ANNOTATION_BLOCK_START

{ "artifact_annotation_header": { "artifact_id_of_host": "adr_os_001_md", "g_annotation_created": 3, "version_tag_of_host_at_annotation": "1.2.0" }, "payload": { "description": "Retrofitted to comply with ADR-OS-032: Canonical Models and Frameworks Registry & Enforcement.", "artifact_type": "DOCUMENTATION", "purpose_statement": "To ensure framework compliance and improve architectural decision clarity.", "authors_and_contributors": [{ "g_contribution": 3, "identifier": "Hybrid_AI_OS" }, { "g_contribution": 4, "identifier": "Framework_Compliance_Retrofit" }], "internal_dependencies": ["adr_os_template_md", "adr_os_032_md"], "linked_issue_ids": [] } }

ANNOTATION_BLOCK_END

ADR-OS-001: Core Operational Loop & Phasing

- **Status**: Proposed
- **Date**: 2025-06-06
- **Deciders**: [List of decision-makers]
- **Reviewed By**: [List of reviewers]

Context

The Hybrid_AI_OS requires a predictable, structured, and traceable process for managing complex projects from user request to completion. An ad-hoc or purely reactive model would lead to inconsistencies, poor traceability, and difficulty in coordinating autonomous agents. We need a defined operational lifecycle that provides clear states, transitions, and objectives for the OS and its agents at every step.

Assumptions

- [ ] The OS always persists `state.txt.ph` atomically between phases.
- [ ] Agent health checks succeed before phase transitions.
- [ ] All agents have access to a consistent view of the current phase state.
- [ ] Phase transitions are triggered only after all required objectives of the current phase are met.
- [ ] The state machine governing phase transitions is free of deadlocks and livelocks.
- [ ] Human intervention points are clearly defined and respected by the OS.
- [ ] All artifacts required for a phase transition are available and valid at the time of transition.
- [ ] Distributed traceability (e.g., `trace_id`) is reliably propagated across all phase transitions and agent actions.
- [ ] The OS can recover gracefully from partial failures during phase transitions.
- [ ] All compliance requirements from referenced ADRs (e.g., ADR-OS-023, ADR-OS-024, etc.) are up-to-date and enforced.

This section was expanded in response to issue_assumptions.txt to surface implicit assumptions and improve framework compliance.

Frameworks/Models Applied

This ADR applies the following canonical models and frameworks (per ADR-OS-032):

Theory of Constraints (ToC) v1.0

- **Compliance Proof:** The five-phase loop explicitly identifies bottlenecks at phase transitions, enabling systematic identification and resolution of constraints in the workflow.
- **Self-Critique:** Requires robust monitoring to accurately identify bottlenecks, which adds overhead.

KISS (Keep It Simple, Stupid) v1.0

- **Compliance Proof:** The operational loop uses exactly five phases with clear, single-purpose objectives for each phase.
- **Self-Critique:** The phased approach might be too rigid for highly dynamic or exploratory tasks.

Explicit Diagramming v1.0

- **Compliance Proof:** State machine diagram showing phase transitions is implied by the five-phase description.
- **Self-Critique: NON-COMPLIANCE:** Actual state transition diagram is missing and should be added.
- **Mitigation:** Future revision will include explicit state machine diagram.

Distributed Systems Principles v1.0

- **Compliance Proof:** The "Distributed Systems Implications" section explicitly addresses idempotency, asynchronicity, event ordering, partition tolerance, and observability.
- **Self-Critique:** Implementation details for partition tolerance during phase transitions need further specification.

Assumption Surfacing v1.0

- **Compliance Proof:** Explicit assumptions section with checkboxes for validation.
- **Self-Critique:** Only three assumptions listed; operational loop likely has more implicit assumptions that should be surfaced.

Decision

Decision:

> We will adopt a five-phase, event-driven operational loop: **ANALYZE, BLUEPRINT, CONSTRUCT, VALIDATE, and IDLE**. The OS will transition between these phases based on the completion of prior phase objectives, the state of planning artifacts, and explicit user directives. This loop provides the foundational state machine for all OS operations. The current phase is tracked in `state.txt.ph`.
>
> #### Distributed Systems Implications
>
> The execution of this operational loop and the transitions between phases MUST adhere to the following cross-cutting policies to ensure robustness in a distributed environment:
>
> - **Idempotency & Retries (ADR-OS-023):** All actions that mutate state during a phase transition MUST be idempotent to prevent errors from transient failures.
> - **Asynchronicity (ADR-OS-024):** While the loop is logically sequential, communication *within* a phase (e.g., an agent performing a task) MAY be asynchronous. Handoffs between phases, however, are major state transitions that are typically resolved before the next phase begins.
> - **Event Ordering (ADR-OS-027):** All events generated within the loop MUST be timestamped according to the event ordering policy, using vector clocks where causality is critical.
> - **Partition Tolerance (ADR-OS-028):** The core state management (`state.txt`) is a CP system. In a network partition, it will prioritize consistency, potentially halting phase transitions until the partition is resolved.
> - **Observability (ADR-OS-029):** Every phase transition and significant action within a phase MUST be part of a distributed trace, propagating the `trace_id`.

Confidence: Medium

Rationale

1. **Clarity & Predictability**
2. Self-critique: The phased approach might be too rigid for highly dynamic or exploratory tasks.
3. Confidence: High
4. **Separation of Concerns**
5. Self-critique: Increased complexity in coordinating handoffs between specialized agents for each phase.

6. Confidence: High
7. **Quality Gates**
8. Self-critique: Transitions could become bottlenecks if validation processes are slow or overly strict.
9. Confidence: Medium
10. **Alignment with Software Development**
11. Self-critique: The analogy might not perfectly fit all types of AI-driven projects, leading to conceptual friction.
12. Confidence: Medium
13. **Supports Theory of Constraints**
14. Self-critique: Requires robust monitoring to accurately identify bottlenecks, which adds overhead.
15. Confidence: Medium
16. **Detailed Phase Definitions**
17. Self-critique: The defined transitions might not cover all edge cases or failure modes, requiring future refinement.
18. Confidence: Medium

Alternatives Considered

1. **Monolithic "Do Task" Model**: A single phase where the AI takes a request and tries to do everything (analyze, plan, code, test) in one go. Rejected due to lack of traceability, scalability, and control for complex tasks.
2. Confidence: High
3. **Purely Agile/Kanban Model**: A board of "tasks" without the strategic `Initiative Plan` layer. Rejected as it lacks the high-level strategic planning and lifecycle stage management needed for long-term, multi-stage projects. Our model incorporates Kanban/TOC principles *within* the structured phases.
4. Confidence: High

Consequences

- **Positive:** Provides a highly structured and auditable workflow. Enforces a "plan before doing" and "verify after doing" discipline. Creates clear points for potential human intervention or review. Modular design supports specialized AI agents for each phase.
- **Negative:** Can introduce overhead for very small, simple tasks compared to a purely conversational approach. The effectiveness of the loop relies heavily on the quality of the artifacts generated in each phase.

Clarifying Questions

- How will the OS handle tasks that require jumping back to a previous phase (e.g., a validation failure requiring more construction)?
- What is the mechanism for a human to override a phase transition?
- How are partial or failed phase transitions detected and recovered, especially in distributed or partially available environments?
- What are the escalation and notification procedures if a phase transition is blocked or deadlocked?
- How does the OS ensure traceability and auditability of all phase transitions, including manual overrides?

This template integrates explicit assumption-surfacing, confidence indicators, self-critiques, and clarifying questions as per ADR-OS-021.

ANNOTATION_BLOCK_START

{ "artifact_annotation_header": { "artifact_id_of_host": "adr_os_002_md", "g_annotation_created": 5, "version_tag_of_host_at_annotation": "1.2.0" }, "payload": { "description": "Retrofitted to comply with ADR-OS-032: Canonical Models and Frameworks Registry & Enforcement.", "artifact_type": "DOCUMENTATION", "purpose_statement": "To ensure framework compliance and improve architectural decision clarity.", "authors_and_contributors": [{ "g_contribution": 5, "identifier": "Hybrid_AI_OS" }, { "g_contribution": 4, "identifier": "Framework_Compliance_Retrofit" }], "internal_dependencies": ["adr_os_template_md", "adr_os_032_md"], "linked_issue_ids": [] } }

ANNOTATION_BLOCK_END

ADR-OS-002: Hierarchical Planning Model

- **Status**: Proposed
- **Date**: 2025-06-06
- **Deciders**: [List of decision-makers]
- **Reviewed By**: [List of reviewers]

Context

Complex projects require more than a simple, flat list of tasks. To ensure strategic alignment, provide context to agents, and manage large scopes, a multi-layered planning approach is necessary. We need a structure that connects high-level strategic intent to the granular, tactical work performed by coding agents.

Assumptions

- [ ] The planning hierarchy maps directly to agent specialization (e.g., strategic vs. tactical agents).
- [ ] The overhead of creating multiple planning artifacts is justified by the gain in traceability and context scoping.
- [ ] All agents can interpret and act upon their assigned planning layer without ambiguity.
- [ ] The linkage between planning artifacts (e.g., Request → Analysis → Initiative → Execution) is reliably maintained and observable.
- [ ] The system can handle asynchronous creation and linkage of planning artifacts without loss of context or traceability.
- [ ] The planning artifact store is always available and consistent, or recovers gracefully from partitions.
- [ ] Supervisor Agent(s) have sufficient capability to manage transitions and relationships between planning layers.
- [ ] All compliance requirements from referenced ADRs (e.g., ADR-OS-024, ADR-OS-027, ADR-OS-028, ADR-OS-029) are up-to-date and enforced.

This section was expanded in response to issue_assumptions.txt to surface implicit assumptions and improve framework compliance.

Frameworks/Models Applied

This ADR applies the following canonical models and frameworks (per ADR-OS-032):

Theory of Constraints (ToC) v1.0

- **Compliance Proof:** The hierarchical planning model identifies bottlenecks at each planning level and enables optimization by separating strategic constraints from tactical ones.
- **Self-Critique:** Requires sophisticated "Supervisor Agent" to manage transitions, which could become a bottleneck itself.

KISS (Keep It Simple, Stupid) v1.0

- **Compliance Proof:** Four-tier hierarchy (Request -> Analysis -> Initiative -> Execution) provides necessary complexity without over-engineering.
- **Self-Critique:** Might feel heavyweight for trivial one-off tasks; balance between simplicity and structure needs refinement.

DRY (Don't Repeat Yourself) v1.0

- **Compliance Proof:** Strategic context is captured once in Initiative Plan and referenced by multiple Execution Plans, avoiding duplication.
- **Self-Critique:** Context-loading logic might still require higher-level plan information, potentially creating coupling.

Explicit Diagramming v1.0

- **Compliance Proof:** Hierarchical flow diagram showing Request -> Analysis -> Initiative -> Execution is implied.
- **Self-Critique: NON-COMPLIANCE:** Actual hierarchy diagram is missing and should be added.
- **Mitigation:** Future revision will include explicit planning hierarchy diagram.

Distributed Systems Principles v1.0

- **Compliance Proof:** The "Distributed Systems Implications" section addresses asynchronicity, event ordering, partition tolerance, and observability for planning artifacts.
- **Self-Critique:** Planning artifact store as CP system may impact availability during partitions.

Assumption Surfacing v1.0

- **Compliance Proof:** Explicit assumptions section with checkboxes for validation.
- **Self-Critique:** Only three assumptions listed; hierarchical planning likely has more implicit assumptions about agent capabilities and coordination.

Traceability v1.0

- **Compliance Proof:** "Golden thread" from request to execution ensures full traceability through the planning hierarchy.
- **Self-Critique:** If not enforced, the "golden thread" can be broken, negating the primary benefit.

Decision

Decision:

> We will adopt a multi-tiered, hierarchical planning model that flows from a high-level `Request` down to specific `Execution Plans`. The primary artifacts in this hierarchy are: **`Request` -> `Analysis Report` -> `Initiative Plan` -> `Execution Plan`**. This structure ensures that all work is traceable back to a strategic objective and an originating request.
>
> #### Distributed Systems Implications
>
> The relationships and state transitions between these planning artifacts must be managed with the following considerations:
>
> - **Asynchronicity (ADR-OS-024):** The creation and linkage of these artifacts may be an asynchronous process. For example, submitting a `Request` may immediately return an ID, while the subsequent `Analysis Report` is generated in the background.
> - **Event Ordering (ADR-OS-027):** The causal link between planning artifacts (e.g., this `Execution Plan` was created by that `Initiative Plan`) must be captured using appropriate logical clocks.
> - **Partition Tolerance (ADR-OS-028):** The planning artifact store must behave as a CP system. In a partition, it may become impossible to create new plans or link existing ones until consistency is restored.
> - **Observability (ADR-OS-029):** Every step in the planning hierarchy's lifecycle (creation, update, linkage) must be part of a distributed trace, ensuring the "golden thread" from request to execution is fully observable.

Confidence: High

Rationale

1. **Traceability**
2. Self-critique: If not enforced, the "golden thread" can be broken, negating the primary benefit.
3. Confidence: High
4. **Context Scoping**
5. Self-critique: An agent might still require context from a higher-level plan, leading to more complex context-loading logic.
6. Confidence: High
7. **Separation of Strategic and Tactical Concerns**
8. Self-critique: A rigid separation might hinder agility if a tactical discovery requires immediate strategic reprioritization.
9. Confidence: Medium
10. **Enables Phased Rollout**
11. Self-critique: Defining clear, non-overlapping `initiative_lifecycle_stages` can be challenging for highly interconnected systems.
12. Confidence: Medium
13. **Hierarchy Components & Flow**
14. Self-critique: The strict flow could be inefficient for urgent, small-scale bug fixes.
15. Confidence: Medium

Alternatives Considered

1. **Flat Task List**: A single global list of tasks or issues. Rejected because it lacks strategic context, makes prioritization difficult for large projects, and offers poor traceability for the "why" behind a task.
2. Confidence: High
3. **Two-Tiered (Plan -> Tasks)**: A simpler model with just plans and tasks. Rejected because it merges strategic ("what is the overall goal of this 6-month project?") and tactical ("what specific files do I change today?") planning into a single artifact, which becomes unwieldy and lacks the phased lifecycle management provided by the `Initiative Plan` layer.
4. Confidence: High

Consequences

- **Positive:** Creates a highly structured and organized project management system. Enforces strategic alignment for all tactical work. Provides appropriate levels of abstraction for different agents or human roles. The structured artifacts at each level serve as durable, long-term memory for the project.
- **Negative:** Introduces overhead; creating all these planning artifacts for a very trivial one-off task might feel heavyweight. Requires a sophisticated "Supervisor Agent" to manage the transitions and relationships between these planning layers.

Clarifying Questions

- How will the system handle a `Request` that spawns multiple, independent `Initiative Plans`?
- What is the process for archiving or closing out a completed `Initiative Plan` and its children?
- How are planning artifact linkages (e.g., parent-child relationships) validated and repaired if broken?
- What is the recovery process if the planning artifact store becomes inconsistent or partially unavailable?
- How does the system ensure that context and traceability are preserved during asynchronous or concurrent plan creation?

This template integrates explicit assumption-surfacing, confidence indicators, self-critiques, and clarifying questions as per ADR-OS-021.

ANNOTATION_BLOCK_START

{ "artifact_annotation_header": { "artifact_id_of_host": "adr_os_003_md", "g_annotation_created": 7, "version_tag_of_host_at_annotation": "1.2.0" }, "payload": { "description": "Retrofitted to comply with ADR-OS-032: Canonical Models and Frameworks Registry & Enforcement.", "artifact_type": "DOCUMENTATION", "purpose_statement": "To ensure framework compliance and improve architectural decision clarity.", "authors_and_contributors": [{ "g_contribution": 7, "identifier": "Hybrid_AI_OS" }, { "g_contribution": 4, "identifier": "Framework_Compliance_Retrofit" }], "internal_dependencies": ["adr_os_template_md", "adr_os_032_md"], "linked_issue_ids": [] } }

ANNOTATION_BLOCK_END

ADR-OS-003: Artifact Annotation Strategy (`EmbeddedAnnotationBlock`)

- **Status**: Proposed
- **Date**: 2025-06-06
- **Deciders**: [List of decision-makers]
- **Reviewed By**: [List of reviewers]

Context

For an autonomous agent system to effectively manage, understand, and modify a codebase over time, it cannot rely solely on reading the primary content of files. It requires rich, structured metadata about each artifact's purpose, history, dependencies, quality, and relationship to the overall project. This metadata needs to be durable, version-controlled, and live alongside the artifact it describes, avoiding reliance on external databases or transient agent memory.

Assumptions

- [ ] The chosen embedding method (comment block vs. JSON key) will cover all relevant text-editable file types.
- [ ] The performance overhead of parsing JSON from comment blocks is acceptable.
- [ ] Agents can be reliably programmed to maintain the integrity of the annotation block.
- [ ] All supported file formats can accommodate the annotation block without breaking primary functionality.
- [ ] Annotation parsing and validation logic is robust against malformed or partially corrupted blocks.
- [ ] The annotation block schema is stable and versioned to prevent breaking changes for agents.
- [ ] Updates to annotation blocks and global_registry_map.txt are atomic or recoverable to prevent divergence.
- [ ] All compliance requirements from referenced ADRs (e.g., ADR-OS-023, ADR-OS-029) are up-to-date and enforced.

This section was expanded in response to issue_assumptions.txt to surface implicit assumptions and improve framework compliance.

Frameworks/Models Applied

This ADR applies the following canonical models and frameworks (per ADR-OS-032):

DRY (Don't Repeat Yourself) v1.0

- **Compliance Proof:** Artifact metadata is stored once in the embedded annotation block, eliminating need to duplicate context in external systems or agent memory.
- **Self-Critique:** Risk of divergence between annotation block and mirrored data in global_registry_map.txt if updates fail mid-process.

Single Source of Truth v1.0

- **Compliance Proof:** The embedded annotation block serves as the canonical source for all artifact metadata, avoiding multiple competing sources.
- **Self-Critique:** Requires disciplined agents and strong validation to maintain integrity of the single source.

Assumption Surfacing v1.0

- **Compliance Proof:** Explicit assumptions section with checkboxes for validation about embedding methods, performance, and agent reliability.
- **Self-Critique:** Only three assumptions listed; annotation strategy likely has more implicit assumptions about file formats and parsing capabilities.

Distributed Systems Principles v1.0

- **Compliance Proof:** The "Distributed Systems Implications" section addresses idempotency and observability for annotation operations.
- **Self-Critique:** Missing explicit handling of partition tolerance and event ordering for annotation updates.

Explicit Diagramming v1.0

- **Compliance Proof:** Annotation block structure and embedding patterns are described textually.

- **Self-Critique: NON-COMPLIANCE:** Actual diagram showing annotation block structure and embedding methods is missing.
- **Mitigation:** Future revision will include explicit annotation architecture diagram.

Version Control Integration v1.0

- **Compliance Proof:** Annotations are embedded directly in files to leverage version control synergy and avoid external database dependencies.
- **Self-Critique:** Large, frequent annotation changes could bloat git history.

Self-Describing Systems v1.0

- **Compliance Proof:** Every artifact becomes self-describing through embedded metadata, reducing cognitive load and improving agent understanding.
- **Self-Critique:** Increases file size and requires parsing overhead; balance between self-description and performance needs monitoring.

Decision

Decision:

> We will mandate the use of a comprehensive **`EmbeddedAnnotationBlock`** for all text-editable Project Artifacts managed by the OS. This block will be a structured JSON object containing all critical metadata for the artifact. It will be embedded in a comment block for most files, and as a top-level `_annotationBlock` key in JSON-root files.
>
> #### Distributed Systems Implications
>
> The creation and maintenance of the `EmbeddedAnnotationBlock` must adhere to the following policies:
>
> - **Idempotency (ADR-OS-023):** Any operation that creates or modifies an artifact and its annotation block MUST be idempotent. This prevents the creation of duplicate artifacts or corrupted annotations if the operation is retried.
> - **Observability (ADR-OS-029):** Every modification to an annotation block MUST be captured in a distributed trace. The `trace_id` of the operation causing the change SHOULD be stored within the annotation's history to provide a perfect audit trail from effect back to cause.

Confidence: High

Rationale

1. **Durable Context**
2. Self-critique: An agent could maliciously or accidentally corrupt the JSON within the annotation block, breaking subsequent parsing.
3. Confidence: High
4. **Single Source of Truth**
5. Self-critique: There is a risk of divergence between the annotation block and mirrored data in the `global_registry_map.txt` if an update fails mid-process.
6. Confidence: Medium
7. **Version Control Synergy**
8. Self-critique: Large, frequent changes to annotations (e.g., test results) could bloat the git history.
9. Confidence: High
10. **Supports Agent Specialization**
11. Self-critique: Requires a well-defined and stable schema for the annotation block to prevent breaking changes for specialized agents.
12. Confidence: High
13. **Enables Constraint Enforcement**
14. Self-critique: The locking mechanism for requisites could be too rigid, preventing necessary evolution of the system.
15. Confidence: Medium

Alternatives Considered

1. **Centralized Metadata Database**: Storing all artifact metadata in an external database. Rejected due to the risk of desynchronization between the database and the actual files in version control and increased system complexity.
2. Confidence: High
3. **Sidecar Metadata Files**: Storing metadata in a companion file (e.g., `Button.tsx.meta.json`). Rejected because it doubles the number of files to manage and increases the risk of the metadata file being separated from its source artifact.
4. Confidence: High

Consequences

- **Positive:** Creates self-describing, intelligent artifacts. Massively improves traceability and context durability. Provides a robust mechanism for enforcing architectural and design constraints. Reduces the cognitive load on agents.

- **Negative:** Increases file size. Requires disciplined agents and a strong `VALIDATE` phase to ensure integrity. Introduces minor performance overhead for parsing.

Clarifying Questions

- What is the recovery and validation process if an `EmbeddedAnnotationBlock` becomes corrupted or is missing required fields?
- How are annotation updates coordinated in distributed or concurrent agent scenarios to prevent race conditions or partial updates?
- What is the fallback or handling strategy for binary or non-text-editable files that cannot contain embedded annotations?
- How is the annotation schema versioning managed to ensure backward compatibility and safe evolution?
- What are the audit and traceability requirements for changes to annotation blocks, and how are these enforced?

This template integrates explicit assumption-surfacing, confidence indicators, self-critiques, and clarifying questions as per ADR-OS-021.

ANNOTATION_BLOCK_START

{ "artifact_annotation_header": { "artifact_id_of_host": "adr_os_004_md", "g_annotation_created": 9, "version_tag_of_host_at_annotation": "1.2.0" }, "payload": { "description": "Retrofitted to comply with ADR-OS-032: Canonical Models and Frameworks Registry & Enforcement.", "artifact_type": "DOCUMENTATION", "purpose_statement": "To ensure framework compliance and improve architectural decision clarity.", "authors_and_contributors": [{ "g_contribution": 9, "identifier": "Hybrid_AI_OS" }, { "g_contribution": 4, "identifier": "Framework_Compliance_Retrofit" }], "internal_dependencies": ["adr_os_template_md", "adr_os_032_md"], "linked_issue_ids": [] } }

ANNOTATION_BLOCK_END

ADR-OS-004: Global Event Tracking & Versioning (g and v)

- **Status**: Proposed
- **Date**: 2025-06-06
- **Deciders**: [List of decision-makers]
- **Reviewed By**: [List of reviewers]

Context

In an event-driven, asynchronous system where multiple actions can occur over time, a reliable mechanism is needed to sequence events, establish causality, ensure data integrity, and provide a comprehensive audit trail. Simple timestamps are prone to clock skew and don't provide a clear, causal sequence of OS-level events. Furthermore, when multiple agents could potentially interact with OS Control Files, a mechanism to prevent race conditions and stale data writes is required.

Assumptions

- [ ] The global event counter (g) is always incremented atomically and never skipped.
- [ ] The version counter (v) is checked and incremented on every write to a mutable OS Control File.
- [ ] All agents interacting with OS Control Files implement the read-check-increment-write cycle for v.
- [ ] The system can detect and recover from failed or partial increments of g or v.
- [ ] Logical clocks (e.g., vector clocks) are used where causal ordering is required across distributed agents.
- [ ] The audit trail is only as reliable as the consistent use of g and v by all agents.
- [ ] All compliance requirements from referenced ADRs (e.g., ADR-OS-027, ADR-OS-028, ADR-OS-029) are up-to-date and enforced.

This section was expanded in response to issue_assumptions.txt to surface implicit assumptions and improve framework compliance.

Frameworks/Models Applied

This ADR applies the following canonical models and frameworks (per ADR-OS-032):

Distributed Systems Principles v1.0

- **Compliance Proof:** The "Distributed Systems Implications" section explicitly addresses logical clocks, partition tolerance, and observability for event tracking.
- **Self-Critique:** CP system behavior during partitions may halt operations; needs careful consideration of availability trade-offs.

CAP Theorem v1.0

- **Compliance Proof:** Explicitly chooses Consistency and Partition tolerance (CP system) for state.txt, acknowledging availability trade-offs.
- **Self-Critique:** High contention on state.txt could impact system availability even without network partitions.

Event Ordering v1.0

- **Compliance Proof:** Global event counter g provides total ordering of events; integration with vector clocks for causal ordering in distributed scenarios.
- **Self-Critique:** Simple integer counter may not scale to distributed, multi-node deployments without additional coordination mechanisms.

Optimistic Locking v1.0

- **Compliance Proof:** Version counter v implements optimistic locking pattern to prevent stale writes and race conditions.
- **Self-Critique:** All agents must implement read-check-increment-write cycle; missing this pattern could lead to data corruption.

Assumption Surfacing v1.0

- **Compliance Proof:** Explicit assumptions section with checkboxes for validation about atomic increments and version checking.
- **Self-Critique:** Only three assumptions listed; event tracking system likely has more implicit assumptions about concurrency and failure modes.

Audit Trail v1.0

- **Compliance Proof:** Global event counter provides comprehensive audit trail with unique, time-ordered IDs for all significant OS events.
- **Self-Critique:** Audit trail quality depends on consistent use of g increment; missing or incorrect usage could break traceability.

Decision

Decision:

> We will adopt a dual-mechanism approach for system-wide event tracking and versioning: 1. **Global Event Counter (g)**: A single, monotonically increasing integer, stored in `state.txt`, incremented for every significant OS action. Used for sequencing, timestamping, and unique ID generation. 2. **Instance Version Counter (v)**: A per-file, integer-based version counter, present in the schema of all mutable OS Control Files. Used for optimistic locking to prevent stale writes.
>
> **Distributed Systems Implications**
>
> While g provides a simple total ordering, it is insufficient for complex distributed workflows. This system MUST be augmented by the following policies:
>
> - **Logical Clocks (ADR-OS-027):** The g counter serves as the baseline for event ordering. However, for workflows requiring strict causal history across multiple agents or services, vector clocks MUST be used in addition to g.
> - **Partition Tolerance (ADR-OS-028):** The `state.txt` file, which holds g, is a CP system. During a network partition, the inability to reliably increment g will halt operations in the minority partition that require a new global event ID.
> - **Observability (ADR-OS-029):** The value of g at the time of an operation must be included in all structured logs and traces. This, combined with a `trace_id`, allows events to be correlated both by their absolute sequence and their causal flow.

Confidence: High

Rationale

1. **Causal Ordering (g)**
2. Self-critique: If g is not updated atomically, event order could be ambiguous.
3. Confidence: High
4. **Data Integrity (v)**
5. Self-critique: If agents do not check v before writing, stale data could overwrite newer changes.
6. Confidence: High
7. **Simplicity & Durability**
8. Self-critique: The simplicity of integer counters may not scale to distributed, multi-node deployments without additional coordination. This is explicitly addressed by ADR-OS-027 and ADR-OS-028, which introduce logical clocks and partition tolerance protocols for more complex scenarios.
9. Confidence: High

Alternatives Considered

1. **Timestamp-based Versioning**: Relies on synchronized clocks, which can be unreliable. Rejected in favor of a monotonic counter for sequencing.
2. Confidence: High
3. **Pessimistic Locking (File Locks)**: More complex and can reduce concurrency. Rejected in favor of optimistic locking for this use case.
4. Confidence: High

Consequences

- **Positive:** Provides a robust, system-wide, ordered log of all significant events. Prevents data corruption from stale writes. Creates unique, time-ordered, and human-readable IDs for core entities. Simple to implement and maintain.
- **Negative:** `state.txt` becomes a point of high contention. All agents must implement the read-check-increment-write cycle for the v field.

Clarifying Questions

- How will the system handle concurrent attempts to increment g and ensure atomicity across distributed agents?
- What is the recovery process if v or g is accidentally skipped, duplicated, or corrupted?
- How are audit trail gaps or inconsistencies detected and remediated?
- What mechanisms are in place to coordinate event and version counters in multi-node or partitioned deployments?
- How is the integrity and completeness of the event log validated, and what is the escalation process for detected anomalies?

This template integrates explicit assumption-surfacing, confidence indicators, self-critiques, and clarifying questions as per ADR-OS-021.

ANNOTATION_BLOCK_START

{ "artifact_annotation_header": { "artifact_id_of_host": "adr_os_005_md", "g_annotation_created": 10, "version_tag_of_host_at_annotation": "1.2.0" }, "payload": { "description": "Retrofitted to comply with ADR-OS-032: Canonical Models and Frameworks Registry & Enforcement.", "artifact_type": "DOCUMENTATION", "purpose_statement": "To ensure framework compliance and improve architectural decision clarity.", "authors_and_contributors": [{ "g_contribution": 10, "identifier": "Hybrid_AI_OS" }, { "g_contribution": 4, "identifier": "Framework_Compliance_Retrofit" }], "internal_dependencies": ["adr_os_template_md", "adr_os_032_md"], "linked_issue_ids": [] } }

ANNOTATION_BLOCK_END

ADR-OS-005: Directory Structure & File Naming Conventions

- **Status**: Accepted
- **Date**: 2025-06-07
- **Deciders**: [List of decision-makers]
- **Reviewed By**: [List of reviewers]

Context

A predictable and well-organized file system structure is essential for the Hybrid_AI_OS and any human collaborators. The OS needs a reliable way to locate its operational files, user-provided materials, and the project workspace. Hardcoding paths would make the system brittle and difficult to adapt or package as a distributable tool.

Assumptions

- [ ] A `haios.config.json` file will always be present at the project root where the OS is initialized.
- [ ] The OS has read permissions for the `haios.config.json` file at startup.
- [ ] The paths defined within `haios.config.json` are valid and accessible to the OS.
- [ ] The file system supports the directory structure and naming conventions defined in the config.
- [ ] The OS can detect and handle missing or malformed config files gracefully.
- [ ] The configuration-driven approach is compatible with distributed and multi-agent deployments.
- [ ] All compliance requirements from referenced ADRs (e.g., ADR-OS-032) are up-to-date and enforced.

This section was expanded in response to issue_assumptions.txt to surface implicit assumptions and improve framework compliance.

Frameworks/Models Applied

This ADR applies the following canonical models and frameworks (per ADR-OS-032):

KISS (Keep It Simple, Stupid) v1.0

- **Compliance Proof:** Directory structure uses simple, predictable naming conventions with clear separation of concerns (os_root, project_workspace, etc.).
- **Self-Critique:** Configuration-driven approach adds startup complexity requiring config file parsing before any operations.

DRY (Don't Repeat Yourself) v1.0

- **Compliance Proof:** Single configuration file eliminates hardcoded path duplication throughout the system; all paths defined once in haios.config.json.
- **Self-Critique:** Reliance on single config file might be less suitable for highly complex, multi-repository projects.

Separation of Concerns v1.0

- **Compliance Proof:** Clean separation between OS internal data (os_root), project being built (project_workspace), and build materials (templates, source materials).
- **Self-Critique:** Users might misconfigure paths, leading to unexpected behavior and concern boundary violations.

Configuration-Driven Design v1.0

- **Compliance Proof:** All operational paths defined in central haios.config.json file, making system adaptable and portable.
- **Self-Critique:** Corrupted or invalid config file becomes single point of failure for OS initialization.

Explicit Diagramming v1.0

- **Compliance Proof:** Directory structure diagram is provided showing default layout and path relationships.
- **Self-Critique: PARTIAL COMPLIANCE:** Diagram is textual; visual directory tree diagram would improve clarity.

Assumption Surfacing v1.0

- **Compliance Proof:** Explicit assumptions section with checkboxes for validation about config file presence, permissions, and path validity.
- **Self-Critique:** Only three assumptions listed; directory structure likely has more implicit assumptions about file system capabilities.

Portability v1.0

- **Compliance Proof:** Configuration-driven approach eliminates hardcoded paths, making system distributable and adaptable to different environments.
- **Self-Critique:** Environment variables approach rejected in favor of self-contained config file for better portability.

Decision

Decision:

> We will adopt a **configuration-driven directory structure**. All key paths required for OS operation will be defined in a central project configuration file named **`haios.config.json`**, located at the project root. The OS **MUST** read this file on initialization to determine all operational paths. The OS Control Files will maintain their g-based naming convention.

Confidence: High

Rationale

1. **Flexibility & Configurability**
2. Self-critique: Increased startup complexity as the OS must first parse the config file before it can access any other file.
3. Confidence: High
4. **Explicitness**
5. Self-critique: A corrupted or invalid `haios.config.json` becomes a single point of failure for OS initialization.
6. Confidence: High
7. **Separation of Concerns**
8. Self-critique: Users might misconfigure paths, leading to unexpected behavior.
9. Confidence: Medium
10. **Containment & Scalability**
11. Self-critique: The reliance on a single config file might be less suitable for highly complex, multi-repository projects in the future.
12. Confidence: Medium

Alternatives Considered

1. **Hardcoded Paths**: The previous approach. Rejected as too rigid and not suitable for a distributable tool.
2. Confidence: High
3. **Environment Variables for Paths**: Rejected as it makes project configuration less portable and self-contained compared to a config file committed to the project repository.
4. Confidence: High

Consequences

- **Positive:** System is no longer tied to hardcoded paths, making it robust and portable. Provides a single, clear configuration entry point. Aligns with standard practices for modern development tooling.
- **Negative:** A missing or malformed `haios.config.json` is a fatal startup error. Adds a minor, but acceptable, level of indirection to all file access.

Clarifying Questions

- What is the defined validation schema for `haios.config.json`, and how is it versioned and evolved?
- How does the OS behave if a configured path points to a non-existent or inaccessible directory, and what are the recovery or notification mechanisms?
- How will this configuration approach be extended to support multi-agent, distributed, or multi-repository deployments?
- What is the migration or upgrade process if the directory structure or config schema changes in future versions?
- How are configuration errors, overrides, and environment-specific customizations tracked and audited?

This template integrates explicit assumption-surfacing, confidence indicators, self-critiques, and clarifying questions as per ADR-OS-021.

> **Rationale:**
>
> - **Flexibility & Configurability:** Defining paths in a config file makes the OS adaptable to different project layouts or user preferences. It is a core principle for creating a distributable "Agent Development Kit (ADK)".

- **Explicitness:** `haios.config.json` makes the entire directory layout explicit and self-documenting. There is no "magic" in where the OS looks for its files.
- **Separation of Concerns:** The structure cleanly separates the OS's internal data (defined by `paths.os_root`), the project being built (defined by `paths.project_workspace`), and the materials used for building (defined by `paths.project_templates` and `paths.initial_source_materials`).
- **Containment & Scalability:** The `os_root/initiatives/initiative_<g>/` structure (whose parent `initiatives` path is defined in the config) remains the mechanism for containing all operational artifacts for a specific strategic initiative.

`haios.config.json` and Default Directory Structure Specification:

The OS will expect a `haios.config.json` file at the project root. A default project scaffold will generate this file and the corresponding directories:

Default `haios.config.json` `paths` object: `json "paths": { "os_root": "./os_root", "project_workspace": "./project_workspace", "project_templates": "./project_templates", "initial_source_materials": "./initial_source_materials", "scaffold_definitions": "./os_root/scaffold_definitions" }`

Default Directory Layout corresponding to the config: `[PROJECT_NAME]/ haios.config.json os_root/ state.txt global_issues_summary.txt global_registry_map.txt ... (other global files & dirs) initiatives/ initiative_<g_init>/ ... project_workspace/ project_templates/ initial_source_materials/`

File Naming Conventions (Unchanged):

The decision to use g-based, unique, and descriptive filenames for OS Control Files and OS-generated artifacts remains unchanged. * `request_<g>.txt`, `init_plan_<g>.txt`, `exec_plan_<g>.txt`, `issue_<g>.txt` * `analysis_report_..._g<report_g>.md`, `validation_report_..._g<report_g>.md` * `Scaffold Definition` definitions will use semantic names (e.g., `react_module_v1.json`).

Consequences:

Pros:

- System is no longer tied to hardcoded paths, making it more robust and portable.
- Provides a single, clear configuration entry point for a project.
- Aligns with standard practices for modern development tooling.

Cons:

- Adds one level of indirection: the OS must always read the config file first to know where to operate. This is a negligible and worthwhile trade-off.
- A missing or malformed `haios.config.json` is now a fatal startup error for the OS.

Alternatives Considered:

- **Hardcoded Paths:** The previous approach. Rejected as too rigid and less aligned with a distributable package model.
- **Environment Variables for Paths:** Using environment variables to define paths. Rejected as it makes project configuration less portable and self-contained compared to a config file within the project directory itself.

ANNOTATION_BLOCK_START

{ "artifact_annotation_header": { "artifact_id_of_host": "adr_os_006_md", "g_annotation_created": 12, "version_tag_of_host_at_annotation": "1.2.0" }, "payload": { "description": "Retrofitted to comply with ADR-OS-032: Canonical Models and Frameworks Registry & Enforcement.", "artifact_type": "DOCUMENTATION", "purpose_statement": "To ensure framework compliance and improve architectural decision clarity.", "authors_and_contributors": [{ "g_contribution": 12, "identifier": "Hybrid_AI_OS" }, { "g_contribution": 4, "identifier": "Framework_Compliance_Retrofit" }], "internal_dependencies": ["adr_os_template_md", "adr_os_032_md"], "linked_issue_ids": [] } }

ANNOTATION_BLOCK_END

ADR-OS-006: Scaffolding Process & `Scaffold Definition` Usage

- **Status**: Proposed
- **Date**: 2024-06-06
- **Deciders**: [List of decision-makers]
- **Reviewed By**: [List of reviewers]

Context

To ensure consistency, reduce boilerplate, and accelerate the setup of new projects or components, a standardized and automated scaffolding process is required. This process must create not only the file and directory structure but also embed foundational metadata (`EmbeddedAnnotationBlock`), content placeholders, and initial testing considerations directly into the newly created artifacts.

Assumptions

- [ ] `Scaffold Definition` definitions are well-formed, valid JSON.
- [ ] Boilerplate assets referenced in a `Scaffold Definition` exist at the specified paths within `project_templates/`.
- [ ] The OS has write permissions to the target `project_workspace/` directory.
- [ ] The template processing logic can handle all supported file types and placeholder patterns.
- [ ] The scaffolding process is idempotent and can recover from partial failures.
- [ ] All scaffolded artifacts receive a valid and complete EmbeddedAnnotationBlock at creation.
- [ ] All compliance requirements from referenced ADRs (e.g., ADR-OS-032) are up-to-date and enforced.

This section was expanded in response to issue_assumptions.txt to surface implicit assumptions and improve framework compliance.

Frameworks/Models Applied

This ADR applies the following canonical models and frameworks (per ADR-OS-032):

AAA (Arrange, Act, Assert) v1.0

- **Compliance Proof:** Scaffolding process follows AAA pattern: Arrange (prepare templates), Act (execute scaffold), Assert (validate created artifacts with annotations).
- **Self-Critique:** Initial annotation might become stale if not updated during development, giving false sense of context.

DRY (Don't Repeat Yourself) v1.0

- **Compliance Proof:** Template-based approach eliminates code duplication by reusing boilerplate assets from project_templates/ for multiple component creation.
- **Self-Critique:** Managing the link between Scaffold Definition and its many template assets could become complex.

Explicit Diagramming v1.0

- **Compliance Proof:** Scaffolding process and template structure are described textually.
- **Self-Critique: NON-COMPLIANCE:** Actual diagram showing scaffolding workflow and template relationships is missing.
- **Mitigation:** Future revision will include explicit scaffolding process diagram.

Separation of Concerns v1.0

- **Compliance Proof:** Clear separation between scaffold instructions (Scaffold Definition), template content (project_templates/), and target artifacts (project_workspace/).
- **Self-Critique:** Template processing logic could become complex subsystem to maintain.

Self-Describing Systems v1.0

- **Compliance Proof:** Every scaffolded artifact receives complete EmbeddedAnnotationBlock from inception, making it immediately self-describing.
- **Self-Critique:** Poorly designed Scaffold Definition could enforce bad practices across many components.

Assumption Surfacing v1.0

- **Compliance Proof:** Explicit assumptions section with checkboxes for validation about JSON validity, template existence, and write permissions.
- **Self-Critique:** Only three assumptions listed; scaffolding process likely has more implicit assumptions about template structure and processing capabilities.

Best Practices Enforcement v1.0

- **Compliance Proof:** Scaffolding system enforces consistent project structure and coding conventions through standardized templates.
- **Self-Critique:** Requires upfront investment in creating well-structured Scaffold Definitions and templates.

Decision

Decision:

> We will implement a scaffolding system driven by **`Scaffold Definition`** JSON files that instruct a dedicated **`SCAFFOLDING`-type `Execution Plan`**. This plan will use boilerplate assets from `/project_templates` to create new artifacts in the `/project_workspace`, injecting a complete `EmbeddedAnnotationBlock` into each file upon creation.

Confidence: High

Rationale

1. **Consistency & Best Practices**
2. Self-critique: A poorly designed `Scaffold Definition` could enforce bad practices across many components.
3. Confidence: High
4. **Durable Context from Inception**
5. Self-critique: The initial annotation might become stale if not updated during development, giving a false sense of context.
6. Confidence: High
7. **Separation of Instruction and Content**
8. Self-critique: Managing the link between a `Scaffold Definition` and its many template assets could become complex.
9. Confidence: Medium
10. **Dynamic Customization**
11. Self-critique: The logic for placeholder replacement and customization could become a complex subsystem to maintain.
12. Confidence: Medium

Alternatives Considered

1. **Manual Scaffolding**: Rejected as it is slow, error-prone, and fails to establish the initial `EmbeddedAnnotationBlock` required for autonomous operation.
2. Confidence: High
3. **Simple File Copy**: Rejected because it misses the primary benefit of creating intelligent, context-aware artifacts from inception.
4. Confidence: High

Consequences

- **Positive:** Highly automated and reliable project bootstrapping. Creates "intelligent" artifacts from day one. Enforces project structure and coding conventions. The process is fully auditable.
- **Negative:** Requires an upfront investment in creating well-structured `Scaffold Definition` definitions and templates. The template processing logic can be complex.

Clarifying Questions

- How will versioning and backward compatibility of `Scaffold Definition` files and templates be managed as requirements evolve?
- What is the process for updating or migrating existing components that were created from an older scaffold version?
- How does the system validate the integrity and completeness of scaffolded artifacts, especially their `EmbeddedAnnotationBlock`?
- What are the error recovery and rollback procedures if the scaffolding process fails partway through execution?
- How are customizations and overrides handled for components that deviate from the standard scaffold?

This template integrates explicit assumption-surfacing, confidence indicators, self-critiques, and clarifying questions as per ADR-OS-021.

ANNOTATION_BLOCK_START

{ "artifact_annotation_header": { "artifact_id_of_host": "adr_os_007_md", "g_annotation_created": 13, "version_tag_of_host_at_annotation": "1.2.0" }, "payload": { "description": "Retrofitted to comply with ADR-OS-032: Canonical Models and Frameworks Registry & Enforcement.", "artifact_type": "DOCUMENTATION", "purpose_statement": "To ensure framework compliance and improve architectural decision clarity.", "authors_and_contributors": [{ "g_contribution": 13, "identifier": "Hybrid_AI_OS" }, { "g_contribution": 4, "identifier": "Framework_Compliance_Retrofit" }], "internal_dependencies": ["adr_os_template_md", "adr_os_032_md"], "linked_issue_ids": [] } }

ANNOTATION_BLOCK_END

ADR-OS-007: Integrated Testing Lifecycle & Artifacts

- **Status**: Proposed
- **Date**: 2025-06-08
- **Deciders**: [List of decision-makers]
- **Reviewed By**: [List of reviewers]

Context

An autonomous development system can inadvertently produce flawed code or "hallucinate" test success. To ensure project quality and reliability, a rigorous, evidence-based testing lifecycle must be an explicit and non-negotiable part of the development process. Claims of success must be backed by verifiable proof.

Assumptions

- [ ] A trusted, isolated environment is available for the `Testing Agent` to execute tests.
- [ ] `Test Results Artifacts` have a consistent, machine-parsable format.
- [ ] The `Validation Agent` has read access to both the `Test Script` and `Test Results` artifacts.
- [ ] The integrity and security of the isolated test environment is continuously monitored and enforced.
- [ ] The system can detect and handle flaky or intermittent test failures.
- [ ] All test artifacts are versioned and traceable to their source execution and agent.
- [ ] All compliance requirements from referenced ADRs (e.g., ADR-OS-032) are up-to-date and enforced.

This section was expanded in response to issue_assumptions.txt to surface implicit assumptions and improve framework compliance.

Frameworks/Models Applied

This ADR applies the following canonical models and frameworks (per ADR-OS-032):

AAA (Arrange, Act, Assert) v1.0

- **Compliance Proof:** Testing lifecycle follows AAA pattern: Arrange (create test scripts), Act (execute tests via Testing Agent), Assert (validate results in VALIDATE phase).
- **Self-Critique:** System becomes dependent on integrity of Testing Agent's isolated environment.

Separation of Duties v1.0

- **Compliance Proof:** Strict separation between Coding Agent (creates tests), Testing Agent (executes tests), and Validation Agent (verifies results).
- **Self-Critique:** Introduces process overhead requiring more Execution Plans and complex orchestration between agent roles.

Evidence-Based Verification v1.0

- **Compliance Proof:** Claims of test success must be backed by verifiable Test Results Artifacts; no self-reporting allowed.
- **Self-Critique:** Added artifact types (Test Script, Test Results) increase complexity of data ecosystem.

Idempotency v1.0

- **Compliance Proof:** Test execution can be repeated safely; Testing Agent produces consistent results for same inputs.
- **Self-Critique:** Flaky tests or intermittent failures need special handling within this lifecycle.

Assumption Surfacing v1.0

- **Compliance Proof:** Explicit assumptions section with checkboxes for validation about isolated environment, artifact formats, and access permissions.
- **Self-Critique:** Only three assumptions listed; testing lifecycle likely has more implicit assumptions about test environments and failure modes.

Zero Trust Security v1.0

- **Compliance Proof:** No agent is trusted to self-report test success; independent Testing Agent and Validation Agent provide verification chain.
- **Self-Critique:** "Fox guarding the henhouse" problem eliminated but at cost of increased operational complexity.

First-Class Citizen Principle v1.0

- **Compliance Proof:** Testing becomes first-class citizen with dedicated artifacts, agents, and lifecycle phases rather than afterthought.
- **Self-Critique:** Requires secure, isolated process for Testing Agent which adds infrastructure requirements.

Decision

Decision:

> We will implement a closed-loop, evidence-based testing lifecycle with a strict **separation of duties**. A `Coding Agent` may create/update `Test Script` artifacts, but only a trusted `Testing Agent` may execute them and produce the official, "signed" `Test Results Artifact`. The `VALIDATE` phase acts as an auditor, verifying this evidence to update the quality status of source artifacts.

Confidence: High

Rationale

1. **Prevents "Fox Guarding the Hennhouse"**
2. Self-critique: This introduces process overhead, requiring more `Execution Plans` and more complex orchestration between agent roles.
3. Confidence: High
4. **Evidence over Declaration**
5. Self-critique: The system becomes dependent on the integrity of the `Testing Agent`'s isolated environment.
6. Confidence: High
7. **Makes Testing a First-Class Citizen**
8. Self-critique: The added artifact types (`Test Script`, `Test Results`) increase the complexity of the data ecosystem.
9. Confidence: High

Alternatives Considered

1. **Trust-Based Model**: Allowing the `CONSTRUCT` agent to run tests and self-report success. Rejected as fundamentally unverifiable and insecure.
2. Confidence: High
3. **Validation Phase Runs Tests**: Rejected because it conflates the concerns of *executing* tests (a `CONSTRUCT`-like activity) with *validating the results of execution* (a `VALIDATE` activity).
4. Confidence: High

Consequences

- **Positive:** Creates a high-trust, verifiable quality assurance process. The system becomes inherently self-auditing. Dramatically reduces the risk of AI shortcuts or errors going undetected.
- **Negative:** Increased operational overhead due to more plans and artifacts. Requires a secure, isolated process for the `Testing Agent`.

Clarifying Questions

- How is the "trustworthiness" of the `Testing Agent`'s execution environment technically enforced?
- What is the exact schema for the `Test Results Artifact`?
- How are flaky tests or intermittent failures handled within this lifecycle?

This template integrates explicit assumption-surfacing, confidence indicators, self-critiques, and clarifying questions as per ADR-OS-021.

ANNOTATION_BLOCK_START

{ "artifact_annotation_header": { "artifact_id_of_host": "adr_os_008_md", "g_annotation_created": 16, "version_tag_of_host_at_annotation": "1.2.0" }, "payload": { "description": "Retrofitted to comply with ADR-OS-032: Canonical Models and Frameworks Registry & Enforcement.", "artifact_type": "DOCUMENTATION", "purpose_statement": "To ensure framework compliance and improve architectural decision clarity.", "authors_and_contributors": [{ "g_contribution": 16, "identifier": "Hybrid_AI_OS" }, { "g_contribution": 4, "identifier": "Framework_Compliance_Retrofit" }], "internal_dependencies": ["adr_os_template_md", "adr_os_032_md"], "linked_issue_ids": [] } }

ANNOTATION_BLOCK_END

ADR-OS-008: OS-Generated Reporting Strategy

- **Status**: Proposed
- **Date**: 2024-05-31
- **Deciders**: [List of decision-makers]
- **Reviewed By**: [List of reviewers]

Context

For a human supervisor to effectively manage and trust the Hybrid_AI_OS, the system's reasoning, progress, and findings must be transparent and comprehensible. Relying on raw logs or structured data files is insufficient. The OS needs a formal mechanism to synthesize information and present it in a clear, narrative, and evidence-based format.

Assumptions

- [ ] Human-readable reports are a critical requirement for project oversight and trust.
- [ ] Markdown is a suitable and sufficient format for these reports.
- [ ] The overhead of generating reports is an acceptable trade-off for transparency.
- [ ] Report templates and outlines are versioned and consistently applied across all report types.
- [ ] The system can detect and mitigate "report fatigue" or low-quality, formulaic self-assessments.
- [ ] All reports are linked to their source data and traceable via trace_id and event references.
- [ ] All compliance requirements from referenced ADRs (e.g., ADR-OS-032) are up-to-date and enforced.

This section was expanded in response to issue_assumptions.txt to surface implicit assumptions and improve framework compliance.

Frameworks/Models Applied

This ADR applies the following canonical models and frameworks (per ADR-OS-032):

Traceability v1.0

- **Compliance Proof:** Reports provide transparent audit trail of AI reasoning, progress, and findings with clear linkage to source data and decisions.
- **Self-Critique:** Reports might oversimplify complex issues, potentially misleading human reviewers.

Distributed Systems Principles v1.0

- **Compliance Proof:** Reports include trace_id references and event correlation to support distributed observability.
- **Self-Critique: PARTIAL COMPLIANCE:** Missing explicit discussion of report generation in distributed/partitioned scenarios.

Self-Critique Methodology v1.0

- **Compliance Proof:** All reports include mandatory self-assessment sections forcing structured critical thinking about decisions and findings.
- **Self-Critique:** AI might learn to "game" self-assessment sections, providing formulaic rather than genuinely critical answers.

Human-Centered Design v1.0

- **Compliance Proof:** Reports designed for human comprehension with narrative synthesis rather than raw data dumps.
- **Self-Critique:** If reports are too verbose or frequent, could lead to "report fatigue" and be ignored.

Assumption Surfacing v1.0

- **Compliance Proof:** Explicit assumptions section with checkboxes for validation about human requirements, format suitability, and overhead acceptance.
- **Self-Critique:** Only three assumptions listed; reporting strategy likely has more implicit assumptions about human reading patterns and information needs.

Structured Thinking Enforcement v1.0

- **Compliance Proof:** Predefined report outlines enforce consistent, structured analysis and presentation across all AI-generated reports.
- **Self-Critique:** Maintaining consistency across different report templates could become maintenance burden.

Evidence-Based Decision Making v1.0

- **Compliance Proof:** Reports synthesize evidence from multiple sources and provide clear reasoning chains for conclusions.
- **Self-Critique:** Quality of reports dependent on AI's synthesis and writing capabilities.

Decision

Decision:

> The OS will be responsible for generating three primary types of human-readable reports as annotated Markdown (`.md`) artifacts: **`Analysis Report`** (post-`ANALYZE`), **`Validation Report`** (post-`VALIDATE`), and **`Progress Review`** (on-demand). Each report will follow a predefined, structured outline that includes sections for critical self-assessment.

Confidence: High

Rationale

1. **Transparency & Auditability**
2. Self-critique: The reports might oversimplify complex issues, potentially misleading a human reviewer.
3. Confidence: High
4. **Exploiting the Human Constraint**
5. Self-critique: If reports are too verbose or frequent, they could lead to "report fatigue" and be ignored.
6. Confidence: Medium
7. **Durable, Shareable Artifacts**
8. Self-critique: Maintaining consistency across different report templates could become a maintenance burden.
9. Confidence: High
10. **Enforcing Structured Thinking**
11. Self-critique: The AI might learn to "game" the self-assessment sections, providing formulaic rather than genuinely critical answers.
12. Confidence: Medium

Alternatives Considered

1. **Directly Reading JSON Files**: Rejected as inefficient, not user-friendly, and lacking narrative synthesis.
2. Confidence: High
3. **Simple Log Output**: Rejected as it lacks structure, synthesis, and a clear "final verdict" for key project phases.
4. Confidence: High

Consequences

- **Positive:** Makes the AI's thought process visible and auditable. Provides high-quality, decision-ready information to human supervisors. Promotes more rigorous and self-critical operational behavior from the AI.
- **Negative:** Adds computational and time overhead to the OS lifecycle. The quality of reports is dependent on the AI's synthesis and writing capabilities.

Clarifying Questions

- What are the specific schemas/outlines for each of the three report types, and how are they versioned and evolved?
- How can a human or agent trigger an on-demand `Progress Review`, and what is the audit trail for such triggers?
- What mechanisms will be in place to detect and mitigate low-quality, formulaic, or "gamed" self-assessments in reports?
- How does the system handle report generation, consistency, and traceability in distributed or partitioned environments?
- What is the process for updating, archiving, or deprecating report templates as project requirements evolve?

This template integrates explicit assumption-surfacing, confidence indicators, self-critiques, and clarifying questions as per ADR-OS-021.

ANNOTATION_BLOCK_START

{ "artifact_annotation_header": { "artifact_id_of_host": "adr_os_009_md", "g_annotation_created": 19, "version_tag_of_host_at_annotation": "1.2.0" }, "payload": { "description": "Retrofitted to comply with ADR-OS-032: Canonical Models and Frameworks Registry & Enforcement.", "artifact_type": "DOCUMENTATION", "purpose_statement": "To ensure framework compliance and improve architectural decision clarity.", "authors_and_contributors": [{ "g_contribution": 19, "identifier": "Hybrid_AI_OS" }, { "g_contribution": 4, "identifier": "Framework_Compliance_Retrofit" }], "internal_dependencies": ["adr_os_template_md", "adr_os_032_md"], "linked_issue_ids": [] } }

ANNOTATION_BLOCK_END

ADR-OS-009: Issue Management & Summarization

- **Status**: Proposed
- **Date**: 2025-06-09
- **Deciders**: [List of decision-makers]
- **Reviewed By**: [List of reviewers]

Context

A robust system requires a formal mechanism for tracking, managing, and resolving issues. A simple, unstructured log is insufficient. We need a system that treats issues as first-class entities, linked to their context (plans, artifacts), and summarized at different levels for effective project oversight.

Assumptions

- [ ] A file-based, hierarchical issue management system is sufficient for the OS's internal needs.
- [ ] The overhead of maintaining summary files is acceptable for the performance gains in retrieving issue overviews.
- [ ] Issues can be adequately represented as structured JSON data.
- [ ] The file system can handle the volume and concurrency requirements of issue management.
- [ ] The system can detect and recover from synchronization bugs across issue, summary, and global files.
- [ ] The issue schema is extensible to capture all necessary nuances and cross-initiative dependencies.
- [ ] All compliance requirements from referenced ADRs (e.g., ADR-OS-032) are up-to-date and enforced.

This section was expanded in response to issue_assumptions.txt to surface implicit assumptions and improve framework compliance.

Frameworks/Models Applied

This ADR applies the following canonical models and frameworks (per ADR-OS-032):

Theory of Constraints (ToC) v1.0

- **Compliance Proof:** Two-tiered issue management system identifies bottlenecks at initiative and global levels, enabling systematic constraint resolution.
- **Self-Critique:** Maintaining consistency across three files for every issue update could become a source of synchronization bugs.

First-Class Citizen Principle v1.0

- **Compliance Proof:** Issues are treated as first-class entities with dedicated file structure (issue_.txt) rather than simple log entries.
- **Self-Critique:** File-based approach assumes sufficient for OS needs; may not scale to enterprise-level issue volumes.

Hierarchical Organization v1.0

- **Compliance Proof:** Three-tier structure (individual issues -> initiative summaries -> global summary) provides scalable organization.
- **Self-Critique:** Bi-directional linking between issues, plans, and artifacts adds complexity to update logic.

Traceability v1.0

- **Compliance Proof:** Issues linked to their context (plans, artifacts) providing complete audit trail and relationship mapping.
- **Self-Critique:** If issue schema is too rigid, may not capture all necessary nuances of problems.

Performance Optimization v1.0

- **Compliance Proof:** Summary files provide performance gains for retrieving issue overviews without parsing all individual issue files.
- **Self-Critique:** Structure assumes UI will be primary consumer; less convenient for command-line inspection.

Assumption Surfacing v1.0

- **Compliance Proof:** Explicit assumptions section with checkboxes for validation about file-based sufficiency, overhead acceptance, and JSON representation.
- **Self-Critique:** Only three assumptions listed; issue management likely has more implicit assumptions about file system performance and concurrency.

Scalability v1.0

- **Compliance Proof:** Hierarchical summarization prevents single monolithic issue file that would become contention point and slow to parse.
- **Self-Critique:** Three-file consistency requirement increases logical complexity for every issue update.

Decision

Decision:

> We will implement a two-tiered, file-based issue management system. Each issue will be a distinct **`issue_<g>.txt`** file. These will be indexed by an **`initiative_issues_summary_*.txt`** at the initiative level, and all initiative summaries will be indexed by a single **`global_issues_summary.txt`** at the root.

Confidence: High

Rationale

1. **Atomicity & Traceability**
2. Self-critique: The bi-directional linking between issues, plans, and artifacts adds complexity to the update logic.
3. Confidence: High
4. **Rich Context**
5. Self-critique: If the issue schema is too rigid, it may not capture all necessary nuances of a problem.
6. Confidence: Medium
7. **Scalability & Performance**
8. Self-critique: Maintaining consistency across three files for every issue update could be a source of synchronization bugs.
9. Confidence: High
10. **Supports Supervisor/Cockpit UI**
11. Self-critique: This structure assumes a UI will be the primary consumer; it might be less convenient for simple command-line inspection.
12. Confidence: High

Alternatives Considered

1. **External Issue Tracker (e.g., GitHub Issues)**: Rejected for the core OS to maintain self-containment and a unified data model, though an external synchronization agent could be a future feature.
2. Confidence: High
3. **Single Monolithic Issue File**: Rejected as it would become a massive point of contention and would be slow to parse and update as the project grows.
4. Confidence: High

Consequences

- **Positive:** Provides a highly structured and robust system for tracking project issues. Clear separation of detailed records from high-level summaries. Deeply integrates issue tracking with the development lifecycle.
- **Negative:** Requires the OS to maintain data consistency across three levels of files (individual, initiative summary, global summary) for every issue update, increasing logical complexity.

Clarifying Questions

- What is the precise schema for `issue_<g>.txt`, and how is it versioned and evolved as requirements change?
- What is the trigger and mechanism for synchronizing the summary files? Is it immediate, batched, or event-driven, and how are failures detected and recovered?
- How are cross-initiative dependencies between issues tracked, visualized, and resolved?
- What audit and recovery mechanisms exist for desynchronization or corruption between individual, initiative, and global summaries?
- How does the system scale and maintain performance as the number of issues grows to enterprise levels?

This template integrates explicit assumption-surfacing, confidence indicators, self-critiques, and clarifying questions as per ADR-OS-021.

■# ANNOTATION_BLOCK_START { "artifact_annotation_header": { "artifact_id_of_host": "adr_os_010_md", "g_annotation_created": 22, "version_tag_of_host_at_annotation": "1.2.0" }, "payload": { "description": "Retrofitted to comply with ADR-OS-032: Canonical Models and Frameworks Registry & Enforcement.", "artifact_type": "DOCUMENTATION", "purpose_statement": "To ensure framework compliance and improve architectural decision clarity.", "authors_and_contributors": [{ "g_contribution": 22, "identifier": "Hybrid_AI_OS" }, { "g_contribution": 4, "identifier": "Framework_Compliance_Retrofit" }], "internal_dependencies": ["adr_os_template_md", "adr_os_032_md"], "linked_issue_ids": [] } }

ANNOTATION_BLOCK_END

ADR-OS-010: Constraint Management & Locking Strategy

- **Status**: Proposed
- **Date**: 2025-06-09
- **Deciders**: [List of decision-makers]
- **Reviewed By**: [List of reviewers]

Context

Autonomous AI agents can "drift" or "forget" critical project-level constraints over time. To ensure long-term project integrity, a mechanism is needed to make these constraints durable and explicitly non-mutable by standard agent operations.

Assumptions

- [ ] AI "drift" is a real and significant risk to long-term project stability.
- [ ] A data-centric locking mechanism is an effective way to mitigate this risk.
- [ ] The OS and its agents can be reliably programmed to respect these lock fields.
- [ ] The semantics and enforcement of all `_locked*` fields are clearly defined and versioned.
- [ ] The system can detect and recover from failed or partial lock/unlock operations.
- [ ] The override and audit trail process is robust and tamper-evident.
- [ ] All compliance requirements from referenced ADRs (e.g., ADR-OS-023, ADR-OS-028) are up-to-date and enforced.

This section was expanded in response to issue_assumptions.txt to surface implicit assumptions and improve framework compliance.

Frameworks/Models Applied

This ADR applies the following canonical models and frameworks (per ADR-OS-032):

Distributed Systems Principles v1.0

- **Compliance Proof:** "Distributed Systems Implications" section addresses idempotency and partition tolerance for lock operations.
- **Self-Critique:** Lock checking in distributed environment needs more comprehensive failure mode analysis.

Data-Centric Security v1.0

- **Compliance Proof:** Locking mechanism places constraints directly on data rather than relying on agent-level access control.
- **Self-Critique:** Over-locking could lead to excessive rigidity requiring frequent human intervention.

Fail-Safe Design v1.0

- **Compliance Proof:** Agents MUST halt and log BLOCKER issue when encountering locked constraints, preventing unauthorized modifications.
- **Self-Critique:** Override process requiring new Request could become bottleneck if not managed efficiently.

Assumption Surfacing v1.0

- **Compliance Proof:** Explicit assumptions section about AI drift risk, locking effectiveness, and agent compliance reliability.
- **Self-Critique:** Only three assumptions listed; constraint management likely has more implicit assumptions about lock semantics.

Architectural Integrity v1.0

- **Compliance Proof:** Granular boolean lock fields protect foundational decisions from inadvertent agent changes.
- **Self-Critique:** Meaning of different lock types adds cognitive load for developers and agents.

Audit Trail v1.0

- **Compliance Proof:** Override process creates formal, auditable trail for high-stakes constraint changes.
- **Self-Critique:** Audit trail quality depends on proper logging of override decisions and rationale.

Proximity Principle v1.0

- **Compliance Proof:** Lock fields placed directly within schemas near the data they protect for contextual awareness.
- **Self-Critique:** Decoupled constraints in global file would be harder for agents to discover contextually.

Decision

Decision:

We will implement a granular, data-centric locking strategy using specific boolean fields (e.g., `_fieldname_locked`, `_locked_entry_definition`) directly within the schemas of OS Control Files and `EmbeddedAnnotationBlocks`. An agent encountering a `true` lock on an item it must modify **MUST** halt, log a `BLOCKER` issue, and set its task to `BLOCKED`.

Distributed Systems Implications

Operations that set or check these locks must be robust in a distributed environment.

- **Idempotency (ADR-OS-023):** Any OS action that sets or removes a lock (`_locked*` flag) MUST be idempotent. A second attempt to apply the same lock should result in success, not an error.
- **Partition Tolerance (ADR-OS-028):** Checking a lock is a read operation and can proceed in any partition. However, changing a lock is a write operation. Since OS Control Files are CP systems, an attempt to change a lock in a minority partition will fail until the partition heals, preventing inconsistent lock states across the system.

Confidence: High

Rationale

1. **Enforces Architectural Integrity**
2. Self-critique: Over-locking could lead to excessive rigidity and require frequent human intervention for minor, legitimate changes.
3. Confidence: High
4. **Durable, Proximate Constraints**
5. Self-critique: The meaning of different lock types (e.g., `_list_immutable` vs. `_locked_entry_definition`) adds cognitive load for developers and agents.
6. Confidence: High
7. **Clear Override Path**
8. Self-critique: The override process, requiring a new `Request` and explicit authorization, could become a bottleneck if not managed efficiently.
9. Confidence: Medium

Alternatives Considered

1. **Role-Based Access Control (RBAC)**: Considered complementary, not mutually exclusive. RBAC is agent-centric; this is data-centric. Combining them could be a future enhancement.
2. Confidence: High
3. **Constraints in a Single Global File**: Rejected because it decouples the constraint from the data it protects, making it harder for an agent to be contextually aware of the lock.
4. Confidence: High

Consequences

- **Positive:** Provides strong guardrails for autonomous agents. Makes project constraints explicit and machine-readable. Creates a formal, auditable process for high-stakes changes.
- **Negative:** Adds complexity to schemas and agent logic, as they must always check for locks. Could introduce rigidity if locks are not applied judiciously.

Clarifying Questions

- What is the definitive list of all `_locked*` field types and their precise meanings, and how is this list versioned and maintained?
- What is the process for a supervisor to review, approve, and audit a lock override request, and how is this process made tamper-evident?
- How does this locking mechanism apply to binary or non-text-based artifacts, and what are the limitations?
- How are distributed lock state, partition healing, and lock conflict resolution handled in multi-agent or partitioned environments?
- What is the process for evolving the lock schema or semantics as new constraint types are identified?

This template integrates explicit assumption-surfacing, confidence indicators, self-critiques, and clarifying questions as per ADR-OS-021.

■# ANNOTATION_BLOCK_START { "artifact_annotation_header": { "artifact_id_of_host": "adr_os_011_md", "g_annotation_created": 25, "version_tag_of_host_at_annotation": "1.2.0" }, "payload": { "description": "Retrofitted to comply with ADR-OS-032: Canonical Models and Frameworks Registry & Enforcement.", "artifact_type": "DOCUMENTATION", "purpose_statement": "To ensure framework compliance and improve architectural decision clarity.", "authors_and_contributors": [{ "g_contribution": 25, "identifier": "Hybrid_AI_OS" }, { "g_contribution": 4, "identifier": "Framework_Compliance_Retrofit" }], "internal_dependencies": ["adr_os_template_md", "adr_os_032_md"], "linked_issue_ids": [] } }

ANNOTATION_BLOCK_END

ADR-OS-011: Task Failure Handling & Remediation

- **Status**: Proposed
- **Date**: 2025-06-09
- **Deciders**: [List of decision-makers]
- **Reviewed By**: [List of reviewers]

Context

Tasks will inevitably fail. A robust autonomous system must have a predictable, safe, and traceable process for handling these failures without corrupting state or halting progress indefinitely. A simple "crash and stop" approach is insufficient.

Assumptions

- [ ] Automated, stateful rollback of file system changes is too complex and risky to be reliable.
- [ ] Failures, once they exceed a simple retry threshold, require formal tracking as `Issues`.
- [ ] Human or supervisor-level intelligence is necessary to diagnose and plan remediation for non-trivial failures.
- [ ] The retry policy and escalation thresholds are clearly defined and versioned.
- [ ] The system can distinguish between task failure and unreachable dependencies in distributed environments.
- [ ] The human_attention_queue and remediation process are auditable and tamper-evident.
- [ ] All compliance requirements from referenced ADRs (e.g., ADR-OS-023, ADR-OS-028, ADR-OS-029) are up-to-date and enforced.

This section was expanded in response to issue_assumptions.txt to surface implicit assumptions and improve framework compliance.

Frameworks/Models Applied

This ADR applies the following canonical models and frameworks (per ADR-OS-032):

Fail-Safe Design v1.0

- **Compliance Proof:** "Log, Isolate, and Remediate" strategy ensures system fails safely without corrupting state or halting indefinitely.
- **Self-Critique:** Approach is intentionally heavyweight; simple task failures require creating new issue and plan.

Distributed Systems Principles v1.0

- **Compliance Proof:** Addresses idempotency, partition tolerance, and observability for failure handling in distributed environments.
- **Self-Critique:** Distinguishing between task failure and unreachable dependency needs clearer implementation guidelines.

Escalation Management v1.0

- **Compliance Proof:** Formal escalation to human_attention_queue when failures exceed retry thresholds.
- **Self-Critique:** Human attention queue presentation mechanism needs definition.

Traceability v1.0

- **Compliance Proof:** Every failure creates Issue with complete context and trace_id linkage for debugging.
- **Self-Critique:** Relies on quality of initial failure report; poor failure_details make diagnosis difficult.

Assumption Surfacing v1.0

- **Compliance Proof:** Explicit assumptions about rollback complexity, formal tracking needs, and human intelligence requirements.
- **Self-Critique:** Only three assumptions listed; failure handling likely has more implicit assumptions about retry policies and state management.

Decision

Decision:

We will adopt a **"Log, Isolate, and Remediate"** strategy. On task failure beyond retries, the agent MUST: 1) Update the task status to `FAILED` with details. 2) Log a new `Issue` containing all failure context. 3) Halt the current plan and escalate to the `human_attention_queue`. Remediation will occur via a new, explicit `REMEDIATION`-type `Execution Plan`.

Distributed Systems Implications

The failure handling process must be robust against the challenges of a distributed environment.

- **Idempotency & Retries (ADR-OS-023):** The initial, pre-`FAILED` state retry attempts mentioned in "Alternatives Considered" MUST adhere to the universal retry policy, using exponential backoff and circuit breakers to avoid retry storms. The task execution itself must be idempotent to ensure retries are safe.
- **Partition Tolerance (ADR-OS-028):** In a network partition, an agent must be able to distinguish between task failure and an unreachable dependency. If a required service is in a separate, unreachable partition, the task should be marked `BLOCKED`, not `FAILED`, to await partition healing.
- **Observability (ADR-OS-029):** Every task failure, retry attempt, and state change (`PENDING` -> `ACTIVE` -> `FAILED`) MUST be captured as events within a distributed trace. The resulting `Issue` must be linked to the `trace_id` of the failure for end-to-end debugging.

Confidence: High

Rationale

1. **Avoids Complex State Management**
2. Self-critique: This approach is intentionally heavyweight. The failure of even a simple task requires creating a new issue and plan, which could slow down development for trivial errors.
3. Confidence: High
4. **Traceability of Remediation**
5. Self-critique: It relies on the quality of the initial failure report. If the agent's `failure_details` are poor, the resulting `Issue` may be difficult to diagnose.
6. Confidence: High
7. **Leverages Existing Mechanisms**
8. Self-critique: This could lead to a proliferation of small, single-task `REMEDIATION` plans, cluttering the project history.
9. Confidence: High

Alternatives Considered

1. **Automated Artifact Rollback**: Rejected due to the high complexity of managing which files to revert and ensuring a consistent state across all related artifacts and OS Control Files.
2. Confidence: High
3. **Conversational Debugging Loop**: This is used *within* a task's retry attempts. The "Log, Isolate, Remediate" process is for when that inner loop fails, ensuring formal tracking.
4. Confidence: High

Consequences

- **Positive:** Highly robust and safe. Creates a complete, auditable history of all failures and their resolutions. Simple to implement as it builds on existing OS primitives.
- **Negative:** Slower recovery for simple failures. A typo that fails a task might require a new plan to fix, which can feel heavyweight.

Clarifying Questions

- What is the standard retry and escalation policy before a task is officially marked as `FAILED`, and how is this policy versioned and enforced?
- How does the `human_attention_queue` get presented to the supervisor, and what is the audit trail for human interventions?
- Can a `REMEDIATION` plan itself fail, and if so, does it trigger a new, nested remediation? How is remediation recursion managed?
- How does the system distinguish between task failure and unreachable dependencies in distributed or partitioned environments?
- What mechanisms are in place to audit, validate, and roll back failure and remediation events, especially in the presence of partial or conflicting updates?

This template integrates explicit assumption-surfacing, confidence indicators, self-critiques, and clarifying questions as per ADR-OS-021.

■# ANNOTATION_BLOCK_START { "artifact_annotation_header": { "artifact_id_of_host": "adr_os_012_md", "g_annotation_created": 12, "version_tag_of_host_at_annotation": "1.2.0" }, "payload": { "description": "Retrofitted to comply with ADR-OS-032: Canonical Models and Frameworks Registry & Enforcement.", "artifact_type": "DOCUMENTATION", "purpose_statement": "To ensure framework compliance and improve architectural decision clarity.", "authors_and_contributors": [{ "g_contribution": 12, "identifier": "Hybrid_AI_OS" }, { "g_contribution": 4, "identifier": "Framework_Compliance_Retrofit" }], "internal_dependencies": ["adr_os_template_md", "adr_os_032_md"], "linked_issue_ids": [] } }

ANNOTATION_BLOCK_END

ADR-OS-012: Dynamic Agent Management

- **Status**: Proposed
- **Date**: 2024-05-31
- **Deciders**: [List of decision-makers]
- **Reviewed By**: [List of reviewers]

Context

For the Hybrid_AI_OS to evolve into a truly autonomous and adaptable system, especially one supporting multiple specialized AI agents ("vessels"), the management of these agents cannot be a static, hardcoded configuration. The system requires a dynamic way to register, configure, and monitor the agent personas available for executing tasks. This mechanism needs to be robust, transparent, and controllable via OS-level operations, aligning with the potential for a "cockpit" UI for human supervision.

Assumptions

- [ ] The "Index + Individual File" pattern is performant enough for the expected number of agents.
- [ ] OS-level actions provide a sufficient security model for managing agent lifecycles.
- [ ] The agent registry and cards are versioned and auditable for all changes.
- [ ] The system can detect and recover from registry or card corruption or partial updates.
- [ ] The agent management process is robust against concurrency and race conditions.
- [ ] All compliance requirements from referenced ADRs (e.g., ADR-OS-032) are up-to-date and enforced.

Frameworks/Models Applied

This ADR applies the following canonical models and frameworks (per ADR-OS-032):

Distributed Systems Principles v1.0

- **Compliance Proof:** "Distributed Systems Implications" section addresses security, topology & health, partition tolerance, and observability for agent registry.
- **Self-Critique:** Agent registry as CP system may impact availability during partitions; needs careful consideration.

Scalability v1.0

- **Compliance Proof:** "Index + Individual File" pattern prevents single monolithic registry file that would become contention point.
- **Self-Critique:** Index file could still become contention point if many processes check it frequently.

Dynamic Configuration v1.0

- **Compliance Proof:** Runtime agent management without restarting OS or manual config file editing.
- **Self-Critique:** Requires high-privilege "Supervisor" agent, creating potential single point of failure.

Service Discovery v1.0

- **Compliance Proof:** Agent registry serves as source of truth for service discovery with self-registration and health propagation.
- **Self-Critique:** Polling superseded by subscription model but implementation complexity increases.

Assumption Surfacing v1.0

- **Compliance Proof:** Explicit assumptions about Index pattern performance, OS-level security sufficiency, and agent lifecycle management.
- **Self-Critique:** Only three assumptions listed; dynamic agent management likely has more implicit assumptions about concurrency and failure modes.

Decision

Decision:

We will implement a dynamic, file-based agent management system using the established **"Index + Individual File"** pattern. This system will consist of two new types of OS Control Files:

1. **`os_root/agent_registry.txt` (The Index):** A single, top-level file that serves as an index of all known and available agent personas. It will map `persona_id`s to the file path of their detailed "Agent Card" and mirror key status information for quick lookups.
2. **`os_root/agents/persona_<id>.txt` (The Agent Card):** Individual, detailed files for each agent persona. Each "Agent Card" will contain the complete configuration for that role, including its capabilities, currently assigned AI model, model-specific parameters, operational status, and a history log.

Rationale

Scalability & Performance

- Self-critique: The index file (`agent_registry.txt`) could still become a contention point if many processes need to check it frequently.
- Confidence: High

Dynamic & Controllable

- Self-critique: Requires a high-privilege "Supervisor" agent, creating a potential single point of failure or control bottleneck.
- Confidence: High

Consistency with OS Design Patterns

- Self-critique: Over-reliance on one pattern might lead to ignoring a better-suited pattern for a specific problem in the future.
- Confidence: High

Rich, Granular Configuration

- Self-critique: Could lead to overly complex Agent Cards that are difficult to manage or audit manually.
- Confidence: High

Alternatives Considered

Static Configuration (`haios.config.json`): Placing the agent roster in the main config file. Rejected as too rigid, not suitable for dynamic runtime changes, and less scalable for detailed agent configurations.

- Confidence: High

Single Monolithic Registry File: A single file containing all details for all agents. Rejected because it could become very large and unwieldy to parse and update, violating the principle of isolating detailed records.

- Confidence: High

Consequences

- **Positive:** Provides a flexible and scalable system for managing a fleet of diverse AI agents. Enables runtime configuration of agents without restarting the OS or manually editing project configuration files. Keeps the core `haios.config.json` clean and focused on static, foundational project settings. Creates a clear, auditable trail for changes to agent configurations via updates to these OS Control Files.
- **Negative:** Introduces two new OS Control File schemas that need to be defined and managed. Requires the OS to have a privileged "manager" capability to modify the agent registry and cards.

Clarifying Questions

- What is the process for recovering from a corrupted or partially updated agent registry or Agent Card file?
- How are concurrent updates to the agent registry and Agent Cards coordinated to prevent race conditions or data loss?
- What mechanisms are in place to audit and roll back unauthorized or erroneous changes to agent configurations?
- How does the system handle the addition or removal of agent personas at runtime, especially under high load or during network partitions?
- What are the escalation and notification procedures if the "Supervisor" agent fails or becomes unavailable?

This template integrates explicit assumption-surfacing, confidence indicators, self-critiques, and clarifying questions as per ADR-OS-021.

ANNOTATION_BLOCK_START

{ "artifact_annotation_header": { "artifact_id_of_host": "adr_os_013_md", "g_annotation_created": 13, "version_tag_of_host_at_annotation": "1.2.0" }, "payload": { "description": "Retrofitted to comply with ADR-OS-032: Canonical Models and Frameworks Registry & Enforcement.", "artifact_type": "DOCUMENTATION", "purpose_statement": "To ensure framework compliance and improve architectural decision clarity.", "authors_and_contributors": [{ "g_contribution": 13, "identifier": "Hybrid_AI_OS" }, { "g_contribution": 4, "identifier": "Framework_Compliance_Retrofit" }], "internal_dependencies": ["adr_os_template_md", "adr_os_032_md"], "linked_issue_ids": [] } }

ANNOTATION_BLOCK_END

ADR-OS-013: Pre-Execution Readiness Checks

- **Status**: Proposed
- **Date**: 2024-05-31
- **Deciders**: [List of decision-makers]
- **Reviewed By**: [List of reviewers]

Context

Executing a task within an `Execution Plan` can be computationally expensive and may have side effects. Attempting to execute a task when its environmental prerequisites (e.g., required tools, configuration files, dependencies, input artifacts) are missing or invalid leads to guaranteed failure, wasted resources, and potentially corrupted state. A proactive mechanism is needed to verify that a task is ready to be executed *before* its primary work commences.

Assumptions

- [ ] The executing agent has the necessary permissions to check for all required prerequisites on the filesystem.
- [ ] The cost of performing the readiness check is significantly lower than the cost of a failed task execution.
- [ ] The information in `global_registry_map.txt` is up-to-date and accurate.
- [ ] The readiness check logic is robust against transient and distributed environment changes.
- [ ] The system can detect and recover from partial or failed readiness checks.
- [ ] The readiness check process is idempotent and auditable.
- [ ] All compliance requirements from referenced ADRs (e.g., ADR-OS-023, ADR-OS-029) are up-to-date and enforced.

This section was expanded in response to issue_assumptions.txt to surface implicit assumptions and improve framework compliance.

Frameworks/Models Applied

This ADR applies the following canonical models and frameworks (per ADR-OS-032):

Fail-Fast Principle v1.0

- **Compliance Proof:** Readiness Check as mandatory first step prevents expensive task failures by catching environmental issues early.
- **Self-Critique:** For simple, low-risk tasks, formal check overhead might slightly outweigh benefit.

KISS (Keep It Simple, Stupid) v1.0

- **Compliance Proof:** Simple pass/fail readiness check with clear BLOCKED status and issue creation for failures.
- **Self-Critique:** Readiness agent itself might require significant context, shifting complexity rather than eliminating it.

Distributed Systems Principles v1.0

- **Compliance Proof:** "Distributed Systems Implications" section addresses idempotency and observability for readiness checks.
- **Self-Critique:** Missing explicit handling of network dependencies and service availability checks.

Assumption Surfacing v1.0

- **Compliance Proof:** Explicit assumptions about agent permissions, check costs, and registry accuracy.
- **Self-Critique:** Only three assumptions listed; readiness checking likely has more implicit assumptions about environment stability.

Separation of Concerns v1.0

- **Compliance Proof:** Clear separation between environmental verification (readiness) and task execution (primary work).
- **Self-Critique:** Quality of BLOCKER issue depends on quality of checking logic; poor check could lead to misleading issue.

Decision

Decision:

We will mandate that a **"Readiness Check"** is the first step in the execution of any task within the `CONSTRUCT` phase.

This check is a formal, non-negotiable procedure where the executing agent verifies the existence, accessibility, and basic validity of all environmental prerequisites for the task. The prerequisites are determined by analyzing the task's `inputs` and, crucially, its `context_loading_instructions`.

If the Readiness Check fails: 1. The agent **MUST NOT** proceed with the main execution of the task. 2. The agent **MUST** set the task's `status` to `BLOCKED`. 3. The agent **MUST** log a new `Issue` of type `BLOCKER`, detailing precisely which prerequisites are missing or invalid. 4. The agent's work on the task ceases, and the OS waits for a `REMEDIATION` plan to fix the environment before the original task can be retried.

The output of a successful Readiness Check can be a simple internal "pass" or, for complex tasks like test execution, it can be a formal **`Readiness Assessment`** artifact (.md) detailing the status of all components.

Implementation within the `CONSTRUCT` Phase

The `AI_Execute_Next_Viable_Task_...` core action will be updated to include this logic:

```
1. Select viable task. 2. Update state.ct_id to this task. 3. --> **PERFORM READINESS CHECK for the task:** a.
Parse task.inputs and task.context_loading_instructions. b. Verify existence of all referenced artifact IDs in
`global_registry_map.txt`. c. Verify existence of required files/tools on the file system (e.g., `.env.local`,
`k6` binary if needed). d. Verify project dependencies are installed (e.g., `node_modules` exists if
`package.json` is a context). e. **IF CHECK FAILS:** i. Update task status to `BLOCKED`. ii. Create a `BLOCKER`
issue with details. iii. Update state.st to `BLOCK_INPUT` (awaiting remediation plan). iv. Cease processing this
task. f. **IF CHECK PASSES (and is complex enough to warrant it):** i. Optionally generate a
`readiness_assessment_gX.md` artifact. 4. --> **EXECUTE PRIMARY TASK WORK (only if Readiness Check passed).** 5.
... (Continue with normal task completion logic).
```

Distributed Systems Implications

The Readiness Check procedure must be designed for a distributed environment.

- **Idempotency (ADR-OS-023):** The entire Readiness Check process MUST be idempotent. Running the check multiple times on an unchanged environment must yield the same result without causing side effects. This is critical for retrying a `BLOCKED` task after a remediation plan has run.
- **Observability (ADR-OS-029):** The start, pass, and fail outcomes of every Readiness Check MUST be recorded as distinct events within the task's distributed trace. A failed check's event must contain structured data about which specific prerequisite failed, enabling automated analysis of systemic environmental issues.

Confidence: High

Rationale

Fail-Fast Principle
- Self-critique: For some very simple, low-risk tasks, the overhead of a formal check might slightly outweigh the benefit.
- Confidence: High

Reduces "Context Overwhelm"
- Self-critique: The "readiness" agent itself might require significant context to perform its checks correctly, shifting complexity rather than eliminating it.
- Confidence: Medium

Explicit Error Reporting
- Self-critique: The quality of the `BLOCKER` issue is dependent on the quality of the checking logic; a poor check could lead to a misleading issue.
- Confidence: High

Improved System Stability
- Self-critique: This doesn't prevent logical errors within the task itself, only environmental ones. The system can still enter an inconsistent state due to flawed task logic.
- Confidence: High

Alternatives Considered

No Explicit Check (Reactive Failure): Letting tasks fail mid-execution and then trying to parse the error logs. Rejected as messy, unreliable, and provides poor-quality error reporting.

- Confidence: High

A Single Validation Phase for Environment: Having one big check at the beginning of an `Execution Plan`. Rejected because different tasks within the same plan may have different environmental needs, making a single upfront check insufficient or overly broad. A per-task check is more precise.

- Confidence: High

Consequences

- **Positive:** Dramatically increases the reliability and robustness of task execution. Makes debugging environmental setup issues straightforward via the generated `BLOCKER` issues. Fits perfectly with the "separation of concerns" principle for agent actions.
- **Negative:** Adds a small amount of overhead at the beginning of every task for the verification step. This is a highly worthwhile trade-off for the increased reliability.

Clarifying Questions

- How are complex dependencies (e.g., service availability over a network) handled by this check, and how is this coordinated with ADR-OS-026 and ADR-OS-028?
- What is the retry policy for a task that fails its readiness check, and how are repeated failures escalated or audited?
- How is readiness state and prerequisite verification kept consistent and up-to-date in distributed or rapidly changing environments?
- What mechanisms are in place to audit, trace, and analyze readiness check failures for systemic issues?
- How is the readiness check logic versioned and evolved as new types of prerequisites or environmental requirements are identified?

This template integrates explicit assumption-surfacing, confidence indicators, self-critiques, and clarifying questions as per ADR-OS-021.

■# ANNOTATION_BLOCK_START { "artifact_annotation_header": { "artifact_id_of_host": "adr_os_014_md", "g_annotation_created": 14, "version_tag_of_host_at_annotation": "1.2.0" }, "payload": { "description": "Retrofitted to comply with ADR-OS-032: Canonical Models and Frameworks Registry & Enforcement.", "artifact_type": "DOCUMENTATION", "purpose_statement": "To ensure framework compliance and improve architectural decision clarity.", "authors_and_contributors": [{ "g_contribution": 14, "identifier": "Hybrid_AI_OS" }, { "g_contribution": 4, "identifier": "Framework_Compliance_Retrofit" }{ "g_contribution": 4, "identifier": "Framework_Compliance_Retrofit" }], "internal_dependencies": ["adr_os_template_md", "adr_os_032_md""adr_os_032_md"], "linked_issue_ids": [] } }

ANNOTATION_BLOCK_END

ADR-OS-014: Project Guidelines Artifact

- **Status**: Proposed
- **Date**: 2024-05-31
- **Deciders**: [List of decision-makers]
- **Reviewed By**: [List of reviewers]

Context

To ensure consistency and quality, and to combat AI "drift" or "forgetting" of critical project standards, there is a need for a durable, central repository of project-wide rules, conventions, and procedures. These guidelines must be easily accessible and referenceable by all AI agents during planning and execution phases.

Assumptions

- [ ] Agents can reliably parse and adhere to instructions provided in Markdown guideline files.
- [ ] The `global_registry_map.txt` provides a stable way to reference and retrieve guideline artifacts.
- [ ] The overhead of managing and versioning guideline artifacts is less than the cost of inconsistent agent behavior.
- [ ] The guidelines management process is robust against outdated or conflicting rules.
- [ ] The system can detect and recover from missing or corrupted guideline artifacts.
- [ ] All guideline artifacts are versioned and auditable for changes.
- [ ] All compliance requirements from referenced ADRs (e.g., ADR-OS-032) are up-to-date and enforced.

This section was expanded in response to issue_assumptions.txt to surface implicit assumptions and improve framework compliance.

Frameworks/Models Applied

This ADR applies the following canonical models and frameworks (per ADR-OS-032):

DRY (Don't Repeat Yourself) v1.0

- **Compliance Proof:** Central repository eliminates duplication of project standards across multiple locations; single source of truth for guidelines.
- **Self-Critique:** Guidelines becoming outdated could enforce incorrect behavior; requires regular review and maintenance process.

Single Source of Truth v1.0

- **Compliance Proof:** Project Guidelines artifact store serves as canonical source for all project-wide rules, conventions, and procedures.
- **Self-Critique:** Malicious or poorly designed plan could intentionally omit guideline context, bypassing the control.

Assumption Surfacing v1.0

- **Compliance Proof:** Explicit assumptions about agent parsing capabilities, registry stability, and overhead/benefit trade-offs.
- **Self-Critique:** Only three assumptions listed; guidelines management likely has more implicit assumptions about agent compliance and maintenance processes.

Traceability v1.0

- **Compliance Proof:** Guidelines are version-controlled Project Artifacts with EmbeddedAnnotationBlocks providing complete audit trail.
- **Self-Critique:** Audit trail only proves guideline was loaded, not that agent perfectly adhered to it; validation steps still critical.

Human-Centered Design v1.0

- **Compliance Proof:** Guidelines designed to combat AI drift and provide human-readable standards for project oversight.
- **Self-Critique:** Different underlying LLMs might interpret same natural language guidelines differently, leading to subtle inconsistencies.

First-Class Citizen Principle v1.0

- **Compliance Proof:** Guidelines treated as first-class Project Artifacts with full lifecycle support, versioning, and registry integration.
- **Self-Critique:** As more guidelines are added, managing dependencies and ensuring they don't contradict each other becomes more complex.

Decision

Decision:

We will establish the concept of a **`Project Guidelines` artifact store**, which will be a designated collection of Markdown documents within the project workspace (e.g., located at `project_workspace/docs/guidelines/`). These are version-controlled Project Artifacts, complete with `EmbeddedAnnotationBlocks`.

Key guideline artifacts will be created to house specific types of standards. A crucial initial example is the **`testing_guidelines.md`** artifact, which will contain procedural checklists, such as the **"Bias Prevention Checklist"** for validating test results.

All relevant `Execution Plan` tasks (e.g., development, testing, validation, critique) **MUST** include a `context_loading_instructions` entry that explicitly loads the appropriate guideline artifacts. The AI agent executing the task is then required to adhere to and/or apply the procedures outlined in the loaded guidelines.

Implementation Example

1. A `testing_guidelines.md` artifact is created in `project_workspace/docs/guidelines/` and registered in `global_registry_map.txt`. Its content includes the `Bias Prevention Checklist`.
2. An `Execution Plan` of type `CRITICAL_ASSESSMENT` is blueprinted.
3. A task within this plan will have the following `context_loading_instructions`: `json { "context_id": "ctx_bias_checklist", "description": "Load the mandatory Bias Prevention Checklist to guide the critical assessment of test results.", "load_type": "ARTIFACT_CONTENT", "source_reference": { "type": "ARTIFACT_ID", "value": "testing_guidelines_md_artifact_id" }, "priority": "CRITICAL", "is_input_for_prompt": true }`
4. The agent executing the task receives the content of the checklist as part of its prompt and must structure its response and actions according to the checklist's items.

Confidence: High

Rationale

Durable Memory
- Self-critique: If guidelines become outdated, they could enforce incorrect behavior. This requires a process for regular review and maintenance.
- Confidence: High

Explicit Instruction
- Self-critique: A malicious or poorly designed plan could intentionally omit the guideline context, bypassing the control. This relies on robust plan validation.
- Confidence: High

Consistency
- Self-critique: Different underlying LLMs might interpret the same natural language guidelines differently, leading to subtle inconsistencies.
- Confidence: Medium

Auditable Procedures
- Self-critique: The audit trail only proves the guideline was loaded, not that the agent perfectly adhered to it. Validation steps are still critical.
- Confidence: High

Evolvability
- Self-critique: As more guidelines are added, managing their dependencies and ensuring they don't contradict each other becomes more complex.
- Confidence: Medium

Alternatives Considered

Embedding all rules in system prompts: Rejected as it's not version-controlled, not easily auditable per-project, and can quickly bloat the base system prompt.
- Confidence: High

Relying on AI's general knowledge: Rejected as it leads to inconsistent application of standards and "AI drift."
- Confidence: High

Consequences

- **Positive:** Significantly improves the reliability and consistency of AI agent actions. Creates a formal feedback loop where operational learning can be captured and standardized. Makes the AI's adherence to standards explicit and verifiable.
- **Negative:** Adds an extra layer of artifacts to manage. Requires discipline in the `BLUEPRINT` phase to ensure tasks correctly reference the guideline artifacts.

Clarifying Questions

- What is the process for proposing and ratifying a new project guideline?
- How are conflicting rules between different guideline documents resolved?
- How is guideline versioning managed, and what is the process for deprecating or updating outdated guidelines?
- What mechanisms are in place to ensure agents consistently load and apply the correct guidelines for each task?
- How are guideline artifacts audited for compliance, and what is the escalation process for detected violations?

This template integrates explicit assumption-surfacing, confidence indicators, self-critiques, and clarifying questions as per ADR-OS-021.

■# ANNOTATION_BLOCK_START { "artifact_annotation_header": { "artifact_id_of_host": "adr_os_015_md", "g_annotation_created": 15, "version_tag_of_host_at_annotation": "1.2.0" }, "payload": { "description": "Retrofitted to comply with ADR-OS-032: Canonical Models and Frameworks Registry & Enforcement.", "artifact_type": "DOCUMENTATION", "purpose_statement": "To ensure framework compliance and improve architectural decision clarity.", "authors_and_contributors": [{ "g_contribution": 15, "identifier": "Hybrid_AI_OS" }, { "g_contribution": 4, "identifier": "Framework_Compliance_Retrofit" }], "internal_dependencies": ["adr_os_template_md", "adr_os_032_md"], "linked_issue_ids": [] } }

ANNOTATION_BLOCK_END

ADR-OS-015: Precision Context Loading

- **Status**: Proposed
- **Date**: 2024-05-31
- **Deciders**: [List of decision-makers]
- **Reviewed By**: [List of reviewers]

Context

Modern Large Language Model (LLM) agents operate within a finite context window. Loading entire documents or artifacts as context for a task is highly inefficient and often counterproductive. It floods the context with irrelevant information, increases operational costs (token usage), and can lead to "context overwhelm" or "prompt contamination," where the agent is distracted or confused by non-essential data.

Assumptions

- [ ] The OS's orchestrator can be equipped with reliable logic to parse and slice artifacts based on lines or patterns.
- [ ] The structure of artifacts is consistent enough for pattern-based slicing to be effective.
- [ ] The agent creating the `Execution Plan` is capable of identifying the precise sections of context needed for a task.
- [ ] The system can detect and recover from pattern or line-based slicing failures.
- [ ] The context loading process is auditable and versioned for traceability.
- [ ] All compliance requirements from referenced ADRs (e.g., ADR-OS-032) are up-to-date and enforced.

Frameworks/Models Applied

This ADR applies the following canonical models and frameworks (per ADR-OS-032):

Performance Optimization v1.0

- **Compliance Proof:** Precision context loading maximizes context window value and reduces operational costs by loading only relevant data.
- **Self-Critique:** Pattern parsing overhead might introduce small delays; incorrectly specified patterns could cause silent failures.

KISS (Keep It Simple, Stupid) v1.0

- **Compliance Proof:** Simple line-based and pattern-based slicing mechanisms provide straightforward, understandable context loading.
- **Self-Critique:** Adding source_location_details increases complexity for plan creators; requires sophisticated planning agents.

Assumption Surfacing v1.0

- **Compliance Proof:** Explicit assumptions about OS parsing capabilities, artifact structure consistency, and agent planning sophistication.
- **Self-Critique:** Only three assumptions listed; precision loading likely has more implicit assumptions about pattern stability and artifact formats.

Separation of Concerns v1.0

- **Compliance Proof:** Clear separation between context identification (planning phase) and context loading (orchestrator execution).
- **Self-Critique:** Planning agent must understand both task requirements and artifact structure, potentially coupling concerns.

Fail-Safe Design v1.0

- **Compliance Proof:** Fallback behavior needed for missing patterns; system should gracefully handle slicing failures.
- **Self-Critique:** Current design doesn't specify fallback behavior for pattern matching failures, potential for silent context loss.

Cost-Effectiveness v1.0

- **Compliance Proof:** Precision loading directly reduces token usage and operational costs by eliminating irrelevant context.
- **Self-Critique:** Compute overhead for pattern matching might offset some token savings for very small contexts.

Decision

Decision:

We will implement a **Precision Context Loading** mechanism. The `context_loading_instructions` array within each `Task` object in an `Execution Plan` will be enhanced to support granular, targeted data retrieval.

This will be achieved by adding an optional `source_location_details` object to each context loading instruction. This object will allow the specification of: 1. **Line-based slicing:** Using `start_line` and `end_line` numbers to extract a specific portion of a text-based artifact. 2. **Pattern-based slicing:** Using `start_pattern` and `end_pattern` (string or regex) to dynamically locate and extract a specific, named section of a document (e.g., a specific chapter in a Markdown file).

The OS's orchestrator, when preparing a prompt for an agent, **MUST** process these instructions to construct a minimal, highly-relevant context payload.

Implementation Details

- The `Task Object` schema within `exec_plan_schema.md` will be updated to include the `source_location_details` sub-object in its `context_loading_instructions`.
- The agent orchestrator component of the OS engine will be responsible for implementing the file-reading and slicing logic based on these instructions before assembling the final prompt for an AI agent.
- An `Execution Plan` task for "onboarding" a new agent can now be a series of context-loading steps that walk the agent through key sections of project documentation, one chunk at a time.

Example Use Case

To instruct an agent to write a test based on a specific ADR section, the `context_loading_instructions` would not load the entire ADR. Instead, it would specify: `json { "description": "Load only the 'Decision' section of ADR-007.", "source_reference": { "type": "ARTIFACT_ID", "value": "adr_007_testing_lifecycle_gX" }, "source_location_details": { "start_pattern": "## Decision", "end_pattern": "## Rationale" } }`

Confidence: High

Rationale

Maximizes Context Window Value

- Self-critique: Incorrectly specified patterns or line numbers could lead to loading the wrong context or no context at all, causing silent failures.
- Confidence: High

Reduces Operational Costs

- Self-critique: The logic for slicing might add a small amount of compute overhead before the main LLM call.
- Confidence: High

Prevents "Context Overwhelm"

- Self-critique: An agent might still be overwhelmed if the *precisely loaded* context is itself extremely dense or complex.
- Confidence: High

Enhances Security & Data Hygiene

- Self-critique: This relies on the planning agent correctly identifying what is and isn't sensitive. A flawed plan could still load sensitive data.
- Confidence: Medium

Robustness to Change

- Self-critique: Pattern-based slicing is more robust but can still break if key headings or structural elements are changed without updating the corresponding execution plans.
- Confidence: Medium

Alternatives Considered

Whole-File Loading: The default, simple approach. Rejected as inefficient and unscalable for any non-trivial project with documentation.

- Confidence: High

Automated Summarization/RAG: Using an intermediate AI call to summarize a document before adding it to context. Rejected as it adds latency, cost, and a potential layer of information loss or misinterpretation. Direct, precise slicing is more reliable.

- Confidence: High

Consequences

- **Positive:** Drastically improves the efficiency and effectiveness of agent context. Lowers operational costs. Improves agent focus and output quality.
- **Negative:** Adds complexity to the context-loading logic within the OS orchestrator. The `BLUEPRINT` phase agent must be sophisticated enough to correctly identify and specify these precise sections when creating `Execution Plans`.

Clarifying Questions

- What is the fallback behavior if a `start_pattern` or `end_pattern` is not found in the source artifact?

- How will this mechanism handle non-text or binary artifacts?
- How are context loading failures detected, logged, and surfaced to the user or orchestrator?
- What validation or testing is in place to ensure that context slices are accurate and do not omit critical information?
- How does the system handle updates to artifact structure that may invalidate existing pattern-based context loading instructions?

This template integrates explicit assumption-surfacing, confidence indicators, self-critiques, and clarifying questions as per ADR-OS-021.

ANNOTATION_BLOCK_START

{ "artifact_annotation_header": { "artifact_id_of_host": "adr_os_016_md", "g_annotation_created": 16, "version_tag_of_host_at_annotation": "1.2.0" }, "payload": { "description": "Retrofitted to comply with ADR-OS-032: Canonical Models and Frameworks Registry & Enforcement.", "artifact_type": "DOCUMENTATION", "purpose_statement": "To ensure framework compliance and improve architectural decision clarity.", "authors_and_contributors": [{ "g_contribution": 16, "identifier": "Hybrid_AI_OS" }, { "g_contribution": 4, "identifier": "Framework_Compliance_Retrofit" }], "internal_dependencies": ["adr_os_template_md", "adr_os_032_md"], "linked_issue_ids": [] } }

ANNOTATION_BLOCK_END

ADR-OS-016: Live Execution Status Tracking

- **Status**: Proposed
- **Date**: 2025-06-09
- **Deciders**: [List of decision-makers]
- **Reviewed By**: [List of reviewers]

Context

As the OS executes tasks within an `exec_plan`, a significant amount of dynamic state is generated (task statuses, completion percentages, retry logs, test results). Placing this mutable, high-frequency data directly within the `exec_plan_<g>.txt` file violates the principle of plan immutability established in ADR-OS-010 (Constraint Locking). It forces agents to edit their own locked-down specifications, creating architectural tension and risking data corruption.

Assumptions

- [ ] The filesystem can handle the write frequency to the `exec_status_<g_plan>.txt` file without performance degradation.
- [ ] A separate status file provides better contention management than updating a single large plan file.
- [ ] The global `state.txt` can reliably point to the currently active status file.
- [ ] The status file schema is robust against partial writes and concurrent updates.
- [ ] The system can detect and recover from status/plan file desynchronization.
- [ ] All compliance requirements from referenced ADRs (e.g., ADR-OS-027, ADR-OS-029) are up-to-date and enforced.

Frameworks/Models Applied

This ADR applies the following canonical models and frameworks (per ADR-OS-032):

Fail-Safe Design v1.0

- **Compliance Proof:** Retry mechanism with exponential backoff provides graceful degradation for transient failures; maximum retry limits prevent infinite loops.
- **Self-Critique:** Retry logic complexity might introduce new failure modes; incorrect retry categorization could waste resources.

Distributed Systems Principles v1.0

- **Compliance Proof:** Exponential backoff with jitter prevents retry storms; addresses distributed system failure patterns and cascading failures.
- **Self-Critique:** Missing explicit handling of partial work completion and state consistency during retries.

Performance Optimization v1.0

- **Compliance Proof:** Intelligent retry reduces unnecessary work repetition; exponential backoff optimizes resource usage during failure recovery.
- **Self-Critique:** Retry overhead and delay might impact overall system throughput; aggressive retries could consume more resources than immediate failure.

Assumption Surfacing v1.0

- **Compliance Proof:** Explicit assumptions about failure categorization, backoff effectiveness, and retry limit configuration.
- **Self-Critique:** Only three assumptions listed; retry logic likely has more implicit assumptions about task idempotency and state management.

Traceability v1.0

- **Compliance Proof:** Retry attempts and outcomes are logged with task context for pattern analysis and debugging.
- **Self-Critique:** Trace data might not capture enough context to distinguish between different types of transient failures.

Escalation Management v1.0

- **Compliance Proof:** After maximum retries exceeded, clear escalation path to human intervention with detailed failure context.

- **Self-Critique:** Escalation effectiveness depends on quality of failure categorization and diagnostic information provided.

Decision

Decision:

We will architecturally separate the **immutable plan** from its **mutable execution status**. This will be achieved by:

1. **Stripping all dynamic/live state fields** from the `exec_plan_<g>.txt` schema. The `Execution Plan` becomes a purely definitional, immutable "work order" after its `DRAFT` phase is complete and its definitional fields are locked.
2. Introducing a new, dedicated OS Control File: **`exec_status_<g_plan>.txt`**. This file will be created alongside its corresponding `Execution Plan` and will serve as the single, mutable source of truth for all live execution progress, metrics, logs, and status updates for that plan.

Implementation Details

- **Creation:** When an `Execution Plan` is blueprinted, a corresponding `exec_status_*.txt` file will be created in the same directory, initialized with "0% complete" status.
- **Updates:** During the `CONSTRUCT` phase, executing agents write *only* to this `exec_status_*.txt` file to report progress, log retries, and record test outcomes.
- **Security:** Status updates can be "signed" with the `persona_id` of the writing agent and protected with a simple hash chain to ensure integrity.
- **Lifecycle:** After the `VALIDATE` phase for a plan is complete, the `exec_status_*.txt` file is effectively frozen. Its final state can be summarized in the `Validation Report`, and the file itself is then considered `ARCHIVED`.
- **State Reference:** The global `state.txt` will contain a `current_exec_status_id_ref` for convenient access to the live status of the currently active plan.

Distributed Systems Implications

The `exec_status_*.txt` file acts as a shared log and is subject to the following policies:

- **Asynchronicity (ADR-OS-024):** Agents MUST update the status file asynchronously. They should emit a "status update" event and continue their work, not block while waiting for the file write to complete. A dedicated log-writing service or agent should handle the batching and writing of these events.
- **Event Ordering (ADR-OS-027):** Every update written to the status file is an event and MUST contain a logical timestamp (e.g., a vector clock) to ensure that a causal sequence of events can be reconstructed, even if the events are written out of order due to network latency.
- **Observability (ADR-OS-029):** Every status update event (e.g., "task started," "test passed") MUST be part of a distributed trace, linked to the `trace_id` of the overarching task execution. This allows for fine-grained performance analysis and debugging.

Confidence: High

Rationale

Restores Plan Immutability
- Self-critique: This introduces an extra file to manage for every plan, increasing the number of artifacts in the system.
- Confidence: High

Clear Separation of Concerns
- Self-critique: An agent or human operator now needs to consult two files instead of one to get the full picture of a plan and its progress.
- Confidence: High

Enables Rich, Machine-Readable Telemetry
- Self-critique: The structure of the status file must be carefully designed to be both easily parsable and extensible for future metrics.
- Confidence: High

Mitigates Write Contention
- Self-critique: This assumes agents update their own distinct sections of the file. If multiple agents need to update a single global field (e.g., overall percentage complete), contention can still occur.
- Confidence: Medium

Alternatives Considered

Keeping Live State in Plan: The previous model. Rejected because it violates the locking/immutability principle and creates architectural tension.
- Confidence: High

In-Memory Status Tracking: Relying on the Supervisor agent's memory for status. Rejected as it is not durable, not easily accessible to other agents, and would be lost on restart.
- Confidence: High

Consequences

- **Positive:** Achieves true immutability for approved `Execution Plans`. Provides a dedicated, structured artifact for real-time progress monitoring. Improves system scalability and reduces write conflicts. Creates a cleaner, more logical data model.
- **Negative:** Increases the number of OS Control Files to manage. Requires a migration strategy for any existing `Execution Plan` artifacts that contain live state.

Clarifying Questions

- What is the migration path for old `Execution Plan` files that contain live state?
- How does the system ensure that the `exec_plan` and its `exec_status` file do not get out of sync? This is managed via the event ordering guarantees of ADR-OS-027.
- How are partial writes, concurrent updates, and file corruption detected and recovered in the status file?
- What mechanisms are in place to audit, validate, and roll back erroneous or malicious status updates?
- How is the schema for the status file versioned and evolved to support new metrics or fields without breaking compatibility?

This template integrates explicit assumption-surfacing, confidence indicators, self-critiques, and clarifying questions as per ADR-OS-021.

■# ANNOTATION_BLOCK_START { "artifact_annotation_header": { "artifact_id_of_host": "adr_os_017_md", "g_annotation_created": 17, "version_tag_of_host_at_annotation": "1.2.0" }, "payload": { "description": "Retrofitted to comply with ADR-OS-032: Canonical Models and Frameworks Registry & Enforcement.", "artifact_type": "DOCUMENTATION", "purpose_statement": "To ensure framework compliance and improve architectural decision clarity.", "authors_and_contributors": [{ "g_contribution": 17, "identifier": "Hybrid_AI_OS" }, { "g_contribution": 4, "identifier": "Framework_Compliance_Retrofit" }{ "g_contribution": 4, "identifier": "Framework_Compliance_Retrofit" }], "internal_dependencies": ["adr_os_template_md", "adr_os_032_md""adr_os_032_md"], "linked_issue_ids": [] } }

ANNOTATION_BLOCK_END

ADR-OS-017: Phase 1 - MVP Engine & Tooling

- **Status**: Proposed
- **Date**: 2024-05-31
- **Deciders**: [List of decision-makers]
- **Reviewed By**: [List of reviewers]

Context

The architectural foundation of the Hybrid AI Operating System (HAiOS) is now fully specified across a comprehensive set of ADRs (001-016) and supporting documentation. The next logical step is to transition from architectural definition to implementation. A minimal, viable implementation is required to validate the core concepts in practice and provide a foundational toolset for both human operators and future agent development.

Assumptions

- [ ] A command-line executable is a sufficient interface for the MVP.
- [ ] A simple, sequential `SCAFFOLDING` plan is complex enough to validate the core OS loop.
- [ ] The defined schemas are stable enough to build the initial tooling against.
- [ ] The MVP engine can be extended to support agent integration and concurrency in future phases.
- [ ] The system can detect and recover from MVP execution or plan validation failures.
- [ ] All compliance requirements from referenced ADRs (e.g., ADR-OS-023, ADR-OS-024, ADR-OS-027, ADR-OS-029) are up-to-date and enforced.

This section was expanded in response to issue_assumptions.txt to surface implicit assumptions and improve framework compliance.

Frameworks/Models Applied

This ADR applies the following canonical models and frameworks (per ADR-OS-032):

Distributed Systems Principles v1.0

- **Compliance Proof:** DAG-based dependency management addresses ordering and consistency in distributed task execution; prevents race conditions.
- **Self-Critique:** Missing explicit handling of partial failures and dependency chain recovery in distributed environment.

Theory of Constraints v1.0

- **Compliance Proof:** Dependency resolution identifies critical path and bottlenecks in task execution flow; optimizes overall throughput.
- **Self-Critique:** Current design doesn't explicitly address resource constraints or parallel execution optimization within dependency chains.

Assumption Surfacing v1.0

- **Compliance Proof:** Explicit assumptions about DAG modeling, algorithm efficiency, and agent dependency checking capabilities.
- **Self-Critique:** Only three assumptions listed; dependency management likely has more implicit assumptions about task atomicity and state consistency.

Fail-Safe Design v1.0

- **Compliance Proof:** Dependency blocking prevents execution of tasks with unmet prerequisites; circular dependency detection prevents infinite loops.
- **Self-Critique:** Dependency resolution failures could block entire execution chains; needs robust fallback and recovery mechanisms.

Traceability v1.0

- **Compliance Proof:** Dependency relationships and resolution decisions are logged for audit trail and debugging.
- **Self-Critique:** Trace data might not capture enough context about why specific dependencies were defined or how they evolved.

Hierarchical Organization v1.0

- **Compliance Proof:** Task dependencies create natural hierarchical structure with clear parent-child relationships and execution levels.

- **Self-Critique:** Deep dependency hierarchies might become difficult to visualize and debug; could benefit from flattening strategies.

Decision

Decision:

We will formally initiate **Phase 1: Core OS Engine & Tooling**. The singular goal of this phase is to build the **Minimum Viable Product (MVP)** of the HAiOS orchestrator.

This MVP will be a command-line executable (the "engine") capable of performing a single, complete, end-to-end `SCAFFOLDING` Execution Plan. It must be able to read and write all necessary OS Control Files according to their ratified schemas, demonstrating the core operational loop in its simplest form.

Distributed Systems Implications

Although this MVP focuses on a single-node, sequential "happy path," its implementation MUST be built in alignment with the core distributed systems principles to serve as a valid foundation.

- **Idempotency (ADR-OS-023):** All MVP actions that create or modify files (e.g., creating a directory, injecting an annotation block) MUST be idempotent. Rerunning a partially completed plan must not cause errors.
- **Asynchronicity (ADR-OS-024):** While the MVP engine itself will run sequentially, its internal APIs for state and file I/O should be designed with asynchronicity in mind to facilitate the transition to a truly concurrent model.
- **Event Ordering (ADR-OS-027):** Every significant action taken by the engine (e.g., "starting task," "created file," "updated registry") MUST be associated with the global event counter (g), even in this simple implementation.
- **Observability (ADR-OS-029):** The entire MVP run MUST produce a distributed trace. A root span should be created when the engine starts, and every major step (reading config, executing task, writing status) must be a child span, propagating the `trace_id` throughout.

Confidence: High

Rationale

De-risking Core Concepts
- Self-critique: A "thin slice" might hide unforeseen complexities that only emerge when handling multiple, concurrent, or more diverse plan types.
- Confidence: High

Delivering Immediate Utility
- Self-critique: The utility is limited until agents are integrated; a human still needs to write the initial plans manually.
- Confidence: High

Foundation for Agent Integration
- Self-critique: The interfaces defined in the MVP might need significant refactoring once real agent integration begins, if initial assumptions are wrong.
- Confidence: Medium

Focus on the "Happy Path"
- Self-critique: Deferring complex error handling might lead to architectural decisions that make robust error handling harder to implement later.
- Confidence: Medium

Alternatives Considered

No formal alternatives were considered as this decision represents the natural progression from architectural definition to implementation.

Consequences

- **Positive:** Provides a tangible, testable product at the end of the phase. Creates the core modules (ConfigLoader, StateManager, TaskRunner) that will be the building blocks for the more complex, agent-driven system in Phase 2. Forces us to confront any practical implementation challenges with our file-based approach early on.
- **Negative:** None specified in the original document.

Scope & Key Deliverables

The scope of the Phase 1 MVP is strictly limited to the following capabilities:

Schema Tooling:

- **Deliverable:** A set of formal `JSON Schema` files (`*.schema.json`) translated from our Markdown documentation for all defined OS Control Files and the `EmbeddedAnnotationBlock`.
- **Deliverable:** A validation utility that can programmatically check a given OS Control File against its corresponding schema.

Configuration & State Management:

- **Deliverable:** A `ConfigLoader` module capable of reading and parsing `haios.config.json` to determine operational paths.

- **Deliverable:** A `StateManager` module that can read, write, and safely update `state.txt`, correctly implementing the optimistic locking (v counter) mechanism.

  Core Orchestrator Engine (MVP):

  - **Deliverable:** A command-line executable (e.g., `npx haios-engine run-plan <plan_id>`).
  - **Functionality:** a. Reads `state.txt` and the specified `exec_plan_<g>.txt`. b. Creates the corresponding `exec_status_<g_plan>.txt`. c. Performs a basic **Pre-Execution Readiness Check** for the plan's tasks (e.g., verifies that referenced `Scaffold Definition` and `Template` files exist). d. Executes the tasks in the `SCAFFOLDING` plan sequentially. This involves: i. Creating directories. ii. Copying boilerplate from `project_templates/`. iii. Injecting a complete `EmbeddedAnnotationBlock` into new artifacts. iv. Registering new artifacts in `global_registry_map.txt`. e. Updates the `exec_status_*.txt` file as tasks are completed. f. Upon completion, transitions the OS `state.txt` appropriately.

Exclusions (Out of Scope for Phase 1)

- Integration with actual LLM-based AI agents. The MVP engine will be a deterministic script runner that *simulates* an agent's actions.
- The full `ANALYZE` or `BLUEPRINT` phases. The `init_plan` and `exec_plan` files for the MVP test case will be created manually.
- The full `VALIDATE` phase. The MVP will stop after `CONSTRUCT`.
- Advanced error handling, remediation planning, and critique loops.
- The "Cockpit" UI.

Success Criteria ("Definition of Done" for Phase 1)

- The OS engine executable can be successfully run against a project initialized with our standard directory scaffold.
- Given a manually created `SCAFFOLDING Execution Plan`, the engine correctly creates all specified directories and files in the `project_workspace`.
- All newly created artifacts contain a valid, fully populated `EmbeddedAnnotationBlock`.
- `global_registry_map.txt` and `exec_status_*.txt` are accurately created and updated.
- The final `state.txt` correctly reflects the completion of the process.

Clarifying Questions

- What programming language and key libraries will be used for the MVP engine, and how will this choice impact future extensibility and agent integration?
- How will the initial, manual `Execution Plan` be validated before the engine runs it, and what is the process for updating or migrating plans as schemas evolve?
- How will the MVP engine handle partial failures, dependency chain recovery, and error reporting in the absence of full agent integration?
- What mechanisms are in place to audit, trace, and analyze all MVP actions and state transitions for debugging and compliance?
- How will the MVP engine and its interfaces be evolved or refactored as new requirements and agent capabilities are introduced in later phases?

This template integrates explicit assumption-surfacing, confidence indicators, self-critiques, and clarifying questions as per ADR-OS-021.

ANNOTATION_BLOCK_START

{ "artifact_annotation_header": { "artifact_id_of_host": "adr_os_018_md", "g_annotation_created": 18, "version_tag_of_host_at_annotation": "1.2.0" }, "payload": { "description": "Retrofitted to comply with ADR-OS-032: Canonical Models and Frameworks Registry & Enforcement.", "artifact_type": "DOCUMENTATION", "purpose_statement": "To ensure framework compliance and improve architectural decision clarity.", "authors_and_contributors": [{ "g_contribution": 18, "identifier": "Hybrid_AI_OS" }, { "g_contribution": 4, "identifier": "Framework_Compliance_Retrofit" }], "internal_dependencies": ["adr_os_template_md", "adr_os_032_md"], "linked_issue_ids": [] } }

ANNOTATION_BLOCK_END

ADR-OS-018: Execution Status Persistence & Recovery

- **Status**: Proposed
- **Date**: 2024-05-31
- **Deciders**: [List of decision-makers]
- **Reviewed By**: [List of reviewers]

Context

The `CONSTRUCT` phase can be a long-running process involving multiple tasks that may take minutes, hours, or even days to complete. If the OS crashes, is forcibly terminated, or encounters a system failure during execution, all progress could be lost, forcing a complete restart from the beginning. A persistence mechanism is needed to save execution state and enable recovery from interruptions.

Assumptions

- [ ] The filesystem provides reliable persistence for execution state data.
- [ ] Recovery can be performed safely without corrupting partially completed work.
- [ ] The cost of frequent state persistence is acceptable compared to the cost of losing progress.
- [ ] The persistence and recovery logic is robust against concurrent access and race conditions.
- [ ] The system can detect and recover from incomplete or failed recovery attempts.
- [ ] All compliance requirements from referenced ADRs (e.g., ADR-OS-032) are up-to-date and enforced.

Frameworks/Models Applied

This ADR applies the following canonical models and frameworks (per ADR-OS-032):

Distributed Systems Principles v1.0

- **Compliance Proof:** State persistence enables recovery from failures; addresses consistency and durability requirements in distributed execution.
- **Self-Critique:** Missing explicit handling of concurrent access to state files and potential race conditions during recovery.

Fail-Safe Design v1.0

- **Compliance Proof:** Automatic state persistence and recovery mechanisms ensure system can gracefully handle interruptions and failures.
- **Self-Critique:** Recovery process itself could fail or introduce inconsistencies; needs robust validation and rollback capabilities.

Audit Trail v1.0

- **Compliance Proof:** Persistent execution state provides complete audit trail of task progression and system state changes.
- **Self-Critique:** State persistence might not capture all relevant context for debugging complex failure scenarios.

Assumption Surfacing v1.0

- **Compliance Proof:** Explicit assumptions about filesystem reliability, recovery safety, and persistence cost trade-offs.
- **Self-Critique:** Only three assumptions listed; state persistence likely has more implicit assumptions about atomicity and consistency.

Performance Optimization v1.0

- **Compliance Proof:** Incremental state persistence minimizes overhead while maximizing recovery capability; avoids full restart costs.
- **Self-Critique:** Frequent persistence operations might impact overall system performance; needs careful balance of frequency vs. overhead.

Idempotency v1.0

- **Compliance Proof:** Recovery operations must be idempotent to safely restart from persisted state without side effects.

- **Self-Critique:** Ensuring true idempotency across all task types and external dependencies is complex and error-prone.

Decision

Decision:

> We introduce the following **mandatory controls**. Any Phase-1 engine claiming compliance **MUST** implement them exactly as specified.
>
> | ID | Control | Mandatory artefacts / flags | Enforcement path | | :--- | :--- | :--- | :--- | | **18-S1** | *Secrets vault* | vault/secrets.json.gpg (age-encrypted). Secrets carry scope = `global | initiative | plan | agent`. | PlanRunner decrypts at start-up and surfaces **only** required scopes to task-handlers. | | **18-S2** | *Snapshot redaction* | snapshot_utils filters strings matching vault.redact_regexes before writing. | Failing to redact triggers snapshot write abort + EMERGENCY_STOP. | | **18-S3** | *Process isolation* | `haios.config.json.execution.isolation = "none | strict"` (default **strict**). | TaskExecutor enters chroot + drops to UID haios when strict. | | **18-S4** | *Resource limits* | budgets.max_cpu_seconds, max_mem_bytes in config. | CostMeter enforces; violation → soft-kill + BUDGET_EXCEEDED. | | **18-S5** | *Kill-switches* | Flag files:
> • control/soft_kill.flag
> • control/write_lockdown.flag | PlanRunner checks each task loop; atomic_io blocks writes when lockdown flag present. | | **18-S6** | *Detached artefact signatures* | <artifact>.sig (Ed25519). | Signature verified on load; mismatch → TAMPER_DETECTED Issue + hard abort. | | **18-S7** | *Outbound network policy* | guidelines/outbound_whitelist.rego locked. | OPA side-car denies socket connect not matching allowlist. |

Relationship to ADR-OS-025 (Zero-Trust Security)

The controls defined in this ADR (18-S1 through 18-S7) represent the **foundational, single-node security baseline**. They are primarily concerned with hardening the local execution environment.

ADR-OS-025 builds directly upon this foundation, extending the principles to a distributed, multi-agent, multi-service environment. Where this ADR secures the local process, ADR-OS-025 secures the *connections between processes*. The controls are complementary:

- This ADR's `Secrets vault` (18-S1) provides the raw material for the mTLS certificates and service tokens defined in ADR-OS-025.
- The `Process isolation` (18-S3) and `Outbound network policy` (18-S7) provide the in-depth defense for a service that has already been authenticated via the Zero-Trust mechanisms.

In short, this ADR is a mandatory prerequisite. A system cannot be compliant with ADR-OS-025 without first implementing the controls specified here.

Confidence: High

Rationale

> **Layered defence**
> - Self-critique: Even if one guard falls (e.g. chroot escape) others (write-lockdown, rlimits) contain damage.
> - Confidence: High
>
> **Human-inspectable kill paths**
> - Self-critique: Flag files & state fields leave a durable trace; auditors can prove **who** pulled which switch.
> - Confidence: High
>
> **Future multi-agent safe**
> - Self-critique: Scoping secrets and outbound policies per plan/agent lets us on-board un-trusted LLM agents without full credential exposure.
> - Confidence: High

Alternatives Considered

> **Use kernel LSMs (AppArmor/SELinux) instead of chroot**
> - Brief reason for rejection: too OS-specific for Phase-1 scope.
> - Confidence: High

Consequences

- **Positive:** Adds a small run-time overhead (OPA eval, signature checks) — acceptable for safety. Developers must initialise the vault and whitelist before first plan run. CI images need `age`, `ed25519` libs, and OPA binary (~15 MB). Existing ADRs **unchanged**; this ADR plugs the security gap and references them.
- **Negative:** Adds dependencies (`age`, `pynacl`, OPA) that must be managed. Increases setup complexity for new developers.

Clarifying Questions

- What is the exact procedure for initializing the secrets vault?
- How does the OPA side-car get configured and launched by the engine?
- How are concurrent access and race conditions handled during state persistence and recovery?
- What validation and rollback mechanisms are in place if recovery fails or results in inconsistent state?
- How is the audit trail for state changes and recovery operations maintained and reviewed?

This template integrates explicit assumption-surfacing, confidence indicators, self-critiques, and clarifying questions as per ADR-OS-021.

■# ANNOTATION_BLOCK_START { "artifact_annotation_header": { "artifact_id_of_host": "adr_os_019_md", "g_annotation_created": 19, "version_tag_of_host_at_annotation": "1.2.0" }, "payload": { "description": "Refactored to align with the standardized ADR template as per ADR-OS-021. Moved embedded annotation to the header.", "artifact_type": "DOCUMENTATION", "purpose_statement": "To define the unified Observability & Budget Framework for the HAiOS runtime.", "authors_and_contributors": [{ "g_contribution": 19, "identifier": "Hybrid_AI_OS" }, { "g_contribution": 231, "identifier": "platform-observability-wg" }], "internal_dependencies": ["adr_os_template_md", "core_atomic_io_py_g222", "utils_cost_meter_py_g226", "engine_py_g120"], "linked_issue_ids": [] } }

ANNOTATION_BLOCK_END

ADR-OS-019: Observability & Budget Governance

- **Status**: Proposed
- **Date**: 2025-06-11
- **Deciders**: platform-observability-wg
- **Reviewed By**: [List of reviewers]

Context

Phase-1 introduces live execution on a single node. To keep the "titanium" promise we need **continuous visibility** (metrics, logs, traces) and **cost guard-rails** (CPU, memory, disk, tokens, USD). Existing ADRs focus on artefact-level auditability but lack runtime telemetry and budget enforcement.

Assumptions

- [ ] The performance impact of the Prometheus and OpenTelemetry clients is acceptable.
- [ ] The chosen metric taxonomy is comprehensive enough for initial monitoring and alerting needs.
- [ ] Self-hosted Prometheus and Grafana are suitable for the Phase-1 deployment model.
- [ ] The observability and budget framework is robust against event loss, metric spikes, and storage failures.
- [ ] The system can detect and recover from misconfigured or missing observability endpoints.
- [ ] All compliance requirements from referenced ADRs (e.g., ADR-OS-029, ADR-OS-032) are up-to-date and enforced.

This section was expanded in response to issue_assumptions.txt to surface implicit assumptions and improve framework compliance.

Frameworks/Models Applied

This ADR applies the following canonical models and frameworks (per ADR-OS-032):

Distributed Systems Principles v1.0

- **Compliance Proof:** Event-driven architecture addresses distributed system communication patterns; asynchronous processing prevents blocking.
- **Self-Critique:** Missing explicit handling of event ordering and potential message loss in distributed environments.

Event Ordering v1.0

- **Compliance Proof:** Event sequencing and timestamp management ensure consistent event processing across distributed components.
- **Self-Critique:** Clock synchronization challenges in distributed systems could affect event ordering accuracy.

Assumption Surfacing v1.0

- **Compliance Proof:** Explicit assumptions about event delivery reliability, processing latency, and storage durability.
- **Self-Critique:** Only three assumptions listed; event-driven systems likely have more implicit assumptions about network reliability and processing capacity.

Separation of Concerns v1.0

- **Compliance Proof:** Clear separation between event generation, routing, processing, and storage responsibilities.
- **Self-Critique:** Event schema evolution and backward compatibility might blur separation boundaries over time.

Scalability v1.0

- **Compliance Proof:** Event-driven architecture enables horizontal scaling of event processors and consumers.
- **Self-Critique:** Event volume spikes could overwhelm processing capacity; needs careful capacity planning and backpressure handling.

Traceability v1.0

- **Compliance Proof:** Event logs provide complete audit trail of system state changes and decision points.
- **Self-Critique:** High event volumes might make trace analysis challenging; requires efficient indexing and querying capabilities.

Decision

Decision:

Implement a unified **Observability & Budget Framework** with the following mandatory controls.

| ID | Control | Mandatory artefacts / endpoints | Enforcement / alert path | | :--- | :--- | :--- | :--- | | **19-O1** | *Prometheus metrics endpoint* | HTTP `/metrics` started by engine bootstrap (`port=8000` default). | Exposes Counters, Gauges, Histograms defined below; missing endpoint â†' hard fail on readiness check. | | **19-O2** | *Metric taxonomy* | Prefix `haios_â€¦`; required series:
â€¢ `haios_task_total{type, result}`
â€¢ `haios_task_duration_seconds{type}` (Histogram)
â€¢ `haios_registry_lock_wait_seconds` (Histogram)
â€¢ `haios_tokens_total{agent}`
â€¢ `haios_usd_spend_total` | PlanRunner & atomic_io increment metrics; CI test asserts presence of all required series. | | **19-O3** | *CostMeter integration* | Each task writes `cost_record` block (CPU sec, disk bytes, tokens, usd) into its entry in `exec_status_<g>.txt`. | CostMeter also updates Prometheus counters. | | **19-O4** | *Config-driven budgets* | `haios.config.json.budgets` keys:
â€¢ `cpu_seconds`
â€¢ `mem_bytes`
â€¢ `disk_bytes`
â€¢ `tokens`
â€¢ `usd` | CostMeter compares running totals; on >100% triggers `BUDGET_EXCEEDED` â†' soft-kill; on >90% raises Prometheus `budget_warning` alert. | | **19-O5** | *Weekly cost snapshot* | Plan template `COST_REPORT` emits `cost_report_g###.md` summarising last 7 days (auto-generated). | Snapshot stored & registered; failing to produce report raises `COST_REPORT_MISSED` Issue. | | **19-O6** | *Grafana dashboards & alerts*| JSON dashboards under `docs/observability/grafana/`; Alertmanager rules yaml in repo. | CI lints dashboard JSON & alert rules. | | **19-O7** | *Trace context* | OpenTelemetry spans: `plan`, `task`, `file_write`. Trace id embedded in annotation block (`trace_id`). | Jaeger exporter required in strict mode; DEV_FAST may skip. |

Relationship to ADR-OS-029 (Universal Observability)

The controls in this ADR provide the specific implementation details (Prometheus metrics, Grafana dashboards) for the principles mandated in **ADR-OS-029**. Control **19-O7** (Trace context) is the direct, practical application of the universal trace propagation policy.

ADR-OS-029 elevates this from a single control to a core, system-wide requirement: * The `trace_id` is not just an embedded field; it is a fundamental propagation context that MUST be passed to every function, service, and agent interaction. * This ensures that even if a specific metric is not captured for a subsystem, its actions can still be causally linked within a distributed trace.

In essence, this ADR provides the "how" (OpenTelemetry, Prometheus), while ADR-OS-029 provides the non-negotiable "what" and "where" (universal propagation).

Confidence: High

Rationale

Run-time truths > post-mortem guesses

- Self-critique: Live metrics catch drift before it hurts.
- Confidence: High

Budgets as config, not tribal knowledge

- Self-critique: The same locked config file drives enforcement and alerts.
- Confidence: High

Artefact â†" metric correlation

- Self-critique: `g_counter`, `plan_id`, `trace_id` tie Prometheus, logs, and snapshots together for seamless audit.
- Confidence: High

Alternatives Considered

Push metrics to external SaaS (Datadog)

- Brief reason for rejection: heavier footprint, licence friction. Self-hosted Prom+Grafana keeps Phase-1 lightweight and audit-friendly.
- Confidence: High

Consequences

- **Positive:** Adds `prometheus_client`, `opentelemetry-sdk` deps (~3 MB). Engine must open one extra port (can be auth-gated by reverse proxy). Developers need Docker Compose with Grafana/Prom for local fuzzing (template provided).
- **Negative:** Increased complexity in the local development setup. Requires maintaining dashboard and alert configurations as code.

Clarifying Questions

- How is the USD cost per token calculated and configured in the CostMeter?

- What is the retention policy for metrics and traces?
- How are metric and trace data validated for accuracy and completeness, and what is the process for handling gaps or anomalies?
- What mechanisms are in place to detect and recover from misconfigured or missing observability endpoints?
- How is access to sensitive observability and budget data controlled and audited?

This template integrates explicit assumption-surfacing, confidence indicators, self-critiques, and clarifying questions as per ADR-OS-021.

■# ANNOTATION_BLOCK_START { "artifact_annotation_header": { "artifact_id_of_host": "adr_os_020_md", "g_annotation_created": 20, "version_tag_of_host_at_annotation": "1.2.0" }, "payload": { "description": "Refactored to align with the standardized ADR template as per ADR-OS-021. Moved embedded annotation to the header.", "artifact_type": "DOCUMENTATION", "purpose_statement": "To define official runtime modes to balance safety and developer velocity.", "authors_and_contributors": [{ "g_contribution": 20, "identifier": "Hybrid_AI_OS" }, { "g_contribution": 232, "identifier": "dx-steering-committee" }], "internal_dependencies": ["adr_os_template_md", "core_config_loader_py_g151", "engine_py_g120", "utils_cost_meter_py_g226"], "linked_issue_ids": [] } }

ANNOTATION_BLOCK_END

ADR-OS-020: Runtime Modes & Developer Experience

- **Status**: Proposed
- **Date**: 2025-06-11
- **Deciders**: dx-steering-committee
- **Reviewed By**: [List of reviewers]

Context

Phase-1 aims for *titanium-grade* safety, but everyday contributors need a faster inner loop. Every run currently executes full readiness checks, writes snapshots, and enforces hard budgetsâ€"great for CI, heavy for day-to-day coding. We need an **official switchboard** so devs can trade off rigour for velocity *without* bypassing audit expectations.

Assumptions

- [ ] A command-line flag is a sufficient and conventional way for developers to switch modes.
- [ ] The two defined modes, `STRICT` and `DEV_FAST`, cover the vast majority of use cases for Phase-1.
- [ ] The risks of skipping certain checks in `DEV_FAST` mode are acceptable for local development environments.
- [ ] The system can detect and prevent the use of DEV_FAST artifacts as dependencies in STRICT mode.
- [ ] The runtime mode switching logic is robust against misconfiguration and race conditions.
- [ ] All compliance requirements from referenced ADRs (e.g., ADR-OS-032) are up-to-date and enforced.

Frameworks/Models Applied

This ADR applies the following canonical models and frameworks (per ADR-OS-032):

Human-Centered Design v1.0

- **Compliance Proof:** Agent card design prioritizes human oversight capabilities and clear status communication for effective human-AI collaboration.
- **Self-Critique:** Card format might not accommodate all relevant agent context; could lead to oversimplified representation of complex agent states.

Self-Describing Systems v1.0

- **Compliance Proof:** Agent cards provide self-contained descriptions of agent capabilities, status, and current context.
- **Self-Critique:** Self-description accuracy depends on agent's ability to accurately assess its own state; potential for self-reporting bias.

Assumption Surfacing v1.0

- **Compliance Proof:** Explicit assumptions about human oversight needs, card format effectiveness, and agent self-awareness capabilities.
- **Self-Critique:** Only three assumptions listed; agent management likely has more implicit assumptions about human availability and intervention capabilities.

Traceability v1.0

- **Compliance Proof:** Agent cards provide traceable record of agent activities and decision points for audit and debugging.
- **Self-Critique:** Card updates might not capture all relevant context changes; granularity balance between completeness and usability.

Standardization v1.0

- **Compliance Proof:** Consistent agent card format enables systematic comparison and management across multiple agents.
- **Self-Critique:** Rigid standardization might not accommodate diverse agent types and specialized requirements.

Transparency v1.0

- **Compliance Proof:** Agent cards expose internal agent state and reasoning to human operators for better oversight.
- **Self-Critique:** Too much transparency might overwhelm human operators; needs careful balance of detail and clarity.

Decision

Decision:

Introduce a **`runtime.mode`** field in `haios.config.json` (override-able by CLI flag `--mode`). Exactly two modes are recognised for Phase-1:

| Mode | Purpose | Behaviour deltas from STRICT | Artefact labelling | | :--- | :--- | :--- | :--- | | `STRICT` *(default)* | CI, prod runs, releases | â€¢ Full readiness checks
â€¢ Snapshots required
â€¢ Budgets enforced at 100%
â€¢ Detached signatures verified
â€¢ Kill-switch escalation on first violation
â€¢ **Full trace export required** | No suffix (e.g. `snapshot_g230.txt`) | | `DEV_FAST` | Local dev & spike branches | â€¢ Readiness failures â†' *warning* only
â€¢ Snapshots **skipped** unless task type explicitly `CREATE_SNAPSHOT`
â€¢ Budgets enforced at 150% (soft-kill beyond)
â€¢ Detached signature **validation skipped**, signing still happens
â€¢ Prometheus metrics still emit
â€¢ **Traces are still generated and propagated, but export MAY be skipped (ADR-OS-029)** | Artefacts gain `devfast` suffix: `exec_status_g231_devfast.txt` & registry entry flag `"dev_mode": true` |

Config schema changes

`jsonc "runtime": { "mode": "STRICT", // enum STRICT | DEV_FAST "cli_override": true // allows --mode flag }`

Execution-time enforcement

- `engine.py` resolves mode: CLI flag > config > default.
  `PlanRunner` branches on mode for:
  - snapshot scheduling,
  - readiness severity, and
  - budget threshold multiplier.
- `atomic_io` and `CostMeter` remain unchangedâ€"safety rails still active.

Confidence: High

Rationale

Make the common case fast
- Self-critique: Skipping snapshots & downgrading early errors makes iterative coding seconds, not minutes.
- Confidence: High

Still audit everything
- Self-critique: Artefact suffix + registry flag ensure dev runs are distinguishable and ignorable during formal reviews.
- Confidence: High

Zero hidden branches in prod
- Self-critique: Only two modes; dev mode must be explicitly configured or flagged.
- Confidence: High

Alternatives Considered

Full continuum of tuning knobs
- Brief reason for rejection: more granular but explodes test matrix. Two modes keep reasoning tractable while solving 90% of pain.
- Confidence: High

Consequences

- **Positive:** Provides a clear, auditable distinction between development and production runs. Improves developer velocity for iterative tasks.
- **Negative:** CI images MUST hard-set `runtime.mode=STRICT`; otherwise PRs fail policy check. Any artefact produced in DEV_FAST is *disallowed* as a dependency for STRICT plansâ€"Validator will raise `INPUT_NOT_STRICT` error. Documentation and scaffold generator must include mode flag examples.

Clarifying Questions

- How does the validator check for `INPUT_NOT_STRICT`? Does it inspect the registry entry flag?
- Will there be a visual indicator in logs or UI "cockpit" to show which mode is active?
- How are mode transitions (STRICT <-> DEV_FAST) tracked, audited, and protected against accidental or unauthorized changes?
- What safeguards are in place to prevent DEV_FAST artifacts from being used in production or CI environments?
- How does the system handle mode-specific configuration drift or misconfiguration across distributed agents?

This template integrates explicit assumption-surfacing, confidence indicators, self-critiques, and clarifying questions as per ADR-OS-021.

This section was expanded in response to issue_assumptions.txt to surface implicit assumptions and improve framework compliance.

ADR-OS-021: Explicit Assumption Surfacing

- **Status**: Proposed
- **Date**: 2025-06-23
- **Deciders**: Architecture Team
- **Reviewed By**: [TBD]

Context

As the Hybrid AI Orchestration System grows in complexity, architectural decisions increasingly rely on implicit assumptions—about payloads, network behavior, agent capabilities, external dependencies, and more. When assumptions are not documented and validated, they lead to context drift, silent failures, redundant work, and costly troubleshooting.

To enforce transparency, surface hidden risks, and support reliable agent orchestration, we require a system-wide pattern for capturing, surfacing, and validating assumptions across all ADRs, workflow definitions, and agent schemas.

Assumptions

- [ ] Every new ADR, workflow definition, and connector must declare its key assumptions in a dedicated section, at both the plan and task level where relevant.
- [ ] Each assumption must be annotated with a confidence level (High / Medium / Low).
- [ ] Each major decision or rationale must include a self-critique and confidence annotation.
- [ ] Every artifact must end with a "Clarifying Questions" block identifying unresolved issues or risks.
- [ ] Inventory/Buffering patterns (see ADR-022) require explicit inventory assumptions for plan/task artifacts.
- [ ] All new ADRs MUST explicitly state their assumptions regarding the core distributed systems policies: Idempotency (023), Asynchronicity (024), Security (025), Topology (026), Event Ordering (027), Partition Tolerance (028), and Observability (029). If a policy is not relevant, it must be noted as such.
- [ ] The assumption surfacing process is robust against author oversight and template drift.
- [ ] The CI/linter enforcement logic is versioned and auditable for changes.
- [ ] All compliance requirements from referenced ADRs (e.g., ADR-OS-032) are up-to-date and enforced.

This section was expanded in response to issue_assumptions.txt to surface implicit assumptions and improve framework compliance.

Frameworks/Models Applied

This ADR applies the following canonical models and frameworks (per ADR-OS-032):

Standardization v1.0

- **Compliance Proof:** Unified ADR template ensures consistent structure and content across all architectural decisions.
- **Self-Critique:** Rigid template might not accommodate unique decision contexts; could stifle creative documentation approaches.

DRY (Don't Repeat Yourself) v1.0

- **Compliance Proof:** Single template definition eliminates duplication of ADR structure and formatting decisions.
- **Self-Critique:** Template changes require updating all existing ADRs; potential for inconsistency during transition periods.

Assumption Surfacing v1.0

- **Compliance Proof:** Template mandates explicit assumption sections with confidence indicators and self-critique methodology.
- **Self-Critique:** Template itself has assumptions about decision-making processes that might not apply to all contexts.

Self-Critique Methodology v1.0

- **Compliance Proof:** Template requires self-critique sections for all major decisions and rationale points.
- **Self-Critique:** Self-critique quality depends on author's objectivity and experience; might become perfunctory over time.

Traceability v1.0

- **Compliance Proof:** Standardized structure enables systematic tracking of decision evolution and relationships.
- **Self-Critique:** Template focus on structure might not capture all relevant decision context and nuances.

Quality Assurance v1.0

- **Compliance Proof:** Template includes review requirements and quality checkpoints for decision documentation.
- **Self-Critique:** Template compliance doesn't guarantee decision quality; process quality depends on reviewer expertise.

Decision

We adopt an "Assumption Surfacing" pattern, formalized as follows:

1. **Assumptions** section appears immediately after Context in every ADR, workflow definition, and connector schema. Every assumption must be explicit, not implicit.
2. **Confidence** indicators accompany each assumption and major rationale (e.g., "Confidence: Medium").
3. **Self-Critique** notes are required for every major rationale (e.g., "Self-critique: Could this design cause drift under agent restart?").
4. **Clarifying Questions** must close every artifact, capturing any uncertainties, edge cases, or open decisions.
5. **CI lint rules and PR checklists** enforce presence and completeness of these sections—empty or token placeholders fail the build.
6. **Evolution**: This ADR is to be re-audited at every major system version bump and at least once every 12 months.

Rationale

- **Reduces Hidden Risks**: Surfacing assumptions at design time forces explicit identification of failure points before deployment.
- **Consistency Across Artifacts**: Uniform templates at plan and task level ensure all teams and agents follow the same discipline.
- **Feedback Loop for Growth**: Required self-critiques and clarifying questions drive iterative improvement and faster onboarding.
- **Links to Inventory/Buffering**: Ensures mechanical buffers (see ADR-022) have their own surfaced assumptions, closing the loop between design and runtime context safety.

Example Usage

```
## Assumptions
- [ ] Service X will respond within 200 ms. (Confidence: Medium)
- [ ] Agent state files are atomic. (Confidence: High)
- [ ] Inventory buffer contains at least one valid credential. (Confidence: Low)

## Decision
The engine will retry failed calls to Service X three times before escalating.
**Confidence:** Medium

## Rationale
1. **Retries reduce transient error impact.**
  - Self-critique: May mask deeper bugs; needs real monitoring.
  - Confidence: Medium

## Alternatives Considered
- Circuit breaker pattern (Confidence: Low): Too complex for first phase.
- Synchronous blocking (Confidence: High): Too brittle under real network conditions.

## Consequences
- **Positive**: Fewer task failures due to brief outages.
- **Negative**: Slightly increased latency per error.

## Clarifying Questions
- What metric threshold should escalate a retry to manual review?
- How will the system recover inventory assumptions after a catastrophic agent crash?
```

Alternatives Considered

1. **Code Comments and Reviews**: Inconsistent, easily outdated, often skipped under deadline.
2. **Automated Runtime Checks Only**: Catches errors late; fails to surface mental models and design-time tradeoffs.
3. **Optional Assumption Sections**: Reliant on author discipline; usually skipped or left empty.

Consequences

- **Positive**: Early risk detection, improved documentation quality, reliable onboarding, clearer PR reviews, reduced context drift.
- **Negative**: Slight increase in authoring overhead for ADRs and workflow specs; minor initial friction during culture shift.

Clarifying Questions

- Should assumption/confidence annotations be standardized (e.g., dropdown in templates) or freeform, and how will this impact author compliance and review quality?
- What CI/linter rules will be considered a pass vs. fail, and how will these rules evolve as the system and templates change?
- How should we treat legacy artifacts that lack assumption surfacing—require migration, flag as technical debt, or allow exceptions?

- Is per-assumption confidence sufficient, or should we require confidence at plan, task, and artifact level for full traceability?
- What mechanisms are in place to audit, validate, and continuously improve the assumption surfacing process across all artifacts and system versions?

See ADR-OS-022 for details on inventory/buffer assumptions. This pattern is now required for all major system artifacts and decision records.

■# ADR-OS-022: Mechanical Inventory Buffer

- **Status**: Proposed
- **Date**: 2025-06-23
- **Deciders**: Architecture Team
- **Reviewed By**: [TBD]

Context

After contextâ€‘drift events (e.g., agent restart, truncated prompt, or file lock contention) AI Builder agents sometimes regenerate artifacts that already exist, wasting compute and occasionally overwriting newer work. In Eliyahu Goldrattâ€™s *The Goal* the production line solves a similar problem with physical buffers of inventory staged before each workâ€‘centre. We will replicate that idea by giving every **execution plan** and optionally each **task node** an explicit, versioned **mechanical inventory** of reusable items (code snippets, credentials, preâ€‘computed results, file handles, etc.). Agents consume from the buffer instead of reâ€‘creating resources, and Supervisor/Manager agents own mutation rights.

Assumptions

- [ ] Plan-level inventory captures all shared resources required by multiple tasks.
- [ ] Task-level inventory contains only items relevant to that specific node.
- [ ] Inventory state is persisted in Git-tracked annotation blocks; delta logs guarantee crash safety.
- [ ] Garbage-collection keeps annotation files from unbounded growth.
- [ ] The inventory buffer protocol is robust against race conditions, zombie reservations, and log replay errors.
- [ ] The system can detect and recover from inventory/annotation desynchronization or corruption.
- [ ] All compliance requirements from referenced ADRs (e.g., ADR-OS-023, ADR-OS-024, ADR-OS-027, ADR-OS-029, ADR-OS-032) are up-to-date and enforced.

Decision

Twoâ€‘tier Scope *Planâ€‘level* inventory lives in the executionâ€‘plan annotation; *taskâ€‘level* inventory lives in each taskâ€™s annotation block.

Persistent, Fileâ€‘backed Storage Inventory arrays sit inside the existing `EmbeddedAnnotationBlock` payload. On engine restart the orchestrator reâ€‘hydrates from disk and replays `inventory_delta_<g>.log` files.

Optimistic Reservation Protocol First consumer marks an item `RESERVED` with its `agent_id`; the same agent must promote to `CONSUMED` or the janitor will roll back.

Expiration & Garbage Collection Every item must carry `expires_at_g` **or** `ttl_seconds`. A Supervisor janitor loop prunes items every **100 global events** (default; configurable).

Delta Logging Every mutation (create, reserve, consume, expire) is appended to a compact, appendâ€‘only delta file so no updates are lost across crashes.

Access Control *Manager* & *Supervisor* agents have **read/write**; *Builder* agents have **readâ€‘only**. Builder requests to add inventory must be escalated via an `ADD_INVENTORY` task handled by Manager.

Confidence: Medium

Schema Changes (version `2.1`)

```
   // --- Mechanical Inventory Buffer ---
+  "inventory": [
+    {
+      "item_id": "str",                  // unique identifier
+      "item_type": "str",                // CODE_SNIPPET | RESULT | FILE_HANDLE | CREDENTIAL | â€¦
+      "quantity": 1,                       // integer >= 0
+      "meta": { â€¦ },                        // freeâ€‘form metadata
+      "lifecycle_status": "CREATED|RESERVED|CONSUMED|EXPIRED",
+      "reserved_by_agent_id": "str|null",
+      "g_created": 123,
+      "g_last_modified": 125,
+      "expires_at_g": 225,                // OR null
+      "ttl_seconds": 604800               // OR null (one week)
+    }
+  ]|null,
```

`inventory_delta_<g>.log` (appendâ€‘only):

```
<g> CREATED  item_id=snippet_42  item_type=CODE_SNIPPET  qty=1 â€¦
<g> RESERVED item_id=snippet_42  by=agent.build.7
<g> CONSUMED item_id=snippet_42  by=agent.build.7
```

Rationale

1. **Stops Redundant Work** Items fetched once are reâ€'used many times. *Selfâ€'critique*: Could outdated snippets cause stale bugs? *Confidence*: Medium
2. **Versionâ€'Controlled Truth** Git history gives audit & rollback. *Selfâ€'critique*: Commits may bloat; GC mitigates. *Confidence*: High
3. **Crashâ€'Safe via Delta Log** No singleâ€■file commit race. *Selfâ€'critique*: One more control file to manage. *Confidence*: High
4. **Minimal Locking Overhead** Optimistic reservation avoids a lock server. *Selfâ€'critique*: Zombie reservations require janitor cleanup. *Confidence*: Medium

Alternatives Considered

Alternative	Reason Rejected	Confidence
Global shared inventory only	Hard to enforce task isolation, high contention	High
Ephemeral inâ€'memory cache	Lost after restart; canâ€™t audit	Medium
Heavyweight distributed cache (Redis)	Adds infra complexity, violates fileâ€'centric design	High

Distributed Systems Implications

This inventory system is a microcosm of a distributed state machine and MUST adhere to the following policies:

- **Idempotency (ADR-OS-023):** All inventory operations (create, reserve, consume, expire) MUST be idempotent. A retried operation to create `item_id_42` must succeed without creating a duplicate.
- **Asynchronicity (ADR-OS-024):** The `inventory_delta_<g>.log` is an event stream. Agents MUST publish events to this stream asynchronously. A dedicated "Inventory Manager" agent or service should be responsible for consuming this stream and updating the canonical state in the `EmbeddedAnnotationBlock`.
- **Event Ordering (ADR-OS-027):** The g counter in the delta log provides a total ordering. For more complex, causally-related inventory operations, vector clocks MUST be added to the event schema to ensure the state can be correctly reconstructed even if logs are processed out of order.
- **Observability (ADR-OS-029):** Every event written to the delta log MUST be part of a distributed trace. This allows for complete visibility into an item's lifecycle, from creation to consumption, across multiple agents and tasks.

Consequences

- **Positive**: Reduces regeneration churn; enforces deterministic reuse; provides audit trail.
- **Negative**: Slight schema complexity and janitor overhead; larger file diffs.

Clarifying Questions

- Should janitor cadence be configurable per project (e.g., time-based vs. g-based), and what are the trade-offs for different project sizes and agent activity levels?
- What policy governs Builder requests to escalate `ADD_INVENTORY` tasks, and how are these requests audited and approved?
- Do we need a separate retention policy for consumed items, or is garbage collection (GC) sufficient for all inventory types?
- How does the system ensure distributed consistency and recovery from log replay errors, zombie reservations, or annotation desynchronization?
- What is the process for evolving the inventory item schema and protocol as new resource types or lifecycle states are introduced?

See ADR-OS-021 for the overall Assumption-Surfacing pattern. Schema bump from 2.0 → 2.1 is backward compatible (empty inventory arrays are valid under 2.1).

■# ADR-OS-023: Universal Idempotency and Retry Policy

- **Status**: Proposed
- **Date**: YYYY-MM-DD
- **Deciders**: [List of decision-makers]
- **Reviewed By**: [List of reviewers]

Context

The current architecture has some provisions for retries (ADR-OS-011) but lacks a universal, enforceable policy for idempotency and robust retry mechanisms like exponential backoff and circuit breakers. This gap can lead to "retry storms," redundant processing, and inconsistent state when operations are repeated due to transient network failures. This ADR mandates a cross-cutting policy to ensure all external and internal API calls are safe to retry.

Assumptions

- [ ] Standard libraries for implementing exponential backoff and circuit breakers are available and can be integrated into the core engine.
- [ ] The overhead of managing idempotency keys and state is acceptable for the increased reliability.
- [ ] All stateful services can provide a mechanism to detect and reject replayed requests.
- [ ] The idempotency and retry policy is robust against key collisions, replay attacks, and partial failures.
- [ ] The system can detect and recover from retry/circuit breaker misconfiguration or state corruption.
- [ ] All compliance requirements from referenced ADRs (e.g., ADR-OS-032) are up-to-date and enforced.

This section was expanded in response to issue_assumptions.txt to surface implicit assumptions and improve framework compliance.

Decision

Decision:

> We will mandate universal idempotency for all external and internal API calls and enforce a standardized retry policy. 1. **Idempotency:** All mutable API endpoints MUST require an `Idempotency-Key`. The service is responsible for storing the result of the first successful call for that key and returning the cached response for any subsequent retries. 2. **Retry Policy:** All clients (agents, services) initiating network calls MUST wrap them in a retry mechanism that implements exponential backoff with jitter and a circuit breaker pattern. 3. **Standard:** This policy applies to both OS-level internal communications and any external service interactions managed by an agent.

Confidence: High

Rationale

1. **Preventing Duplicate Mutations**
2. Self-critique: Enforcing idempotency adds complexity to both the client (must generate and send keys) and the server (must store and check keys). For some purely read-only operations, this is unnecessary overhead.
3. Confidence: High
4. **Avoiding Retry Storms**
5. Self-critique: A poorly configured circuit breaker (e.g., threshold too low) could trip too easily, reducing availability. Configuration must be carefully managed.
6. Confidence: High
7. **System-wide Consistency**
8. Self-critique: Mandating this everywhere could be overly rigid. There might be rare cases where a different, specialized retry policy is needed. The framework should allow for controlled exceptions.
9. Confidence: Medium

Alternatives Considered

1. **Ad-hoc Implementation**: Allow each agent/service to implement its own retry logic.
2. Brief reason for rejection: Leads to inconsistent behavior, is difficult to audit, and almost guarantees that some components will have naive or dangerous retry logic.
3. Confidence: High
4. **No Retries**: Simply let operations fail and escalate immediately.
5. Brief reason for rejection: Not resilient to transient failures, which are common in distributed systems. This would lead to poor availability and unnecessary escalations.
6. Confidence: High

Consequences

- **Positive:** Drastically improves the system's resilience to transient network failures. Prevents data corruption and inconsistent state from duplicated operations. Creates predictable, system-wide behavior for error handling.
- **Negative:** Increases the implementation complexity for all network-facing components. Requires a shared library or standard for generating idempotency keys and managing retry state.

Clarifying Questions

- What is the standard format and TTL for an idempotency key?
- How are the default parameters for exponential backoff and circuit breakers configured and potentially overridden?
- How do we handle idempotent retries for long-running, multi-step operations (sagas)?

This template integrates explicit assumption-surfacing, confidence indicators, self-critiques, and clarifying questions as per ADR-OS-021.

■# ADR-OS-024: Asynchronous and Eventual Consistency Patterns

- **Status**: Proposed
- **Date**: YYYY-MM-DD
- **Deciders**: [List of decision-makers]
- **Reviewed By**: [List of reviewers]

Context

Most inter-agent and service communication within the OS currently assumes a synchronous, request-response model. This creates tight coupling and can lead to cascading failures and poor latency, especially for operations that are long-running or don't require an immediate response. This ADR defines standard patterns for asynchronous communication to improve system resilience, scalability, and loose coupling.

Assumptions

- [ ] A reliable and scalable message bus or event log technology (e.g., Kafka, NATS, Redis Streams) can be integrated into the OS.
- [ ] Developers and agents can correctly identify which operations are suitable for eventual consistency versus strong consistency.
- [ ] The system can tolerate the time lag inherent in eventually consistent operations.
- [ ] The asynchronous/eventual consistency patterns are robust against message loss, bus outages, and event replay errors.
- [ ] The system can detect and recover from event bus misconfiguration, partitioning, or replay attacks.
- [ ] All compliance requirements from referenced ADRs (e.g., ADR-OS-032) are up-to-date and enforced.

Decision

Decision:

> We will adopt a set of standard patterns for asynchronous and eventually consistent communication between agents and services. 1. **"Fire-and-Forget" Events:** For notifications and non-critical logging, agents SHOULD publish events to a central message bus without waiting for a response. 2. **Sagas for Long-Running Processes:** For multi-step operations that require coordination, the Saga pattern WILL be used. A coordinating agent will emit a series of commands and wait for corresponding "success" or "failure" events, issuing compensating actions for any failures. 3. **Eventual Consistency:** State that does not need to be transactionally consistent (e.g., reporting dashboards, aggregated logs) SHOULD be populated by consumers reading from an event log, embracing eventual consistency.

Confidence: High

Rationale

1. **Decoupling and Resilience**
2. Self-critique: The message bus itself can become a single point of failure. It must be deployed in a highly available configuration.
3. Confidence: High
4. **Improved Performance and Scalability**
5. Self-critique: Asynchronous systems are more complex to debug and reason about, as the execution flow is not linear. Tooling for distributed tracing becomes essential.
6. Confidence: High
7. **Enabling Long-Running Workflows**
8. Self-critique: Implementing sagas correctly is complex, especially the compensating actions for rollbacks. This requires careful design and testing.
9. Confidence: Medium

Alternatives Considered

1. **Synchronous Calls Everywhere**: The current implicit model.
2. Brief reason for rejection: Does not scale, is not resilient to component failures, and leads to poor resource utilization as threads/processes are blocked on I/O.
3. Confidence: High
4. **RPC with Callbacks**: Using direct RPC-style calls with callbacks for completion.
5. Brief reason for rejection: Creates tight, point-to-point coupling between services. The caller needs to know the address of the callee, making the topology rigid.
6. Confidence: High

Consequences

- **Positive:** Creates a more resilient, scalable, and loosely coupled architecture. Improves perceived performance for users/callers as they are not blocked on long-running tasks. Enables complex, multi-step workflows that can survive individual component restarts.
- **Negative:** Increases the operational complexity by adding a message bus component. Requires developers to have a deeper understanding of distributed systems patterns. Makes debugging and end-to-end testing more challenging.

Clarifying Questions

- Which specific message bus technology will be the primary standard for the OS, and how will high availability and failover be ensured?
- What are the mandatory fields, headers, and versioning requirements for all events published to the bus (e.g., trace ID, source agent, schema version)?
- How are "dead letters" (messages that repeatedly fail processing) handled, escalated, and audited for root cause analysis?
- What distributed tracing and debugging tools are required to support asynchronous workflows, and how are trace IDs propagated across event boundaries?
- What is the process for evolving and standardizing async/eventual consistency patterns as new use cases and technologies emerge?

This template integrates explicit assumption-surfacing, confidence indicators, self-critiques, and clarifying questions as per ADR-OS-021.

■# ADR-OS-025: Zero-Trust Internal Security Baseline

- **Status**: Proposed
- **Date**: YYYY-MM-DD
- **Deciders**: [List of decision-makers]
- **Reviewed By**: [List of reviewers]

Context

While ADR-OS-018 establishes a strong security baseline for the OS runtime, it does not explicitly mandate a "zero-trust" posture for internal communications. Agents and services may implicitly trust calls from within the same network, creating a potential vulnerability if one component is compromised. This ADR formalizes a zero-trust model, requiring that no communication is trusted by default, regardless of its origin.

Assumptions

- [ ] A lightweight, fast, and secure mechanism for issuing and validating internal access tokens (e.g., JWT, PASETO) is available.
- [ ] The performance overhead of authenticating and authorizing every internal API call is acceptable.
- [ ] Mutual TLS (mTLS) can be practically implemented across all internal services to encrypt traffic and verify service identity.
- [ ] The zero-trust security model is robust against token forgery, replay attacks, and certificate compromise.
- [ ] The system can detect and recover from token/certificate misconfiguration or authority compromise.
- [ ] All compliance requirements from referenced ADRs (e.g., ADR-OS-018, ADR-OS-032) are up-to-date and enforced.

This section was expanded in response to issue_assumptions.txt to surface implicit assumptions and improve framework compliance.

Decision

Decision:

> We will enforce a zero-trust security model for all internal, agent-to-agent, and service-to-service communication. 1. **Authentication:** Every internal API request MUST present a valid, short-lived authentication token. Services MUST validate this token before processing any request. 2. **Authorization:** Services MUST enforce least-privilege access, authorizing the validated token holder to perform only the requested action on the specific resource. 3. **Encryption:** All internal network traffic MUST be encrypted using mutual TLS (mTLS), ensuring both client and server can verify each other's identity.

Confidence: High

Rationale

1. **Defense in Depth**
2. Self-critique: A compromised token-issuing authority would be a critical failure point, allowing widespread unauthorized access. This component must be exceptionally secure.
3. Confidence: High
4. **Contains Blast Radius**
5. Self-critique: The complexity of managing certificates for mTLS and rotating tokens can be high. This requires significant automation and robust PKI infrastructure.
6. Confidence: Medium
7. **Explicit Trust Boundaries**
8. Self-critique: Moving from implicit trust to explicit zero-trust requires a significant engineering investment and a cultural shift for developers.
9. Confidence: High

Alternatives Considered

1. **Network Perimeter Security**: Relying on firewalls and network segmentation to secure the internal network.
2. Brief reason for rejection: This is a classic "castle-and-moat" model. Once an attacker is inside the perimeter, they can move laterally with little resistance. It is not sufficient for a modern, dynamic system.
3. Confidence: High
4. **API Keys**: Using static, long-lived API keys for service-to-service authentication.
5. Brief reason for rejection: Long-lived credentials are a significant security risk. They are more likely to be leaked and are not easily revocable in case of a breach.
6. Confidence: High

Consequences

- **Positive:** Drastically improves the security posture by eliminating implicit trust. Limits the "blast radius" of a compromised component. Provides a clear, auditable trail of all internal actions.

- **Negative:** Introduces performance overhead for token validation and mTLS handshakes on every call. Increases the complexity of the CI/CD pipeline, which must manage secrets and certificate distribution.

Clarifying Questions

- What token format (e.g., JWT) and which claims are mandatory for internal authentication?
- What is the chosen solution for Certificate Authority and automated certificate rotation for mTLS?
- How are initial "bootstrap" credentials securely delivered to a new agent or service?

This template integrates explicit assumption-surfacing, confidence indicators, self-critiques, and clarifying questions as per ADR-OS-021.

■# ADR-OS-026: Dynamic Topology, Health Checking, and Failure Propagation

- **Status**: Proposed
- **Date**: YYYY-MM-DD
- **Deciders**: [List of decision-makers]
- **Reviewed By**: [List of reviewers]

Context

ADR-OS-012 defines a registry for dynamic agents, but it relies on agents polling for changes. The system lacks a proactive mechanism for service discovery, standardized health checks, and propagating status changes (e.g., an agent becoming unhealthy) to dependent components. This ADR establishes a formal protocol for managing the lifecycle and health of agents and services within the OS.

Assumptions

- [ ] All agents and services can expose a standardized HTTP `/health` endpoint.
- [ ] The central agent registry can handle the write load of frequent heartbeat updates from all active agents.
- [ ] The underlying network supports a lightweight subscription model (e.g., WebSockets, gRPC streams) for propagating status changes.
- [ ] The health checking and failure propagation protocol is robust against registry outages, network partitions, and heartbeat loss.
- [ ] The system can detect and recover from registry/health monitor misconfiguration or overload.
- [ ] All compliance requirements from referenced ADRs (e.g., ADR-OS-012, ADR-OS-032) are up-to-date and enforced.

This section was expanded in response to issue_assumptions.txt to surface implicit assumptions and improve framework compliance.

Decision

Decision:

> We will implement a standardized protocol for dynamic topology, health checking, and status propagation. 1. **Service Discovery:** The `agent_registry.txt` (from ADR-OS-012) will be the single source of truth for available services. Agents will register themselves upon startup. 2. **Health Checking:** Every agent/service MUST expose a `/health` endpoint that returns its operational status (`UP`, `DEGRADED`, `DOWN`). A central health monitor will periodically poll these endpoints. 3. **Heartbeating:** Every active agent MUST send a regular heartbeat to the agent registry to signal its liveness. Failure to receive a heartbeat within a configured interval will mark the agent as `UNHEALTHY`. 4. **Failure Propagation:** The agent registry will expose a subscription-based stream of status changes. Supervisor agents and other interested components MUST subscribe to this stream to be immediately notified when a dependency's status changes.

Confidence: High

Rationale

1. **Proactive Failure Detection**
2. Self-critique: Centralized health checking can be a bottleneck and a single point of failure. A decentralized, peer-to-peer gossip protocol could be more resilient but is significantly more complex to implement.
3. Confidence: High
4. **Dynamic System Topology**
5. Self-critique: Relying on a single registry for discovery couples all services to it. If the registry is down, no new connections can be made.
6. Confidence: Medium
7. **Reduces Stale State**
8. Self-critique: The "correct" interval for health checks and heartbeats is a difficult tuning problem. Too frequent, and it creates excess network traffic; too infrequent, and the system is slow to react to failures.
9. Confidence: High

Alternatives Considered

1. **DNS-based Discovery**: Using DNS SRV records for service discovery.
2. Brief reason for rejection: DNS can have high propagation delays (due to caching), making it slow to react to changes in topology. It doesn't include a built-in health checking or status propagation mechanism.
3. Confidence: High
4. **Manual Configuration**: Hardcoding agent addresses in configuration files.
5. Brief reason for rejection: Extremely brittle and static. Does not support dynamic scaling or failover. This is the problem ADR-OS-012 was created to solve.
6. Confidence: High

Consequences

- **Positive:** Creates a robust and self-healing system that can react quickly to component failures. Enables dynamic scaling of agents without manual reconfiguration. Provides a clear, centralized view of the health of the entire system.
- **Negative:** Adds network traffic due to health checks and heartbeats. Requires a highly available agent registry and health monitoring service. Increases the complexity of the agent lifecycle.

Clarifying Questions

- What is the standard schema and versioning strategy for the `/health` endpoint response, and how are custom health signals supported?
- What is the protocol, failover, and security model for the status update subscription stream (e.g., gRPC, WebSockets), and how are missed updates handled?
- How do we prevent a "thundering herd" problem when a critical service comes back online and all subscribers try to connect at once?
- What mechanisms are in place to ensure high availability, failover, and recovery for the agent registry and health monitoring service?
- How are health check and heartbeat intervals tuned, monitored, and adapted to balance responsiveness and network overhead in different deployment scenarios?

This template integrates explicit assumption-surfacing, confidence indicators, self-critiques, and clarifying questions as per ADR-OS-021.

■# ADR-OS-027: Global and Vector Clock Event Ordering

- **Status**: Proposed
- **Date**: YYYY-MM-DD
- **Deciders**: [List of decision-makers]
- **Reviewed By**: [List of reviewers]

Context

The OS currently uses a global event counter (g) to provide a total ordering of events. While simple, this centralized counter can become a bottleneck and does not adequately capture the causal relationships between events in a distributed system (e.g., event B was caused by event A, even if they occurred on different agents). This ADR specifies when to use the global counter versus more sophisticated logical clocks to preserve causality.

Assumptions

- [ ] The overhead of maintaining and transmitting vector clocks is acceptable for the operations that require strict causal ordering.
- [ ] Developers and agents can correctly identify workflows where simple ordering is insufficient and causal history is required.
- [ ] A standard library for implementing Lamport timestamps and vector clocks is available.
- [ ] The event ordering policy is robust against clock drift, agent restarts, and vector clock overflows.
- [ ] The system can detect and recover from event ordering/counter misconfiguration or corruption.
- [ ] All compliance requirements from referenced ADRs (e.g., ADR-OS-032) are up-to-date and enforced.

This section was expanded in response to issue_assumptions.txt to surface implicit assumptions and improve framework compliance.

Decision

Decision:

> We will augment the existing global counter with a formal policy for using logical clocks where causality is critical. 1. **Global Counter (g):** The default mechanism for timestamping and ordering events where a simple, total order is sufficient. It remains the standard for artifact versioning and logging. 2. **Lamport Timestamps:** For distributed workflows that require a total ordering of events without a central coordinator (e.g., distributed transactions), Lamport timestamps MUST be used. 3. **Vector Clocks:** For workflows where it is essential to know if an event A "happened before" event B (causality), vector clocks MUST be attached to all relevant events and messages. This is critical for debugging and for systems that need to reconcile concurrent, independent state changes.

Confidence: Medium

Rationale

1. **Preserving Causality**
2. Self-critique: Vector clocks can grow large in systems with many agents, adding significant overhead to every message. Their use should be limited to where they are strictly necessary.
3. Confidence: High
4. **Enabling Correct State Reconciliation**
5. Self-critique: Reasoning about vector clocks is notoriously difficult for humans. This increases the cognitive load on developers and makes debugging more complex if not supported by good tooling.
6. Confidence: High
7. **Choosing the Right Tool for the Job**
8. Self-critique: The policy requires agents/developers to make a nuanced choice between three different clock types. Incorrectly choosing a weaker clock could lead to subtle and hard-to-diagnose bugs.
9. Confidence: Medium

Alternatives Considered

1. **Global Counter Everywhere**: The current model.
2. Brief reason for rejection: Does not capture causality, which is a fundamental requirement for reasoning about and debugging distributed workflows. It can lead to incorrect state reconciliation.
3. Confidence: High
4. **Physical Clocks (NTP)**: Relying on synchronized physical clocks to order events.
5. Brief reason for rejection: Clock skew between servers makes physical clocks unreliable for determining the precise order of events in a distributed system. Logical clocks are designed to solve this problem.
6. Confidence: High

Consequences

- **Positive:** Enables the system to correctly reason about causal relationships between events. Prevents a large class of subtle bugs related to event ordering and concurrent state updates. Provides a robust foundation for building more complex, reliable distributed workflows.
- **Negative:** Increases the complexity of the event/messaging infrastructure. Adds overhead to messages carrying vector clocks. Requires developers to be trained on the proper use of logical clocks.

Clarifying Questions

- What is the standard library and data structure for representing vector clocks?
- How does the system handle a "new" agent joining and initializing its vector clock?
- What visualization tools will be provided to help developers debug causal event histories?

This template integrates explicit assumption-surfacing, confidence indicators, self-critiques, and clarifying questions as per ADR-OS-021.

■# ADR-OS-028: Partition Tolerance and Split-Brain Protocol

- **Status**: Proposed
- **Date**: YYYY-MM-DD
- **Deciders**: [List of decision-makers]
- **Reviewed By**: [List of reviewers]

Context

The current architecture does not have an explicit, system-wide protocol for how to behave during a network partition. Different components might make different choices, leading to inconsistent state and a "split-brain" scenario where different parts of the system operate on conflicting data. This ADR formalizes the trade-offs (per the CAP theorem) for each major component and defines a protocol for detection, behavior during a partition, and state reconciliation after the partition heals.

Assumptions

- [ ] The system has a reliable mechanism (e.g., from ADR-026) to detect network partitions.
- [ ] For any stateful component, we can clearly define whether it should prioritize Consistency (CP) or Availability (AP) during a partition.
- [ ] Automated tools can be built to assist in the reconciliation of conflicting states after a partition is resolved.
- [ ] The partition tolerance and reconciliation protocol is robust against split-brain, quorum loss, and reconciliation errors.
- [ ] The system can detect and recover from partition protocol misconfiguration or consensus library failures.
- [ ] All compliance requirements from referenced ADRs (e.g., ADR-OS-032) are up-to-date and enforced.

Decision

Decision:

> We will implement a formal protocol for network partition tolerance. 1. **Explicit CAP Trade-off:** Every stateful service or component (e.g., agent registry, plan executor) MUST declare its strategy in the face of a partition: * **CP (Consistency/Partition Tolerance):** The component will become unavailable (e.g., read-only or fully down) rather than serve potentially stale or conflicting data. This is the default for critical state like the agent registry. * **AP (Availability/Partition Tolerance):** The component will remain available for reads and writes, accepting that its state may diverge from other partitions. This is suitable for non-critical, eventually consistent data like logs. 2. **Split-Brain Prevention:** For all CP components, a quorum-based mechanism (e.g., Raft, Paxos) MUST be used to ensure that only the majority partition can accept writes, preventing a split-brain scenario. 3. **State Reconciliation:** For AP components, a formal, semi-automated process for reconciling divergent data MUST be defined. This may involve "last-write-wins" strategies, CRDTs, or escalating to a human operator for manual merging.

Confidence: High

Rationale

1. **Guarantees Predictable Behavior**
2. Self-critique: Implementing quorum-based consensus protocols like Raft is extremely difficult and complex. Using a proven, off-the-shelf library (e.g., etcd, Zookeeper) is a much safer approach than building our own.
3. Confidence: High
4. **Prevents Data Corruption**
5. Self-critique: For some complex data types, automatic reconciliation is impossible. The plan for manual escalation must be robust and well-documented.
6. Confidence: High
7. **Makes Trade-offs Explicit**
8. Self-critique: Forcing a binary CP/AP choice might be too simplistic for some components that could offer more nuanced behavior (e.g., allow stale reads but block writes).
9. Confidence: Medium

Alternatives Considered

1. **Assume No Partitions**: Ignoring the possibility of network partitions.
2. Brief reason for rejection: In any real-world distributed system, partitions are a certainty. Ignoring them guarantees data corruption and unpredictable failures.
3. Confidence: High
4. **Always Prioritize Availability (AP)**: Let all components remain writeable during a partition.
5. Brief reason for rejection: This inevitably leads to split-brain scenarios for critical state, resulting in data loss and a system that is impossible to reason about.
6. Confidence: High

Consequences

- **Positive:** Makes the system resilient to network partitions. Prevents catastrophic data corruption due to split-brain. Forces a clear, conscious decision about the trade-offs between consistency and availability for every component.
- **Negative:** Significantly increases the complexity of the architecture, especially for CP components that require a consensus system. The reconciliation process for AP components can be complex to design and implement.

Clarifying Questions

- What is the standard consensus library/protocol (e.g., Raft via etcd) that will be used for CP components, and how will upgrades or migrations be managed?
- What specific data reconciliation strategies (e.g., last-write-wins, CRDTs, manual merge) will be used for the initial set of AP components, and how are conflicts audited and resolved?
- How are clients (agents) made aware that a service is currently unavailable due to a partition, and what is the failover or retry policy?
- What mechanisms are in place for partition detection, recovery, and audit of partition events across the system?
- How will the partition tolerance protocol and component CAP trade-offs be evolved as new requirements or technologies emerge?

This template integrates explicit assumption-surfacing, confidence indicators, self-critiques, and clarifying questions as per ADR-OS-021.

■# ADR-OS-029: Universal Observability and Trace Propagation

- **Status**: Proposed
- **Date**: YYYY-MM-DD
- **Deciders**: [List of decision-makers]
- **Reviewed By**: [List of reviewers]

Context

While ADR-OS-019 establishes a baseline for observability, it does not mandate a universal standard for propagating trace context across agent and service boundaries, especially in asynchronous workflows. Without a standard, it becomes nearly impossible to reconstruct the full end-to-end lifecycle of a request as it flows through multiple components, making debugging and performance analysis extremely difficult.

Assumptions

- [ ] A standard for trace context propagation (e.g., W3C Trace Context) can be adopted and enforced across all services.
- [ ] The chosen tracing library can automatically instrument both synchronous (HTTP) and asynchronous (message bus) communications.
- [ ] The performance overhead of generating, propagating, and exporting trace data is acceptable for production workloads.
- [ ] The observability and trace propagation policy is robust against trace loss, sampling errors, and context breakage.
- [ ] The system can detect and recover from tracing misconfiguration or backend outages.
- [ ] All compliance requirements from referenced ADRs (e.g., ADR-OS-019, ADR-OS-026, ADR-OS-032) are up-to-date and enforced.

Decision

Decision:

> We will mandate universal, cross-component trace propagation and a minimum standard for metrics and logs for all orchestrated actions. 1. **Trace ID Propagation:** Every request entering the system will be assigned a unique `trace_id`. This `trace_id` MUST be propagated through all subsequent synchronous calls (e.g., in HTTP headers) and asynchronous messages (e.g., in message headers). 2. **Span Generation:** Every component (agent, service) processing a request MUST generate a "span" representing its unit of work, linking it to the parent span via the propagated trace context. 3. **Mandatory Metrics/Logs:** Every orchestrated action must emit a minimum set of structured logs and metrics (e.g., start time, end time, duration, status, `trace_id`), in addition to the heartbeats defined in ADR-026.

Confidence: High

Rationale

1. **End-to-End Visibility**
2. Self-critique: Full, high-fidelity tracing can generate a massive volume of data, which can be expensive to store and process. Sampling strategies may be necessary for high-traffic services.
3. Confidence: High
4. **Simplified Debugging**
5. Self-critique: If a single component in the chain fails to propagate the trace context, the trace is broken. This requires 100% compliance, which can be hard to enforce across a diverse set of services.
6. Confidence: High
7. **Performance Analysis**
8. Self-critique: The act of instrumenting code and exporting trace data adds a small amount of latency to every operation. This must be measured and monitored.
9. Confidence: High

Alternatives Considered

1. **Log Correlation**: Relying on manually correlating logs from different services using a shared request ID.
2. Brief reason for rejection: Brittle, labor-intensive, and often impossible to do correctly in complex, asynchronous workflows. It doesn't provide the rich parent-child relationship and timing information of a proper trace.
3. Confidence: High
4. **Per-Service Observability**: Allowing each service to have its own observability strategy without a shared context.
5. Brief reason for rejection: This creates isolated silos of information, making it impossible to get a holistic view of the system's behavior.
6. Confidence: High

Consequences

- **Positive:** Provides complete, end-to-end visibility into request flows across the entire system. Dramatically simplifies the process of debugging production issues and identifying performance bottlenecks.

- **Negative:** Requires all components to be integrated with a standard tracing library. Adds a dependency on a distributed tracing backend (e.g., Jaeger, Zipkin). Can generate a large volume of telemetry data that must be managed.

Clarifying Questions

- What is the standard tracing library and backend (e.g., OpenTelemetry SDK with Jaeger exporter) to be used, and how will upgrades or migrations be managed?
- What is the policy for trace sampling in high-volume services, and how is sampling adapted to balance cost, fidelity, and performance?
- What specific tags and attributes are mandatory for all generated spans, and how is the span schema versioned and evolved?
- What mechanisms are in place to enforce and audit compliance with trace propagation and span generation across all components?
- How is trace data retention, storage, and privacy managed, especially for sensitive or regulated environments?

This template integrates explicit assumption-surfacing, confidence indicators, self-critiques, and clarifying questions as per ADR-OS-021.

■# ADR-OS-030: Archetypal Agent Roles and Protocols

- **Status:** Proposed
- **Date:** 2025-06-23
- **Deciders:** Architecture Team
- **Reviewed By:** [TBD]

Context

The Hybrid AI Orchestration System relies on agents (â€œpersonasâ€■) with specific operational responsibilities. Until now, agent roles, access boundaries, and escalation protocols have been loosely defined, leading to role drift, privilege creep, and ad hoc recovery flows.

To achieve deterministic, auditable, and robust distributed operation, we must define a small, closed set of *archetypal agents* with locked behavioral contracts, access rights, escalation/fallback paths, and lifecycle protocols. This aligns with both distributed system best practices and Goldrattâ€™s â€œwork-centerâ€■ specialization: every function and failure must have one, and only one, responsible agent type.

Assumptions

- [ ] Only five core agent archetypes are required at present (Supervisor, Manager, Builder, Janitor, Auditor).
- [ ] Each agent archetype is protocol-defined (not just named), with explicit permissions and forbidden actions.
- [ ] No agent may cross its boundaries except via explicit, protocol-defined escalation or fallback.
- [ ] Extensions or hybrid roles must be proposed via a future ADR and pass architectural review.
- [ ] The agent role and protocol system is robust against privilege escalation, role drift, and protocol misconfiguration.
- [ ] The system can detect and recover from agent registry or protocol definition errors.
- [ ] All compliance requirements from referenced ADRs (e.g., ADR-OS-012, ADR-OS-032) are up-to-date and enforced.

This section was expanded in response to issue_assumptions.txt to surface implicit assumptions and improve framework compliance.

Decision

We hereby formalize the following archetypal agents, their protocols, and their access boundaries:

1. Supervisor

- **Role:** Final authority for orchestration, recovery, plan approval, and override.

  Permissions:

  R/W all plans, inventories, agent states, and constraints.

- Can unlock or forcibly recover any system state.

  Forbidden:

  Cannot directly execute atomic Builder tasks.

  Escalation/Fallback:

  All critical errors, deadlocks, and â€œunownedâ€■ failures escalate to Supervisor.

  Lifecycle:

  Must always be available (human or highly privileged AI).

2. Manager

- **Role:** Task delegation, plan-level orchestration, inventory mutation, and agent assignment.

  Permissions:

  R/W to plan artifacts and inventory.

- Can assign and promote/demote Builder, Janitor, Auditor agents.

  Forbidden:

  Cannot override locked constraints (must escalate to Supervisor).

- Cannot GC/expire inventory (Janitor only).

  Escalation/Fallback:

  Any denied inventory mutation escalates to Supervisor.

3. Builder

- **Role:** Execute atomic tasks; does not mutate shared state except working artifacts.

  Permissions:

  R to plans, tasks, inventory.

- W to designated working artifact only.

  Forbidden:

  Cannot mutate plan-level inventory, state, or promote/demote other agents.

  Escalation/Fallback:

  If required resource is missing, must request via Manager.

4. Janitor

- **Role:** Automated GC, buffer cleanup, health/heartbeat tasks.

  Permissions:

  R to all state.

- W only to expired/orphaned items (inventory, plans).

  Forbidden:

  Cannot assign/execute Builder tasks or mutate active artifacts.

  Escalation/Fallback:

  Reports GC errors to Manager.

5. Auditor

- **Role:** System audit, log reading, trace/metrics analysis.

  Permissions:

  R all logs, plans, state, events.

  Forbidden:

  No mutation of any artifact or agent.

  Escalation/Fallback:

  May recommend but not execute recovery.

Rationale

- **Role purity:** Prevents privilege creep, confusion, and accidental state mutation.
- **Predictable escalation:** System failures and gaps are never â€œunownedâ€■â€”every error escalates through a fixed path.
- **Access control:** Every plan, inventory, and annotation block operation is authorized only if the requesting agent matches protocol.
- **Extensible:** New archetypes only by explicit ADR.

Protocol Diagrams

1. Access Control Matrix

| Agent | Plan Artifact | Task Artifact | Inventory | Agent Registry | Logs/Events | GC/Orphans | |------------|:------------:|:-------------:|:---------:|:-------------:|:-----------:|:----------:| | Supervisor | R / W | R / W | R / W | R / W | R / W | R / W | | Manager | R / W | R / W | R / W | R / W | R | R | | Builder | R | R / W | R | R | R | - | | Janitor | R | R | W (expired/orphaned only) | R | W (GC events) | W | | Auditor | R | R | R | R | R | R | R = Read, W = Write

Builder only writes to assigned task artifacts, never plan/inventory. Janitor writes only to expired/orphaned items (via GC protocol).

1. Escalation & Fallback Flow

Agent Escalation Paths

flowchart TD B[Builder] M[Manager] J[Janitor] S[Supervisor] A[Auditor]

```
B -- "Resource missing or forbidden" --> M
M -- "Denied or locked action" --> S
J -- "GC error or orphan unresolved" --> M
M -- "Cannot resolve/approve" --> S
A -- "Audit report/issue" --> M
M -- "If unavailable" --> S
J -- "If Manager unavailable" --> S
```

Explanation: Builder escalates all forbidden actions or missing resources to Manager. Manager escalates denied/locked actions to Supervisor. Janitor reports GC issues to Manager (or Supervisor if Manager is unavailable). Auditor can only recommend or report, not escalate state changes directly.

1. Agent-to-System Interaction Map

graph TD Supervisor -- R/W --> PlanArtifact Supervisor -- R/W --> Inventory Supervisor -- R/W --> AgentRegistry Supervisor -- R/W --> LogsEvents

```
Manager -- R/W --> PlanArtifact
Manager -- R/W --> Inventory
Manager -- R/W --> AgentRegistry

Builder -- R --> PlanArtifact
Builder -- R/W --> TaskArtifact
Builder -- R --> Inventory

Janitor -- R --> PlanArtifact
Janitor -- W (expired/orphaned) --> Inventory
Janitor -- W --> LogsEvents

Auditor -- R --> PlanArtifact
Auditor -- R --> Inventory
Auditor -- R --> AgentRegistry
Auditor -- R --> LogsEvents
```

1. Minimal State Machine for Builder

stateDiagram-v2 [*] --> Idle Idle --> RequestTask: Assigned task RequestTask --> Working: Starts execution Working --> Done: Completes execution Working --> NeedsHelp: Missing resource/forbidden NeedsHelp --> Waiting: Escalates to Manager Waiting --> Working: Receives help/resource Done --> Idle

---

Example Schema Contracts

```
// Example: Builder agent_card excerpt
{
 "persona_id": "builder_XYZ",
 "archetype": "BUILDER",
 "allowed_actions": ["READ_PLAN", "EXECUTE_TASK", "READ_INVENTORY"],
 "forbidden_actions": ["MUTATE_PLAN", "MUTATE_INVENTORY", "PROMOTE_AGENT"],
 "escalation_path": "MANAGER",
 "access_scope": {
   "plan": "READ",
   "task": "READ/WRITE",
   "inventory": "READ"
 }
}
```

Alternatives Considered

- **Loose, flexible personas:** Led to privilege creep and accidental cross-role mutations.

- **Code-only enforcement:** Lacked clarity for documentation, onboarding, and review.

Consequences

- **Positive:** Tighter system safety, auditability, and onboarding clarity.
- **Negative:** Slight rigidityâ€"new workflows must justify new roles in future ADRs.

Clarifying Questions

- Should archetype enforcement be implemented in code, config, or both?
- How are temporary hybrid roles handled during migration periods?
- Do escalation paths require explicit fallback (if Manager down, escalate to Supervisor, etc.)?

This ADR will govern all future agent design, schema, and protocol work. Any new agent type or capability requires explicit ADR review.

ADR-OS-031: Pre-Initiative Source Artifact Standards

Status: Proposed Date: 2025-06-23 Deciders: Architecture Team Reviewed By: [TBD]

Context

Project failures and execution drift are often caused not by poor implementation, but by missing, ambiguous, or inconsistent source artifacts before execution even begins. To ensure robust, agent- and human-driven execution, we mandate a high standard for all upstream source documents (the pre-initiative phase) which define, constrain, and derisk every downstream plan.

Assumptions

- [ ] Execution plans and initiative docs require clearly defined, agent-parsable, and fully explicit upstream artifacts.
- [ ] Oral or ad hoc documentation is forbidden as a basis for initiative planning.
- [ ] Explicit assumption surfacing, confidence/self-critique, and diagramming are as mandatory as text sections.
- [ ] Framework/model compliance must be stated and justified in every artifact.
- [ ] The pre-initiative artifact standard is robust against missing, ambiguous, or inconsistent documentation.
- [ ] The system can detect and recover from artifact/section omissions or schema drift.
- [ ] All compliance requirements from referenced ADRs (e.g., ADR-OS-021, ADR-OS-032) are up-to-date and enforced.

This section was expanded in response to issue_assumptions.txt to surface implicit assumptions and improve framework compliance.

Decision

We mandate that every pre-initiative effort must include, at minimum, the following structured source artifacts, each agent- and human-consumable:

1. **Vision/North Star**

   Brief statement of why, success criteria, and primary stakeholders.

   PRD (Product Requirements Doc) / MRD (Market Requirements Doc)

4. User, business, and stakeholder requirements (JTBD, user stories, constraints, personas).

   Explicit non-negotiables and legal/compliance constraints.

   TRD (Technical Requirements Doc) / Architecture Overview

7. Context/system diagrams.
8. Stack choices, tradeoff rationale, integration/interface points.
9. DS constraints (consistency, partition tolerance, etc.).

   Security and threat models.

   Design System & Pattern Registry (if applicable)

12. Design principles (KISS, DRY, a11y, etc.).

   Pattern/component catalog (with references to diagrams, visual assets).

   Assumption & Constraint Register

   List of all explicit assumptions, each with confidence, self-critique, and mitigation.

   Execution/Delivery Planning Outline

   Major milestones, dependencies, responsible archetypes/roles, timeline (skeleton OK pre-approval).

   Diagram & Protocol Pack

19. State diagrams, sequence diagrams, flow/data models.
20. All must be referenced by section in other artifacts.

Minimum agent-compliance rules:

- All artifacts are stored as agent-readable Markdown (or convertible schema), never locked PDF/image.
- Each artifact includes a Frameworks/Models Applied section, listing all governing patterns (e.g., KISS, ToC, JTBD, etc.), and proof-of-compliance.
- Each artifact must surface clarifying questions and open issues at submission.
- CI/lint must fail merges if required artifacts, sections, or compliance fields are missing.

Rationale

- **No artifact, no execution:** Initiatives must not begin until upstream context is fully explicit, de-risked, and agent-parsable.
- **All context is agent-loadable:** Eliminates unexplained implicits and guarantees continuity for both human and automated agents.
- **Derisking is enforced:** Early surfacing of assumptions, constraints, and diagrams removes the most common causes of failure and scope drift.

Alternatives Considered

- **Lean/just-in-time docs:** Led to chronic drift, poor agent onboarding, and inconsistent human handoff.
- **PDF/slide decks:** Unparsable by agents; caused missed context and downstream translation errors.

Consequences

- **Positive:** Maximal clarity, minimal unexplained implicit context, easier review/onboarding, less rework.
- **Negative:** Slight upfront authoring friction, but offset by sharply reduced project risk and scope drift.

Clarifying Questions

- Should CI require diagram/visual compliance as strictly as text sections?
- Is there a minimal, fast-path artifact set for ultra-small/trivial initiatives?
- How often must source artifacts be reviewed/updated during long initiatives?

This ADR is foundational: downstream plans, PRs, and initiative launches will be rejected if they do not link to a complete, agent- and human-compliant pre-initiative artifact set as defined here.

ADR-OS-032: Canonical Models and Frameworks Registry & Enforcement

Status: Proposed Date: 2025-06-23 Deciders: Architecture Team Reviewed By: [TBD]

Context

Most software projects claim to follow best practices, but drift, partial adoption, or ambiguous cultural norms lead to silent technical debt and regressions. To achieve consistently high standards, de-risking, and legible execution, this system formalizes a Registry of Canonical Models and Frameworks. All major artifacts (ADRs, roadmaps, plans, schemas, designs, test specs, etc.) must explicitly declare which models/frameworks they apply, how compliance is demonstrated, and what critiques or exceptions exist.

Assumptions

- [ ] Best-practice models/frameworks must be explicitly cataloged and referenced, not assumed.
- [ ] Agents and humans both need to reference and apply these models/frameworks in reasoning, review, and execution.
- [ ] Compliance with models/frameworks is provable (via logic, critique, code, diagrams, or enforcement).
- [ ] Word contracts alone are insufficient; visual, structural, and behavioral contracts (diagrams, test patterns, checklists) are also mandatory.
- [ ] The models/frameworks registry and enforcement process is robust against drift, omission, and schema evolution.
- [ ] The system can detect and recover from registry/annotation omissions or compliance failures.
- [ ] All compliance requirements from referenced ADRs (e.g., ADR-OS-021) are up-to-date and enforced.

This section was expanded in response to issue_assumptions.txt to surface implicit assumptions and improve framework compliance.

Decision

We create and maintain a versioned, agent-readable Registry of Canonical Models and Frameworks, including but not limited to:

- Distributed Systems Principles (e.g., The Network Is Not Reliable, CAP Theorem)
- Theory of Constraints (ToC)
- AAA Testing (Arrange-Act-Assert)
- KISS (Keep It Simple, Stupid)
- DRY (Dont Repeat Yourself)
- Assumption Surfacing / Second Order Thinking
- Jobs-To-Be-Done (JTBD)
- User Stories
- Self-Critique
- Explicit Diagramming (state, flow, data, sequence, etc.)
- [Any additional models, named and versioned]

Registry Requirements

Each model/framework entry includes:

- **Name & Version:** E.g., KISS v1.0
- **Definition:** What is required/forbidden
- **Compliance Heuristics:** What counts as proof (diagram, code, linter, etc.)
- **Enforcement Mode:** Required, Recommended, Optional
- **Agent/Inventory Representation:** Schema for agent/toolkit use

Artifact Compliance Rules

Every major artifact (ADR, plan, doc, schema, test, etc.):

- Lists all governing models/frameworks
- Surfaces, for each, how compliance is achieved/proven (with self-critique)
- Flags any exceptions or non-compliance with justification
- Links to registry entry (with version)
- Includes diagrams/visuals when required by the model
- CI/linter must fail merges if required frameworks are omitted, unproven, or unjustified.

Rationale

- **No cultural slop:** Every practice is explicit, versioned, and auditable.
- **Agent/automation ready:** Agents can reference and enforce models/frameworks in code or reviews.

- **Evolvable:** Models/frameworks can be updated, deprecated, or versioned as rigor or context grows.

Alternatives Considered

- **Team culture/docstrings only:** Led to silent drift, partial adoption, and inconsistent agent reasoning.
- **Enforcement by code only:** Misses design and non-code artifacts; does not serve human review.

Consequences

- **Positive:**
- Architectural, design, and execution standards become legible and enforceable for humans and agents.
- Faster onboarding, higher rigor, and safer automation.
- **Negative:**
- Upfront documentation cost; must maintain registry and artifact annotations.

Clarifying Questions

- Should some models/frameworks be system mandatory (cannot be omitted)?
- Who owns and updates the registry?
- How are exceptions (temporary non-compliance) tracked and resolved?

This ADR is a system requirement: all downstream ADRs, roadmaps, plans, schemas, and designs must reference and comply with at least the applicable models/frameworks in the registry, or surface their non-compliance with critique and rationale.

ANNOTATION_BLOCK_START

{ "artifact_annotation_header": { "artifact_id_of_host": "adr_os_033_md", "g_annotation_created": 252, "version_tag_of_host_at_annotation": "1.0.0" }, "payload": { "description": "Defines the Cookbook & Recipe Management system for capturing, validating, and reusing implementation patterns across HAiOS projects.", "artifact_type": "DOCUMENTATION", "purpose_statement": "To establish a formal system for capturing proven implementation patterns as reusable recipes, ensuring DRY principles and consistent quality across all HAiOS implementations.", "authors_and_contributors": [{ "g_contribution": 252, "identifier": "Hybrid_AI_OS" }, { "g_contribution": 250, "identifier": "Third_Party_Architectural_Review" }], "internal_dependencies": ["adr_os_template_md", "adr_os_032_md", "adr_os_021_md", "3rdpartyeval-10.md"], "linked_issue_ids": [] } }

ANNOTATION_BLOCK_END

ADR-OS-033: Cookbook & Recipe Management System

- **Status**: Proposed
- **Date**: 2025-01-28
- **Deciders**: [List of decision-makers]
- **Reviewed By**: [List of reviewers]

Context

HAiOS agents repeatedly implement similar patterns across different projects and contexts - API client configurations, error handling strategies, testing frameworks, deployment patterns, and architectural components. Currently, these patterns are either:

1. **Reimplemented from scratch** each time, leading to inconsistency and wasted effort
2. **Copied informally** without validation or standardization
3. **Lost** when projects end, creating institutional knowledge gaps

This violates core architectural principles (DRY, KISS) and creates quality inconsistencies across the HAiOS ecosystem. Additionally, the system lacks a formal mechanism for capturing, validating, and evolving proven implementation patterns, leading to repeated architectural decisions and potential regression of quality standards.

The current `docs/cookbook/` directory exists but lacks formal structure, validation processes, or integration with the broader HAiOS governance model.

Assumptions

- [ ] Implementation patterns can be abstracted into reusable templates without losing essential context or becoming overly generic.
- [ ] The Recipe validation process can distinguish between high-quality, proven patterns and experimental or context-specific implementations.
- [ ] Agents can be trained/configured to consistently discover and apply relevant Recipes during implementation tasks.
- [ ] The Recipe format can capture sufficient metadata (prerequisites, constraints, alternatives) to enable safe reuse across different contexts.
- [ ] Recipe versioning and evolution can be managed without breaking existing implementations that depend on older Recipe versions.
- [ ] The Cookbook system can integrate with existing HAiOS governance (ADR compliance, testing requirements) without creating excessive overhead.
- [ ] Recipe discovery and selection can be made efficient enough to not slow down implementation tasks.
- [ ] The system can handle Recipe conflicts when multiple Recipes could apply to the same implementation scenario.

This section was expanded to surface implicit assumptions about pattern abstraction, validation sophistication, agent training, and system integration complexity.

Frameworks/Models Applied

This ADR applies the following canonical models and frameworks (per ADR-OS-032):

DRY (Don't Repeat Yourself) v1.0

- **Compliance Proof:** Recipe system explicitly captures and reuses proven implementation patterns, eliminating redundant implementation work across projects.
- **Self-Critique:** Risk of over-abstraction leading to Recipes that are too generic to be useful or too specific to be reusable.

KISS (Keep It Simple, Stupid) v1.0

- **Compliance Proof:** Recipe format focuses on essential implementation details with clear, minimal structure for maximum usability.
- **Self-Critique:** Balancing simplicity with comprehensive metadata requirements may create tension between usability and completeness.

Single Source of Truth v1.0

- **Compliance Proof:** Cookbook serves as the authoritative source for all validated implementation patterns, preventing conflicting or outdated pattern usage.

- **Self-Critique:** Recipe versioning complexity could undermine single source of truth if multiple versions create confusion about which is authoritative.

Quality Assurance v1.0

- **Compliance Proof:** Recipe validation process ensures only proven, tested patterns enter the Cookbook, maintaining consistent quality standards.
- **Self-Critique:** Validation process must be rigorous enough to catch quality issues while not being so strict as to reject innovative but valid patterns.

Evidence-Based Development v1.0

- **Compliance Proof:** All Recipes must include evidence of successful implementation and testing before canonization.
- **Self-Critique:** Evidence requirements must be comprehensive; insufficient evidence validation could allow unproven patterns to be treated as authoritative.

Separation of Concerns v1.0

- **Compliance Proof:** Clear separation between Recipe definition (what), Recipe application (how), and Recipe validation (quality assurance).
- **Self-Critique:** Integration points between Recipe system and existing HAiOS components may blur separation boundaries.

Audit Trail v1.0

- **Compliance Proof:** Complete lineage tracking from pattern discovery through validation to Recipe creation and usage across projects.
- **Self-Critique:** Comprehensive audit trails may become voluminous and require sophisticated tooling for effective navigation and analysis.

Assumption Surfacing v1.0

- **Compliance Proof:** Eight explicit assumptions about pattern abstraction, validation processes, agent integration, and system complexity.
- **Self-Critique:** Recipe system likely has additional implicit assumptions about pattern categorization and cross-project applicability.

Decision

Decision:

We will implement a **Cookbook & Recipe Management System** that formalizes the capture, validation, and reuse of proven implementation patterns across all HAiOS projects. This system will enforce DRY principles while maintaining quality standards through a structured validation process.

Recipe Structure & Format

Recipe Definition Schema:

```
{
 "recipe_header": {
   "recipe_id": "recipe_[category]_[name]",
   "title": "Human-readable recipe name",
   "category": "api_client|error_handling|testing|deployment|architecture",
   "version": "semantic version (1.0.0)",
   "status": "VALIDATED|EXPERIMENTAL|DEPRECATED",
   "created_g": "global event counter",
   "last_validated_g": "most recent validation event"
 },
 "applicability": {
   "tech_stack": ["nodejs", "python", "react"],
   "project_types": ["service", "library", "ui_component"],
   "prerequisites": ["dependency1", "dependency2"],
   "constraints": ["memory < 512MB", "latency < 100ms"]
 },
 "implementation": {
   "description": "Clear problem statement and solution approach",
   "code_template": "Parameterized code template with {{variables}}",
   "configuration_example": "Complete working example",
   "testing_strategy": "How to validate the implementation"
 },
 "validation_evidence": {
   "successful_implementations": ["project_id_1", "project_id_2"],
   "test_results": "Evidence of successful testing",
   "performance_metrics": "Benchmarks or performance characteristics",
   "alternatives_considered": "Why this approach over alternatives"
 },
 "maintenance": {
   "known_issues": ["issue1", "issue2"],
   "evolution_path": "How this Recipe might evolve",
   "deprecation_strategy": "Migration path if Recipe becomes obsolete"
  }
}
```

Recipe Lifecycle Management

1. **Pattern Discovery**: Agents identify recurring implementation patterns during project work
2. **Recipe Proposal**: Create experimental Recipe in `/proposals/recipes/` with initial evidence
3. **Validation Process**: Recipe undergoes validation by designated Recipe Validator agent:
4. Code quality assessment
5. Evidence verification (minimum 2 successful implementations)
6. Framework compliance check (DRY, KISS, etc.)
7. Integration testing with existing Recipes
8. **Canonization**: Validated Recipes move to `docs/cookbook/recipes/` with VALIDATED status
9. **Usage Tracking**: Monitor Recipe application across projects for effectiveness metrics
10. **Evolution**: Regular review process for Recipe updates, deprecation, and replacement

Recipe Categories & Organization

Directory Structure:

```
docs/cookbook/
■■■ recipes/
■   ■■■ api_client/
■   ■■■ error_handling/
■   ■■■ testing/
■   ■■■ deployment/
■   ■■■ architecture/
■■■ recipe_index.md
■■■ validation_criteria.md
```

Integration with HAiOS Governance

ADR Compliance: All Recipes must demonstrate compliance with relevant ADRs (testing standards, security controls, etc.)

Framework Integration: Recipes must explicitly reference and comply with canonical frameworks from ADR-OS-032

Event Tracking: Recipe creation, validation, and usage tracked via global event counter g

Distributed Systems Implications: Recipe system MUST adhere to cross-cutting policies: - **Idempotency (ADR-OS-023):** Recipe application produces consistent results regardless of retry attempts - **Event Ordering (ADR-OS-027):** All Recipe lifecycle events properly timestamped and ordered - **Observability (ADR-OS-029):** Recipe usage and effectiveness captured in distributed traces - **Zero Trust (ADR-OS-025):** Recipe validation operates under zero-trust principles

Confidence: High

Rationale

1. **DRY Principle Enforcement**
2. Self-critique: Systematic capture and reuse of proven patterns eliminates redundant implementation work and ensures consistency.

   Confidence: High

   Quality Standardization

5. Self-critique: Validation process ensures only proven, tested patterns become canonical, raising overall implementation quality.

   Confidence: High

   Knowledge Preservation

8. Self-critique: Formal Recipe system prevents loss of institutional knowledge when projects end or team members change.

   Confidence: High

   Implementation Velocity

11. Self-critique: Reusable Recipes accelerate development by providing proven starting points for common implementation challenges.

   Confidence: Medium

Architectural Consistency

14. Self-critique: Standardized patterns ensure consistent architectural approaches across all HAiOS projects.
15. Confidence: High

Alternatives Considered

1. **Informal Pattern Sharing**: Continue current ad-hoc pattern sharing without formal structure.
2. Brief reason for rejection: Leads to inconsistency, quality variations, and knowledge loss. Violates DRY principles.

Confidence: High

External Pattern Libraries: Use existing pattern libraries (e.g., design patterns, architectural templates).

5. Brief reason for rejection: External libraries lack HAiOS-specific context and governance integration. May not align with HAiOS principles.

Confidence: Medium

Code Generation Tools: Implement automated code generation instead of reusable patterns.

8. Brief reason for rejection: Less flexible than Recipe system and doesn't capture contextual knowledge or decision rationale.

Confidence: Medium

Wiki-Based Documentation: Use informal wiki or documentation for pattern sharing.

11. Brief reason for rejection: Lacks validation processes, versioning, and integration with HAiOS governance model.
12. Confidence: High

Consequences

- **Positive:**
- Eliminates redundant implementation work across projects
- Ensures consistent quality and architectural approaches
- Preserves institutional knowledge in formal, searchable format
- Accelerates development velocity through proven starting points
- Enables systematic improvement of implementation patterns over time

Integrates with existing HAiOS governance and quality assurance processes

Negative:

- Adds overhead for Recipe creation and validation processes
- Risk of over-abstraction making Recipes too generic or too specific
- Requires agent training for effective Recipe discovery and application
- Recipe versioning and evolution complexity
- Potential Recipe conflicts requiring resolution processes

Clarifying Questions

- What specific validation criteria should be applied to ensure Recipe quality while not being overly restrictive to innovation?
- How should Recipe versioning be managed to balance stability for existing implementations with evolution for improved patterns?
- What metrics should be tracked to measure Recipe effectiveness and identify candidates for deprecation or improvement?
- How should Recipe conflicts be resolved when multiple Recipes could apply to the same implementation scenario?
- What is the governance process for Recipe evolution, including who has authority to update or deprecate existing Recipes?
- How should Recipe discovery be optimized to ensure agents can quickly find relevant patterns without being overwhelmed by choices?
- What is the migration strategy for existing informal patterns in the current `docs/cookbook/` directory?

This template integrates explicit assumption-surfacing, confidence indicators, self-critiques, and clarifying questions as per ADR-OS-021.

ANNOTATION_BLOCK_START

{ "artifact_annotation_header": { "artifact_id_of_host": "adr_os_034_md", "g_annotation_created": 253, "version_tag_of_host_at_annotation": "1.0.0" }, "payload": { "description": "Defines the Orchestration Layer and Session Management system for coordinating multi-agent workflows and maintaining conversational context across HAiOS operations.", "artifact_type": "DOCUMENTATION", "purpose_statement": "To establish a formal orchestration system that manages agent coordination, session state, and workflow execution while providing a unified interface for human operators.", "authors_and_contributors": [{ "g_contribution": 253, "identifier": "Hybrid_AI_OS" }, { "g_contribution": 250, "identifier": "Third_Party_Architectural_Review" }], "internal_dependencies": ["adr_os_template_md", "adr_os_032_md", "adr_os_021_md", "adr_os_030_md", "3rdpartyeval-10.md", "docs/source/roadmaps/phase1_to_2.md"], "linked_issue_ids": [] } }

ANNOTATION_BLOCK_END

ADR-OS-034: Orchestration Layer & Session Management

- **Status**: DEFERRED
- **Date**: 2025-01-28
- **Deciders**: [List of decision-makers]
- **Reviewed By**: [List of reviewers]

_ update: While a correct long-term vision, the "Human Gap Analysis" revealed that this is a "Phase 2+" concern. Focusing on it now would be a distraction from the immediate, necessary work of hardening the core engine and improving the solo operator's workflow. Action: Shelve this ADR for now. Revisit after the "Minimum Viable Foundry" milestone is complete. _

Context

HAiOS currently operates through individual agent interactions without a unified orchestration layer to coordinate multi-agent workflows, maintain conversational context, or provide session continuity. This creates several critical limitations:

1. **No Session Persistence**: Conversational context and work state are lost between interactions
2. **Agent Coordination Gaps**: Multiple agents working on related tasks lack coordination mechanisms
3. **Workflow Fragmentation**: Complex workflows spanning multiple phases lack unified management
4. **Human Interface Inconsistency**: No standardized interface for human operators to monitor and control HAiOS operations
5. **State Management Complexity**: No centralized system for managing distributed state across agents and sessions

The current architecture referenced in `docs/source/roadmaps/phase1_to_2.md` identifies the need for a "Cockpit" interface but lacks the underlying orchestration infrastructure to support comprehensive session management and agent coordination.

Additionally, the existing agent roles defined in ADR-OS-030 (Supervisor, Manager, Executor, etc.) need an orchestration layer to coordinate their interactions effectively and maintain operational coherence across complex, multi-phase workflows.

Assumptions

- [ ] Session state can be effectively serialized and persisted without losing critical context or creating performance bottlenecks.
- [ ] The orchestration layer can coordinate multiple agents without creating single points of failure or coordination bottlenecks.
- [ ] Workflow definitions can capture sufficient detail to enable automated orchestration while remaining maintainable and understandable.
- [ ] The Cockpit interface can provide meaningful visibility and control without overwhelming operators with excessive detail.
- [ ] Session management can handle concurrent operations and agent coordination without race conditions or state corruption.
- [ ] The orchestration system can integrate with existing HAiOS governance (event tracking, audit trails) without creating architectural conflicts.
- [ ] Workflow recovery and error handling can be implemented effectively within the orchestration framework.
- [ ] The system can scale to handle multiple concurrent sessions and complex multi-agent workflows without performance degradation.
- [ ] Agent communication protocols can be standardized sufficiently to enable reliable orchestration across different agent types.

This section was expanded to surface implicit assumptions about state management complexity, coordination mechanisms, scalability, and integration challenges.

Decision

Decision:

> We will implement an **Orchestration Layer & Session Management System** that provides unified coordination of multi-agent workflows, persistent session state management, and a standardized interface for human operators to monitor and control HAiOS operations.

Orchestration Layer Architecture

Core Components:

1. **Session Manager**: Maintains persistent session state, context, and conversation history
2. **Workflow Orchestrator**: Coordinates multi-agent workflows based on defined workflow specifications
3. **Agent Coordinator**: Manages agent lifecycle, communication, and resource allocation
4. **Cockpit Interface**: Provides human operators with visibility and control over HAiOS operations
5. **State Synchronizer**: Ensures consistency between distributed agent state and centralized orchestration state

Session Management System

Session Structure:

```
{
 "session_header": {
   "session_id": "session_[timestamp]_[uuid]",
   "created_g": "global event counter",
   "last_activity_g": "most recent activity",
   "status": "ACTIVE|SUSPENDED|COMPLETED|FAILED",
   "human_operator_id": "operator identification"
 },
 "session_context": {
   "conversation_history": "Complete interaction history",
   "active_workflows": ["workflow_id_1", "workflow_id_2"],
   "agent_assignments": {"agent_id": "current_task_assignment"},
   "shared_state": "Cross-agent shared data and context"
 },
 "session_metadata": {
   "project_context": "Associated project or initiative",
   "priority_level": "HIGH|MEDIUM|LOW",
   "resource_constraints": "Memory, time, cost limitations",
   "checkpoint_frequency": "Auto-save interval"
 }
}
```

Confidence: High

Consequences

- **Positive:**
- Enables coherent multi-agent workflows with proper coordination
- Provides persistent session state and work continuity
- Offers unified human interface for HAiOS operations monitoring and control

  Improves system reliability through centralized state management and recovery

  Negative:

- Adds architectural complexity with centralized orchestration layer
- Potential single point of failure if orchestration layer fails
- Performance overhead from centralized coordination and state management

---

This template integrates explicit assumption-surfacing, confidence indicators, self-critiques, and clarifying questions as per ADR-OS-021.

ANNOTATION_BLOCK_START

{ "artifact_annotation_header": { "artifact_id_of_host": "adr_os_035_md", "g_annotation_created": 251, "version_tag_of_host_at_annotation": "1.0.0" }, "payload": { "description": "Defines the Crystallization Protocol and Gatekeeper Agent for formal knowledge validation and canonization.", "artifact_type": "DOCUMENTATION", "purpose_statement": "To establish a formal two-space system that safely isolates exploratory work from canonical system state while providing auditable knowledge integration.", "authors_and_contributors": [{ "g_contribution": 251, "identifier": "Hybrid_AI_OS" }, { "g_contribution": 250, "identifier": "Third_Party_Architectural_Review" }], "internal_dependencies": ["adr_os_template_md", "adr_os_032_md", "adr_os_021_md", "3rdpartyeval-10.md"], "linked_issue_ids": [] } }

ANNOTATION_BLOCK_END

ADR-OS-035: The Crystallization Protocol & Gatekeeper Agent

- **Status**: SUPERSEDED
- **Date**: 2025-01-28
- **Deciders**: [List of decision-makers]
- **Reviewed By**: [List of reviewers]

_ UPDATE: This was a good idea, but it's been replaced by a more powerful and integrated concept. The function of the "Canonizer-Agent" is now better understood as a part of the CI/CD pipeline's lint stage, not a standalone agent. _

Context

The operator's creative, exploratory process (regenerating responses, refining inputs, iterative development) introduces beneficial chaos that drives innovation and discovery. However, this creative process risks polluting the canonical, stable state of the system with transient, unvalidated artifacts. The current system lacks a formal mechanism to distinguish between exploratory work and validated knowledge, creating potential for system corruption when experimental artifacts are inadvertently treated as authoritative.

Additionally, the system currently has no defined protocol for how human creativity and AI iteration are integrated into the formal architecture. This creates a critical gap between the creative process that generates new knowledge and the rigorous governance that maintains system integrity.

Assumptions

- [ ] The file system supports creating and managing a parallel directory structure for exploratory work.
- [ ] The Gatekeeper Agent can be implemented with sufficient validation sophistication to prevent corrupted artifacts from entering canonical state.
- [ ] The validation process can be made deterministic and auditable while still allowing for creative exploration.
- [ ] Operators will adopt the discipline of working in the exploration space rather than directly modifying canonical artifacts.
- [ ] The two-space architecture can be maintained without creating excessive operational overhead.
- [ ] The validation criteria can be made comprehensive enough to catch all categories of potential corruption.
- [ ] The crystallization process can preserve the intent and context of exploratory work during canonization.
- [ ] All compliance requirements from referenced ADRs (e.g., ADR-OS-032, ADR-OS-021) are up-to-date and enforced.

This section was expanded to surface implicit assumptions about validation sophistication, operator discipline, and process overhead.

Frameworks/Models Applied

This ADR applies the following canonical models and frameworks (per ADR-OS-032):

Evidence-Based Development v1.0

- **Compliance Proof:** The Gatekeeper Agent performs formal validation checks with evidence requirements before allowing canonization, ensuring all canonical knowledge is evidence-based.
- **Self-Critique:** Validation criteria must be comprehensive; incomplete validation could allow corrupted artifacts to pass through.

Separation of Concerns v1.0

- **Compliance Proof:** Clear separation between exploration space (creative chaos) and canonical state (validated truth) with distinct purposes and access patterns.
- **Self-Critique:** Maintaining separation requires operator discipline; accidental cross-contamination could compromise system integrity.

Single Source of Truth v1.0

- **Compliance Proof:** Canonical state remains the single source of truth, with exploration space explicitly marked as non-authoritative.
- **Self-Critique:** Risk of confusion about which space contains the authoritative version of evolving artifacts.

Quality Assurance v1.0

- **Compliance Proof:** Formal gatekeeping process ensures only validated, coherent artifacts become part of canonical system state.
- **Self-Critique:** Validation process could become a bottleneck if not properly streamlined and automated.

Audit Trail v1.0

- **Compliance Proof:** Complete audit trail of crystallization decisions, validation results, and artifact lineage from exploration to canonization.
- **Self-Critique:** Detailed audit trails may become voluminous and difficult to navigate without proper tooling.

Assumption Surfacing v1.0

- **Compliance Proof:** Explicit assumptions about validation sophistication, operator discipline, process overhead, and validation comprehensiveness.
- **Self-Critique:** Eight assumptions listed; crystallization protocol likely has additional implicit assumptions about tool integration and workflow adoption.

Distributed Systems Principles v1.0

- **Compliance Proof:** Protocol addresses consistency (canonical vs. exploration state), availability (non-blocking exploration), and partition tolerance (independent workspaces).
- **Self-Critique:** Distributed implications of multi-operator exploration spaces not fully addressed in current design.

Fail-Safe Design v1.0

- **Compliance Proof:** System defaults to rejecting artifacts that fail validation rather than allowing potentially corrupted knowledge into canonical state.
- **Self-Critique:** Overly strict validation could reject valid innovations; balance between safety and innovation acceptance is critical.

Decision

Decision:

> We will implement the **Crystallization Protocol**, a formal two-space system enforced by a new agent persona, the **Gatekeeper Agent**. This protocol establishes a clear separation between exploratory and canonical work while providing an auditable mechanism for knowledge integration.

Two-Space Architecture

Exploration Space: - Location: `/proposals` directory (or configurable path in `haios.config.json`) - Purpose: Dedicated, non-canonical directory where all interactive, exploratory artifacts are stored - Status: These artifacts have no official status and are explicitly marked as experimental - Permissions: Full read/write access for exploration and iteration

Canonized State: - Location: Official `docs/` and `os_root/` directories - Purpose: The single source of truth representing validated, official system knowledge - Status: All artifacts have passed formal validation and represent authoritative system state - Permissions: Write access only through Crystallization Protocol

Crystallization Protocol Workflow

1. **Exploration Phase**: Operator works freely in the exploration space, generating and refining artifacts without constraints
2. **Crystallization Request**: Operator signals desire to formalize a proposal from the exploration space
3. **Gatekeeper Invocation**: The Gatekeeper Agent is invoked to perform validation
4. **Validation Execution**: Agent performs comprehensive validation checks:
5. Schema conformance against established schemas
6. Consistency checks against existing canonized ADRs and artifacts
7. Link integrity verification
8. Framework compliance validation (per ADR-OS-032)
9. Assumption surfacing completeness (per ADR-OS-021)
10. **Validation Decision**:
11. **Success**: Artifact is committed to canonical state with full audit trail
12. **Failure**: Process halts with detailed feedback; artifact remains in exploration space for refinement

Distributed Systems Implications

The Crystallization Protocol MUST adhere to the following cross-cutting policies:

- **Idempotency (ADR-OS-023):** All crystallization operations MUST be idempotent. Retrying a crystallization request with the same exploration artifact produces the same validation result.
- **Event Ordering (ADR-OS-027):** All validation steps and crystallization events MUST be properly timestamped and ordered using the global event counter `g`.
- **Observability (ADR-OS-029):** Every crystallization attempt MUST be captured in a distributed trace with full validation details and decision rationale.
- **Zero Trust (ADR-OS-025):** The Gatekeeper Agent operates under zero-trust principles, validating all artifacts regardless of source or previous validation history.

Confidence: High

Rationale

1. **System Integrity Protection**
2. Self-critique: The two-space architecture protects canonical state from experimental corruption while preserving creative freedom.

   Confidence: High

   Formalized Human-AI Interface

5. Self-critique: Creates a precise, auditable workflow for integrating human creativity with formal system governance.

   Confidence: High

   Quality Ratchet Mechanism

8. Self-critique: Ensures architectural quality and consistency can only increase over time through formal validation.

   Confidence: High

   Innovation Enablement

11. Self-critique: Provides safe space for experimentation without compromising system stability.

   Confidence: Medium

   Audit Trail Completeness

14. Self-critique: Creates complete lineage from creative exploration to canonical knowledge with validation evidence.
15. Confidence: High

Alternatives Considered

1. **Direct Canonical Editing**: Continue allowing direct modification of canonical artifacts.
2. Brief reason for rejection: Risks system corruption from experimental changes and lacks validation discipline.

   Confidence: High

   Version Control Branching: Use git branches for exploration with merge reviews.

5. Brief reason for rejection: Git workflows don't provide the formal validation semantics and artifact-level governance required by HAiOS.

   Confidence: High

   Manual Review Process: Human-only review for canonization decisions.

8. Brief reason for rejection: Doesn't scale and lacks the systematic validation checks needed for complex architectural artifacts.
9. Confidence: Medium

Consequences

- **Positive:**
- Protects system integrity while enabling creative exploration
- Formalizes knowledge integration process with complete audit trails
- Creates a "quality ratchet" ensuring canonical knowledge quality only increases
- Enables safe experimentation and iteration

  Provides clear workflow for human-AI collaborative knowledge creation

  Negative:

- Adds process overhead for canonizing new knowledge
- Requires operator discipline to work in exploration space
- Gatekeeper Agent implementation complexity
- Potential bottleneck if validation process is too slow or rigid

Clarifying Questions

- What specific validation checks should the Gatekeeper Agent perform, and how comprehensive should schema and consistency validation be?

- How should the system handle partial crystallization (e.g., only some artifacts from an exploration session are ready for canonization)?
- What is the fallback procedure if the Gatekeeper Agent fails or produces inconsistent validation results?
- How should the exploration space be organized and managed to prevent it from becoming cluttered with obsolete experiments?
- What metrics and observability should be implemented to monitor the health and effectiveness of the crystallization process?
- How should the protocol handle multi-operator scenarios where multiple people are exploring related concepts simultaneously?

This template integrates explicit assumption-surfacing, confidence indicators, self-critiques, and clarifying questions as per ADR-OS-021.

ADR-OS-037: Adaptive Task Execution Protocol

- **Status**: Proposed
- **Date**: 2025-06-26
- **Deciders**: Architecture Team
- **Reviewed By**: [TBD]

Context

Foundation models possess different intrinsic execution capabilities. A "holistic" model may succeed at a complex, single-shot task, while a "literal" model requires explicit, sequential sub-tasking to avoid failure. A rigid Task Executor that attempts all tasks in the same way is inefficient and brittle, failing to leverage the unique strengths of each available model. This ADR formally separates the definition of a task from the strategy used to execute it.

Assumptions

- [ ] Foundation models have predictable and classifiable execution paradigms (e.g., "holistic" vs. "literal").
- [ ] The `aiconfig.json` file is the reliable and definitive source for model capability profiles.
- [ ] The cost of the internal BLUEPRINT call for task decomposition is less than the cost of a failed or low-quality single-shot execution.
- [ ] The system can reliably determine the active persona model and access its profile at runtime.
- [ ] The defined execution strategies (Direct and Iterative) cover the majority of required execution paradigms.

Models/Frameworks Applied

Separation of Duties (Registry v1.0):
- *Proof:* The decision explicitly separates the `what` (the task in the Execution Plan) from the `how` (the execution strategy chosen by the Task Executor). The BLUEPRINT agent defines the former, while the Task Executor dynamically determines the latter.
- *Self-critique:* This introduces a new dependency where the Task Executor must have out-of-band knowledge of model capabilities, which could become a point of failure if not maintained.
- *Exceptions:* None.

Strategy Pattern (Design Pattern):
- *Proof:* The Task Executor is refactored to act as a dispatcher that selects a specific execution strategy (`DirectExecutionStrategy`, `IterativeDecompositionStrategy`) based on runtime conditions (the model's `execution_paradigm`).
- *Self-critique:* Over-engineering is a risk if only two strategies exist. However, it provides a clear extension point for future, more nuanced execution methods.
- *Exceptions:* None.

Decision

Mandate an Adaptive Task Execution Protocol. The Task Executor agent will be a strategy-based dispatcher that dynamically selects the appropriate execution method based on the active foundation model's profile.

Core Components:

Schema Enhancement (`aiconfig.json`): The `model_profiles` object must contain a `capabilities` object for each model. This object must include the key:
- `execution_paradigm`: (Enum: `single_shot_holistic`, `iterative_decomposition_required`)

Core Logic Refactor (Task Executor): The Task Executor will implement the Strategy Pattern.

Flow:
1. Receive Task object.
2. Read the `active_persona_model` from `aiconfig.json`.
3. Look up the model's `execution_paradigm` in its profile.
4. Dispatch to the corresponding execution strategy module.

Execution Strategies Defined:
- **DirectExecutionStrategy:** Triggered by `single_shot_holistic`. Executes the task as a single, atomic action.

IterativeDecompositionStrategy: Triggered by `iterative_decomposition_required`. Initiates a nested loop:
- **A (Nested Blueprint):** Make an internal call to the BLUEPRINT agent to decompose the high-level task into sub-tasks.
- **B (Sub-Task Loop):** Execute the resulting sub-tasks sequentially.
- **C (Finalization):** Mark the parent task as complete only after the sub-task sequence is finished.

Confidence: High

Rationale

Maximizes Agent Effectiveness:

- Allows the system to leverage the unique strengths of each model by choosing the optimal execution mode.
- *Self-critique:* The manual classification of models is subjective and may not capture the full nuance of a model's capabilities.
- *Confidence:* High

Increases Runtime Robustness:

- Proactively avoids errors by selecting a safer, more granular execution path for models known to struggle with complex, single-shot tasks.
- *Self-critique:* The nested decomposition loop introduces its own failure modes (e.g., failure in the sub-task planning step).
- *Confidence:* Medium

Decouples Planning from Execution:

- Keeps the primary Execution Plan clean and high-level. The implementation details of task decomposition are abstracted away from the main plan.
- *Self-critique:* This abstraction can make debugging more difficult, as the true execution path is not visible in the primary plan artifact.
- *Confidence:* High

Alternatives Considered

Monolithic Executor: Continue with a single, rigid execution method.

- *Reason for rejection:* Inefficient and brittle, as it fails to adapt to different model capabilities, leading to higher failure rates.

Bake Decomposition into All Plans: Require the BLUEPRINT agent to always decompose complex tasks, regardless of the target model.

- *Reason for rejection:* Unnecessarily inflates plan complexity and execution time for powerful, holistic models that do not require it.

Consequences

Positive:

- Improved performance and reliability by matching tasks to the most suitable execution strategy.
- Cleaner, more abstract Execution Plans.
- Creates actionable data (`strategy_used`) for future automated performance tuning.

Negative:

- Increases the complexity of the Task Executor.
- Introduces a new maintenance burden: keeping model capability profiles in `aiconfig.json` accurate and up-to-date.

Clarifying Questions

- How will the system handle a model whose `execution_paradigm` is not defined in its profile? Should it default to the safest strategy (iterative)?
- What is the mechanism for updating model capability profiles as new models are added or existing ones are fine-tuned?
- Should the `IterativeDecompositionStrategy` have a depth limit to prevent infinite recursion?
- How are sub-task failures handled within the nested loop? Does a single sub-task failure cause the entire parent task to fail?

This ADR is now compliant with the standards set in ADR-OS-021 and ADR-OS-032.

ADR-OS-038: The Plan Validation & Governance Gateway

- **Status**: Proposed
- **Date**: 2025-06-26
- **Deciders**: Architecture Team
- **Reviewed By**: [TBD]

Context

An Execution Plan can be syntactically correct and schema-compliant but still be strategically flawed, economically inefficient, or behaviorally mismatched for its assigned agent. Approving such a plan leads to wasted resources, runtime failures, and low-quality outcomes. This ADR introduces a formal, automated "peer review" for the BLUEPRINT agent's work to prevent such failures.

Assumptions

- [ ] The logic, cost, and behavioral characteristics of a plan can be reliably linted and validated before execution.
- [ ] The governing artifacts (`planning_guidelines.md`, `aiconfig.json`) are accurate and available to the linter.
- [ ] The benefits of pre-execution validation outweigh the computational cost of running the Plan Linter.
- [ ] The Plan Linter is capable of constructing an accurate causal graph of tasks to detect logical errors.
- [ ] A sufficiently accurate cost model exists to make economic linting meaningful.

Models/Frameworks Applied

Evidence-Based Development (Registry v1.0):
- *Proof:* The decision mandates an automated, evidence-based gateway that requires every Execution Plan to pass explicit checks (semantic, economic, behavioral) before it can be approved. The plan itself becomes the evidence to be validated.
- *Self-critique:* The linter's checks are only as good as their governing artifacts. An outdated cost model or incomplete planning guideline could provide a false sense of security.
- *Exceptions:* None.

Separation of Duties (Registry v1.0):
- *Proof:* It establishes a formal separation between the BLUEPRINT agent (the creator of the plan) and the Plan Linter (the automated reviewer of the plan). This enforces an impartial, automated quality gate.
- *Self-critique:* This adds a potential bottleneck. If the linter is slow or overly strict, it could impede development velocity.
- *Exceptions:* None.

Decision

Mandate a new, non-negotiable **Plan Validation Gateway** stage in the HAiOS lifecycle. This gateway consists of a **Plan Linter** that executes a series of automated checks after the BLUEPRINT phase and before the CONSTRUCT phase.

Core Component: The Plan Linter

This component will execute three categories of checks:

Semantic & Causal Linting:
- **Function:** Validates the logical integrity of the plan.
- **Checks:** Constructs a causal graph to detect orphan resources, temporal logic errors, and TDD sequence violations.
- **Governing Artifact:** `planning_guidelines.md`

Economic Linting:
- **Function:** Validates the resource efficiency of the plan.
- **Checks:** Estimates the plan's cost (tokens, time) against the initiative's budget and recommends more efficient Cookbook recipes.
- **Governing Artifact:** `cost_model` section in `aiconfig.json`

Behavioral Linting:
- **Function:** Validates the plan's suitability for the assigned agent.
- **Checks:** Compares task complexity against the target model's capabilities (from `model_profiles` in `aiconfig.json`) and raises a `MODEL_MISMATCH_WARNING` if misaligned.
- **Governing Artifact:** `model_profiles` section in `aiconfig.json`

Confidence: High

Rationale

Shifts Quality Control "Left":

- Moves validation from the expensive, reactive VALIDATE phase to a cheap, proactive linting phase before work begins, saving time and resources.
- *Self-critique:* The linter cannot catch all possible strategic flaws, only those that can be programmatically defined.
- *Confidence:* High

Formalizes "Architectural Review":

- Automates the expert checks a senior architect would perform for logic, efficiency, and feasibility, ensuring consistency and rigor.
- *Self-critique:* Automation may lack the nuanced judgment of a human reviewer, potentially allowing through plans that are technically correct but strategically unwise.
- *Confidence:* Medium

Harnesses Stochasticity Proactively:

- Acts as a crucial filter that shapes the creative but potentially chaotic output of the BLUEPRINT agent into a plan that is guaranteed to be logical, economical, and executable.
- *Self-critique:* An overly aggressive linter could stifle innovative or unconventional plans that might otherwise be successful.
- *Confidence:* Medium

Alternatives Considered

Manual Peer Review Only: Rely solely on human engineers to review every plan.

- *Reason for rejection:* Slow, expensive, and prone to human error and inconsistency. Not scalable for a highly autonomous system.

Post-Execution Validation Only: Catch errors only in the VALIDATE phase.

- *Reason for rejection:* Inefficient and wasteful, as it allows flawed plans to consume significant resources before being caught.

Consequences

Positive:

- Prevents a significant class of errors before they consume resources.
- Enforces architectural and economic best practices automatically.
- Improves the overall quality and reliability of Execution Plans.

Negative:

- Introduces a new component (the Plan Linter) that must be developed and maintained.
- Adds a step to the workflow, which could slightly increase the time from planning to execution.

Clarifying Questions

- How will the Plan Linter be implemented and integrated into the CI/CD pipeline?
- What happens when a plan fails validation? Is there an automated feedback loop to the BLUEPRINT agent for correction?
- How are the linting rules in the governing artifacts versioned and updated?
- Should there be a mechanism to manually override the linter for exceptional cases, and if so, how is that governed?

This ADR is now compliant with the standards set in ADR-OS-021 and ADR-OS-032.

ADR-OS-039: The Argus Protocol (Continuous Runtime Auditing)

- **Status**: Proposed
- **Date**: 2025-06-26
- **Deciders**: Architecture Team
- **Reviewed By**: [TBD]

Context

Governance and validation in the current architecture are primarily "pre-flight" checks (Plan Validation Gateway) or "post-mortem" reviews (VALIDATE phase). This creates a gap during the CONSTRUCT phase where emergent, unexpected behavior (due to model drift, dependency changes, or runtime anomalies) can go undetected until after significant resources have been wasted or a flawed artifact has been produced. The system lacks a real-time, active auditor.

Assumptions

- [ ] A stream of system events can be reliably ingested and processed in near real-time.
- [ ] A statistical model of "normal" operational behavior can be accurately built and maintained.
- [ ] Deviations from the normal operational baseline are reliable indicators of meaningful anomalies.
- [ ] The Auditor-Agent has the necessary permissions to access the event bus and the kill-switch flags.
- [ ] Granting a system agent autonomous authority to halt the entire OS is an acceptable and necessary safety feature.

Models/Frameworks Applied

Evidence-Based Development (Registry v1.0):
- *Proof:* The protocol makes the runtime process itself the evidence to be validated. The Auditor-Agent continuously compares real-time event streams (the evidence) against a model of expected behavior.
- *Self-critique:* The anomaly detection model is a potential single point of failure. If it is poorly trained or becomes outdated, it could lead to false positives (unnecessary shutdowns) or false negatives (missed anomalies).
- *Exceptions:* None.

Separation of Duties (Registry v1.0):
- *Proof:* It establishes a new, distinct duty—active, real-time process auditing—and assigns it to a dedicated agent persona (the Auditor-Agent), which operates independently of the core execution agents.
- *Self-critique:* The Auditor-Agent has significant power (kill-switch authority). A bug or compromise in this single agent could have system-wide consequences.
- *Exceptions:* None.

Decision

Mandate the **Argus Protocol**, a system for continuous, real-time runtime auditing, executed by a dedicated **Auditor-Agent** persona.

Core Components:

Agent Persona Update (ADR-OS-030):
- The **Auditor-Agent** is an active, persistent process monitor that runs concurrently with all other operations.

Core Logic (Auditor-Agent):
- **A) Real-Time Event Ingestion:** The agent subscribes to the primary system event bus to consume all significant events (e.g., `task_started`, `file_written`, `token_cost_logged`).

  B) Anomaly Detection Engine: The agent maintains a real-time statistical model of the system's normal operational behavior to detect deviations in:
  - **Economic Anomalies:** Spikes in cost or duration.
  - **Behavioral Drift:** Changes in model success/failure rates or strategy needs.
  - **Causal Integrity Faults:** Events occurring out of logical sequence.

New Governance Mechanism (Kill-Switch Authority):
- The Auditor-Agent is granted write-access to the kill-switch flags defined in ADR-OS-018.
- **On Critical Anomaly:** The agent writes to `control/hard_kill.flag` to halt the OS and logs a P0 issue.
- **On Warning Anomaly:** The agent writes to `control/write_lockdown.flag` to pause new actions and logs a P1 issue for review.

Confidence: Medium

Rationale

Completes the Governance Loop:
- The system gains governance at all three stages: Pre-flight (ADR-OS-038), In-flight (this ADR), and Post-flight (VALIDATE phase), providing comprehensive oversight.

- *Self-critique:* This adds significant complexity to the overall system architecture.
- *Confidence:* High

Proactive Failure Prevention:

- Moves beyond preventing *planned* errors to detecting and halting *emergent*, unplanned errors in real time.
- *Self-critique:* The effectiveness is entirely dependent on the quality of the anomaly detection model.
- *Confidence:* Medium

Enables True Self-Awareness:

- The data collected by the Auditor-Agent is the necessary input for future systems that can perform automated performance tuning and self-correction.
- *Self-critique:* This is a long-term vision; the immediate implementation will be a simpler version of this goal.
- *Confidence:* Medium

Alternatives Considered

Log Analysis Only: Rely on post-mortem analysis of logs to find anomalies.

- *Reason for rejection:* This is a reactive approach that cannot prevent damage from in-progress failures.

Embedded Checks in Core Agents: Have each agent be responsible for its own runtime monitoring.

- *Reason for rejection:* Violates Separation of Duties and leads to duplicated, inconsistent monitoring logic across the system.

Consequences

Positive:

- Provides a critical safety net against emergent, real-time failures.
- Completes the system's governance and validation framework.
- Lays the foundation for future self-tuning capabilities.

Negative:

- Introduces a highly complex new component (the Auditor-Agent and its anomaly detection engine).
- Grants a single agent the authority to halt the entire system, which is a significant risk.
- The system will incur a constant, low-level resource cost from the auditing process.

Clarifying Questions

- What specific event bus technology will be used?
- What initial algorithms will be used for the anomaly detection engine, and how will it be trained?
- What are the precise criteria for a "Critical Anomaly" vs. a "Warning Anomaly"?
- Is there a manual override to prevent or reverse a kill-switch action taken by the Auditor-Agent?

This ADR is now compliant with the standards set in ADR-OS-021 and ADR-OS-032.

Architecture Decision Records (ADR) Index

This directory contains all Architecture Decision Records for the Hybrid AI Operating System (HAiOS). ADRs document important architectural decisions, their context, rationale, and consequences.

Quick Navigation

■■ Core Concepts & Data Models (Foundation)

- ADR-OS-001 - **Core Operational Loop & Phasing** - The five-phase state machine: ANALYZE → BLUEPRINT → CONSTRUCT → VALIDATE → IDLE.
- ADR-OS-002 - **Hierarchical Planning Model** - Multi-tiered planning from `Request` → `Analysis` → `Initiative Plan` → `Execution Plan`.
- ADR-OS-003 - **Artifact Annotation Strategy** - Mandates the `EmbeddedAnnotationBlock` for self-describing artifacts.
- ADR-OS-004 - **Global Event Tracking & Versioning** - g counter for total event ordering and v versioning for optimistic locking.
- ADR-OS-005 - **Directory Structure & File Naming** - Configuration-driven project layout via `haios.config.json`.
- ADR-OS-009 - **Issue Management & Summarization** - Structured, tiered issue tracking from individual files to global summaries.

■■ Operational Patterns & Execution

- ADR-OS-006 - **Scaffolding Process** - Automated artifact creation using `Scaffold Definition` files and templates.
- ADR-OS-010 - **Constraint Management & Locking Strategy** - Using `_locked` fields to enforce architectural integrity and prevent agent drift.
- ADR-OS-011 - **Task Failure Handling & Remediation** - "Log, Isolate, and Remediate" strategy for robust error handling.
- ADR-OS-013 - **Pre-Execution Readiness Checks** - Mandates verification of task prerequisites to prevent guaranteed failures.
- ADR-OS-015 - **Precision Context Loading** - Efficient, targeted context loading for LLM agents using line/pattern slicing.
- ADR-OS-016 - **Live Execution Status Tracking** - Separating immutable plans from mutable status files (`exec_status_*.txt`).
- ADR-OS-018 - **Execution Status Persistence & Recovery** - *(Note: Content is about foundational security controls, not persistence. Needs review.)*
- ADR-OS-022 - **Mechanical Inventory Buffer** - Prevents redundant work by staging reusable resources in a crash-safe buffer.

■ Agent & Tool Management

- ADR-OS-012 - **Dynamic Agent Management** - Runtime agent registration and configuration via an agent registry and "Agent Cards".
- ADR-OS-030 - **Archetypal Agent Roles & Protocols** - Defines a fixed set of agent roles (Supervisor, Manager, etc.) with strict permissions and escalation paths.
- ADR-OS-033 - **Cookbook & Recipe Management System** - **■■ PROPOSED** - Formal system for capturing, validating, and reusing implementation patterns across HAiOS projects.
- ADR-OS-034 - **Orchestration Layer & Session Management** - **■■ PROPOSED** - Unified coordination of multi-agent workflows with persistent session state and Cockpit interface.
- ADR-OS-035 - **The Crystallization Protocol & Gatekeeper Agent** - **■■ PROPOSED** - Formal two-space system for validating exploratory work before canonization.

■ Quality, Governance & Meta-Architecture

- ADR-OS-007 - **Integrated Testing Lifecycle** - Evidence-based testing with a strict separation of duties between agents.
- ADR-OS-008 - **OS-Generated Reporting Strategy** - Mandates `Analysis`, `Validation`, and `Progress` reports for human oversight.
- ADR-OS-014 - **Project Guidelines Artifact** - A durable, version-controlled home for project standards, conventions, and checklists.
- ADR-OS-021 - **Explicit Assumption Surfacing** - Mandates that all ADRs surface assumptions, confidence levels, and self-critiques.
- ADR-OS-031 - **Pre-Initiative Source Artifact Standards** - Defines the required set of upstream documents (PRD, TRD, etc.) for any new initiative.
- ADR-OS-032 - **Canonical Models and Frameworks Registry & Enforcement** - A registry of best practices (KISS, DRY, ToC) that artifacts must explicitly reference and prove compliance with.

■ Distributed Systems & Cross-Cutting Policies

- ADR-OS-019 - **Observability & Budget Governance** - Defines Prometheus metrics, cost tracking, and budget enforcement.
- ADR-OS-020 - **Runtime Modes & Developer Experience** - Defines `STRICT` and `DEV_FAST` modes to balance safety and velocity.
- ADR-OS-023 - **Universal Idempotency & Retry Policy** - A cross-cutting policy for safe retries with exponential backoff and circuit breakers.
- ADR-OS-024 - **Asynchronous and Eventual Consistency Patterns** - Standardizes on event-driven communication, sagas, and eventual consistency.
- ADR-OS-025 - **Zero-Trust Internal Security Baseline** - Mandates mTLS and token-based auth for all internal service communication.
- ADR-OS-026 - **Dynamic Topology, Health Checking, and Failure Propagation** - Defines service discovery, heartbeats, and status propagation.
- ADR-OS-027 - **Global and Vector Clock Event Ordering** - Specifies the use of logical clocks to preserve causality in distributed workflows.
- ADR-OS-028 - **Partition Tolerance and Split-Brain Protocol** - Defines explicit CAP trade-offs and reconciliation strategies for network partitions.
- ADR-OS-029 - **Universal Observability and Trace Propagation** - Mandates end-to-end distributed tracing via universal `trace_id` propagation.

■ Implementation & Phasing

- ADR-OS-017 - **Phase 1 - MVP Engine & Tooling** - Defines the scope and deliverables for the initial implementation phase.

Contributing to ADRs

When proposing a new ADR: 1. Use the next available ADR-OS-XXX number. 2. Follow the established template format defined in `adr_os_template.md` and mandated by `ADR-OS-021`. 3. Update this index with the new ADR. 4. Ensure the ADR includes a valid `EmbeddedAnnotationBlock`. 5. Ensure the ADR complies with the governance standards in `ADR-OS-031` and `ADR-OS-032`.