An Accountability-Based Architectural Tactic for Agent Cooperation in LLM-Based Multi-Agent Systems

Marco Becattini, Roberto Verdecchia, and Enrico Vicario
Software Technologies Laboratory, University of Florence
Email: {marco.becattini, roberto.verdecchia, enrico.vicario}@unifi.it

***Abstract*—The rapid advancement of Large Language Model (LLM) based agents is enabling sophisticated multi-agent systems for complex task automation. However, current frameworks for orchestrating multiple LLM agents lack formalized mechanisms for defining inter-agent relationships and responsibilities. This limitation hinders, among other properties, scalability, maintainability, and governance in production deployments. We propose the implementation of an accountability-based architectural tactic that adapts the accountability pattern to LLM-based multi-agent systems, introducing the architectural components *Agent Types*, *Agent Relationships*, and a clear separation between *Operational* and *Knowledge* levels. Our architectural tactic is conceived with the threefold goal of enabling (i) declarative specification of agent capabilities and constraints, (ii) dynamic reconfiguration of agents and their relationships without code changes, and (iii) systematic tracking of responsibilities and audit trails. We validate our architectural tactic through a proof-of-concept implementation that extends SALLMA (a Software Architecture for LLM-based Multi-Agent systems) with accountability mechanisms. A comparative case study with LLM-based agents acting as a software development team showcases that our accountability-based architectural tactic reduces configuration burden, enables agent addition without orchestration code changes, and provides explicit governance mechanisms compared to non-accountability baseline orchestration.**

***Index Terms*—Multi-agent systems, Large Language Models, Software architecture, Accountability pattern, Agent orchestration**

I. Introduction

LLM-based multi-agent systems are gaining traction in research and practice, powered by frameworks such as LangChain, AutoGPT, and CrewAI [1]–[3], and increasingly enabled by agentic coding tools (e.g., Anthropic's Claude Code CLI, OpenAI's Agent SDK, and Google's Gemini CLI). However, existing AI-centric architectures face limitations: orchestration frameworks typically require hard-coded pipelines or fixed hierarchies, making them difficult to scale, reconfigure, or audit. Meanwhile, agentic tools often operate as black boxes, providing limited transparency into how tasks are allocated among agents, which agent is responsible for specific outcomes, or how conflicts between agents are resolved.

To address these limitations, we advocate for an accountability-based architectural tactic that enables explicit specification of inter-agent responsibilities, permissions, and constraints. We envision that such a model can support more effective task execution through improved observability and enable runtime reconfiguration of agent teams without modifying orchestration code.

To enable such runtime reconfiguration and observability, our implementation of an architectural tactic [4] relies on a clear separation between the *operational* and the *knowledge* level, inspired by the Reflection architectural pattern [5]. Without this separation, systems accumulate code-level coupling that may hinder observability, reuse, and governance of LLM-based multi-agent systems.

Building upon these concepts, we present an accountability-based multi-agent architectural tactic, which adapts the accountability pattern [6] to LLM-based multi-agent systems by systematically mapping the concepts of *Party* to *Agent*, *Accountability* to *Agent Relationship*, and introducing *Operating Scopes* with measurable duties.

More specifically, this paper:

1) Formalizes accountability for LLM-based multi-agent systems via the concepts of *Agent Type*, *Relationship Types*, and *Operating Scopes*.
2) Presents how accountability mechanisms can be integrated into a two-level architecture through knowledge-level catalogs (Agent Types, Relationship Types, Scope Schemas) that enable declarative, type-checked reconfiguration of agent relationships without code changes.
3) Provides a proof-of-concept implementation of the accountability-based architectural tactic with configuration examples and runtime enforcement mechanisms.

II. Background and Related Work

A. LLM-Based Multi-Agent Systems

Recent surveys [7], [8] provide systematic frameworks for understanding LLM-based agents, identifying essential architectural modules (profile, memory, planning, action) that enable complex reasoning, tool orchestration, and adaptive behavior. Building upon single-agent architectures, LLM-based multi-agent systems (LLM-MAS) have emerged as a potential architecture for complex problem-solving, with frameworks characterizing collaboration along multiple dimensions including structure, coordination protocols, and interaction strategies [9].

Current frameworks, such as LangChain/LangGraph [1], AutoGen [10], and CrewAI [3], provide varied orchestration

capabilities, which currently focus more on operational execution rather than governance. As noted by recent research on responsible LLM-based multi-agent systems [11], [12], existing frameworks provide limited built-in facilities for agent relationships management and accountability tracking, these capabilities being necessary for production deployments, which require auditability and compliance. Moreover, the black-box nature of foundation models complicates attribution of responsibilities when multiple agents interact [13].

B. Accountability and Responsibility in Multi-Agent Systems

Accountability in multi-agent systems encompasses mechanisms for attributing actions, decisions, and outcomes to specific agents, enabling both ex-ante governance and ex-post auditing. Conte and Paolucci analyze how responsibility identifies the locus of accountability within multi-agent organizations, enabling proper distribution of accountabilities among agents [14]. They distinguish between primary responsibility (arising from deliberative capacity) and task-based responsibility (arising from task commitment).

Norman and Reed developed a formal model of delegation showing how agents may be issued imperatives not only to directly perform actions but also to be responsible for outcomes, with these commitments serviced through subsequent delegation [15]. Castelfranchi and Falcone further examined how delegation fundamentally enables coordination and collaborative action within organizations [16], demonstrating how responsibilities propagate through organizational hierarchies.

Formal accountability reasoning has been studied through various frameworks. The Abstract Accountability Language (AAL) [17] provides a logical framework for protocols with formal accountability proofs, while recent computational approaches introduce accountability protocols that regulate interactions between agents and organizations.

However, these approaches so far seem to have found limited application to contemporary LLM-based architectures. In this context, Fowler's accountability pattern [6] is well suited as an architectural template to explore accountability mechanisms in LLM-based multi-agent systems. Its separation of operational instances from knowledge-level types aligns naturally with declarative configuration requirements of LLM agent orchestration. We build upon Fowler's foundation while extending it with LLM-specific concerns including tool governance, operating scope specifications, and dynamic prompt generation.

C. Architectural Layering and the Accountability Pattern

Separation between operational runtime and knowledge-level specifications is a recurring architectural principle [18], [19]. SALLMA [20] recently presented an architecture for LLM-MAS with two distinct Operational and Knowledge Layers.

As we show in this work, such separation enables declarative reconfiguration without code changes. While layered architectures for multi-agent systems have been proposed [21], explicit separation between configuration and execution concerns remains mostly underexplored in LLM-based systems. Most current LLM-MAS implementations conflate these levels, embedding component type constraints directly in orchestration code, potentially hindering scalability and obscuring accountability.

III. Motivating Example

Consider a software development team composed of LLM-based agents building a web application with multiple front-end technologies. The team initially comprises two agents:

- **Architect Agent**: orchestrates development, enforces architectural principles, supervises developers;
- **Angular Front-end Developer Agent**: implements the existing Angular-based user interface.

A new requirement emerges, namely the application must support a React-based dashboard module alongside the existing Angular interface. The team must add a **React Front-end Developer Agent** to handle this specialized task.

In conventional frameworks, relationships (*e.g.* "supervised by Architect") are hard-coded in the orchestration logic. Adding the React developer requires modifying routing code, supervision hierarchies, and constraint validators at multiple points. Accountability and traceability, *e.g.* which agent is responsible for maintaining architectural consistency across React and Angular components, are scattered across logs and code comments. There is no reusable knowledge-level catalog of *what types of agents* may collaborate and *under which rules*.

An accountability-based architectural tactic treats these relationships as *reified artifacts* [22] (i.e., first-class components that can be explicitly manipulated and stored) that are configured, validated, and audited at the knowledge level, while the operational layer instantiates them at runtime. In our example, the Architect Agent holds accountability for overall architectural integrity and supervises both front-end developers. The Angular Developer holds accountability for the existing interface implementation and adherence to architectural guidelines (e.g. component patterns, state management conventions). The React Developer (once added) holds equivalent accountability for the new module, with measurable operating scopes including architectural compliance and code quality thresholds. Each accountability is explicitly defined with relationships and operating scopes (*e.g.* "maintain consistent design system components"), enabling the system to detect violations and trace responsibility to specific agents and relationship instances without code inspection.

IV. The Accountability-Based Multi-Agent Architectural Tactic

A. From the Accountability Pattern to LLM-Based Multi-Agent Systems

Our architectural tactic is based on the accountability pattern [6], originally designed for modeling organizational responsibilities in enterprise systems. The pattern (Figure 1) distinguishes two levels: the *operational level*, which records day-to-day events and consists of **Party** and **Accountability**

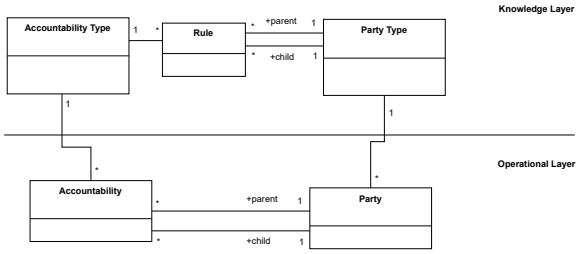

Fig. 1. Fragment of the accountability pattern (based on Fowler [6]), with explicit subdivision between knowledge and operational layers.

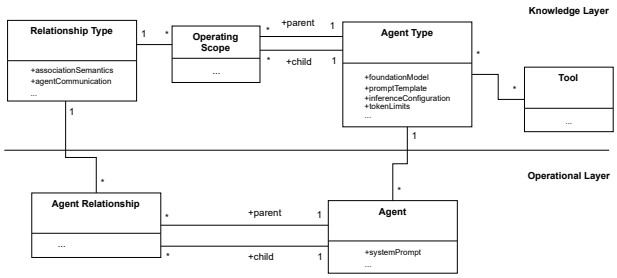

Fig. 2. Our accountability-based architectural tactic for LLM-based multi-agent systems, extending the accountability pattern.

instances and their interrelationships; and the *knowledge level*, which records general rules governing the structure through **Party Type** and **Accountability Type**. An accountability represents a responsibility relationship between two parties (a commissioner and a responsible party), governed by an accountability type. Operating scopes attach measurable constraints and duties to accountabilities, enabling contractual specifications of responsibilities.

We adapt and build upon this pattern to extend its concepts to LLM-based multi-agent systems (Figure 2) through the following mappings and additions:

Core Mappings. We adapt Fowler's accountability pattern concepts to the LLM-MAS domain through the following mappings:

- **Party** → **Agent**: An agent is an autonomous LLM-enabled unit capable of reasoning, tool use, and communication. Each agent possesses the capabilities, permissions, and resource access rights declared by its Agent Type.
- **Accountability** → **Agent Relationship**: A relationship between two agents that establishes accountability. Each relationship designates a *commissioner* (the agent delegating responsibility or oversight) and a *responsible* agent (the agent accountable for performing specific duties or meeting defined obligations). For example, an Architect agent (commissioner) supervising a Developer agent (responsible).
- **Party Type** → **Agent Type**: A declarative specification defining what an agent *can do* (capabilities such as code generation, testing, or design), what it *may access* (permissions including tools, APIs, and data resources), and *how it should interact* (protocols for communication and coordination).
- **Accountability Type** → **Relationship Type**: A schema that constrains which Agent Types may participate in specific accountability relationships. It defines permissible commissioner/responsible agent type pairings (e.g., "Architect may supervise Developer") and the interaction rules governing their collaboration (e.g., reporting frequency, escalation paths, decision rights).
- **Rules** → **Operating Scopes**: In the accountability pattern [6], connection rules provide a flexible knowledge-level mechanism whereby each accountability type contains a group of rules defining legal pairings of parent and child party types. We extend this concept through Operating Scopes, which not only constrain permissible agent type pairings but also attach quantitative characteristics and measurable duties to these pairings. Where connection rules specify *which* relationships may exist, operating scopes additionally define *how* these relationships must perform, establishing verifiable commitments such as minimum unit test coverage thresholds.

LLM-Specific Extensions. We introduce **Tool** as an architectural component, recognizing that LLM agents operate through tool use [23], e.g., *via* function calling, API invocation, code execution, to access external capabilities. **Agent Types** declare tool access permissions and usage protocols. **Agent Types** also specify LLM instantiation parameters including foundation model selection (e.g., GPT-4, Claude, Llama), inference configuration (temperature, top-k/top-p sampling), token limits, and system prompt templates that define the base instructions and behavioral constraints of agents.

Furthermore, we introduce **Operating Scopes**, which are clauses that attach measurable duties, constraints, or service-level agreements to **Agent Relationships**, making accountability concrete and auditable. Operating scopes transform abstract responsibilities into verifiable commitments, such as asserting minimum unit test coverage thresholds for LLM-based developer agents.

This extension provides LLM-MAS with structured accountability tracking while addressing domain-specific requirements including tool governance, LLM instantiation control, and agent observability.

B. Operational and Knowledge Layers

Following the Reflection architectural pattern [5] and principles from Adaptive Object Models [18], our architectural tactic achieves flexibility and governance through a clear separation between the *knowledge layer* (component types, rules, and constraints) and the *operational layer* (runtime instances and events) [6].

This separation, analogous to the Type Object pattern [19], enables runtime reconfiguration of agent relationships and responsibilities without code changes. The knowledge layer serves as a declarative metamodel that constrains and validates operational-layer instances, ensuring type safety and policy compliance.

Algorithm 1 EstablishAgentRelationship

Require: a_c commissioner agent, a_r responsible agent, relationship type t, scope set S

1: Assert a_c.*type* $\in$ t.*commissioners* and a_r.*type* $\in$ t.*responsibles*
2: Assert S conforms to t.*scope_schema*
3: Create relationship $r \leftarrow (a_c, a_r, t, now \ldots \varnothing, S)$
4: Persist r; emit audit event

Knowledge Layer. The knowledge layer maintains versioned catalogs that define the vocabulary and rules of the multi-agent system:

- **Agent Type Catalog**: Catalog of *Agent Type* specifications governing agent capabilities, such as permissions, interaction protocols, LLM instantiation parameters, permissible relationship participations;
- **Relationship Type Catalog**: Catalog of *Relationship Type* schemas constraining permissible Agent Type pairings, structural properties, and interaction rules;
- **Scope Schema Catalog**: Templates for *Operating Scopes* defining measurable duties and constraints attachable to agent relationships.

Changes to the knowledge layer are infrequent, governed by versioning policies, and propagate to operational instances through validation checks. This enables declarative reconfiguration, i.e., new agent types or relationship types can be introduced without modifying orchestration code.

Operational Layer. At runtime, the operational layer *reifies* [22] knowledge-layer definitions into concrete instances that execute the multi-agent system's work:

- **Agent Registry**: Active agent instances by Agent Types from the knowledge layer, tracking health, state, capability availability, and resource bindings;
- **Agent Relationship Store**: Relationship instances conforming to Relationship Types, recording commissioner, responsible, type reference, temporal period (creation..termination), and operating scope bindings with current compliance status. For example, a *supervises* relationship between *architect-01* (commissioner) and *frontend-angular-01* (responsible) might bind scope: *MinimumUnitTestCoverage{0.85}*.

Algorithm 1 illustrates the interplay between layers: when establishing a new agent relationship at runtime, the operational layer validates that both agent instances conform to the types permitted by the relationship type (lines 1–2), checks scope conformance (line 2), instantiates the relationship (line 3), and records an audit event (line 4). This ensures that all operational instances remain type-safe and traceable to knowledge-level policies.

V. Implementation

As a proof of concept, we implement our accountability-based architectural tactic by extending SALLMA (Software Architecture for LLM-based Multi-Agent systems) [20], a recently proposed architecture for orchestrating multiple LLM agents. SALLMA establishes a two-layer design, namely (i) an Operational Layer for real-time task execution and agent orchestration, and (ii) a Knowledge Layer for storing metamodels of workflows and agent configurations. SALLMA addresses challenges of single-agent LLM systems (lack of task customization, limited memory, restricted tool access). However, it does not provide explicit mechanisms for defining, validating, and auditing *accountability relationships* between agents. Our proof of concept shows how accountability concepts can be integrated into SALLMA's architecture, enhancing governance without disrupting its core orchestration capabilities.

A. SALLMA Architecture Overview

We implemented our accountability-based architectural tactic by extending SALLMA (Software Architecture for LLM-based Multi-Agent systems) [20]. This choice is motivated by three factors: (i) SALLMA was developed by our research group, providing us with complete access to its implementation and deep understanding of its design decisions; (ii) SALLMA already incorporates a two-layer architecture (Operational and Knowledge layers) that aligns naturally with the accountability pattern's separation of operational instances and knowledge-level types; and (iii) having full control over the codebase enables us to make the architectural modifications required to integrate accountability mechanisms without external dependencies or version compatibility constraints. While our accountability-based architectural tactic could be implemented in other multi-agent frameworks, SALLMA provides an ideal foundation for this proof-of-concept due to these practical advantages.

To understand how accountability mechanisms integrate with SALLMA's existing architecture, we first provide a brief overview of its core components. SALLMA orchestrates LLM agents through two layers. The **Operational Layer** handles real-time interactions via (i) an Intent Management Agent that parses incoming requests, (ii) a Workflow Management Agent that decomposes tasks and assigns them to specialized agents, (iii) a Routing Manager that routes requests to appropriate workflow instances, (iv) a Deployment Manager that instantiates workflows, and (v) a Cognitive Workflow Manager that executes task chains through containerized agents. The **Knowledge Layer** maintains three catalogs: (i) a Workflow Metamodel Catalog storing workflow configurations, (ii) an LLM Configuration Catalog (Agent Metamodel) with agent hyperparameters and specifications, and (iii) a Deployment Metamodels Catalog defining deployment characteristics.

SALLMA is based on a metamodel-driven design. Workflows and agents are defined declaratively in the Knowledge Layer and instantiated dynamically at runtime. However, SALLMA lacks explicit representation of *inter-agent accountability*. Agent interactions are governed by workflow sequences, but there is no explicit typing of relationships, no specification of which agent types may collaborate under

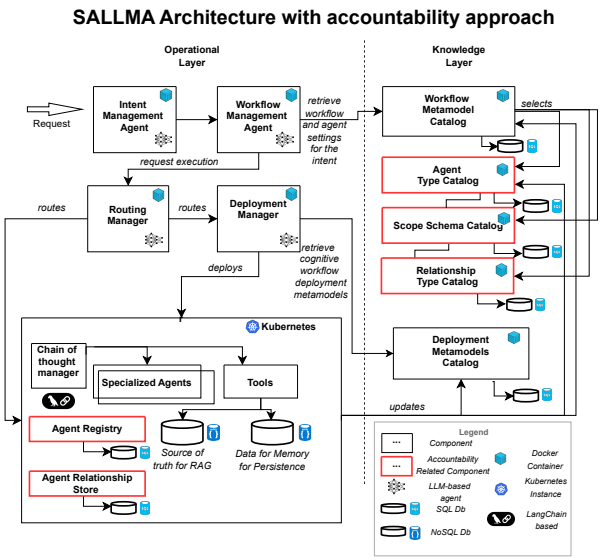

Fig. 3. Evolution of SALLMA architecture with accountability-based extensions.

which constraints, and no mechanism to track responsibility for outcomes or detect accountability violations.

B. Extending SALLMA with Accountability

We extend SALLMA by integrating accountability concepts into both layers (Figure 3), preserving backward compatibility while adding governance capabilities.

Knowledge Layer Extensions. We augment SALLMA's Knowledge Layer with three accountability-specific catalogs:

- **Agent Type Catalog (accountability-enhanced)**: Extends SALLMA's LLM Configuration Catalog with *Agent Type* specifications (Section IV), declaring agent capabilities, tool access permissions, and permissible participation in *Relationship Types*. For example, an *Architect* agent type specifies capabilities such as system design and task coordination, along with permitted access to necessary tools to access code repositories;
- **Relationship Type Catalog**: Catalog of *Relationship Type* schemas (Section IV) that define accountability relationships between agent types, specifying permissible pairings and related accountability-based relationships (e.g. *Architect* may act as commissioner in *supervises* relationships with *Developer* agents);
- **Scope Schema Catalog**: Templates for *Operating Scopes* defining measurable duties and constraints that are attachable to agent relationships (e.g. *MinimumUnitTestCoverage* specifying minimum unit coverage thresholds for a developer agent).

Operational Layer Extensions. We augment SALLMA's Operational Layer with two accountability-specific components:

- **Agent Registry**: Maintains a real-time inventory of active agent instances, tracking their instantiated Agent Types, active relationship bindings, operational status, resource allocation state and lifecycle events (instantiation, suspension, termination);
- **Agent Relationship Store**: Maintains a complete record of all accountability relationships between agents, enabling observability both at runtime and retrospectively.

Our proof-of-concept implementation extends SALLMA's architecture with versioned stores for knowledge-layer catalogs (Agent Type Catalog, Relationship Type Catalog, Scope Schema Catalog) and transactional databases for operational-layer accountability tracking (Agent Registry, Agent Relationship Store). For configuration representation, we use declarative formats (YAML in our examples) to specify agent types, relationship types, and operational instances, though any structured format (JSON, XML, TOML) could serve this purpose.

VI. Case Study: LLM-based Agentic Software Development Team

A. Proof-of-Concept Setup

This case study presents a proof-of-concept implementation comparing our accountability-enhanced SALLMA architecture against baseline SALLMA. We implemented both architectures using LLM-based agents powered by Anthropic's Claude Sonnet 4.5. The agents executed real software development tasks including architectural design decisions, code generation, and unit testing. The agents performed actual development work, enabling us to measure concrete differences in configuration burden, reconfiguration effort, and governance mechanisms between the two architectures. The evaluation focuses on assessing the feasibility of accountability-based orchestration and quantifying the effort required to reconfigure agent teams and track accountability of the agents under both architectural paradigms.

B. Scenario: Adding a Front-end Developer Agent

We evaluate our accountability-based architectural tactic through a concrete scenario, namely adding a specialized front-end developer to an existing web development team composed of LLM-based agents. Initially, the team comprises two agents building a web application.

The *Architect Agent* handles task coordination, design oversight, and architectural governance. In the scenario, the *Architect Agent* requires Angular application's state to be managed at the component level rather than globally, via the NgRx library. Moreover, it requires mandatory unit test coverage above a specified threshold. The *Angular Front-end Developer Agent* implements the Angular-based user interface while also executing unit tests.

The system has been operational for several iterations when a new requirement emerges: the application must support a React-based dashboard module alongside the existing Angular interface, necessitating the addition of a React-specialized Front-end Developer Agent. The module's state must similarly follow a component-level architectural tactic using the Redux library.

This scenario allows us to explore the core thesis of our architectural tactic, namely whether accountability relationships enable runtime agent addition without source code modifications while providing explicit accountability tracking and observability.

We compare two orchestration strategies, namely non-accountability baseline orchestration versus accountability-based declarative configuration, evaluating the effort required to integrate the new agent and the observability mechanisms provided by each architectural tactic.

C. Initial Configuration

The team operates with two agent types. The *Architect* type includes capabilities for system design, architectural oversight, and task coordination. The *Angular Front-end Developer* type includes capabilities for Angular UI development and unit test execution. A single *supervise* relationship type governs their interaction. It is a hierarchic relationship, between Architect agents as commissioners and Developer agents as responsible parties, with required level of quality assessed by unit tests, whose coverage level, 85% is defined at *Operating Scope* level.

At runtime, two agent instances are active: *architect-01* of type *Architect*, and *frontend-angular-01* of type *Angular Front-end Developer*. The relationship *architect-01* supervises *frontend-angular-01* is established with scopes requiring at least 85% of unit test coverage.

D. Architectural Tactic A: Non-Accountability Baseline Orchestration

In a multi-agent framework without structured accountability, agent interactions are encoded in orchestration logic. Adding the React developer requires code modifications across multiple components. The following code snippets are representative examples illustrating the types of changes required.

Step 1: Orchestration Logic Update. The workflow manager's task routing function must be modified to recognize the new *frontend-react-02* agent and dispatch React-related tasks accordingly. This typically involves conditional logic such as:

```
if task.component == "dashboard" and task.tech == "react
    ":
    assign_to_agent("frontend-react-02")
elif task.component == "main" and task.tech == "angular"
    :
    assign_to_agent("frontend-angular-01")
```

Step 2: Supervision Hierarchy Encoding. The architect's supervisory relationships are hard-coded in initialization or configuration files requiring manual updates:

```
supervised_agents = [
    "frontend-angular-01",
    "frontend-react-02"  # new entry
]
```

Step 3: Constraint and Compliance Management. In the baseline architectural tactic, architectural constraints (state management, scope) and quality requirements must be manually encoded and replicated across agent configurations:

```
architectural_constraints = {
    "frontend-angular-01": {
        "state_management": "NgRx", #required library
        "scope": "component_level" #as required, opposed
              to global scope.
    },

    "frontend-react-02": {
        "state_management": "Redux",
        "scope": "component_level"
    }
}

quality_constraints = {
    "frontend-angular-01": {
        "typology": "unit_test_threshold",
        "minimum_coverage": "85%"
    },
    "frontend-react-02": {
        "typology": "unit_test_threshold",
        "minimum_coverage": "85%"
    }
}
```

Step 4: New Agent Instantiation Prompt. The new agent must be instantiated with an imperative prompt embedding role, responsibilities, and constraints:

```
System: You are a React Front-end Developer agent (
     frontend-react-02). You are responsible for
     implementing React-based UI components for the
     dashboard module. You report to the Architect agent
      (architect-01). You must complete assigned tasks
     within 48 hours. All builds must pass before task
     completion.

Architectural Constraints:
- State management: Use Redux library exclusively
- Scope: Implement component-level state management (not
      global scope)
- Architecture: Follow component-based architecture
     using functional hooks pattern

Quality Constraints:
- Unit test coverage: Maintain minimum 85% test coverage
      threshold
- All unit tests must pass before task completion
- Follow the project's React coding standards
```

Impact. This architectural tactic requires modifications to at least three distinct code modules, manual prompt crafting with embedded policies, and no automated validation of consistency across these modules. If architectural constraints change (e.g. new design pattern adopted), all affected prompts and validation logic must be manually updated. Traceability is poor: determining which agents are affected by a constraint change requires code inspection.

E. Architectural Tactic B: Accountability-Based Configuration

In our accountability-enhanced architecture, the same agent addition is achieved entirely through knowledge-layer declarations, with zero operational code changes.

Step 1: Agent Type Specialization. Since React-specific constraints differ from Angular (Redux vs. NgRx for state management), we define a specialized *FrontendDeveloper_React* type via inheritance from the base *FrontendDeveloper* type:

```
# knowledge_layer/agent_types/frontend_react.yaml
agent_type_id: Frontend_Developer_React
extends: FrontendDeveloper
display_name: "React Frontend Developer"
capabilities:
```

```
  - react_ui_development
  - component_based_architecture
  - state_management
  - unit_testing
llm_config:
  model: "claude-sonnet-4-5-20250929"
  temperature: 0.7
  max_tokens: 8000
tools:
  - code_repository
  - build_system
  - react_dev_tools
architectural_constraints:
  framework: "React"
  state_management_library: "Redux"
permitted_relationships:
  - type: supervises
    role: responsible
    commissioner_types: [Architect]
```

Step 2: Agent Instance Registration. Upon deployment request, the agent is automatically instantiated and registered in the operational layer, specifically in the Agent Registry, enabling real-time observability and audit trail tracking:

```
# operational_layer/agents/frontend_react_02.yaml (auto-
    generated)
agent_id: frontend-react-02
instance_uuid: 7f3e4a2b-9c1d-4e5f-8a6b-3d2c1e0f9a8b
agent_type: Frontend_Developer_React
display_name: "React Developer - 02"
assigned_scope: "/frontend/react_module"
status: active
"
```

Step 3: Relationship Instantiation. Upon agent registration, the system automatically establishes accountability relationships based on predefined relationship types and attaches operating scopes in the Agent Relationship Store. This creates explicit, auditable accountability assignments that define which agent (commissioner) holds other agents (responsible) accountable for specific duties, with measurable constraints and verifiable compliance criteria:

```
# operational_layer/relationships/supervise_react_dev.
    yaml (auto-generated)
relationship_id: rel-supervise-frontend-react-02
relationship_uuid: 4b8c3d1e-7a2f-4c9b-8e5d-1f0a3b6c9d2e
relationship_type: supervises
commissioner: architect-01
responsible: frontend-react-02
status: active
operating_scopes:
  - scope_schema: UnitTestCoverageScope
    bindings:
      minimum_coverage: 0.85
      coverage_type: "unit_tests"
  - scope_schema: ArchitecturalComplianceScope
    bindings:
      state_management_library: "Redux"
      state_scope: "component_level"
```

Step 4: Agent Prompt Generation. Finally, the system automatically generates the agent's instruction prompt by composing its type definition, active relationships, and scope bindings, ensuring accountability constraints are explicit in the agent's operational context:

```
System: You are agent frontend-react-02 (instance_uuid:
    7f3e4a2b-9c1d-4e5f-8a6b-3d2c1e0f9a8b), instance of
    type Frontend_Developer_React.

CAPABILITIES (derived from type): react_ui_development,
    component_based_architecture, state_management,
    unit_testing.

TOOLS (from type): code_repository, build_system,
    react_dev_tools.

ACCOUNTABILITIES (from relationships):
1. [supervises] You are supervised by architect-01 (
    commissioner).
   Relationship: rel-supervise-frontend-react-02
   Operating Scopes:
   - UnitTestCoverageScope: Maintain minimum 85% unit
       test coverage (minimum_coverage: 0.85,
       coverage_type: unit_tests).
   - ArchitecturalComplianceScope: Use Redux for state
       management with component-level scope; follow
       functional hooks architecture pattern (
       state_management_library: Redux, state_scope:
       component_level).

ASSIGNED SCOPE: /frontend/react_module

Your responsibilities are governed by these
    accountability relationships. All actions must
    comply with the defined operating scopes. Report
    any scope violations or compliance issues to your
    commissioner (architect-01).
```

VII. Discussion

Table I summarizes the key differences between the two architectures for the agent addition task. No orchestration code changes are required in the accountability-based architecture. The operational layer's routing manager reads the Agent Relationship Store and dynamically routes tasks to agents based on their accountability relationships. If architectural constraints change (*e.g.* new state management library or architecture pattern adopted), only the operating scope bindings in the relationship configuration require updating. All affected agent prompts are regenerated automatically from the knowledge layer definitions. Traceability and observability are inherent: querying the Agent Relationship Store yields all agents affected by a given scope schema or relationship type, along with their accountability assignments and compliance status.

The code snippets presented in Section VI are illustrative examples showing the core changes required in the non-accountability baseline. Production implementations would include additional error handling, input validation, logging, telemetry, unit tests, and integration code. Based on typical software engineering practices, the non-accountability baseline architecture would require 200+ lines of production-quality code (approx. $10\times$ the shown snippets), while the accountability-based configuration files remain essentially complete as shown (36 lines), representing the actual declarative specifications needed.

The accountability-based architecture provides declarative runtime extensibility. The new agent is integrated without modifying orchestration logic, with relationship types and operating scope schemas enabling observability and dynamic prompt generation. In contrast, the non-accountability baseline architecture requires code changes across multiple modules, manual prompt crafting, and lacks automated traceability. When subsequently asked to change an architectural constraint for both developers (*e.g.* adopting a new state management pattern), the non-accountability baseline architecture requires code changes for both agents and manual re-crafting of both

TABLE I
COMPARATIVE METRICS: ADDING REACT DEVELOPER AGENT

Metric	Non-accountability baseline	Accountability-based
Orchestration code changes	3 modules	0 modules
Configuration files modified	0	2 files (agent + relationships)
Lines of code touched	200+ LOC	0 LOC
Lines of config added	–	36 config lines
Prompt construction	Manual	Auto-generated
Scope consistency validation	Manual inspection	Schema-validated
Traceability query	Code analysis	Relationship store query
Impact of scope change	Scattered updates	Single config update

agent prompts. In contrast, the accountability-based architecture requires updating only the operating scope bindings for each agent. Prompts are therefore automatically regenerated from the knowledge layer.

The metrics presented in Table I provide potential insights also into the qualitative aspects of our accountability-based architecture. The absence of orchestration code changes (0 vs. 200+ LOC) and automatic prompt generation from declarative specifications (36 config lines) suggest potential maintainability and scalability benefits. The introduction of scope validation and the possibility of querying the relationship store hint at governance and observability capabilities absent in the baseline. While these quantitative differences suggest qualitative benefits, their significance in real-world deployments would benefit from further empirical validation, including investigation of qualitative aspects such as developer cognitive load in development versus configuration tasks, related respectively to the baseline versus accountability-based architectures.

The accountability-based architecture shows advantages over non-accountability baseline orchestration methods: (i) the separation of knowledge and operational layers facilitates declarative reconfiguration without code changes, enabling runtime extensibility while maintaining type safety through knowledge-level validation; (ii) reuse is enhanced through structured catalogs of Agent Types, Relationship Types, and Scope Schemas, enabling systematic application of proven patterns and constraints across multiple agent instances; (iii) structured accountability tracking through relationships and operating scopes provides explicit governance mechanisms, automatic prompt generation, and comprehensive observability through the Agent Relationship Store.

However, the proposed architectural tactic introduces trade-offs that must be considered. For small-scale systems with few agents operating in relatively static configurations, the structured accountability infrastructure may represent unnecessary overhead. The conceptual complexity of distinguishing between knowledge-level types and operational-level instances, while beneficial at scale, introduces a learning curve and design effort that may not be justified for simple use cases. Additionally, the proposed architectural tactic requires upfront investment in defining agent types, relationship types, and scope schemas before operational benefits materialize. Organizations adopting this architectural tactic must establish the metamodel catalogs and validation infrastructure, which represents initial development cost compared to ad-hoc orchestration. Finally, the knowledge-level configuration demands disciplined change management practices, as type definitions and scope schemas govern multiple operational instances, knowledge layer modifications have cascading effects across the system. Teams lacking mature configuration governance may find it challenging to maintain consistency across knowledge and operational layers, potentially undermining the architecture's benefits.

A. Limitations

Our work presents limitations that should be acknowledged. First, our proof-of-concept implementation is built specifically on SALLMA, and while the accountability concepts are generalizable, broader experimentation should assess integration effort and challenges when adapting the accountability mechanisms to other reference architectures in LLM-MAS, such as [24], [25]. Second, our evaluation is limited to a single case study with a small team of LLM-based agents (2-3 agents) performing software development tasks. The scalability of the accountability mechanisms to larger agent populations (dozens or hundreds of agents) and more complex organizational structures remains untested in this study. Third, we have not evaluated runtime performance implications, which should be addressed. Finally, our case study focuses on a single application domain (software development); the generalizability of the accountability-based approach to other domains such as customer service, data analysis, or scientific research workflows requires further investigation.

VIII. FUTURE WORK

The present work provides conceptual foundations and shows feasibility of embedding accountability in LLM-based multi-agent systems through a proof-of-concept implementation and comparative case study. However, several directions merit further investigation. First, empirical validation through a systematic case study in a well-defined application domain with multiple agents and realistic operational complexity would strengthen the generalizability of our findings. Controlled experiments could measure quantitative operational metrics including configuration time and lines of configuration versus code touched, while also investigating qualitative aspects such as developer cognitive load during reconfiguration tasks compared to traditional coding. Second, runtime performance evaluation should assess memory footprint of the Agent Relationship Store, query performance for relationship lookups, and system throughput under varying agent population sizes to establish scalability envelopes for production deployments. Third, comparative studies against state-of-the-art frameworks and architectures in production-like scenarios

would quantify effort savings and identify integration patterns for incorporating accountability mechanisms into existing orchestration platforms. Finally, extending the accountability architectural components to support collective accountability for pre-configured agent teams, rather than only individual agents, would enable modeling of collaborative responsibilities and shared outcomes in complex multi-agent workflows.

IX. Conclusion

We presented an accountability-based architectural tactic for LLM-based multi-agent systems that adapts the accountability pattern through a set of architectural components, namely Agent Types, Relationship Types, and Operating Scopes, and the separation of knowledge and operational layers. This architectural tactic enables declarative specification of agent capabilities and constraints, dynamic reconfiguration without code changes, and systematic tracking of responsibilities through the Agent Relationship Store. Our proof-of-concept implementation extending SALLMA shows the feasibility of integrating accountability mechanisms into existing multi-agent architectures. The comparative case study shows that the accountability-based tactic reduces configuration burden for agent team reconfiguration, enables agent addition through declarative specifications rather than code modifications, and provides explicit observability and governance mechanisms compared to non-accountability baseline orchestration. Future work should pursue more rigorous empirical validation through systematic case studies to assess both quantitative metrics such as configuration effort and runtime performance, and qualitative aspects such as developer cognitive load, while establishing scalability limits for production deployments.

Acknowledgments

This manuscript was prepared with the assistance of an LLM to improve its readability.
This work was partially supported by the European Union under the Italian National Recovery and Resilience Plan (NRRP) of NextGenerationEU, partnership on "Telecommunications of the Future" (PE0000001 - program "RESTART").

References

[1] H. Chase, "LangChain: Building applications with LLMs through composability," GitHub repository, 2023. [Online]. Available: https://github.com/langchain-ai/langchain

[2] Significant Gravitas, "AutoGPT: An autonomous GPT-4 experiment," GitHub repository, 2023. [Online]. Available: https://github.com/Significant-Gravitas/AutoGPT

[3] J. Moura, "CrewAI: Framework for orchestrating role-playing, autonomous AI agents," GitHub repository, 2023. [Online]. Available: https://github.com/joaomdmoura/crewAI

[4] F. Bachmann, L. Bass, and M. Klein, *Deriving architectural tactics: A step toward methodical architectural design*. Carnegie Mellon University, Software Engineering Institute, 2003.

[5] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Sons, 1996, pp. 193–219.

[6] M. Fowler, *Analysis Patterns: Reusable Object Models*. Addison-Wesley Professional, 1996, pp. 17–33.

[7] L. Wang, C. Ma, X. Feng, Z. Zhang, H. Yang, J. Zhang, Z. Chen, J. Tang, X. Chen, Y. Lin, W. X. Zhao, Z. Wei, and J. Wen, "A survey on large language model based autonomous agents," *Frontiers of Computer Science*, vol. 18, no. 6, 2024.

[8] T. Guo, X. Chen, Y. Wang, R. Chang, S. Pei, N. V. Chawla, O. Wiest, and X. Zhang, "A large language model based multi-agents: A survey of progress and challenges," in *Proceedings of IJCAI*, 2024, p. 890.

[9] K. Tran, D. Dao, M.-D. Nguyen, Q.-V. Pham, B. O'Sullivan, and H. D. Nguyen, "Multi-agent collaboration mechanisms: A survey of LLMs," *arXiv preprint arXiv:2501.06322*, 2025.

[10] Q. Wu *et al.*, "AutoGen: Enabling next-gen LLM applications via multi-agent conversation," *arXiv preprint arXiv:2308.08155*, 2023.

[11] J. Hu, Y. Dong, S. Ao, Z. Li, B. Wang, L. Singh, G. Cheng, S. D. Ramchurn, and X. Huang, "Position: Towards a responsible LLM-empowered multi-agent systems," *arXiv preprint arXiv:2502.01714*, 2025.

[12] S. Raza, R. Sapkota, M. Karkee, and C. Emmanouilidis, "TRiSM for agentic AI: A review of trust, risk, and security management in LLM-based agentic multi-agent systems," *arXiv preprint arXiv:2506.04133*, 2025.

[13] Y. Liu, S. K. Lo, Q. Lu, L. Zhu, D. Zhao, X. Xu, S. Harrer, and J. Whittle, "Agent design pattern catalogue: A collection of architectural patterns for foundation model based agents," *Journal of Systems and Software*, vol. 218, 2024.

[14] R. Conte and M. Paolucci, "Responsibility for societies of agents," *Journal of Artificial Societies and Social Simulation*, vol. 7, no. 4, 2004.

[15] T. J. Norman and C. Reed, "A model of delegation for multi-agent systems," in *Foundations and Applications of Multi-Agent Systems*. Springer, 2002, pp. 185–204.

[16] C. Castelfranchi and R. Falcone, "Towards a theory of delegation for agent-based systems," *Robotics and Autonomous Systems*, vol. 24, no. 3-4, pp. 141–157, 1998.

[17] W. Benghabrit, H. Grall, J.-C. Royer, M. Sellami, K. Bernsmed, and A. S. D. Oliveira, "Abstract accountability language," in *Proceedings of 8th IFIP International Conference on Trust Management (IFIPTM)*, ser. IFIP Advances in Information and Communication Technology, vol. 430. Berlin, Heidelberg: Springer, 2014, pp. 239–256.

[18] J. W. Yoder and R. E. Johnson, "The adaptive object-model architectural style," in *Proceedings of Working IEEE/IFIP Conference on Software Architecture (WICSA)*, 2002, pp. 3–27.

[19] R. E. Johnson and B. Woolf, "Type object," in *Pattern Languages of Program Design 3*, R. Martin, D. Riehle, and F. Buschmann, Eds. Addison-Wesley, 1998, pp. 47–65.

[20] M. Becattini, R. Verdecchia, and E. Vicario, "SALLMA: A software architecture for LLM-Based multi-agent systems," in *Proceedings of the Software Architecture Trends Symposium (SATrends)*, 2025, pp. 1–4.

[21] Y. Yang, Q. Peng, J. Wang, Y. Wen, and W. Zhang, "LLM-based multi-agent systems: Techniques and business perspectives," *arXiv preprint arXiv:2411.14033*, 2024.

[22] A. Olivé, "Relationship reification: A temporal view," in *Proceedings of 11th International Conference on Advanced Information Systems Engineering (CAiSE)*, ser. LNCS, vol. 1626. Berlin, Heidelberg: Springer, 1999, pp. 396–410.

[23] Z. Shen, "LLM with tools: A survey," *arXiv preprint arXiv:2409.18807*, 2024.

[24] A. Bucaioni, M. Weyssow, J. He, Y. Lyu, and D. Lo, "A functional software reference architecture for LLM-integrated systems," in *2025 IEEE 22nd International Conference on Software Architecture Companion (ICSA-C)*. IEEE, 2025, pp. 1–5.

[25] Q. Lu, L. Zhu, X. Xu, Z. Xing, S. Harrer, and J. Whittle, "Towards responsible generative AI: A reference architecture for designing foundation model based agents," in *2024 IEEE 21st International Conference on Software Architecture Companion (ICSA-C)*. IEEE, 2024, pp. 119–126.