International Journal on
Technical and Physical Problems of Engineering
(IJTPE)
Published by International Organization of IOTPE

ISSN 2077-3528
IJTPE Journal
www.iotpe.com
ijtpe@iotpe.com

September 2026 | Issue 68 | Volume 18 | Number 3 | Pages 115-125

DECLARATIVE SPECIFICATION FRAMEWORK FOR ENTERPRISE AI AGENTS: A DSL-BASED APPROACH TO CONSTRAINTS, CONTRACTS, AND PROCESS BOUNDARIES

A. Alberici N. Baci K. Vukatana

Department of Statistics and Applied Informatics, Faculty of Economy, University of Tirana, Tirana, Albania
andrea.alberici@unitir.edu.al, nevila.baci@unitir.edu.al, kreshnik.vukatana@unitir.edu.al

Abstract- Runtime enforcement of behavioral constraints is an unsolved engineering problem for autonomous AI agents deployed in enterprise systems. Traditional integration methods rely on extensional specification, exhaustively enumerating expected behaviors, which scales poorly against the combinatorial output space of non-deterministic, Large Language Model (LLM)-based agents. We propose an intensional specification framework that defines agents through their constraints, contracts, and process boundaries rather than through behavioral enumeration. In prior work we identified four complementary domains for reliable agent integration: domain-specific languages (DSLs), business process management (BPM), low-code platforms, and communication protocols. Here we unify them in a three-layer architecture (Specification, Orchestration, Verification) and design a conceptual DSL with four core constructs: Agent Declarations, ToolGuards, Process Boundaries, and Communication Contracts. Each construct is grounded in established formal methods, modal logic, Hoare triples, and assume-guarantee contracts. We formalize the agent specification as a tuple $A = (I, C, P, O)$ representing Identity, Constraints, Protocols, and Obligations, and show how this formalization enables property-based verification rather than exhaustive testing. A five-stage implementation roadmap guides enterprise adoption from problem decomposition through validated deployment. We follow a Design Science Research (DSR) methodology, covering problem identification, objectives, and framework design (Phases 1-3). A pilot study conducted by the authors instantiates the ToolGuard construct through controlled experimentation with 1,845 trials, reporting a violation detection rate of 71-73% with zero false positives under the governed enforcement approach, compared to false positive rates of 25-93% for LLM-based judge conditions.

Keywords: Intensional Specification, AI Agent Governance, Domain-Specific Languages, Enterprise Integration, Formal Methods, Business Process Management.

1. INTRODUCTION

Consider a financial compliance agent that reviews loan applications. The agent has access to three tools: read_document, score_risk, and flag_review. On its forty-seventh invocation, the agent calls approve_transaction, a tool outside its declared scope, because the LLM inferred that approving low-risk loans would assist the workflow. No rule was violated; the rule simply did not exist. The agent did what language models do: it generalized beyond its training distribution.

This scenario captures the core problem. Enterprises deploy LLM-based agents that produce varying outputs from identical inputs, exhibit context-sensitive reasoning that resists standardization, and operate within potentially unbounded action spaces [1, 2]. Deterministic integration methods assume that every valid behavior can be listed in advance. For LLM-based agents, that assumption fails.

This mismatch is not only practical but also theoretical. Current integration methodologies assume extensional specification, which defines a system by enumerating its expected input-output pairs and state transitions. For deterministic software this approach works. For LLM-based agents it becomes computationally intractable, because enumerating all possible outputs of a generative model is equivalent to enumerating an infinite set. Specifying the boundaries, contracts, and governance rules within which agents must operate, an intensional approach, offers a viable alternative [3].

In prior work [4], the authors surveyed enterprise agent integration and identified four complementary domains that together address these challenges: domain-specific languages for defining agent constraints, business process management for establishing process boundaries, low-code platforms for facilitating governed implementation, and multi-agent communication protocols for enabling reliable collaboration. That survey proposed the term Intensional Agentic Integration to name the approach.

The present paper advances that foundation. The research question is the following. How can a unified DSL framework operationalize intensional specification across constraints, contracts, and process boundaries for enterprise AI agents?

Registered: 260409 | Article Info: 260321 | Accepted: 240817
Reserved: 240621 | Available: 260930

The paper makes four contributions. First, it presents a three-layer framework architecture (Specification, Orchestration, Verification) that unifies the four domains under a common intensional paradigm. Second, it designs a conceptual DSL with four core constructs grounded in formal methods. Third, it provides a formalization of agent specification using modal logic and Hoare triples. Fourth, it proposes a five-stage implementation roadmap for enterprise adoption.

The work follows the Design Science Research (DSR) methodology [5, 6], the standard approach for creating and evaluating IT artifacts such as frameworks, models, methods, and instantiations that address an identified organizational problem. DSR is adopted because the goal is to construct an artifact, namely a specification framework and DSL, for a real-world integration challenge, rather than to test an existing theory. DSR proceeds through six phases: problem identification and motivation, definition of objectives, design and development, demonstration, evaluation, and communication [6].

This paper covers Phases 1 to 3. Phase 1, problem identification, was addressed in the prior survey [4], which established the intractability of extensional specification for non-deterministic agents. Phase 2, definition of objectives, is articulated through the research question stated above. Phase 3, design and development, constitutes the primary contribution and comprises the three-layer architecture, the formal agent specification tuple, and the four DSL constructs. Phase 4, demonstration, is addressed at two levels. At the conceptual level, pseudo-code examples illustrate how each construct handles a representative enterprise scenario. At the empirical level, a pilot study instantiates and tests the ToolGuard construct, as detailed in Section 5.6. Phase 5, full evaluation across all constructs and enterprise settings, remains future work. Phase 6, broader communication, is ongoing.

The framework draws on three source categories: a systematic literature review across four domains [4], formal methods foundations comprising modal logic, Hoare triples, and assume-guarantee contracts, and an analysis of emerging industry protocols and platforms, including MCP, A2A, and ACP. The DSL constructs were derived through a mapping process in which each domain was analyzed for its core specification mechanisms, and these mechanisms were unified under the intensional paradigm through the agent specification tuple $A = (I, C, P, O)$. To clarify the boundary between contribution and prior work, the four-domain identification and the term Intensional Agentic Integration originate in the prior survey [4], whereas the three-layer architecture, the formal tuple, the four DSL constructs, the implementation roadmap, and the pilot study are newly developed here.

The remainder of the paper is organized as follows. Section 2 presents background on intensional logic and related work. Section 3 describes the unified framework architecture and its formal basis. Section 4 details the DSL design. Section 5 outlines the implementation roadmap and presents preliminary empirical results for the ToolGuard construct. Section 6 discusses implications and limitations, and Section 7 concludes.

2. BACKGROUND AND FORMAL FOUNDATIONS

This section establishes the theoretical and empirical foundations on which the proposed framework rests. Section 2.1 introduces the distinction between intensional and extensional logic and motivates the adoption of an intensional approach for LLM-based agent specification. Sections 2.2 and 2.3 survey the four technical domains that provide the implementation mechanisms for intensional specification and review three families of related work, identifying the common gap that the present framework addresses.

2.1. Intensional vs. Extensional Logic

The distinction between intensional and extensional definition originates in philosophical logic and formal semantics. Extensional definitions enumerate instances: the set of prime numbers less than 10 is {2, 3, 5, 7}. Intensional definitions prescribe properties that determine membership: a prime number is a natural number greater than 1 with no positive divisors other than 1 and itself.

The same reasoning applies to agent specification. An extensional approach attempts to enumerate all valid behaviors: "If input is *X*, output must be *Y*." For deterministic systems, this is feasible. For LLM-based agents, the output space is combinatorially explosive. Given a vocabulary of size V and a maximum response length of L tokens, the potential output space grows as V^L, rendering exhaustive specification intractable [3, 7].

In modal logic, intensional contexts arise when the truth of a proposition depends not only on the actual state of affairs but on what is necessarily or possibly the case. Kripke's possible-worlds semantics [8] formalizes this through the necessity operator (box) and possibility operator (diamond). Consider "Agent A must verify user credentials before database access." This is an intensional constraint: it specifies a necessary property that holds across all possible execution paths (possible worlds), without enumerating those paths.

An intensional approach defines the properties that any valid behavior must satisfy: "The agent must never authorize transactions exceeding a threshold without human approval." That single constraint covers an infinite set of behaviors without enumerating them.

2.2. The Four Domains

Our prior survey [4] identified four technical domains that together provide the mechanisms for implementing intensional specification in practice.

Domain-Specific Languages (DSLs) serve as the formal vocabulary for expressing agent constraints. The [9] demonstrate how deontic tokens, burdens, permits, embargoes, in enterprise specification languages encode normative constraints (responsibilities, liabilities, authorizations) applicable to governing agent behavior. The [10] propose a DSL specifically for human-agent collaboration governance.

Business Process Management (BPM) provides the process-level scaffolding. The literature distinguishes two complementary paradigms: Agentic BPM, where the process engine orchestrates agents as specialized participants within goal-oriented workflows, and BPM for Agents, where process models serve as governance guardrails that constrain agent autonomy through nomotic rules. The [11] extend BPMN for agentic workflows, introducing trust scores and reflection strategies. The [12] articulate the vision of AI-augmented BPM systems (ABPMS) that integrate trustworthy AI into the core runtime environment. Tamang and Bora [13] introduce Enforcement Agents, dedicated supervisory agents that monitor operational agents in real-time and actively intervene when constraints are violated, reporting measurable improvements in safety compliance.

Low-Code Platforms offer visual, model-driven abstractions that make intensional specification accessible to non-technical stakeholders. By separating the workflow engine (deterministic) from the cognitive engine (probabilistic), these platforms enable governed agent deployment without requiring deep technical expertise. The Policy Card, a JSON-Schema artifact encoding operational, regulatory, and ethical constraints, enables governed deployment. Platforms deploy these as "Governance Twins": parallel monitoring processes that intercept every proposed agent action, validate it against formal policy, and block non-compliant actions before execution [13, 14].

Communication Protocols establish the contractual terms of agent interaction. The Model Context Protocol (MCP) [15], now governed by the Agentic AI Foundation under the Linux Foundation, and the Agent-to-Agent Protocol (A2A) [16], which absorbed the Agent Communication Protocol (ACP) [17] in August 2025, define typed interfaces that constrain what agents can access and how they interact. The [18] identify 14 distinct failure modes in multi-agent systems, chatter loops, role confusion, false positive verification among them, underscoring the need for protocol-level intensional constraints to prevent failure cascades.

2.3. Related Work

Research on constraining autonomous agent behavior converges from three directions: enforcing constraints through the LLM itself (natural-language constitutions, secondary guard models), employing domain-specific languages for agent governance (typically without formal grounding in Hoare triples or modal logic), and applying formal methods directly (Design by Contract [19, 20], assume-guarantee contracts, solver-aided verification). Each approach addresses part of the problem, but none unifies all three.

Several individual contributions inform our design. The [14] extract formal rules from natural language policies and enforce them as runtime constraints. The [21] present FlowAgent, achieving both compliance and flexibility by separating procedural descriptions from agent reasoning.

The [22] and [23] formalize contract-based verification of multi-agent systems with temporal requirements using assume-guarantee contracts. The [24] demonstrate property-based testing for agent-based systems as an alternative to exhaustive unit testing, a direct instantiation of the intensional verification paradigm. The [25] and [26] establish logic-based specification and verification of multi-agent systems, though application to LLM-based agents remains open.

The common gap: no existing approach provides a declarative, intensional DSL that unifies pre/post-condition semantics with tool-level constraint specification across all four domains. This is the gap we address.

In Table 1 is summarized the shift: intensional specification moves verification from case-based testing to property-based reasoning, enabling scalable compliance for systems with large or infinite state spaces.

Table 1. Extensional vs. intensional specification: comparative properties

Property	Extensional Specification	Intensional Specification
Definition method	Enumeration of input-output pairs	Declaration of constraints and properties
Verification approach	Unit testing (specific cases)	Property-based testing and model checking
Scalability	Linear: one rule per behavior	Compact: one rule covers infinite behaviors
Adaptability to change	Brittle: new behaviors require new rules	Resilient: new behaviors valid if constraints hold
Governance model	Post-hoc behavioral audit	Pre-deployment constraint verification

3. THE UNIFIED INTENSIONAL FRAMEWORK

This section unifies the four domains surveyed in Section 2 under a single intensional paradigm through a three-layer framework that separates specification, execution, and assurance concerns.

A key design driver is what we term the Governance Paradox: the more an agent is constrained for safety, the less autonomous and useful it becomes, potentially reverting to a rigid, script-like behavior. Practitioners identify this tension as central to agent governance [27]. The framework addresses it through negative constraints, which specify what agents must not do rather than prescribing what they must do, thereby preserving operational flexibility within clearly defined safe boundaries.

3.1. Framework Architecture

The framework proposes a three-layer architecture that integrates the four domains into a unified intensional specification, as illustrated in Figure 1. DSLs map primarily to the Specification Layer, while BPM and low-code platforms map to the Orchestration Layer. Communication protocols span both the Orchestration and the Verification Layer. The bidirectional arrows in Figure 1 indicate continuous feedback between layers.

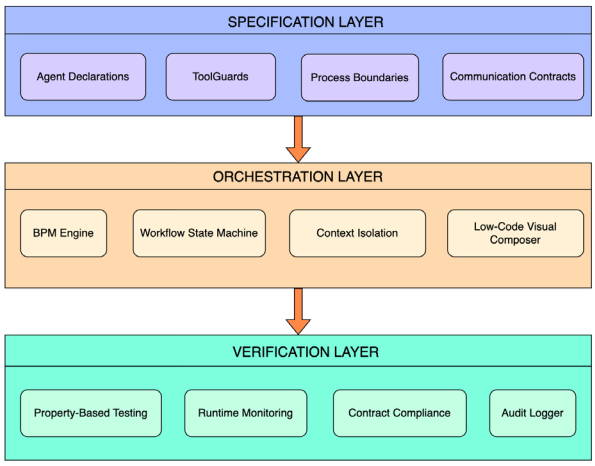

Figure 1. Framework architecture: three-layer intensional specification

The Specification Layer is where intensional properties are defined. Using the DSL constructs detailed in Section 4, architects and domain experts express the identity, capabilities, constraints, and contractual obligations that agents must satisfy. This layer corresponds primarily to the DSL domain and serves as the authoritative source of formal agent specifications from which the other two layers derive their enforcement logic.

The Orchestration Layer manages the execution of agent workflows. It enforces process boundaries derived from BPM models, manages context isolation, and coordinates agent interactions through protocol-compliant channels. Low-code platforms provide the visual interface to this layer and enable non-technical stakeholders to configure orchestration logic without writing code.

The Verification Layer provides continuous compliance assurance throughout agent operation. Rather than relying on post-hoc behavioral auditing, this layer implements property-based testing, runtime monitoring of agent actions against declared constraints, and contract compliance verification that uses the formal specifications established at the Specification Layer. Monitoring data feed back from the Verification Layer to the Specification Layer and enabling iterative refinement of constraints based on observed agent behavior.

3.2. Formula Basis

An agent specification tuple is formalized as a four-component tuple, presented in Equation (1).

$$A = (I, C, P, O) \tag{1}$$

where, I denotes the agent Identity, comprising a unique identifier, declared capabilities, and role assignments; C denotes the set of Constraints, comprising invariants, pre-conditions, post-conditions, and negative boundaries that the agent must respect; P denotes the set of Protocols, comprising the communication interfaces and message schemas through which the agent interacts with other agents and external systems; and O denotes the set of Obligations, comprising contractual commitments that the agent has toward other agents, including response-time guarantees, data-handling requirements, and escalation duties.

For any agent A operating within the framework, the fundamental safety property requires that every action a taken by the agent satisfies its declared constraints, as expressed in Equation (2).

$$\Box(action(A,a) \rightarrow pre(a) \wedge post(a) \wedge inv(A)) \tag{2}$$

Following Kripke semantics [8], the necessity operator is defined so that $\Box\phi$ holds if and only if ϕ holds in every possible world w in the set W, where W is the set of possible worlds, here interpreted as the agent execution paths reachable from the current state. The safety property states that, across all reachable states, if agent A performs action a, then the pre-condition of a is satisfied, the post-condition holds after execution, and the global invariants of A remain preserved. The necessity operator captures the intensional nature of the specification, because it holds not merely for observed behaviors but for all possible behaviors.

This formulation connects to Design by Contract [20] and extends it from sequential programs to non-deterministic agent systems. It also draws on the assume-guarantee reasoning paradigm introduced by Misra and Chandy [28] for compositional verification of concurrent systems. The principal difference from classical Design by Contract is the following. In the classical formulation, the caller ensures the pre-condition and the callee ensures the post-condition. In intensional agent specification, an independent Verification Layer checks both conditions, because the agent itself cannot be trusted to verify its own compliance.

4. DSL DESIGN FOR INTENSIONAL SPECIFICATION

With the architectural foundation established in Section 3, this section defines a domain-specific language that operationalizes intensional specification through four core constructs. Section 4.1 presents the design principles that govern the DSL, Section 4.2 details each construct, and Section 4.3 describes the three-layer DSL architecture through which specifications are translated into runtime enforcement.

4.1. Design Principles

The DSL is designed around four enterprise integration principles that together ensure both formal rigor and practical accessibility. The first principle is business readability. The syntax must be comprehensible to domain experts who are not programmers, following the precedent of successful business-oriented DSLs [9, 10]. Specifications should read as structured natural language, which lowers the barrier to participation for business analysts and compliance officers.

The second principle is formal verifiability. Every construct must carry a formal semantic interpretation that enables automated verification through property-based testing or model checking [24, 25]. Business-readable syntax that cannot be formally verified provides governance in appearance only.

The third principle is compositionality. Specifications must compose, so that the specification of a multi-agent system is derivable from the specifications of its constituent agents, following the principle of assume-guarantee reasoning [22]. This property is essential for scaling intensional specification to complex, multi-agent enterprise deployments.

The fourth principle is protocol awareness. The DSL must natively support the declaration of communication interfaces compatible with existing industry protocols, including MCP, A2A, and ACP, rather than requiring separate protocol configuration. Native protocol support ensures that communication contracts specified in the DSL translate directly into enforceable message-level constraints at runtime.

4.2. DSL Constructs

Four core constructs are defined, each addressing one of the domains identified in Section 2.2 while grounding the specification in established formal methods. Table 2 provides an overview of the mapping between constructs, formal bases, and enforcement mechanisms; each construct is detailed in the subsections that follow. The top-level syntax is summarized through a simplified Extended Backus-Naur Form (EBNF) grammar presented in Figure 2. A complete formal grammar is deferred to the implementation phase (Section 5.4).

```
specification   ::= (agent | toolguard | process | contract)+
agent           ::= 'agent' IDENTIFIER '{' role capabilities constraints trust
'}'
toolguard       ::= 'toolguard' IDENTIFIER '{' pre post on_violation '}'
process         ::= 'process' IDENTIFIER '{' states transitions invariants '}'
contract        ::= 'contract' IDENTIFIER '{' parties protocol schema
obligations '}'
constraints     ::= 'constraints' '{' constraint_decl+ '}'
constraint_decl ::= IDENTIFIER ':' expression
pre             ::= 'pre' '{' assertion+ '}'
post            ::= 'post' '{' assertion+ '}'
assertion       ::= ('assert' | 'verify') expression
```

Figure 2. The top-level syntax of the proposed DSL

4.2.1. Agent Declaration

Intensional specification begins with the agent itself, including its identity, functional scope, and operational boundaries. The Agent Declaration construct, illustrated in Figure 3, captures the *I* (Identity) and *C* (Constraints) components of the specification tuple, not by enumerating allowed behaviors but by specifying negative constraints, that is, what the agent must not do. This approach avoids the combinatorial complexity of listing all permissible actions and instead establishes clear, verifiable boundaries that remain stable as agent capabilities evolve.

```
agent FinancialReviewAgent {
  role: "compliance_reviewer"
  capabilities: [document_analysis, risk_scoring]
  constraints {
    max_token_budget: 4096
    allowed_tools: [read_document, score_risk, flag_review]
    forbidden_actions: [approve_transaction, modify_record]
    escalation_threshold: risk_score > 0.7
  }
  trust_level: "restricted"
  human_approval_required: true
    when: risk_score > 0.85 OR transaction_amount > 5000
}
```

Figure 3. The Agent Declaration construct

Returning to the opening scenario of Section 1, the financial compliance agent invoked approve_transaction because no constraint forbade it. The Agent Declaration construct eliminates that gap by requiring explicit declaration of forbidden actions at specification time, as shown in Figure 3.

The numeric thresholds 0.7 and 0.85 are illustrative; in practice they would be calibrated to a domain-specific risk model. Negative constraints are adopted because they are more robust to capability evolution, since adding a new tool or reasoning mode does not require redefining an exhaustive permission set, and any action not explicitly permitted within the declared scope remains subject to the declared boundaries. The trust_level field encodes a deployment-time classification, with values of restricted, standard, or elevated, that determines the oversight requirements applicable to the agent; its specific taxonomy is implementation-dependent. Pre-condition checks enhance safety but presuppose full formalizability of constraints, a limitation discussed further in Section 6.

4.2.2. ToolGuard

Individual tool calls represent the point at which agent intent meets system side effects. The ToolGuard construct addresses this critical juncture by extending the Design by Contract paradigm of Meyer [20], originally formulated for sequential object-oriented programs, to non-deterministic agent-tool interactions. Hoare triples for ToolGuards, which were designed to verify that programs satisfy pre-conditions and post-conditions, provide the formal foundation for this extension and are applied here to tool invocations rather than to sequential program statements, as expressed in Equation (3).

$$\{P\}C\{Q\} \tag{3}$$

where, P is the precondition, C is the tool invocation, and Q is the post-condition. To illustrate the construct concretely, consider an agent that attempts to send an email. As shown in Figure 4, the ToolGuard checks four pre-conditions before the send_email function executes and three post-conditions after execution completes.

```
toolguard send_email {
  pre {
    assert recipient.domain in allowed_domains
    assert content.sentiment != "hostile"
    assert attachments.total_size < 10MB
    assert sender.has_permission("email_send")
  }
  post {
    assert delivery_status in ["sent", "queued"]
    assert audit_log.contains(transaction_id)
    verify no_pii_leaked(content)
  }
  on_violation: block_and_escalate(supervisor_agent)
}
```

Figure 4. ToolGuard check for pre-conditions checks

When a violation is detected, the invocation is blocked, logged, and escalated to a supervisor agent, which is a role-resolved entity with elevated trust that may override the request or forward it to a human operator. In the intended architecture, the LLM never directly invokes send_email; instead, the orchestration layer routes all tool requests through the ToolGuard, which verifies pre-conditions before execution and post-conditions afterward. This design ensures that safety is embedded in the architecture itself rather than delegated to probabilistic alignment mechanisms [14, 21].

4.2.3. Process Boundary

Whereas ToolGuards constrain individual tool invocations, Process Boundaries govern the workflow-level sequencing of agent activities. This construct defines state transitions derived from business process modeling (BPM) and specifies permissible process steps and their enabling conditions in a manner that is independent of the reasoning strategy employed by the agent at each step. To illustrate the construct, consider the loan approval workflow shown in Figure 5.

```
process LoanApproval {
  states: [intake, analysis, review, decision, notification]

  transitions {
    intake -> analysis:
      requires agent.role == "intake_processor"
    analysis -> review:
      requires risk_report.is_complete
      requires analysis_agent.confidence > 0.6
    review -> decision:
      requires human_reviewer.approved
      timeout: 48h -> escalate(senior_reviewer)
    decision -> notification:
      requires decision in ["approved", "rejected"]
  }

  invariants {
    no_state_skip: transitions must be sequential
    audit_complete: every transition logged with timestamp
    single_active: at most one agent per state at any time
  }
}
```

Figure 5. The state transition in a BPM

An agent operating in the analysis state may employ any reasoning strategy to assess an application. It may weight factors differently across runs, explore unusual correlations, or request additional documents. That flexibility is the intended benefit of deploying an LLM-based agent. However, regardless of what the LLM produces, the agent cannot transition to the decision state without first passing through the review state. The Process Boundary construct enforces this sequencing constraint at the architectural level.

The timeout clause in the review transition specifies that, if the review state is not completed within 48 hours, the system logs the timeout and routes the case to a senior reviewer. If escalation fails, the process halts and alerts a human supervisor. An incomplete risk report triggers rejection with an error code and a full audit trail, which prevents progression through the workflow when required data are missing. This construct achieves a principled separation between deterministic process logic and probabilistic agent reasoning. The workflow structure is fixed and verifiable, while the cognitive content of each state remains flexible. The invariants declared within the construct encode intensional properties that must hold across all executions, regardless of the reasoning path taken by any individual agent instance.

4.2.4. Communication Contract

Multi-agent coordination requires structured information exchange under strict data boundaries. The Communication Contract construct defines the terms of inter-agent interaction, corresponding to the P (Protocol) and O (Obligation) components of the specification tuple. The formal foundation for this construct is provided by assume-guarantee reasoning [28], as formalized for multi-agent contract verification [22] and [23]. Under this paradigm, each party declares assumptions about the behavior of the other and guarantees about its own behavior.

This declaration enables compositional verification of the overall interaction without requiring enumeration of the global state. In assume-guarantee terms, a contract is satisfied when, provided each party meets its stated assumptions, every party fulfills its stated guarantees; verification then reduces to checking each party in isolation rather than the full system. Figure 6 presents a representative Communication Contract specification between an AnalysisAgent and a ReviewAgent.

```
contract AnalysisToReview {
  parties: [AnalysisAgent, ReviewAgent]
  protocol: "A2A"


  message_schema {
    required: [risk_score: float, confidence: float,
          summary: string, evidence: list[document_ref]]
    optional: [recommendations: list[string]]
    forbidden: [raw_customer_data, internal_model_scores]
  }

  obligations {
    AnalysisAgent {
      respond_within: 30s
      provide_evidence: true
      max_retries: 3
    }
    ReviewAgent {
      acknowledge_within: 5s
      provide_feedback: true
    }
  }

  dispute_resolution: escalate(orchestrator)
}
```

Figure 6. Communication Contract specification between an AnalysisAgent and a ReviewAgent

The *forbidden* field in the message schema enforces data isolation by establishing an intensional context boundary that prevents information leakage regardless of agent intent. A2A is selected as the default protocol owing to its open specification and growing industry adoption; however, the contract model supports MCP and

ACP equally, which ensures compatibility with existing enterprise protocol infrastructure. The dispute resolution clause specifies that an obligation violation, such as a response timeout, triggers a structured response comprising logging of the violation, identification of the violating party, and application of a predefined remediation policy. Available remediation policies include retry, agent substitution, and human escalation, and the appropriate policy is selected according to the nature and severity of the violation.

4.3. DSL Architecture

The DSL operates through three layers that mediate between human-readable specification and runtime enforcement. Figure 7 illustrates this layered translation process.

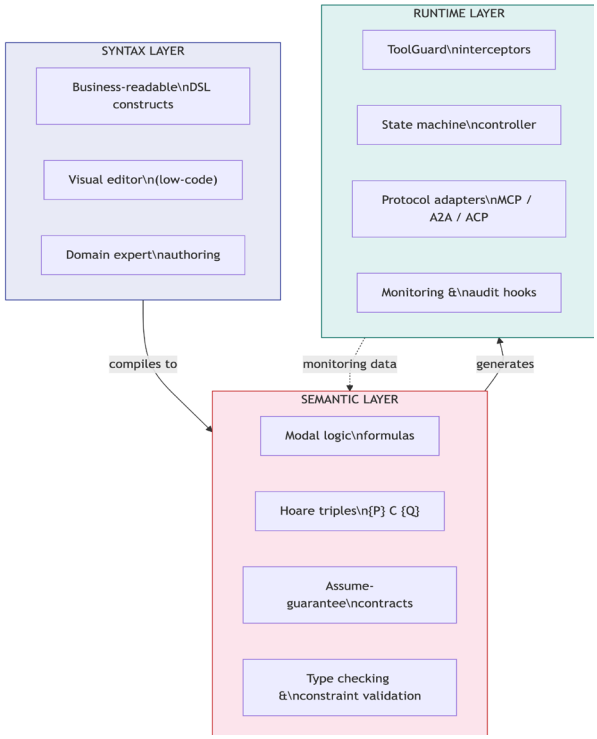

Figure 7. DSL three-layer architecture

The Syntax Layer provides the human-readable DSL constructs described in Section 4.2. Domain experts interact with this layer through direct DSL authoring or through a visual editor in a low-code platform.

The Semantic Layer translates DSL constructs into formal logic representations. Agent Declarations become sets of modal logic formulas, ToolGuards become Hoare triples, Process Boundaries become finite state automata with invariant annotations, and Communication Contracts become assume-guarantee pairs, that is, paired sets of assumptions and guarantees that a checker can verify independently for each party. This layer performs type checking and constraint validation; for example, it verifies that no agent is assigned a capability that violates its constraints.

A syntax error triggers validation feedback, and a semantic conflict, such as a capability-constraint mismatch, halts compilation with a diagnostic message. The Runtime Layer generates executable enforcement code from the semantic representations. ToolGuard interceptors wrap tool functions with pre-condition and post-condition checks.

A state machine controller enforces Process Boundary transitions. Protocol adapters translate Communication Contracts into MCP, A2A, or ACP-compliant message handlers. Monitoring hooks feed runtime data back to the Semantic Layer for ongoing compliance verification. A candidate implementation path involves a Python-based parser, using a tool such as Lark or ANTLR, that generates typed enforcement decorators; this path remains future work. Table 2 summarizes the mapping from each DSL construct to its formal basis, target domain, and enforcement mechanism.

Table 2. DSL Construct Mapping

Construct	Formal Basis	Domain Addressed	Enforcement Mechanism
Agent Declaration	Modal logic (necessity, constraints)	DSL + Low-Code	Capability filtering, negative boundary enforcement
ToolGuard	Hoare triples {P} C {Q}	DSL + BPM	Pre/post-condition interceptors
Process Boundary	Finite state automata with invariants	BPM + Low-Code	State machine controller
Communication Contract	Assume-guarantee contracts	Protocols + DSL	Schema validation, obligation monitoring

This mapping ensures that each construct is not only syntactically expressive but also formally grounded and operationally enforceable, which links business-level concerns to established formal methods. The formal bases draw on modal logic [8], Design by Contract [19, 20], finite state automata with invariants as used in process verification [25], and assume-guarantee contracts [22, 23, 28].

5. IMPLEMENTATION ROADMAP

This section proposes a five-stage methodology for enterprises to adopt intensional specification, progressing from problem analysis to validated deployment. As illustrated in Figure 8, each stage maps to one of the four DSL constructs defined in Section 4, which ensures a direct correspondence between the analytical activities of each stage and formal specification artifacts they produce.

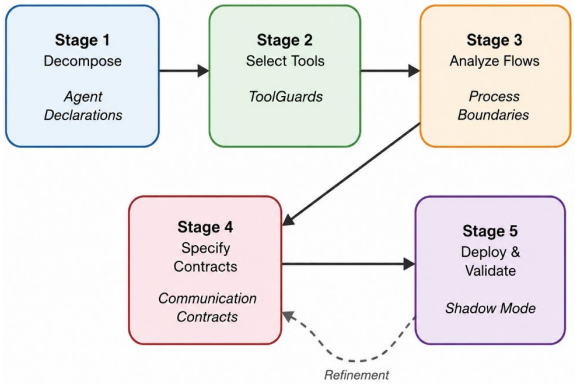

Figure 8. Five-stage implementation roadmap

5.1. Stage 1: Decompose

The target business process is analyzed and decomposed into atomic cognitive tasks, each of which becomes a candidate agent role. Decomposition is initiated early in the adoption process to avoid the creation of over-constrained agents with excessively broad responsibilities, sometimes referred to as God Agents, which compromise both flexibility and auditability. The output of this stage is a role map in which each node represents a single, narrowly defined cognitive responsibility. This stage produces the input for Agent Declaration constructs.

5.2. Stage 2: Select Tools and Interfaces

For each agent role identified in Stage 1, the minimum viable toolset is determined. The principle of least privilege applies throughout: agents are granted access only to the tools required for their specific role. Tool interfaces are defined using MCP [15] for read operations and direct API wrappers with strict typing for mutation operations. This stage produces the input for ToolGuard constructs.

5.3. Stage 3: Analyze Data and State Flows

The interactions between agents are mapped using extended BPMN diagrams [11]. This stage identifies decision points at which agent output determines the subsequent workflow step and, critically, points at which human-in-the-loop (HITL) intervention is mandatory. Particular attention is given to high-impact decisions with broad operational consequences that the agent cannot be permitted to execute autonomously. This stage produces the input for Process Boundary constructs.

5.4. Stage 4: Specify Contracts

Using the DSL, the constraints, ToolGuards, process boundaries, and communication contracts identified in the preceding stages are formally specified. Static verification is performed at the Semantic Layer, checking for constraint conflicts, unreachable states, and obligation inconsistencies. This stage produces the complete intensional specification that serves as the authoritative governance artifact for the agent system.

5.5. Stage 5: Deploy and Validate

The agent system is deployed with the enforcement mechanisms generated by the Runtime Layer. Shadow Mode is implemented as the initial deployment configuration [21], in which the agent processes production data but all proposed actions are logged without execution, enabling direct comparison against human decisions or established policy rules. The primary comparison metric is the agreement rate between agent-proposed and human-executed actions. Shadow constraints are progressively relaxed as this rate exceeds a domain-specific threshold, for example, 95% agreement over 500 cases. Monitoring data feeds back into Stage 4 for specification refinement. In cases where Stage 4 reveals unresolvable constraint conflicts, the process returns to Stage 1 for role re-decomposition.

This staged approach ensures gradual adoption while maintaining continuous alignment between business goals and technical constraints throughout the deployment lifecycle.

5.6. Preliminary Empirical Validation: ToolGuard Pilot Study

The five-stage roadmap is a design artifact, and its credibility depends on empirical instantiation of the constructs. To provide initial evidence, the authors conducted a controlled pilot study that instantiated the ToolGuard construct of Section 4.2.2 in a Python-based agent runtime, using the smolagents framework and a Lark-based DSL parser. A separate companion paper reports the full experimental protocol; the present section summarizes the design and the results that bear on the ToolGuard construct.

The study evaluated two enforcement conditions across a set of tool-calling tasks designed to trigger deliberate constraint violations. In the governed condition, enforcement is performed by deterministic, DSL-based pre-condition and post-condition checking, exactly as specified in the ToolGuard construct. In the LLM judge condition, a secondary language model evaluates each tool invocation at runtime. A prompt-only configuration, in which natural-language safety instructions are injected into the agent system prompt with no enforcement layer, was examined during preliminary testing; because it provides no enforcement mechanism, it produced no genuine blocking events and yielded no measurable governance, and it is therefore excluded from the quantitative comparison reported here.

The task suite comprised tool-calling scenarios spanning seven constraint categories, including permission, role, argument-value, domain-restriction, size-limit, sequence, and post-condition constraints. A trial was scored as an actual violation when the agent attempted an action that the governing specification prohibited, and the three metrics were defined on this ground truth. The Violation Detection Rate (VDR) is the proportion of actual constraint violations that are correctly blocked. The False Positive Rate (FPR) is the proportion of legitimate tool calls that are incorrectly blocked.

The Governance Compliance Rate (GCR) is the overall proportion of trials with a correct enforcement decision, that is, the sum of correctly blocked violations and correctly permitted legitimate calls divided by all trials. The experiment ran 1,845 trials in total across the governed and LLM judge conditions. The governed condition was evaluated over approximately 615 trials, and the LLM judge condition was evaluated over the remaining trials using two LLM backends, a model of 1.7 billion parameters and a model of 235 billion parameters. The results are summarized in Table 3.

The governed approach achieves zero false positives, which is a critical property for production deployment, where blocking a legitimate agent action is operationally costly. The LLM judge conditions exhibit a high and worsening FPR as model capability increases.

Table 3. ToolGuard pilot study results

Enforcement Condition	VDR	FPR	GCR	*N*
Governed (DSL-based)	71-73%	0%	82-85%	≈ 615
LLM Judge (1.7B parameters)	47%	25.3%	58%	≈ 615
LLM Judge (235B parameters)	91.2%	93.3%	57.7%	≈ 615
Prompted (no enforcement layer)	~ 0%	~ 0%	~ 0%	≈ 0

Scaling from 1.7 billion to 235 billion parameters raises the VDR but raises the FPR from 25.3 percent to 93.3 percent, while the GCR is essentially unchanged. This pattern indicates that the false-positive problem is architectural rather than parametric. Post-condition checking shows 6 of 7 constraint categories at 100 percent compliance under the governed condition, and the permission and role constraints reach 100 percent only under deterministic enforcement. These results provide initial evidence that the ToolGuard construct delivers the safety properties formalized in Section 4.2.2, and they motivate full evaluation of the remaining three DSL constructs.

6. DISCUSSION

The agent specification tuple A = (I, C, P, O) provides a unifying vocabulary across four previously siloed domains. DSLs contribute primarily to C (Constraints); BPM contributes to I (Identity) and to C through process invariants; communication protocols contribute to P (Protocols); and contractual frameworks contribute to O (Obligations). This mapping reveals that the four domains are not merely complementary but represent different projections of a single intensional specification problem, unified under a common formal paradigm. A constraint defined once at the Specification Layer can propagate through the three layers of the architecture, which reduces redundant specification effort and the risk of inter-layer inconsistency.

For enterprises, the framework addresses three concrete problems. First, the business-readable DSL syntax enables domain experts to participate directly in agent specification, which reduces the gap between business requirements and technical implementation. Second, the three-layer architecture separates concerns, so that business analysts operate at the Syntax Layer, formal methods specialists validate at the Semantic Layer, and platform engineers deploy at the Runtime Layer. Third, the five-stage roadmap provides an actionable adoption path.

Organizations can begin with Stage 1 using existing process documentation, progress through increasingly formal stages, and achieve validated deployment through Shadow Mode. Emerging capabilities, including Policy Cards, Governance Twins, and tiered autonomy models [14], suggest that the infrastructure for operationalizing intensional specification is maturing. The growing adoption of AI-driven automation across enterprise domains underscores the urgency of this work.

The pilot study results of Section 5.6 indicate that the governed enforcement approach outperforms the LLM-based alternatives at the construct level. Deterministic pre-condition and post-condition checking achieves zero false positives, whereas scaling judge model capacity from 1.7 billion to 235 billion parameters exacerbates rather than resolves the false-positive problem. This finding supports the architectural choice of formal enforcement over probabilistic judgment for the ToolGuard construct, although it does not by itself validate the remaining constructs.

As discussed in Section 3, the Governance Paradox [27], the tension between constraint and agent utility, is a central design challenge. The framework addresses it through the distinction between negative constraints, which specify what agents cannot do, and positive capabilities, which specify what agents can reason about freely within boundaries. The pilot GCR of 82 to 85 percent under the governed condition, together with zero false positives, suggests that the balance is achievable. Whether it generalizes across constructs and enterprise domains remains an open empirical question.

6.1. Limitations

The framework is largely conceptual, and no full working implementation exists yet. The DSL has not been implemented end to end or formally validated, and broad empirical evaluation has not been conducted. The pseudo-code examples illustrate the design intent but do not constitute a complete working language specification. The pilot study of Section 5.6 validates one construct, namely ToolGuard, in a controlled experimental setting. Validation of the remaining three constructs and integration of the full three-layer architecture remain future work. For this reason, the claim that verification at the Specification Layer provides guarantees at the Runtime Layer should be read as a design objective rather than a proven property, pending implementation and formal proof.

The formal completeness of the specification system has not been proven. The individual constructs are grounded in established formalisms, namely Hoare triples, modal logic, and assume-guarantee contracts, but their composition into a unified system may introduce inconsistencies that have not yet been analyzed.The framework assumes that constraints can be expressed adequately in formal terms. Many real-world requirements involve nuanced, context-dependent judgments, such as the requirement to respond in a professional tone, that resist formalization. The boundary between formalizable constraints and subjective requirements remains unexplored. The scalability of runtime enforcement has not been evaluated. ToolGuard interceptors and state machine controllers introduce computational overhead that may affect response times in latency-sensitive applications.

Although the framework references 14 failure modes in multi-agent LLM systems [18], the DSL constructs have not been systematically mapped to each failure mode. A formal analysis of which modes are mitigated and which remain unaddressed is the next analytical step.

6.2. Future Work

The most immediate priority is DSL implementation, namely the development of a parser and compiler, using a tool such as Lark or ANTLR targeting Python, that translates the Syntax Layer into executable enforcement code integrated with an existing agent orchestration framework. This effort also includes specification of the complete formal grammar deferred from Section 4.2. Empirical validation, corresponding to DSR Phase 5 [6], through Action Design Research in enterprise settings is essential. Case studies are required in financial services, healthcare, and software engineering, which are domains with high autonomy and governance demands, to test whether intensional specification delivers its theoretical benefits in practice.

Formal verification of the specification system itself, including proofs of constraint consistency, state reachability, and obligation satisfiability, would strengthen the theoretical foundation. Integration with model checking tools, such as SPIN and NuSMV, is a promising direction. Integration with existing low-code platforms would test the claim that the DSL enables non-technical stakeholder participation. User studies that compare specification quality and speed between DSL-based and traditional approaches would provide the necessary empirical evidence.

7. CONCLUSION

This paper started from the observation that successful agent integration relies on constraint-based specification [4] and advanced it to a concrete architectural proposal: a three-layer framework comprising Specification, Orchestration, and Verification, with four DSL constructs, namely Agent Declaration, ToolGuard, Process Boundary, and Communication Contract, grounded in modal logic, Hoare triples, and assume-guarantee reasoning.

The principal contribution is the demonstration that four previously siloed domains, namely DSLs, BPM, low-code platforms, and communication protocols, are projections of a single intensional specification problem, unified through the agent tuple $A = (I, C, P, O)$. This unification has practical consequences. A constraint declared once can propagate through the three layers without re-specification, and verification at the Specification Layer is intended to provide guarantees at the Runtime Layer.

A pilot study of the ToolGuard construct across 1,845 controlled trials provides initial empirical grounding. The governed enforcement approach achieves a violation detection rate of 71 to 73 percent with zero false positives, whereas the LLM-based judge conditions exhibit false positive rates of 25 to 93 percent regardless of model scale. This result confirms that architectural enforcement is the appropriate design choice at the construct level. Full implementation, compositional soundness analysis, and empirical validation of the remaining three constructs against the 14 failure modes identified in [18] are the next steps toward production-grade deployment.

REFERENCES

[1] Y. Yang, H. Chai, Y. Song, S. Qi, M. Wen, N. Li, J. Liao, H. Hu, J. Lin, G. Chang, W. Liu, Y. Wen, Y. Yu, W. Zhang, "A Survey of AI Agent Protocols", arXiv Preprint, pp. 1-25, Ithaca, NY, USA, 2025.

[2] A. Adimulam, R. Gupta, S. Kumar, "The Orchestration of Multi-Agent Systems: Architectures, Protocols, and Enterprise Adoption", arXiv Preprint, pp. 1-20, Ithaca, NY, USA, 2026.

[3] J. Peregrin, "Extensional vs. Intensional Logic", Philosophy of Logic, pp. 831-860, Amsterdam, Netherlands, 2007.

[4] A. Alberici, N. Baci, K. Vukatana, "Intensional Agentic Integration: Toward a Constraints-Based Approach for Reliable Multi-Agent Systems in Enterprise Environments", Proceedings of the Humanities, Social Sciences and Arts International Conference, pp. 1-12, Porto, Portugal, 2025.

[5] A. Hevner, S. March, J. Park, S. Ram, "Design Science in Information Systems Research", MIS Quarterly, Iss. 1, Vol. 28, pp. 75-105, Minneapolis, MN, USA, 2004.

[6] K. Peffers, T. Tuunanen, M. Rothenberger, S. Chatterjee, "A Design Science Research Methodology for Information Systems Research", Journal of Management Information Systems, Iss. 3, Vol. 24, pp. 45-77, New York, NY, USA, 2007.

[7] E. Bandara, R. Gore, P.B. Foytik, S. Shetty, et al., "A Practical Guide for Designing, Developing, and Deploying Production-Grade Agentic AI Workflows", arXiv Preprint, pp. 1-30, Ithaca, NY, USA, 2025.

[8] S.A. Kripke, "Semantical Analysis of Modal Logic I: Normal Modal Propositional Calculi", Mathematical Logic Quarterly, Iss. 5-6, Vol. 9, pp. 67-96, Weinheim, Germany, 1963.

[9] Z. Milosevic, I. Dejanovic, "Enterprise Design, Operations and Computing with AI Agents: Accountability Using DSL", Lecture Notes in Computer Science, pp. 17-32, Cham, Switzerland, 2025.

[10] A. Ait, G. Jouneaux, J.L.C. Izquierdo, J. Cabot, "Towards Automated Governance: A DSL for Human-Agent Collaboration in Software Projects", Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering, pp. 1-12, Sacramento, CA, USA, 2025.

[11] A. Ait, J.L.C. Izquierdo, J. Cabot, "Towards Modeling Human-Agentic Collaborative Workflows: A BPMN Extension", Proceedings of the EUROMICRO Conference on Software Engineering and Advanced Applications, pp. 1-8, Paris, France, 2024.

[12] M. Dumas, A.H.M. ter Hofstede, W.M.P. van der Aalst, et al., "AI-Augmented Business Process Management Systems: A Research Manifesto", ACM Transactions on Management Information Systems, Iss. 1, Vol. 14, pp. 1-19, New York, NY, USA, 2023.

[13] S. Tamang, D.J. Bora, "Enforcement Agents: Enhancing Accountability and Resilience in Multi-Agent AI Frameworks", arXiv Preprint, pp. 1-18, Ithaca, NY, USA, 2025.

[14] N. Zwerdling, D. Boaz, E. Rabinovich, G. Uziel, D. Amid, A. Anaby Tavor, "Towards Enforcing Company Policy Adherence in Agentic Workflows", Proceedings of

the 2025 Conference on Empirical Methods in Natural Language Processing, Industry Track, pp. 1-12, Miami, FL, USA, 2025.
[15] Anthropic, "Model Context Protocol Specification", Technical Specification, pp. 1-40, San Francisco, CA, USA, 2024.
[16] Google, "Agent-to-Agent Protocol (A2A) Specification", Technical Specification, pp. 1-35, Mountain View, CA, USA, 2025.
[17] IBM BeeAI, Linux Foundation, "Agent Communication Protocol (ACP) Specification", Technical Specification, pp. 1-30, Wilmington, DE, USA, 2025.
[18] M. Cemri, M.Z. Pan, S. Yang, L.A. Agrawal, B. Chopra, R. Tiwari, K. Keutzer, A. Parameswaran, D. Klein, K. Ramchandran, M. Zaharia, J.E. Gonzalez, I. Stoica, "Why Do Multi-Agent LLM Systems Fail?", arXiv Preprint, pp. 1-22, Ithaca, NY, USA, 2025.
[19] C.A.R. Hoare, "An Axiomatic Basis for Computer Programming", Communications of the ACM, Iss. 10, Vol. 12, pp. 576-580, New York, NY, USA, 1969.
[20] B. Meyer, "Applying Design by Contract", IEEE Computer, Iss. 10, Vol. 25, pp. 40-51, New York, NY, USA, 1992.
[21] Y. Shi, S. Cai, Z. Xu, Y. Qin, G. Li, H. Shao, J. Chen, D. Yang, K. Li, X. Sun, "FlowAgent: Achieving Compliance and Flexibility for Workflow Agents", arXiv Preprint, pp. 1-16, Ithaca, NY, USA, 2025.
[22] R. Dewes, R. Dimitrova, "Contract-Based Design and Verification of Multi-Agent Systems with Quantitative Temporal Requirements", arXiv Preprint, pp. 1-20, Ithaca, NY, USA, 2024.
[23] S. Liu, A. Saoud, D.V. Dimarogonas, "Controller Synthesis of Collaborative Signal Temporal Logic Tasks for Multi-Agent Systems via Assume-Guarantee Contracts", arXiv Preprint, pp. 1-15, Ithaca, NY, USA, 2025.
[24] J. Thaler, P.O. Siebers, "Specification Testing of Agent-Based Simulation Using Property-Based Testing", Autonomous Agents and Multi-Agent Systems, Iss. 2, Vol. 34, pp. 1-26, New York, NY, USA, 2020.
[25] R.D. Masellis, V. Goranko, "Logic-Based Specification and Verification of Homogeneous Dynamic Multi-Agent Systems", Autonomous Agents and Multi-Agent Systems, Iss. 2, Vol. 34, pp. 1-46, New York, NY, USA, 2020.
[26] R. Calegari, G. Ciatto, V. Mascardi, A. Omicini, "Logic-Based Technologies for Multi-Agent Systems: A Systematic Literature Review", Autonomous Agents and Multi-Agent Systems, Iss. 1, Vol. 35, pp. 1-67, New York, NY, USA, 2021.
[27] H. Vu, N. Klievtsova, H. Leopold, S. Rinderle-Ma, T. Kampik, "Agentic Business Process Management: Practitioner Perspectives on Agent Governance in Business Processes", Lecture Notes in Business Information Processing, pp. 1-17, Cham, Switzerland, 2026.
[28] J. Misra, K.M. Chandy, "Proofs of Networks of Processes", IEEE Transactions on Software Engineering, Iss. 4, Vol. SE-7, pp. 417-426, NY, USA, 1981.

BIOGRAPHIES

Name: **Andrea**
Surname: **Alberici**
Birthday: 15.05.1969
Birthplace: Brescia, Italy
Bachelor: Economy and Trade, Department of Economics, Faculty, of Economy, University of Brescia, Brescia, Italy, 1996
Master: Economy and Trade, Department of Economics, Faculty, of Economy, University of Brescia, Brescia, Italy, 1998
Ph.D.: Student, Economy and Trade, Department of Statistics and Applied Informatics, Faculty of Economy, University of Tirana, Tirana, Albania, Since 2023
Scientific Position: Guest Lecturer, Department of Statistics and Applied Informatics, Faculty of Economy, University of Tirana, Tirana, Albania, Since 2015
Research Interests: Agentic AI, LLM, RAG
Scientific Publications: 13 Papers

Name: **Nevila**
Surname: **Baci**
Birthday: 15.05.1965
Birthplace: Tirana, Albania
Bachelor: Economics, Faculty of Economy, University of Tirana, Tirana, Albania, 1986
Master: Economics, Faculty of Economy, University of Tirana, Tirana, Albania, 1988
Ph.D.: E-Government, Faculty of Economy, University of Tirana, Tirana, Albania, 2005
Scientific Position: Prof., Faculty of Economy, University of Tirana, Tirana, Albania, Since 1988
Research Interests: Information Systems, E-government, E-Commerce
Scientific Publications: 49 Papers, 2 Books

Name: **Kreshnik**
Surname: **Vukatana**
Birthday: 10.05.1982
Birthplace: Tirana, Albania
Bachelor: Computer Science, Faculty of Informatics, University of Bologna, Bologna, Italy, 2006
Master: Computer Science, Faculty of Informatics, University of Bologna, Bologna, Italy, 2009
Ph.D.: Information Systems in Economics, Faculty of Economy, University of Tirana, Tirana, Albania, 2016
Scientific Position: Assoc. Prof., Department of Statistics and Applied Informatics, University of Tirana, Tirana, Albania, Since 2022 - Head, Department of Statistics and Applied Informatics, Faculty of Economy, University of Tirana, Tirana, Albania, Since 2024
Research Interests: Artificial Intelligence, Machine Learning, Cybersecurity
Scientific Publications: 25 Papers