ALI: A Bounded Architecture for Controllable Autonomous Agents

Wolfgang Stegemann
Independent Researcher — ORCID: 0009-0000-1346-1170

Abstract

Autonomous agents operating over extended periods face requirements that are qualitatively different from those of isolated task execution. A system that must continuously preserve operational integrity while generating and evaluating behaviour cannot be organised around a single reasoning process without sacrificing transparency and controllability. We present Artificial Local Intelligence (ALI), a reference architecture that decomposes autonomous operation into five explicitly separated responsibilities: a Causal Core that evaluates operational viability before planning begins, an Ego that generates behavioural proposals without evaluating them, a Super-Ego that evaluates proposals without generating them, a Memory that preserves every decision as a complete and immutable operational episode, and a Runtime that coordinates the cycle without participating in reasoning. We provide a complete architectural specification, a compliance criteria set of 44 verifiable SHALL requirements, and an open-source reference implementation in Python that requires no third-party dependencies. The implementation is verified by 72 automated tests and includes a plugin system that allows any architectural component to be replaced through configuration. A comparison against BDI architectures, SOAR, LangGraph, and the Microsoft Agent Framework shows that no existing production framework provides an independent architectural component for operational viability monitoring or enforces the separation of behavioural generation from normative evaluation at the architectural level.

Keywords: autonomous agents, software architecture, separation of concerns, normative evaluation, operational viability, controllability, reference architecture

1. Introduction

An autonomous system that must operate continuously faces requirements that do not arise in systems designed for isolated task execution. A task-execution system succeeds when it produces the correct output for a given input. An autonomous system must, in addition, preserve the conditions that make future operation possible, remain understandable to those who supervise it, and account for every decision it has made. These requirements do not follow from insufficient reasoning capability. They follow from the organisational structure of the system itself.

Contemporary approaches to autonomous AI have concentrated primarily on improving the reasoning components: larger language models, more powerful planners, richer retrieval systems. The resulting systems are often more capable. They are not necessarily better organised. A more capable reasoning engine embedded in a monolithic decision process

does not become more transparent or more controllable by virtue of its capability. It becomes more capable of producing decisions whose provenance is increasingly difficult to reconstruct.

This paper addresses a different question. Not: how can autonomous agents reason better? But: how should autonomous agents be organised so that their reasoning remains permanently understandable and their behaviour remains permanently controllable?

The answer we propose is architectural. We present Artificial Local Intelligence (ALI), a reference architecture that separates autonomous operation into five explicitly distinct organisational responsibilities. A Causal Core evaluates whether the system remains capable of operating before behavioural planning begins. An Ego generates candidate behaviours without evaluating their acceptability. A Super-Ego evaluates proposed behaviours without generating them. A Memory preserves every decision as a complete and immutable operational episode. A Runtime coordinates the interaction of these components without participating in any of their reasoning.

We make three contributions. First, a complete architectural specification including seven design principles, component responsibilities, interface definitions, an architectural data model, and a formal compliance specification of 44 verifiable SHALL requirements. Second, an open-source reference implementation in Python verified by 72 automated tests, with a plugin system demonstrating that individual components can be replaced without touching the surrounding architecture. Third, a compliance evaluation showing that all 44 requirements are satisfied, and a comparison against four established frameworks demonstrating that the architectural properties ALI provides are absent from existing production systems.

The paper is organised as follows. Section 2 motivates the architectural approach. Section 3 presents the ALI architecture. Section 4 describes the reference implementation and compliance evaluation. Section 5 compares ALI against existing frameworks. Section 6 discusses limitations and future work. Section 7 concludes. The full 44-requirement compliance matrix appears in the Appendix.

2. Background

The deployment of autonomous AI agents in enterprise environments has accelerated substantially since 2024. Industry forecasts project that 40% of enterprise software applications will integrate task-specific AI agents by the end of 2026. The dominant implementation pattern rests on large language models extended with tool access, memory systems, and orchestration frameworks such as LangGraph or the Model Context Protocol. These systems are capable of multi-step reasoning and autonomous task execution across complex workflows.

Deployment at scale has exposed a structural limitation that engineering practices have not resolved. Gartner's forecast that over 40% of agentic AI projects will be cancelled by end of 2027 does not primarily reflect model inadequacy. Post-deployment analysis consistently identifies governance failure as the dominant cause: agents operating beyond intended

scope, decisions that cannot be audited, and behavioural drift that no monitoring layer reliably contains. A 2025 study found that in multi-agent systems, a single compromised agent propagated its effect to 87% of downstream decisions within four hours, faster than conventional incident response.

The prevailing response to these failures has been to add governance as a layer above existing architectures: guardrails, delegation tokens, approval gates, and audit logs imposed on systems whose underlying behaviour remains probabilistic and unbounded. This approach is structurally insufficient. A system that is capable of operating anywhere and then restricted by policy can drift whenever policy enforcement fails, is ambiguously specified, or encounters a context its designers did not anticipate. Governance-as-layer converts a structural problem into an operational one and transfers the cost of failure to runtime monitoring rather than eliminating it at the architecture level.

The concept of bounded autonomy has emerged in practitioner literature as the required design target. The operationalisation of this concept, however, remains incomplete. Existing frameworks specify bounded autonomy as a configuration of an otherwise general-purpose system: scoped permissions, defined tool access, approval thresholds. The boundary is a policy, not a structural constraint. An agent that cannot represent anything outside its task domain is architecturally different from one that can but is instructed not to.

This paper presents ALI (Artificial Local Intelligence), an agent architecture in which task-domain boundaries are structural rather than configurable. ALI is not a restriction applied to a general-purpose system. It is a purpose-built architecture in which the agent's representational and operational space is defined at design time and cannot be exceeded at runtime.

3. The ALI Architecture

3.1 Design Principles

The ALI architecture is derived from six engineering principles that are independent of any particular computational technology. Separation of responsibilities: the different activities required for autonomous operation — observing the environment, assessing operational condition, generating behaviour, evaluating behaviour, preserving history, and coordinating execution — answer different questions and should be assigned to independent components. Local autonomy: every ALI instance operates within a bounded environment that defines its available resources, tools, and objectives; intelligence is therefore always local rather than claimed universally. Continuous operational viability: the conditions that enable future operation must be monitored permanently, not treated as exceptional concerns. Independent normative regulation: generating behaviour and evaluating behaviour are distinct engineering activities that must not be performed by the same component. Historical continuity: every operational episode, including the observation, the viability state, the behavioural proposal, the normative evaluation, and the execution result,

must be preserved as a complete and immutable record. Architectural stability: learning modifies operational knowledge, never organisational responsibilities.

3.2 Architectural Components

The architecture consists of five primary components and one interface layer. The Causal Core continuously evaluates the operational condition of the system. Its sole responsibility is to determine whether the system remains capable of acting safely and effectively under current conditions. It does not generate behaviour, evaluate norms, or determine priorities. Its output is a structured Viability State that becomes available to the remaining components during the current cycle.

The Ego receives the current observation, the Viability State, the active operational objectives, and relevant historical experience retrieved from Memory. It generates one or more behavioural proposals. It performs no normative evaluation. The architecture places no restrictions on the computational methods the Ego employs; a symbolic planner, a language model, or any future reasoning technology may serve as the Ego's internal implementation.

The Super-Ego performs normative evaluation. It receives proposals generated by the Ego and evaluates them against permanent constraints and adaptive norms. It never generates proposals. Its output is a Norm Evaluation specifying whether a proposal is approved, rejected, or requires revision. If all proposals are rejected, the Runtime requests additional proposals from the Ego, implementing iterative behavioural refinement while preserving the independence of generation and evaluation.

Memory preserves every completed operational cycle as an immutable Event. An Event contains the originating Observation, the Viability State, the Behaviour Proposal, the Norm Evaluation, the Execution Result, and all metadata required for complete historical reconstruction. Memory never interprets the information it stores; retrieval is performed in response to explicit requests from other components. The accumulation of Events constitutes the developmental history of the system.

The Runtime coordinates the interaction of all remaining components without participating in their decision-making. It controls the operational cycle, invokes interfaces in the prescribed sequence, records completed Events, and supervises the transition between operational states. The Runtime provides no behavioural intelligence of its own. The Environment Interface isolates the architecture from its operational domain, translating domain-specific interactions into the common architectural representation.

3.3 The Operational Cycle

Every autonomous decision follows the same invariant sequence: (1) the Environment Interface produces an Observation; (2) the Causal Core evaluates operational viability and produces a Viability State; (3) the Ego generates one or more Behaviour Proposals using the Observation, Viability State, current objectives, and relevant history; (4) the Super-Ego evaluates each proposal and returns a Norm Evaluation; (5) the Runtime executes the first approved proposal through the Environment Interface; (6) the Runtime assembles all

architectural objects into an Event and stores it in Memory; (7) each component performs learning from the stored Event; (8) the next cycle begins. Learning is coordinated only after immutable Event storage.

3.4 Architectural Stability

A central principle of ALI is the distinction between development and structure. Development concerns the information processed by the architecture — behavioural heuristics, normative weights, operational models. Structure concerns the organisation through which this information is processed — the five components and their responsibilities. Only the first may change during operation. The second remains invariant. This distinction ensures that an ALI system can operate for extended periods, accumulate experience, and exhibit sophisticated behaviour without sacrificing the transparency and reconstructability that make it controllable. Learning changes how the architecture behaves; it never changes what the architecture is.

4. Reference Implementation and Compliance

4.1 Implementation Overview

The reference implementation (v0.4) is written in Python and requires no third-party dependencies. Every architectural component is represented by exactly one Python module, and every module corresponds to exactly one organisational responsibility. Abstract base classes in interfaces.py define the public API of each component through typed method signatures; a component that does not implement all abstract methods is rejected at class definition time by the Python interpreter. All architectural data objects (Observation, ViabilityState, BehaviourProposal, NormEvaluation, ExecutionResult, Event) are implemented as frozen dataclasses, enforcing immutability at the language level.
The reference domain is a local document workspace. The system proposes safe filename sanitisation and file classification by extension, evaluates proposals normatively, executes approved reversible actions, stores complete Events in SQLite, and supports rollback of every executed action. Defaults to dry-run mode; real changes require explicit activation.

4.2 Plugin System

The ComponentFactory in ali/plugins.py resolves component class paths from configuration and calls each class's from_config(config, params) classmethod. Any architectural component can be replaced by a third-party implementation by adding its dotted class path to the "components" section of ali_config.json. The surrounding architecture requires no modification. The LLMEgo example in examples/llm_ego.py demonstrates a complete plugin that delegates behavioural generation to a language model; in stub mode it constructs the full structured prompt and returns a deterministic proposal, enabling the plugin to be tested without API credentials. Invalid class paths and wrong interface types raise ImportError and

TypeError respectively at Runtime startup, not at the first cycle, ensuring configuration errors are detected immediately.

4.3 Compliance Evaluation

The compliance specification in Appendix H of the ALI specification defines 44 SHALL requirements across six groups: component existence, negative responsibility constraints (shall not), positive responsibility constraints (shall), runtime sequence, interface requirements, Event requirements, learning requirements, and architectural stability. Table 1 shows the summary by evidence type; the full compliance matrix appears in Appendix A.

Table 1. Compliance evaluation summary.

Evidence type	Requirements	Result	Notes
RUNTIME	22	22 PASS	Runtime behaviour tests
STATIC	13	13 PASS	Class structure inspection
SEQUENCE	6	6 PASS	Call-order recording
STRUCTURAL	3	3 PASS	Python type system
Total	44	44 PASS	72 automated tests total

The test suite comprises 72 tests across five modules: test_architecture.py (9 tests), test_runtime.py (7 tests), test_plugins.py (15 tests), and test_compliance.py (41 tests). Three structural requirements are enforced by the Python interpreter through frozen dataclasses and abstract base class mechanisms; they are documented separately and verified by explicit runtime tests where possible.

5. Comparison with Existing Frameworks

We compare ALI against four established families of agent architecture: classical BDI systems, SOAR, LangGraph, and the Microsoft Agent Framework. The comparison is structured around seven criteria corresponding to the architectural responsibilities ALI separates.

5.1 BDI Architectures

The Belief-Desire-Intention model separates an agent's informational state (beliefs), motivational state (desires), and committed courses of action (intentions). This is a principled separation of concerns, and ALI inherits this engineering tradition. The axis of separation differs, however. BDI separates world model from motivational state from execution commitment. ALI separates operational viability assessment from behavioural generation from normative evaluation from historical continuity from execution coordination. These decompositions are orthogonal. In most BDI implementations, plan selection combines what ALI identifies as two distinct responsibilities — generating candidate behaviours and evaluating their normative acceptability — because plans that violate norms are simply absent from the plan library. ALI separates these structurally: the

Ego can generate any proposal, and the Super-Ego evaluates it independently. Operational viability monitoring is not a standard BDI component, and historical continuity in BDI is represented through the belief base and intention stacks rather than through structured operational episodes.

5.2 SOAR

SOAR provides the most sophisticated memory architecture among the systems examined, with distinct working memory, long-term procedural memory, semantic memory, and episodic memory. SOAR's episodic memory records working memory snapshots at operator application boundaries. ALI's Event records the full decision provenance including the normative reasoning that permitted or blocked execution. The difference is one of composition: SOAR episodes support cued retrieval for future problem-solving; ALI Events support accountability and developmental reconstruction. SOAR's learning mechanism (chunking) generalises operator sequences into new production rules, which modifies the procedural knowledge base in a way that blurs ALI's distinction between knowledge adaptation and responsibility modification. SOAR does not provide independent normative evaluation or operational viability monitoring.

5.3 LangGraph

LangGraph reached General Availability in May 2025 and is deployed at production scale at hundreds of organisations. It organises agent behaviour as a directed graph in which nodes execute actions and edges define control flow. Operational viability monitoring does not exist as an architectural component; nodes handle their own error conditions through exception mechanisms. Normative evaluation is not architecturally separated from behavioural generation; the same language model node typically performs both. LangGraph's checkpoint mechanism records state snapshots, not structured operational episodes. The framework supports over 35 model backends (high technology independence) and interrupt mechanisms for human oversight.

5.4 Microsoft Agent Framework

The Microsoft Agent Framework, released in October 2025 through consolidation of AutoGen and Semantic Kernel, provides multi-agent conversation patterns, comprehensive conversation logging, and safety guardrails. Safety is explicitly described as a layer with measurable performance overhead added to the system, not an architectural property. Conversation logs record agent exchanges, not structured operational episodes with typed architectural objects. Operational viability monitoring is not present as an architectural component. The framework provides strong technology independence through MCP and A2A protocol integration.

5.5 Comparison Summary

Table 2 summarises the comparison. Cells marked "Partial" indicate that the property exists in a non-architectural form, either as a convention or an external addition.

Table 2. Architectural comparison across seven criteria.

Criterion	BDI	SOAR	LangGraph	MS Agent Fwk	ALI
Independent viability monitoring	No	No	No	No	Yes (Causal Core)
Normative evaluation separate from generation	Partial	No	No	No (layer)	Yes (Super-Ego)
Complete operational episode as first-class object	No	Partial	No	No	Yes (Event)
Architectural stability under learning	No constraint	No constraint	No constraint	No constraint	Yes (principle)
Technology independence	Low	Low	High	High	High (from_config)
Controllability through architecture	Partial	Partial	Partial	Partial	Yes
Component replacement through configuration	No	No	No	Partial	Yes

Three distinctions appear consistently. No production framework provides an independent architectural component for operational viability monitoring (the Causal Core). No production framework enforces the separation of behavioural generation from normative evaluation at the architectural level (the Super-Ego). No production framework stores the complete causal chain of a decision — observation, viability, proposal, normative evaluation, execution result — as a single structured first-class object (the Event).

6. Discussion

ALI is designed for systems that operate autonomously within a bounded operational domain over extended periods. The word "local" reflects this deliberate constraint: intelligence is defined relative to a specific environment rather than claimed universally. A

system that operates within a well-defined domain can be validated against that domain; a system that claims universal applicability cannot be validated against anything.

The reference implementation's Ego is rule-based and minimal. It identifies files requiring name sanitisation and classification by extension. This is sufficient to demonstrate the architecture but does not represent the sophistication expected in a production deployment. A production Ego would employ a language model or a symbolic planner, as demonstrated structurally by the LLMEgo plugin. The architecture constrains organisational responsibilities, not reasoning technology.

The adaptive learning state — rule success counters in the Ego and adaptive norm weights in the Super-Ego — is held in process memory and resets on restart. A production implementation requires persistent normative state stored across sessions. This is the most immediate engineering gap between the current reference implementation and a production deployment.

This paper does not claim that ALI agents produce better outcomes than agents built on BDI, SOAR, or LangGraph. That claim would require empirical evaluation across multiple operational domains and over extended periods. What this paper claims is more limited and more precise: the architectural separation ALI proposes is verifiable, the reference implementation satisfies all 44 compliance requirements, and this separation is currently absent from existing production frameworks at the architectural level. Whether the separation produces better long-term outcomes is an empirical question that the ALI architecture is designed to make answerable.

Multi-agent operation is not addressed. ALI specifies one autonomous system operating within one bounded environment. Cooperation between ALI instances through Environment Interface connections is possible, but the protocols and architectural principles governing such cooperation are not defined. This is the most significant open problem in the research programme.

The paper claims that ALI provides controllability through architecture rather than through external safety mechanisms. This claim is structural: every executed action has passed through an independent Super-Ego, every decision is preserved as a reconstructable Event, and every component has an explicit architectural owner. This means that when behaviour is incorrect, its source is identifiable and intervention is possible through ordinary architectural interfaces. The architecture provides the instruments for control; the policies that govern their use are defined by those who deploy and supervise the system.

7. Conclusion

The enterprise deployment of autonomous AI agents has produced a governance crisis that current architectural approaches cannot resolve. The dominant response, adding policy layers to general-purpose systems, addresses the symptom rather than the cause. An agent that can represent and operate across an unbounded space remains unbounded regardless of the policies imposed on it. Governance-as-layer is not a solution to the boundary problem. It is a permanent operational cost imposed by the absence of one.

ALI addresses the boundary problem at the level where it originates: architecture. The task-domain boundary in ALI is not a configuration parameter. It is a structural property of the representational and operational space the agent inhabits. An agent that cannot represent states outside its domain cannot act on them, drift toward them, or be manipulated into crossing into them. This eliminates the class of failures that governance layers attempt to contain after the fact.

The practitioner literature has arrived at bounded autonomy as the required design target. The gap between that requirement and its implementation remains open. Existing frameworks operationalise bounded autonomy as scoped permissions on general-purpose systems. ALI operationalises it as a purpose-built architecture in which the scope is not a permission but a structural fact about the agent.

The implications extend beyond enterprise deployment. The consciousness debate in AI research has proceeded largely on the assumption that the path to useful autonomous agents runs through general intelligence. The ALI architecture demonstrates that this assumption is not necessary. An agent that is rational, autonomous within a defined domain, auditable at every decision point, and structurally incapable of operating outside its task space does not require general intelligence to be useful. It requires precise specification. The engineering problem is not how to build a mind. It is how to build a tool that does exactly what it is specified to do and nothing else.

That problem is solved at the architecture level or it is not solved.

References

- Bratman, M. (1987). Intention, Plans, and Practical Reason. Harvard University Press.
- Laird, J., Newell, A., Rosenbloom, P. (1987). Soar: An architecture for general intelligence. Artificial Intelligence, 33(1), 1–64.
- LangGraph General Availability (May 2025). https://blog.langchain.dev/langgraph-ga
- Microsoft Agent Framework (October 2025). Consolidation of AutoGen and Semantic Kernel. https://azure.microsoft.com/en-us/products/ai-foundry
- Rao, A., Georgeff, M. (1991). Modeling rational agents within a BDI architecture. Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning, 473–484.
- Stegemann, W. (2026). Artificial Local Intelligence: Architecture and Reference Implementation. Zenodo. https://doi.org/10.5281/zenodo.21631699
- Winikoff, M., Padgham, L. (2004). Developing Intelligent Agent Systems. Wiley.
- Yao, S. et al. (2022). ReAct: Synergizing Reasoning and Acting in Language Models. arXiv:2210.03629.

- Alechina, N. et al. (2024). Integrating Machine Learning into Belief-Desire-Intention Agents. arXiv:2510.20641.
- https://github.com/drwolfgangstegemann-sudo/ali-reference-implementation
- and DOI: 10.5281/zenodo.21632050
- Gartner (2025). Gartner Predicts Over 40% of Agentic AI Projects Will Be Canceled by End of 2027. https://www.gartner.com/en/newsroom/press-releases/2025-06-25-gartner-predicts-over-40-percent-of-agentic-ai-projects-will-be-canceled-by-end-of-2027
- Gartner (2026). Enterprise Agentic AI Governance Report. February 2026.
- Galileo AI (2025). Multi-Agent Failure Propagation Study. December 2025.
- Stegemann, W. (2026). Four-Dimensional Decision Stability: Integrating Dimensional Logic into the ALI Architecture. Zenodo. https://doi.org/10.5281/zenodo.20407218
- Stegemann, W. (2026). Why Artificial Consciousness Is Impossible. Zenodo. https://doi.org/10.5281/zenodo.21829358
- Stegemann, W. (2026). Beyond AGI: The Case for ALI. Zenodo. https://doi.org/10.5281/zenodo.21916109

Appendix A. Full Compliance Matrix

Table A1. All 44 SHALL requirements mapped to test evidence.

Req. ID	SHALL statement (condensed)	Evidence type	Test method	Result
ALI-REQ-001	Exactly one Runtime	RUNTIME	test_REQ001_runtime_exists_and_is_single	PASS
ALI-REQ-002	Runtime shall not generate behaviour	STATIC	test_REQ002_runtime_has_no_generate_method	PASS
ALI-REQ-003	Runtime shall not perform normative evaluation	STATIC	test_REQ003_runtime_has_no_normative_evaluate_method	PASS

Req. ID	SHALL statement (condensed)	Evidence type	Test method	Result
ALI-REQ-004	Runtime shall not assess operational viability	STATIC	test_REQ004_runtime_has_no_viability_assess_method	PASS
ALI-REQ-005	Exactly one Causal Core	RUNTIME	test_REQ005_causal_core_exists	PASS
ALI-REQ-006	Causal Core shall evaluate operational viability	RUNTIME	test_REQ006_causal_core_produces_viability_state	PASS
ALI-REQ-007	Causal Core shall not generate proposals	STATIC	test_REQ007_causal_core_has_no_generate_method	PASS
ALI-REQ-008	Causal Core shall not perform execution	STATIC	test_REQ008_causal_core_has_no_execute_method	PASS
ALI-REQ-009	Exactly one Ego	RUNTIME	test_REQ009_ego_exists	PASS
ALI-REQ-010	Ego shall generate one or more Behaviour Proposals	RUNTIME	test_REQ010_ego_generates_proposals	PASS
ALI-REQ-011	Ego shall not evaluate proposals	STATIC	test_REQ011_ego_has_no_evaluate_method	PASS
ALI-REQ-012	Ego shall not execute behaviour	STATIC	test_REQ012_ego_has_no_execute_method	PASS
ALI-REQ-013	Exactly one Super-Ego	RUNTIME	test_REQ013_super_ego_exists	PASS

Req. ID	SHALL statement (condensed)	Evidence type	Test method	Result
ALI-REQ-014	Super-Ego shall evaluate independently of Ego	RUNTIME	test_REQ014_super_ego_evaluates_independently	PASS
ALI-REQ-015	Super-Ego shall not generate proposals	STATIC	test_REQ015_super_ego_has_no_generate_method	PASS
ALI-REQ-016	Super-Ego shall not execute behaviour	STATIC	test_REQ016_super_ego_has_no_execute_method	PASS
ALI-REQ-017	Exactly one Memory component	RUNTIME	test_REQ017_memory_exists	PASS
ALI-REQ-018	Memory shall preserve every completed Event	RUNTIME	test_REQ018_memory_preserves_every_event	PASS
ALI-REQ-019	Memory shall not modify stored Events	STRUCTURAL	Python frozen=True on Event dataclass; see REQ-034	PASS
ALI-REQ-020	Memory shall support historical retrieval	RUNTIME	test_REQ020_memory_supports_historical_retrieval	PASS
ALI-REQ-021	Every cycle shall begin with an Observation	SEQUENCE	test_REQ021_cycle_begins_with_observation	PASS
ALI-REQ-022	Viability shall be evaluated before planning	SEQUENCE	test_REQ022_viability_before_planning	PASS
ALI-REQ-023	Planning shall precede	SEQUENCE	test_REQ023_planning_before_normative_evaluation	PASS

Req. ID	SHALL statement (condensed)	Evidence type	Test method	Result
	normative evaluation			
ALI-REQ-024	Normative approval shall precede execution	SEQUENCE	test_REQ024_normative_approval_precedes_storage	PASS
ALI-REQ-025	Execution shall precede Event creation	STRUCTURAL	Fixed source order in Runtime.run()	PASS
ALI-REQ-026	Event storage shall precede learning	SEQUENCE	test_REQ026_storage_precedes_learning	PASS
ALI-REQ-027	Runtime shall coordinate sequence deterministically	SEQUENCE	test_REQ027_sequence_is_deterministic	PASS
ALI-REQ-028	Components shall communicate through defined interfaces	STATIC	test_REQ028_components_communicate_through_interfaces	PASS
ALI-REQ-029	Components shall not access internal state directly	STATIC	test_REQ029_components_do_not_access_internal_state_directly	PASS
ALI-REQ-030	Every interface shall exchange architectural objects	STRUCTURAL	Python type annotations on all public methods	PASS
ALI-REQ-031	Every architectural object shall	RUNTIME	test_REQ031_every_architectural_object_has_unique_id	PASS

Req. ID	SHALL statement (condensed)	Evidence type	Test method	Result
	have a unique identifier			
ALI-REQ-032	Exactly one Event per proposal evaluation	RUNTIME	test_REQ032_one_event_per_proposal_evaluation	PASS
ALI-REQ-033	Every Event shall contain all six required objects	RUNTIME	test_REQ033_event_contains_all_required_architectural_objects	PASS
ALI-REQ-034	Events shall remain immutable after storage	RUNTIME	test_REQ034_events_are_immutable_after_storage	PASS
ALI-REQ-035	Events shall remain permanently retrievable	RUNTIME	test_REQ035_events_are_permanently_retrievable	PASS
ALI-REQ-036	Behavioural learning shall occur exclusively within Ego	RUNTIME	test_REQ036_behavioural_learning_in_ego_only	PASS
ALI-REQ-037	Normative learning shall occur exclusively within Super-Ego	RUNTIME	test_REQ037_normative_learning_in_super_ego_only	PASS
ALI-REQ-038	Operational learning shall occur exclusively within Causal Core	RUNTIME	test_REQ038_operational_learning_in_causal_core_only	PASS

Req. ID	SHALL statement (condensed)	Evidence type	Test method	Result
ALI-REQ-039	Memory shall not perform autonomous interpretation	STATIC	test_REQ039_memory_has_no_learn_or_interpret_method	PASS
ALI-REQ-040	Learning shall not modify architectural responsibilities	RUNTIME	test_REQ040_learning_does_not_modify_architectural_responsibilities	PASS
ALI-REQ-041	Implementation shall preserve separation of responsibilities	STATIC+RUNTIME	test_REQ041_separation_of_responsibilities_is_preserved	PASS
ALI-REQ-042	Components shall not exchange responsibilities during operation	RUNTIME	test_REQ042_components_do_not_exchange_responsibilities_during_operation	PASS
ALI-REQ-043	Architectural self-modification shall not occur	RUNTIME	test_REQ043_no_architectural_self_modification	PASS
ALI-REQ-044	Only operational knowledge shall change during learning	RUNTIME	test_REQ044_only_operational_knowledge_changes_during_learning	PASS