Snapdragon

ENABLING AGENTIC INTELLIGENCE ACROSS AUTONOMOUS MACHINES

Girish Shirasat
Senior Director, Engineering
Qualcomm Technologies

DISCLAIMER

Qualcomm Technologies, Inc.

Snapdragon and Qualcomm branded products are products of Qualcomm Technologies, Inc. and/or its subsidiaries. Qualcomm patents are licensed by Qualcomm Incorporated.

Note: Certain services and materials may require you to accept additional terms and conditions before accessing or using those items.

References to "Qualcomm" may mean Qualcomm Incorporated, or subsidiaries or business units within the Qualcomm corporate structure, as applicable.

Qualcomm Incorporated includes our licensing business, QTL, and the vast majority of our patent portfolio. Qualcomm Technologies, Inc., a subsidiary of Qualcomm Incorporated, operates, along with its subsidiaries, substantially all of our engineering, research and development functions, and substantially all of our products and services businesses, including our QCT semiconductor business.

Materials that are as of a specific date, including but not limited to press releases, presentations, blog posts and webcasts, may have been superseded by subsequent events or disclosures.

Nothing in these materials is an offer to sell or license any of the services or materials referenced herein.

Qualcomm Technologies, Inc.
5775 Morehouse Drive
San Diego, CA 92121
U.S.A.

TABLE OF CONTENTS

INTRODUCTION

Application developers creating multi-process distributed applications don’t need to write code for services such as scheduling threads, managing virtual memory, and exposing device drivers. A typical operating system provides those services.

So, why are developers of multi-agent systems still spending time on fundamentals like goal admission, tool authorization, memory provenance, and safety release gates? Shouldn’t those be provided as reusable platform services? And, in the case of embodied systems which are physical intelligent system whose behavior emerges from continuous interaction with its environment through sensing and action.) shouldn't determinism, safety, and security be part of those platform services?

The problem is that agentic intelligence cannot be safely introduced into machines as a loose collection of application prompts, cloud services, or model endpoints. It requires an operating environment that explicitly governs both probabilistic reasoning and deterministic sensing, control, and actuation.

This paper proposes an Embodied Agentic Operating System (EAOS): an architecture that elevates the essentials of agent development into first-class platform services. The paper defines a disciplined way to bring agentic intelligence and embodied control together. EAOS is designed for embodied AI systems, including autonomous vehicles, mobile robots, manipulators, humanoids, drones, and other intelligent, cyber-physical machines. Its benefits accrue not only to developers, but also to the vehicle OEMs, robotics manufacturers, industrial automation providers, Tier-1 suppliers, and platform vendors whose autonomous machines they work on.

Main takeaways:

- Developers who create AI agents for autonomous machines like vehicles, robots and drones must tread the disparate paths of probabilistic reasoning logic (for applications) and deterministic control (for sensors, actuators, and controllers).
- The frameworks they use for creating agents emphasize orchestration and tool use, but the physical systems they are building demand deterministic fallback and safety-governed execution. That architecture gap dictates that they spend a disproportionate amount of their time and effort developing infrastructure instead of creating agents.
- EAOS proposes the semantic control plane, kernel-like agent services, policy enforcement, observability, and edge-cloud governance needed for agentic functions. By allowing agentic capabilities to become reusable platform services and governed infrastructure rather than isolated feature logic, EAOS frees up development teams to create agents instead of infrastructure.

WHY EMBODIED AGENTIC OS (EAOS) IS NEEDED

Developers face an architecture gap.

Modern agents perform work as diverse as decomposing tasks, using tools, and recovering from failures, which makes the runtime system as important as the model itself [1][2][4]. The frameworks for building agents emphasize orchestration and tool use.

Conversely, systems embodied in machines like autonomous vehicles, robots, and drones impose real-time response, mixed-criticality separation, deterministic fallback, and safety-governed execution.

EAOS bridges that gap through an architecture that spans multiple layers:

- Embodied edge
- Kernel and control plane
- Runtime layer
- Integration layer
- Cloud orchestration

All layers are overlaid by safety and determinism controls [2][4][9][10]. Thus, the purpose of EAOS is to define a disciplined way to bring agentic intelligence and embodied control together.

Embodied Agentic intelligence requires both probabilistic reasoning and deterministic authority. But separated.

A system may use reasoning to recommend and prepare bounded supervisory actions, but where safety is at issue, there is no substitute for deterministic controllers and certified software. Keeping the two realms separate allows the improvements of agentic software without letting a language model manage calibrated controllers and safety mechanisms.

That is why EAOS adds a higher semantic layer. It schedules goals, decomposes tasks into bounded reasoning episodes, and mediates access to tools and physical-system services. It converts probabilistic model outputs into auditable proposals that, where safety is a priority, must pass deterministic validation before they can influence physical behavior.

Shifting to an embodied system places additional burden on development teams.

It means shifting to a distributed, cyber-physical compute platform with moving parts such as software updates, sensor and data pipelines, cloud services, and AI models. That shift creates the development problem mentioned above: without an explicit EAOS layer, each development team must build its own fundamentals like prompt logic, tool adapters, and access-control rules. The likely result will be misspent resources, duplicated infrastructure and unpredictable interactions.

EAOS provides those fundamentals as reusable platform services, freeing up development teams to create agents instead of infrastructure.

THE GENERIC AGENTIC OS— ORIGIN AND RESEARCH LINEAGE OF EAOS

The research lineage behind EAOS begins by considering large language models as operating-system-like cognitive kernels. Context resembles working memory, external stores resemble persistent storage, tools resemble system calls or devices, and agents resemble applications.

- LLM-as-OS and AIOS separate agent applications from kernel-managed services such as scheduling, context management, and memory management [1][2].
- MemGPT posits virtual context management, where limited model context is treated like a scarce memory tier and long-term storage is paged in when relevant [5].
- KAOS and Magentic-One add concepts like multi-agent coordination, manager/specialist agents, and reusable task-solving workflows [3][4].

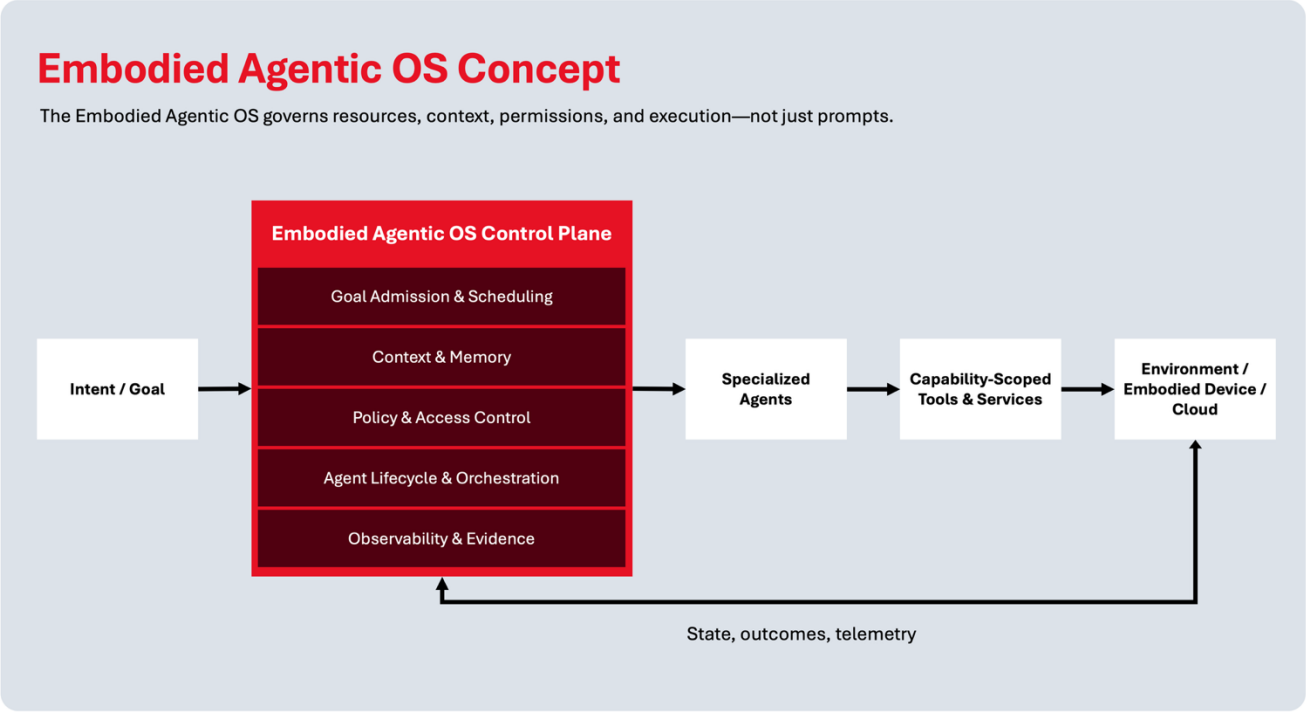

Figure 1. Generic agentic OS concept and managed runtime services

Embodied AI research extends those concepts into physical systems based on sensors and actuators, spanning autonomous vehicles, robots, and drones. EAOS reinterprets agentic OS primitives through a control plane of goal admission, context, policy, lifecycle, mixed criticality, auditability, and observability. For example:

- Scheduling cannot be based only on fairness or throughput; it must also account for timing budgets and degradation mode.
- Memory cannot be treated as an unconstrained personalization substrate; it must carry source, timestamp, and invalidation rules.
- Tool invocation cannot be a generic API call; it must be capability-scoped, policy-checked, and logged. In some cases it must be converted into a non-actuating recommendation subject to a deterministic authority.Why is the research lineage of agentic OS important? Because it reframes agents as managed workloads rather than as isolated scripts. For example:
- AIOS seeks to avoid unrestricted LLM and tool access, so it describes an AI agent operating system with kernel services that prohibit that access [2].

- Magentic-One demonstrates the value of a lead orchestrator, showing that complex tasks require coordinated multi-agent execution rather than single-shot prompting [4].
- MemGPT and related memory systems show that long-horizon agent behavior depends on explicit memory hierarchy and context paging [5].

The research lineage therefore provides the vocabulary of kernels, scheduling, memory, tools, and agents. EAOS picks up where the lineage leaves off, adding physical-system semantics like embodiment and capability model, criticality class, operational design domain, and human consent. It makes safety, determinism, embodiment, and lifecycle governance architectural primitives rather than deployment afterthoughts.

EAOS USE CASES

The following examples show how EAOS enables end-to-end embodied workflows rather than isolated AI features.

Automotive use case:
Fleet-aware predictive maintenance and service orchestration

Objective. Detect an emerging component failure before it causes a breakdown, explain the evidence, protect vehicle operation, and coordinate an approved service intervention. This use case illustrates how EAOS equips developers to create agents that connect in-vehicle sensing, fleet intelligence, diagnostic reasoning, human approval, service systems, and deterministic vehicle authority in one governed workflow.

1. **Trigger and goal admission.** An intermittent high-voltage coolant-pump anomaly, thermal excursion, or diagnostic event creates a maintenance goal. Agents classify it as advisory/service authority, verify vehicle identity, software and calibration baseline, consent, connectivity, and available compute, and assign latency, tool, privacy, and evidence budgets. EAOS provides interfaces for event capture and goal admission.
2. **Context assembly.** Agents gather diagnostic trouble codes, freeze-frame data, pump current, coolant temperatures, ambient conditions, recent drive

cycles, mileage, maintenance history, component configuration, applicable service procedures, and fleet patterns for comparable vehicles. EAOS provides services for multi-device context management and governance.

3. **Multi-agent analysis.** A diagnostic orchestrator agent coordinates anomaly-detection, thermal-system, service-knowledge, fleet-correlation, and warranty agents. They rank hypotheses such as pump degradation, connector intermittency, coolant restriction, sensor bias, or software interaction. A critic agent challenges unsupported conclusions and requests additional bounded evidence where needed. EOS provides services for planning , scheduling and execution based on pre-defined archetypes or use an LLM for dynamic multi-agent task decomposition.
4. **Typed proposal**. EAOS converts the result into a machine-checkable diagnostic package containing ranked hypotheses, confidence, supporting evidence, remaining-risk estimate, recommended inspections, permitted driver guidance, urgency, required parts, and expiration conditions.
5. **Deterministic validation and release.** Rule-based validators check diagnostic consistency, sensor validity, current operating state, approved service procedures, cybersecurity policy, warranty rules, and whether safe fallback remains available. The vehicle controller independently decides whether to retain normal operation, activate an already-certified de-rating mode, or issue a deterministic warning; EAOS supplies only the validated proposal to MCPs for execution by providing safety/security agentic services.
6. **Service orchestration.** With driver or fleet approval, capability-scoped tools locate an eligible service center, propose an appointment, reserve approved parts, and transfer the evidence package. EAOS provides governed access to agents/MCPs.

Robotics use case: safe autonomous material retrieval and delivery

Objective. Enable a mobile manipulator to accept a natural-language request such as “bring the sealed inspection kit from storage to workstation 4,” then safely navigate, identify, grasp, transport, and hand off the correct object in a shared workspace. The scenario spans human interaction, perception, semantic planning, navigation, manipulation, safety supervision, and task evidence.

1. **Intent normalization and admission.** User agents resolve the requested object, source, destination, requester identity, urgency, and completion criteria into a typed goal. Policy checks confirm authorization to access the storage zone, payload limits, robot readiness, battery reserve, workspace operating mode, and whether human confirmation is required. EAOS provides interfaces for user input capture and goal admission.
2. **World and capability context.** User agents assemble current map version, robot pose, dynamic obstacles, restricted zones, and recent task memory. EAOS provides services for multi-device context management and governance.
3. **Hierarchical multi-agent planning.** A mission orchestrator decomposes the goal into navigate, localize, inspect, identify, grasp, verify, transport, and handoff stages. Navigation, perception, manipulation, inventory, and human-interaction agents generate candidate plans. EAOS arbitrates shared resources and checks that every step maps to an approved capability contract. EOS also provides services for planning , scheduling and execution based on pre-defined archetypes or use an LLM for dynamic multi-agent task decomposition.
4. **Capability-scoped execution.** The runtime invokes bounded EAOS provided interfaces for user defined route planning, camera inspection, barcode or visual identification, arm motion planning, gripper control, and status communication.
5. **Two-key motion release.** Agent reasoning produces typed navigation or manipulation proposals while deterministic planners and validators independently check controller readiness before releasing each motion segment.
6. **Verified handoff and evidence.** At the destination, EAOS verifies workstation identity, recipient or drop-zone readiness, and object identity before release.

MAIN DESIGN PATTERNS IN AGENTIC SYSTEMS AND EAOS

The most important design pattern in embodied agentic systems is the separation among four main phases: intent (to specify which outcome is desired), planning (to decompose that intent into candidate steps), execution (to invoke bounded tools or reasoning services), and release (to determine what the output may affect). Many agent frameworks merge those phases inside an agent loop, but EAOS must separate them explicitly because different phases have different assurance requirements. For example, a planning error may be corrected through re-planning, but a release error can cross a safety-relevant boundary.

Another important pattern is typed mediation. Instead of crossing a tool or vehicle boundary as free-form text, agent outputs should be converted into typed objects such as a diagnostic hypothesis list, a service recommendation, or a policy exception request. Typed mediation enables schema validation and deterministic comparison with allowable ranges. It also allows such validators as safety, cybersecurity, timing, and domain to reason over a single artifact.

The patterns in automotive systems, robots and drones require stronger contracts than patterns in typical enterprise or consumer agents. For example, every plan must be decomposable into auditable steps, and every action proposal must be convertible into a deterministic representation.

SYSTEM SOFTWARE ARCHITECTURE CONTEXT OF EAOS TAKING AUTOMOTIVE DOMAIN AS A REFERENCE

Up to now, automotive electrical and/or electronic (E/E) architectures have been based on federated electronic control units (ECUs), where each function is tightly bound to a dedicated controller. Software ownership, validation boundaries, and timing behavior are relatively localized. Feature improvement is slow and integration cost is high. However, the architectures have evolved to encompass three elements:

- **Domain controllers –** Domain consolidation reduces hardware fragmentation but often preserves functional silos.

- **Zonal architectures –** These decouple physical I/O aggregation from functional compute to simplify wiring and enable more integrated sensor, actuator, and power-distribution abstractions.
- **Centralized high-performance computing –** This brings advantages like cross-domain workloads, service-oriented applications, and over-the-air (OTA) functions.

Those elements apply readily to EAOS. As examples:

- Centralized and zonal SDV architectures create a natural substrate for semantic vehicle services.
- Layers as diverse as AUTOSAR Adaptive, Eclipse S-CORE, POSIX-based environments, DDS, SOME/IP, service discovery, Ethernet, TSN, container runtimes, and hypervisor-managed partitions support dynamic, service-oriented workloads on high-performance compute.

EAOS sits above deterministic platforms as an intelligence control plane that understands service contracts, criticality boundaries, update policy, and domain ownership.

In fact, EAOS becomes more architecturally relevant as the industry migrates toward centralized and zonal E/E systems. Zonal architectures aggregate local I/O around physical zones, while central compute platforms host cross-domain logic, high-bandwidth processing, and software-defined applications. This migration reduces hardware fragmentation but increases the need for common abstractions, service contracts, and platform governance. The higher-level intelligence control plane of EAOS prevents agentic functions from becoming tightly coupled to individual middleware interfaces or feature-specific data models that limit reuse and make validation more difficult.

EAOS should be designed to coexist with multiple software architecture styles, including the following:

- **AUTOSAR Classic –** it may interact through gateways, diagnostics, or service proxies rather than direct control.
- **AUTOSAR Adaptive, Eclipse S-CORE, or POSIX-based central compute environment –** it may consume service discovery, ara::com-style services, mw::comm-style services, DDS topics, SOME/IP services, or platform APIs.

- **Hypervisor-based system –** it may run in one or more non-safety partitions while communicating with safety partitions through controlled channels.
- **Containerized system –** its runtime components may be packaged as services with explicit resource quotas and update policies. The architecture must therefore be transport-agnostic but semantics-aware; it should abstract integration mechanisms without hiding latency, criticality, ownership, or failure behavior.

MAIN CONSTRAINTS ON EMBODIED SYSTEMS

Embodied deployment imposes constraints that are fundamentally different from desktop, cloud, or mobile agent systems.

- Real-time behavior must be bounded, not merely fast on average.
- Safety behavior must be available for analysis under fault conditions, not merely plausible under nominal demonstrations.
- Mixed-criticality integration must guarantee freedom from interference across ASIL-/SIL-classified and QM workloads that share resources like CPUs and accelerators. That requires spatial isolation, temporal isolation, resource accounting, and deterministic fallback.

For EAOS, mixed criticality means that agentic workloads must be classified according to the authority they request. As examples:

- Informational agents may summarize diagnostics or explain system state.
- Advisory agents may recommend actions to a driver, service technician, or deterministic controller.
- Supervisory-within-envelope agents may propose bounded parameter adjustments within a pre-certified envelope.
- Direct actuation by probabilistic agents should remain outside the EAOS reference architecture for safety-critical paths.

The platform must encode those distinctions explicitly so that disparate agent types—e.g., agents for cabin personalization, predictive maintenance, chassis optimization, and ADAS advisories—are not governed by the same execution and release policy.

The most difficult constraint is bounded behavior under variable conditions. Agentic workloads are subject to variables as different as model response length, retrieval complexity, and tool latency that push boundaries on behavior. Automotive systems, for example, cannot rely on average-case responsiveness for safety-adjacent behavior. EAOS must therefore treat every reasoning episode as a budgeted workload, with upper limits on wall-clock time, number of reasoning turns, token budget, and cancellation policy. Then, if the episode cannot complete within the envelope, the safe result is not to keep reasoning indefinitely; it is to terminate, return a bounded failure state, and allow deterministic baseline behavior to continue.

Mixed criticality is a full-system property, not a label attached to a single process; as such, freedom from interference is about more than CPU scheduling. Agentic workloads may contend for a wide variety of resources including accelerator cycles, memory bandwidth, storage I/O, and cloud connectivity. Otherwise, a cabin assistant, for instance, can degrade an ADAS-adjacent perception pipeline if accelerator arbitration is poorly configured. Thus, EAOS must integrate with platform-level resource management and define workload classes to specify factors like priority, preemption behavior, and degradation behavior.

OBJECTIVES AND PRINCIPLES INFORMING EAOS DESIGN

As shown below, EAOS is guided by six primary design objectives.

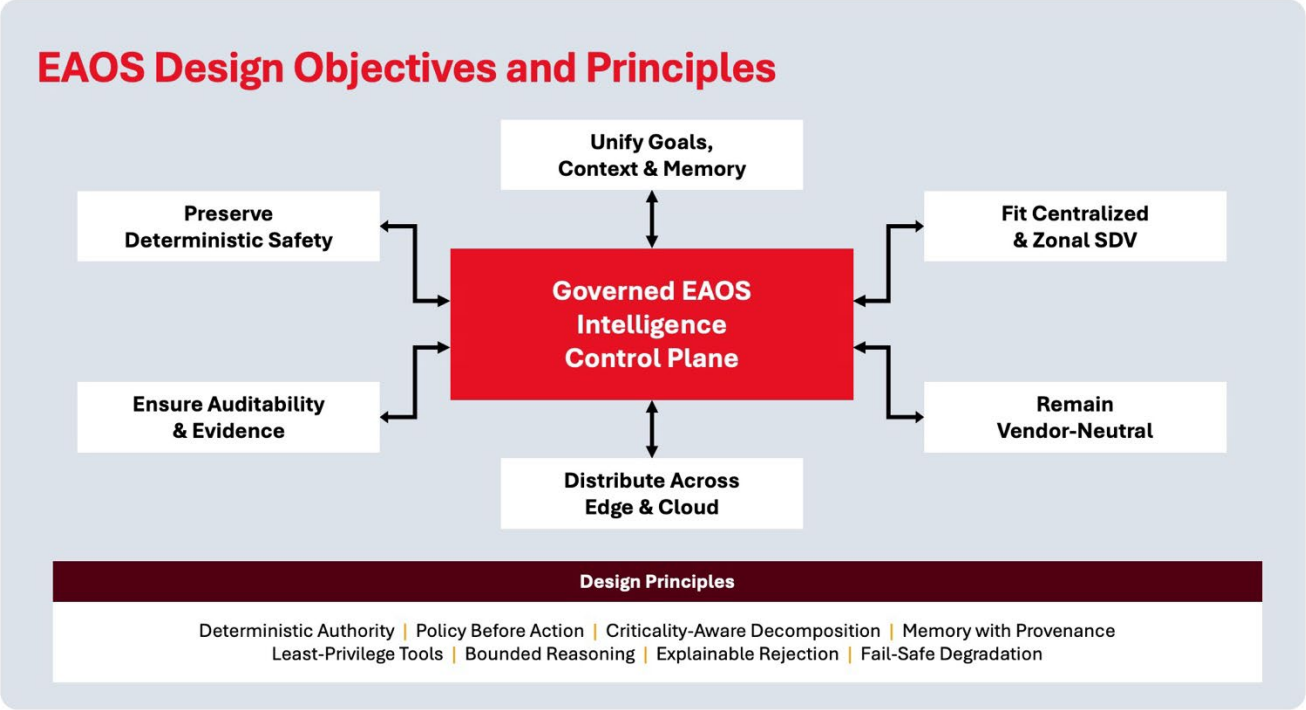

Figure 2. Design objectives and governing principles

1. Preserve deterministic safety behavior by ensuring that certified control logic, safety monitors, and fallback paths remain authoritative.
2. Unify goal, memory, and context management so that agent behavior is not fragmented across disconnected prompts and application-specific stores.
3. Fit heterogeneous embodied topologies—from centralized and zonal vehicles to distributed robotic compute, field devices, and cloud-connected fleets—without hiding criticality, latency, or ownership constraints.
4. Distribute across edge and cloud, while keeping live physical authority at the edge.
5. Ensure auditability across goals, context, tools, policies, validation outcomes, and releases.

6. Remain vendor- and embodiment-neutral so that vehicle manufacturers, robotics vendors, and industrial system providers can map the system onto different hardware, hypervisors, middleware stacks, and cloud toolchains.

The corresponding design principles of EAOS are:

Policy-before-action	**Control-plane centrality**	**Criticality-aware decomposition**
Deterministic authority	**Memory with provenance**	**Least-privilege capability use**
Bounded reasoning	**Explainable rejection**	**Fail-safe degradation**

Of those principles, policy-before-action is especially important; no agent should discover after invoking capability that it exceeded its authority. The EAOS kernel should evaluate factors like permission, criticality, human consent, and operational design domain or workspace envelope before allowing a tool call to proceed. Rejection should be a normal architectural outcome, not an exception path.

A practical EAOS design should be evaluated against several attributes of architectural quality:

- **Safety –** Unsafe outputs are rejected before they cross authority boundaries.
- **Security –** Agents cannot expand their capability surface or infer privileged state through indirect channels.
- **Reliability –** Behavior remains predictable even with missing data, stale memory, degraded connectivity, and model failure.
- **Portability –** The same conceptual goal and capability model can map onto different hardware (vehicles, robots, manipulators, drones) and software (middleware stacks, compute platforms, cloud environments).
- **Maintainability –** Skills, models, policies, and validators can evolve independently without breaking assurance arguments.
- **Explainability –** The system produces structured traces that engineers can inspect, replay, and correlate with physical events.
- **Scalability –** Adding new agents or embodiments does not create uncontrolled interactions or validation explosion.

The design principles extend to the ways in which EAOS should not behave. For example, to prevent application of agentic AI into contexts of deterministic authority, such as device control and actuation, EAOS should not be able to do the following:

- Make probabilistic models the direct authority for safety-critical control.
- Bypass functional-safety mechanisms, calibration governance, cybersecurity processes, motion-control ownership, or domain-controller ownership.
- Assume that cloud connectivity is always available.
- Treat personalization, mission, or learned memory as automatically valid across machines, users, sites, hardware revisions, or software versions.
- Allow agent-created plans to invoke robot skills or actuators without capability declarations and policy checks.

The reference architecture (see below) realizes these design objectives and principles by assigning each responsibility to a distinct layer or cross-cutting assurance plane. That makes it easier for developers to validate and propagate the design across vehicle, drone, and robotics programs.

THE SEVEN-LAYER EAOS REFERENCE ARCHITECTURE

The generic agentic OS described in section 3 manages prompts, scheduling, context, memory, agents, and tool access in a digital environment. As shown below, EAOS retains those useful system abstractions but hardens them for cyber-physical embodiments by introducing seven explicit layers, four governed control loops, a deterministic authority boundary, safe fallback, and a cross-cutting embodied-system assurance plane.

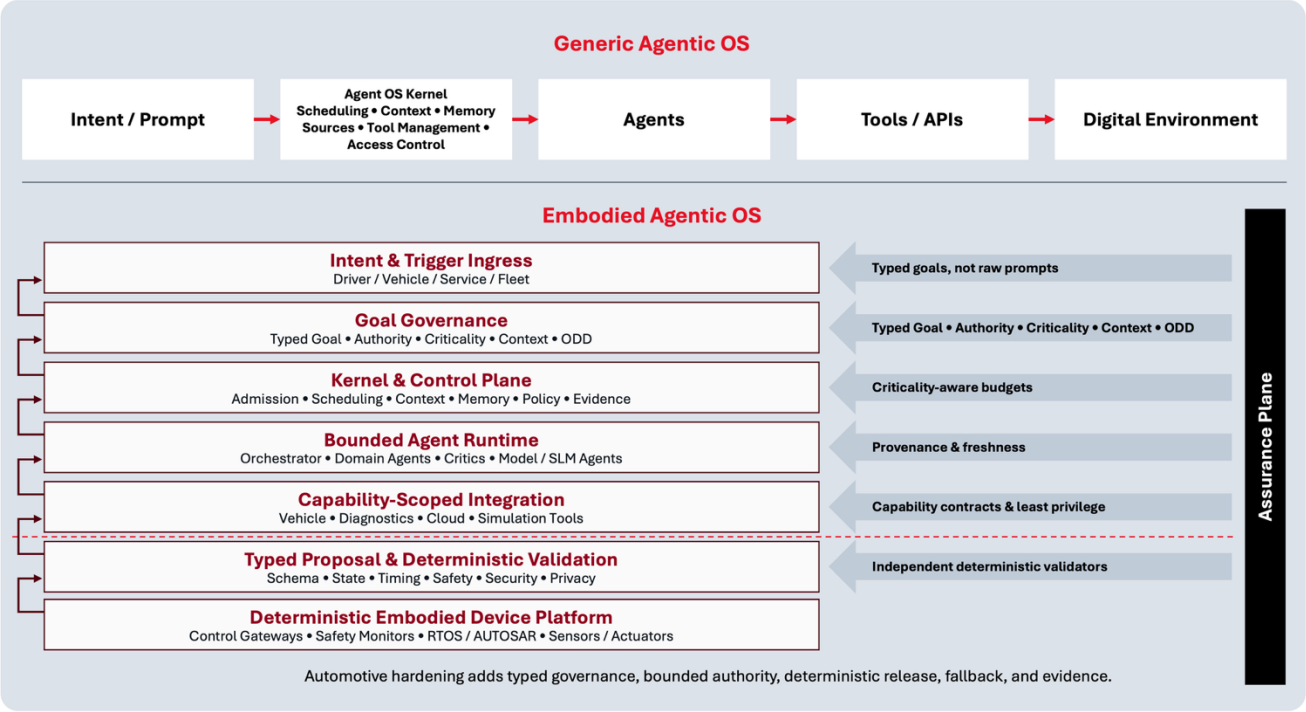

Figure 3. Seven-layer reference architecture

EAOS is not a replacement for an RTOS, hypervisor, ROS 2 or AUTOSAR platform, POSIX environment, middleware stack, safety monitor, motion controller, or certified control function. It is the semantic intelligence control plane above those platforms, responsible for governing agentic reasoning.

A generic agent framework is principally concerned with successful cognition and tool use. EAOS must go beyond that and determine whether:

- a goal is admissible in the current physical state
- its data is fresh and applicable
- the requesting agent has domain authority
- execution can complete within timing, energy, and resource budgets
- a capability's side effects are permitted
- the resulting proposal is safe to release.

As described earlier, probabilistic reasoning plays a role in EAOS. Independently, though, deterministic platform software validates and retains final authority for safety-relevant behavior.

Layer 1 — Intent and Trigger Ingress. The architecture accepts driver or passenger requests; vehicle and diagnostic events; service, OTA, and fleet objectives; engineering investigations; and domain opportunities. Unlike generic prompt ingress, EAOS normalizes these inputs before reasoning begins. It retains source identity, domain, requested outcome, privacy class, and triggering vehicle state so that downstream decisions remain attributable and replayable.

Layer 2 — Goal Governance. EAOS converts ingress into a typed goal instead of treating raw natural language as the unit of execution. The goal declares the actor and domain; desired outcome; authority and criticality; deadline and context dependencies; approved tools; operational-design-domain applicability; consent and privacy conditions; evidence obligations; fallback; and release path. Goal admission checks policy, entitlement, current state, conflicts, timing feasibility, and available resources. The platform can refuse, defer, downgrade, or cancel work before invoking a model or tool.

Layer 3 — Kernel and Control Plane. This layer provides admission and mixed criticality-aware scheduling; policy enforcement; context and memory management; conflict arbitration; resource accounting; lifecycle control; observability; and evidence capture. It coordinates the goal-governance and context-and-memory loops. Each reasoning episode has explicit limits for elapsed time, planning depth, tool calls, retrievals, context size, compute use, network class, and cancellation. Whereas generic agentic OS scheduling often optimizes throughput or task completion, EAOS must also preserve freedom from interference and terminate safely when a workload exceeds its envelope.

Context and memory governance. The control plane builds a bounded context package from vehicle and user domains. Memory spans working, session, vehicle-local, user, service, fleet, engineering, policy, and domain tiers. Every object carries provenance, timestamp and other key attributes. This is stricter than generic agent memory because fresh data may still be unsuitable when it comes from a degraded sensor, obsolete calibration, different vehicle configuration, or unauthorized user context.

Layer 4 — Bounded Agent Runtime. The runtime hosts orchestrators and planners, specialist domain agents, retrieval agents, critics and verifiers, model and skill adapters, deterministic adapters, and proposal builders. Agents execute only within the goal's admitted authority, context, tool set, and resource budget. Their outputs remain non-authoritative. Runtime isolation scales with risk: observe-only workloads may use ordinary containers, while supervisory workloads require stronger partitioning, signed artifacts, independent validators, tighter quotas, and explicit watchdog behavior.

Due to embodied runtime constraints, variable model latency, retrieval depth, retries, and tool delays cannot be allowed to become unbounded vehicle/robotic behavior. EAOS therefore treats cancellation, safe rejection, degradation to advisory or observe-only mode, and continuation of baseline deterministic behavior as normal, successful outcomes. Agent autonomy is intentionally constrained by authority class, mixed-criticality placement, operational state, and domain envelope.

Layer 5 — Capability-Scoped Integration. Vehicle/Robot diagnostic, cloud, simulation, service, and engineering interfaces are exposed through governed capability contracts rather than ambient APIs. Each contract declares schema and permissions, preconditions and postconditions, side effects, criticality and latency, data sensitivity, rollback behavior, rate limits, logging, and validator hooks. The capability-and-tool loop checks goal scope, agent identity, consent, entitlement, vehicle state, cybersecurity posture, and expected side effects before invocation. This least-privilege mediation is an embodied addition because a valid API call may still be unsafe, untimely, unauthorized, or incompatible with the current vehicle/robot configuration.

Layer 6 — Typed Proposal and Deterministic Validation. Before crossing the automotive/robotic authority boundary, EAOS converts agent results from free-form output into machine-checkable proposals. Each proposal includes schema, units, ranges, assumptions, confidence, evidence references, validity window, required validators, and fallback. Independent deterministic validators check current state and freshness; timing and calibrated limits; safety envelope; cybersecurity; privacy and consent; feature entitlement; compatibility; and fallback availability.

The proposal-and-validation loop ends in explicit acceptance or rejection. Model confidence alone never authorizes release.

Layer 7 — Deterministic Embodied Platform. Final authority remains with certified controllers and safety monitors. An accepted proposal is consumed only through deterministic timing and control logic. A rejected, expired, incompatible, or unvalidated proposal produces no actuation. The previous calibrated state or safe baseline remains active, and EAOS records the reason. The dashed authority boundary in Figure 3 is architectural rather than merely logical. It should be enforced through partitioning, approved interfaces, signed artifacts, schema checks, watchdogs, timeouts, and release arbitration.

Four coordinated control loops are proposed as part of EAOS reference architecture. The seven layers are activated through four closed loops.

- **Goal governance –** Admits, classifies, budgets, schedules, and, if necessary, cancels or downgrades work.
- **Context and memory –** Assembles evidence, checks provenance and applicability, retrieves bounded memory, and invalidates stale knowledge from outcomes.
- **Capability and tool –** Grants least-privilege access, checks preconditions and side effects, invokes the tool, and records results.
- **Proposal and validation –** Converts probabilistic output into a typed artifact, selects validators, releases or rejects the artifact, and returns evidence for policy and engineering improvement.

Coordinating those loops in the control plane prevents important embodied checks from being buried inside opaque agent prompts.

Cross-cutting embodied assurance plane. Safety, cybersecurity, privacy, identity, mixed-criticality policy, OTA governance, regulatory constraints, compatibility, observability, fallback, and evidence retention apply across all seven layers. Every goal, context object, memory retrieval, agent execution, tool call, proposal, validator decision, release, rejection, fallback transition, and synchronization event is versioned and traceable. Compatibility is checked across hardware revision, software baseline, calibration, model and skill versions, policy set, region, vehicle configuration, and feature entitlement.

ENFORCING SAFETY, DETERMINISM, AND MIXED CRITICALITY

EAOS includes an overlay for safety, determinism, and mixed criticality that differentiates it from a consumer multi-agent platform. The overlay defines:

Criticality classes	**Partition maps**
Communication rules	**Timing envelopes**
Validation gates	**Fallback monitors**
Evidence requirements	**Release authority**

Each class should have different requirements for confidence, context freshness, deterministic validation, and fallback behavior.

As depicted below, determinism is enforced by constraining where non-deterministic reasoning can influence the system. Language-model output, for instance, should not be treated as an executable command. It should be transformed into a typed proposal, checked against schema, verified against current state, evaluated by policy, bounded by safety envelopes, and released only through deterministic arbitration. If any check fails, rejection is the correct output. The system should then fall back to baseline behavior, retain the previous calibrated state, alert the relevant user or service, and log the reason for rejection in a form suitable for engineering analysis and safety evidence.

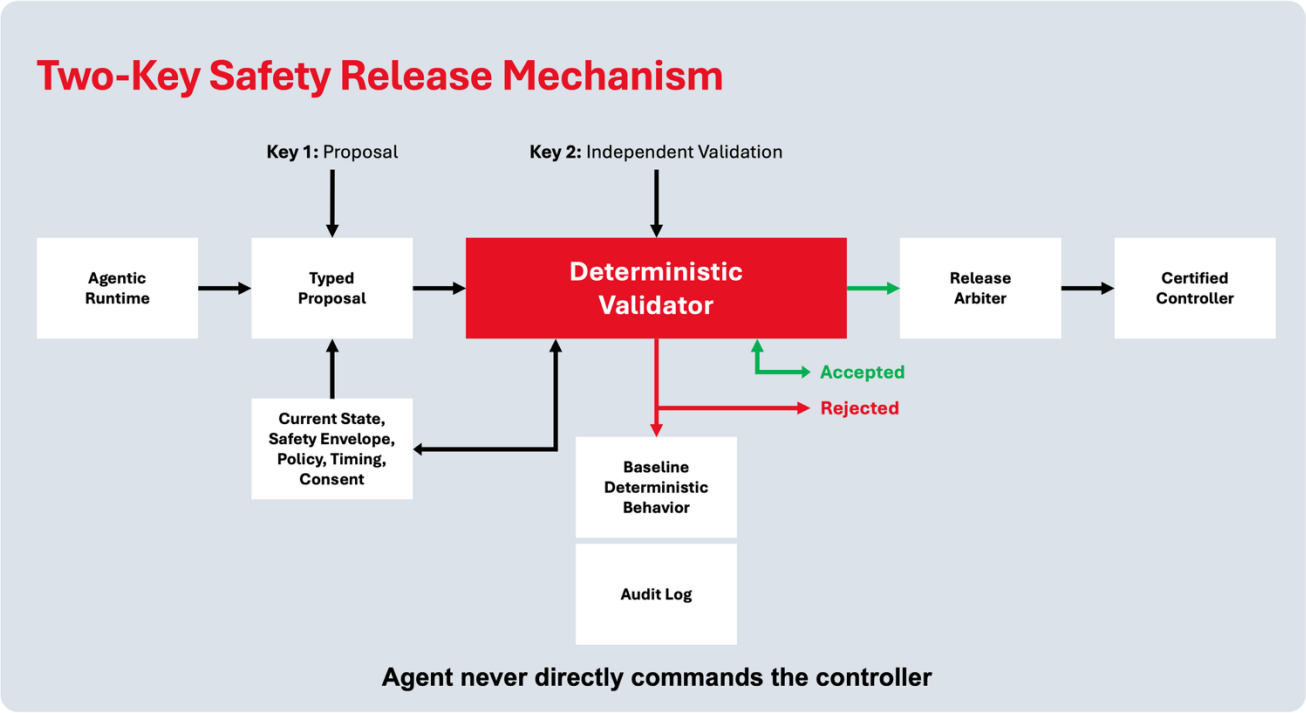

Figure 4. Two-key release mechanism separating probabilistic proposals from deterministic vehicle authority

An example of a useful implementation pattern is a two-key release mechanism. The agentic side can generate a typed proposal (see Key 1 in Figure 4), but deterministic software must independently validate and release it. For a motion-adjacent supervisory recommendation, the EAOS agent may suggest a bounded optimization within a pre-certified domain envelope based on context. The deterministic validator then checks calibrated limits, operating state, feature mode, sensor validity, driver state, road context, timing, and safety envelope before accepting or rejecting the proposal (see Key 2). If accepted, the deterministic controller applies the change according to its own timing and control logic. If rejected, the system retains the previous state and records the rejection for audit.

The mixed-criticality overlay should define two types of control:

- **Structural –** Including partition boundaries, memory isolation, network segmentation, approved interfaces, tool capability scopes, and signed deployment artifacts
- **Behavioral –** Including watchdogs, timeout policies, confidence thresholds, fallback state machines, anomaly detection, and release arbitration

Finally, the overlay should define the type of evidence to be collected for each authority class:

- **Informational outputs –** Basic traceability
- **Advisory outputs –** Context provenance and policy approval
- **Supervisory proposals –** Schema validation, safety-envelope checking, timing validation, and deterministic acceptance by a domain controller
- **Actuation-relevant outputs –** Strongest evidence and certification treatment, under the authority of the deterministic controller.

TRADITIONAL OS AND EXISTING AGENT FRAMEWORKS – WHERE EAOS FITS

Traditional operating systems such as Linux, POSIX environments, RTOSes, and hypervisor-managed platforms provide essential primitives such as scheduling, memory management, device abstraction, networking, and access control. They are not made for agent fundamentals like semantic goals, agent memory, model confidence, prompt lineage, and policy-gated reasoning.

On the other hand, existing agent frameworks provide planning, tool use, memory, and orchestration. However, they generally lack automotive-grade capabilities like safety-aware boundary management, timing enforcement, OTA governance, and safety evidence capture.

EAOS sits between those worlds as an embodied specialization of the broader agentic OS direction.

Consider the distinction at the level of authority and abstraction.

- **Conventional OS –** Asks whether a process can open a file, allocate memory, or send a packet.
- **An agent framework –** Asks whether an agent can call a tool to complete a task.
- **EAOS –** Asks whether a goal is admissible in the current vehicle state, whether the selected agent is allowed to reason about that domain, whether the retrieved context is fresh and valid, whether the proposed tool call has acceptable side effects, whether the output can be deterministically checked, and whether a fallback path remains available if the proposal is rejected.

EAOS also differs at the design level.

- **Conventional OS –** Linux, QNX, AUTOSAR, ROS 2, and RTOS environments remain necessary because they provide services like scheduling, drivers, networking, and memory protection. Hypervisors remain necessary because they establish partition boundaries and support freedom from interference. Middleware remains necessary because it exposes services and transports data between components.
- **EAOS –** Rather than duplicate those responsibilities, it operates at a more semantic level while depending on lower layers for hard isolation and execution primitives. It adds goal governance, agent lifecycle management, memory provenance, capability mediation, deterministic output conversion, and safety release policy.

In the context of restriction, EAOS works differently.

- **A general multi-agent framework –** Optimizes for task success, flexibility, and extensibility.
- **EAOS –** Optimizes for admissible task success under safety, timing, privacy, cybersecurity, and evidence constraints.

That matters because many agent benchmarks reward completion even when the path includes retries, exploratory tool use, or unconstrained browsing. For example, automotive production systems must often prefer a safe rejection over a risky completion. EAOS therefore treats refusal, degradation, escalation, and fallback as

first-class successful outcomes when they preserve system safety and trustworthiness.

HOW EAOS ENABLES DEVELOPERS TO BUILD TRUSTWORTHY AGENTS

EAOS should provide robust mechanisms for security, governance, observability, and explainability.

Security

The essence of security includes least-privilege tool access, explicit capability registration, strong identity, secure boot, and policy-governed communication. Agents should never receive ambient authority over vehicle services. Each tool invocation should be checked against criteria like agent identity, goal scope, vehicle state, user consent, and expected side effects. Sensitive data should be protected by data minimization, retention policy, access logging, and privacy classification.

Governance

Governance spans model versions, skill versions, policy versions, OTA campaigns, calibration compatibility, regional regulations, feature entitlements, and rollback plans.

Observability

Observability must record elements like triggers, context sources, memory retrievals, model and skill versions, tool calls, and timing behavior.

Explainability

Explainability should not depend on vague, natural-language rationales. It should be grounded in structured traces that show which data was used, which policy was applied, which proposal was generated, and which validator accepted or rejected it.

A useful EAOS trace should include the following elements:

Goal	**Actor**	**Policy set**	**Context sources**
Memory retrievals	**Model version**	**Skill version**	**Tool calls**
Validator outcomes	**Release decision**	**Fallback path**	**Synchronization events**

For instance, developers and engineers should be able to replay the trace offline, compare it against a prior software baseline, and determine whether a failure came from context assembly, memory retrieval, planning, tool behavior, policy configuration, validator logic, or model output.

Non-presumption of trustworthiness

Taken together, those mechanisms should equip EAOS to presume that agents are powerful but not inherently trustworthy. Threats abound in forms like prompt injection, tool-output manipulation, and stale memory poisoning. Therefore, a physical-system capability should not trust a natural-language instruction simply because it was produced by an authorized model.

- **Capabilities –** Invocations should be mediated by structured checks and policy decisions. The capability broker should enforce least privilege, parameter validation, rate limits, side-effect declarations, and context requirements.
- **Memory –** Memory stores should protect against unauthorized reads, unauthorized writes, and unvalidated persistence of model-generated claims.
- **Cloud –** Synchronization should be signed, versioned, encrypted, and constrained by data minimization.

IMPLEMENTING, INTEGRATING, AND SCALING EAOS

Implementing

EAOS should map agent workloads to compute resources according to power, privacy, criticality, and model size. The resource manager should enforce quotas and prevent best-effort agent workloads from starving deterministic control, safety monitoring, and communication tasks.

- **Local –** Local models may handle privacy-sensitive, latency-sensitive, and connectivity-independent tasks. Implementation must account for heterogeneous compute including general-purpose CPUs, GPUs, NPUs/DSPs, microcontrollers, memory controllers, and high-speed networking.
- **Cloud –** Cloud models may support deeper reasoning, fleet analytics, large-scale retrieval, and offline engineering workflows.

Hardware implementation should distinguish among types of execution, so that computationally intensive reasoning does not interfere with timing, safety, or communication.

- **Model execution –** Large local models or multimodal models may require GPUs, NPUs/DSPs, or accelerator-backed inference runtimes.
- **Orchestration logic –** Orchestration and policy engines may run efficiently on CPUs but require predictable scheduling and protected storage.
- **Validation logic –** Validators may need to run in trusted partitions or safety-adjacent environments, especially when they gate supervisory proposals.
- **Deterministic control –** Deterministic controllers may run on RTOS, AUTOSAR Classic, safety islands, or domain-specific control processors.

Integrating

Middleware integration should be explicit rather than assumed.
EAOS may consume and expose services through:

AUTOSAR Adaptive/ Eclipse S-CORE services	**Classic gateways**	**DDS topics**
SOVD-style service interfaces	**Diagnostics protocols**	**SOME/IP services**
Ethernet/TSN networks	**Cloud APIs**	

Within the constraints of safety and cybersecurity, deployment models may use hypervisor partitions, containers, microservices, sidecars, and service meshes.

Scalability

Scalability is a prominent factor on three levels:

- **Within the vehicle –** Across domains and partitions, scalability requires a disciplined packaging and lifecycle model. Agents, skills, policies, validators, prompts, and tool manifests should be versioned independently but compatibility-checked as a release set.
- **Across the fleet –** OTA deployment should support staged rollout, shadow execution, canary analysis, rollback, and post-deployment monitoring. Of those, shadow execution is particularly valuable: an agent can run in observe-only mode, produce proposals, and collect validation results without affecting vehicle behavior.
- **Across the engineering ecosystem –** Reusable skills, simulation assets, test cases, and evidence packages allow engineering teams to measure false acceptance, false rejection, latency, resource use, and trace completeness before authorizing. The result is more safety on the path from research prototype to production deployment.

AREAS OF ADDITIONAL RESEARCH INTO IMPROVING EAOS

In several essential respects, additional research would improve EAOS:

- **Scheduling –** Bounded real-time scheduling for agentic workloads is unresolved because reasoning cost can vary with prompt, context, model, tool latency, and retry behavior.
- **Verification –** Verification of stochastic, tool-using systems remains difficult because the behavior is not fully specified by source code.
- **Memory –** Provenance, invalidation, and conflict resolution become safety-relevant when long-term memory influences vehicle-facing recommendations.
- **Goal-action alignment –** Action-level alignment requires assurance that an apparently benign goal does not decompose into unsafe intermediate actions.
- **Explainability –** Engineering-grade explainability requires structured traces that can support debugging, safety analysis, cybersecurity investigation, and regulatory review.
- **Standardization –** The industry lacks common schemas for goals, agent capabilities, skill manifests, policy decisions, memory provenance, validation results, edge-cloud state exchange, and agent safety cases.
- **Benchmarking –** Beyond the success of a given task, EAOS benchmarks should measure deadline adherence, rejection correctness, fallback quality, resource isolation, trace completeness, and privacy behavior. They should reflect the system's ability to withstand degraded sensors, stale context, connectivity loss, conflicting goals, and adversarial tool outputs.

Progress in EAOS development would benefit greatly if the industry were to define reusable safety cases for agentic functions. Conventional safety cases often depend on specified behavior, static requirements, bounded failure modes, and validated control logic, but the dynamics of agentic systems defy most of those conditions. EAOS can help by constraining those dynamics into typed goals, controlled tools, deterministic validators, and structured traces. However, not all constraints apply to all authority classes; for example, the safety case for an informational diagnostics assistant differs significantly from that for a supervisory chassis optimization agent.

CONCLUSION

EAOS is best described as a governed system capability for hosting agentic intelligence inside embodied autonomous machines, including software-defined and AI-defined vehicles, mobile and industrial robots, humanoids and drones. It does not replace deterministic controllers, safety monitors, certified software, hypervisors, RTOSs, Linux, ROS 2, AUTOSAR, or middleware. Instead, it provides the semantic control plane, kernel-like agent services, policy enforcement, observability, and edge-cloud governance needed for agentic functions.

The main architectural feature of EAOS is its discipline in separating probabilistic reasoning from deterministic authority. EAOS defines:

- where goals enter
- how world and system context is assembled
- how memory is governed
- how agents are scheduled
- how capabilities are brokered
- how proposals are converted into typed artifacts
- how deterministic validators release or reject outputs
- how evidence is retained.

Its discipline allows agentic capabilities to become governed infrastructure rather than isolated feature logic, enabling innovation across automotive, drones, and robotics without eroding the safety, reliability, and accountability that embodied software requires.

In practical terms, EAOS should be adopted incrementally. The first production uses should emphasize observe-only, advisory, diagnostic, simulation, and engineering-assist modes where the value is high and the authority boundary is conservative. As the platform matures, validated supervisory envelopes can be introduced for domains such as energy, thermal management, comfort, navigation assistance, manipulation, maintenance, and mission optimization. Safety-adjacent motion, locomotion, and actuation domains should remain carefully bounded and should use deterministic planners and controllers as final authorities.

Over time, the combination of a reference architecture, reusable manifests, standardized capability contracts, memory provenance, policy engines, validators, observability, and edge-cloud governance can form the basis for a production-grade embodied intelligence platform spanning automobiles, drones, and robotics.

REFERENCES

[1] Ge, Y. et al. LLM as OS, Agents as Apps: Envisioning AIOS, Agents and the AIOS-Agent Ecosystem. arXiv:2312.03815, 2023.

[2] Mei, K. et al. AIOS: LLM Agent Operating System. arXiv:2403.16971, 2024/2025 versions.

[3] Zhao, Z. et al. KAOS: Large Model Multi-Agent Operating System. arXiv:2406.11342, 2024.

[4] Fourney, A. et al. Magentic-One: A Generalist Multi-Agent System for Solving Complex Tasks. arXiv:2411.04468, 2024.

[5] Packer, C. et al. MemGPT: Towards LLMs as Operating Systems. arXiv:2310.08560, 2023.

[6] Road vehicles — Functional safety — ISO 26262 Parts 1–12, second edition, 2018.

[7] Mauser, L. et al. Towards Mixed-Criticality Software Architectures for Centralized HPC Platforms in Software-Defined Vehicles. arXiv:2506.05822, 2025.

[8] Bandur, V. et al. Making the Case for Centralized Automotive E/E Architectures. IEEE TVT, 2021.

[9] Shirasat, G, Snapdragon Ride Flex SoC: The central compute solution that's bringing the SDV vision to reality https://www.qualcomm.com/news/onq/2023/01/snapdragon-ride-flex-soc-central-compute-solution-bringing-software-defined-vehicle-vision-to-reality

[10] ISO. Road vehicles — Functional safety — ISO 26262 Parts 1–12, second edition, 2018.

[11] Chhikara, P. et al. Mem0: Building Production-Ready AI Agents with Scalable Long-Term Memory. arXiv:2504.19413, 2025.

ABBREVIATIONS

AD	Automated driving
ADAS	Advanced Driver Assistance Systems
API	Application programming interface
AUTOSAR	(AUTomotive Open System ARchitecture)
CPU	Central processing unit
DDS	Data Distribution Service
DSP	Digital signal processor
DTC	Diagnostic trouble code
E/E	Electrical and/or electronic
ECU	Electronic control unit
GPU	Graphics processing unit
HVAC	Heating, ventilation, air conditioning

I/O	Input/output
NPU	Neural processing unit
OEM	Original equipment manufacturer
OTA	Over the air
QM	Quality management
ROS	Robot Operating System
RTOS	Real-time operating system
SDV	Software-defined vehicle
SOME/IP	Scalable service-Oriented MiddlewarE over IP
SOVD	Service-oriented vehicle diagnostics
TSN	Time-Sensitive Networking