (e.g., using short-lived tokens via OAuth and an identity service for contexts).

3.5 Example Security Policy Outline

To illustrate, here's a hypothetical scenario:

- Each context has a **context identity certificate** issued by a central authority. Brokers trust that CA, so any context connecting with a valid cert is allowed (some mapping from cert to context name).
- Each agent type is associated with a role or permission set. E.g., "SensorAgent" role can publish on `sensors.*` topics, subscribe `alerts.*` but cannot create new agents or call migration. "ManagerAgent" role can do migration commands and subscribe to all topics.
- The context's SecurityManager has a configuration that maps agent type to allowed topics and actions.
- When deploying an agent, you mark which role it needs, and context enforces accordingly.
- If an external user wants to create an agent on the fly, they must call an API with a token that allows agent creation of certain type. The context will verify that token and allow creation, then tie that token's identity "User123" to the agent's context (maybe store in a map of agent->owner). Later if that agent tries something that user wouldn't be allowed (like reading someone else's data topic), the SecurityManager can block it.

This layered approach should be clearly communicated in any implementation derived from this spec.

3.6 Summary of Security Considerations

In summary, AMCP's security framework ensures:

- Only vetted participants join communications (through auth).
- Participants have the minimum necessary privileges (through topic/action authorization).
- Data remains confidential and tamper-proof in transit (through encryption and potential signatures).
- The design anticipates both accidental misuse (bugs) and malicious actors, containing each agent's capabilities through the context mediation (no direct uncontrolled network access from agent code).

Next, we consider how this model differentiates **public vs private endpoints** in practice.

---

4. Public vs Private Endpoint Management

Large systems often need to expose some functionality to external systems or users (public), while keeping the internal mechanism and communications private. AMCP supports this via clear separation of **public endpoints** and **private mesh communications**.
Definition:

- **Private Endpoints:** These are internal channels used by agents and contexts to communicate within the mesh. They are not accessible by external systems directly. Examples: the broker addresses and topics used by AgentContexts, the `agent.*.control` topics, internal event topics like `orders.fulfilled` (assuming only internal agents publish/subscribe).
- **Public Endpoints:** Interfaces exposed to outside world (could be API endpoints, message topics open to external publishers/subscribers, or UI sockets) that allow external injection or extraction of information into/from the agent mesh. They act as points where external requests enter, and responses exit.

AMCP itself is transport-agnostic, so "endpoint" can mean:

- An HTTP REST or gRPC API served by a component that translates to AMCP events.
- A broker's external listener (for example, Kafka can have an external listener on a different port, or Solace can have a DMZ broker node) through which external publishers send data on specific topics.
- A specific *agent* designated to interface with external world, e.g. an agent listening on a socket or one that queries a database etc., effectively bridging.

4.1 Gateway and Proxy Agents

One recommended pattern is to use **Gateway Agents** for public interaction:

- These agents run inside the mesh (so they benefit from all the internal routing, reliability, etc.), but they are fed by external input or send output externally.
- For example, a *HTTP Gateway Agent* might run an embedded HTTP server (or, more likely, an HTTP server is run alongside that agent in the same context, which forwards requests as events to the agent).
- When an external HTTP request comes in (say a GET for some resource), the gateway agent receives it (perhaps as an event like `http.request.<sessionId>` with details). The agent then interacts with other agents via AMCP events to fulfill the request, then sends a response via the HTTP server back to the client.

Alternatively, bridging can happen at the broker level:

- Some brokers (Solace, NATS) support bridging protocols for external clients. E.g., Solace can accept MQTT or WebSocket connections from external and map them to internal topics with restrictions. NATS can have a **Leaf Node** which connects an external NATS server in DMZ to the internal NATS cluster, passing only allowed subjects.

4.2 Network Segmentation

In deployment, one might segment:

- **Internal Broker**: Listens only on an internal network interface (cluster network). All AgentContexts connect here.
- **Public Broker interface**: If needed, a separate listener (with separate auth) for external. For Kafka, one can configure advertised listeners such that internal clients use a plain or different host, external use a proxy or separate port with SASL required.

Often it's safer to not expose the main broker at all, but use a proxy service. For instance:

- Deploy a **REST API service** (or use API Gateway like Apigee / Azure API Mgmt) that receives an HTTP call, then produces a Kafka message or calls a context API to create an agent or publish an event.
- The external client never directly connects to Kafka or Solace; it goes through the gateway which you secure heavily.

In our spec, we can highlight a couple of strategies:

- **Direct Broker Access with ACL:** e.g., IoT scenario – many devices connect via MQTT to a Solace broker in the cloud. They authenticate individually and each has permission to a subset of topics (like only their device data topic). The broker thereby becomes multi-tenant: devices talk into the event mesh. Internal agents subscribe to those topics to process device data. This is a valid approach but requires robust ACL setup and possibly a separate broker tier to isolate high loads or untrusted traffic from core traffic (Solace can do bridging between brokers with DMR, so maybe an external broker cluster links to internal).
- **Dedicated Gateway Microservices:** e.g., a Public REST API in front. This microservice likely uses the AMCP client (e.g. acts as an AgentContext too or at least as a broker publisher). It might quickly create a transient agent to handle each request or route through a persistent gateway agent.

4.3 Managing Endpoints in Kubernetes

In Kubernetes deployment (Section 8), typical approach:

- For public endpoints: use a Kubernetes **Ingress or LoadBalancer Service** to expose an HTTP endpoint to outside, linked to a gateway pod (which might run the gateway agent or service).
- For internal-only services: ensure their Services are ClusterIP only (no external exposure) or enforce network policies / service mesh to disallow external access.

Private agent communication is inside the cluster (or through VPN for multi-datacenter). **Public** hits an ingress which terminates TLS and goes to the appropriate service.

4.4 Use of Namespaces and Virtual Networks

In cloud, maybe one uses separate VPC/VNet for external vs internal components:

- The broker might have two network interfaces, one in DMZ subnet and one in internal subnet, and restrict what topics on DMZ side.

Encapsulation of Public traffic: Possibly tag topics as external if needed. For instance, if external devices publish to topics `external.sensor.<id>`, internal agents could then pick up `external.sensor.*` but internal publish on `internal.*` topics. This naming or separate topic space can help enforce policies (like perhaps block any attempt of external publishing to an `internal.` topic by an ACL rule).

4.5 Lifecycle of Public Requests

To tie it together, consider a sample workflow from outside:

1. A user calls a REST API `POST /recommendation?item=123`.
2. The API Gateway authenticates the user (e.g. JWT token).
3. The request is forwarded to a *RecommendationService Agent* via an internal event: The gateway might publish an event `user.request.recommendation` with payload containing item=123 and user info.
4. The RecommendationServiceAgent (running in cluster) receives this event (it subscribed to `user.request.recommendation`). It then interacts with other internal agents (e.g., a ProfileAgent that provides user profile, an InventoryAgent for item availability) by publishing and awaiting events internally.
5. Once it gathers a recommendation result, it publishes a `user.response.recommendation.<reqId>` event.
6. The Gateway, which might have subscribed to response events tagged by reqId, catches it and transforms that to an HTTP response to the user, then closes the connection.

In that sequence, the user never directly touched the internal agents except through a narrow event interface. The gateway did validation (ensured the user can request that, rate limiting perhaps) and the internal mesh did the heavy lifting.

Ensuring no leak from private to public unintentionally: For example, making sure internal topics with sensitive data (like `internal.dbpassword.update`) are not accessible from outside by topic naming and ACLs.
Therefore, **public endpoint management** in AMCP is about creating well-defined, secure entry/exit points, and relying on the mesh internally for everything else.

4.6 Summary

- Public endpoints should be *minimal and hardened*, often implemented by dedicated gateway components.
- Private endpoints constitute the rich mesh connectivity which should remain shielded.
- The design should avoid directly exposing the internal broker broadly – either restrict via network or credentials or both.
- This scheme allows the system to handle external requests without sacrificing the decoupling and scalability of the internal event-driven architecture.

With security and endpoint considerations covered, we move to integration specifics with different broker technologies.

---

5. Integration with Event Brokers (Solace, Kafka, NATS)

AMCP is designed to be **broker-agnostic**, but practical deployment uses an event broker technology. We detail how to realize AMCP's features on three representative platforms:

- **Solace PubSub+** (an enterprise message broker supporting topics, queuing, DMR bridging, etc.)
- **Apache Kafka** (a scalable distributed log system with topic partitions, consumer groups)
- **NATS** (a lightweight cloud-native messaging system with subject-based Pub/Sub and optional JetStream persistence)

Each of these has different strengths and quirks. We cover:

- **Topic Mapping and Subscription Model** for each.
- **Delivery Semantics** (and how to achieve at-least-once or exactly-once if needed).
- **Control Channel Implementation** for agent lifecycle commands.
- **Scaling and Partitioning** considerations.

We present a comparative summary first, then specific integration notes for each.
<style>
:root {
--accent: #464feb;
--timeline-ln: linear-gradient(to bottom, transparent 0%, #b0beff 15%, #b0beff 85%, transparent 100%);
--timeline-border: #ffffff;
--bg-card: #f5f7fa;
--bg-hover: #ebefff;
--text-title: #424242;

```
--text-accent: var(--accent);
--text-sub: #424242;
--radius: 12px;
--border: #e0e0e0;
--shadow: 0 2px 10px rgba(0, 0, 0, 0.06);
--hover-shadow: 0 4px 14px rgba(39, 16, 16, 0.1);
--font: "Segoe Sans", "Segoe UI", "Segoe UI Web (West European)", -apple-system,
"system-ui", Roboto, "Helvetica Neue", sans-serif;
--overflow-wrap: break-word;
}

@media (prefers-color-scheme: dark) {
:root {
--accent: #7385ff;
--timeline-ln: linear-gradient(to bottom, transparent 0%, transparent 3%, #6264a7 30%,
#6264a7 50%, transparent 97%, transparent 100%);
--timeline-border: #424242;
--bg-card: #1a1a1a;
--bg-hover: #2a2a2a;
--text-title: #ffffff;
--text-sub: #ffffff;
--shadow: 0 2px 10px rgba(0, 0, 0, 0.3);
--hover-shadow: 0 4px 14px rgba(0, 0, 0, 0.5);
--border: #3d3d3d;
}
}

@media (prefers-contrast: more),
(forced-colors: active) {
:root {
--accent: ActiveText;
--timeline-ln: ActiveText;
--timeline-border: Canvas;
--bg-card: Canvas;
--bg-hover: Canvas;
--text-title: CanvasText;
--text-sub: CanvasText;
--shadow: 0 2px 10px Canvas;
--hover-shadow: 0 4px 14px Canvas;
--border: ButtonBorder;
}
}

.insights-container {
display: grid;
grid-template-columns: repeat(2,minmax(240px,1fr));
padding: 0px 16px 0px 16px;
gap: 16px;
margin: 0 0;
font-family: var(--font);
```

```
}

.insight-card:last-child:nth-child(odd){
grid-column: 1 / -1;
}

.insight-card {
background-color: var(--bg-card);
border-radius: var(--radius);
border: 1px solid var(--border);
box-shadow: var(--shadow);
min-width: 220px;
padding: 16px 20px 16px 20px;
}

.insight-card:hover {
background-color: var(--bg-hover);
}

.insight-card h4 {
margin: 0px 0px 8px 0px;
font-size: 1.1rem;
color: var(--text-accent);
font-weight: 600;
display: flex;
align-items: center;
gap: 8px;
}

.insight-card .icon {
display: inline-flex;
align-items: center;
justify-content: center;
width: 20px;
height: 20px;
font-size: 1.1rem;
color: var(--text-accent);
}

.insight-card p {
font-size: 0.92rem;
color: var(--text-sub);
line-height: 1.5;
margin: 0px;
overflow-wrap: var(--overflow-wrap);
}

.insight-card p b, .insight-card p strong {
font-weight: 600;
}
```

```
.metrics-container {
display:grid;
grid-template-columns:repeat(2,minmax(210px,1fr));
font-family: var(--font);
padding: 0px 16px 0px 16px;
gap: 16px;
}

.metric-card:last-child:nth-child(odd){
grid-column:1 / -1;
}

.metric-card {
flex: 1 1 210px;
padding: 16px;
background-color: var(--bg-card);
border-radius: var(--radius);
border: 1px solid var(--border);
text-align: center;
display: flex;
flex-direction: column;
gap: 8px;
}

.metric-card:hover {
background-color: var(--bg-hover);
}

.metric-card h4 {
margin: 0px;
font-size: 1rem;
color: var(--text-title);
font-weight: 600;
}

.metric-card .metric-card-value {
margin: 0px;
font-size: 1.4rem;
font-weight: 600;
color: var(--text-accent);
}

.metric-card p {
font-size: 0.85rem;
color: var(--text-sub);
line-height: 1.45;
margin: 0;
overflow-wrap: var(--overflow-wrap);
}
```

```
.timeline-container {
position: relative;
margin: 0 0 0 0;
padding: 0px 16px 0px 56px;
list-style: none;
font-family: var(--font);
font-size: 0.9rem;
color: var(--text-sub);
line-height: 1.4;
}

.timeline-container::before {
content: "";
position: absolute;
top: 0;
left: calc(-40px + 56px);
width: 2px;
height: 100%;
background: var(--timeline-ln);
}

.timeline-container > li {
position: relative;
margin-bottom: 16px;
padding: 16px 20px 16px 20px;
border-radius: var(--radius);
background: var(--bg-card);
border: 1px solid var(--border);
}

.timeline-container > li:last-child {
margin-bottom: 0px;
}

.timeline-container > li:hover {
background-color: var(--bg-hover);
}

.timeline-container > li::before {
content: "";
position: absolute;
top: 18px;
left: -40px;
width: 14px;
height: 14px;
background: var(--accent);
border: var(--timeline-border) 2px solid;
border-radius: 50%;
transform: translateX(-50%);
```

```
box-shadow: 0px 0px 2px 0px #00000012, 0px 4px 8px 0px #00000014;
}

.timeline-container > li h4 {
margin: 0 0 5px;
font-size: 1rem;
font-weight: 600;
color: var(--accent);
}

.timeline-container > li h4 em {
margin: 0 0 5px;
font-size: 1rem;
font-weight: 600;
color: var(--accent);
font-style: normal;
}

.timeline-container > li * {
margin: 0;
font-size: 0.9rem;
color: var(--text-sub);
line-height: 1.4;
}

.timeline-container > li * b, .timeline-container > li * strong {
font-weight: 600;
}
@media (max-width:600px){
.metrics-container,
.insights-container{
grid-template-columns:1fr;
}
}
</style>
<div class="insights-container">
<div class="insight-card">
<h4>Solace PubSub+ Integration</h4>
<p>Use hierarchical topics (e.g., <code>agent/123/control</code>,
<code>orders/created</code>) and wildcards for flexible subscriptions. Leverage Solace’s
<strong>Direct Messaging</strong> for high-speed, and <strong>Guaranteed
Messaging</strong> for critical control events (ensuring delivery with broker ack). Dynamic
Message Routing (DMR) automatically distributes subscriptions across a Solace mesh,
making multi-datacenter agent meshes possible without extra effort. Use client usernames
and VPNs to segregate public vs private topics.</p>
</div>
<div class="insight-card">
<h4>Apache Kafka Integration</h4>
<p>Define Kafka topics for each logical subject (e.g., <code>orders.created</code> as a
topic name). Lacking wildcards, pre-create needed topics or use a naming convention.
```

Partitioning: use a key (like AgentID) for control topics so that all events for one agent go to the same partition (ensuring order). Use consumer groups: each AgentContext runs a consumer group member that filters events for its agents. Durable log ensures at-least-once delivery; implement deduplication in context if needed. Use Kafka ACLs to restrict topic access, and separate clusters or MirrorMaker for bridging public input if necessary.

<h4>NATS Integration</h4>
Use NATS subjects with hierarchy (e.g., <code>agent.123.control</code>, <code>orders.created</code>). NATS supports wildcards (<code>*</code> and <code>></code>) for flexible subscriptions. By default, NATS is fire-and-forget (at-most-once delivery), but enable JetStream for persistence on crucial channels (like control subjects) to get replay/at-least-once. NATS's low latency suits quick agent interactions. Use Queue Groups for load balancing if multiple instances of a subscriber service exist (though typically each agent is a unique subscriber). Keep NATS clusters internal; use NATS Leaf Nodes or WebSockets for bridging external connections, with credentials limiting accessible subjects.

Show more lines
The above insight cards capture the essence for each integration. Now we elaborate on each:

5.1 Solace PubSub+ Details

Topic Hierarchy: Solace topics are a sequence of levels separated by `/` (or any delimiter, often `/`). We can design topics like:

- Agent control: `agent/{agentId}/control` (or use dot notation `agent.{id}.control` – Solace treats both similarly as hierarchy).
- General events: e.g. `orders/created`, `orders/processed`, `alerts/critical`.
- Context control: `context/{contextName}/control` for migration and admin signals.

Solace supports wildcards: `*` for one level and `>` for multi-level trailing. For example, a context can subscribe to `agent/>` to get all agent control messages (then filter by which agents it hosts). But more efficiently, the context could subscribe specifically to `agent/{id}/control` for each agent it has.

Direct vs Guaranteed Messaging:

- *Direct Messaging* in Solace is faster, non-persistent (at-most-once delivery). Use this for high-volume, non-critical events between agents where a lost message is acceptable (or where higher-level retry logic exists).
- *Guaranteed Messaging* uses message spooling and acknowledgment to ensure delivery (at-least-once, potentially with ordering guarantees if using a single flow). Use Guaranteed for:
    - Control commands (so that, for example, a migration command is not lost even if a context was briefly disconnected).
    - Important updates that must not be missed (maybe critical alerts).

We might configure two Solace endpoints: one durable queue or subscription for control events per context, and direct topics for normal events.
Dynamic Message Routing (DMR): Solace's DMR can connect multiple brokers into an **event mesh** where subscriptions propagate. If the AMCP spans multiple Solace brokers (for scalability or geo-distribution), when a context on Broker A subscribes to `agent/XYZ/control`, DMR will propagate that subscription to Broker B if an agent publishes that topic there, ensuring the message routes to Broker A. This fits AMCP's needs to scale across regions seamlessly.
Scaling and Flow Control: Solace can handle large numbers of topics and subscribers. It also has features like **slow subscriber handling** – if an agent (or context) is too slow to consume Guaranteed messages, Solace can drop the subscription or take action; the context design should aim not to block too long on `onBrokerMessage`. Back-pressure can also be managed: e.g., using Solace's queue endpoints to buffer if an agent can't keep up, but then that breaks pub/sub semantics slightly (the queue would load-balance or hold messages).
Security in Solace: Use separate Message VPNs for different environments or trust zones. For example:

- Internal communications in one VPN (only contexts connect).
- If external devices connect via MQTT, maybe use a separate VPN and link with DMR with restricted subscriptions (only forward needed topics across VPNs).
- Client usernames: each context could have a unique username login; ensure the username's profile only allows it to subscribe/publish to certain patterns (like it cannot subscribe to another team's confidential topics if multi-tenant).
- Possibly utilize OAuth integration where each context fetches a token from an auth service and uses it to connect.

Example Mapping:

- The AgentContext's `BrokerClient` for Solace could be implemented with Solace's Java or C API. On `subscribeAgent`, call `session.subscribe(topic)`.
- On migration, unsubscribing from topics will propagate and stop routing to old context.

Note on Exactly Once: If the application requires exactly-once, Solace can use Guaranteed messaging + eliminating duplicates (with DMQ or duplicate detection on the consumer, though Solace itself doesn't automatically do end-to-end exactly once, it does at-least-once from broker's perspective). In practice, the agent or context would need to detect and ignore duplicates via MessageID.

5.2 Apache Kafka Details

Topic Design: Kafka topics are not hierarchical out of the box (no wildcard subscribe). Each distinct event type or message category should correspond to a Kafka topic:

- We might directly use strings with dots in them as topic names (Kafka allows dots and it's common for naming).
- e.g., `agent.control` could be one topic where all agent control events go, or better, one topic per agent for direct messaging would be too many topics (Kafka can handle thousands, but millions is an issue).

    - Instead, likely one **Agent Control Topic** partitioned by agentId. For instance, `agents.control` topic with a partition key = agentId ensures all commands for a particular agent land in one partition (keeping order). Each context's consumer can filter messages for agents it owns by checking the agentId in the message.
- Application topics: If fine-grained, one per event type (like `orders.created`, `orders.updated`). If you want wildcard-like behavior (subscribe to all orders events), the consumer must subscribe to both topics explicitly (Kafka's consumer API allows subscribing to a *pattern* – with modern Kafka, you can subscribe with a regex, e.g. `^orders\.` to match all topics starting with "orders." – this is a feature of client libraries, not broker itself).
- If the number of distinct event types is large or not known in advance, Kafka requires a strategy for topic creation (either create on the fly via admin API when an agent publishes a new type, or use fewer topics and incorporate a sub-type field in the message).

Consumer Group and Message Distribution:

- Each AgentContext could run one Kafka **consumer group** (with a single consumer thread or maybe multiple threads, but likely one per context to sequentialize messages).
- If all contexts share one consumer group for all topics, Kafka would partition messages among them (like load balancing). But we *don't* want arbitrary contexts receiving, we want specifically the context that hosts the target agent to get its messages.
- So, a better approach: Each context has its own **consumer group ID** (like using the context name as group ID). Then each context will get a copy of each message published? Actually, if they use different group IDs, they each independently consume the topics fully – not desirable because then every context would see all events.
- Instead, think differently: Treat each agent or each context as separate consumer logic:
    - For control events: Could have a single topic `agent.control` with replication to all and contexts filter, but that means every context's consumer sees all control events and ignores those not for it – inefficient at scale.
    - Alternatively, one topic per context for control: e.g., `context.node1.control`, `context.node2.control`. Then a migrating command to move agent to node2 goes to that topic. That requires knowing context names, etc.
    - Actually easier: we already said each context has an identity. We can have a Kafka topic for context-specific control: e.g., `context.node1.commands` etc. Kafka topics are cheap enough if number of contexts is not too high (like tens or hundreds).
    - Then each context's consumer subscribes only to its own control topic (and maybe a global broadcast topic if needed for things like "all contexts do X").
    - For agent messaging topics (non-control): If multiple contexts might host subscribers for a topic, Kafka doesn't do selective routing – all consumers in a group share a load, otherwise each group gets all messages. We likely have one consumer group per context per topic if we want splitting, which again

implies duplication of consumption if multiple contexts subscribe to the same event.
    - However, in AMCP, typically each agent subscribes to some topics, and that agent is only in one context. Another agent in another context might subscribe to the same topic, meaning *both should get the event*. To achieve that in Kafka, they must be in different consumer groups (so they each get a copy). If they accidentally share a group, Kafka would give the message to only one of them.
    - So simplest: each agent or each context uses a unique consumer group ID that ensures it independently receives topics. This may lead to large number of consumer groups (at least one per context, possibly one per agent if each had separate connection, but contexts aggregate that).
    - Actually, one consumer group per context is fine: If two contexts both subscribe to `orders.created`, each has its own group, so each gets all messages on `orders.created`. That's exactly what we want for pub/sub semantics (multiple independent subscribers).
    - Performance wise, many consumer groups reading the same topic is okay (just more offset tracking, etc.). The number of contexts (thus groups) might be moderate.
- **Producer Partitioning:** Whenever publishing, ensure a key that groups relevant messages:
    - For events that are replies to a specific request, using the request's key ensures they go to the same partition (not necessarily needed unless ordering among related messages matters).
    - For control events, using agentId or context as key as said.
    - For broader events, if ordering not crucial, perhaps key by something meaningful or leave key null for round-robin.

Kafka as a Durable Log: It retains messages for a configurable time. This is beneficial for replay or for late joining agents to catch up on missed events (if within retention). In an AMCP context, typically agents only process live events, but one could imagine an agent coming up and reading recent events (like to rebuild some state table).
Integration Implementation:

- The `BrokerClient.publish(msg)` in context would map to a KafkaProducer sending to `msg.topic` with `key = msg.senderAgent` or other relevant key (or for control, key = agentId or context).
- The context's `BrokerClient` would have a KafkaConsumer subscribing to topics of interest. But as interest changes (agents subscribe/unsubscribe), how to manage?
    - Kafka Java client does allow dynamic subscription with regex; or one can call consumer.subscribe(List topics) each time the list changes. But rebalancing the consumer group could cause a brief pause.
    - Alternatively, each subscription in AMCP spawns a lightweight dedicated consumer thread for that topic and context (less efficient).
    - More straightforward: Each context at startup subscribes to *all topics that might ever be relevant.* Which is not feasible if unlimited.
    - Perhaps maintain a single consumer and whenever an agent adds a new topic, call consumer.subscribe(updatedTopicList). Kafka will handle rebalancing partitions to the same consumer since it's alone in its group. That is plausible.

    - Implementation detail: Use of `consumer.assign(partitions)` could give even more control but complex.
- **Ordered consumption:** If one context's consumer is handling many topics, events from them will be polled from Kafka in some interleaved order. If an agent needs strict ordering across two topics (rare, and probably not needed since logically separate topics can be processed async), we can consider that not guaranteed. If needed, use one topic instead or do sequencing in agent.

Broker Setup:

- If bridging external, might use Kafka Connect or MirrorMaker to funnel data in/out. E.g., external producers send to a cluster in DMZ, MirrorMaker replicates needed topics to internal Kafka that contexts read from. This adds latency but secures internal cluster.

Kafka Broker Sizing:

- Partition counts: If we have many agents, one might think one partition per agent control or one topic per agent – likely untenable if agents are thousands.
- Instead, one control topic with maybe 10-50 partitions could handle many agents by agentId key hashing into those partitions.
- The ordering per agent is preserved because one agentId always hashes to same partition; two different agents might share a partition but that's fine.
- If an agent moves contexts, it doesn't matter – whichever context now is responsible will still consume from that same partition but with that agent's events.

Kafka Summary: Kafka can support AMCP, but requires careful planning of topics and keys due to lack of true dynamic pub/sub. It shines in reliability and throughput, and the log nature could be exploited for audit (e.g. storing all events for debugging).

5.3 NATS Details

Subject Namespace: NATS uses subjects with dot-separated tokens (similar to topics). E.g., `orders.created` can be a subject. Wildcards:

- `>` matches any number of tokens (must be at end), `*` matches one token.
- So a subscriber can do `orders.*` to get both `orders.created` and `orders.updated`.
- This fits AMCP well (very similar to Solace's model).

Subscription Handling: Each AgentContext would have one NATS connection (which is single TCP and can have many subs).

- When an agent subscribes a subject, context just calls `conn.subscribe(subject, callback)`. NATS server then will send any messages on those subjects to that callback.
- If two agents in same context subscribe to overlapping subjects, the context will get duplicates (one per subscription). But here we can optimize by using one subscription with wildcard and dispatch internally. Considering each agent subscribes individually, it's fine to let NATS handle it; overhead is minor usually.

Delivery: By default, NATS does not store messages – if no subscriber at the moment, message is lost; if subscriber is slow or disconnected, message gets dropped (unless a buffer size is set and overflown).

- For non-critical events between agents, this is okay (especially if pattern is request/response where if a response missed, maybe a timeout triggers a retry).
- For critical events like control or important notifications, we enable **JetStream** (NATS persistence):
    - One can create a JetStream *Stream* that stores certain subjects (like `agent.*.control` or `>` for everything, though storing everything might be heavy).
    - Then define *Consumers* that represent each context or agent. But JetStream's consumption is similar to Kafka's: each consumer is sort of like a stateful subscription that tracks read position. If we want multiple consumers to get the same message, we either make them separate Consumer instances (not durable, ephemeral) or have multiple durable configured per agent which is not feasible to manage manually.
    - Possibly simpler: Use JetStream for the control subjects with a single Consumer per context that pulls its relevant control messages (maybe filter by agentId).
    - Actually, NATS has a feature: subscribe with option to have messages persist (i.e., make a durable consumer automatically). The code can remain largely the same but enabling JetStream and durable name means if the context disconnects, it can resume at last seen message.
    - For non-critical ephemeral topics, continue with core NATS.

NATS Queue Groups: In NATS, if multiple subscribers use the same subject and specify a queue group name, messages on that subject go to only one member of the group (load balancing). We don't want that normally (we want fan-out), so ensure agents use unique queue names (or none, which means they each get a copy).

- However, queue groups could be used intentionally: say you have 3 instances of a "Data processing agent" and you want load-balancing rather than all 3 handling the same event, you could put them in a queue group "dataProc" on subject "sensor.data". Then each event goes to exactly one of the instances (whichever is least busy presumably). This is more microservice-like, but could be a use case (like stateless scaling).

Cluster and Supercluster: NATS can cluster multiple servers for scale and resilience; and superclusters for geo distribution. It doesn't automatically propagate subscriptions like Solace DMR, but in a supercluster, there are gateways for cluster-of-cluster comms that do propagate subscriptions between clusters (similar concept to DMR).

- So, a multi-datacenter NATS deployment can also ensure an event published in region A goes to subscriber in region B who expressed interest, via its routing.

Performance: NATS is known for low latency (sub-millisecond for local cluster). Good for very fast interactions (maybe if agents are doing lots of small talk).

- It is memory-bound for throughput in core mode (since it doesn't hit disk).

- With JetStream, there's disk I/O but still quite efficient for moderate loads.

Security in NATS:

- Use NATS NKeys or JWT. With JWT, an admin can issue a user JWT that includes allowed publish/subscribe subjects.
- For an AgentContext connecting, give it a JWT that allows `agent.<agentId>.control` subscribe for those it hosts? But an agent's ID might not be known at time of connecting. Instead, maybe give the context JWT that allows subscribe on `agent.*.control` but publish on that only if it might have multiple, or allow all if contexts are fully trusted inside cluster.
- If external connecting, restrict their subject permissions strictly.
- NATS can also require TLS. At least server <-> server usually uses TLS; clients too can use TLS with client certs if configured.

Mapping Implementation:

- `BrokerClient.publish` -> `natsConnection.publish(msg.topic, msg.payloadBytes)`. If JetStream: maybe `jsConnection.publish(...)`.
- Subscribe: `natsConnection.subscribe(subject, handlerFn)` returning a subscription handle. Save handle if needed to unsubscribe later.
- On unsubscribe in migration: do `sub.unsubscribe()` or drain.
- If using JetStream durable consumer, one might use `jsConnection.subscribe(subject, { durable: contextName, deliver_policy: All/New/Last })` to ensure if context restarts it gets messages.
- But careful: If context restarts and didn't ack messages, JetStream might redeliver old ones, causing duplicates. The context or agent should check message IDs if that matters.

One subtlety: how to get a guarantee that a migration command is delivered exactly once to dest context? With JetStream, if dest context's consumer is durable and ack-based, the origin can publish MIGRATE_REQUEST as JetStream message and wait for ack. If ack not received, JetStream will redeliver to dest (or to another if rebalanced). But since only dest subscribed to its `context.name.control`, that's fine. So it can get delivered once, and dest sends an ack or response.
NATS Summary: It provides the flexibility akin to Solace for topics, with simpler deployment (just a binary to run for server). For production, likely enable JetStream for critical flows. Monitoring (like NATS server can show number of connections, subs) is needed to ensure resource usage is okay if we have thousands of subs.

5.4 Comparative Table of Key Features

To crystallize differences:

Feature	Solace PubSub+	Apache Kafka	NATS
Topic Model	Hierarchical Topics with wildcards; dynamic subscribe/unsubscribe supported at runtime	Flat Topics (no built-in wildcard; subscribe by explicit list or regex in client)	Hierarchical Subjects with * and > wildcards; dynamic subscribe at runtime

Feature	Solace PubSub+	Apache Kafka	NATS
Delivery Modes	Direct (at-most-once, no ack) and Guaranteed (persistent, acked at-least-once)	Persistent log (messages retained; consumers manual ack or auto commit offset); at-least-once by default, exactly-once with extra layer (e.g. idempotent consumer, EOS transactions for producers)	Core (fire-and-forget, at-most-once), JetStream (persistent, ack for at-least-once). In core mode, no ack needed (server drops after sending).
Ordering	Guaranteed order per topic subscription (Solace preserves order of messages on a topic per consumer)	Ordered per partition. If multiple partitions for a topic, ordering is per partition only. Keying by agent ensures order for that agent.	Ordered per subscriber per subject (NATS preserves in order to each subscriber for each subject published serially). Across subjects, order not guaranteed.
Scaling Consumers	Many consumers can subscribe same topic – each gets a copy (pub/sub). Or use Exclusive Queues for load-balance (1 of N gets msg). DMR allows multi-broker scaling.	If consumers share a group, they load-balance (only one gets each message). If separate group, each gets a copy. Thousands of consumer groups supported (offset tracked for each). Partition scaling for throughput (parallel consumption).	Many consumers can subscribe (each gets copy). Or use Queue Group for load-balance (similar to Solace exclusive queue). Clustering and gateway allows scale-out brokers.
Dynamic Topology	Highly dynamic – create topics just by using them; DMR spreads new subs dynamically. Good for irregular, evolving topic sets.	Requires topic creation (by admin or auto create if enabled). Usually static set of topics for known message types. New topics require cluster metadata update, which is doable programmatically but not as seamless.	Fully dynamic – subjects need no pre-config. Subscribers declare interest on the fly and get those messages immediately. Very flexible for new agent types.
Latency	Low (sub-ms) for Direct in single broker; slight overhead for Guaranteed (needs disk or replication ack). DMR adds a bit across regions.	Typically 2-10 ms end-to-end in many setups (due to commit log and batching). Tunable with linger, batch size. Good throughput but slightly higher latency than in-memory brokers.	Very low for core (often ~1/4 ms in local cluster). With JetStream, latency increases (writing to disk) but still quite fast for moderate loads. Good for real-time interactions.

Feature	Solace PubSub+	Apache Kafka	NATS
Broker Setup	Enterprise-grade, comes as appliance or software. Supports HA clusters and global meshes.	Distributed cluster (brokers coordinate via Zookeeper or KRaft controller). Scales horizontally with partitions. Many cloud-managed options.	Simple single binary for server; can cluster several for HA. Light footprint. Cloud offerings exist.
Security Features	Auth via username/LDAP/OAuth, per-topic-per-VPN ACL, TLS, DMR link encryption. Fine-grained control of client perms.	Auth via SASL (PLAIN, SCRAM, GSSAPI) or mTLS; ACLs on topic (prefix) for read/write. TLS encryption.	Auth via NKEY/JetStream JWT (with subject permissions in JWT); or user/pass. TLS encryption supported.
Special	Built-in replay (for Guaranteed) via queues, but typically once consumed off queue, gone (unless last-value cache feature used). Also has request-reply correlation built in on API level if used.	Stores all messages for retention period, can rewind or join late consumers to past data (replay processing possible). Has Kafka Streams for processing log but outside core.	JetStream persistence gives replay and durable subscriptions if needed. NATS also supports an event streaming layer (Stan, now replaced by JetStream). Typically used for live data distribution.

The above table provides a quick reference for implementers to choose or configure according to needs.

5.5 Ensuring AMCP Behavior on each Broker

It's worth noting a few **pitfalls and solutions** for each integration:

- **Solace**: Ensure to configure the number of Guaranteed flows conservatively (e.g., one session per context). If each agent tried to have its own flow, that doesn't scale. We use context-level flows.
- **Kafka**: Watch out for consumer *rebalancing* delays when contexts join or leave (during that time, consumption pauses). For stable agent mesh (not constantly adding contexts), this is fine. Keep the number of concurrently consumed partitions per context manageable (you may want each context to have at least as many threads as partitions it consumes in parallel).
- **NATS**: Without JetStream, a slow consumer could be problematic (NATS will drop the connection if the client can't read fast enough). So context should read from socket promptly. We may use NATS *auto-unsubscribe* feature if we know we only need certain number of messages or to avoid overload. If needed, JetStream to not lose messages where that matters, at cost of throughput.

---

6. Use Cases Demonstrating AMCP

We present three real-world inspired use cases of AMCP in action. These illustrate different facets of the protocol: agent collaboration, mobility, integration with external systems, and the benefits of the event-driven approach.

6.1 Use Case 1: Smart Factory IoT Coordination

Scenario: A manufacturing plant has dozens of machines (presses, CNC machines, robots) each monitored by an AI agent, plus higher-level supervisor agents coordinating workflow and maintenance. Sensors continuously produce data. The goal is to detect anomalies and dynamically reconfigure the production line by moving "analysis" agents closer to where issues occur.
AMCP Deployment:

- Each machine has a local **MachineAgent** running on an edge device (AgentContext) near the machine. These MachineAgents publish periodic status events (e.g., `machine.press1.vibration` on the mesh).
- A pool of **AnalyzerAgents** subscribe to sensor topics. Normally, one AnalyzerAgent might handle data for multiple machines, but if a particular machine's data shows anomaly, an AnalyzerAgent can be *dispatched* to that machine's local context to run analysis with lower latency (moving the agent closer to data source).
- A **CoordinatorAgent** (in a central context) monitors high-level events like `machine.*.anomaly_detected`. When something is flagged, it issues a control command to migrate an AnalyzerAgent to the machine's context.
- There is also a **MaintenanceSchedulerAgent** that subscribes to prolonged anomaly events and publishes `maintenance.request` events if a machine likely needs service. Another agent listens to `maintenance.request` and integrates with the factory's maintenance ticketing (perhaps via an external API call).

Flow:

1. MachineAgent publishes continuous telemetry: e.g., `machine.press1.status` with payload of sensor readings. AnalyzerAgents subscribed to `machine.press1.#` (or a general wildcard) receive it.
2. An AnalyzerAgent detects a trend indicating a possible failure (e.g., vibration above threshold). It publishes `machine.press1.anomaly_detected` event.
3. The CoordinatorAgent (subscribed to `machine.*.anomaly_detected`) gets this. It picks an idle AnalyzerAgent (or spawns a new one) and calls `dispatchAgent` to move that agent into the context that hosts `MachineAgent` for press1.
4. The migration happens via the broker control topic. Now the AnalyzerAgent is running at the edge, where it can subscribe specifically to high-frequency topics from that machine that were not sent to the central broker due to bandwidth (perhaps MachineAgent only locally publishes high-rate data). The AnalyzerAgent can directly get data via the local context after migration, or perhaps connect to a local data stream.
5. The AnalyzerAgent performs detailed analysis (maybe using a heavy ML model) and decides this is a critical issue. It publishes `alerts.critical` event featuring machine press1 and recommended action.
6. A **DashboardAgent** subscribed to `alerts.critical` notifies human operators. Meanwhile, MaintenanceSchedulerAgent picks it up and schedules maintenance by

calling an external system (through a bridging agent which sends an HTTP POST outside).
7. After the crisis, the CoordinatorAgent may migrate the AnalyzerAgent back to a central server (or keep it there if continuous monitoring needed).

Demonstrated AMCP Features:

- **Publish/Subscribe Decoupling:** MachineAgents didn't know which AnalyzerAgent would process data. They just kept publishing. When the situation changed, an agent moved to follow the data.
- **Agent Mobility:** AnalyzerAgent migrated to the machine's node to reduce latency (maybe the edge node had a special GPU for analysis too).
- **Dynamic Scalability:** New AnalyzerAgents can be created or moved on the fly when multiple machines exhibit issues at once. The event mesh naturally balances since each agent filters by machine ID events.
- **Integration:** The maintenance request shows integration: an external system was notified by an agent. If that external system could also feed back into AMCP (e.g., to confirm maintenance scheduled), it could publish into a public endpoint which a MaintenanceSchedulerAgent picks up.
- **Security Consideration:** Edge AgentContexts likely only allow certain types of agents (like analyzers authorized by central Coordinator). The protocol ensures that the Coordinator's migration command is authenticated and the destination context accepts it because it came on a secure control topic from an authorized source.

Outcome: The factory benefits from real-time reconfiguration of its analytics effort. Issues are caught faster and handled, downtime is minimized. The loosely-coupled agents and event mesh made it easy to plug in new machine types or algorithms without rewriting central logic.

6.2 Use Case 2: E-commerce Order Processing with Multi-Agent Workflow

Scenario: An online retailer employs multiple specialized agents to fulfill orders: InventoryAgent (checks stock), PaymentAgent (processes payments), ShippingAgent (arranges shipping), RecommendationAgent (suggest addons). When a customer places an order, these agents must coordinate. The company wants to be able to add or update these capabilities independently and handle surges gracefully.
AMCP Setup:

- The front-end web uses a **WebAPI Gateway Agent** (as discussed in Section 4) which receives new order events from the web (e.g., when an order is placed, it publishes `order.created` with order details).
- **OrderOrchestratorAgent:** Subscribed to `order.created`, this agent's job is to coordinate the others for each order. It could create a child agent to track each order or just handle via correlation IDs.
- The Orchestrator publishes requests:
    - `inventory.check` (with order items) – InventoryAgent subscribed to `inventory.check` processes it and replies on `inventory.result` topic.
    - `payment.process` – PaymentAgent similarly handles and replies on `payment.result`.

    - It might publish these in parallel or sequence, and use correlation IDs or include order ID in the event payload.
- Each specialized agent (InventoryAgent, PaymentAgent, etc.) can be scaled horizontally: if throughput grows, multiple instances can subscribe to the same request topic using a queue group such that only one handles each request. For example, two PaymentAgents in a queue group "payments" both subscribe to `payment.process` – each order payment will be handled by one of them (load-balanced).
- As responses come in:
    - InventoryAgent publishes either `inventory.result` (with available = true/false for each item).
    - PaymentAgent publishes `payment.result` (success or failure, transaction id).
    - OrchestratorAgent subscribed to these result topics (filtered by correlation or an orderId field in message) collects outcomes.
- If all good, Orchestrator then publishes `shipping.schedule` – ShippingAgent triggers shipping label creation and responds `shipping.confirm` with tracking number.
- Finally, Orchestrator publishes `order.fulfilled` or `order.failed` depending on outcomes, which the WebAPI Gateway might use to notify the user (or the front-end could have been polling, or an asynchronous push notification via another agent).
- Additionally, a RecommendationAgent might subscribe to `order.fulfilled` to update models or send a follow-up suggestion email (through EmailAgent).

Use of AMCP:

- **Parallel Processing:** Because each step is an event, the Orchestrator can issue multiple requests at once and continue when all responses arrive, leveraging the asynchronous nature. It doesn't block a thread; it just waits for messages while possibly starting other order processes.
- **Loose Coupling:** To add a new step (say FraudCheckAgent for payments), one can insert another event without affecting others – Orchestrator can be updated to publish `fraud.check` before `payment.process`. Agents don't directly call each other, so adding an agent means just wiring its subscriptions and publications.
- **Scalability:** During a sale, many orders come in. By deploying multiple InventoryAgents and PaymentAgents, they naturally compete on the request topics. Kafka or NATS will load distribute (via group or single topic multi-consumer approach) so that each request is handled once by one instance. The OrchestratorAgent might also be multiple (each handling different orders) – or one OrchestratorAgent that spawns a lightweight child agent per order to distribute load. AMCP allows either pattern (child agent per order is feasible: Orchestrator could call `createAgent(OrderHandlerAgent)` for each new order event, passing order details, and that agent then coordinates and terminates after done; this is more dynamic, or it could manage state internally).
- **Error Handling:** If PaymentAgent returns failure, Orchestrator can publish a compensating event like `inventory.release_hold` to release stock, and `order.failed` to not trigger shipping. Each agent only cares about their piece, Orchestrator makes the decisions via events.

Integration with External Services:

- PaymentAgent likely calls an external payment gateway (like Stripe API). This agent would be implemented to handle that blocking call, perhaps and then publish result.
- ShippingAgent might call courier API (like UPS, FedEx).
- These external calls happen within those agents, keeping external integration code isolated from other parts. The rest of agents just see events, not actual HTTP details.
- If those external systems sent webhooks (like payment gateway confirming asynchronously), an incoming webhook can be turned into an event e.g. `payment.webhook` which PaymentAgent can subscribe to (to catch late confirmations). Alternatively, PaymentAgent might poll or do synchronous processing.

Security and Endpoint:

- The WebAPI Gateway ensures only legitimate order events are introduced (checks user auth, etc).
- Internally, only authorized agents can publish e.g. `payment.process` (the Orchestrator only). If a rogue agent tried to do so, security manager could prevent it unless it's an agent allowed to orchestrate.
- Sensitive info like credit card details might be encrypted by PaymentAgent or only handled within PaymentAgent (the actual card number wouldn't be broadcast over the event mesh in plaintext; instead maybe a token or reference).
- Each agent runs with minimal privileges (InventoryAgent might have permission to only topics relevant to inventory and perhaps a DB connection for stock levels).

Result: The retailer achieves a flexible microservices-like architecture, but with the benefits of event-driven decoupling:

- Easy to add new features (like sending a promotional coupon event after purchase).
- System naturally handles spikes by scaling instances and relying on the broker to queue requests if needed (Kafka would buffer in topic; Solace could use a queue endpoint if we want to queue beyond memory).
- Failure isolation: If ShippingAgent goes down, orders will proceed up to that point and possibly get marked delayed, but it doesn't crash the whole system. New ShippingAgents can pick up when available, even replay shipping requests from log in Kafka scenario.

AMCP essentially acts as the "spinal cord" of this distributed brain of e-commerce operations, with each agent specialized but working in concert.

6.3 Use Case 3: Multi-Agent Customer Service Chatbot

Scenario: A customer service system uses multiple AI agents to handle user inquiries in a chat. Instead of one monolithic bot, they have:

- **ConversationOrchestratorAgent** that manages the dialogue flow.
- Specialized agents: **FAQAgent** (answers common questions), **OrderLookupAgent** (retrieves customer's orders), **EscalationAgent** (decides if human handoff needed), etc.
- The conversation flows with possibly multiple turns, and different agents may chime in or be consulted behind the scenes for info.

How AMCP fits:

- The user's chat interface (web or mobile) connects to a **ChatGateway Agent** (similar to the web API case, perhaps using WebSocket). This Gateway publishes each user message as an event like `chat.message.in.<sessionId>` with content and metadata.
- ConversationOrchestratorAgent subscribes to all incoming messages (`chat.message.in.*`). When it sees a new message for session XYZ, it either:
  - If no agent for that session yet, it creates one: e.g., spawns a **SessionAgent sessionXYZ** dedicated to that conversation (so that it can maintain context like what the user has asked so far).
  - Or if already exists, it forwards the message to that session's agent by publishing `chat.session.XYZ` with content (or calls the SessionAgent's specific control, but simpler to pub/sub is fine).
- SessionAgent (per conversation) is a composite: it holds partial knowledge and will query other agents:
  - If user asked "Where is my order 12345?", SessionAgent might publish `order.lookup.request` with order number and userID.
  - OrderLookupAgent subscribed to that returns `order.lookup.result` with status shipped/delivered etc.
  - SessionAgent receives that and formulates a reply: maybe "Your order 12345 was delivered on Jan 5."
  - It publishes a `chat.message.out.XYZ` event with the reply text.
- ChatGatewayAgent subscribes to `chat.message.out.*` and when it sees one for session XYZ, it sends it in real-time to the user's interface (over WebSocket or server-sent event).
- For a different query, SessionAgent could consult FAQAgent:
  - e.g., user asks "What is your return policy?" SessionAgent might either know it's a FAQ type and directly ask FAQAgent by publishing `faq.query` or perhaps it just broadcasts the user question on some topic where multiple agents attempt to answer:
  - Possibly use a pattern where each specialized agent tries to answer: they subscribe to `chat.question` and check if they can handle it. But that could get complicated (multiple responses).
  - Better is orchestrator logic: SessionAgent decides which agent to ask based on classification: e.g., known keywords or a classifier. For fallback, if uncertain, ask FAQAgent first, if it has an answer, use it; else ask another or escalate to human.
- EscalationAgent monitors sentiment or repeated failure events (like session has had multiple "I don't know" outcomes). It could subscribe to `chat.session.*` events to analyze conversation flows in parallel, and if triggers met, publish `chat.escalate.XYZ`. SessionAgent sees escalate and then notifies ChatGateway to hand off to a human, or invites a human agent into that session's topic.

Highlighted Features:

- **Agent Creation per Session:** This use case shows dynamic agent lifecycle – each user session can cause a new agent to be spawned to hold context. When session ends, that agent is destroyed. This isolates state per conversation easily, and we can even allocate heavy resources (like an LLM context) to each only for duration needed.

- **Parallel internal queries:** The SessionAgent can query multiple sources concurrently via events – e.g., ask both FAQAgent and OrderLookupAgent, whichever responds first with high confidence it chooses, or merges info.
- **Loose coupling and upgradability:** We can improve any piece: if later we add a **ProductRecommenderAgent** to suggest related products in a conversation, we can do so by having SessionAgent (or a higher orchestrator) publish a `recommendation.request` event at appropriate times. The new agent handles that, no other existing agent code needs to change except the orchestrator logic to call it. Even that orchestrator could be updated on the fly (if we trust replacing the code or as a new version, etc.), with minimal disturbance because interactions are via events.
- **Scalability:** Many chat sessions at once? Each yields an agent, which can be distributed across multiple contexts (maybe one context per region or per some grouping hosts certain session agents). If one context is full (too many session agents), new ones can be created in another context (the orchestrator would need to target that context in createAgent, or pick context by load info).
- With Kafka or NATS broker, topics based on session ID ensure only those interested see the traffic. The broker easily handles thousands of small topics for each chat (especially since they last minutes not permanently – though topics are reused if use a prefix like chat.message.in and sessionId as a token).
- **Failure Recovery:** If a context running session agents crashes, those session agents are gone. The orchestrator (top-level ConversationOrchestratorAgent) could detect that (broker will send maybe a notice if that context's connection dropped, or no response in conversation triggers them to create a new session agent and possibly apologize to user). Alternatively, use a different strategy: run session agents in a robust cluster or persist some conversation state in external store so they can be resurrected. At least, the system will fail softly (one conversation might be interrupted rather than the whole chatbot service going down).
- **Observability:** Because all interactions are events, we can log them or even replay a conversation from event logs for debugging. If one agent gives a poor answer, we can see exactly which agent and input led to it by checking `chat.*` events and intermediate queries.

Security & Privacy:

- User data in chats is sensitive. The events `chat.message.in.XYZ` should probably be confined to that session's context or at least not broadcast widely. The design above already scopes by sessionId, so only the orchestrator and session agent get it. But other agents (like FAQAgent) maybe don't need full user identity, just the question text. SessionAgent can pass minimal info.
- If multiple users are handled, ensure an agent for one session doesn't access another's topics. This is naturally enforced if sessions have unique IDs in topics and no agent subscribes to a wildcard covering others except orchestrator, which is a trusted component. This is a form of multitenancy isolation at the topic level.

Benefit: Instead of one giant chatbot that is hard to maintain, they have modular agents, which can be maintained by different teams (one team improves FAQ knowledge base agent, another the order system integration, etc.). Using AMCP as the integration bus means they all work together seamlessly.

These three use cases demonstrate how AMCP can be applied across domains (IoT/edge, enterprise microservices, AI assistants) to handle dynamic, complex workflows with decoupled, mobile, and scalable agents. Each scenario took advantage of different features:

- Use case 1 emphasized mobility and edge coordination.
- Use case 2 emphasized orchestrating transactional workflows and scaling functional micros-ervices.
- Use case 3 emphasized dynamic agent spawning and parallel knowledge integration.

In all cases, AMCP provided a resilient, flexible backbone for communication.

---

7. Unit Test Examples (Rust, Java, Python)

To solidify understanding and provide guidance for implementation verification, here we present example **unit tests** for core AMCP functionalities in three languages: Rust, Java, and Python. These tests are written in a style suitable for each language's testing framework and illustrate expected behaviors:

- Agent message send/receive.
- Agent migration across contexts.
- Security enforcement (e.g., unauthorized subscribe).
- Integration logic (like an orchestrator gets correct responses).

We'll focus on one representative test scenario implemented in each language to show how an AI code generation tool could produce analogous code.

7.1 Rust Unit Test Example

We assume a Rust implementation where `AgentContext`, `Agent`, etc., are provided (perhaps via an `amcp` crate). We test a simple ping-pong between two agents on the same context to ensure messaging works and ordering is preserved:
Rust
#[cfg(test)]
mod tests {
use super::*;
use amcp::AgentContext;
use std::sync::{Arc, Mutex};

struct EchoAgent {
// The agent might hold some state, e.g., a prefix string
prefix: String,
}
impl Agent for EchoAgent {
fn on_message(&mut self, msg: Message, ctx: &mut AgentContext) {
if msg.topic == "test.ping" {
// Reply with "prefix: <payload>"
let reply_payload = format!("{}: {}", self.prefix, msg.payload_as_str().unwrap());
ctx.publish(Message::new("test.pong", reply_payload));

```
}
}
}

#[test]
fn test_ping_pong_message_flow() {
let mut context = AgentContext::new("TestContext");
// Create an EchoAgent with prefix "Echo"
let agent = context.create_agent::<EchoAgent>("EchoAgentType", None,
Some(json!({"prefix": "Echo"}))).unwrap();
// Subscribe the agent to "test.ping"
context.subscribe_agent(&agent, "test.ping");

// Shared variable to capture response for assertion
let received_pong: Arc<Mutex<Option<String>>> = Arc::new(Mutex::new(None));
// Subscribe a temporary inline handler for "test.pong" to capture the reply
let captured = received_pong.clone();
context.subscribe("test.pong", move |msg| {
let text = msg.payload_as_str().unwrap().to_string();
*captured.lock().unwrap() = Some(text);
});

// Publish a ping message
context.publish(Message::new("test.ping", "Hello"));

// Run the context event loop for a short time to process (assuming async or threaded runtime)
context.flush_events(); // hypothetical function to process pending messages

// Now check that we got the pong
let pong_opt = received_pong.lock().unwrap();
assert!(pong_opt.is_some());
assert_eq!(pong_opt.as_ref().unwrap(), "Echo: Hello");

context.shutdown();
}
}
```

Show more lines

Explanation: We define a simple `EchoAgent` that listens for `test.ping` and responds on `test.pong` with a prefix. The test sets it up, sends a ping, then uses a subscribed closure to record the pong message. We flush events (in a real implementation this might happen in a background thread or we explicitly pump the event loop). Finally, we assert the content. This test checks:

- The agent was successfully created and subscribed.
- A published message was delivered to the agent's on_message.
- The agent's publish (via context) sent out a reply that our test subscriber received.
- The content transformation ("Echo: Hello") is correct, confirming the agent's logic executed.

7.2 Java (JUnit) Unit Test Example

We use a hypothetical Java API similar to our pseudocode. We test an agent migration scenario:

1. Create two contexts.
2. Create an agent on Context1 subscribed to a topic.
3. Dispatch the agent to Context2.
4. Verify that after migration:
    - The agent is no longer in Context1.
    - The agent appears in Context2 (maybe via a callback or by checking context state).
    - The agent can still receive a message on its subscribed topic now that it's on Context2 (ensures subscription moved).

Java

```
import org.junit.Test;
import static org.junit.Assert.*;

public class MigrationTest {
static class CounterAgent extends Agent {
public int counter = 0;
@Override
public void onMessage(Message msg, AgentContext ctx) {
if ("counter.inc".equals(msg.getTopic())) {
counter += (Integer) msg.getPayload();
}
}
}

@Test
public void testAgentMigrationRetainsStateAndSubcription() throws Exception {
AgentContext ctx1 = new AgentContext("Ctx1");
AgentContext ctx2 = new AgentContext("Ctx2");
// Connect contexts to a shared broker, e.g. both using same underlying broker client

// Create agent on ctx1
CounterAgent agent = ctx1.createAgent(CounterAgent.class, "AgentA", null);
ctx1.subscribeAgent(agent, "counter.inc");

// Send a message to increment counter
ctx1.publish(new Message("counter.inc", 5)); // agent should receive and increment
// Give some time or explicitly process events:
ctx1.processOne(); // hypothetical: process one message from queue
assertEquals(5, agent.counter);

// Migrate agent from ctx1 to ctx2
boolean dispatched = ctx1.dispatchAgent(agent.getId(), "Ctx2");
assertTrue(dispatched);
// Simulate that ctx2 receives migration request and processes it:
ctx2.processOne(); // process MIGRATE_REQUEST
// ctx2 should have the agent now
```

```
Agent migratedAgent = ctx2.getAgent(agent.getId());
assertNotNull(migratedAgent);
assertTrue(migratedAgent instanceof CounterAgent);
CounterAgent agentOnCtx2 = (CounterAgent) migratedAgent;
// State should be retained:
assertEquals("Counter state carried over", 5, agentOnCtx2.counter);

// The agent should no longer exist on ctx1
assertNull("Agent removed from old context", ctx1.getAgent(agent.getId()));

// Test that subscription moved: send another increment, should be received in ctx2
ctx2.publish(new Message("counter.inc", 3));
ctx2.processOne(); // deliver to agent on ctx2
assertEquals("CounterAgent should receive increment after migration", 8,
agentOnCtx2.counter);

ctx1.shutdown();
ctx2.shutdown();
}
}
```

Show more lines

Explanation: We define CounterAgent to track a count. The test increments it, then migrates it. We then publish again and verify the count increased further on context2. We explicitly call `processOne()` to simulate the message loop since we don't have actual threads in this test. In a true system, those might be replaced by a small sleep or callback mechanism. We then check the agent's presence and state in both contexts. This test validates:

- Migration moves agent instance and retains its field `counter`.
- The subscription to "counter.inc" moved – since after migration, a publish on ctx2 still reached the agent (if we incorrectly left subscription on ctx1, ctx2 agent might not get message; if we didn't resubscribe on ctx2, no receive).
- Proper cleanup of old context.

It also indirectly tests security if, say, dispatchAgent had security checks (ensuring authorized and context exists—here we assume true returned means allowed).

7.3 Python Unit Test Example

Using `pytest`, we test a scenario with two agents exchanging messages and a SecurityManager blocking an unauthorized action. We assume a Python implementation with classes similar to pseudocode, perhaps using asyncio for event loop.

Python

```
import pytest
from amcp import AgentContext, Agent, Message, SecurityError

# Define two simple agents: one that sends a message on start, and one that receives it.
class GreeterAgent(Agent):
def on_start(self, params):
# on start, send greeting to a topic
self.context.publish(Message(topic="greetings", payload=f"Hello from {self.id}"))
```

```
class ReceiverAgent(Agent):
def __init__(self):
super().__init__()
self.received_msg = None
def on_message(self, msg):
if msg.topic == "greetings":
self.received_msg = msg.payload

def test_publish_subscribe_with_security():
# Set up a context with a restrictive security manager (disallow certain topics)
ctx = AgentContext("TestCtx")
# Suppose SecurityManager can be configured to disallow "forbidden" topic
ctx.security_manager.disallow_subscribe("forbidden.topic")

# Create ReceiverAgent and subscribe to "greetings"
recv_agent = ctx.create_agent(ReceiverAgent, agent_id="Receiver1")
ctx.subscribe_agent(recv_agent, "greetings")
# Create GreeterAgent which will publish on start
greet_agent = ctx.create_agent(GreeterAgent, agent_id="Greeter1")
# The GreeterAgent's on_start should have published a message
# Run the event loop to deliver the message
ctx.poll_once() # hypothetical method to process one event

# Now the ReceiverAgent should have gotten the message
assert recv_agent.received_msg is not None
assert recv_agent.received_msg.startswith("Hello from Greeter1")

# Test security: attempt to subscribe ReceiverAgent to a forbidden topic
with pytest.raises(SecurityError):
ctx.subscribe_agent(recv_agent, "forbidden.topic")
```

Show more lines

Explanation: This test creates a receiver and a greeter. The greeter automatically sends a greeting on startup (testing that on_start hook works and that publish in on_start reaches others). We then poll events so the receiver gets it, and we assert the content. Then we intentionally try an unauthorized action: subscribing to "forbidden.topic" which our SecurityManager is set to disallow. We expect a `SecurityError` exception. This verifies that the security layer is active.

For the test to run, the framework inside AgentContext must:

- Call `on_start` after creating an agent.
- The SecurityManager must throw exception on disallowed subscribe.
- `ctx.poll_once()` processes the published greeting to the subscriber.

In a real async scenario, we might integrate with asyncio loop and use `await ctx.idle()` or similar, but for unit test, synchronous polling or callback hooking is used for determinism.

Summary of Tests:

- Rust: tested event sending and content handling in one context.
- Java: tested agent migration across contexts.

- Python: tested event sending plus a security restriction.

These show multi-language perspective but all follow the specification's logic. The expected outcome in each is derived from the specification (e.g., after migration state is retained; unauthorized subscribe raises error).
They can guide implementers to ensure their code passes such tests and also help AI code generators to see usage patterns (creating context, agents, publishing, etc.) and implement accordingly.

---

8. Deployment Architecture for Kubernetes (On-Prem & Cloud)

Finally, we describe how to deploy an AMCP-based system on Kubernetes, covering both on-premises clusters and cloud environments. We discuss containerization of components, service definitions, configuration via Helm charts, service mesh integration, and observability setup. The goal is to illustrate a reference deployment that ensures scalability, resilience, and security.

8.1 Microservices and Containers

Each **AgentContext** (which may host one or many agents) is packaged as a container image. In many cases, you might create one context per microservice domain:

- For example, one context runs all Inventory-related agents, another runs all Payment agents, etc., or
- One context per agent type, or
- Even one context per agent instance (though that could be overhead if thousands). More practically group by type or role.

Container Images:

- Include the runtime (JVM for Java, or a Rust binary, etc.), the code for the agents, and the common AMCP library.
- Could have one image per distinct agent type or one image with multiple agent classes where which ones start is configurable.

Pods and Scaling:

- Each AgentContext container could be a Deployment (so it can replicate and self-heal).
- But if you replicate an AgentContext (multiple pods running same context code concurrently), they would appear as separate contexts logically in AMCP (with different context names). You wouldn't typically replicate one context name, except if they coordinate via external means. Instead, if you need N instances of a particular agent type, you might run N contexts each hosting an instance (or one context hosting N threads, but in container scaling view, N pods each an AgentContext is easier to scale via K8s).

- Example: PaymentAgentContext Deployment with replicas=3, each will connect to broker with unique context ID maybe "PaymentCtx-1", "PaymentCtx-2" etc. The orchestrator knows or broadcast requests such that any can handle.

Broker Deployment on Kubernetes:

- **Solace**: There is a PubSub+ Helm chart for deploying a broker cluster (requires persistent storage if using guaranteed messaging). Alternatively, use Solace Cloud (external) and connect via IP (less internal control).
- **Kafka**: Use Strimzi operator or Confluent operator to deploy a Kafka cluster. Zookeeper or KRaft nodes, etc. Or use a cloud service like AWS MSK or Confluent Cloud (then no pods for Kafka in cluster, just connectivity).
- **NATS**: Deploy a NATS cluster (there are official Helm charts or just deploy nats-streaming/nats-jetstream as StatefulSet).

On-Prem vs Cloud:

- On-prem: likely you deploy the broker inside your cluster or in your data center as VMs. You have full control and possibly want to manage it via K8s too.
- Cloud: you might offload broker to a managed service (Kafka on Confluent Cloud, or Azure EventHub if it could speak similar protocols, etc.). If using managed broker, your agent pods need network egress to that cloud service and proper credentials.
- Connectivity: On prem cluster might have better local network; cloud might have to ensure VPC connectivity or use public endpoints of broker (with TLS).
- Also on prem you might not have certain cloud monitoring services, but can use open-source ones (Prometheus, etc.), whereas on cloud you can integrate with cloud monitoring (CloudWatch, Azure Monitor).

8.2 Helm Chart Structure

We can create a Helm chart to deploy the whole AMCP application stack:

- **Broker** sub-chart (optional or external).
- **AgentContexts**:
  - Each context could be a release in Helm or one Helm chart with multiple Deployments in templates.
  - For example, chart values have a list of contexts to deploy: their name, number of replicas, environment variables (like broker address, credentials, relevant config).
  - Helm iterates to create Deployment+Service for each context type.

Configuration via ConfigMap/Secrets:

- Broker connection info (hostname, ports, possibly credentials) as a K8s Secret mounted or env vars.
- Security config, e.g., the rules for SecurityManager might be supplied via ConfigMap (so you can tweak allowed topics without rebuilding image).
- Agent initial parameters or certain toggles can also be in ConfigMap.

Service Mesh (Istio/Linkerd):

- To enforce mTLS and fine-grained traffic policies inside cluster, injecting a sidecar proxy in each pod (like Istio Envoy).
- This can encrypt all pod-to-pod traffic. For broker, if broker pods and agent pods all have sidecars, their communication is encrypted and authenticated at mesh level.
- Could also use network policies to restrict which pods can talk to broker pods (only agent pods can).
- If multi-namespace, possibly restrict by namespace too.

Public Access:

- If we have a GatewayAgent as a pod, and we want to expose it externally (e.g., via HTTP), we create a K8s Service of type LoadBalancer or an Ingress for it.
- Ingress Controller (nginx, etc.) can route `api.myservice.com/chat` to the ChatGateway service on cluster.
- If external devices publish to broker, either they connect via a public LB (for NATS or Solace) – then ensure security on that interface. Or more likely, use a dedicated ingress component:
    - e.g. an MQTT Bridge: some IoT use-case might have an MQTT broker in DMZ bridging to NATS inside. That complicates architecture but is doable.

Kubernetes objects layering:
Let's illustrate a minimal architecture with a diagram focusing on components:
<style>
:root {
--accent: #464feb;
--timeline-ln: linear-gradient(to bottom, transparent 0%, #b0beff 15%, #b0beff 85%, transparent 100%);
--timeline-border: #ffffff;
--bg-card: #f5f7fa;
--bg-hover: #ebefff;
--text-title: #424242;
--text-accent: var(--accent);
--text-sub: #424242;
--radius: 12px;
--border: #e0e0e0;
--shadow: 0 2px 10px rgba(0, 0, 0, 0.06);
--hover-shadow: 0 4px 14px rgba(39, 16, 16, 0.1);
--font: "Segoe Sans", "Segoe UI", "Segoe UI Web (West European)", -apple-system, "system-ui", Roboto, "Helvetica Neue", sans-serif;
--overflow-wrap: break-word;
}

@media (prefers-color-scheme: dark) {
:root {
--accent: #7385ff;
--timeline-ln: linear-gradient(to bottom, transparent 0%, transparent 3%, #6264a7 30%, #6264a7 50%, transparent 97%, transparent 100%);
--timeline-border: #424242;
--bg-card: #1a1a1a;
--bg-hover: #2a2a2a;

```
--text-title: #ffffff;
--text-sub: #ffffff;
--shadow: 0 2px 10px rgba(0, 0, 0, 0.3);
--hover-shadow: 0 4px 14px rgba(0, 0, 0, 0.5);
--border: #3d3d3d;
}
}

@media (prefers-contrast: more),
(forced-colors: active) {
:root {
--accent: ActiveText;
--timeline-ln: ActiveText;
--timeline-border: Canvas;
--bg-card: Canvas;
--bg-hover: Canvas;
--text-title: CanvasText;
--text-sub: CanvasText;
--shadow: 0 2px 10px Canvas;
--hover-shadow: 0 4px 14px Canvas;
--border: ButtonBorder;
}
}

.insights-container {
display: grid;
grid-template-columns: repeat(2,minmax(240px,1fr));
padding: 0px 16px 0px 16px;
gap: 16px;
margin: 0 0;
font-family: var(--font);
}

.insight-card:last-child:nth-child(odd){
grid-column: 1 / -1;
}

.insight-card {
background-color: var(--bg-card);
border-radius: var(--radius);
border: 1px solid var(--border);
box-shadow: var(--shadow);
min-width: 220px;
padding: 16px 20px 16px 20px;
}

.insight-card:hover {
background-color: var(--bg-hover);
}
```

```
.insight-card h4 {
margin: 0px 0px 8px 0px;
font-size: 1.1rem;
color: var(--text-accent);
font-weight: 600;
display: flex;
align-items: center;
gap: 8px;
}

.insight-card .icon {
display: inline-flex;
align-items: center;
justify-content: center;
width: 20px;
height: 20px;
font-size: 1.1rem;
color: var(--text-accent);
}

.insight-card p {
font-size: 0.92rem;
color: var(--text-sub);
line-height: 1.5;
margin: 0px;
overflow-wrap: var(--overflow-wrap);
}

.insight-card p b, .insight-card p strong {
font-weight: 600;
}

.metrics-container {
display:grid;
grid-template-columns:repeat(2,minmax(210px,1fr));
font-family: var(--font);
padding: 0px 16px 0px 16px;
gap: 16px;
}

.metric-card:last-child:nth-child(odd){
grid-column:1 / -1;
}

.metric-card {
flex: 1 1 210px;
padding: 16px;
background-color: var(--bg-card);
border-radius: var(--radius);
border: 1px solid var(--border);
```

```
text-align: center;
display: flex;
flex-direction: column;
gap: 8px;
}

.metric-card:hover {
background-color: var(--bg-hover);
}

.metric-card h4 {
margin: 0px;
font-size: 1rem;
color: var(--text-title);
font-weight: 600;
}

.metric-card .metric-card-value {
margin: 0px;
font-size: 1.4rem;
font-weight: 600;
color: var(--text-accent);
}

.metric-card p {
font-size: 0.85rem;
color: var(--text-sub);
line-height: 1.45;
margin: 0;
overflow-wrap: var(--overflow-wrap);
}

.timeline-container {
position: relative;
margin: 0 0 0 0;
padding: 0px 16px 0px 56px;
list-style: none;
font-family: var(--font);
font-size: 0.9rem;
color: var(--text-sub);
line-height: 1.4;
}

.timeline-container::before {
content: "";
position: absolute;
top: 0;
left: calc(-40px + 56px);
width: 2px;
height: 100%;
```

```
background: var(--timeline-ln);
}

.timeline-container > li {
position: relative;
margin-bottom: 16px;
padding: 16px 20px 16px 20px;
border-radius: var(--radius);
background: var(--bg-card);
border: 1px solid var(--border);
}

.timeline-container > li:last-child {
margin-bottom: 0px;
}

.timeline-container > li:hover {
background-color: var(--bg-hover);
}

.timeline-container > li::before {
content: "";
position: absolute;
top: 18px;
left: -40px;
width: 14px;
height: 14px;
background: var(--accent);
border: var(--timeline-border) 2px solid;
border-radius: 50%;
transform: translateX(-50%);
box-shadow: 0px 0px 2px 0px #00000012, 0px 4px 8px 0px #00000014;
}

.timeline-container > li h4 {
margin: 0 0 5px;
font-size: 1rem;
font-weight: 600;
color: var(--accent);
}

.timeline-container > li h4 em {
margin: 0 0 5px;
font-size: 1rem;
font-weight: 600;
color: var(--accent);
font-style: normal;
}

.timeline-container > li * {
```

```
margin: 0;
font-size: 0.9rem;
color: var(--text-sub);
line-height: 1.4;
}

.timeline-container > li * b, .timeline-container > li * strong {
font-weight: 600;
}
@media (max-width:600px){
.metrics-container,
.insights-container{
grid-template-columns:1fr;
}
}
</style>
<ul class="timeline-container">
<li>
<h4>Build Phase: Containerization</h4>
<p>Create Docker images for each AgentContext type (with agents and AMCP runtime). Use a CI pipeline to build and push these images.</p>
</li>
<li>
<h4>Deploy Phase: Broker Setup</h4>
<p>Deploy the event broker. For example, use a Helm chart to install a NATS cluster (3 pods StatefulSet) or a Kafka cluster (with Zookeeper ensemble). Verify broker service is reachable within cluster (e.g., service DNS names for brokers).</p>
</li>
<li>
<h4>Deploy Phase: Agent Contexts</h4>
<p>Deploy AgentContext pods using a Helm chart. Each context runs as a Deployment with desired replicas. On startup, each pod registers with the broker (using configured address/credentials) and announces its presence via events.</p>
</li>
<li>
<h4>Service Mesh & Security</h4>
<p>Istio sidecars are injected into each pod (if using), automatically securing traffic. Network policies restrict broker ports to agent namespace only. Kubernetes Secrets provide broker credentials to pods as env vars. RBAC rules ensure pods cannot mount others' secrets.</p>
</li>
<li>
<h4>Observability Setup</h4>
<p>Deploy Prometheus for metrics and configure it to scrape metrics from broker and agent pods (if they expose /metrics). Deploy EFK (Elasticsearch-Fluentd-Kibana) for log aggregation of agent logs. Use OpenTelemetry SDK in agent code to emit traces of inter-agent calls, collected by Jaeger.</p>
</li>
<li>
<h4>Scale & Manage</h4>
```

Use HorizontalPodAutoscaler for contexts if they represent stateless services that can scale (e.g., more PaymentAgent pods when CPU high). Use liveness probes for pods (an HTTP or TCP check) to auto-restart hung agents. The broker StatefulSet can also be configured with PodDisruptionBudgets for safe rolling updates. Use Helm upgrades to rollout new agent versions or configuration changes.

Show more lines
Explanation of timeline: It's more a sequence of deployment steps than time, but it captures:

- Building container images (pre-deploy).
- Deploying broker.
- Deploying agent contexts.
- Integrating security (sidecars, network policies, secrets).
- Setting up monitoring.
- Operational scaling and update strategies.

This covers both an on-prem scenario (the cluster could be on company servers) and cloud (the cluster could be EKS/AKS/GKE, procedure similar; or using cloud services for broker and monitoring could replace some steps).

8.3 Observability and Logging

An important aspect in deployment is making sure we can monitor and debug:

- **Metrics:** Each AgentContext can expose metrics like number of agents running, number of messages in/out, processing latency, via an HTTP endpoint (using e.g. Prometheus client library). The broker also exposes metrics (Kafka via JMX, NATS has monitoring, Solace via SEMP or SNMP).
- Prometheus in cluster scrapes these metrics and one can set alerts (e.g., if message queue size grows too large, or an agent's error count spikes).
- **Logging:** Agents should log significant events (especially errors) to stdout (Kubernetes will capture it). Fluentd or Filebeat can ship logs to a centralized store. Log fields should include agent ID, context, maybe correlation IDs to tie together events.
- **Tracing:** In a distributed workflow (like the order processing scenario), using tracing can greatly help. If each event carries a TraceID and possibly SpanID, and if AgentContext logs or propagates it to subsequent events, you can reconstruct the flow:
    - For example, Orchestrator receives `order.created` (assigns a TraceID), then publishes `inventory.check` with same TraceID. InventoryAgent logs processing that TraceID, etc. Using OpenTelemetry, we can start a span on receiving an event and end on sending next.
    - A Jaeger deployment can collect traces from instrumentation in code.
- **Kubernetes-specific:**
    - Use `kubectl logs` to inspect an agent if needed.
    - Use `kubectl port-forward` to broker or context metrics endpoints if not exposing them externally.
    - Possibly leverage service mesh observability (like with Istio, you get some telemetry of traffic between services, but here a lot goes through broker rather

than direct service-to-service HTTP, so may not see inside broker interactions).

- **Chaos Testing:** On K8s, you might simulate node failure to ensure agents migrate or recover gracefully. For example, kill a pod running an agent context and see if the orchestrator or system recreates missing agents if needed. This ties back to having some supervision: one approach is to have some persistent store of what agents should exist (like a controller agent), but our current design expects ephemeral dynamic creation. So maybe just accept that if a pod dies, all its agents die unless orchestrator or something notices and recreates. Alternatively, avoid losing important state by using migration proactively or storing critical data externally.

8.4 Differences On-Prem vs Cloud Specifics

- **On-Prem:**
  - You might not have fully managed database for logs or metrics; you'll rely on open-source stack all deployed in cluster (ELK, Prom, etc.).
  - Networking: On-prem connections might not traverse NAT or internet, simpler in some ways; solution must also handle if multiple physical sites (so event mesh bridging).
  - Likely stricter change management - maybe prefer stable static deployments over constantly auto-scaling.
- **Cloud:**
  - Possibly auto-scale more freely with demand (spinning VMs or pods).
  - Use cloud monitoring: e.g., Azure Application Insights for distributed tracing if instrument in code, or CloudWatch metrics for CPU, etc. But those can integrate with open telemetry too.
  - Could use cloud broker services as mentioned, which simplifies broker maintenance but you need secure connectivity (like VPC peering or allow specific IP).
  - For global reach, if deploying across multi-regions, could have multiple K8s clusters and connect their brokers (like Solace DMR across regions).

Deployment Example: Let's consider an example with NATS in Kubernetes:

- Use Helm to deploy a 3-node NATS cluster (with JetStream enabled) in namespace "message-broker".
- Deploy deployments:
  - `inventory-context` (image contains InventoryAgent code, it might also host Orchestrator if appropriate).
  - `payment-context`, `shipping-context`, etc.
  - Each loads configuration (like topics, or registers subscription on start).
- We use Istio so each pod gets a sidecar, and we have a mesh policy requiring mTLS, so NATS cluster and agents communicate over TLS (with Istio sidecar handling encryption on hop, though NATS also supports TLS, we might either disable NATS TLS internally because Istio covers it or double encrypt).
- The Helm chart configures each context with environment: NATS_SERVER = "nats://nats-service.message-broker.svc.cluster.local:4222", NATS_CREDS from a Secret.

- We put all agent pods in namespace "amcp-agents". We apply a NetworkPolicy there: only allow egress to `message-broker` namespace on NATS port 4222 (so pods can't accidentally or maliciously call other external services unless allowed).
- We expose an Ingress route for the API gateway if needed, or maybe skip if not required.
- Use Prometheus Operator to discover metrics from pods in "amcp-agents" (the pods expose e.g. a `/metrics` on port 9100).
- Kibana reads logs that FluentBit collects from those pods.
- Everything runs, and we can manage via Helm upgrades: e.g., to roll out a new version of PaymentAgent, we build a new image tag and update values.yaml for that deployment, Helm upgrade triggers pods rolling update (K8s ensures rolling update with maybe maxUnavailable 1, etc).
- The system stays running 24/7, with the broker statefulset providing continuity for message storage (JetStream) if pods restart.

Disaster Recovery & Cloud:

- If an entire cluster fails (on-prem DC outage), you might rely on multi-cluster event mesh bridging. Or backup restore if using persistent broker.
- Cloud might allow easier multi-AZ or multi-region replication.

In summary, Kubernetes provides the orchestration needed to run AMCP components reliably. Proper use of controllers like Helm, service mesh, and monitoring closes the loop for a production-ready deployment.

Conclusion

This technical specification has outlined AMCP in detail: from formal definitions and proofs of core properties, through pseudocode of mechanisms, a comprehensive security model, endpoint management strategies, integration guidelines for major event brokers, illustrative use cases, down to concrete testing and deployment considerations on Kubernetes.
AMCP's design marries the flexibility of publish/subscribe event-driven systems with the unique requirements of multi-agent collaboration (like agent mobility and contextual isolation), all underpinned by robust security and scalability.
By following this specification, implementers can build an AMCP-compliant system, and AI code generation tools can use the provided details (pseudocode, data structures, test cases) to assist in producing correct and secure implementations in various programming languages.
The end result is an agent mesh platform capable of powering complex, adaptive AI systems in domains ranging from industrial IoT and enterprise workflows to cloud-based AI services, benefiting from the modularity, responsiveness, and resilience that AMCP provides.