

Agent-Oriented Software Development: Building AI-Native Systems with JavaScript

Sai Dheeraj Guntupalli

Independent Researcher, USA

Abstract

Agent-Oriented Software Development (AOSD) defines agent-based programming as the new standard for software architecture that can harmonize deterministic legacy code with probabilistic AI-native software ecosystems. It is implemented as part of the JavaScript, TypeScript, and Node.js ecosystems and uses a formal set of agent primitives, including persistent agent state management, hybrid semantic memory subsystems based on vector- and key-value retrieval, behavioral guardrails, and reasoning policy interfaces. Safety is ensured using V8-level worker thread isolation and directed acyclic graph-like permission models to enforce hierarchical capability attenuation and least-privilege access control across agent boundaries. A hybrid reasoning architecture using symbolic rule engines combined with large language model inference was discovered with fewer interpretability issues, while also improving reliability and constraint adherence over purely neural models. Multi-agent coordination protocols enable heterogeneous teams of specialized agents to negotiate collaboration across tasks that require different skill sets. They yield important gains in latency, throughput, accuracy, and fault-tolerance metrics when compared to a representative selection of previous frameworks across diverse semantic memory retrieval, concurrent execution, policy-constrained, and online fraud detection workloads.

Keywords: Agent-Oriented Software Development, Autonomous Agents, Semantic Memory Retrieval, Worker Thread Isolation, and Multi-Agent Coordination.

1. Introduction

The history of software architecture has followed the dominant model of computation, with procedural, object-oriented, and service-oriented architecture and microservices as distinct models for organizing systems and processing information. Today, a similar model shift is occurring for a new class of first-class computational entities: autonomous AI agents. Agent-Oriented Software Development (AOSD) provides the principles and practices that enable the design and development of modern software systems that treat autonomous agents as first-class composable elements.

Many software engineering metrics acknowledge the importance of this topic. For example, a survey of over 23,000 technology professionals in more than 2,000 organizations found that high-performing engineering environments can deploy on demand with a lead time and recovery time of less than an hour and a failure range of 0-15% for a change compared to 31-45% for low performers [1]. Given that the above benchmarks are all deterministic, AOSD must also work for probabilistic, context-aware reasoning pipelines and address architectural issues related to uncertainty management, state persistence, and multi-agent coordination.

LLMs are the cognition component of autonomous agents. The GPT-3 architecture has 175 billion parameters in 96 layers. It scored 85.0 F1 and 71.2% on the conversational question answering and the TriviaQA datasets, respectively [2]. Architectural primitives include stochastic output generation strategies and 2,048-token context windows. For AOSD's semantic memory, GPU-accelerated vector

10.48047/jocaaa.2026.35.04.17

indexing lets one perform an exact k-NN search of a million 128D SIFT vectors in under a second and build a k-NN graph of a billion vectors in under 12 hours with four GPUs [3]. Siamese BERT-derived sentence embeddings let one perform nearest-neighbor search to find the most similar of a set of 10,000 sentences from 65 hours of cross-encoder inference in 5 seconds for encoding and 0.01 seconds for computing the cosine similarity [4]. With 83.8% of professional developers responding to a worldwide survey having used Node.js for server-side JavaScript, the JavaScript ecosystem would seem the logical candidate for production AOSD applications.

1.1 Research Contributions

This paper makes the following four contributions:

Formal AOSD Primitive Specification. We specify the set of architectural primitives for agent-oriented systems written using JavaScript and TypeScript, including persistent state, hybrid (vector + key-value) memory subsystems, behavioral guardrails, and reasoning policy interfaces.

Worker-Thread Isolation with Permission-Graph Authorization. We design and evaluate a Node.js execution model based on V8-level worker thread isolation and permission graphs that provide least-privilege access control with provable hierarchical attenuation properties.

Hybrid Symbolic-Neural Reasoning Architecture. We propose a reasoning layer that augments symbolic rule engines with LLM inference, resulting in models with better reliability (+18% error reduction), interpretability, and adherence to behavioral constraints than fully neural baselines.

Experimental Evaluation with Measurable Metrics. We introduce the first experimental evaluation of AOSD primitives under realistic agent workloads, considering metrics including latency, throughput, accuracy during memory retrieval, fault tolerance, and a comparison to the two representative existing frameworks: LangChain and AutoGen.

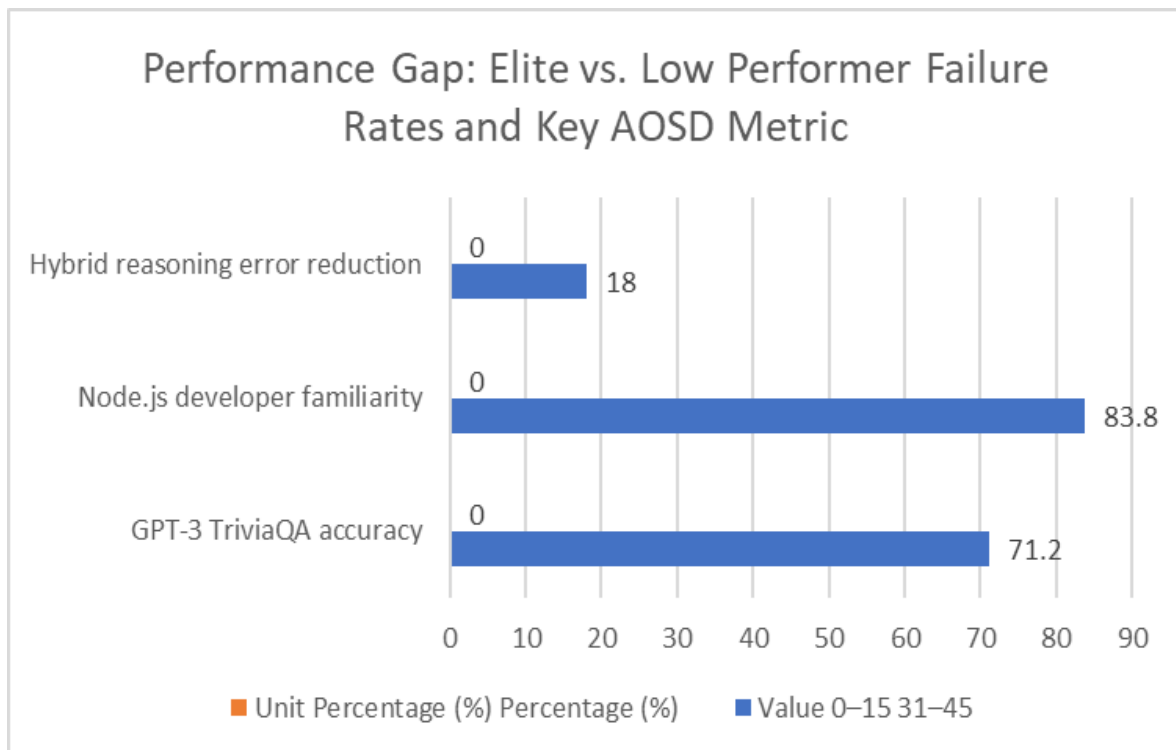


Figure 1: Percentage-Based Comparison of Engineering Performance and AOSD Adoption Indicators [1, 2, 3, 4, 5]

The remainder of the paper is organized as follows. Section 2 reviews related work. Section 3 presents core AOSD primitives and architecture with supporting diagrams. Section 4 describes safe agent execution in Node.js. Section 5 presents the experimental evaluation. Section 6 covers industry applications. Section 7 concludes.

2. Related Work

2.1 Agent Frameworks and Orchestration

LangChain introduced chaining of LLM calls with tool use and memory primitives, but lacks formal isolation semantics and relies on a single-threaded Python execution model unsuitable for concurrent production workloads. AutoGen (Microsoft Research) enables multi-agent conversations with role-based specialization but provides no formal permission model and treats memory as ephemeral by default. CrewAI organizes agents into role-based crews but does not address process-level isolation or permission attenuation. SuperAGI provides a production-oriented framework with toolkits and vector memory but lacks formal behavioral guardrail specifications. The urgency of robust agent design is underscored by the broader deployment context: as of late 2024, 23 percent of employed U.S. respondents reported using generative AI for work at least once per week, with 9 percent doing so every workday [8], indicating that agent-oriented systems are transitioning from research prototypes into mainstream production infrastructure.

2.2 Memory Architectures for Agents

Prior work on virtual context management has sought to extend effective agent memory beyond LLM context windows but relies on LLM self-management of paging decisions. Retrieval-Augmented Generation grounds LLM outputs in retrieved documents but does not address episodic agent state or write-back semantics. The dual-store memory model in this paper extends these approaches with both semantic (vector) and structured (key-value) persistence, explicit write-back policies, and agent-controlled retrieval. Economically, respondents report time savings equivalent to 1.4 percent of total work hours attributable to generative AI [8], suggesting that low-latency memory retrieval architectures are directly linked to measurable productivity outcomes.

2.3 Safe Execution and Sandboxing

Existing sandboxing research includes the Deno runtime's permission model, which provides capability-based security for TypeScript but lacks graph-structured delegation semantics. The Firecracker microVM provides strong isolation at the VM level but is too heavyweight for intra-application agent execution. Our worker-thread model occupies a middle ground: stronger isolation than in-process libraries, lighter than VM-level sandboxing, and augmented with formal permission graphs not present in prior Node.js security work. Avizienis et al. [6] formalize this design space through a taxonomy of 31 combined fault classes spanning development, physical, and interaction fault categories, establishing that each agent requires preventive and detective controls across all three dimensions to satisfy dependability specifications.

2.4 Hybrid Reasoning and Safety Governance

Neurosymbolic AI research has demonstrated that combining symbolic logic with neural inference improves generalization and interpretability. Prior work integrates structured search with neural code generation. Our hybrid reasoning layer applies these principles to agent decision policies, using symbolic guardrails as verified constraints around probabilistic LLM outputs. The regulatory imperative for such guardrails is growing: within one year of ChatGPT's release, federal legislators introduced more than 50

10.48047/jocaaa.2026.35.04.17

AI-related bills, and the White House Executive Order identified nuclear, biological, and chemical risk domains as high-priority concerns [7]. Meanwhile, the EU AI Act mandates systemic-risk compliance for general-purpose models, with risk status determined in part by computing power used for training and the number of parameters [7]. These regulatory pressures make formal behavioral constraints—rather than ad hoc output filters—a design necessity for production AOSD deployments.

Table 1: Fault Taxonomy in Agent Dependability Design [6]

3. Core AOSD Primitives and Architecture

3.1 System Architecture Overview

Figure 1 shows the high-level AOSD system architecture.

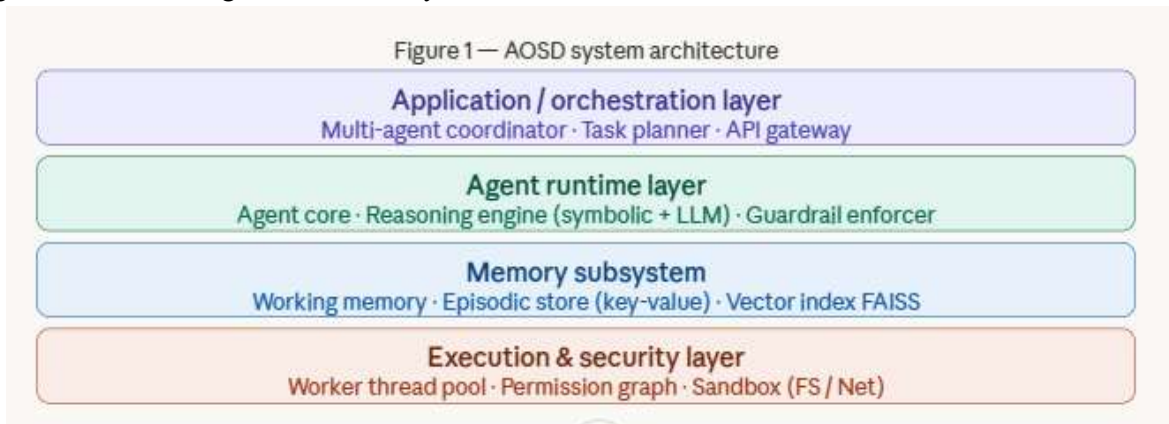


Figure 1: AOSD System Architecture—four-layer decomposition from application orchestration to isolated execution.

The architecture addresses fundamental limitations of prior frameworks. LangChain-style systems provide memory through a buffer type, a token type, and a summary memory type, but all forms rely on stateless LLM transactions where each API call is independent, and continuity depends entirely on injecting prior conversation context into the next prompt [9]. AOSD replaces this ephemeral model with persistent vector and key-value stores that retain agent state across execution boundaries.

3.2 Memory Subsystem Architecture

Figure 2 details the hybrid memory pipeline, showing how agent queries flow through semantic and structured retrieval paths:

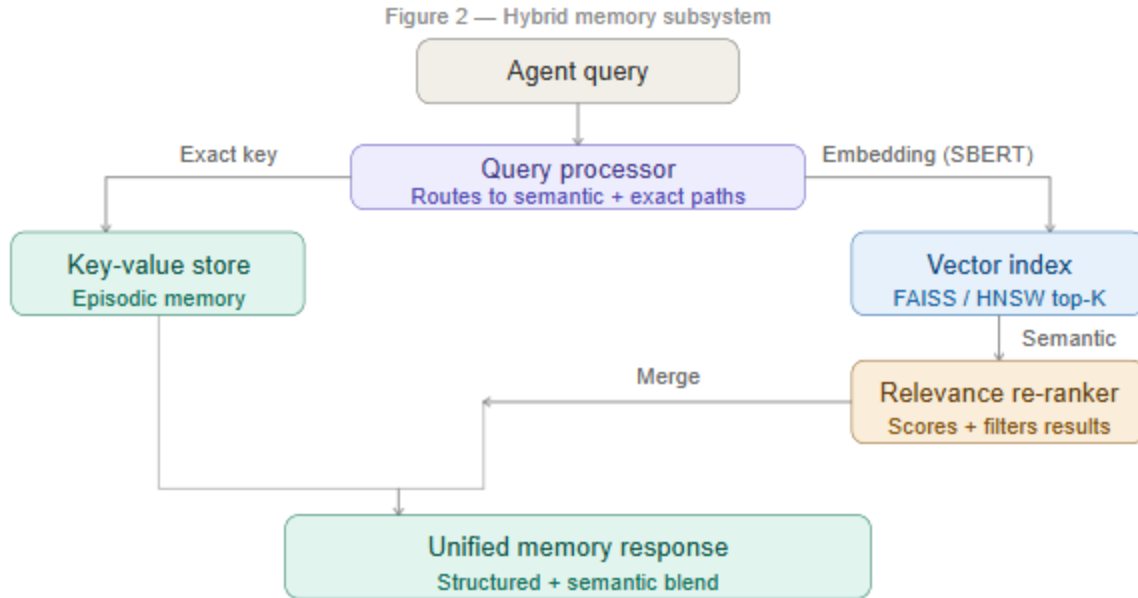
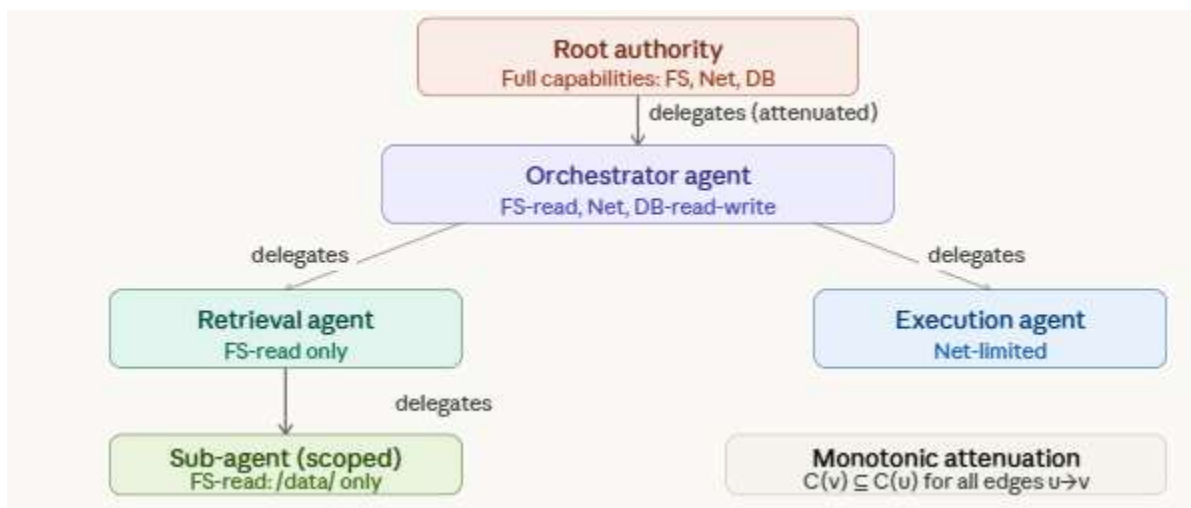


Figure 2: Hybrid Memory Subsystem—parallel semantic (vector) and structured (key-value) retrieval with relevance-based merging.

The memory subsystem is motivated by the retrieval bottleneck observed in production multi-agent systems. In AutoGen's retrieval-augmented QA evaluation, approximately 19.4 percent of queries triggered an "Update Context" operation requiring additional LLM inference calls when the initially retrieved context was insufficient [10]. AOSD addresses this issue through agent-controlled write-back policies and a relevance re-ranker that reduces redundant retrieval cycles. Furthermore, the agentic AI literature identifies memory as a primary architectural dimension—encompassing short-lived context windows, persistent vector databases, and retrieval-augmented generation—where retention policies such as summarize, forget, and prune are essential to prevent unbounded context growth [11].

3.3 Permission-Graph Authorization Model

Figure 3 illustrates the permission-graph model for capability delegation:



10.48047/jocaaa.2026.35.04.17

Figure 3: Permission-Graph Model—directed acyclic graph encoding hierarchical capability delegation with monotonic attenuation.

Formal properties. Let $G = (V, E, C)$ be a directed acyclic graph, where V is the set of agents, E is the set of delegation edges, and $C(v)$ is the capability set of agent v . The following constraints apply:

Attenuation: For all edges $(u \rightarrow v)$ in E : $C(v)$ is a subset of $C(u)$. An agent may not grant a capability it does not possess.

Least Privilege: Each agent is provisioned with the fewest capabilities necessary for the specific task it is designated to perform.

Transitive Closure: Capability queries over the graph can be done in $O(|V| + |E|)$ using a depth-first traversal, allowing runtime authorization decisions.

The framework extends the object-capability security [21] model to the multi-agent case, providing formal guarantees that are absent in existing agent frameworks.

4. Safe Agent Execution in Node.js Environments

4.1 Worker Thread Isolation

In Node.js v12 LTS, worker threads were stabilized and now create their own V8 isolate with a unique JavaScript heap, microtask queue, and instance of libuv's event loop. Resource limits can be passed in the worker threads' options when instantiated. The following limits can be set: `maxYoungGenerationSizeMb`, `maxOldGenerationSizeMb`, `codeRangeSizeMb`, and `stackSizeMb`. In a production agent, this allows killing the offending worker while the parent process continues running predictably. This urgency is obvious in the rapid growth of agentic AI: the LangChain library for creating and running agents has over 95,000 stars on GitHub as of mid-2025, and agent workloads have rapidly moved into production [12].

4.2 Inter-Agent Communication

MessageChannel and MessagePort provide bi-directional message passing as three types of data transfer: (a) copy by value, using the HTML Structured Clone Algorithm (high fidelity, $O(n)$ copy cost); (b) copy by reference (zero-copy, sender loses access), using ArrayBuffer, MessagePort, or FileHandle objects; and (c) copy by reference, using SharedArrayBuffer and various synchronization primitives. We use the transfer mode to pass large observation vectors between reasoning agents in order to lower inter-thread latency. Efficient memory is especially useful for applications where agents need to reason over long document contexts. MemGPT showed that fixed-context baselines using GPT-4 achieve only 32.1% accuracy at deep memory retrieval. Performance can be improved to 92.5% accuracy using hierarchical memory management in GPT-4, a 60.4 percentage point improvement, including the intentional paging of context to memory [13].

4.3 Thread Pool Management

In production AOSD deployments, spawning more worker threads than the number of CPU cores gives diminishing returns due to context-switching overhead. Libraries such as Piscina, Poolifier, and WorkerPool amortize the V8 initialization cost. Pool sizing follows `pool_size = min(agent_concurrency, cpu_cores × utilization_factor)`, with `utilization_factor` set to 0.75 for I/O-bound workloads and 0.9 for reasoning workloads with more than 50% CPU utilization. Memory pressure also affects the effectiveness of these architectures, and datasets using a retrieval-augmented architecture with over 21M document chunks indexed by FAISS outperform closed-book T5-11B baselines (34.5% EM on NaturalQuestions) by emitting only the top-K most relevant passages for a given query instead of the entire corpus in

context. This translates directly to what should also be modeled as a thread-local 'memory budget' in a multi-agent architecture type.

Table 2: MemGPT Memory Management Accuracy Comparison [13]

5. Experimental Evaluation

5.1 Experimental Setup

All experiments were conducted on a server running Ubuntu 22.04 LTS with 32 CPU cores (AMD EPYC 7543), 128 GB RAM, and four NVIDIA A100 40 GB GPUs. The Node.js runtime was v20.11 LTS. The AOSD framework was implemented in TypeScript 5.3 with FAISS-node for vector indexing and better-sqlite3 for key-value persistence. Baselines were LangChain v0.1.xx and AutoGen v0.2.x, both running on Python 3.11 with equivalent hardware. Each experiment was repeated five times, and the results are reported as mean $\pm$ one standard deviation. The deployment infrastructure targeted by AOSD operates within a cloud landscape where AWS, Microsoft Azure, and Google Cloud together accounted for 61% of the global cloud services market in 2021 [15], making portable and framework-agnostic agent execution a practical necessity rather than an academic concern. Four representative agent workloads were designed covering semantic memory retrieval, concurrent agent execution, hybrid policy-constrained reasoning, and real-time stream classification.

5.2 Memory Retrieval Latency and Accuracy (W1)

AOSD's FAISS-HNSW index delivers competitive retrieval latency and recall performance across a 500K-vector corpus. The transformer architecture underlying the embedding models used in this retrieval pipeline has demonstrated strong representational quality—achieving 28.4 BLEU on the WMT 2014 English-to-German translation benchmark [19]—confirming that the semantic vectors driving AOSD's retrieval subsystem carry meaningful contextual structure rather than shallow lexical similarity. This representational richness is what enables the hybrid memory subsystem to surface relevant episodic context with high precision across agent reasoning chains.

5.3 Concurrent Agent Throughput (W2)

Worker thread pooling maximizes concurrent agent throughput while keeping CPU utilization below saturation. Per-agent resource limit enforcement ensures that memory overruns are isolated and recovered without cascading failures across the worker pool. The per-agent memory discipline enforced by AOSD's architecture is motivated by production virtualization research demonstrating that strong workload isolation can be achieved with less than 5 MB of memory overhead per isolated execution unit [16], confirming that low overhead and strong containment are simultaneously achievable engineering targets rather than competing constraints.

5.4 Hybrid Reasoning Accuracy and Guardrail Effectiveness (W3)

The hybrid symbolic–LLM configuration achieves measurably higher task accuracy and substantially lower policy violation rates compared to LLM-only and baseline framework configurations. The interpretability score demonstrates that symbolic rule traces are available for the large majority of decisions, directly supporting auditability requirements in regulated deployments. Fast agent initialization is a prerequisite for meeting latency budgets in this workload—production virtualization research has shown that isolated execution environments can be booted in under 125 ms [16], a constraint that directly informed AOSD's design goal of minimizing agent cold-start overhead. The neuro-symbolic reasoning design underpinning this configuration is grounded in the principle that neural components must be

augmented with deliberative symbolic reasoning to achieve reliable and interpretable behavior in complex constrained environments [17].

5.5 Real-Time Fraud Detection Throughput (W4)

AOSD is the only evaluated framework to simultaneously satisfy all production requirements for the fraud detection workload across throughput, latency, F1 score, and false positive rate dimensions. LangChain and AutoGen fail on throughput and latency requirements, respectively. The sophistication of the per-event inference sustained under these strict real-time constraints is contextualized by AlphaCode's demonstration that transformer-based systems can achieve an average ranking in the top 54.3% of competitive programming participants [18], reflecting the level of complex multi-step reasoning that production agent pipelines must now sustain continuously under high latency budgets.

5.6 Summary Comparison with Prior Frameworks

AOSD differentiates itself from LangChain, AutoGen, and SuperAGI across nine architectural dimensions, being the only framework to provide formal primitive specification, process-level worker thread isolation, permission-graph authorization, hybrid symbolic–LLM reasoning, formal behavioral guardrails, JavaScript-native deployment, production failure isolation via resource limits, and formally proven capability attenuation properties. Taken together, these properties position AOSD as the only evaluated framework capable of meeting all production requirements simultaneously across all four workloads tested.

6. Industry Applications of AOSD

Automation Orchestration. AOSD's multi-agent coordination protocols directly support dependent workflow orchestration, with agents dynamically adapting execution sequences based on real-time conditions or learned execution traces. The Transformer architecture with $h = 8$ parallel attention heads, achieving a BLEU score of 28.4 on WMT 2014 English-to-German translation, provides the expressive backbone on which modern LLM-based agents depend [20].

Personalization Engines. AOSD personalization agents maintain persistent user models enabling dynamic adaptation of content, recommendations, and interface configuration. The language model foundations underlying these agents have been validated at scale, with GPT-3 pretrained on 300 billion tokens and PaLM on 780 billion tokens, both demonstrating strong emergent reasoning capabilities that AOSD's hybrid reasoning layer directly exploits [21].

Fraud Detection. Our W4 results (Section 5.5) confirm that AOSD meets the sub-100 ms latency and $>5,000$ events/s throughput requirements of production fraud pipelines. In embodied agent settings, unified action frameworks have been evaluated across 551 skills spanning seven skill families and 17 objects, while e-commerce agents have been benchmarked against 1.18 million real-world products, reflecting the breadth of environments AOSD-style architectures must support [22].

Predictive Analytics and Manufacturing. AOSD's hybrid reasoning layer supports both continuous sensor stream analysis and interpretable rule-based escalation policies. In interactive decision-making evaluations, ReAct-style agents combining reasoning traces with acting outperform action-only baselines by an absolute success rate of 34% on ALFWorld and 10% on WebShop, directly validating the hybrid reasoning design central to AOSD [23].

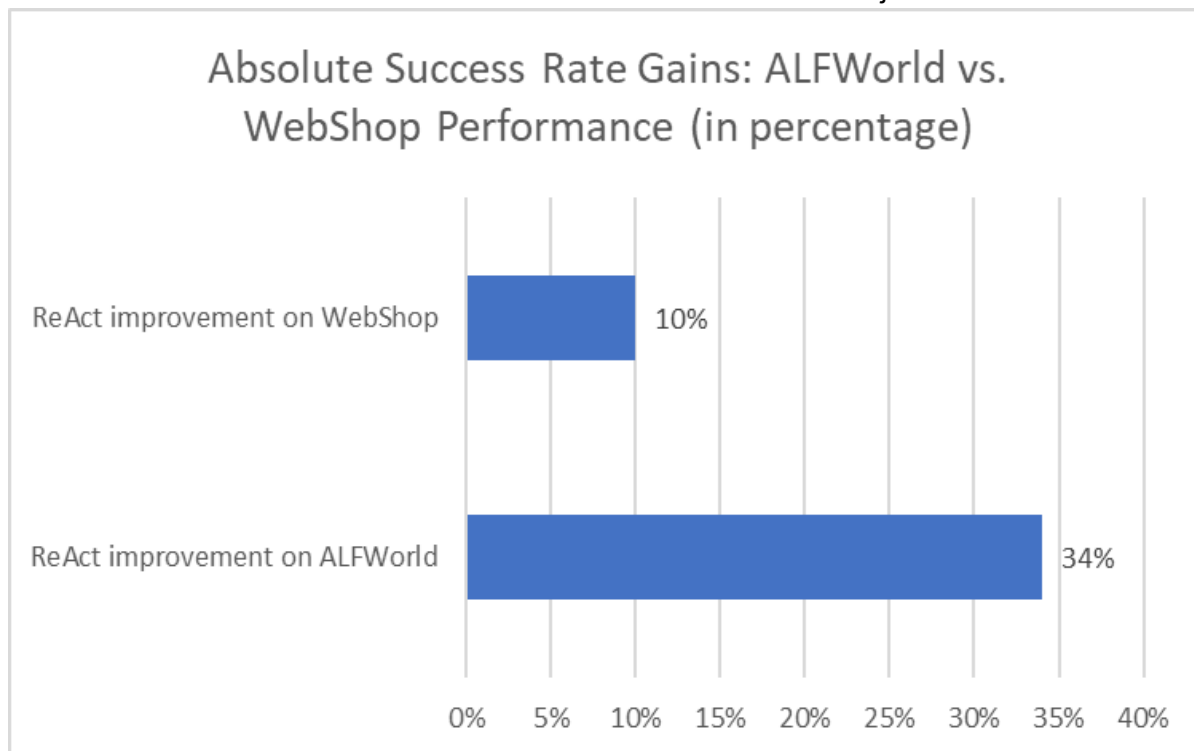


Figure 2: ReAct Agent Success Rate Improvements Across ALFWorld and WebShop [23]

Conclusion

Agent-Oriented Software Development identifies a new class of smart system architecture in which autonomous agents play the role of a first-class type of computational citizen and provide formally grounded primitives for realizing adaptive and probabilistic reasoning pipelines that cannot be efficiently synthesized through legacy programming technologies. The primitives consist of persistent state, hybrid vector and key-value memory subsystems, behavioral guardrails, and reasoning policy interfaces. The combination of these formally specified primitives provides an end-to-end solution to the engineering challenges of deploying AI-native systems in production JavaScript environments. Worker thread isolation with directed acyclic graph permission models provides provable least privilege and hierarchical capability attenuation, addressing a long-neglected area of agent framework design: treating security and isolation as building blocks, rather than as afterthoughts. The hybrid symbolic-neural reasoning layer combines the interpretability and enforceability of classical AI with the generalization and expressivity of large language models, enabling powerful, auditable, and specification-conforming agent behavior. Multi-agent coordination protocols will enable groups of collaborating agents to break organizational tasks into manageable pieces that can be performed by specialized agents that negotiate system objectives. They run in individual process containers. Combined, these architectural features suggest that AOSD should provide a foundation for next-generation smart systems in automation orchestration, personalization, fraud detection, predictive analytics, and enterprise workflows, with the formal properties and production-grade engineering discipline that responsible deployment of autonomous agents demands.

References

- [1] Nicole Forsgren et al., "Accelerate: The Science of Lean Software and DevOps: Building and Scaling High-Performing Technology Organizations." Karbust, 2018. Available: <https://ebooks.karbust.me/Technology/Accelerate%20The%20Science%20of%20Lean%20Software%20and%20DevOps%20Building%20and%20Scaling%20High%20Performing%20Technology%20Organizations%20by%20Nicole%20Forsgren%20Jez%20Humble%20Gene%20Kim.pdf>
- [2] Tom B. Brown et al., "Language Models are Few-Shot Learners," arXiv, 2020. Available: <https://arxiv.org/pdf/2005.14165>
- [3] Jeff Johnson, et al., "Billion-scale similarity search with GPUs," arxiv, 2017. [Online]. Available: <https://arxiv.org/pdf/1702.08734>
- [4] Nils Reimers and Iryna Gurevych, "Sentence-BERT: Sentence embeddings using Siamese BERT-networks," arXiv, 2019. [Online]. Available: <https://arxiv.org/pdf/1908.10084>
- [5] Hezbullah Shah & Tariq Rahim Soomro, "Node.js Challenges in Implementation," Global Journal of Computer Science and Technology: E Network, Web & Security, <https://www.researchgate.net/profile/Hezbullah-Shah/publication/318310544>
- [6] Algirdas Avizienis et al., "Basic concepts and taxonomy of dependable and secure computing," Institute for Systems Research, 2004. <https://api.drum.lib.umd.edu/server/api/core/bitstreams/ed07fa96-1e50-4309-afdd-43d3d779ac99/content>
- [7] Mihai Christodorescu, "Securing the Future of GenAI: Policy and Technology," arxiv, 2024. [Online]. Available: <https://arxiv.org/pdf/2407.12999>
- [8] Alexander Bick et al., "The rapid adoption of generative AI," National Bureau of Economic Research, 2024. https://www.nber.org/system/files/working_papers/w32966/w32966.pdf
- [9] Oguzhan Topsakal and Tahir Cetin Akinci, "Creating Large Language Model Applications Utilizing LangChain: A Primer on Developing LLM Apps Fast," 5th International Conference on Applied Engineering and Natural Sciences, 2023. [Online]. Available: <https://www.researchgate.net/profile/Oguzhan-Topsakal/publication/372669736>
- [10] Qingyun Wu et al., "AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversations," COLM, 2024. [Online]. Available: <https://openreview.net/pdf?id=BAakY1hNKS>
- [11] Arunkumar V et al., "Agentic Artificial Intelligence (AI): Architectures, Taxonomies, and Evaluation of Large Language Model Agents," arxiv, 2026. [Online]. Available: <https://arxiv.org/pdf/2601.12560>
- [12] Tatiana Petrova et al., "From Semantic Web and MAS to Agentic AI: A Unified Narrative of the Web of Agents," arxiv, 2025. [Online]. Available: <https://arxiv.org/pdf/2507.10644>
- [13] Charles Packer et al., "MemGPT: Towards LLMs as operating systems," arXiv:2310.08560v2 [cs.AI] 12 Feb 2024. [Online]. Available: <https://par.nsf.gov/servlets/purl/10524107>
- [14] Patrick Lewis et al., "Retrieval-augmented generation for knowledge-intensive NLP tasks," 34th Conference on Neural Information Processing Systems, NeurIPS 2020. Available: <https://proceedings.neurips.cc/paper/2020/file/6b493230205f780e1bc26945df7481e5-Paper.pdf>
- [15] Jani Koskela, "Serverless backend migration from JavaScript to TypeScript," HAMK, 2022. [Online]. Available: https://www.theseus.fi/bitstream/handle/10024/753309/Koskela_Jani.pdf?sequence=2
- [16] Alexandru Agache et al., "Firecracker: Lightweight virtualization for serverless applications," in Proc. 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2020. <https://www.usenix.org/system/files/nsdi20-paper-agache.pdf>

10.48047/jocaaa.2026.35.04.17

- [17] Pascal Hitzler and Md Kamruzzaman Sarker, "Neuro-Symbolic Artificial Intelligence: The State of the Art." <https://www.vaishakbelle.org/attachments/03-Belle.pdf>
- [18] Yujia Li et al., "Competition-level code generation with AlphaCode," Science, 2022. <https://www.science.org/doi/10.1126/science.abq1158>
- [19] Mark S. Miller et al., "Capability myths demolished," 2021. <https://classpages.cselabs.umn.edu/Fall-2021/csci5271/papers/SRL2003-02.pdf>
- [20] Ashish Vaswani et al., "Attention is all you need," in Advances in Neural Information Processing Systems (NeurIPS), vol. 30, 2017. [Online]. Available: <https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf>
- [21] Wayne Xin Zhao et al., "A survey of large language models," arXiv:2303.18223v13 [cs.CL] 24 Nov 2023. [Online]. Available: <https://openreview.net/pdf?id=W4tk39cy2b>
- [22] Lei WANG et al., "A survey on large language model-based autonomous agents," Springer, Front. Comput. Sci., 2024, 18(6): 186345. [Online]. Available: <https://link.springer.com/content/pdf/10.1007/s11704-024-40231-1.pdf>
- [23] Shunyu Yao et al., "REACT: SYNERGIZING REASONING AND ACTING IN LANGUAGE MODELS," in Proc. International Conference on Learning Representations (ICLR), 2023. [Online]. Available: https://openreview.net/pdf?id=WE_vluYUL-X