Why the Agentix Operating Engine Matters: A Monolithic vs Microservice Comparison of Agentic AI on Heterogeneous SoCs

Siavash Alamouti, Fay Arjomandi, Michel Burger, Jeremy Hsu
mimik Technology Inc.

Abstract

Agentic AI workloads are moving from research-scale prototypes toward production deployments where many agents run concurrently on heterogeneous SoCs that combine CPU, GPU, and NPU compute. The architectural choices that matter at production scale are not just about silicon; they are also about the agentic operating environment that organizes how agents are deployed, updated, secured, and coordinated across resources and across devices. This paper compares two deployment models on the same heterogeneous SoC platforms: a monolithic application binary that bundles multiple agents into a single process (the pattern most agentic stacks ship today), and a microservice substrate with dynamic resource discovery, service routing, and zero-trust security across devices. We find that monolithic deployment requires substantially more memory than microservice deployment for the same workload (roughly an order of magnitude on a multi-tenant healthcare case at ward scale, with the monolithic footprint exceeding even a 128 GB unified memory ceiling), and that cluster scaling under the microservice substrate delivers substantially more effective capacity than equivalent monolithic clusters, because the monolithic footprint is memory-capped per device and cannot pool burst capacity across devices. Combined with a set of operational properties that microservice deployment provides as architectural defaults (per-agent updates, fault isolation, zero-trust security, service discovery, cluster elasticity), the result is that the agentic operating environment is not optional infrastructure but a multiplier on the silicon investment in production agentic AI deployments.

1 Introduction

Agentic AI workloads on heterogeneous SoCs combine three classes of compute (CPU, GPU, NPU) with three workload tiers (TinyML for continuous sensing, sLLM for reasoning, LLM for conversational fallback). These workloads run as multiple coordinating agents, each with its own state, models, and lifecycle, on hardware that ranges from smartphone-class devices to workstation-class platforms with unified memory in the 64 to 128 GB range. The architectural question is not just which silicon to use but how the silicon is exercised by the agentic operating environment that organizes the agents.

This paper examines that question at the level of the software engine that runs the agents. Most agentic stacks shipping today are monolithic application binaries that bundle multiple agents into a single process. Operator placement is decided at build time, agent endpoints are wired in code or configuration, and there is no mechanism for runtime discovery, placement, or relocation. An alternative is a microservice substrate where each agent is an independent service, model artifacts are deduplicated across agents, agents are discovered and routed at runtime, and the substrate can place work dynamically across resources and across devices in a cluster. The substrate model is

described by prior work as the Agentix-Native[1] architecture [2] and is implemented by mimik's Agentix Operating Engine (mimOE).

The scope of agentic AI considered here is the cognitive layer above any safety-critical real-time control stack. We address workloads with latency budgets of seconds to minutes (perception, reasoning, conversation, monitoring, coordination) and not workloads with sub-10-millisecond hard real-time requirements (control loops, safety interlocks, sensor fusion for collision avoidance). The latter requires a different runtime architecture (real-time OS with certified scheduling guarantees) and is outside the scope of this analysis.

We compare monolithic and microservice deployment on two architectural classes of heterogeneous SoCs: a CPU-centric class with a strong CPU alongside a capable GPU and NPU and 128 GB unified memory (we use the AMD X100 as the named example), and a GPU-centric class with a dominant GPU, a moderate CPU, and 128 GB unified memory (we use an NVIDIA Jetson Thor-class part as the named example) [1]. We evaluate both deployment models against five use cases: a robot perception pipeline, an industrial sensing deployment, single-patient ambient health monitoring, multi-tenant health monitoring at hospital scale, and a conversational agent. The comparison shows two large effects:

- **Memory footprint.** Monolithic deployment loads model artifacts per agent. Microservice deployment deduplicates shared backbone models across agents. For the multi-tenant healthcare workload at 15 patients, monolithic deployment requires roughly 136 GB versus about 14 GB for microservice deployment, exceeding the 128 GB ceiling that both classes provide, while the microservice footprint fits on both.

- **Cluster-scale capacity.** Microservice deployment removes the per-device memory cap and adds a statistical pooling gain as cluster size grows. Monolithic deployment, memory-capped per device and statically placed, does neither. One X100 under microservice runs a full ward with its per-device capacity set by operational work on the CPU, while a monolithic X100 is memory-capped at about 14 patients. Across a cluster the effect compounds: a 3-device fleet serves roughly 435 patients under microservice deployment against about 42 under monolithic, and a 10-device fleet serves roughly 1850 against 140.

We also identify a set of operational properties (per-agent updates, fault isolation, zero-trust security, service discovery, cluster elasticity, observability) that microservice deployment provides as architectural defaults and that monolithic deployment either does not provide or requires substantial per-application engineering to approximate. These properties matter across the verticals where agentic AI is being deployed: healthcare, automotive (cognitive layer only), industrial, defense, retail and hospitality, and consumer products. Some of them have vertical-specific stakes worth calling out.

We deliberately did not attempt to quantify single-device cross-resource placement (NPU work falling back to GPU or CPU at higher cost when one resource saturates). That analysis depends sensitively on cost multipliers, queueing model, and placement policy choices that cannot be settled without measurements we do not currently have. We discuss this briefly as a research direction and focus the paper on the two effects (memory and cluster) that can be quantified rigorously with the current data.

The rest of the paper proceeds as follows. Section 2 defines the two deployment models concretely. Section 3 presents the memory comparison. Section 4 presents the cluster scaling compar-

[1]The "x" in Agentix denotes a cross-platform design: the same microservice agents run across CPU-, GPU-, and NPU-centric devices, from compute-capable end devices such as smart cameras and robots up to servers.

ison. Section 5 covers the operational and lifecycle properties. Section 6 summarizes the argument for the agentic operating environment.

2 Monolithic vs Microservice Deployment

A monolithic agentic stack is one application binary that bundles multiple agents into a single process. All agent inference, coordination, state management, and routing happens within that process. The model artifacts (ONNX, GGUF, or similar) are the same model files that a microservice deployment would use; what differs is how they are loaded and how mimOE manages them.

A microservice agentic stack runs each agent as an independent service with its own lifecycle and its own state. mimOE provides resource discovery, service discovery, dynamic placement, message routing, and authentication between agents. We refer to this layer, described by prior work as the Agentix-Native substrate [2], as mimOE throughout.

Six concrete differences between the two models shape the analysis that follows:

1. **Per-agent lifecycle.** Microservice deployment manages each agent independently. Monolithic deployment shares one process lifecycle across all agents.

2. **Service discovery across devices.** Microservice deployment provides runtime discovery of agent endpoints. Monolithic deployment wires endpoints in code or configuration at build time.

3. **Dynamic resource placement within a device.** Microservice deployment can place operators on resources at runtime. Monolithic deployment decides operator placement at build time.

4. **Cross-device load balancing.** Microservice deployment allows agents to be relocated to peer devices with spare capacity. Monolithic deployment pins agents to specific devices at deploy time.

5. **Model artifact deduplication.** Microservice deployment loads each shared model once and references it from multiple agent instances. Monolithic deployment loads model artifacts per agent that uses them.

6. **Authentication.** Microservice deployment provides framework-level zero-trust authentication on inter-agent calls. Monolithic deployment relies on application code to implement security boundaries between agents, if any.

The model files on disk are identical in both deployments. The differences above are about what mimOE does with those files, not about the files themselves.

2.1 Where Containerization Fits

Containerization (Docker, Podman, container runtimes orchestrated by Kubernetes or similar) is an increasingly common deployment pattern on platforms like the X100 and the Jetson family. It is worth addressing explicitly because it is the obvious intermediate point between monolithic and microservice deployment, and a reader could reasonably ask whether containerized deployment closes the gaps reported in this paper.

Containerization addresses a subset of the differences listed above. Each agent in its own container has an independent process and memory space, so fault isolation is provided. Per-agent updates are possible because one container image can be replaced without redeploying the rest of the stack, though the model artifacts the container loads at runtime are still loaded per container. Per-container metrics, logs, and traces are produced by default by standard container tooling. Heterogeneous fleets become tractable because container images abstract most of the underlying OS differences for matching CPU architectures.

Containerization does not address the two effects that drive the headline findings of this paper. Container runtimes do not deduplicate model weights across containers at runtime; each container that needs a model loads its own copy into its own process memory. Container image layer sharing reduces disk footprint but not the resident memory footprint that matters for platform capacity. Kubernetes and similar cluster orchestrators schedule containers across devices but do not route individual agent calls across container instances based on real-time load. Service mesh layers (Istio, Linkerd) can do call-level routing but they do not provide agentic AI semantics such as per-agent state coordination, shared model registries, or zero-trust authentication tied to agent identity.

A containerized deployment is therefore a partial substitute for the microservice substrate on the operational dimension but not on the memory or cluster dimensions. The memory comparison in Section 3 and the cluster comparison in Section 4 apply to containerized monolithic deployment in the same way they apply to non-containerized monolithic deployment, because the per-agent model loading and the absence of call-level cluster routing are runtime properties above the container layer. Where containerization helps most is on the operational properties covered in Section 5, and we note specifically which properties containerization addresses there.

3 Memory Footprint Comparison

3.1 Memory Models

Under microservice deployment with model deduplication, total memory is:

$$M_{ms} = M_{runtime} + N \cdot M_{agent\_state} + \sum_i (1 - \delta) \cdot M_{model,i} + M_{extra}$$

where N is the agent count, M_{agent_state} is the per-agent runtime state, $M_{model,i}$ is the size of each shared model, δ is the deduplication factor (we use $\delta = 0.59$ from prior characterization of the mimik runtime [2]), and M_{extra} accounts for resident data such as patient history in the multi-tenant case.

Under monolithic deployment, each agent loads its own copy of the model tier it uses, and there is no shared model registry:

$$M_{mono} = M_{binary} + N \cdot M_{agent\_state} + \sum_i n_i \cdot M_{model,i} + M_{extra}$$

where n_i is the number of agents using each model tier and M_{binary} is the application binary footprint (we use 80 MB as a representative value for a multi-agent Python plus ML stack).

The model artifacts are identical in both cases; the difference is whether they are shared or replicated.

3.2 Use Case Results

Figure 1 compares memory footprint across five representative use cases, on a logarithmic vertical axis. The X100's 128 GB ceiling and the GPU-centric and NPU-centric classes' 64 GB ceiling are shown as horizontal lines.

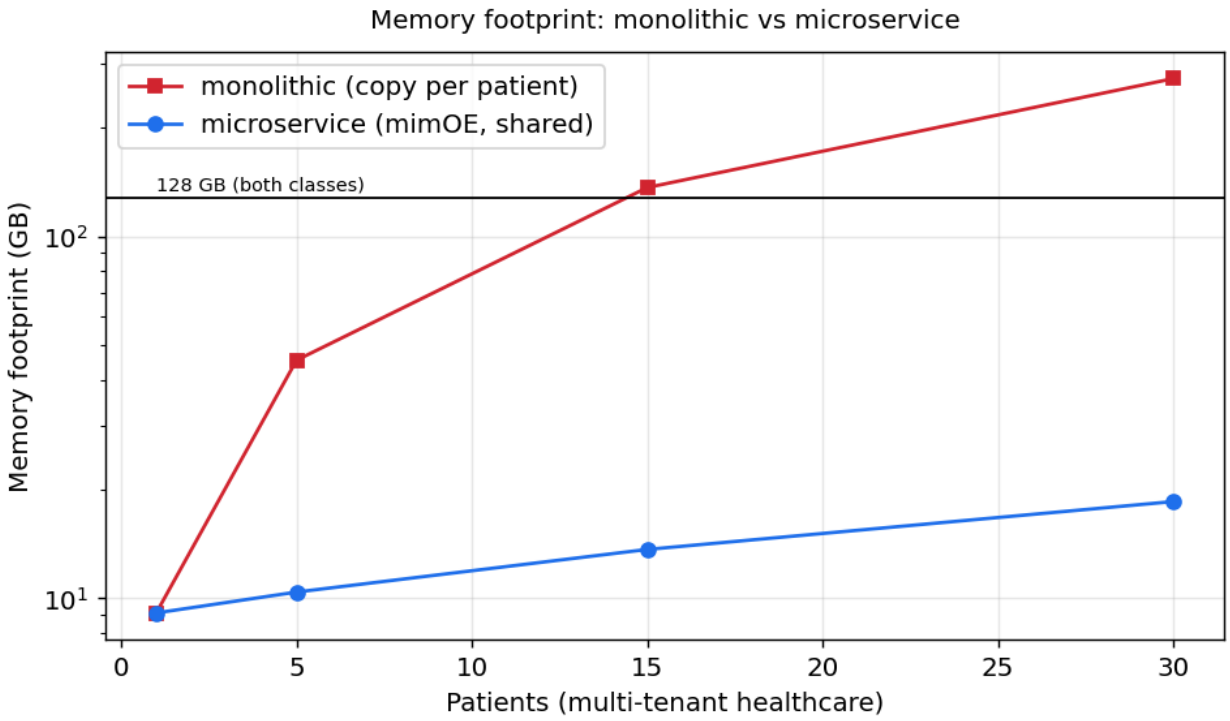

Figure 1: Memory footprint across use cases under microservice and monolithic deployment, with platform capacity ceilings shown. Log scale on the vertical axis.

The pattern is consistent: monolithic deployment uses more memory than microservice deployment across all use cases, with the gap widening as agent count and per-agent model complexity grow. The numerical values are:

Use case	Microservice (GB)	Monolithic (GB)	Ratio
Camera + Robot (5 agents)	2.1	3.0	1.4x
Industrial sensing (8 agents)	1.5	1.6	1.1x
Health monitoring (smartphone, 1 patient)	9.1	9.1	1.0x
Health monitoring (hospital, 15 patients)	13.6	136.4	10.0x
Conversational agent (5 agents)	15.4	17.2	1.1x

3.3 Multi-Tenant Deployment Is Where the Gap Matters

For a single patient the two deployment models coincide: each loads the model set once, so the footprint is about 9 GB either way and there is nothing to deduplicate across tenants. The gap opens with multi-tenancy. In this composition the biometrics are computed on the wearable and sensor devices, so the on-SoC stack per patient is six light domain summary reasoners plus two sLLM, a coordinator and a Llama 3.1 8B companion, which together dominate the footprint at roughly 9 GB of model weight. Monolithic deployment loads that full set again for every patient, while microservice deployment loads each model once and references it from every tenant, as described in Section 2. At 15 patients the monolithic footprint reaches about 136 GB and the microservice footprint about 14 GB, a 10x difference. The monolithic footprint exceeds the 128 GB ceiling that both classes provide, while the microservice footprint fits comfortably on both. The ratio grows

with tenancy, reaching roughly 15x at a full 30-patient ward, because the monolithic cost scales with the model set per patient while the microservice cost scales only with per-patient history and state.

This is where the architectural argument depends on the microservice substrate. Memory capacity alone does not resolve the multi-tenant workload: at 15 patients the monolithic footprint exceeds the 128 GB ceiling that both classes provide, so under monolithic deployment the workload does not fit on either class. Deduplication under the microservice substrate is what brings the workload back within reach, independently of which class runs it.

The deduplication effect grows with multi-tenancy. Each additional patient in the deployment adds another set of agents that share the same backbone models. Under microservice deployment the marginal memory cost per patient is the patient history and per-agent state. Under monolithic deployment it is the patient history plus a fresh copy of every model tier the agents use. The two cost curves diverge as multi-tenancy scales.

4 Cluster-Scale Capacity Comparison

4.1 Statistical Pooling Under Microservice Deployment

Statistical pooling theory gives a useful result for cluster-scale deployment: when agent loads on different devices are independent and mimOE supports dynamic placement, a cluster of n devices sustains higher per-device utilization than a single device alone at the same reliability target. Specifically, the sustainable per-device utilization with cluster pooling is:

$$\mu_n^{max} = 1 - z_p \cdot \frac{CV}{\sqrt{n}}$$

where CV is the coefficient of variation of the load on a single device, z_p is the normal quantile corresponding to the target saturation probability p, and $\sqrt{n}$ scaling reflects how variance averages out across the cluster.

The pooling result assumes that work can be dynamically routed to a device with spare capacity at the moment of demand. This requires the microservice substrate. Without dynamic discovery, routing, and security across devices, agents are statically placed at deploy time and the cluster's effective capacity collapses to n times the single-device sustainable utilization with no pooling benefit.

4.2 Cluster Capacity Across the Two Platform Classes

Figure 2 shows the effective patient capacity of clusters of 1, 2, 3, 5, and 10 devices under both deployment models. The single-device baseline differs by deployment. Under monolithic deployment the device is memory-capped at about 14 patients on either class, since both provide 128 GB, and it does not pool. Under microservice deployment memory is not the constraint; the per-device capacity is set by operational work on the CPU, on the order of 234 patients on the X100 at saturation, and the substrate pools burst capacity across devices. We use $CV = 0.4$ and a 5 percent saturation probability target as the modeling baseline, so a single microservice device sustains about 34 percent of its capacity in isolation, rising with cluster size.

The pattern is consistent: microservice deployment grows faster than monolithic because it removes the memory cap and pools burst capacity, while monolithic scales linearly at its memory-capped per-device baseline. For the X100, a 3-device fleet serves roughly 435 patients under microservice deployment against about 42 under monolithic, and a 10-device fleet serves roughly 1850

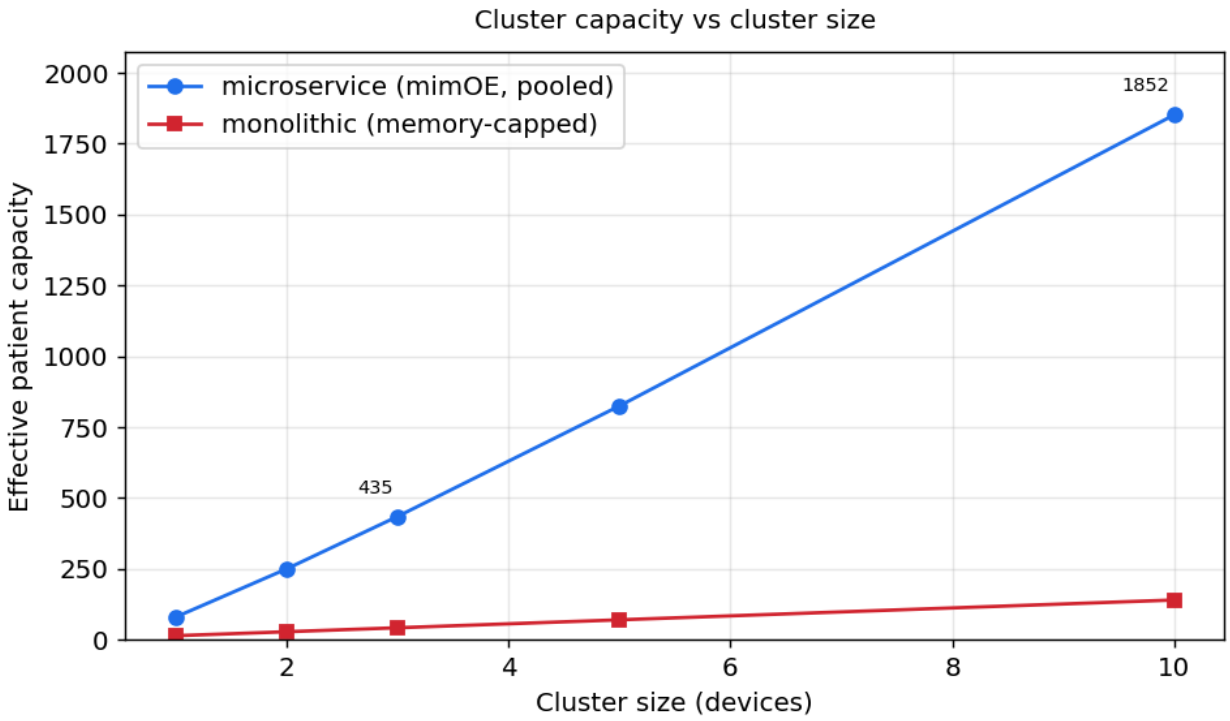

Figure 2: Effective cluster capacity vs cluster size under microservice and monolithic deployment, for each architectural class. Microservice deployment delivers a statistical pooling gain that grows with cluster size; monolithic deployment scales linearly with n.

against 140. For the GPU-centric class the relative gain from pooling is the same (driven by the pooling formula), but the absolute microservice scale is smaller because its lower CPU capacity sets a lower per-device ceiling, on the order of 104 patients against 234 on the X100; the monolithic baseline is similar on both classes, since both provide 128 GB.

4.3 Cluster Composition and Mixed Deployments

The pooling result holds when mimOE can place any agent on any device in the cluster. The CPU-centric class enables this directly because each device can absorb the full agentic workload mix; mimOE is not constrained by per-device specialization in its placement decisions. GPU-centric cluster members impose placement constraints: their smaller CPU binds first under operational load, so operationally intensive agents crowd onto them quickly, which limits the placement options when the cluster includes GPU-centric members and the workload is operationally intensive rather than dominated by a single large model.

A cluster of CPU-centric devices delivers the full pooling benefit across any workload mix the deployment encounters. A cluster of GPU-centric devices delivers the pooling benefit most fully within the single-large-model regime the class targets, and binds on CPU under operationally intensive mixes. A heterogeneous cluster (mixed classes) delivers something in between, with placement constraints that mimOE must respect. The CPU-centric class provides broader agentic workload coverage per SKU on a single device, and this compounds at cluster scale into broader cluster-level workload coverage.

4.4 Why Monolithic Deployment Does Not Pool

The cluster capacity gain under microservice deployment comes from three properties of mimOE working together: dynamic resource discovery (each device advertises its current load), dynamic service discovery (any agent instance can be located on any device), and continuum-transparent routing (callers do not need to know which device hosts the callee). When one device's load spikes, mimOE can route incoming calls to a peer device with spare capacity. Bursts on one device do not need to be served by that same device.

Monolithic deployment does not have these properties. Each device runs its own application binary, with agents statically placed at deploy time and inter-device communication implemented by application code rather than framework primitives. When one device's load spikes, there is no general mechanism to relocate or duplicate agents to a peer. The cluster's behavior is the sum of n independent devices, each provisioned for its own worst case, with no headroom sharing across the cluster.

This is not a property of the silicon; it is a property of mimOE. A CPU-centric SoC running a monolithic stack delivers the same cluster behavior as n independent CPU-centric devices. mimOE is what unlocks the pooling, and without it the cluster gain is zero regardless of silicon class.

5 Operational and Lifecycle Properties

Beyond the memory and cluster-scale capacity effects above, mimOE provides a set of operational properties as architectural defaults. Monolithic deployment can in principle approximate any of these properties with sufficient per-application engineering, but the engineering cost is paid per application rather than amortized across mimOE. The following subsections describe each property, what monolithic deployment can and cannot offer in its place, and where the property has vertical-specific stakes worth noting.

5.1 Per-Agent Updates and Update Payload Size

Microservice deployment updates one agent at a time. The update payload is the changed agent (typically a few MB or less, since the model artifact remains the same and is referenced from the shared registry). The rest of the stack continues running during the update.

Monolithic deployment updates the whole application binary on any agent change. The update payload is the full binary plus all bundled models that ship together (typically hundreds of MB to several GB). The whole stack restarts on each update, with a deployment window for staged rollout to avoid downtime.

In healthcare, automotive cognitive-layer deployments, industrial monitoring, and defense, update frequency directly affects the rate at which improvements reach the field. A monolithic stack with a 30-day verification cycle on each binary release imposes a different velocity of improvement than a microservice stack where individual agents can be updated in days. Consumer products are less affected by this difference because verification overhead is lower, but the bandwidth cost of full-binary updates on devices with metered connectivity still matters.

5.2 Regulatory Verification Scope

This is the property with the largest vertical-specific stakes. In regulated domains, every shipping unit requires verification, and the scope of verification is set by the shipping unit boundary.

In healthcare devices subject to FDA Premarket Notification (510(k)) or equivalent international regulations, a monolithic binary requires verification of the whole binary on each release. A microservice deployment can ship updates to individual agents and verify only the changed agent against its declared interface. The verification cycle for the whole binary in medical devices typically runs 6 to 18 months; the verification cycle for an individual agent module can be on the order of weeks for incremental changes.

In automotive cognitive-layer applications (driver state monitoring, in-vehicle assistant, long-term vehicle health, fleet coordination), the regulatory layer above the safety-critical stack is less stringent than ASIL D requirements but still substantial. UN R155 and R156 cybersecurity and software update regulations apply to vehicle software, and per-agent verification scope reduces the surface area that each update must defend. This applies only to the cognitive layer; the safety-critical real-time control stack operates under different regulatory and engineering constraints that are outside the scope of this analysis.

In defense, certification under DO-178C (avionics software) and the Risk Management Framework (RMF) for cyber programs uses similar logic: smaller verification units accelerate the certification cycle and reduce the cost of incremental changes.

In financial services, audit and compliance reviews track agent behavior. Per-agent observability and discrete update boundaries make the audit trail clearer than aggregate behavior of a single monolithic process.

In industrial deployments, ISO 27001 and IEC 62443 controls on operational technology software benefit from the same scope-reduction logic.

In retail, hospitality, and consumer products, regulatory burden is lower and the advantage of per-agent verification is smaller. The advantage still exists (faster iteration cycle, smaller update payload), but the cost difference between whole-binary and per-agent verification is less material.

5.3 Fault Isolation

Microservice deployment isolates each agent in its own process or container. A fault in one agent (memory corruption, segmentation fault, infinite loop, unhandled exception) is contained to that agent. The remaining agents continue running; the affected agent restarts independently.

Monolithic deployment runs all agents in one process. A fault in any agent can take down the whole process. Recovery requires restarting the whole stack, which during the restart window leaves no agents available.

Across verticals this matters wherever availability is a real requirement. In healthcare a fault in one agent should not take down the patient monitoring service. In industrial monitoring a fault in one agent should not blind the operator to all sensors. In retail a fault in one agent should not bring down the entire store-level assistant. The vertical-independent point is that the fault propagation radius differs by deployment model; the vertical-specific point is how much that matters for the deployment in question.

5.4 Zero-Configuration Service Discovery

Microservice deployment provides runtime discovery: agents advertise their capabilities, and any caller can locate them at runtime by capability or by name. Adding a new agent to a running stack requires no configuration changes elsewhere; the new agent registers and becomes available to callers.

Monolithic deployment wires endpoints in code or configuration files at build time. Adding a new agent requires editing wiring code or config files and redeploying the binary. Removing or

replacing an agent has similar overhead.

The practical consequence is that microservice deployment supports incremental evolution of the agent set without coordinated redeployment. This matters in any vertical where the agent set is expected to grow over time (consumer assistants gaining new capabilities, industrial sensors adding new analytics, healthcare adding new clinical agents). It is less important in deployments where the agent set is frozen at deployment time.

5.5 Zero-Trust Security

Microservice deployment provides framework-level authentication on inter-agent calls. Each call carries a verifiable identity, and the called agent can decide what to allow based on that identity. A compromised agent cannot impersonate another agent or access another agent's data without explicit authorization.

Monolithic deployment runs all agents in the same process memory space with no security boundary between them. A compromised agent can read or modify the state of any other agent in the same process, since they share an address space. Application code can implement boundaries, but they are not architectural defaults.

The stakes vary by vertical. In healthcare multi-tenant deployments (multiple patients on one device), per-agent authentication is the architectural property that lets one patient's data remain inaccessible to agents operating on behalf of another patient. In defense, where the threat model is explicitly adversarial, framework-level zero trust is closer to a binding requirement than a discretionary feature. In financial services, audit trails of per-agent activity benefit from authenticated identity on each call. In automotive cognitive-layer deployments, vehicle-to-vehicle and vehicle-to-cloud interactions benefit from authenticated agent identity. In retail, hospitality, and consumer single-user single-device deployments, the threat model is less adversarial and the advantage is more about defense-in-depth than a hard requirement.

5.6 Heterogeneous Device Fleets and the Same Runtime Everywhere

Microservice deployment uses the same agent images across devices with different underlying silicon, as long as mimOE supports each silicon class. Fleet management scales as mimOE supports more silicon, not as the number of platform-specific binaries grows.

Monolithic deployment targets a specific hardware configuration per binary. Different devices in a fleet require different binaries, each maintained separately. Fleet management complexity grows with the number of distinct hardware classes the deployment spans.

This matters in verticals where the fleet is intrinsically heterogeneous. Healthcare deployments typically mix ward-level workstation-class devices with bedside tablet-class devices and clinician-carried phones. Automotive cognitive-layer deployments mix in-vehicle SoCs across vehicle model years and trim levels. Industrial deployments mix factory-floor devices with management-tier servers. Retail and hospitality deployments mix in-store and back-office hardware. The horizontal pattern across verticals is the same: real fleets are heterogeneous and the agentic operating environment that absorbs that heterogeneity is the one that scales.

5.7 Cluster Discovery and Elasticity

Microservice deployment with cluster discovery detects new devices joining the cluster automatically and starts routing work to them. Devices leaving the cluster (planned maintenance or unplanned failure) have their work drained to peers without manual intervention.

Monolithic deployment scales by deploying additional binary instances and updating routing manually. Elastic capacity scaling is an engineering project, not a routine operation.

The vertical with the largest stakes here is any deployment that experiences variable demand and benefits from elastic capacity. Hospital wards have peak periods and quiet periods. Factory floors have shift changes. Retail has open-hours and closed-hours patterns. Defense missions have surge requirements. Single-device or fixed-fleet consumer deployments exercise this less.

5.8 Operational Observability

Microservice deployment produces per-agent metrics, logs, and distributed traces by default because each agent is a separate observable unit. Diagnosing production issues becomes substantially faster because the system structure is visible to monitoring tools.

Monolithic deployment produces aggregate metrics and logs at the process level. Tracing one agent's behavior within the process requires instrumenting the application code. Diagnosis time grows with the complexity of the bundled stack.

This property is uniformly important across verticals because every production deployment eventually requires diagnosis of unexpected behavior. The difference is one of mean time to diagnose rather than a categorical capability difference; monolithic deployment can produce per-agent observability through application engineering, but the engineering is paid per deployment.

6 Conclusion

This paper compared monolithic and microservice deployment of agentic AI workloads on heterogeneous SoCs and found that mimOE is what makes production-scale deployment work. Three findings support this conclusion.

Memory deduplication under microservice deployment is necessary to fit multi-tenant agentic workloads within the memory ceilings of even the largest current silicon. The 15-patient health monitoring workload requires about 14 GB under microservice deployment and about 136 GB under monolithic deployment. Only the microservice case fits on the X100; the monolithic case exceeds even its 128 GB ceiling.

Cluster-scale capacity under microservice deployment far exceeds equivalent monolithic clusters, because microservice deployment removes the per-device memory cap and pools burst capacity while monolithic deployment does neither. A 3-device X100 fleet serves roughly 435 patients under microservice and about 42 under monolithic. A 10-device fleet serves roughly 1850 versus 140. The gain comes from the runtime's ability to load models once and to dynamically route work across devices in the cluster, which monolithic deployment cannot do.

Operational and lifecycle properties of microservice deployment (per-agent updates, regulatory verification scope, fault isolation, zero-configuration service discovery, zero-trust security, heterogeneous fleet management, cluster elasticity, per-agent observability) are architectural defaults rather than per-application engineering. The properties apply across the verticals where agentic AI is being deployed (healthcare, automotive cognitive layer, industrial, defense, retail and hospitality, consumer), with some properties carrying larger stakes in regulated and multi-tenant verticals than in consumer single-device deployments.

mimOE is therefore not commodity infrastructure. For OEMs and integrators evaluating platforms for production agentic AI, the choice of deployment model determines whether the silicon's capabilities translate to deliverable capacity, whether the deployment scales beyond a single device, and whether the operational properties needed for regulated and multi-tenant environments are

available as defaults or have to be built per application. mimOE is the multiplier on the silicon investment in production agentic AI.

References

[1] S. Alamouti, F. Arjomandi, M. Burger, J. Hsu. *Architectural Fit for Production-Scale Agentic AI on Heterogeneous SoCs*. mimik Technology Inc., 2026.

[2] S. Alamouti, F. Arjomandi, M. Burger, J. Hsu. *Agentix-Native: A Distributed Microservice Substrate for Agentic AI*. mimik Technology Inc., 2025.