American Journal of Interdisciplinary Research and Innovation (AJIRI)

ISSN: 2833-2237 (ONLINE)

VOLUME 5 ISSUE 3 (2026)

PUBLISHED BY

E-PALLI PUBLISHERS, DELAWARE, USA

American Journal of Interdisciplinary Research and Innovation (AJIRI)

Volume 5 Issue 3, Year 2026
ISSN: 2833-2237 (Online)
DOI: https://doi.org/10.54536/ajiri.v5i3.8311
https://journals.e-palli.com/home/index.php/ajiri

Architectural Foundations for Latency-Aware Scalability in AI-Enhanced Enterprise Systems

Daniil Sergeevich MARTYNOV[1*]

Article Information

Received: May 28, 2026

Accepted: August 04, 2026

Published: September 16, 2026

Keywords

AI-Enhanced Systems, Enterprise Scalability, Latency-Aware Architecture, Zero-Trust Security

ABSTRACT

Enterprise software now routes access-control decisions and code-level performance tuning through machine learning components sitting directly inside the request path. The literature has not kept pace. Distributed-systems scholarship on scalability and the newer body of work on AI-driven architecture have grown along separate tracks, and few sources spell out how the two concerns should share a single latency budget. Point solutions attack individual pieces of the problem, and the problem as a whole goes unaddressed. Static policy engines evaluate rules they did not generate from behavioral evidence. Canary-analysis platforms decide whether to promote a change without asking what that change should contain. Compiler-level optimizers raise throughput without governing how a proposed transformation earns the right to reach production traffic. This paper examines two production-documented architectures that fold detection, decision, verification, and staged deployment into one closed loop, one built for continuous security enforcement and the other for continuous performance optimization, and asks whether the integration amounts to a genuine architectural advance over the fragmented tooling it responds to. Drawing on architectural decompositions, three enterprise deployment case studies, and a controlled ablation experiment, the analysis identifies a governance pattern shared by both systems: every learned proposal answers to deterministic verification and staged production exposure before it reaches a live user. The pattern is then checked against established findings on tail latency, consistency-availability trade-offs, and microservice decomposition overhead.

INTRODUCTION

Enterprise request paths that once carried only deterministic business logic now carry inference calls, behavioral scoring routines, and automated code transformations. Each competes for the same millisecond-scale latency budget that end users simply experience as the system's responsiveness. Architectural decisions once confined to network topology, database sharding, and cache placement must now also account for a model's inference latency, memory footprint, and confidence threshold, and for how those quantities interact with service level objectives the surrounding distributed system was never designed to meet. Dean and Barroso (2013) showed that modest per-component latency variability compounds across a fan-out of dependent services into tail latencies that dominate what users perceive. Their finding predates the routine embedding of machine learning inference into production request paths, so it says nothing about the added variability model serving contributes. Armbrust *et al.* (2010) showed that elastic provisioning changes the economics of scale but does not eliminate the queueing dynamics that turn added load into added delay. Later work on prediction-serving infrastructure found that inference workloads inherit these dynamics and add their own: batching delay, cold-start overhead, memory pressure from model weights (Crankshaw *et al.*, 2017; Gujarati *et al.*, 2020). A poorly budgeted inference call on a shared code path does more than trigger an isolated timeout: it degrades the tail latency of every request that crosses it, and the resulting violations propagate outward to contractual penalties, customer churn, and the cascading failures that distributed-systems engineers have spent decades learning to contain through mechanisms far less intelligent than a machine-learning model under load, and far more predictable.

Google's BeyondCorp initiative demonstrated as early as 2014 that access decisions could rest on device and user context rather than network location (Ward & Beyer, 2014). As documented, though, the architecture evaluates a largely static set of trust signals at the access-proxy layer. It does not generate new enforcement policies from continuously observed behavioral drift; a human security team still writes the rules the proxy applies. Service-mesh implementations such as Istio solve an adjacent problem, injecting sidecar proxies that enforce mutual TLS, routing, and authorization at every hop of a microservices deployment (Li *et al.*, 2019). Policy engines such as the Open Policy Agent supply a general-purpose evaluation layer that decouples policy decisions from the application code obeying them (Ahmed, 2020). Deciding what a policy should actually say, however, is left to whatever external system supplies the input data and the rule set; neither framework claims that job. Netflix and Google built the Kayenta platform to automate the

[1] Chief Software Solutions Architect, Digital Performance Labs LLC, Kazan, Russia
[*] Corresponding author's e-mail: daniil.sergeevich.m@gmail.com

statistical comparison between a canary population and a baseline, replacing hours of manual graph inspection with a promote-or-rollback judgment (Graff & Sanden, 2018). Kayenta's scope starts only once a candidate change already exists. Where that change came from is outside its concern. The performance side of the enterprise stack shows a comparable split. TVM optimizes the mapping of a fixed computational graph onto target hardware (Chen *et al.*, 2018). BOLT, a post-link binary optimizer, extracts additional throughput from an already-compiled binary using collected execution profiles (Panchenko *et al.*, 2018). Neither reaches the source-level refactoring, memory tuning, and staged rollout that a continuously self-improving codebase actually needs. Four stages make up an AI-enhanced control loop: behavioral detection, policy or patch synthesis, correctness verification, and staged exposure. Dedicated tools cover each stage reasonably well on its own. An architecture that integrates all four across both the security and performance domains of an enterprise system is still comparatively rare.

Two architecturally distinct systems documented by Dazhyma Oiun occupy this rare position, and they form the basis of the analysis developed here. The first, an AI-Driven Adaptive Security Layer described in Oiun's (2026a) monograph, embeds behavioral threat scoring, automated policy generation, and dynamic traffic re-routing inside a single control plane, closing the gap between behavioral detection and policy synthesis that BeyondCorp, Istio, and the Open Policy Agent leave to human authors outside the system. The second, a Neural Code Optimization Engine documented in the same monograph and evaluated through a controlled ablation experiment in an associated conference contribution (Oiun, 2026b, 2026c), closes the matching gap on the performance side, connecting large-language-model-assisted static analysis and hardware-level dynamic profiling, stages TVM and BOLT each address only partially, to an automated refactoring and staged-rollout mechanism that resembles Kayenta's promotion judgment in spirit but targets the correctness and performance of a machine-generated code transformation rather than an arbitrary deployment. The two architectures reward examination together because Oiun's design builds explicitly on several precedents already discussed here, the checker-synthesis method of Yang *et al.* (2025) among them, along with the staged-exposure discipline documented in the site-reliability-engineering literature. What Oiun adds is the assembly itself: one governed loop, not separate pieces left for an enterprise to integrate on its own.

MATERIALS AND METHODS

The analysis rests on a structured comparative synthesis of documentary sources, built from three categories of material, each assembled through its own search and screening procedure.

Candidate case architectures came from a search of production engineering literature, industry monographs, and conference proceedings on enterprise security and performance-optimization systems published between 2014 and 2026. Search terms combined "zero trust," "behavioral detection," "policy orchestration," "code optimization," and "canary deployment." Three criteria decided which systems made the cut. Documentation had to be detailed enough to show how responsibility is split across the four pipeline stages introduced above. At least one production deployment needed a before-and-after quantitative evaluation. And the architecture had to cover both the decision-generation and decision-verification sides of the pipeline, not just one. Two systems cleared all three bars: an AI-Driven Adaptive Security Layer and a Neural Code Optimization Engine, documented by Dazhyma Oiun (2026a, 2026b, 2026c). These became the primary case architectures. Six systems cleared only one or two criteria, BeyondCorp, Istio, the Open Policy Agent, Kayenta, TVM, and BOLT, and were kept instead as comparison systems spanning the range of partial coverage the argument needs (Ahmed, 2020; Chen *et al.*, 2018; Graff & Sanden, 2018; Li *et al.*, 2019; Panchenko *et al.*, 2018; Ward & Beyer, 2014).

The third category is theoretical and empirical literature, chosen not for a comprehensive review of distributed-systems theory but because each source gives a specific constraint or baseline against which a claim in the case architectures could be tested. Tail-latency amplification (Dean & Barroso, 2013). Elastic provisioning and prediction serving (Armbrust *et al.*, 2010; Crankshaw *et al.*, 2017; Gujarati *et al.*, 2020). The CAP theorem (Gilbert & Lynch, 2002). Edge-computing constraints (Shi *et al.*, 2016). Microservice decomposition overhead (Blinowski *et al.*, 2022; Faustino *et al.*, 2022). Large-language-model serving bottlenecks (Kwon *et al.*, 2023). Yang *et al.*'s (2025) checker-synthesis method rounds out the set, included because the case architectures build on it directly.

Three analytical passes then ran across the full corpus. The first breaks each system into its four pipeline stages and separates request-path components from asynchronous ones. The second pulls out every reported quantitative claim, latency percentiles, throughput, memory footprint, precision, false-positive rates, and traces each back to the design decision responsible for it. The third checks the resulting figures against the literature identified above, sorting each design choice into one of three bins: consistent with an established result, qualified by it, or in tension with it.

RESULTS AND DISCUSSION

Architectural decomposition of the AI-Driven Adaptive Security Layer

The architecture that Oiun documents assigns four named subsystems to a control plane that governs the request path of a distributed application without sitting directly inside it (Oiun, 2026a). The Behavioral Threat Signature Engine continuously analyzes telemetry streams to generate reusable detection signatures. The Machine Learning Anomaly Detection Engine computes

real-time risk scores for entities and sessions, combining density-based unsupervised clustering, graph-based lateral-movement analysis built on graph neural networks with attention weighting over neighboring nodes, and sequence models that evaluate temporal event ordering. The Auto-Policy Enforcement Orchestrator turns detection outputs into versioned, testable policy artifacts, structured rule sets rather than free-form configuration text. The Zero-Trust Re-Routing Engine carries out the resulting decisions, diverting suspicious traffic away from production resources toward sandboxed inspection paths. What sets Oiun's decomposition apart from a conventional intrusion-detection pipeline, and from the BeyondCorp and service-mesh approaches discussed above, is a line the architecture draws inside itself. Some components must answer within the latency budget of the live request, the sidecar proxies enforcing authorization decisions at the service-mesh layer among them. Others run on a longer telemetry-processing cycle, like the retraining pipeline that periodically refreshes behavioral baselines. The split matters in practice: it lets Oiun's design keep expensive inference off the hot path for most requests, escalating only the sessions a lightweight initial score flags as ambiguous to the heavier graph-neural-network analysis, a cascading pattern that mirrors the accuracy-latency trade-off documented in model-serving research (Crankshaw *et al.*, 2017).

Several quantitative targets Oiun reports show how tightly the architecture couples security effectiveness to latency and resource discipline (Oiun, 2026a). Telemetry collection runs on kernel-level instrumentation through the extended Berkeley Packet Filter. Oiun frames it as a way to observe system calls and network flows with overhead low enough not to perturb the request path it protects, a choice that echoes the broader profiling literature's preference for sampling-based measurement over exhaustive instrumentation wherever production suitability matters. Policy updates from the orchestrator reach distributed enforcement points within seconds of a detection event, a pace that pushes the consistency model toward the eventual-consistency end of the spectrum Gilbert and Lynch (2002) formalized: demanding strong consistency across every sidecar proxy in a large deployment would add coordination latency the security use case cannot absorb. Oiun's design reserves stronger guarantees, quorum-based confirmation, for a narrower category, emergency threat blocks, where a brief window of inconsistent enforcement costs less than the coordination delay would. At the edge, working under the computational and connectivity limits Shi *et al.* (2016) identified as typical of such deployments, Oiun reports pruning combined with eight-bit quantization removing roughly 90% of model parameters at a cost of under 1% accuracy. The figure matches the broader model-compression literature, and it decides something practical: whether behavioral scoring runs locally on a resource-constrained device or needs a round trip to a centralized inference service, the very network latency the edge deployment exists to avoid. Automated enforcement targets a false-positive rate of 1-5% for actions that block traffic outright. Oiun set the threshold there because higher rates produced the alert fatigue that erodes analyst trust no matter how accurate the underlying detection is.

Architectural decomposition of the Neural Code Optimization Engine

Oiun's second architecture runs on a structurally similar split between proposal generation and admission control. The Neural Code Optimization Engine takes shape as a four-stage pipeline, candidate discovery, hotspot attribution, transformation synthesis, admission control (Oiun, 2026b). Neural components generate hypotheses and reusable static analyzers; deterministic mechanisms keep the authority to decide which hypotheses actually reach production, a division of labor that tracks closely with the boundary between behavioral scoring and policy enforcement in the security architecture above. Candidate discovery draws directly on the checker-synthesis method of Yang *et al.* (2025). The method trains a model on paired before-and-after code examples from historical defect fixes and outputs an executable static analyzer, not a natural-language suggestion. Oiun credits it with cutting the engineering effort of building a new detector by an order of magnitude compared with hand-coded rule sets (Oiun, 2026c). The companion monograph expands the four-stage picture into six named subsystems. An LLM-based static analysis engine built on the checker-synthesis principle. A dynamic profiling and heatmap visualization component that maps resource consumption onto source code through hierarchical telemetry aggregation. An automated refactoring engine applying transformations as typed, semantics-preserving recipes built at the abstract-syntax-tree level, producing diffs that stay reviewable and reversible. A machine-learning-based memory and garbage-collection tuning module. A safety and validation subsystem combining static re-analysis with comprehensive test execution. And a controlled rollout component managing canary traffic splitting and automated rollback, narrower in scope than Kayenta's statistical judgment on arbitrary production changes but structurally comparable to it (Graff & Sanden, 2018). Every proposed transformation passes through this safety and validation subsystem before it sees production traffic, answering a concern raised in the automated-program-repair literature: a model can produce a patch that satisfies a constrained test suite while still being wrong under broader program invariants, exactly the failure mode that motivated the checker-synthesis approach behind Oiun's static analysis component (Yang *et al.*, 2025).

The controlled ablation Oiun reports gives the clearest available evidence for the causal-attribution capability the broader engine is meant to exercise continuously in production (Oiun, 2026b). Four variants of a single workload isolate a hot path that performs denylist membership testing while assembling an output artifact. Variant A is the unmodified baseline: membership

through a linear list scan, output construction through repeated string concatenation. Variant B replaces only the membership structure, swapping in a hash set. Variant C replaces only the output construction, using list accumulation followed by a single join. Variant D combines both changes. Variant B alone drives a median latency reduction of roughly two orders of magnitude against the baseline, which pins hot-path membership cost as the dominant factor behind the measured difference. Variant C on its own barely moves latency, but it measurably raises traced peak memory. Variant D inherits that memory cost and still posts the best latency figures of the four.

Beyond the controlled ablation, Oiun reports three production deployments that extend the same architectural pattern to enterprise scale (Oiun, 2026c). The first is a financial transaction processing system handling more than 50,000 requests per second, under a service level agreement requiring 95% of requests to complete within 200 ms. Static analysis flagged 127 candidate anti-patterns; manual review confirmed 93 as genuine optimization opportunities. The applied fixes replaced sequential single-row database queries with bulk operations expressed through WHERE IN clauses and batch inserts, cutting database round trips by 85%. Separately, a machine-learning model trained on garbage-collection logs recommended a 40% increase in young-generation heap size together with a migration from the CMS collector to G1GC, bringing median garbage-collection pause time down from 180 ms to 68 ms. Together, these interventions took median request latency from 145 ms to 78 ms, and the 95th percentile from 310 ms to 165 ms. Sustainable throughput rose from 52,000 to 71,000 requests per second, and per-instance memory footprint fell 18%. The second deployment is a 47-service e-commerce platform that had been suffering checkout timeouts during peak shopping events. Its product catalog service added a GraphQL interface, letting client applications request only the fields they needed instead of complete product objects, added database read replicas, and cached query results with invalidation keyed to product update timestamps; average service latency dropped from 180 ms to 45 ms. The inventory service swapped a thrash-prone cache for Redis-backed negative caching and circuit breakers, cutting backend calls by 91%. The recommendation engine moved its machine-learning inference out of the synchronous request path entirely, serving pre-computed results with a rule-based fallback, and took 99th-percentile latency from 450 ms to 85 ms. Averaged across the platform, end-to-end latency fell from 520 ms to 185 ms, checkout completion climbed from 87% to 96%, and the changes were tied to an estimated $2.3 million in additional annual revenue alongside a 31% cut in infrastructure cost, despite a 15% rise in peak traffic. The third case is a real-time analytics pipeline built on Kafka for ingestion and Flink for windowed stream processing. Replacing full JSON deserialization with schema-based protocol-buffer parsing, extracting only the fields each consumer actually needed, cut per-message processing latency from 35 ms to 15 ms. Replacing nested iterations with hash-based lookups in Flink's stateful operators brought algorithmic complexity down from quadratic to linear and cut memory allocations by 67%. Combined with machine-learning-guided batch-size tuning for the downstream time-series database, these changes brought end-to-end pipeline latency down from 850 ms to 320 ms, while raising the maximum sustainable ingestion rate from 2.5 million to 3.8 million events per second. Aggregated across all three deployments, Oiun reports a median latency reduction of 57%, a median throughput increase of 37%, and infrastructure cost reductions ranging from 18% to 31%, figures the monograph frames explicitly as case-study evidence.

Discussion

Both of Oiun's architectures share a governance pattern. A learned component drafts a proposal; a separate deterministic mechanism approves or rejects it. The pattern converges with an older insight from large-scale systems research: bounding and managing variability through explicit mechanisms tends to succeed where trying to eliminate it outright does not. Dean and Barroso (2013) argued that tail-tolerant techniques, hedged requests and cross-request adaptation among them, work precisely because they accept that some fraction of components will behave unpredictably, and design the surrounding system to absorb that unpredictability rather than fight it (Dean & Barroso, 2013). Oiun's canary-based admission control, applied to both policy updates and code transformations, extends this same logic from raw request latency to the correctness and performance impact of an automated change. A small traffic fraction gets exposed to the change; its observed behavior becomes the basis for expanding or rolling back that exposure. No pre-deployment correctness guarantee could substitute, because the underlying transformation cannot supply one (Oiun, 2026a, 2026c).

The case for treating Oiun's two architectures as an advance, rather than a recombination of existing parts, rests on where the integration removes a manual handoff that the point solutions surveyed in the introduction otherwise leave standing. BeyondCorp's access proxy and the Open Policy Agent's rule evaluation both assume a human, or a separate governance process, has already decided what the policy should say (Ward & Beyer, 2014; Ahmed, 2020). Oiun's security architecture drops that assumption. It feeds the same behavioral evidence that triggers a risk score directly into policy generation, folding detection and authorship into one auditable pipeline. Kayenta answers a narrower question extremely well: is an already-proposed change safe to expand (Graff & Sanden, 2018)? It has no opinion on where that change came from. Oiun's optimization engine supplies exactly the missing half, generating the candidate transformation from static and dynamic evidence before handing it to a

staged-rollout mechanism that plays a role comparable to Kayenta's own judgment stage.

Accepting eventual consistency for most policy propagation, and reserving stronger guarantees for a narrow category of emergency actions, applies directly the constraint Gilbert and Lynch (2002) proved unavoidable in any distributed service that must stay both available and partition-tolerant. The theorem itself supplies no rule for which side of that trade-off a given action should fall on. Oiun's architecture adds exactly that: a per-policy criterion, weighing the operational cost of a brief window of inconsistent enforcement against the coordination latency that stronger consistency would add across every enforcement point in a large deployment.

The independent literature's strongest qualification of Oiun's architecture concerns latency cost: any AI-driven control plane necessarily adds decomposition, and decomposition has a price. Blinowski *et al.* (2022) found a monolithic implementation outperforming its microservice-based counterpart on a single machine. Faustino *et al.* (2022) measured latency increases from 201% to nearly 7,000% when migrating a monolith stepwise through an intermediate modular stage into a full microservice architecture. Both results sit in direct tension with any architecture that adds further service hops, sidecar proxies, and inference calls onto a system that is already decomposed. Oiun's security architecture answers this tension by budgeting explicitly for the latency its own components add. The cascading design, a lightweight score filtering most traffic while only ambiguous cases escalate to expensive graph-based or generative analysis, is exactly the mechanism model-serving research recommends for controlling inference latency in a request path (Crankshaw *et al.*, 2017; Gujarati *et al.*, 2020). The edge-deployment figures for model compression point the same way: inference cost is treated here as a resource to budget and actively shrink, not one to ignore. The optimization engine's numbers are less clean. Its case-study figures describe latency reductions achieved by removing conventional bottlenecks, inefficient database queries, garbage-collector misconfiguration, and they do not isolate the net latency contribution of the engine's own inference calls against a baseline that lacks the engine entirely. How much of the reported 46% to 64% improvement would survive a comparison baseline that had already stripped out those unrelated bottlenecks is a question the case studies do not answer.

The governance pattern examined here carries an organizational implication too, one neither of Oiun's monographs states outright but one that follows naturally from the architecture as described. Treat a machine-generated code transformation or security policy as a first-class change, subject to the same canary discipline as ordinary application code, and the teams reviewing such changes need to become fluent in reading model-generated diffs, interpreting confidence scores, and setting the traffic-fraction and rollback thresholds that decide how fast a bad proposal gets caught. The requirement echoes a broader concern from the site-reliability literature: automation without proportional investment in operator understanding tends to shift failure modes rather than remove them, trading manual configuration errors for a comparable rate of misconfigured automation. The case-study figures examined here come from deployments already staffed by teams experienced enough to build Oiun's underlying engines. They say little about how the same governance pattern would fare in an organization without that accumulated operational maturity.

CONCLUSION

The comparison developed across this article points to a coherent architectural pattern: learned proposal generation paired with deterministic admission control, staged production exposure, and explicit resource budgeting for inference cost. Dazhyma Oiun has applied that pattern independently but consistently across the security and performance domains (Oiun, 2026a, 2026b, 2026c). Measured against the fragmented landscape surveyed at the outset, BeyondCorp's static trust evaluation, service-mesh and policy-agent frameworks that enforce but do not author policy, Kayenta's promotion judgment with no opinion on what is being promoted, compiler-level optimizers that stop at the boundary of the running system, Oiun's two architectures stand out for one thing: they close the loop from behavioral or performance evidence through to governed production exposure inside a single accountable pipeline. For enterprise architects weighing whether to embed a similar AI-driven control loop into their own systems, the central design question is not whether the underlying metric can improve. Both architectures already demonstrate substantial, documented improvement in isolation. The question is whether the surrounding admission-control and rollback infrastructure is mature enough to make that improvement trustworthy under the same latency and correctness constraints governing the rest of the system. The three enterprise case studies and the controlled ablation examined here move this question from a matter of principle to a matter of documented engineering practice. Independent replication of the figures remains the next necessary step for the field.

REFERENCES

Ahmed, M. (2020). Introducing policy as code: The Open Policy Agent (OPA). *Cloud Native Computing Foundation Blog*.

Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., & Zaharia, M. (2010). A view of cloud computing. *Communications of the ACM, 53*(4), 50–58. https://doi.org/10.1145/1721654.1721672

Blinowski, G., Ojdowska, A., & Przybylek, A. (2022). Monolithic vs. microservice architecture: A performance and scalability evaluation. *IEEE Access, 10*, 20357–20374. https://doi.org/10.1109/ACCESS.2022.3152803

Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Shen, H., Cowan, M., Wang, L., Hu, Y., Ceze, L., Guestrin, C., & Krishnamurthy, A. (2018). TVM: An automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation* (OSDI 18) (pp. 578–594).

Crankshaw, D., Wang, X., Zhou, G., Franklin, M. J., Gonzalez, J. E., & Stoica, I. (2017). Clipper: A low-latency online prediction serving system. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation* (NSDI 17) (pp. 613–627).

Dean, J., & Barroso, L. A. (2013). The tail at scale. *Communications of the ACM, 56*(2), 74–80. https://doi.org/10.1145/2408776.2408794

Faustino, D., Goncalves, N., Portela, M., & Silva, A. R. (2022). *Stepwise migration of a monolith to a microservices architecture: Performance and migration effort evaluation*. arXiv. https://doi.org/10.48550/arXiv.2201.07226

Gilbert, S., & Lynch, N. (2002). Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News, 33*(2), 51–59. https://doi.org/10.1145/564585.564601

Graff, M., & Sanden, C. (2018). Automated canary analysis at Netflix with Kayenta. *Netflix Technology Blog*.

Gujarati, A., Karimi, R., Alzayat, S., Hao, W., Kaufmann, A., Vigfusson, Y., & Mace, J. (2020). Serving DNNs like clockwork: Performance predictability from the bottom up. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation* (OSDI 20) (pp. 443–462).

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J. E., Zhang, H., & Stoica, I. (2023). Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles* (SOSP '23) (pp. 611–626). https://doi.org/10.1145/3600006.3613165

Li, W., Lemieux, Y., Gao, J., Zhao, Z., & Han, Y. (2019). Service mesh: Challenges, state of the art, and future research opportunities. In *Proceedings of the 2019 IEEE International Conference on Service-Oriented System Engineering* (SOSE) (pp. 122–125). https://doi.org/10.1109/SOSE.2019.00026

Oiun, D. (2026a). AI-enhanced software architecture for modern applications. *LAP LAMBERT Academic Publishing*.

Oiun, D. (2026b). Neural code optimization as a new paradigm in software engineering. In *Proceedings of the XXII International Scientific and Practical Conference* "Topical Issues of Modern Scientific Research."

Oiun, D. (2026c). Neural optimization and performance engineering of enterprise software. *LAP LAMBERT Academic Publishing*.

Panchenko, M., Auler, R., Nell, B., & Ottoni, G. (2018). *BOLT: A practical binary optimizer for data centers and beyond*. arXiv. https://doi.org/10.48550/arXiv.1807.06735

Shi, W., Cao, J., Zhang, Q., Li, Y., & Xu, L. (2016). Edge computing: Vision and challenges. *IEEE Internet of Things Journal, 3*(5), 637–646. https://doi.org/10.1109/JIOT.2016.2579198

Ward, R., & Beyer, B. (2014). BeyondCorp: A new approach to enterprise security. *Login: The Usenix Magazine, 39*(6), 6–11.

Yang, C., Zhao, Z., Xie, Z., Li, H., & Zhang, L. (2025). *KNighter: Transforming static analysis with LLM-synthesized checkers*. arXiv. https://doi.org/10.48550/arXiv.2503.09002